9-2020

Artificial Intelligence in Pursuit-evasion Games, Specifically in the Scotland Yard Game

Arif M. Alamri

ARTIFICIAL INTELLIGENCE IN PURSUIT-EVASION GAMES, SPECIFICALLY IN THE SCOTLAND YARD GAME

THESIS

Arif M. Alamri, Captain, RSAF
AFIT-ENG-MS-20-S-003

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-S-003

ARTIFICIAL INTELLIGENCE IN PURSUIT-EVASION GAMES, SPECIFICALLY IN THE SCOTLAND YARD GAME

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Arif M. Alamri,

Captain, RSAF

September 2020

AFIT-ENG-MS-20-S-003

ARTIFICIAL INTELLIGENCE IN PURSUIT-EVASION GAMES, SPECIFICALLY IN THE SCOTLAND YARD GAME

Arif M. Alamri, BS

Captain, RSAF

Committee Membership:

Dr. Kenneth M. Hopkinson, Ph.D.
Chair

Major Joan A. Betances, Ph.D.
Member

Dr. Douglas D. Hodson, Ph. D.
Member

AFIT-ENG-MS-20-S-003

Abstract

This research provides a heuristic algorithm for the detectives, who try to collectively capture a criminal known as Mr. X, in the Scotland Yard pursuer-evasion game. In Scotland Yard, a team of detectives attempts to converge on and capture a criminal known as Mr. X. The heuristic algorithm developed in this thesis is designed to emulate human strategies when playing the game rather than using sampling algorithms, as is common in algorithms such as Monte Carlo Tree Search. The algorithm uses the current state of the board at each time step, including the current positions of the detectives as well as the last known position of Mr. X. The heuristic algorithm then analyses all of the possible options. The heuristic algorithm then uses a process of elimination to determine the best possible detective moves by running an appropriately constructed minimum cost flow maximum flow instance. The heuristic algorithm was tested in a series of experiments, in which the algorithm achieved a 57% win rate. This win rate was achieved using a random starting position for each of the pursuer detectives as well as for the evader, Mr. X. When Mr. X started at an easily accessible location, namely position 146, the pursuing detectives were able to capture him 62% of the time. These results show promise for this heuristic in pursuer-evader games like Scotland Yard.

Acknowledgments

In the Name of Allah, the Most Compassionate, the Most Merciful

"And You were Only Given Little Knowledge" The Holy Quran, Chapter 17, Verse 85

Thank God for all the blessings my family and I are blessed with, thank God for the blessing of being able to attend AFIT school and complete my master's degree.

I would like to thank my parents for all the prayers and the support they have given and my wife for the support and patience. I would also like to thank my advisor, Dr. Kenneth Hopkinson, for all the support, motivation, and guidance and Major Betances, who has been of great help with the research and the courses I took with him. Without the effort and the help of you all, this would have never been accomplished.

I would like to dedicate this work to my son.

Arif M. Alamri

Table of Contents

Page

List of Figures

Page

List of Tables

Page

ARTIFICIAL INTELLIGENCE IN PURSUIT-EVASION GAMES, SPECIFICALLY IN THE SCOTLAND YARD GAME

I. Introduction

Ever since computers have been created, the idea of creating an intelligent computer has been in the minds of many people. Artificial computers are meant to emulate human behavior in learning and thought processes. But what do we mean when we say an intelligent computer? According to Pei Wang, "The essence of intelligence is the principle of adapting to the environment while working with insufficient knowledge and resources. Accordingly, an intelligent system should rely on finite processing capacity, work in real-time, open to unexpected tasks, and learn from experience. This working definition interprets "intelligence" as a form of "relative rationality." This definition was the most agreed-upon definition, with 58% (between strongly agree and agree) among respondents (N = 567) [1].

Artificial intelligence can be categorized into six main fields [2]:

1. **Robotics**: using computers to control mechanical machines that have precise movements.
2. **Expert systems**: these systems emulate the rationality of a human expert.
3. **Natural language processing**: it is understanding, producing, and writing human languages.
4. **Speech recognition**: it deals with understanding and interpreting human speeches.
5. **Speech synthesis**: it includes programming the computer to be able to generate and mimic an audible human voice.

6. **Neural networks**: it focuses on enabling computers to deal with imperfect situations. This includes playing games.

Artificial intelligence in pursuit-evasion games, which fall under the neural networks category in the six subfields above, allow the machine to play the game either as an evader or a pursuer while relying on finite processing capacity, to work in real-time, and to learn from experience. There are two types of pursuit-evasion games: games with perfect information and games with imperfect information.

Games with complete information are those where all the information about the pursuers and evaders is known to both of them. Examples of such games are Chess and checkers. Games with imperfect information include those where some of the information about the pursuers or evaders is hidden—for example, Scotland yard.

1.1 Game Theory

Roger Myerson, a Nobel prize winner, defines game theory in his book in [3] as "the study of mathematical models of conflict and cooperation between intelligent, rational decision-makers." Game theory applications are found in many fields: economics, military, computer science, board games with multiagent, and many others. Game theory takes a game and represents it as a set of possible decisions, and then determines the best outcome of the game.

1.2 Problem Statement

The research outlined in this thesis looks to develop an artificial intelligence agent that can play as the detectives in the game Scotland Yard while maintaining resource, time, and complexity constraints. This thesis aims to provide a heuristic algorithm that

allows for cooperation among the detectives as they attempt to capture the evader (Mister X). The results of this research can be extended to real-life pursuit problems in military and police applications.

1.3 Motivation

Pursuit-evasion games have many real-life applications, especially military applications. With many military unmanned pieces of equipment entering service, such algorithms can be used to guide and help execute those military operations more efficiently and intelligently. Military missions pursuing enemies with unmanned aircraft and using drones for search and destroy are examples of such applications. With the thesis aiming to develop an artificially intelligent agent that can pursuit an evader, the thesis algorithm can be used in real-life military operations.

1.4 Assumptions

This thesis is developed based on the Scotland Yard game version 2.4 programmed by Johannes Jowereit and Shashi Mittal. This version has Mister X programmed using a computer-based heuristic algorithm while the detectives are played by human users. This thesis develops an artificial intelligence agent to play the part of the detectives against the already programmed Mister X.

1.5 Thesis Overview

This thesis document is arranged in five chapters. Chapter 2 describes the game Scotland Yard, relevant similar games with imperfect information, and related research. Chapter 3 presents the approach used to develop the artificial intelligence agent algorithm. Chapter 4 presents an analysis of experimental results and a statistical

comparison with related research. Chapter 5 summarizes the research and discusses possibilities for further study.

II. Background and Related Research

2.1 Chapter Overview

The purpose of this chapter is to provide a description of the Scotland Yard game followed by A discussion of related research and algorithms. Section 2.2.1 provides a brief history of the game. Section 2.2.2 discusses the rules of the game. Section 2.3 presents related research. Finally, section 2.4 summarizes the chapter.

2.2 Scotland Yard

2.2.1 Background

The game Scotland Yard was developed by Manfred Burggraf, Dorothy Garrels, Wolf H¨ormann, Fritz Ifland, Werner Scheerer, and Werner Schlegel in 1983. Cryo Interactive Entertainment introduced a Scotland Yard program in 1998. This program, however, only emulates the game for human players without any computerized opponents [4]. In 2008, DTP Young Entertainment released another program for the game. This new version has a computerized player, and was one of the strongest at that time [4].

2.2.2 Rules

Scotland Yard is played by six players: 5 detectives and one hider called Mister X. However, the game originally is a 2-player game: one controls the detectives, and one controls Mister X. The game is a graph-based game with 200 stations. Each station has several links attached to it. Each link has a specific type of transportation method. There are four transportations methods: Taxi (yellow), Bus (green), Underground (red), and Ferry (black), as shown in Figure 1. Each detective has 22 tickets: 10 Taxi, 8 Bus, 4 Underground. Mister X has 4 Taxi, 3 Bus, 3 Underground. Additionally, Mister X has

five black tickets and double moves tickets. Black tickets allow Mister X to use any method of transportation, including Ferry, without announcing the transportation method. Double tickets allow Mister X to play an extra move on his turn (back to back). The game starts after distributing all the tickets with Mister X positioned randomly on one of 13 possible stations, and the detectives are placed randomly on the map. The only rule on placement is no two detectives can be at the same station or on the station of Mister X. The game starts with Mister X's move followed by each detective moves in turn. Neither the detectives nor Mister X can skip a turn without moving. In every turn, Mister X announces only the type of ticket he used except on turns 3, 8, 13, 18, 23; on those turns, Mister X announces the type of ticket and his station number. Each ticket moves a detective or Mister X one station with the condition that no other detective is on that station already, and the detective has the type of ticket needed. The goal of the detectives is to cooperate to catch Mister X by occupying the station he is occupying. The target of Mister X is to escape from the detectives for 23 turns. The game ends in two scenarios: a detective moves to a station that Mister X is occupying, or we reach turn number 23 without catching Mister X. In the former case, the detectives win while Mister X wins in the latter case.

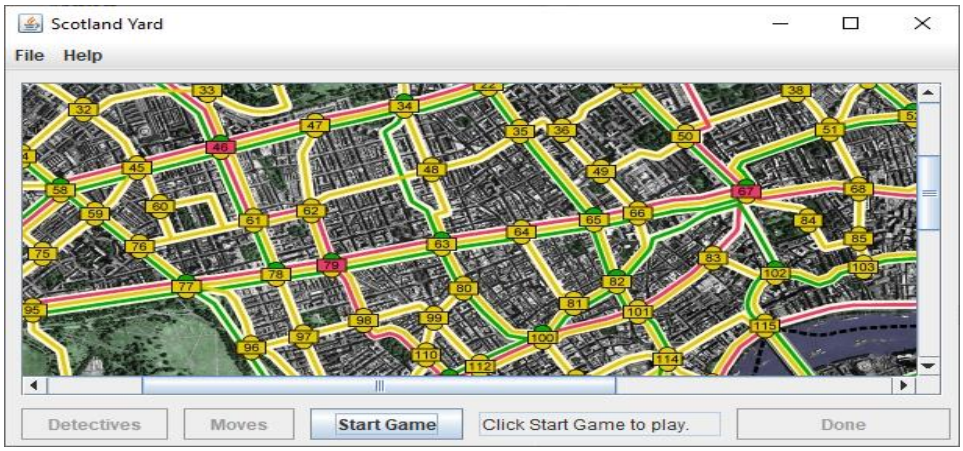

Figure 1. Scotland Yard Map

2.3 Relevant Researches

Scotland Yard is a pursuit-evasion game with imperfect information. Even though Scotland Yard itself is not involved in much academic research, there are a good number of research studies performed on similar games with imperfect information.

Ginsberg, Mathew L. in his paper "GIB: Steps toward an expert-level bridge-playing program" [5] introduces a human-like algorithm that can play the game Bridge as close to a human as possible. Generally, the approach of Ginsburg is a brute force approach with other algorithms used to handle the hidden information. He used two algorithms to play the game. The first one is the Monte Carlo card selection algorithm for card playing, and the second one uses Borel simulations for card biddings. The results of GIB were excellent; it won a competition and played against more than 15,000 players online with a loss rate of 0.2 IMPs/deal [5]. Our approach is somewhat close to the Ginsburg brute force approach while using our own method for handling the hidden information.

In the paper, "Design and Development of Bridge AI bid Program based on Double-hand Solver," [6] by Ruiyong Zhang, Hongkun Qiu, Yajie Wang, Yisheng Xiao, and Jize

Wang, a new design for an AI agent is introduced for bidding in Bridge games. The AI agent uses a brute force method to classify all possible situations using the card type and the card points. The bidding machine handles the imperfect information by first ordering the hand. After that, categorizes the hand using the bidding machine to determine its action. At the same time, it compares the ideal action that the system generates with the opponent's last call/action to check for the legality of the call [6]. Figure 2 below shows a step by step process of the bidding machine AI agent behavior.

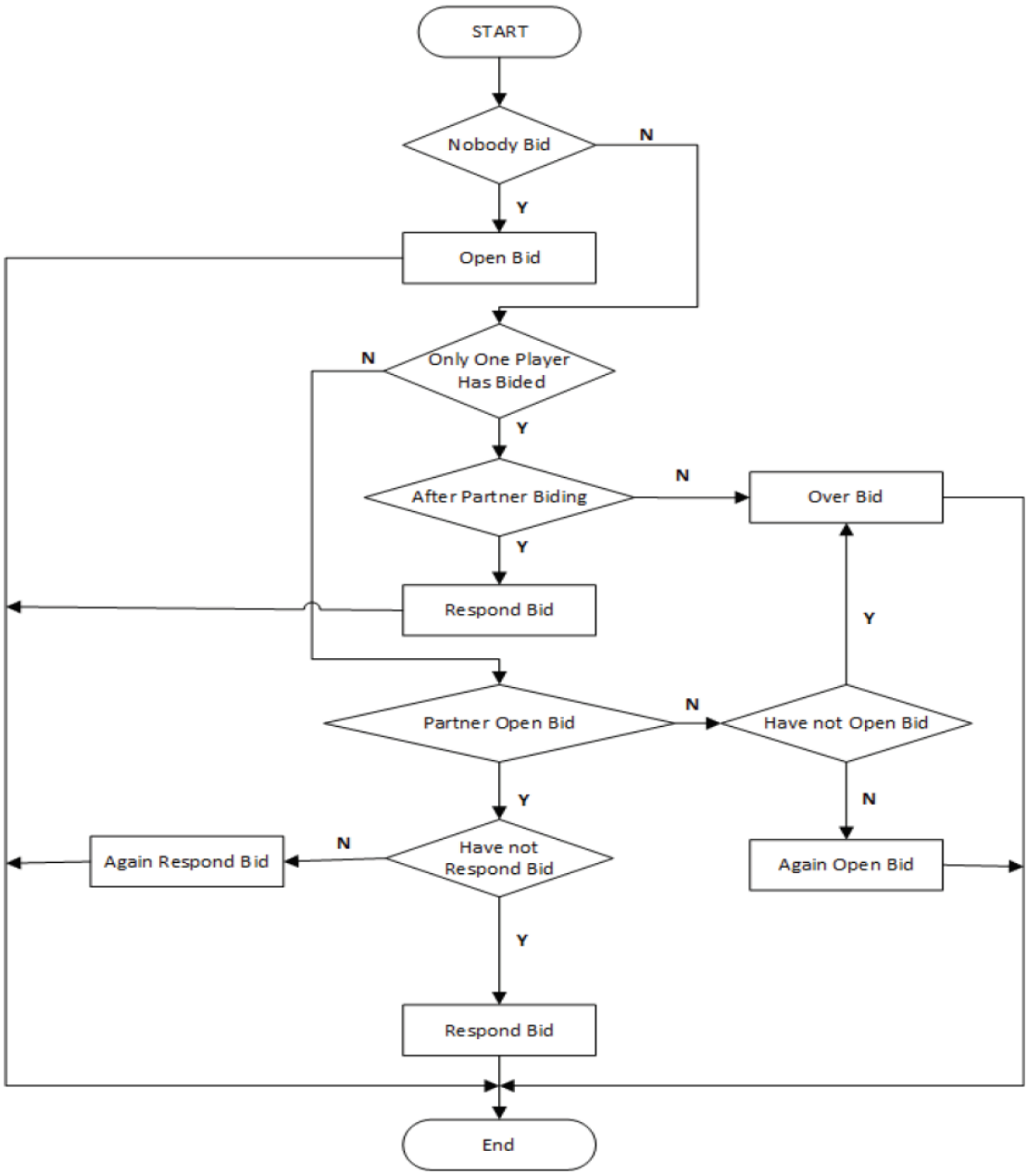

Figure 2. Function control flow chart [6]

Our algorithm uses the same idea by generating all the possibilities and making decisions accordingly.

Ian Frank and David Basin have examined search algorithms in a game with incomplete information [7]. According to the researchers, their approach formalizes the

general problem of games with incomplete information to allow them to identify and overcome two main problems that face any sampling algorithm used to solve such games. The first problem, which is called strategy fusion, affects algorithms that try to apply combined strategies on different levels of subsamples of the same sample of the game. This is a problem because of the main property of the games with incomplete information, namely that the information is not complete, and a solver cannot act the same at any given point of time. The second problem, called non-locality, occurs when one side of the game has more knowledge of the game than the other side. The side with more knowledge will use the information to his advantage by selecting different moves that might result in eliminating some of the search tree nodes. So, when the other side is planning his moves and examining the best search tree nodes, he does not know that the game might never reach this node. That is where the other side makes a mistake, and the non-locality problem of the algorithm arises. The researchers evaluate the sampling algorithms intending to clarify the issues of solving the games with incomplete information. The researcher also provided an algorithm for finding equilibrium points in the algorithms used to solve the games with incomplete information [7].

The paper “Design and implementation of military chess game algorithm based on probability model and situation evaluation” [8] introduced an algorithm using a probability model to play the game. The most important part is the evaluation method used by the researchers to evaluate each move before it is taken. The evaluation model consists of two parts: the situation evaluation and experience evaluation. The system has divided the game into two situations: offensive and defensive. The early period is generally defined as the defensive situation, that is when we need to fully obtain the

information of the enemy pieces, improve the probability distribution model of the system by collecting more data, and then, convert the defensive situation into an offensive situation, and call the offensive function to attack the enemy and achieve the objective. In the offensive mode, the probability distribution matrix is called in reverse because we are considering the current chess piece, the direct threat from the enemy, and the indirect threat from the enemy's next move. The researcher gives an example:

"1. Suppose that our chess piece is the colonel e, find the position line I of the attack target by cyclic traversal in the probability distribution matrix, and find the column position z.

2. The sum of the probability of range from 0 to z in the column is Sum1, the sum of the probability of the column range from z to 8 is Sum2, and the result of Sum2 minus Sum1 is the value of Attack_Direct_Threat. Discard positions 9 and 10 because the 9 and 10 positions correspond to mines and bombs in turn, so it is necessary to use the empirical function to do a particular treatment.

3. Create a simulated chessboard cMap(25)(12), copy the position of the chess piece in cMap(25)(12) to cmap(25)(12), and at the same time, perform position change on our side, assuming that it has reached the position and traverse all the next positions of the enemy. Repeat the above two steps to get the score of Attack_InDirect_Threat". The system showed excellent results against an older system implemented in 2018, winning to losing rate was 8:3 [8].

In the paper "Risk Dominance Strategy in Imperfect Information Multi-Player Game" [9] written by Xuan Wang, Jiajia Zhang, Xinxin Xu, and Zhaoyang Xu a decision-making module was introduced for the game of Military Chess based on the principles of Risk

dominance and Payoff dominance which are an extension of the Nash Equilibrium (NE) solution of the game theory, more on the Nash Equilibrium solution can be found in [10]. In the definition of the NE solution, the authors of the book stated that if the Risk dominance and Payoff dominance come to a contradiction, the Payoff dominance must be the one that is selected neglecting all the effect of Risk dominance. However, in the case of incomplete information games where the level of trust is low, because of the uncertainty of the game condition, we must account for the Risk Dominance. The module proposed is using the Payoff dominance method for schussing the best possible move among many candidates. However, in some cases, as stated above, Risk dominance must be considered. Even though moves that might lead to terrible results have a low probability of appearing, they are still there and could be picked. The Risk dominance part of the module proposed takes care of eliminating such moves from the candidates' moves. The module uses a logarithmic equation to compare two moves, and based on the result, conclude that one of them is Risk dominance. The new module was tested against an older version of the same system without the Risk dominance; results are in table 1. Also, tests were performed against humans on the internet; results are shown in table 2 [9].

Table 1. Results of the Risk dominance against non-Risk dominance [9]

Edition	Win	Draw	Lose	win rate
SiGuoJunQi 3.0	73	6	21	73%
SiGuoJunQi 2.0	21	6	73	21%

Table 2. Results of Risk and non-Risk dominance against internet players [9]

Edition	Win	Draw	Lose	win rate
SiGuoJunQi 3.0	9	2	29	22.5%
SiGuoJunQi 2.0	2	3	35	5%

Darse Billings, Lourdes Peña, Jonathan Schaeffer, and Duane Szafron developed in their research "Using Probabilistic Knowledge and Simulation to Play Poker" [11] an approach to calculate poker game decisions based on probabilities calculating. The paper used probability to fill in the missing information (imperfect game) based on all the gathered information. It runs statistical models for many times then make decisions accordingly. This method, according to the study, showed significant enhancements to the artificial intelligence agent playing the game [11].

In the game Morpion Solitaire, Tristan Cazenave proposed a new algorithm for playing the game in his paper "Monte Carlo Beam Search" [12]. The algorithm is an improvement of the Nested Monte Carlo Tree Search (NMC), which by itself is an enhancement of Monte Carlo Tree Search. The algorithm improves the NMC method for saving the best play at every simulation run by saving two moves instead of one. The most important aspect of the proposed algorithm is that it is faster than NMC. Moreover, Monte Carlo Beam search achieved better results; in fact, it set the world record - at that time - of 82 moves. Figure 3 gives an illustration of the algorithm operation.

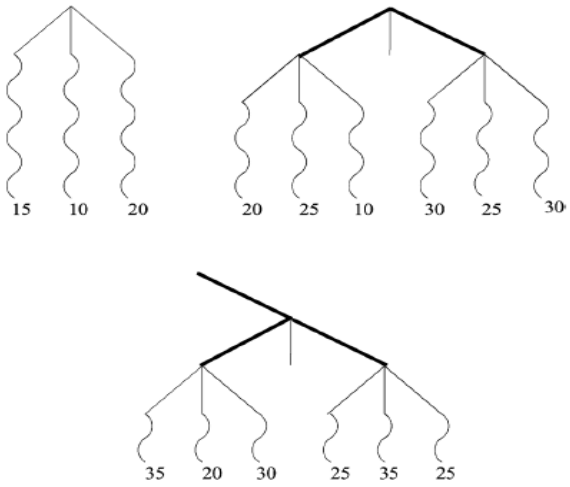

Figure 3. Illustration of beam search with a beam of size two [12]

In [13], three different versions of MCTS were introduced to play the game Kriegspiel. The three approaches can be summarized in Figure 4 below.

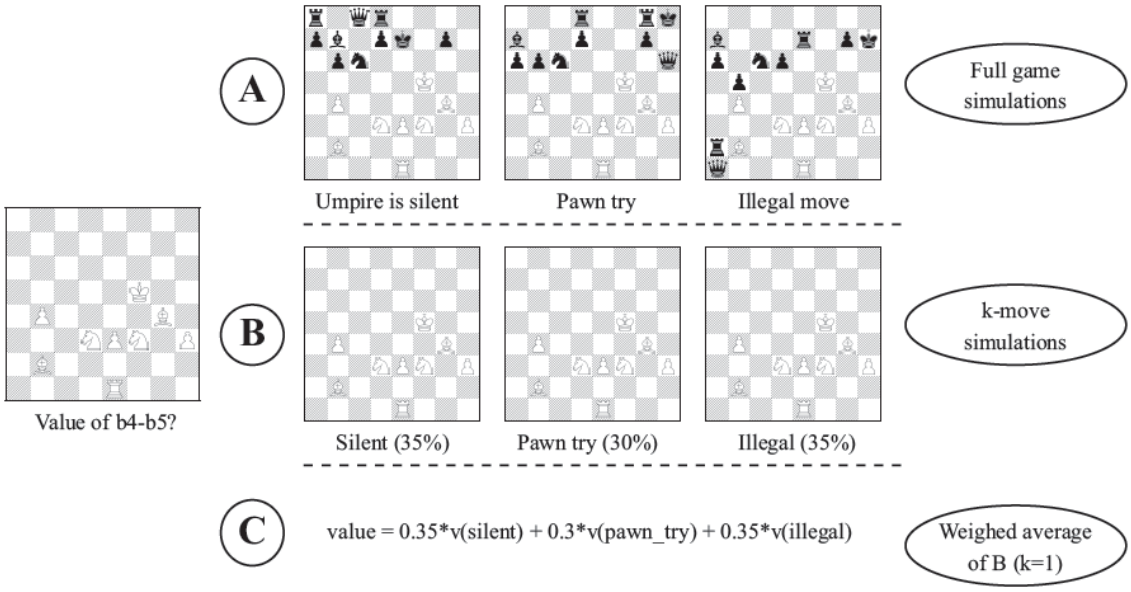

Figure 4. Comparison of three simulation methods [13]

The paper showed that the MCTS could be a good choice for a game with so much hidden information. However, proper care must be taken in the simulation phase to avoid sampling of too much information that is not beneficial. That is why the paper is

abstracting the game with a model where single states themselves do not matter but rather their perception.

Yngvi Björnsson and Hilmar Finnsson introduced a new simulation-based general game player in their paper "CADIAPLAYER: A Simulation-Based General Game Player" [14]. In the beginning, the authors defined the general game playing (GGP) project developed by the logic group at Stanford University. This project provided two main components: Game Description Language (GDL) and GGP Communication Protocol. The former is the full description of the game, including the status, the number of players, legal/illegal moves, and all other data of the game. The later is a game master (GM), which is a server for controlling the matches among the competing agents. This server uses the HTTP protocol for that purpose. For any agent to compete in the (GGP) competition, it must have an HTTP server to communicate with the GM, the ability to understand and communicate using GDL, and AI for strategic thinking. For the module proposed by the researchers, the HTTP server is an external process, and the other two requirements are combined into one playing agent, as shown in Figure 5.

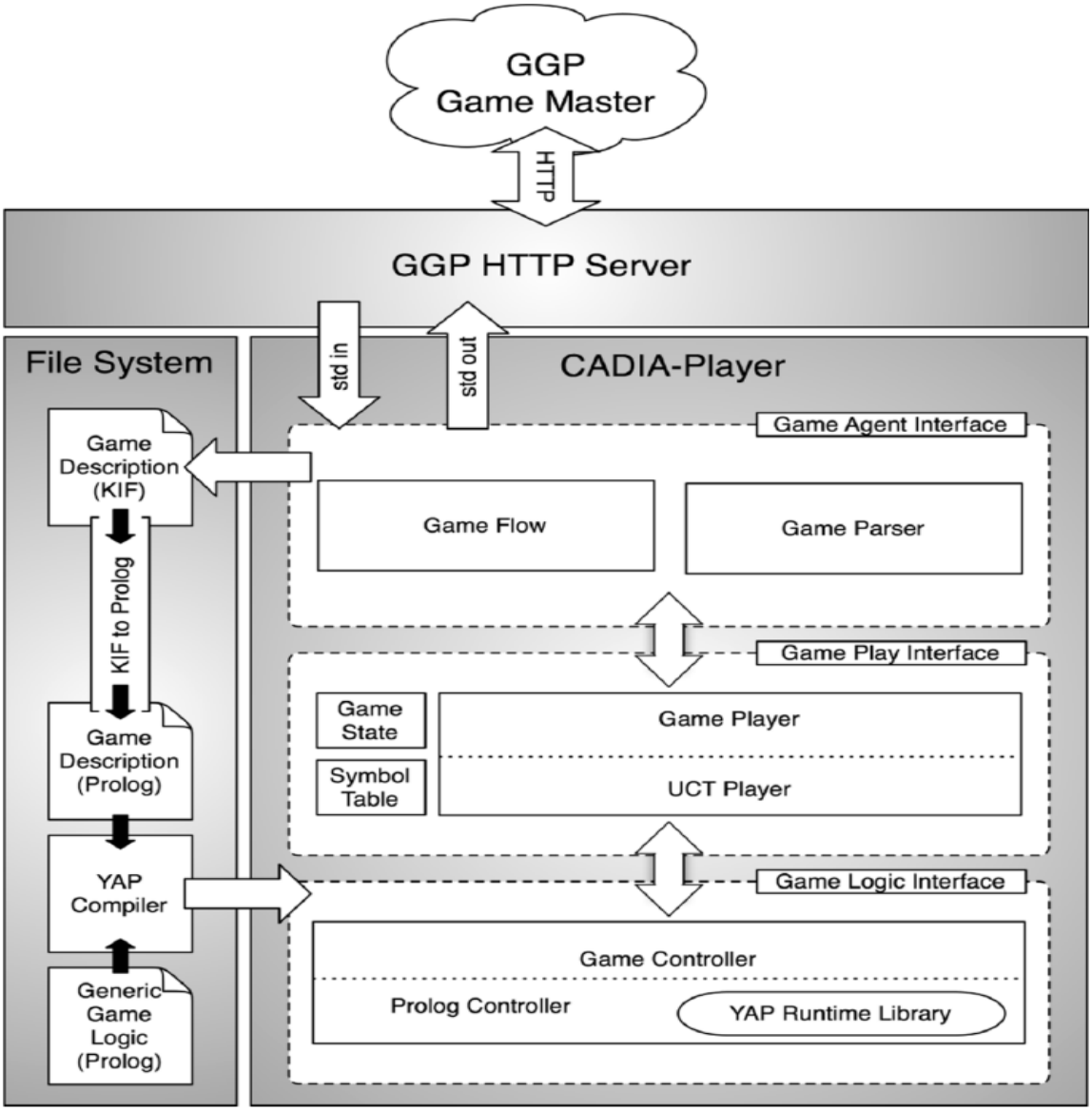

Figure 5. Overview of the CADIA Player Architecture [14]

The game-playing agents have three components: the game-agent interface, the game-play interface, and the game-logic interface. The game-agent interface controls the game flow through interaction and executing the command from the GM. It also converts the moves sent from the GM to the system into the internal format. The game-player interface is the core of the AI of the agent, which thinks and produces the moves of the agent. The main methods used for multiplayer games were mainly simulation-based

search algorithms. The game logic interface holds the state of the game that is the remaining moves, the previous moves, and the changes when one specific move is selected. The module uses Monte Carlo simulations for its decision-making process. The new module showed outstanding results compared to min-max and UCT methods, more details in [14].

Rui Xiongxing and He Yinglai introduced a new algorithm for playing games with imperfect information [15]. The algorithm provided by the researchers is called the UCT-RAVE algorithm. The algorithm is a combination of UCT (Upper Confidence bound applied to Tree) search and RAVE (Rapid Action Value Estimation) method. Along with the algorithm, the Monte Carlo Search method is used as an algorithm for the sampling of the game. UCT is an algorithm that uses UCB (Upper Confidence Bound) formula. It arises from looking for a solution to the problem of the K-armed bandit problem, which is a slot machine that has K handles. A player is allowed to pull a one handle to win a certain amount of money, which is the reward associated with this handle. The problem is what handle should the player pull knowing the reward to each handle. The usual way of pulling handles is pulling the handle according to the accumulated knowledge the player has on the reward associated with each handle, which is called exploitation. However, he might be missing a higher reward with one of the handles that are not exploited. As a result, he might try to choose more handles from the ones that he did not exploit yet; this is called exploration. UCB tries to solve the problem of contradiction between exploitation and exploration; more on UCB can be found in [16]. Rave, on the other hand, is an application of reinforcement learning of the value-based function from UCT. It gathers and assesses the data generated from UCT and gives information to the next

searches done by UCT to increase accuracy. Next, the combination of UCT-RAVE with Monte Carlo search is done to transform a game with incomplete information into a one with complete information. This is done by filling in the missing information in the Monte Carlo Search with the information obtained from the UCT-RAVE algorithm. The algorithm was tested on a multiplayer game, and the winning rate was 95%, which shows a good intelligence of the algorithm [15].

Nathan Sturtevant, in his research "Last-branch and speculative pruning algorithms for max" [17], provided a new technic for pruning algorithms. Since the standard algorithm for n-players games is max algorithm provided by [18], only some of the trees generated by the max algorithm can be pruned using existing pruning methods such as shallow pruning [19] and alpha-beta branch-and-bound pruning because these algorithms depend on the node ordering of the tree and the range of terminal values of the game [20]. The proposed new pruning method, last-branch and speculative pruning, only depends on the node ordering. Last pruning is like directional pruning as it examines nodes left to right without returning to the previously searched nodes. Speculative pruning, on the other hand, is not directional, and it might research an already traversed node. The results of the new algorithm were good in multiple games. For example, in the Chinses Checkers game, the expansion at depth six was reduced from 1.2 million for plain max to 100K for speculative max.

In the paper "Monte Carlo Tree Search for the Game of Scotland Yard" [4], the researchers J. (Pim) A.M. Nijssen and Mark H.M. Winands applied the MCTS algorithm to the game of Scotland Yard. Figure 5 below shows a brief description of MCTS. They provided an enhancement to the MCTS by introducing a new concept called location

categorization, which biases the locations that the hider (Mister X) can be at. The paper showed significant improvements in the winning rate of the detectives over the hider. Finally, the paper compared their algorithm results with the Nintendo DS results and showed that the proposed MCTS perform better.

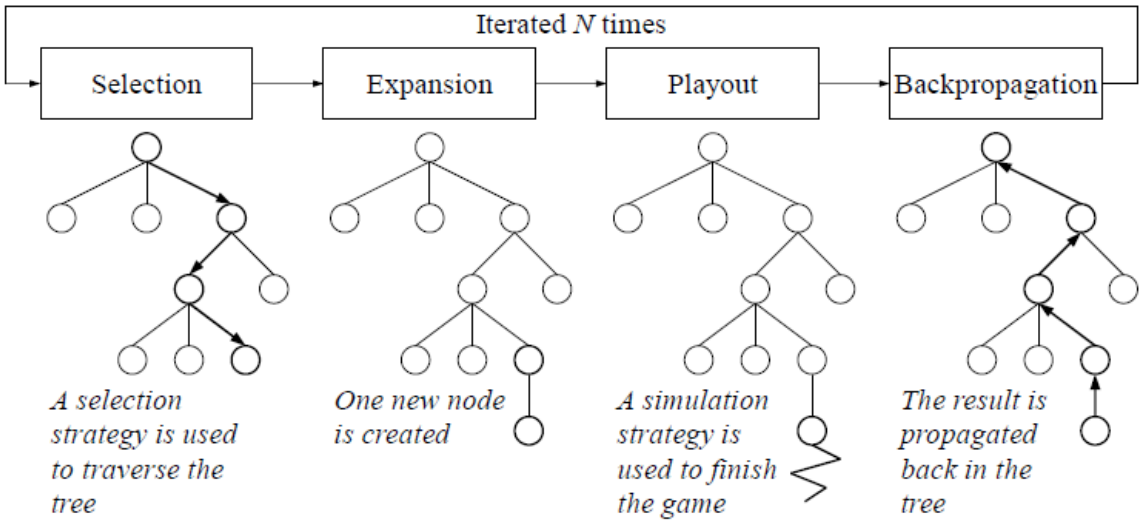

Figure 6. MCTS scheme [4]

2.4 Background Summary

This chapter provided background information about the Scotland Yard game followed by the rules of the game and how it is played. After that, related research that used several different algorithms and approaches to solve the problem of imperfect information in games was listed.

III. Methodology

3.1 Chapter Overview

Section 3.2 will describe the objective of the research along with a description of the version of the Scotland Yard game used. Section 3.3 of the chapter is going to describe some already developed algorithms that have been used while developing our algorithm. We have used two algorithms: section 3.3.1 will describe the Minimum Cost Maximum Flow (MCMF) algorithm, and section 3.3.2 will describe Floyd-Warshall's Algorithm for the shortest path problem. Then, section 3.4 will describe the approach we followed on developing the algorithm we used for the detectives in the game Scotland Yard. After that, a detailed description of the experiments done on the different stages of the development of the algorithm will be provided in section 3.5. Finally, section 3.6 will conclude the chapter with a summary.

3.2 Objectives

The main objective of the research is to develop an algorithm to play the detectives of the Scotland Yard game. The algorithm operates using the version of the game developed by Jowereit and Shashi Mittal version 2.4. The game is written in Java language,and so is the algorithm developed.

3.3 Algorithms Used as Components within our Scotland Yard Algorithm

While developing our algorithm, we have taken advantage of two already developed algorithms that added significant value. Since we are dealing with a graph-based problem and the movement of the detectives can be considered to be a flow within the graph, our algorithm incorporates the MCMF algorithm to allow for cooperation among the

detectives. Also, Floyd-Warshall's Algorithm for the shortest path problem was used. This algorithm allows the detectives to quickly move toward the evader, Mister X, when he announces his position. A detailed description of these two algorithms will be provided next.

3.3.1 Minimum Cost Maximum Flow Algorithm (MCMF)

Consider a directed weighted graph (G) with (N) number of nodes and (E) number of edges. We define a function C(E) to be the cost of flowing on any given edge (E). We also define a function Ca(E) to be the capacity of an edge (E). Let us also define the function F(G) to be a flow in the graph (G) that goes between two nodes (S) the source and (T) the sink. The MCMF problem seeks to find a flow F between (S) and (T) that has minimum cost C(∑E) and, at the same time, to have a maximum flow without exceeding the capacity of any edge Ca(E) in the route from (S) to (T). The maximum flow can involve multiple paths as long as the same number of units flowing out of source (S) ultimately arrive at destination (T). Figure 7 shows an example of the MCMF problem.

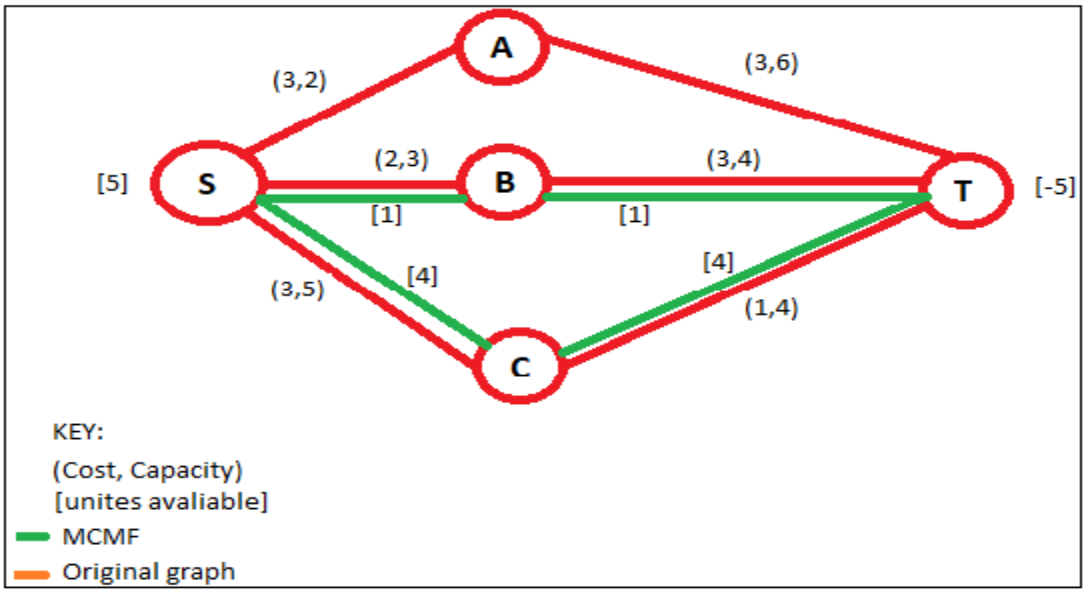

Figure 7. Example of the MCMF problem

The implantation of the MCMF algorithm used is the one provided by Cristinel Ababei in [21], which is an adapted (i.e., ported to C++) version of the famous CS2 algorithm. CS2 is the second version of the scaling algorithm for minimum-cost max-flow provided by [22]. The code is provided in Appendix B.

3.3.2 Floyd-Warshall's Algorithm

Given a graph G = (V, E) where V represents the vertices and E represent the edges. Given E = (a, b) is the edge between two vertices a and b both ∈ V, w(E) is the weight of the edge E. The shortest bath between a and b is the path between a and b were the sum of all the edges weights on the path is the lowest. The weight of edges can be the cost of traveling between the two vertices, the amount of traffic that can flow in a network, or can be the number of vertices between the two vertices. Floyd-Warshall's algorithm is the one used by the developer of the Scotland Yard game that we used in our research. The implementation of the algorithm is provided in Appendix C. Figure 7 below shows an example of the shortest path problem with the weight of edges to be the number of edges between the intended two vertices S and T.

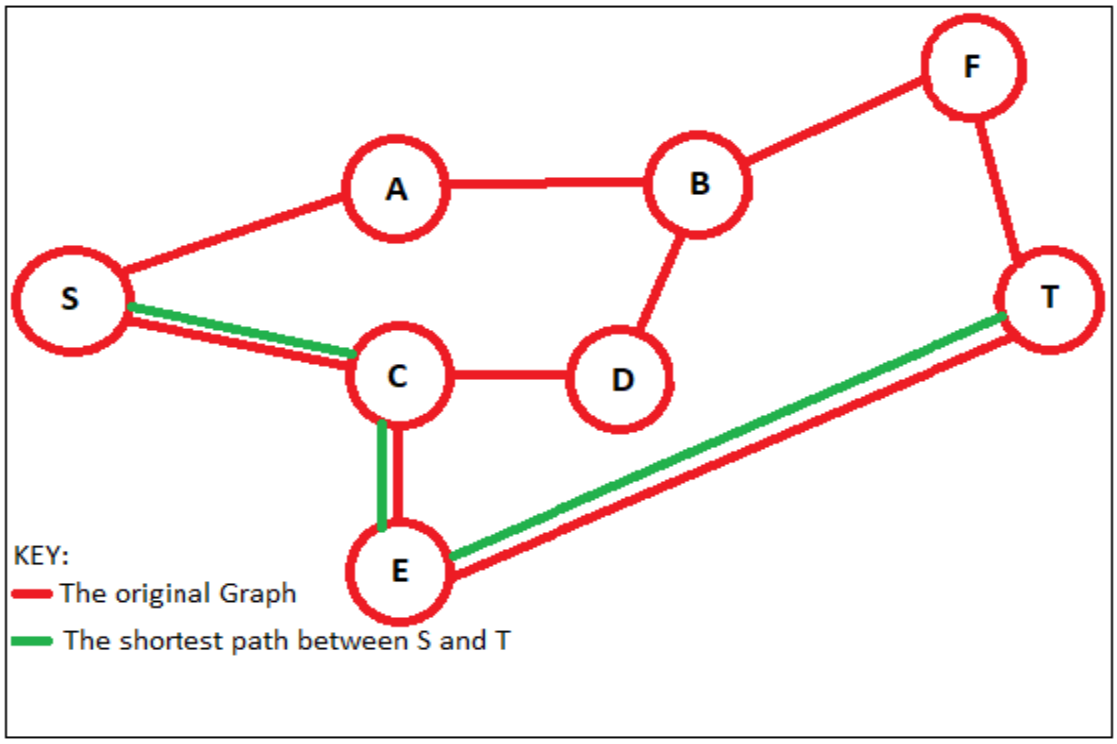

Figure 8. The shortest path between S and T using the number of edges as weight

3.4 The Algorithm for the Detectives in the Game Scotland Yard

The approach followed throughout the development of the algorithm is a domain-specific approach. However, the principles used can be applied to any pursuer and evader problem in real life. In the beginning, the turns of the game can be divided into three stages:

1. **The first two turns**: in these two turns, we do not know anything about the evader's location. So, the moves we make here are based on strategies followed by all players playing the game Scotland Yard. The plan on selecting these two moves is to get all the detectives to a well-connected station by the end of the second play.

Table 3. Detectives starting positions and well-connected stations to head to [23]

Starting position	Well-connected stations to head to	Maximum distance to Mister X
13	51,55,65,66,68,71,79,82,84,88,102,105,111,128,140	6
26	51,52	6
29,91,117	89	5
34	13,23,65,7923,51,66	6
50	23,51,66	6
53	52,55,68	6
94	46,79	6
103,112	67	5
123,138	111,153	6
141	128,140	6
155	139,140,153	6
174	128	6

The well-connected stations are stations with three, or at least two, transportation methods (Taxi, Bus, Underground). Table 1 shows a list of well-connected stations. Our target is to get as many detectives as possible to one of these stations. We start by checking the distance between each detective and all the well-connected stations.Next, we store all the stations with a distance of precisely two turns. After that, we compare the lists and make sure no two detectives are directed to the same station. Detectives with no well-connected stations within two turns are sent to stations that are as close as possible to well-connected stations.

2. **The turns where Mister X reveals its position:** In the turns where Mister X has to reveal its position as well as the method of transportation (3, 8, 13, 18, 23), our approach is to use the Floyd-Warshall's Algorithm for the shortest path between each detective and the possible locations of Mister X in the next play out. So, we first determine the possible moves allowed for Mister X based on the location revealed. Then, we exclude locations that need a type of ticket that Mister X does

not have. After that, we face three cases. In the first case, we have the possible locations in the next play out for Mister X equal to 5, which is the number of detectives. Here, we apply the Floyd-Warshall's Algorithm directly and determine the best move for each detective. We generate the best moves by generating a mapping between every detective and every possible move of Mister X. First, calculate the shortest path, which will be an integer number determining the number of hops between the detective and the locations of Mister X. Then, construct the graph between the detective's positions and Mister X possible positions with the cost of the edges to be the shortest distance value (number of hops). After that, come up with an imaginary source and sink. Then, give the edges from the source a cost of an equal number, and the edges going to the sink the same number; we chose one. Finally, apply the MCMF algorithm to determine which possible move of MisterX each detective should head to; this is done because we want all the detectives to move to the most efficient location, which allows detectives cooperation. Figure 8 shows an example of the matching for this case. The second case is when we have the number of possible moves of Mister X is less than the number of detectives. Here, we move the detectives to generated moves in the same way we did for the first case above. Then, the remaining detectives are sent to the closest well-connected nodes that are determined based on the location of Mister X. Those well-connected nodes are calculated based on Mister X location and table 1 nodes by merely finding the nearest well-connected stations to Mister X that the detectives can go to with minimal hops. The left-behind detective(s) are used on later turns in case Mister X was able to escape the

containment plan. The third case is where the number of possible moves of Mister X is more than the number of detectives. Here, methods of elimination are used to get the number of possible moves equal to the number of detectives. The first method used is the moves with the shortest distance from every detective. Then, stations with very high chances of Mister X been caught are eliminated. These stations where determined according to the analysis of data obtained from thousands of trials on the game.

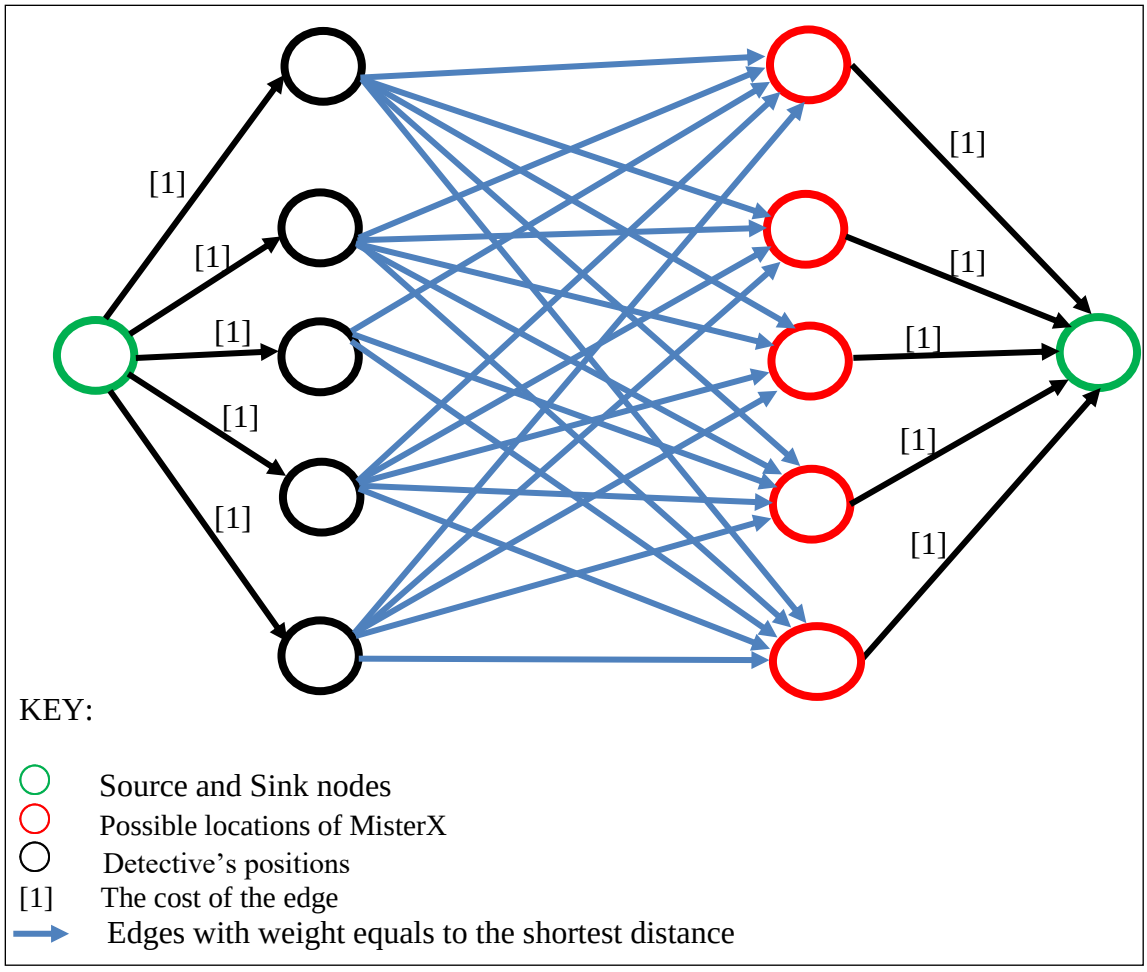

Figure 9. Example of the MCMF graph construction

3. **The play outs in between the reveals of Mister X:** Most of the algorithm's intelligence is on this part of the turns. Starting at the first turn after the reveal (4,

9, 14, 19), the possible moves of Mister X is determined based on the last revealed location. After that, the same method of elimination explained in the second case above is used. The method of transportation used by Mister X when it moved is used to gather the possible locations. These locations are usually more than the number of detectives. One way we discovered by simulations conducted was to determine paths where two moves with two transportations methods are in the same way from the revealed location. Those paths eliminate one of those in the same way since one move will be able to cover the path; figure 10 illustrates the problem.

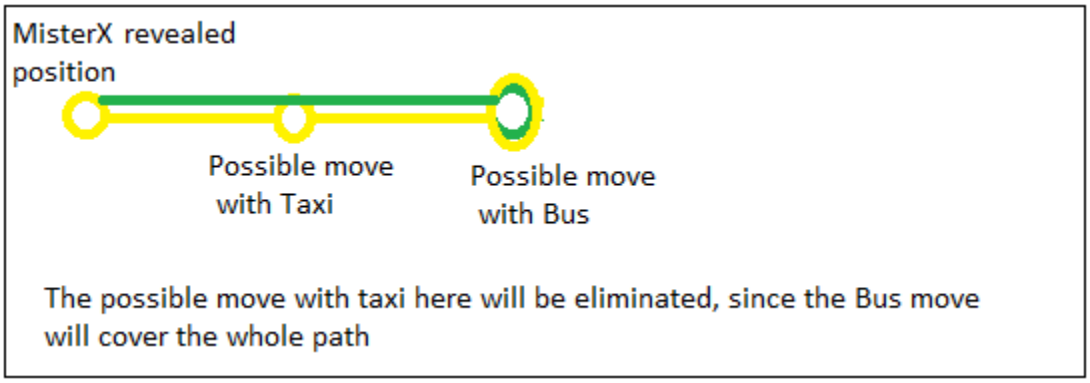

Figure 10. Example of same path move elimination

Also, locations with the closest distance from the five detectives are excluded from the list of possible moves. In the second play out after the revealed location, the same methods apply. Nevertheless, we added a learning methodology to combine the first and the second turns' transportation methods to narrow down our search space. For example, if from the revealed locations, there are ten possible taxi moves. From those ten taxi possible moves, there are only two of them that have bus connections; then if Mister X took a taxi in the first turn and a

bus in the second turn after the reveal, the algorithm would recognize that Mister X is at one of the ends of those two bus moves. This learning methodology was also added to all the turns until the next reveal.

Additionally, we applied an approach to eliminate more moves to narrow our search space. It was developed by running simulations on the game and record all the moves of Mister X on most of the distributions of the detectives' positions. This approach gave us indications of the behavior of Mister X in many different scenarios. It showed a tendency of always avoiding the closest station to each detective. Also, it showed that most of Mister X moves are avoiding underground stations. After all the eliminations and generating of possible locations of Mister X, the results are passed to the Floyd-Warshall's algorithm and MCMF algorithm to select the most efficient turns of the detectives.

3.5 Experiments Done on the Process of Developing the Algorithm

The purpose of this research is to develop an algorithm to play the detectives of the game of Scotland Yard. During the development of the algorithm, we conducted many simulations on many different stages. All the experiments listed below were run into two categories. The first category is with the starting position of Mister X selected randomly. The second set is where we test for every starting position of the 13 possible starting positions of Mister X. The first experiment ran where the detectives knew Mister X's position all the time with no intelligence what so ever. Only Floyd-Warshall's shortest path algorithm was used to move the detectives. After that, the same experiments knowing Mister X position at all the time were run with the addition of the MCMF

algorithms, and the results were improved. Then, experiments were conducted without the knowledge of Mister X's position without any intelligence added. These experiments gave poor results on catching Mister X. Next, we have added only the elimination based on the transportation method, which led to a slight improvement. That when we decided to add the elimination of the closest station to the detectives, good improvements were achieved. We then added the same path moves elimination and the combined path elimination, which gave us an excellent result. The main advantage of the algorithm from the experiments done is that it is a depth one search. The research did not go beyond one level to come up with the best possible results in a reasonable amount of time.

3.6 Methodology Summary

This chapter provided a detailed description of the algorithms used in our research to develop the algorithm for playing the detectives on the game of Scotland Yard. After that, a detailed description of the approach and algorithm developed was provided. Finally, the experiments conducted along the path of the development of the algorithm were described.

IV. Analysis and Results

4.1 Chapter Overview

This chapter will first give a brief description of the experimental setup, list the experiments done to test the efficiency of the algorithm used in the research as well as the results obtained from all the experiments.

4.2 Setup

The engines of both Mister X and the detectives are developed using Java. The algorithm developed is for the detectives only against Mister X's algorithm developed by the designer of the game Jowereit and Shashi Mittal version 2.4, which is used on the research. All the testing is done on an Intel Core i7 7th generation with 2.7 GHz clock speed and 16 GB RAM. All the results obtained from running 10,000 games.

4.3 Results Obtained with Random Starting Position of Mister X

4.3.1 Random Moves of the Detectives

The first experiments ran with the detectives know Mister X's position at all times. This was done to test the effectiveness of the Floyd-Warshall's algorithm on getting as fast as possible to Mister X position. A total of 10,000 experiments were run with the detectives winning 9,500 times, which is 95%. This showed us that even when we know Mister X's position at all the time, we still need the cooperation of the detectives in order to achieve the best results of catching Mister X. After that, 10,000 experiments were run with the setting above and with the addition of the MCMF algorithms to the detectives. The winning rate here was 100%. Because the MCMF algorithm utilizes all the five detectives and efficiently moves them all to contain Mister X, those results were

achieved. Next, 10,000 experiments were run using the rules of the game where Mister X reveals position only at 3, 8, 13, 18, and 23. The movements of the detectives on the revealed turns were using Floyd-Warshall's shortest path algorithm, and the movement on all other moves was random. The obtained results were poor; the winning rate of the detectives was only 7.5 %. The same setting was applied with the addition of the MCMF algorithm to the revealed turns of the detectives. 10,000 games were played with the winning rate of the detectives 10% only. Table 2 below shows a summary of these results.

Table 4. The winning rate of the Detectives for the position always known and random movements case.

	Position of Mister X always known	Game rules
Floyd_Warshall's	95%	7.5%
Floyd_Warshall's and MCMF	100%	10%

4.3.2 All the Moves of the Detectives Controlled

In this section, the results of the experiments done during the development of the entire algorithm are presented. The first set of experiments was run with Floyd-Warshall's algorithm and MCMF with the addition of the transportation elimination method. The winning rate of the detectives obtained based on 10,000 games played was 22%. Next, the method of learning Mister X turns was added. We have learned from the running of many experiments that Mister X often avoids moves that are the closest to anyone of the detectives. Adding this information to the algorithm and playing 10,000 games generated a winning rate of 40% for the detectives. Finally, the addition of the

same path elimination increased the winning rate of the detectives to 57% on 10,000 games played. Table 3 summarizes the results obtained.

Table 5. Results obtained from the algorithm at different stages

Transportation method eliminations	Added Closest stations to the detective's eliminations	Added Same path eliminations
22%	40%	57%

4.4 Results Obtained with Specified Starting position of Mister X

In this section, the starting position of Mister X was determined. One of the 13 possible starting positions was selected every time, and a set of 500 games were played for every starting position at every stage of 3 different stages during the development of the algorithm. The 13 starting positions are illustrated in Table 1. The three stages are: with transportation method eliminations only, adding the closest stations to the detective's eliminations, and the addition of the same path eliminations. Table 4 summarizes the results obtained in all the cases. From table 4, we can see that the starting position of Mister X plays a significant rule in winning and losing. The most detectives' wins are when Mister X starts at position 146 while Mister X has the best chances of winning when starting at position 166.

Table 6. summary of the winning rates of the detectives with the different starting positions of Mister X

	Win rate of the detectives		
Technics used / Starting position	Transportation method eliminations	+Closest stations to detectives' eliminations	+Same path eliminations
146	25%	43%	62%
35	24%	42%	59%
104	24%	42%	58%
71	23%	42%	58%
106	23%	40%	58%
51	23%	39%	58%
78	21%	37%	55%
127	21%	36%	54%
132	19%	35%	54%
45	19%	34%	53%
172	16%	33%	52%
170	15%	33%	51%
166	13%	32%	49%

4.5 Results Summary

This chapter presented the results obtained from the different experiments done on this research. We got 57 % as the best winning rate of the detectives when applying our algorithm against Mister X programmed by Jowereit and Shashi Mittal version 2.4. we also gave a brief on the best and worst starting positions of Mister X.

V. Conclusions and Recommendations

5.1 Chapter Overview

This chapter summarizes the research and the results obtained throughout the development of this thesis. In 5.2, the conclusions of the research, along with the approach followed, will be reviewed. Then, section 5.3 explains the significance of the research. Finally, section 5.4 gives recommendations for future research.

5.2 Conclusions of the Research

The research successfully developed an algorithm to play the detectives in the game of Scotland Yard. The algorithm is based on the maximum flow minimum cost algorithm, the shortest path algorithm, and specific technics in pursuit-evasion games, which are specifically developed for the game of Scotland Yard. The algorithm manages to achieve a 57% success rate in catching Mr X when the starting position is randomized. The research achieved the best winning rate of 62% when Mr X starts at position 146.

5.3 Significance of Research

As the world is moving toward digital military arsenal, the need for AI algorithms is increasing as most of these arsenals will be controlled by AI agents rather than humans. For example, Unmanned Arial Vehicles (UAV) that can be programmed to fly and hit targets without any intervention of human. Principles of programming these UAVs can be improved by using algorithms for pursuit-evasions games. The pursuit-evasion games are relatively close to many military applications. Algorithms on these games can be applied to military UAVs to follow a specific target in a certain way. The developed algorithm can help in missions where the target is moving in a certain way, and there is a

need to corner the target without knowing the exact location at all times, which is the case in most military missions.

5.4 Recommendations for Future Research

Several improvements can be considered for future research on the game of Scotland Yard. Below are three recommendations that can be applied in developing the algorithm:

1. Change the algorithm for Mr X by trying the developed algorithm in this research.
2. Add history records where the moves can be based on probabilities calculations along with the algorithm.
3. Increase the depth of the search beyond level one. Even though this might increase the search time, but it might yield impressive results.

Appendix A. The Game codes

AbstractPlayer.java

```
package game;

import java.util.LinkedList;

/**
 * Abstract class AbstractPlayer - Defines the properties of player
 *
 * @author: Shashi Mittal Date: 14-10-2002
 */
public abstract class AbstractPlayer implements Transport {
	protected Node position; // The position of the player
	protected LinkedList<Move> prevPositions;

	/**
	 * The default constructor. This does not initialize the prevPositions array.
	 * Useful for initializing player in tree search.
	 */
	AbstractPlayer(){
		//Do nothing
	}

	/**
	 * This constructor initializes the position of this player
	 *
	 * @param x
	 *            the position of this player
	 */
	AbstractPlayer(Node x) {
		position = x;
		prevPositions = new LinkedList<Move>();
		prevPositions.add(new Move(position.getPosition(), 0));
	}

	/**
	 * This method gives the position of this player
	 *
	 * @return the position of the player
	 */
	public Node getPosition() {
		return position;
	}

	/**
	 * This method returns the previous positions of the player
	 *
	 * @return the array of the previous node positions of the player
	 */
	public LinkedList<Move> getPrevPos() {
		return prevPositions;
	}

	/**
	 * This method changes the position of the Fugitive without adding it to the
	 * prevPositions array.
	 *
	 * @param n
	 *            the new node position of the detective
	 */
	public void change(Node n) {
		position = n;
	}
}
```

Detective.java

```
package game;

import i18n.I18n;
import java.util.TreeSet;

/**
 * Inherits the class AbstractPlayer to define a detective
 *
 * @author Shashi Mittal
 * @version 2.4 (19-APR-2010)
 */
public class Detective extends AbstractPlayer {

	private int taxiTickets;
	private int busTickets;
	private int ugTickets;

	/**
	 * Initializes the position of the detective
	 *
	 * @param n
	 *            the node position of the detective
	 */
	Detective(Node n) {
		super(n);
		taxiTickets = 10;
		busTickets = 8;
		ugTickets = 4;
	}

	/**
	 * Initializes the detective with the same position and number of tickets as detective d.
	 * Note that this does not copy the prevPositions linked list, to make this constructor
	 * more efficient. It is supposed to be used mainly in the tree search for best moves.
	 *
	 * @param d the detective whose copy is to be made
	 */
	Detective(Detective d){
		this.taxiTickets = d.taxiTickets;
		this.busTickets = d.busTickets;
		this.ugTickets = d.ugTickets;
		this.position = d.position;
	}

	/**
	 * This method changes the position of the detective provided he has the
	 * requisite tickets and the availability of that particular node
	 */
	public void changePosition(Node n, int ticket) {
		boolean canMove = false;
		Link[] links = getPosition().getLinks();
		for (int i = 0; i < links.length; i++) {
			Node temp = links[i].getToNode();
			int type = links[i].getType();
			if ((temp.equals(n)) && (ticket == type))
				canMove = true;
		}
		if (canMove) {
			switch (ticket) {
			case TAXI:
				if (taxiTickets == 0)
					canMove = false;
				else
					taxiTickets--;
				break;
			case BUS:
				if (busTickets == 0)
					canMove = false;
				else
					busTickets--;
				break;
			case UG:
				if (ugTickets == 0)
```

```
                                        canMove = false;
                                else
                                        ugTickets--;
                        }
                }
                if (canMove) {
                        prevPositions.add(new Move(n.getPosition(), ticket));
                        position = n;
                }
        }

        /**
         * This method is used to make the necessary changes in case the detective
         * cannot move i.e. when the detective is stranded either because it does
         * not have the required ticket, or all the neighbor nodes are occupied by
         * other detectives.
         */
        public void setStaticState() {
                Move m = prevPositions.getLast();
                prevPositions.add(new Move(m.getNode(), NONE));
        }

        /**
         * Checks if the detective can make a move or not
         *
         * @return true if the detective can move,false if the detective is stranded
         * @param board
         *            the current board positions
         */
        public boolean canMove(TestBoard board) {
                Node n = getPosition();
                Link[] lk = n.getLinks();
                boolean canMove = false;
                for (int i = 0; i < lk.length; i++) {
                        boolean canGoToThisNode = true;
                        Node toNode = lk[i].getToNode();
                        Detective[] det = board.getDetectives();
                        for (int j = 0; j < det.length; j++)
                                if (toNode.equals(det[j].getPosition()))
                                        canGoToThisNode = false;

                        int t = lk[i].getType();
                        switch (t) {
                        case TAXI:
                                if (taxiTickets <= 0)
                                        if (canGoToThisNode)
                                                canGoToThisNode = false;
                                break;
                        case BUS:
                                if (busTickets <= 0)
                                        if (canGoToThisNode)
                                                canGoToThisNode = false;
                                break;
                        case UG:
                                if (ugTickets <= 0)
                                        if (canGoToThisNode)
                                                canGoToThisNode = false;
                                break;
                        case FERRY:
                                canGoToThisNode = false;
                        }
                        if (canGoToThisNode)
                                canMove = true;
                }
                return canMove;
        }

        /**
         * This method returns the possible moves of the detective in a TreeSet
         *
         * @param board
         *            the board of which this detective is a part of
         * @return all the possible moves in TreeSet
         */
        public TreeSet<Move> getPossibleMoves(TestBoard board) {
                if (!canMove(board))
```

```
                        return null;
                Node n = getPosition();
                Link[] lk = n.getLinks();
                TreeSet<Move> possibleMoves = new TreeSet<Move>();
                for (int i = 0; i < lk.length; i++) {
                        //cannot use ferry, so ignore if the link is of type ferry
                        if (lk[i].getType() == FERRY) continue;

                        boolean canGoToThisNode = true;
                        Node toNode = lk[i].getToNode();
                        Detective[] det = board.getDetectives();
                        for (int j = 0; j < det.length; j++)
                                if (toNode.equals(det[j].getPosition()))
                                        canGoToThisNode = false;

                        int t = lk[i].getType();
                        switch (t) {
                        case TAXI:
                                if (taxiTickets <= 0)
                                        if (canGoToThisNode)
                                                canGoToThisNode = false;
                                break;
                        case BUS:
                                if (busTickets <= 0)
                                        if (canGoToThisNode)
                                                canGoToThisNode = false;
                                break;
                        case UG:
                                if (ugTickets <= 0)
                                        if (canGoToThisNode)
                                                canGoToThisNode = false;
                        }
                        if (canGoToThisNode)
                                possibleMoves.add(new Move(toNode.getPosition(), t));
                }
                return possibleMoves;
        }

        /** This method displays the current status of the detective */
        public String toString() {
                return I18n.tr("DetectiveStatus", getPosition().getPosition(), taxiTickets, busTickets,
ugTickets);
        }

        /**
         * This method calculates the mobility of the detective the mobility is a
         * parameter which depends on the number of tickets of the detectives and
         * on the ways the detective can go from his current position to adjacent
         * positions.
         *
         * @return the mobility of this detective
         */
        public int mobility() {
                int mobility = 0; //3 * taxiTickets + 2 * busTickets + ugTickets;
                for (int i = 0; i < position.getLinks().length; i++)
                        mobility += position.getLinks()[i].getType();
                return mobility;
        }

        /**
         * This method is a "quick and dirty" way of changing the position of a detective.
         * It does not verify if the move is valid or not. This is supposed to be used in
         * the tree search algorithm for finding best moves.
         *
         * @param n the new node position to which the detective will be moved to
         * @param ticket the ticket used for moving to the new position
         */
        public void change(Node n, int ticket) {
                position = n;
                if (ticket == TAXI) taxiTickets --;
                else if (ticket == BUS) busTickets --;
                else if (ticket == UG) ugTickets --;
        }
}
```

Fugitve.java

```java
package game;

/**
 * Uses class Player to define the properties of a fugitive.
 *
 * @author Shashi Mittal
 * @version 2.4 (19-APR-2010)
 */
public class Fugitive extends AbstractPlayer {
	private int blackTickets;

	/**
	 * Initializes the position of the fugitive
	 */
	Fugitive(Node n) {
		super(n);
	}

	/** Set the number of black tickets the fugitive has initially.
	 */
	public void setBlackTickets(int n){
		blackTickets = n;
	}

	/**
	 * This is called when a black ticket is used to move the fugitive.
	 * It decreases the number of black tickets by 1, if not already zero,
	 * and uses a black ticket for the move last made by the fugitive.
	 * @return true if the fugitive can use a black ticket, false otherwise
	 */
	public boolean useBlackTicket(){
		if (blackTickets > 0){
			blackTickets --;
			//retrieve the last move and change its type to black
			Move oldMove = prevPositions.getLast();
			Move newMove = new Move(oldMove.getNode(), BLACK);
			prevPositions.removeLast();
			prevPositions.add(newMove);
			return true;
		}
		return false;
	}

	/**
	 * @return the number of black tickets this fugitive has.
	 */
	public int getBlackTickets(){
		return blackTickets;
	}

	/**
	 * This method changes the position of the fugitive
	 *
	 * @param n
	 *            the node to where fugitive has to be shifted
	 *@return the type of the link connecting n and the previous position of
	 *         fugitive
	 */
	public int changePosition(Node n) {
		int type = 0;
		Link[] lk = position.getLinks();
		position = n;
		for (int i = 0; i < lk.length; i++)
			if (n.equals(lk[i].getToNode()))
				type = lk[i].getType();
		prevPositions.add(new Move(n.getPosition(), type));
		if (type == FERRY){
			if (blackTickets > 0) blackTickets --;
			else throw new IllegalArgumentException();
		}
		return type;
	}
}
```

Link.java

```
package game;

/**
 *Defines the basic properties of a Link.This class can be used to define a
 * graph or a multi-graph with weighted edges.
 *
 * @author ShashiMittal
 * @version 1.0 (18-09-2002)
 */
public class Link {
	@SuppressWarnings("unused")
	private Node from;
	private Node to;
	private int type;

	/**
	 * This constructor initializes the given link
	 *
	 * @param n
	 *            the starting node
	 * @param t
	 *            the type(weight of the node
	 */
	Link(Node n, int t) {
		from = n;
		type = t;
	}

	/**
	 * Initializes the to node of this link
	 *
	 * @param x
	 *            the to node of this link
	 */
	public void setToNode(Node x) {
		to = x;
	}

	/**
	 * Returns the to node of this link
	 *
	 * @return the to node of this link
	 */
	public Node getToNode() {
		return to;
	}

	/**
	 * This method is used to get the type(weight) of the link
	 *
	 * @return the type of the link
	 */
	public int getType() {
		return type;
	}

}
```

MapLabel.java

```
package game;

import i18n.I18n;

import java.io.IOException;
import java.io.File;

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.AlphaComposite;
import java.awt.image.BufferedImage;

import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.Timer;
import javax.imageio.ImageIO;
import javax.swing.JOptionPane;

/**
 * This class extends the JLabel class to provide the functionality of
 * overlaying images (positions of detectives) on top of the game board.
 *
 * @author Johannes Jowereit
 * @version 2.4 (19-APR-2010)
 */
public class MapLabel extends JLabel implements ActionListener {

	private static final long serialVersionUID = 2747458994628364853L;
	private static final int NO_OF_DETECTIVES = 5;

	private BufferedImage flagImage[];

	/**
	 * Contains the positions of the players (Mr. X and the detectives) on the
	 * board. For Mr. X (playerPositions[0]) and each of the detectives
	 * (playerPositions[1..n] where n is the number of detectives) the pixel
	 * coordinates of their position relative to the top left corner of the
	 * image is stored.
	 */
	private Point[] playerPositions = null;

	private int currentPlayer = -1;
	boolean blinkOn = false;
	boolean mrXVisible = false; /* Is Mr. X' Position revealed? */

	Timer blinkTimer = new Timer(1000, this);

	public MapLabel(ImageIcon imageIcon, int numDetectives) {
		super(imageIcon);
		playerPositions = new Point[numDetectives + 1];
		flagImage = new BufferedImage[NO_OF_DETECTIVES + 1];
		for (int i = 0; i <= NO_OF_DETECTIVES; i++) {
			String fileName = "./flag" + Integer.toString(i) + ".gif";
			try {
				flagImage[i] = ImageIO.read(new File(fileName));
			} catch (IOException e) {
				JOptionPane.showMessageDialog(null, I18n.tr("ErrorFileNotFound",
fileName), I18n.tr("ErrorTitle"), JOptionPane.ERROR_MESSAGE);
				System.exit(1);
			}
		}
	}

	public void paint(Graphics g) {
		super.paint(g);

		if (this.playerPositions != null) {
			Graphics2D g2 = (Graphics2D) g;

			for (int i = 0; i < playerPositions.length; i++) {
```

```
                                /**
                                 * The player's position is marked with a numbered flag, except
                                 * when: - The position does not exist OR - The player is the
                                 * current player and the blinking is in the "off" phase OR -
                                 * The player is Mr. X and his position is currently not
                                 * revealed.
                                 */
                                if (playerPositions[i] != null && !(blinkOn && i == currentPlayer)
                                                && !(i == 0 && !mrXVisible)) {

                                        Point playerPos = this.getPlayerPos(i);


        g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.7f));
                                        g2.drawImage(flagImage[i], playerPos.x, playerPos.y -
flagImage[i].getHeight(),
                                                        null);
                                }
                        }
                }
        }

        public void setPlayerPos(int player, int x, int y) {
                this.playerPositions[player] = new Point(x, y);
                this.repaintPlayerPos(player);
        }

        public void setPlayerPos(int player, Point pos) {
                this.setPlayerPos(player, pos.x, pos.y);
        }

        /**
         * Gets the position of the given player's current position on screen. The
         * label may be bigger than the image contained in it, in which case the
         * image will be centered. In this case, an offset has to be added to the
         * position retrieved in the playerPositions[] array.
         *
         * FIXME: Store xOffset and yOffset in private fields and only change them
         * when the label's size changes.
         *
         * @param player
         *            The player (0 = Mr. X, 1..n = detectives) whose position shall
         *            be determined.
         * @return The pixel coordinates of the player's position.
         */
        public Point getPlayerPos(int player) {
                if (player < 0 || player >= playerPositions.length) {
                        return null;
                }

                int xOffset = (this.getWidth() - this.getIcon().getIconWidth() < 0) ? 0
                                : (this.getWidth() - this.getIcon().getIconWidth()) / 2;
                int yOffset = (this.getHeight() - this.getIcon().getIconHeight() < 0) ? 0 : (this
                                .getHeight() - this.getIcon().getIconHeight()) / 2;

                return new Point(xOffset + playerPositions[player].x, yOffset +
playerPositions[player].y);
        }

        /**
         * Sets the currently active player and takes care of the blinking of the
         * current player's position.
         *
         * @param player
         *            The number of the new currently active player.
         */
        public void setCurrentPlayer(int player) {
                if (player != currentPlayer) {
                        /*
                         * Stop the blink timer and repaint the current player's position if
                         * it was currently not visible due to blinking.
                         */
                        if (blinkTimer.isRunning()) {
                                blinkTimer.stop();
                        }
```

```
                        blinkOn = false;
                        if (currentPlayer != -1) {
                                repaintPlayerPos(currentPlayer);
                        }

                        /* Change the current player and start the blinking timer */
                        currentPlayer = player;
                        blinkTimer.start();
                        repaintPlayerPos(currentPlayer);
                }
        }

        /**
         * Repaints the area in which the position of the given player lies.
         *
         * @param player
         *            The number of the detective, or 0 for Mr. X
         */
        public void repaintPlayerPos(int player) {
                Point playerPos = getPlayerPos(player);

                if (player != -1 && playerPos != null) {
                        int width = flagImage[player].getWidth();
                        int height = flagImage[player].getHeight();
                        this.repaint(playerPos.x, playerPos.y - height, width, height);
                }
        }

        @Override
        /**
         * Gets called when the blinking timer fires. Flips the blinking on/off switch
         * and repaints the active player's position.
         */
        public void actionPerformed(ActionEvent evt) {
                if (evt.getSource() == blinkTimer) {
                        this.blinkOn = !this.blinkOn;

                        if (this.playerPositions != null) {
                                this.repaintPlayerPos(currentPlayer);
                        }
                }
        }

}
```

Move.java

```
package game;

import i18n.I18n;
import java.util.StringTokenizer;
import java.util.Comparator;

/**
 * This class encapsulates a move for the palyers.It has two data members,one
 * for representing the node position and the other for representing the ticket
 * Type for the move
 *
 * @author Shashi Mittal
 * @version 2.4 (19-APR-2010)
 */
public class Move implements Comparator<Object>, Transport, Comparable<Object> {
	int nodeIndex;
	int ticketType;

	/**
	 * This constructor initializes the data members of the Move
	 *
	 * @param n
	 *            the node index
	 *@param t
	 *            the ticket Type
	 */
	public Move(int n, int t) {
		nodeIndex = n;
		ticketType = t;
	}

	/**
	 * This constructor initializes the fields using the string representation
	 * of the object of this class e.g. the method toString() returns the string
	 * representation of objects of this class as, for example, 46(Taxi) here 46
	 * is the node position and Taxi is the ticket type. This method takes this
	 * string as the input and then tokenizes it to get the fields.
	 *
	 * @param str
	 *            the string representation of a Move object
	 */
	public Move(String str) {
		StringTokenizer getFields = new StringTokenizer(str, " (");
		nodeIndex = Integer.parseInt(getFields.nextToken());
		String type = getFields.nextToken();
		type = type.substring(0, type.length() - 1);
		if (type.equals("None"))
			ticketType = NONE;
		if (type.equals(I18n.tr("TaxiTicket")))
			ticketType = TAXI;
		if (type.equals(I18n.tr("BusTicket")))
			ticketType = BUS;
		else if (type.equals(I18n.tr("UndergroundTicket")))
			ticketType = UG;
		else if (type.equals("FERRY") || type.equals("BLACK"))
			ticketType = BLACK;
	}

	/**
	 * This method returns the score for this object which is used in the
	 * equals(),compare() and compareTo() methods
	 *
	 * @return the score for this Move
	 */
	public int getScore() {
		return 10 * nodeIndex + ticketType;
	}

	/**
	 * This method is used to get the nodeIndex of this class
	 *
	 * @return the node of the object
	 */
```

```
        public int getNode() {
                return nodeIndex;
        }

        /**
         * This method returns the ticket type
         *
         * @return the ticket type in the object
         */
        public int getType() {
                return ticketType;
        }

        /**
         * This method gives a simple string representation of the objects of this
         * class
         *
         * @return the string representation of object of this class
         */
        public String toString() {
                return "" + nodeIndex + " (" + toStringTicket() + ")";
        }

        /**
         * This method returns the string representation of this move (which is subsequently
         * displayed in the combo box).
         * @return the string representation of the current move using the current locale
         */
    public String toDisplayString() {
        return I18n.tr("Move", nodeIndex, toStringTicket());
    }

        /**
         * This method returns the string representation of the ticket which
         * contained in this object.
         *
         * @return the String representation of the ticket type of this class
         */
        public String toStringTicket() {
                String type = "None";
                if (ticketType == TAXI)
                        type = I18n.tr("TaxiTicket");
                if (ticketType == BUS)
                        type = I18n.tr("BusTicket");
                else if (ticketType == UG)
                        type = I18n.tr("UndergroundTicket");
                else if (ticketType == BLACK || ticketType == FERRY)
                        type = "Black";
                return type;
        }

        /**
         * Compares two objects of this class and returns the score depending on the
         * values given by the getScore() method
         *
         * @param o1
         *            the first Move object
         *@param o2
         *            two second Move object
         *@return positive if score of o1 is greater than o2 0 if the scores are
         *         equal a negative value otherwise
         */
        public int compare(Object o1, Object o2) {
                Move m1 = (Move) o1;
                Move m2 = (Move) o2;
                if (m1.getScore() < m2.getScore())
                        return -1;
                else if (m1.getScore() == m2.getScore())
                        return 0;
                else
                        return 1;
        }

        /**
         * Checks if two objects of this class have the same score
         *
```

```
         * @param m1
         *            the first Move object
         *@param m2
         *            the second Move object
         *@return true if the scores are equal,false otherwise
         */
        public boolean equal(Move m1, Move m2) {
                return (m1.getScore() == m2.getScore());
        }

        /**
         * This method compares this object to another objects
         *
         * @param o
         *            the object which is to be compared to this class
         *@return same as that given by int compare(Object o1,Object o2)
         */
        public int compareTo(Object o) {
                Move m = (Move) o;
                return this.getScore() - m.getScore();
        }
}
```

Node.java

```
package game;

/**
 * Describes the basic properties of a node of the map and the various methods
 * associated with this class.
 *
 * @author Shashi Mittal
 * @version 1.0 (05-09-2002)
 */
public class Node {
	private int position;
	private Link[] links;

	/**
	 * Constructor for the Node Class Initializes the position and links of the
	 * node
	 */
	Node(int pos) {
		position = pos;
	}

	/** Initializes the Links of the Node object */
	public void addLink(Node n, int t) {
		if (links == null) {
			links = new Link[1];
			links[0] = new Link(this, t);
			links[0].setToNode(n);
		} else {
			Link[] temp = new Link[links.length + 1];
			for (int i = 0; i < links.length; i++)
				temp[i] = links[i];
			Link now = new Link(this, t);
			now.setToNode(n);
			temp[links.length] = now;
			links = temp;
		}
	}

	/** Returns the position of this node */
	public int getPosition() {
		return position;
	}

	/** Returns the links of the node */
	public Link[] getLinks() {
		return links;
	}
}
```

PlayGame.java

```
package game;
import i18n.I18n;

import java.awt.Container;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

import javax.swing.DefaultListSelectionModel;
import javax.swing.ImageIcon;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.ScrollPaneConstants;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableModel;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Paths;

import java.util.Arrays;
import java.util.List;
import java.util.Queue;
/**
 * A Dialog Box for showing a Table.
 */
class TableDialog extends JDialog implements ActionListener {
	private static final long serialVersionUID = 1L;

	TableDialog(JFrame parentFrame, String title, TestBoard b, PlayGame playGame, boolean revealAll) {
		super(parentFrame, title, false);
		setResizable(true);
		TableModel model = new PreviousMoves(b, revealAll);
		JTable table = new JTable(model);
		table.getSelectionModel().addListSelectionListener(playGame);
		setSize(500, table.getRowHeight() * (b.getCurrentMoves() + 4));
		setFont(PlayGame.font);
```

```
                    getContentPane().add(new JScrollPane(table));
                    setLocationRelativeTo(null);
          }

          public void actionPerformed(ActionEvent ae) {
                    dispose();
          }
}

/**
 * Extends the abstract table class to show the previous positions of the
 * detectives and Mr. X.
 */
class PreviousMoves extends AbstractTableModel {
          private static final long serialVersionUID = 1L;
          protected TestBoard board;
          protected int numDetectives;
          boolean revealAll;

          public PreviousMoves(TestBoard b, boolean all) {
                    board = b;
                    numDetectives = board.getDetectives().length;
                    revealAll = all;
          }

          public int getRowCount() {
                    return board.getCurrentMoves();
          }

          public int getColumnCount() {
                    // + 1 because the first entry will be for Mr. X
                    return numDetectives + 1;
          }

          public Object getValueAt(int r, int c) {
                    String pos = "";
                    if (c > 0) {
                              // information of detective
                              int ln = board.getDetectives()[c - 1].getPrevPos().size();
                              if (r + 1 < ln)
                                        pos = board.getDetectives()[c - 1].getPrevPos().get(r + 1).toString();
                    } else {
                              // information of Mr. X
                              Fugitive fg = board.getMrX();
                              if (!board.isCheckPoint(r + 1) && !revealAll)
                                        pos += "" + fg.getPrevPos().get(r + 1).toStringTicket();
                              else
                                        pos += fg.getPrevPos().get(r + 1).toString();
                    }
                    return pos;
          }

          public String getColumnName(int c) {
                    if (c > 0)
                              return I18n.tr("DetectiveColumnHeader", c);
                    else
                              return I18n.tr("MrXColumnHeader");
          }
}

/**
 * This class provides the GUI interfacing between the user and the machine It
 * uses the swing functionality to provide a good interfacing
 *
 * @author Shashi Mittal
 * @version 2.4 (19-APR-2010)
 */
public class PlayGame extends JApplet implements ActionListener, Transport, ItemListener,
                    ListSelectionListener {
          private static final long serialVersionUID = 1L;
          private static final int NO_OF_DETECTIVES = 5;

          public static Font font = new Font("SansSerif", Font.PLAIN, 15);
          TestBoard board;
          JFrame parentFrame;
          Container container;
```

```
MapLabel the_map;
String mrX = "Mr. X";
JButton start, det, mx;
JTextField msg;
int currentDetectiveIndex = 0;
Move recentMove;
Move [] D_moves = new Move[5];
Node Check_Xpos;
Move Global_move;
Link [] links_1 ;
Node [] nodes1;
List <Node> XP_nodes = new ArrayList<Node>();
List <Node> list_node1 = new ArrayList<Node>();
List <Link> llinks_1 = new ArrayList <Link>();
List <List<Node>> all_nodes = new ArrayList<List<Node>>();
Queue<Node> queue = new LinkedList<>();
int counter = 0;
Link [][] links_2;
Node [][] nodes_2;
boolean gameStarted = false;
boolean [] canMove = new boolean [] {true,true,true,true,true};
JTextField detectiveStatus;
JButton done;
JComboBox getMove;
JMenuItem newGame, exitGame, about, help, ackn;
GridBagConstraints mapC, detC, mxC, startC, doneC, getMoveC, msgC, detectiveStatusC;

/**
 * This method builds up the basic user interface between the user and the
 * machine
 *
 * @param f
 *            the parent Frame in which the JApplet is encapsulated
 */
void buildUI(JFrame f) {
        parentFrame = f;
        container = parentFrame.getContentPane();
        container.setLayout(new GridBagLayout());

        mapC = new GridBagConstraints();
        mapC.fill = GridBagConstraints.BOTH;

        int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane map = new JScrollPane(the_map = new MapLabel(new ImageIcon("./map.jpg"),
                        TestBoard.getNumberOfDetectives()), v, h);
        mapC.gridx = 0;
        mapC.gridy = 0;
        mapC.gridwidth = 7;
        mapC.ipady = 1000; // Display long map
        mapC.ipady = 600; // Display wide map
        mapC.weightx = 1.0; // should occupy all available horizontal space on
        // resizing
        mapC.weighty = 1.0; // should occupy all available vertical space on
        // resizing
        mapC.insets = new Insets(10, 10, 10, 10);
        container.add(map, mapC);

        det = new JButton(I18n.tr("DetectivesButton"));
        det.addActionListener(this);
        det.setActionCommand("detectives");
        det.setVisible(true);
        if (!gameStarted)
                det.setEnabled(false);
        detC = new GridBagConstraints();
        detC.gridx = 0;
        detC.gridy = 1;
        detC.insets = new Insets(0, 10, 5, 5);
        container.add(det, detC);

        mx = new JButton(I18n.tr("MovesButton"));
        mx.addActionListener(this);
        mx.setActionCommand(mrX);
        mx.setVisible(true);
        if (!gameStarted)
                mx.setEnabled(false);
```

```
                mxC = new GridBagConstraints();
                mxC.gridx = 1;
                mxC.gridy = 1;
                mxC.insets = new Insets(0, 5, 5, 5);
                container.add(mx, mxC);

                start = new JButton(I18n.tr("StartGameButton"));
                start.addActionListener(this);
                start.setActionCommand("start");
                if (gameStarted)
                        start.setEnabled(false);
                startC = new GridBagConstraints();
                startC.gridx = 2;
                startC.gridy = 1;
                startC.insets = new Insets(0, 5, 5, 5);
                container.add(start, startC);

                done = new JButton(I18n.tr("DoneButton"));
                done.addActionListener(this);
                done.setActionCommand("done");
                if (!gameStarted)
                        done.setEnabled(false);
                doneC = new GridBagConstraints();
                doneC.gridx = 6;
                doneC.gridy = 1;
                doneC.insets = new Insets(0, 5, 5, 5);
                doneC.fill = GridBagConstraints.HORIZONTAL; // fill the remaining space
                // at the end of row
                doneC.anchor = GridBagConstraints.EAST;
                container.add(done, doneC);

                msg = new JTextField(50);
                msgC = new GridBagConstraints();
                msgC.gridx = 3;
                msgC.gridy = 1;
                msgC.ipadx = 150; // make this one a little longer
                msgC.insets = new Insets(0, 5, 5, 5);
                container.add(msg, msgC);
                msg.setEditable(false);
                msg.setText(I18n.tr("MsgClickStartGame"));

                detectiveStatus = new JTextField(120);
                detectiveStatus.setEditable(false);
                detectiveStatusC = new GridBagConstraints();
                detectiveStatusC.gridx = 4;
                detectiveStatusC.gridy = 1;
                detectiveStatusC.ipadx = 320;
                detectiveStatusC.insets = new Insets(0, 5, 5, 5);

                getMoveC = new GridBagConstraints();
                getMoveC.gridx = 5;
                getMoveC.gridy = 1;
                getMoveC.insets = new Insets(0, 5, 5, 5);

                parentFrame.setVisible(true);
                addMenu();
        }

        /**
         * This method is used to set the menu for the game.
         */
        private void addMenu() {
                JMenuBar mbar = new JMenuBar();
                parentFrame.setJMenuBar(mbar);
                JMenu fileMenu = new JMenu(I18n.tr("FileMenu"));
                mbar.add(fileMenu);

                newGame = new JMenuItem(I18n.tr("File_NewMenuItem"));
                fileMenu.add(newGame);
                newGame.addActionListener(this);

                exitGame = new JMenuItem(I18n.tr("File_ExitMenuItem"));
                fileMenu.add(exitGame);
                exitGame.addActionListener(this);

                JMenu helpMenu = new JMenu(I18n.tr("HelpMenu"));
```

```
                mbar.add(helpMenu);

                about = new JMenuItem(I18n.tr("Help_AboutMenuItem"));
                helpMenu.add(about);
                about.addActionListener(this);

                help = new JMenuItem(I18n.tr("Help_HelpMenuItem"));
                helpMenu.add(help);
                help.addActionListener(this);
        }

        /**
         * This method handles the menu events.
         *
         * @param source
         *            the menu item which is selected by the user.
         */
        private void handleMenuEvent(Object source) {
                if (source == newGame)
                        reset();
                else if (source == exitGame)
                        System.exit(0);
                else if (source == about)
                        aboutThisGame();
                else if (source == help)
                        help();
        }

        /**
         * This method tells the user about this game.
         */
        private void aboutThisGame() {
                JOptionPane.showMessageDialog(parentFrame, I18n.tr("AboutText"), I18n.tr("AboutTitle"),
                                JOptionPane.INFORMATION_MESSAGE);
        }

        /**
         * This method provides help tips to the user.
         */
        private void help() {
                JOptionPane.showMessageDialog(parentFrame, I18n.tr("HelpText"), I18n.tr("HelpTitle"),
                                JOptionPane.INFORMATION_MESSAGE);

        }

        /**
         * This method is used to handle the events caused by the clicking of the
         * buttons.
         *
         * @param ae
         *            the ActionEvent object which has the details of the event.
         */
        public void actionPerformed(ActionEvent ae) {
                String order = ae.getActionCommand();
                Object source = ae.getSource();
                handleMenuEvent(source);
                if (order.equals("start")) {
                        gameStarted = true;
                        det.setEnabled(true);
                        mx.setEnabled(true);
                        start.setEnabled(false);
                        parentFrame.setVisible(true);
                        msg.setText(I18n.tr("MsgLoading"));
                        board = new TestBoard();

                        for (int i = 1; i <= TestBoard.getNumberOfDetectives(); i++) {
                                the_map.setPlayerPos(i, board.getPoint(board.getDetectives()[i -
1].position
                                                .getPosition()));
                        }

                        the_map.setCurrentPlayer(1);

                        parentFrame.setVisible(true);
                        msg.setText(I18n.tr("MsgLoadingDone"));
                        container.add(detectiveStatus, detectiveStatusC);
```

```
                                try {
                                        moveMrX();
                                } catch (InterruptedException e) {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                                detectiveStatus.setVisible(true);

                        } else if (order.equals("detectives")) {
                                String dets = "";
                                for (int i = 0; i < NO_OF_DETECTIVES; i++)
                                        dets += I18n.tr("DetectiveInfoText", (i + 1),
board.getDetectives()[i].toString())
                                                        + "\n";
                                JOptionPane.showMessageDialog(parentFrame, dets, I18n.tr("DetectiveInfoTitle"),
                                                JOptionPane.INFORMATION_MESSAGE);

                        } else if (order.equals(mrX)) {
                                TableDialog tb = new TableDialog(parentFrame, I18n.tr("PreviousMovesTitle"),
board,
                                                this, false);
                                tb.setVisible(true);
                        } else if (order.equals("done")) {
                                for (int i = 0; i < TestBoard.getNumberOfDetectives(); i++) {
                                if (canMove[i]) {
                                        board.changeDetectivePosition(currentDetectiveIndex,
D_moves[currentDetectiveIndex]);
                                        container.remove(getMove);
                                }

                                int nodePos =
board.getDetectives()[currentDetectiveIndex].position.getPosition();
                                Point currentDetectivePos = board.getPoint(nodePos);
                                the_map.setPlayerPos(currentDetectiveIndex + 1, currentDetectivePos.x,
                                                currentDetectivePos.y);

                                parentFrame.setVisible(true);
                                currentDetectiveIndex = (currentDetectiveIndex + 1) % NO_OF_DETECTIVES;
                                the_map.setCurrentPlayer(currentDetectiveIndex + 1);

                                if (board.isUserWin()) {
                                        humanWin();
                                        break;
                                        }
                                else if (currentDetectiveIndex == 0 && !board.isUserWin()){
                                                try {
                                                        moveMrX();
                                                } catch (InterruptedException e) {
                                                        e.printStackTrace();
                                                }
                                        /*else
                                                        getDetectiveMove();*/
                                }
                                }
        }
        }
        /**
         * Used to move the fugitive in the game.
         * @throws InterruptedException
         */
        private void moveMrX() throws InterruptedException {
                if (board.isUserWin())
                        humanWin();
                Move move = board.moveMrX();
                Global_move  = move;
                String str = "Mr. X moved by ";
                switch (move.getType()) {
                case TAXI:
                        str += "Taxi";
                        break;
                case BUS:
                        str += "Bus";
                        break;
                case UG:
                        str += "Underground";
                        break;
```

```
                case FERRY:
                        str += "unknown transport (Black Ticket)";
                        break;
                case BLACK:
                        str += "unknown transport (Black Ticket)";
                }
                if (board.isCheckPoint()) {
                        str += " and is at the position " + move.getNode();
                        the_map.mrXVisible = true;
                        //the_map.setPlayerPos(0, board.getPoint(move.getNode()).getLocation());
                        JOptionPane.showMessageDialog(parentFrame, str, "Mr. X move",
                                        JOptionPane.INFORMATION_MESSAGE);
                } else {
                        the_map.mrXVisible = false;
                }

                the_map.setPlayerPos(0, board.getPoint(move.getNode()).getLocation());
                /*JOptionPane.showMessageDialog(parentFrame, str, "Mr. X move",
                                JOptionPane.INFORMATION_MESSAGE);*/

                if (board.isMachineWin())
                        machineWin();
                else

                                getDetectiveMove();

        }

        /**
         * Called when the human player has won the game.
         */
        private void humanWin() {
                the_map.mrXVisible = true;
                reset();
                msg.setText(I18n.tr("MsgHumanWin"));
                JOptionPane.showMessageDialog(parentFrame, I18n.tr("HumanWinText"), I18n
                                .tr("HumanWinTitle"), JOptionPane.INFORMATION_MESSAGE);
                repaint();
                displayPrevPos();
        }

        /**
         * Called when the machine has won the game.
         */
        private void machineWin() {
                reset();
                msg.setText(I18n.tr("MsgMachineWin"));
                JOptionPane.showMessageDialog(parentFrame, I18n.tr("MachineWinText"), I18n
                                .tr("MachineWinTitle"), JOptionPane.INFORMATION_MESSAGE);
                detectiveStatus.setText("");
                repaint();
                displayPrevPos();
        }

        /**
         * Displays the previous positions of the players in a dialog box.
         */
        private void displayPrevPos() {
                TableDialog td = new TableDialog(parentFrame, I18n.tr("PreviousMovesTitle"), board, this,
                                true);
                td.setVisible(true);

        }

        /**
         * This method is used to reset the screen when either player wins.
         */
        private void reset() {
                try {
                        det.setEnabled(false);
                        mx.setEnabled(false);
                        start.setEnabled(true);
                        container.remove(getMove);
                        done.setEnabled(false);
                        done.setVisible(false);
                        detectiveStatus.setVisible(false);
```

```
                        the_map.setCurrentPlayer(-1);
                        getMove.repaint();
                } catch (NullPointerException e) {
                }
                currentDetectiveIndex = 0;
                parentFrame.setVisible(true);
                done.setEnabled(false);
                done.repaint();
                msg.setText(I18n.tr("MsgClickStartGame"));
                repaint();
                gameStarted = false;
        }

        /**
         * Called when the move of the detectives has to be taken from the user.
         * @throws InterruptedException
         */


        private void getDetectiveMove() throws InterruptedException {
                getMove = new JComboBox();
                getMove.setSize(new Dimension(120, 30));
                done.setEnabled(true);
                parentFrame.setVisible(true);
                Detective[] detectives = board.getDetectives();
                Link [] Xlinks = null;// array of possible links of MrX
                int [] X_ToNodes = null;//Array of possible moves of MrX
                int [] D_Positions = new int [NO_OF_DETECTIVES];//Array of detectives positions
                int [] matched = new int [10];// Array for the matched detectives and MrXpos
                /*if (board.getCurrentMoves() < 3) {
                        for (int i = 0 ; i < NO_OF_DETECTIVES; i++) {
                                if(detectives[i].canMove(board)) {
                                        TreeSet<Move> mo = detectives[i].getPossibleMoves(board);
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                                                        D_moves[i] =
moves_D[moves_D.length - 1];
                                                                        canMove[i] = true;

                                }
                                else {
                                        detectives[i].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 0 + 1));
                                        getMove.setVisible(false);
                                        canMove[i] = false;
                                }
                        }

                        done.doClick();
                        getMove.repaint();
                        repaint();
                        parentFrame.setVisible(true);
                }*/

                 if(board.getCurrentMoves() < 3 || board.isCheckPoint())
                {
                        //get Mr_x Possible moves save them as integers in X_ToNodes
                        Node Xpos = board.getMrX().getPosition();
                        Check_Xpos = Xpos;
                        Xlinks = Xpos.getLinks();
                        X_ToNodes = new int [Xlinks.length];
                        for(int k = 0 ; k < Xlinks.length ; k++) {
                                X_ToNodes[k] = Xlinks[k].getToNode().getPosition();
                        }
                        System.out.println(X_ToNodes.length);
                        //get the detectives positions at current time, save it as integers in
D_Positions
                        for (int t = 0 ; t < NO_OF_DETECTIVES; t++) {
                                D_Positions[t] = detectives[t].getPosition().getPosition();
                        }

                        int [][] shortest = new int [NO_OF_DETECTIVES][X_ToNodes.length];
                        //get the cost(Shortest distance) of moving from every detective to every
possible move of Mr_X
                        for (int p = 0 ; p < NO_OF_DETECTIVES ; p++) {
                                for(int k = 0; k < X_ToNodes.length; k++) {
```

```
                                        shortest[p][k] =
TestBoard.shortestDistance[D_Positions[p]][X_ToNodes[k]];
                                }
                        }
                        //Write the detective positions, Mr_x possible moves and
                        //the shortest distance to files
                                try {
                                write("C:\\Users\\Alamri\\Desktop\\D_Positions.txt", D_Positions);
                                write("C:\\Users\\Alamri\\Desktop\\MrX_Moves.txt",X_ToNodes);
                                write2D("C:\\Users\\Alamri\\Desktop\\Shortest.txt", shortest);
                                }
                                catch(Exception e) {System.out.println("Exception Handled_Writing");}

                        // System call for the c++ code to get the best route for every detective
                                try{MyCode();}
                                catch(Exception e) {System.out.println("Exception handled_Waitfor()");}
                        // Read the generated file from the C++ code save the read file in an array
"matched"
                        try {readC(matched);        }
                        catch(Exception e) {System.out.println("Exception Handled_Reading");}
                        //generate the moves from the read file and assign it to recentMove...
                                if (detectives[0].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[0].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D1 = 100; // used to hold the value of the
shortest distance
                                                int D1_i = 0;
                                                int Mrx_pos1 = 0;
                                                if(matched[1] == 7)
                                                        Mrx_pos1 = X_ToNodes[0];
                                                else if (matched[1] == 8)
                                                        Mrx_pos1 = X_ToNodes[1];
                                                else if (matched[1] == 9)
                                                        Mrx_pos1 = X_ToNodes[2];
                                                else if (matched[1] == 10)
                                                        Mrx_pos1 = X_ToNodes[3];
                                                else if (matched[1] == 11)
                                                        Mrx_pos1 = X_ToNodes[4];
                                                else if (matched[1] == 12)
                                                        Mrx_pos1 = X_ToNodes[5];
                                                else if (matched[1] == 13)
                                                        Mrx_pos1 = X_ToNodes[6];
                                                else if (matched[1] == 14)
                                                        Mrx_pos1 = X_ToNodes[7];
                                                else if (matched[1] == 15)
                                                        Mrx_pos1 = X_ToNodes[8];
                                                else if (matched[1] == 16)
                                                        Mrx_pos1 = X_ToNodes[9];
                                                else if (matched[1] == 17)
                                                        Mrx_pos1 = X_ToNodes[10];
                                                else if (matched[1] == 18)
                                                        Mrx_pos1 = X_ToNodes[11];
                                                else if (matched[1] == 19)
                                                        Mrx_pos1 = X_ToNodes[12];
                                                // To get the shortest path Node which is determined
by C++ code
                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos1];
                                                if(x < D1) {
                                                        D1 = x;
                                                        D1_i = i;
                                                }

                                                }
                                                D_moves[0] = moves_D[D1_i]; // Detective_1 move
                                                canMove[0] = true;
                                        }
```

```
                                        else {
                                                detectives[0].setStaticState();
                                                msg.setText(I18n.tr("MsgDetectiveStranded", 0 + 1));
                                                getMove.setVisible(false);
                                                canMove[0] = false;
                                        }


                                        if (detectives[1].canMove(board)) {
                                                //get the current detective possible moves
                                                TreeSet<Move> mo = detectives[1].getPossibleMoves(board);
                                                //convert the treeset to array
                                                Move [] moves_D = new Move [mo.size()];
                                                moves_D = mo.toArray(moves_D);
                                                //convert the array to integers positions
                                                int[] Int_moves = new int [moves_D.length];
                                                for(int i = 0; i < moves_D.length ; i++) {
                                                        Int_moves[i] = moves_D[i].getNode();
                                                }
                                                        int D2 = 100; // used to hold the value of the
shortest distance index
                                                        int D2_i = 0;
                                                        int Mrx_pos2 = 0;
                                                        if(matched[3] == 7)
                                                                Mrx_pos2 = X_ToNodes[0];
                                                        else if (matched[3] == 8)
                                                                Mrx_pos2 = X_ToNodes[1];
                                                        else if (matched[3] == 9)
                                                                Mrx_pos2 = X_ToNodes[2];
                                                        else if (matched[3] == 10)
                                                                Mrx_pos2 = X_ToNodes[3];
                                                        else if (matched[3] == 11)
                                                                Mrx_pos2 = X_ToNodes[4];
                                                        else if (matched[3] == 12)
                                                                Mrx_pos2 = X_ToNodes[5];
                                                        else if (matched[3] == 13)
                                                                Mrx_pos2 = X_ToNodes[6];
                                                        else if (matched[3] == 14)
                                                                Mrx_pos2 = X_ToNodes[7];
                                                        else if (matched[3] == 15)
                                                                Mrx_pos2 = X_ToNodes[8];
                                                        else if (matched[3] == 16)
                                                                Mrx_pos2 = X_ToNodes[9];
                                                        else if (matched[3] == 17)
                                                                Mrx_pos2 = X_ToNodes[10];
                                                        else if (matched[3] == 18)
                                                                Mrx_pos2 = X_ToNodes[11];
                                                        else if (matched[3] == 19)
                                                                Mrx_pos2 = X_ToNodes[12];

                                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos2];
                                                                if(x < D2) {
                                                                        D2 = x;
                                                                        D2_i = i;}
                                                        }
                                                        D_moves[1] = moves_D [D2_i];
                                                        canMove[1] = true;
                                                }

                                        else {
                                                detectives[1].setStaticState();
                                                msg.setText(I18n.tr("MsgDetectiveStranded", 1 + 1));
                                                getMove.setVisible(false);
                                                canMove[1] = false;
                                        }

                                        if (detectives[2].canMove(board)) {
                                                //get the current detective possible moves
                                                TreeSet<Move> mo = detectives[2].getPossibleMoves(board);
                                                //convert the treeset to array
                                                Move [] moves_D = new Move [mo.size()];
                                                moves_D = mo.toArray(moves_D);
                                                //convert the array to integers positions
                                                int[] Int_moves = new int [moves_D.length];
```

```
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D3 = 100; // used to hold the value of the
shortest distance index
                                                int D3_i = 0;
                                                int Mrx_pos3 = 0;
                                                if(matched[5] == 7)
                                                        Mrx_pos3 = X_ToNodes[0];
                                                else if (matched[5] == 8)
                                                        Mrx_pos3 = X_ToNodes[1];
                                                else if (matched[5] == 9)
                                                        Mrx_pos3 = X_ToNodes[2];
                                                else if (matched[5] == 10)
                                                        Mrx_pos3 = X_ToNodes[3];
                                                else if (matched[5] == 11)
                                                        Mrx_pos3 = X_ToNodes[4];
                                                else if (matched[5] == 12)
                                                        Mrx_pos3 = X_ToNodes[5];
                                                else if (matched[5] == 13)
                                                        Mrx_pos3 = X_ToNodes[6];
                                                else if (matched[5] == 14)
                                                        Mrx_pos3 = X_ToNodes[7];
                                                else if (matched[5] == 15)
                                                        Mrx_pos3 = X_ToNodes[8];
                                                else if (matched[5] == 16)
                                                        Mrx_pos3 = X_ToNodes[9];
                                                else if (matched[5] == 17)
                                                        Mrx_pos3 = X_ToNodes[10];
                                                else if (matched[5] == 18)
                                                        Mrx_pos3 = X_ToNodes[11];
                                                else if (matched[5] == 19)
                                                        Mrx_pos3 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos3];

                                                        if(x < D3) {
                                                                D3 = x;
                                                                D3_i = i;}
                                                }
                                                D_moves[2] = moves_D [D3_i];
                                                canMove[2] = true;
                                        }
                                else {
                                        detectives[2].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 2 + 1));
                                        getMove.setVisible(false);
                                        canMove[2] = false;
                                }
                                if (detectives[3].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[3].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D4 = 100; // used to hold the value of the
shortest distance index
                                                int D4_i = 0;
                                                int Mrx_pos4 = 0;
                                                if(matched[7] == 7)
                                                        Mrx_pos4 = X_ToNodes[0];
                                                else if (matched[7] == 8)
                                                        Mrx_pos4 = X_ToNodes[1];
                                                else if (matched[7] == 9)
                                                        Mrx_pos4 = X_ToNodes[2];
                                                else if (matched[7] == 10)
                                                        Mrx_pos4 = X_ToNodes[3];
                                                else if (matched[7] == 11)
                                                        Mrx_pos4 = X_ToNodes[4];
                                                else if (matched[7] == 12)
```

```
                                                  Mrx_pos4 = X_ToNodes[5];
                                        else if (matched[7] == 13)
                                                  Mrx_pos4 = X_ToNodes[6];
                                        else if (matched[7] == 14)
                                                  Mrx_pos4 = X_ToNodes[7];
                                        else if (matched[7] == 15)
                                                  Mrx_pos4 = X_ToNodes[8];
                                        else if (matched[7] == 16)
                                                  Mrx_pos4 = X_ToNodes[9];
                                        else if (matched[7] == 17)
                                                  Mrx_pos4 = X_ToNodes[10];
                                        else if (matched[7] == 18)
                                                  Mrx_pos4 = X_ToNodes[11];
                                        else if (matched[7] == 19)
                                                  Mrx_pos4 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                  int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos4];

                                                  if(x < D4) {
                                                            D4 = x;
                                                            D4_i = i;}
                                        }
                                        D_moves[3] = moves_D [D4_i];
                                        canMove[3] = true;
                              }
                    else {
                              detectives[3].setStaticState();
                              msg.setText(I18n.tr("MsgDetectiveStranded", 3 + 1));
                              getMove.setVisible(false);
                              canMove[3] = false;
                    }
                    if (detectives[4].canMove(board)) {
                              //get the current detective possible moves
                              TreeSet<Move> mo = detectives[4].getPossibleMoves(board);
                              //convert the treeset to array
                              Move [] moves_D = new Move [mo.size()];
                              moves_D = mo.toArray(moves_D);
                              //convert the array to integers positions
                              int[] Int_moves = new int [moves_D.length];
                              for(int i = 0; i < moves_D.length ; i++) {
                                        Int_moves[i] = moves_D[i].getNode();
                              }
                                        int D5 = 100; // used to hold the value of the
shortest distance index
                                        int D5_i = 0;
                                        int Mrx_pos5 = 0;
                                        if(matched[9] == 7)
                                                  Mrx_pos5 = X_ToNodes[0];
                                        else if (matched[9] == 8)
                                                  Mrx_pos5 = X_ToNodes[1];
                                        else if (matched[9] == 9)
                                                  Mrx_pos5 = X_ToNodes[2];
                                        else if (matched[9] == 10)
                                                  Mrx_pos5 = X_ToNodes[3];
                                        else if (matched[9] == 11)
                                                  Mrx_pos5 = X_ToNodes[4];
                                        else if (matched[9] == 12)
                                                  Mrx_pos5 = X_ToNodes[5];
                                        else if (matched[9] == 13)
                                                  Mrx_pos5 = X_ToNodes[6];
                                        else if (matched[9] == 14)
                                                  Mrx_pos5 = X_ToNodes[7];
                                        else if (matched[9] == 15)
                                                  Mrx_pos5 = X_ToNodes[8];
                                        else if (matched[9] == 16)
                                                  Mrx_pos5 = X_ToNodes[9];
                                        else if (matched[9] == 17)
                                                  Mrx_pos5 = X_ToNodes[10];
                                        else if (matched[9] == 18)
                                                  Mrx_pos5 = X_ToNodes[11];
                                        else if (matched[9] == 19)
                                                  Mrx_pos5 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                  int x =
```

```
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos5];
                                                        if(x < D5) {
                                                                D5 = x;
                                                                D5_i = i;
                                                                }
                                                }
                                                D_moves[4] = moves_D [D5_i];
                                                canMove[4] = true;
                                        }
                                else {
                                        detectives[4].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 4 + 1));
                                        getMove.setVisible(false);
                                        canMove[4] = false;
                                }
                                done.doClick();
                                getMove.repaint();
                                repaint();
                                parentFrame.setVisible(true);
        }

                else {
                                List<Link> XP_links = new ArrayList<>() ;
                                List <Node> XP_moves_4 = new ArrayList<Node>();
                                List <Node> XP_moves_5 = new ArrayList<Node>();
                                List <Node> XP_moves_6 = new ArrayList<Node>();
                                List <Node> XP_moves_7 = new ArrayList<Node>();
                                if(board.getCurrentMoves() == 4 || board.getCurrentMoves() == 9 ||
board.getCurrentMoves() == 14 ||
                                                board.getCurrentMoves() == 19) {
                                                Xlinks = Check_Xpos.getLinks();
                                                XP_links = new ArrayList<>(Arrays.asList(Xlinks));
                                                switch (Global_move.getType()) {
                                                case TAXI:
                                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                                if(XP_links.get(c).getType() != TAXI) {
                                                                        XP_links.remove(c);
                                                                }
                                                        }
                                                        break;
                                                case BUS:
                                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                                if(XP_links.get(c).getType() != BUS) {
                                                                        XP_links.remove(c);
                                                                }
                                                        }
                                                        break;
                                                case UG:
                                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                                if(XP_links.get(c).getType() != UG) {
                                                                        XP_links.remove(c);
                                                                }
                                                        }
                                                        break;
                                                case FERRY:
                                                        break;
                                                case BLACK:
                                                        break;
                                                }
                                                XP_nodes.clear();
                                                //get the possible nodes MrX is in at this play based on the
ticket type
                                                for(int i = 0; i < XP_links.size() ; i++) {
                                                        XP_nodes.add(XP_links.get(i).getToNode());
                                                }
                                                //get the possible moves of MrX
                                                //Get the possible location it is in, then get the links, then
get the nodes and add them to list of moves
                                                for(int i = 0; i < XP_nodes.size(); i++) {
                                                        Link [] Plinks = XP_nodes.get(i).getLinks();
                                                        for(int j = 0; j < Plinks.length ; j++) {

        if(!XP_moves_4.contains(Plinks[j].getToNode()))
                                                                XP_moves_4.add(Plinks[j].getToNode());
                                                        }
                                                }
```

```
                                        System.out.println("this is play #  " +
board.getCurrentMoves() + "  XP_moves_4 = " + XP_moves_4.size());
                                        //get Mr_x Possible moves save them as integers in X_ToNodes
                                        X_ToNodes = new int [XP_moves_4.size()];
                                        for(int k = 0 ; k < XP_moves_4.size(); k++) {
                                                X_ToNodes[k] = XP_moves_4.get(k).getPosition();
                                        }
                                        XP_moves_4.clear();
                                        // get the detective positions
                                        for (int t = 0 ; t < NO_OF_DETECTIVES; t++) {
                                                D_Positions[t] =
detectives[t].getPosition().getPosition();
                                        }
                                        int [][] shortest = new int
[NO_OF_DETECTIVES][X_ToNodes.length];
                                        //get the cost(Shortest distance) of moving from every
detective to every possible move of Mr_X
                                        for (int p = 0 ; p < NO_OF_DETECTIVES ; p++) {
                                                for(int k = 0; k < X_ToNodes.length; k++) {
                                                        shortest[p][k] =
TestBoard.shortestDistance[D_Positions[p]][X_ToNodes[k]];
                                                }
                                        }
                                        //Write the detective possible positions, Mr_x possible moves
and
                                        //the shortest distance to files
                                                try {
                                                write("C:\\Users\\Alamri\\Desktop\\D_Positions.txt",
D_Positions);

        write("C:\\Users\\Alamri\\Desktop\\MrX_Moves.txt",X_ToNodes);
                                                write2D("C:\\Users\\Alamri\\Desktop\\Shortest.txt",
shortest);

                                                }
                                                catch(Exception e) {System.out.println("Exception
Handled_Writing");}
                                        // System call for the c++ code to get the best route for
every detective

                                                try{MyCode();}
                                                catch(Exception e) {System.out.println("Exception
handled_Waitfor()");}
                                        // Read the generated file from the C++ code save the read
file in an array "matched"
                                        try {readC(matched);        }
                                        catch(Exception e) {System.out.println("Exception
Handled_Reading");}

                                        if (detectives[0].canMove(board)) {
                                                //get the current detective possible moves
                                                TreeSet<Move> mo =
detectives[0].getPossibleMoves(board);

                                                //convert the treeset to array
                                                Move [] moves_D = new Move [mo.size()];
                                                moves_D = mo.toArray(moves_D);
                                                //convert the array to integers positions
                                                int[] Int_moves = new int [moves_D.length];
                                                for(int i = 0; i < moves_D.length ; i++) {
                                                        Int_moves[i] = moves_D[i].getNode();
                                                }
                                                        int D1 = 100; // used to hold the value of
the shortest distance

                                                        int D1_i = 0;
                                                        int Mrx_pos1 = 0;
                                                        if(matched[1] == 7)
                                                                Mrx_pos1 = X_ToNodes[0];
                                                        else if (matched[1] == 8)
                                                                Mrx_pos1 = X_ToNodes[1];
                                                        else if (matched[1] == 9)
                                                                Mrx_pos1 = X_ToNodes[2];
                                                        else if (matched[1] == 10)
                                                                Mrx_pos1 = X_ToNodes[3];
                                                        else if (matched[1] == 11)
                                                                Mrx_pos1 = X_ToNodes[4];
                                                        else if (matched[1] == 12)
                                                                Mrx_pos1 = X_ToNodes[5];
                                                        else if (matched[1] == 13)
                                                                Mrx_pos1 = X_ToNodes[6];
```

```
                                                        else if (matched[1] == 14)
                                                                Mrx_pos1 = X_ToNodes[7];
                                                        else if (matched[1] == 15)
                                                                Mrx_pos1 = X_ToNodes[8];
                                                        else if (matched[1] == 16)
                                                                Mrx_pos1 = X_ToNodes[9];
                                                        else if (matched[1] == 17)
                                                                Mrx_pos1 = X_ToNodes[10];
                                                        else if (matched[1] == 18)
                                                                Mrx_pos1 = X_ToNodes[11];
                                                        else if (matched[1] == 19)
                                                                Mrx_pos1 = X_ToNodes[12];
                                                        // To get the shortest path Node which is
determined by C++ code
                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos1];
                                                        if(x < D1) {
                                                                D1 = x;
                                                                D1_i = i;
                                                        }

                                                        }
                                                        D_moves[0] = moves_D[D1_i]; // Detective_1
move
                                                        canMove[0] = true;
                                                }
                                        else {
                                                detectives[0].setStaticState();
                                                msg.setText(I18n.tr("MsgDetectiveStranded", 0 + 1));
                                                getMove.setVisible(false);
                                                canMove[0] = false;
                                        }


                                        if (detectives[1].canMove(board)) {
                                                //get the current detective possible moves
                                                TreeSet<Move> mo =
detectives[1].getPossibleMoves(board);
                                                //convert the treeset to array
                                                Move [] moves_D = new Move [mo.size()];
                                                moves_D = mo.toArray(moves_D);
                                                //convert the array to integers positions
                                                int[] Int_moves = new int [moves_D.length];
                                                for(int i = 0; i < moves_D.length ; i++) {
                                                        Int_moves[i] = moves_D[i].getNode();
                                                }
                                                        int D2 = 100; // used to hold the value of
the shortest distance index
                                                        int D2_i = 0;
                                                        int Mrx_pos2 = 0;
                                                        if(matched[3] == 7)
                                                                Mrx_pos2 = X_ToNodes[0];
                                                        else if (matched[3] == 8)
                                                                Mrx_pos2 = X_ToNodes[1];
                                                        else if (matched[3] == 9)
                                                                Mrx_pos2 = X_ToNodes[2];
                                                        else if (matched[3] == 10)
                                                                Mrx_pos2 = X_ToNodes[3];
                                                        else if (matched[3] == 11)
                                                                Mrx_pos2 = X_ToNodes[4];
                                                        else if (matched[3] == 12)
                                                                Mrx_pos2 = X_ToNodes[5];
                                                        else if (matched[3] == 13)
                                                                Mrx_pos2 = X_ToNodes[6];
                                                        else if (matched[3] == 14)
                                                                Mrx_pos2 = X_ToNodes[7];
                                                        else if (matched[3] == 15)
                                                                Mrx_pos2 = X_ToNodes[8];
                                                        else if (matched[3] == 16)
                                                                Mrx_pos2 = X_ToNodes[9];
                                                        else if (matched[3] == 17)
                                                                Mrx_pos2 = X_ToNodes[10];
                                                        else if (matched[3] == 18)
                                                                Mrx_pos2 = X_ToNodes[11];
                                                        else if (matched[3] == 19)
```

```
                                                                Mrx_pos2 = X_ToNodes[12];

                                                        for(int i = 0; i < Int_moves.length ; i++)
{
                                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos2];
                                                                if(x < D2) {
                                                                        D2 = x;
                                                                        D2_i = i;}
                                                        }
                                                        D_moves[1] = moves_D [D2_i];
                                                        canMove[1] = true;
                                                }

                                        else {
                                                detectives[1].setStaticState();
                                                msg.setText(I18n.tr("MsgDetectiveStranded", 1 + 1));
                                                getMove.setVisible(false);
                                                canMove[1] = false;
                                        }

                                        if (detectives[2].canMove(board)) {
                                                //get the current detective possible moves
                                                TreeSet<Move> mo =
detectives[2].getPossibleMoves(board);
                                                //convert the treeset to array
                                                Move [] moves_D = new Move [mo.size()];
                                                moves_D = mo.toArray(moves_D);
                                                //convert the array to integers positions
                                                int[] Int_moves = new int [moves_D.length];
                                                for(int i = 0; i < moves_D.length ; i++) {
                                                        Int_moves[i] = moves_D[i].getNode();
                                                }
                                                        int D3 = 100; // used to hold the value of
the shortest distance index
                                                        int D3_i = 0;
                                                        int Mrx_pos3 = 0;
                                                        if(matched[5] == 7)
                                                                Mrx_pos3 = X_ToNodes[0];
                                                        else if (matched[5] == 8)
                                                                Mrx_pos3 = X_ToNodes[1];
                                                        else if (matched[5] == 9)
                                                                Mrx_pos3 = X_ToNodes[2];
                                                        else if (matched[5] == 10)
                                                                Mrx_pos3 = X_ToNodes[3];
                                                        else if (matched[5] == 11)
                                                                Mrx_pos3 = X_ToNodes[4];
                                                        else if (matched[5] == 12)
                                                                Mrx_pos3 = X_ToNodes[5];
                                                        else if (matched[5] == 13)
                                                                Mrx_pos3 = X_ToNodes[6];
                                                        else if (matched[5] == 14)
                                                                Mrx_pos3 = X_ToNodes[7];
                                                        else if (matched[5] == 15)
                                                                Mrx_pos3 = X_ToNodes[8];
                                                        else if (matched[5] == 16)
                                                                Mrx_pos3 = X_ToNodes[9];
                                                        else if (matched[5] == 17)
                                                                Mrx_pos3 = X_ToNodes[10];
                                                        else if (matched[5] == 18)
                                                                Mrx_pos3 = X_ToNodes[11];
                                                        else if (matched[5] == 19)
                                                                Mrx_pos3 = X_ToNodes[12];

                                                        for(int i = 0; i < Int_moves.length ; i++)
{
                                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos3];
                                                                if(x < D3) {
                                                                        D3 = x;
                                                                        D3_i = i;}
                                                        }
                                                        D_moves[2] = moves_D [D3_i];
                                                        canMove[2] = true;
                                                }
                                        else {
```

```
                                        detectives[2].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 2 + 1));
                                        getMove.setVisible(false);
                                        canMove[2] = false;
                                }
                                if (detectives[3].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo =
detectives[3].getPossibleMoves(board);

                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D4 = 100; // used to hold the value of
the shortest distance index

                                                int D4_i = 0;
                                                int Mrx_pos4 = 0;
                                                if(matched[7] == 7)
                                                        Mrx_pos4 = X_ToNodes[0];
                                                else if (matched[7] == 8)
                                                        Mrx_pos4 = X_ToNodes[1];
                                                else if (matched[7] == 9)
                                                        Mrx_pos4 = X_ToNodes[2];
                                                else if (matched[7] == 10)
                                                        Mrx_pos4 = X_ToNodes[3];
                                                else if (matched[7] == 11)
                                                        Mrx_pos4 = X_ToNodes[4];
                                                else if (matched[7] == 12)
                                                        Mrx_pos4 = X_ToNodes[5];
                                                else if (matched[7] == 13)
                                                        Mrx_pos4 = X_ToNodes[6];
                                                else if (matched[7] == 14)
                                                        Mrx_pos4 = X_ToNodes[7];
                                                else if (matched[7] == 15)
                                                        Mrx_pos4 = X_ToNodes[8];
                                                else if (matched[7] == 16)
                                                        Mrx_pos4 = X_ToNodes[9];
                                                else if (matched[7] == 17)
                                                        Mrx_pos4 = X_ToNodes[10];
                                                else if (matched[7] == 18)
                                                        Mrx_pos4 = X_ToNodes[11];
                                                else if (matched[7] == 19)
                                                        Mrx_pos4 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++)
{
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos4];
                                                        if(x < D4) {
                                                                D4 = x;
                                                                D4_i = i;}
                                                }
                                                D_moves[3] = moves_D [D4_i];
                                                canMove[3] = true;
                                        }
                                else {
                                        detectives[3].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 3 + 1));
                                        getMove.setVisible(false);
                                        canMove[3] = false;
                                }
                                if (detectives[4].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo =
detectives[4].getPossibleMoves(board);

                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
```

```
                                                  }
                                                            int D5 = 100; // used to hold the value of
the shortest distance index
                                                            int D5_i = 0;
                                                            int Mrx_pos5 = 0;
                                                            if(matched[9] == 7)
                                                                     Mrx_pos5 = X_ToNodes[0];
                                                            else if (matched[9] == 8)
                                                                     Mrx_pos5 = X_ToNodes[1];
                                                            else if (matched[9] == 9)
                                                                     Mrx_pos5 = X_ToNodes[2];
                                                            else if (matched[9] == 10)
                                                                     Mrx_pos5 = X_ToNodes[3];
                                                            else if (matched[9] == 11)
                                                                     Mrx_pos5 = X_ToNodes[4];
                                                            else if (matched[9] == 12)
                                                                     Mrx_pos5 = X_ToNodes[5];
                                                            else if (matched[9] == 13)
                                                                     Mrx_pos5 = X_ToNodes[6];
                                                            else if (matched[9] == 14)
                                                                     Mrx_pos5 = X_ToNodes[7];
                                                            else if (matched[9] == 15)
                                                                     Mrx_pos5 = X_ToNodes[8];
                                                            else if (matched[9] == 16)
                                                                     Mrx_pos5 = X_ToNodes[9];
                                                            else if (matched[9] == 17)
                                                                     Mrx_pos5 = X_ToNodes[10];
                                                            else if (matched[9] == 18)
                                                                     Mrx_pos5 = X_ToNodes[11];
                                                            else if (matched[9] == 19)
                                                                     Mrx_pos5 = X_ToNodes[12];

                                                            for(int i = 0; i < Int_moves.length ; i++)
{
                                                                     int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos5];
                                                                     if(x < D5) {
                                                                              D5 = x;
                                                                              D5_i = i;
                                                                              }
                                                            }
                                                            D_moves[4] = moves_D [D5_i];
                                                            canMove[4] = true;
                                                  }
                                         else {
                                                  detectives[4].setStaticState();
                                                  msg.setText(I18n.tr("MsgDetectiveStranded", 4 + 1));
                                                  getMove.setVisible(false);
                                                  canMove[4] = false;
                                         }
                                         done.doClick();
                                         getMove.repaint();
                                         repaint();
                                         parentFrame.setVisible(true);
                                }
                                else if (board.getCurrentMoves() == 5 || board.getCurrentMoves() == 10 ||
board.getCurrentMoves() == 15 ||
                                                  board.getCurrentMoves() == 20)
                                {
                                for (int i = 0 ; i < XP_nodes.size(); i++) {
                                         Link [] Plinks = XP_nodes.get(i).getLinks();
                                         for (int j = 0 ; j < Plinks.length ; j++) {
                                                  XP_links.add(Plinks[j]);
                                         }
                                }
                                switch (Global_move.getType()) {
                                case TAXI:
                                         for(int c = 0 ; c < XP_links.size(); c++) {
                                                  if(XP_links.get(c).getType() != TAXI) {
                                                           XP_links.remove(c);
                                                  }
                                         }
                                         break;
                                case BUS:
                                         for(int c = 0 ; c < XP_links.size(); c++) {
                                                  if(XP_links.get(c).getType() != BUS) {
```

```
                                                XP_links.remove(c);
                                        }
                                }
                                break;
                        case UG:
                                for(int c = 0 ; c < XP_links.size(); c++) {
                                        if(XP_links.get(c).getType() != UG) {
                                                XP_links.remove(c);
                                        }
                                }
                                break;
                        case FERRY:
                                break;
                        case BLACK:
                                break;
                        }
                        XP_nodes.clear();
                        //get the possible nodes MrX is in at this play based on the ticket
type
                        for(int i = 0; i < XP_links.size() ; i++) {
                                XP_nodes.add(XP_links.get(i).getToNode());
                        }
                        //get the possible moves of MrX
                        //Get the possible location it is in, then get the links, then get the
nodes and add them to list of moves
                        for(int i = 0; i < XP_nodes.size(); i++) {
                                Link [] Plinks = XP_nodes.get(i).getLinks();
                                for(int j = 0; j < Plinks.length ; j++) {
                                        if(!XP_moves_5.contains(Plinks[j].getToNode()))
                                        XP_moves_5.add(Plinks[j].getToNode());
                                }
                        }
                        System.out.println("this is play #  " + board.getCurrentMoves() + "
XP_moves_5 = " + XP_moves_5.size());
                        //get Mr_x Possible moves save them as integers in X_ToNodes
                        X_ToNodes = new int [XP_moves_5.size()];
                        for(int k = 0 ; k < XP_moves_5.size(); k++) {
                                X_ToNodes[k] = XP_moves_5.get(k).getPosition();
                        }
                        XP_moves_5.clear();
                        // get the detective positions
                        for (int t = 0 ; t < NO_OF_DETECTIVES; t++) {
                                D_Positions[t] = detectives[t].getPosition().getPosition();
                        }
                        int [][] shortest = new int [NO_OF_DETECTIVES][X_ToNodes.length];
                        //get the cost(Shortest distance) of moving from every detective to
every possible move of Mr_X
                        for (int p = 0 ; p < NO_OF_DETECTIVES ; p++) {
                                for(int k = 0; k < X_ToNodes.length; k++) {
                                        shortest[p][k] =
TestBoard.shortestDistance[D_Positions[p]][X_ToNodes[k]];
                                }
                        }
                        //Write the detective possible positions, Mr_x possible moves and
                        //the shortest distance to files
                                try {
                                write("C:\\Users\\Alamri\\Desktop\\D_Positions.txt",
D_Positions);

                                write("C:\\Users\\Alamri\\Desktop\\MrX_Moves.txt",X_ToNodes);
                                write2D("C:\\Users\\Alamri\\Desktop\\Shortest.txt", shortest);
                                }
                                catch(Exception e) {System.out.println("Exception
Handled_Writing");}
                        // System call for the c++ code to get the best route for every
detective
                                try{MyCode();}
                                catch(Exception e) {System.out.println("Exception
handled_Waitfor()");}
                        // Read the generated file from the C++ code save the read file in an
array "matched"
                        try {readC(matched);        }
                        catch(Exception e) {System.out.println("Exception Handled_Reading");}
                        if (detectives[0].canMove(board)) {
                                //get the current detective possible moves
                                TreeSet<Move> mo = detectives[0].getPossibleMoves(board);
                                //convert the treeset to array
```

```
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D1 = 100; // used to hold the value of the
shortest distance
                                                int D1_i = 0;
                                                int Mrx_pos1 = 0;
                                                if(matched[1] == 7)
                                                        Mrx_pos1 = X_ToNodes[0];
                                                else if (matched[1] == 8)
                                                        Mrx_pos1 = X_ToNodes[1];
                                                else if (matched[1] == 9)
                                                        Mrx_pos1 = X_ToNodes[2];
                                                else if (matched[1] == 10)
                                                        Mrx_pos1 = X_ToNodes[3];
                                                else if (matched[1] == 11)
                                                        Mrx_pos1 = X_ToNodes[4];
                                                else if (matched[1] == 12)
                                                        Mrx_pos1 = X_ToNodes[5];
                                                else if (matched[1] == 13)
                                                        Mrx_pos1 = X_ToNodes[6];
                                                else if (matched[1] == 14)
                                                        Mrx_pos1 = X_ToNodes[7];
                                                else if (matched[1] == 15)
                                                        Mrx_pos1 = X_ToNodes[8];
                                                else if (matched[1] == 16)
                                                        Mrx_pos1 = X_ToNodes[9];
                                                else if (matched[1] == 17)
                                                        Mrx_pos1 = X_ToNodes[10];
                                                else if (matched[1] == 18)
                                                        Mrx_pos1 = X_ToNodes[11];
                                                else if (matched[1] == 19)
                                                        Mrx_pos1 = X_ToNodes[12];
                                                // To get the shortest path Node which is determined
by C++ code
                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos1];
                                                if(x < D1) {
                                                        D1 = x;
                                                        D1_i = i;
                                                }

                                                }
                                                D_moves[0] = moves_D[D1_i]; // Detective_1 move
                                                canMove[0] = true;
                                        }
                                else {
                                        detectives[0].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 0 + 1));
                                        getMove.setVisible(false);
                                        canMove[0] = false;
                                }


                                if (detectives[1].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[1].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D2 = 100; // used to hold the value of the
shortest distance index
                                                int D2_i = 0;
                                                int Mrx_pos2 = 0;
                                                if(matched[3] == 7)
                                                        Mrx_pos2 = X_ToNodes[0];
```

```
                                        else if (matched[3] == 8)
                                                Mrx_pos2 = X_ToNodes[1];
                                        else if (matched[3] == 9)
                                                Mrx_pos2 = X_ToNodes[2];
                                        else if (matched[3] == 10)
                                                Mrx_pos2 = X_ToNodes[3];
                                        else if (matched[3] == 11)
                                                Mrx_pos2 = X_ToNodes[4];
                                        else if (matched[3] == 12)
                                                Mrx_pos2 = X_ToNodes[5];
                                        else if (matched[3] == 13)
                                                Mrx_pos2 = X_ToNodes[6];
                                        else if (matched[3] == 14)
                                                Mrx_pos2 = X_ToNodes[7];
                                        else if (matched[3] == 15)
                                                Mrx_pos2 = X_ToNodes[8];
                                        else if (matched[3] == 16)
                                                Mrx_pos2 = X_ToNodes[9];
                                        else if (matched[3] == 17)
                                                Mrx_pos2 = X_ToNodes[10];
                                        else if (matched[3] == 18)
                                                Mrx_pos2 = X_ToNodes[11];
                                        else if (matched[3] == 19)
                                                Mrx_pos2 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos2];

                                                if(x < D2) {
                                                        D2 = x;
                                                        D2_i = i;}
                                        }
                                        D_moves[1] = moves_D [D2_i];
                                        canMove[1] = true;
                                }

                        else {
                                detectives[1].setStaticState();
                                msg.setText(I18n.tr("MsgDetectiveStranded", 1 + 1));
                                getMove.setVisible(false);
                                canMove[1] = false;
                        }

                        if (detectives[2].canMove(board)) {
                                //get the current detective possible moves
                                TreeSet<Move> mo = detectives[2].getPossibleMoves(board);
                                //convert the treeset to array
                                Move [] moves_D = new Move [mo.size()];
                                moves_D = mo.toArray(moves_D);
                                //convert the array to integers positions
                                int[] Int_moves = new int [moves_D.length];
                                for(int i = 0; i < moves_D.length ; i++) {
                                        Int_moves[i] = moves_D[i].getNode();
                                }
                                        int D3 = 100; // used to hold the value of the
shortest distance index
                                        int D3_i = 0;
                                        int Mrx_pos3 = 0;
                                        if(matched[5] == 7)
                                                Mrx_pos3 = X_ToNodes[0];
                                        else if (matched[5] == 8)
                                                Mrx_pos3 = X_ToNodes[1];
                                        else if (matched[5] == 9)
                                                Mrx_pos3 = X_ToNodes[2];
                                        else if (matched[5] == 10)
                                                Mrx_pos3 = X_ToNodes[3];
                                        else if (matched[5] == 11)
                                                Mrx_pos3 = X_ToNodes[4];
                                        else if (matched[5] == 12)
                                                Mrx_pos3 = X_ToNodes[5];
                                        else if (matched[5] == 13)
                                                Mrx_pos3 = X_ToNodes[6];
                                        else if (matched[5] == 14)
                                                Mrx_pos3 = X_ToNodes[7];
                                        else if (matched[5] == 15)
                                                Mrx_pos3 = X_ToNodes[8];
```

```
                                        else if (matched[5] == 16)
                                                Mrx_pos3 = X_ToNodes[9];
                                        else if (matched[5] == 17)
                                                Mrx_pos3 = X_ToNodes[10];
                                        else if (matched[5] == 18)
                                                Mrx_pos3 = X_ToNodes[11];
                                        else if (matched[5] == 19)
                                                Mrx_pos3 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos3];
                                                if(x < D3) {
                                                        D3 = x;
                                                        D3_i = i;}
                                        }
                                        D_moves[2] = moves_D [D3_i];
                                        canMove[2] = true;
                                }
                        else {
                                detectives[2].setStaticState();
                                msg.setText(I18n.tr("MsgDetectiveStranded", 2 + 1));
                                getMove.setVisible(false);
                                canMove[2] = false;
                        }
                        if (detectives[3].canMove(board)) {
                                //get the current detective possible moves
                                TreeSet<Move> mo = detectives[3].getPossibleMoves(board);
                                //convert the treeset to array
                                Move [] moves_D = new Move [mo.size()];
                                moves_D = mo.toArray(moves_D);
                                //convert the array to integers positions
                                int[] Int_moves = new int [moves_D.length];
                                for(int i = 0; i < moves_D.length ; i++) {
                                        Int_moves[i] = moves_D[i].getNode();
                                }
                                        int D4 = 100; // used to hold the value of the
shortest distance index
                                        int D4_i = 0;
                                        int Mrx_pos4 = 0;
                                        if(matched[7] == 7)
                                                Mrx_pos4 = X_ToNodes[0];
                                        else if (matched[7] == 8)
                                                Mrx_pos4 = X_ToNodes[1];
                                        else if (matched[7] == 9)
                                                Mrx_pos4 = X_ToNodes[2];
                                        else if (matched[7] == 10)
                                                Mrx_pos4 = X_ToNodes[3];
                                        else if (matched[7] == 11)
                                                Mrx_pos4 = X_ToNodes[4];
                                        else if (matched[7] == 12)
                                                Mrx_pos4 = X_ToNodes[5];
                                        else if (matched[7] == 13)
                                                Mrx_pos4 = X_ToNodes[6];
                                        else if (matched[7] == 14)
                                                Mrx_pos4 = X_ToNodes[7];
                                        else if (matched[7] == 15)
                                                Mrx_pos4 = X_ToNodes[8];
                                        else if (matched[7] == 16)
                                                Mrx_pos4 = X_ToNodes[9];
                                        else if (matched[7] == 17)
                                                Mrx_pos4 = X_ToNodes[10];
                                        else if (matched[7] == 18)
                                                Mrx_pos4 = X_ToNodes[11];
                                        else if (matched[7] == 19)
                                                Mrx_pos4 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos4];
                                                if(x < D4) {
                                                        D4 = x;
                                                        D4_i = i;}
                                        }
                                        D_moves[3] = moves_D [D4_i];
                                        canMove[3] = true;
```

```
                                        }
                                else {
                                        detectives[3].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 3 + 1));
                                        getMove.setVisible(false);
                                        canMove[3] = false;
                                }
                                if (detectives[4].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[4].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D5 = 100; // used to hold the value of the
shortest distance index
                                                int D5_i = 0;
                                                int Mrx_pos5 = 0;
                                                if(matched[9] == 7)
                                                        Mrx_pos5 = X_ToNodes[0];
                                                else if (matched[9] == 8)
                                                        Mrx_pos5 = X_ToNodes[1];
                                                else if (matched[9] == 9)
                                                        Mrx_pos5 = X_ToNodes[2];
                                                else if (matched[9] == 10)
                                                        Mrx_pos5 = X_ToNodes[3];
                                                else if (matched[9] == 11)
                                                        Mrx_pos5 = X_ToNodes[4];
                                                else if (matched[9] == 12)
                                                        Mrx_pos5 = X_ToNodes[5];
                                                else if (matched[9] == 13)
                                                        Mrx_pos5 = X_ToNodes[6];
                                                else if (matched[9] == 14)
                                                        Mrx_pos5 = X_ToNodes[7];
                                                else if (matched[9] == 15)
                                                        Mrx_pos5 = X_ToNodes[8];
                                                else if (matched[9] == 16)
                                                        Mrx_pos5 = X_ToNodes[9];
                                                else if (matched[9] == 17)
                                                        Mrx_pos5 = X_ToNodes[10];
                                                else if (matched[9] == 18)
                                                        Mrx_pos5 = X_ToNodes[11];
                                                else if (matched[9] == 19)
                                                        Mrx_pos5 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos5];

                                                        if(x < D5) {
                                                                D5 = x;
                                                                D5_i = i;
                                                                }
                                                }
                                                D_moves[4] = moves_D [D5_i];
                                                canMove[4] = true;
                                        }
                                else {
                                        detectives[4].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 4 + 1));
                                        getMove.setVisible(false);
                                        canMove[4] = false;
                                }
                                done.doClick();
                                getMove.repaint();
                                repaint();
                                parentFrame.setVisible(true);
                                }
                                else if (board.getCurrentMoves() == 6 || board.getCurrentMoves() == 11 ||
board.getCurrentMoves() == 16 ||
                                                board.getCurrentMoves() == 21)
                                {
                                for (int i = 0 ; i < XP_nodes.size(); i++) {
```

```
                                        Link [] Plinks = XP_nodes.get(i).getLinks();
                                        for (int j = 0 ; j < Plinks.length ; j++) {
                                                XP_links.add(Plinks[j]);
                                        }
                                }
                                switch (Global_move.getType()) {
                                case TAXI:
                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                if(XP_links.get(c).getType() != TAXI) {
                                                        XP_links.remove(c);
                                                }
                                        }
                                        break;
                                case BUS:
                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                if(XP_links.get(c).getType() != BUS) {
                                                        XP_links.remove(c);
                                                }
                                        }
                                        break;
                                case UG:
                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                if(XP_links.get(c).getType() != UG) {
                                                        XP_links.remove(c);
                                                }
                                        }
                                        break;
                                case FERRY:
                                        break;
                                case BLACK:
                                        break;
                                }
                                XP_nodes.clear();
                                //get the possible nodes MrX is in at this play based on the ticket
type
                                for(int i = 0; i < XP_links.size() ; i++) {
                                        XP_nodes.add(XP_links.get(i).getToNode());
                                }
                                //get the possible moves of MrX
                                //Get the possible location it is in, then get the links, then get the
nodes and add them to list of moves
                                for(int i = 0; i < XP_nodes.size(); i++) {
                                        Link [] Plinks = XP_nodes.get(i).getLinks();
                                        for(int j = 0; j < Plinks.length ; j++) {
                                                if(!XP_moves_6.contains(Plinks[j].getToNode()))
                                                XP_moves_6.add(Plinks[j].getToNode());
                                        }
                                }
                                System.out.println("this is play #  " + board.getCurrentMoves() + "
XP_moves_6 = " + XP_moves_6.size());
                                //get Mr_x Possible moves save them as integers in X_ToNodes
                                X_ToNodes = new int [XP_moves_6.size()];
                                for(int k = 0 ; k < XP_moves_6.size(); k++) {
                                        X_ToNodes[k] = XP_moves_6.get(k).getPosition();
                                }
                                XP_moves_6.clear();
                                // get the detective positions
                                for (int t = 0 ; t < NO_OF_DETECTIVES; t++) {
                                        D_Positions[t] = detectives[t].getPosition().getPosition();
                                }
                                int [][] shortest = new int [NO_OF_DETECTIVES][X_ToNodes.length];
                                //get the cost(Shortest distance) of moving from every detective to
every possible move of Mr_X
                                for (int p = 0 ; p < NO_OF_DETECTIVES ; p++) {
                                        for(int k = 0; k < X_ToNodes.length; k++) {
                                                shortest[p][k] =
TestBoard.shortestDistance[D_Positions[p]][X_ToNodes[k]];
                                        }
                                }
                                //Write the detective possible positions, Mr_x possible moves and
                                //the shortest distance to files
                                        try {
                                        write("C:\\Users\\Alamri\\Desktop\\D_Positions.txt",
D_Positions);
                                        write("C:\\Users\\Alamri\\Desktop\\MrX_Moves.txt",X_ToNodes);
                                        write2D("C:\\Users\\Alamri\\Desktop\\Shortest.txt", shortest);
```

```
                                        }
                                        catch(Exception e) {System.out.println("Exception
Handled_Writing");}
                                // System call for the c++ code to get the best route for every
detective
                                        try{MyCode();}
                                        catch(Exception e) {System.out.println("Exception
handled_Waitfor()");}
                                // Read the generated file from the C++ code save the read file in an
array "matched"
                                try {readC(matched);        }
                                catch(Exception e) {System.out.println("Exception Handled_Reading");}
                                if (detectives[0].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[0].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D1 = 100; // used to hold the value of the
shortest distance
                                                int D1_i = 0;
                                                int Mrx_pos1 = 0;
                                                if(matched[1] == 7)
                                                        Mrx_pos1 = X_ToNodes[0];
                                                else if (matched[1] == 8)
                                                        Mrx_pos1 = X_ToNodes[1];
                                                else if (matched[1] == 9)
                                                        Mrx_pos1 = X_ToNodes[2];
                                                else if (matched[1] == 10)
                                                        Mrx_pos1 = X_ToNodes[3];
                                                else if (matched[1] == 11)
                                                        Mrx_pos1 = X_ToNodes[4];
                                                else if (matched[1] == 12)
                                                        Mrx_pos1 = X_ToNodes[5];
                                                else if (matched[1] == 13)
                                                        Mrx_pos1 = X_ToNodes[6];
                                                else if (matched[1] == 14)
                                                        Mrx_pos1 = X_ToNodes[7];
                                                else if (matched[1] == 15)
                                                        Mrx_pos1 = X_ToNodes[8];
                                                else if (matched[1] == 16)
                                                        Mrx_pos1 = X_ToNodes[9];
                                                else if (matched[1] == 17)
                                                        Mrx_pos1 = X_ToNodes[10];
                                                else if (matched[1] == 18)
                                                        Mrx_pos1 = X_ToNodes[11];
                                                else if (matched[1] == 19)
                                                        Mrx_pos1 = X_ToNodes[12];
                                                // To get the shortest path Node which is determined
by C++ code
                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos1];
                                                if(x < D1) {
                                                        D1 = x;
                                                        D1_i = i;
                                                }

                                                }
                                                D_moves[0] = moves_D[D1_i]; // Detective_1 move
                                                canMove[0] = true;
                                        }
                                else {
                                        detectives[0].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 0 + 1));
                                        getMove.setVisible(false);
                                        canMove[0] = false;
                                }


                                if (detectives[1].canMove(board)) {
```

```
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[1].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D2 = 100; // used to hold the value of the
shortest distance index
                                                int D2_i = 0;
                                                int Mrx_pos2 = 0;
                                                if(matched[3] == 7)
                                                        Mrx_pos2 = X_ToNodes[0];
                                                else if (matched[3] == 8)
                                                        Mrx_pos2 = X_ToNodes[1];
                                                else if (matched[3] == 9)
                                                        Mrx_pos2 = X_ToNodes[2];
                                                else if (matched[3] == 10)
                                                        Mrx_pos2 = X_ToNodes[3];
                                                else if (matched[3] == 11)
                                                        Mrx_pos2 = X_ToNodes[4];
                                                else if (matched[3] == 12)
                                                        Mrx_pos2 = X_ToNodes[5];
                                                else if (matched[3] == 13)
                                                        Mrx_pos2 = X_ToNodes[6];
                                                else if (matched[3] == 14)
                                                        Mrx_pos2 = X_ToNodes[7];
                                                else if (matched[3] == 15)
                                                        Mrx_pos2 = X_ToNodes[8];
                                                else if (matched[3] == 16)
                                                        Mrx_pos2 = X_ToNodes[9];
                                                else if (matched[3] == 17)
                                                        Mrx_pos2 = X_ToNodes[10];
                                                else if (matched[3] == 18)
                                                        Mrx_pos2 = X_ToNodes[11];
                                                else if (matched[3] == 19)
                                                        Mrx_pos2 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos2];

                                                        if(x < D2) {
                                                                D2 = x;
                                                                D2_i = i;}
                                                }
                                                D_moves[1] = moves_D [D2_i];
                                                canMove[1] = true;
                                        }

                                else {
                                        detectives[1].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 1 + 1));
                                        getMove.setVisible(false);
                                        canMove[1] = false;
                                }

                                if (detectives[2].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[2].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D3 = 100; // used to hold the value of the
shortest distance index
                                                int D3_i = 0;
                                                int Mrx_pos3 = 0;
                                                if(matched[5] == 7)
                                                        Mrx_pos3 = X_ToNodes[0];
```

```
                                                else if (matched[5] == 8)
                                                        Mrx_pos3 = X_ToNodes[1];
                                                else if (matched[5] == 9)
                                                        Mrx_pos3 = X_ToNodes[2];
                                                else if (matched[5] == 10)
                                                        Mrx_pos3 = X_ToNodes[3];
                                                else if (matched[5] == 11)
                                                        Mrx_pos3 = X_ToNodes[4];
                                                else if (matched[5] == 12)
                                                        Mrx_pos3 = X_ToNodes[5];
                                                else if (matched[5] == 13)
                                                        Mrx_pos3 = X_ToNodes[6];
                                                else if (matched[5] == 14)
                                                        Mrx_pos3 = X_ToNodes[7];
                                                else if (matched[5] == 15)
                                                        Mrx_pos3 = X_ToNodes[8];
                                                else if (matched[5] == 16)
                                                        Mrx_pos3 = X_ToNodes[9];
                                                else if (matched[5] == 17)
                                                        Mrx_pos3 = X_ToNodes[10];
                                                else if (matched[5] == 18)
                                                        Mrx_pos3 = X_ToNodes[11];
                                                else if (matched[5] == 19)
                                                        Mrx_pos3 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos3];

                                                        if(x < D3) {
                                                                D3 = x;
                                                                D3_i = i;}
                                                }
                                                D_moves[2] = moves_D [D3_i];
                                                canMove[2] = true;
                                        }
                                else {
                                        detectives[2].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 2 + 1));
                                        getMove.setVisible(false);
                                        canMove[2] = false;
                                }
                                if (detectives[3].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[3].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D4 = 100; // used to hold the value of the
shortest distance index

                                                int D4_i = 0;
                                                int Mrx_pos4 = 0;
                                                if(matched[7] == 7)
                                                        Mrx_pos4 = X_ToNodes[0];
                                                else if (matched[7] == 8)
                                                        Mrx_pos4 = X_ToNodes[1];
                                                else if (matched[7] == 9)
                                                        Mrx_pos4 = X_ToNodes[2];
                                                else if (matched[7] == 10)
                                                        Mrx_pos4 = X_ToNodes[3];
                                                else if (matched[7] == 11)
                                                        Mrx_pos4 = X_ToNodes[4];
                                                else if (matched[7] == 12)
                                                        Mrx_pos4 = X_ToNodes[5];
                                                else if (matched[7] == 13)
                                                        Mrx_pos4 = X_ToNodes[6];
                                                else if (matched[7] == 14)
                                                        Mrx_pos4 = X_ToNodes[7];
                                                else if (matched[7] == 15)
                                                        Mrx_pos4 = X_ToNodes[8];
                                                else if (matched[7] == 16)
                                                        Mrx_pos4 = X_ToNodes[9];
```

```
                                        else if (matched[7] == 17)
                                                Mrx_pos4 = X_ToNodes[10];
                                        else if (matched[7] == 18)
                                                Mrx_pos4 = X_ToNodes[11];
                                        else if (matched[7] == 19)
                                                Mrx_pos4 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos4];
                                                if(x < D4) {
                                                        D4 = x;
                                                        D4_i = i;}
                                        }
                                        D_moves[3] = moves_D [D4_i];
                                        canMove[3] = true;
                                }
                        else {
                                detectives[3].setStaticState();
                                msg.setText(I18n.tr("MsgDetectiveStranded", 3 + 1));
                                getMove.setVisible(false);
                                canMove[3] = false;
                        }
                        if (detectives[4].canMove(board)) {
                                //get the current detective possible moves
                                TreeSet<Move> mo = detectives[4].getPossibleMoves(board);
                                //convert the treeset to array
                                Move [] moves_D = new Move [mo.size()];
                                moves_D = mo.toArray(moves_D);
                                //convert the array to integers positions
                                int[] Int_moves = new int [moves_D.length];
                                for(int i = 0; i < moves_D.length ; i++) {
                                        Int_moves[i] = moves_D[i].getNode();
                                }
                                        int D5 = 100; // used to hold the value of the
shortest distance index
                                        int D5_i = 0;
                                        int Mrx_pos5 = 0;
                                        if(matched[9] == 7)
                                                Mrx_pos5 = X_ToNodes[0];
                                        else if (matched[9] == 8)
                                                Mrx_pos5 = X_ToNodes[1];
                                        else if (matched[9] == 9)
                                                Mrx_pos5 = X_ToNodes[2];
                                        else if (matched[9] == 10)
                                                Mrx_pos5 = X_ToNodes[3];
                                        else if (matched[9] == 11)
                                                Mrx_pos5 = X_ToNodes[4];
                                        else if (matched[9] == 12)
                                                Mrx_pos5 = X_ToNodes[5];
                                        else if (matched[9] == 13)
                                                Mrx_pos5 = X_ToNodes[6];
                                        else if (matched[9] == 14)
                                                Mrx_pos5 = X_ToNodes[7];
                                        else if (matched[9] == 15)
                                                Mrx_pos5 = X_ToNodes[8];
                                        else if (matched[9] == 16)
                                                Mrx_pos5 = X_ToNodes[9];
                                        else if (matched[9] == 17)
                                                Mrx_pos5 = X_ToNodes[10];
                                        else if (matched[9] == 18)
                                                Mrx_pos5 = X_ToNodes[11];
                                        else if (matched[9] == 19)
                                                Mrx_pos5 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos5];
                                                if(x < D5) {
                                                        D5 = x;
                                                        D5_i = i;
                                                        }
                                        }
                                        D_moves[4] = moves_D [D5_i];
                                        canMove[4] = true;
                                }
```

```
                        else {
                                detectives[4].setStaticState();
                                msg.setText(I18n.tr("MsgDetectiveStranded", 4 + 1));
                                getMove.setVisible(false);
                                canMove[4] = false;
                        }
                        done.doClick();
                        getMove.repaint();
                        repaint();
                        parentFrame.setVisible(true);
                                }

                        else if (board.getCurrentMoves() == 7 || board.getCurrentMoves() == 12 ||
board.getCurrentMoves() == 17 ||
                                  board.getCurrentMoves() == 22)
                                {
                                for (int i = 0 ; i < XP_nodes.size(); i++) {
                                        Link [] Plinks = XP_nodes.get(i).getLinks();
                                        for (int j = 0 ; j < Plinks.length ; j++) {
                                                XP_links.add(Plinks[j]);
                                        }
                                }
                                switch (Global_move.getType()) {
                                case TAXI:
                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                if(XP_links.get(c).getType() != TAXI) {
                                                        XP_links.remove(c);
                                                }
                                        }
                                        break;
                                case BUS:
                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                if(XP_links.get(c).getType() != BUS) {
                                                        XP_links.remove(c);
                                                }
                                        }
                                        break;
                                case UG:
                                        for(int c = 0 ; c < XP_links.size(); c++) {
                                                if(XP_links.get(c).getType() != UG) {
                                                        XP_links.remove(c);
                                                }
                                        }
                                        break;
                                case FERRY:
                                        break;
                                case BLACK:
                                        break;
                                }
                                XP_nodes.clear();
                                //get the possible nodes MrX is in at this play based on the ticket
type
                                for(int i = 0; i < XP_links.size() ; i++) {
                                        XP_nodes.add(XP_links.get(i).getToNode());
                                }
                                //get the possible moves of MrX
                                //Get the possible location it is in, then get the links, then get the
nodes and add them to list of moves
                                for(int i = 0; i < XP_nodes.size(); i++) {
                                        Link [] Plinks = XP_nodes.get(i).getLinks();
                                        for(int j = 0; j < Plinks.length ; j++) {
                                                if(!XP_moves_7.contains(Plinks[j].getToNode()))
                                                XP_moves_7.add(Plinks[j].getToNode());
                                        }
                                }
                                System.out.println("this is play #  " + board.getCurrentMoves() + "
XP_moves_7 = " + XP_moves_7.size());
                                //get Mr_x Possible moves save them as integers in X_ToNodes
                                X_ToNodes = new int [XP_moves_7.size()];
                                for(int k = 0 ; k < XP_moves_7.size(); k++) {
                                        X_ToNodes[k] = XP_moves_7.get(k).getPosition();
                                }
                                XP_moves_7.clear();
                                // get the detective positions
                                for (int t = 0 ; t < NO_OF_DETECTIVES; t++) {
                                        D_Positions[t] = detectives[t].getPosition().getPosition();
```

```
                                        }
                                        int [][] shortest = new int [NO_OF_DETECTIVES][X_ToNodes.length];
                                        //get the cost(Shortest distance) of moving from every detective to
every possible move of Mr_X
                                        for (int p = 0 ; p < NO_OF_DETECTIVES ; p++) {
                                                for(int k = 0; k < X_ToNodes.length; k++) {
                                                        shortest[p][k] =
TestBoard.shortestDistance[D_Positions[p]][X_ToNodes[k]];
                                                }
                                        }
                                        //Write the detective possible positions, Mr_x possible moves and
                                        //the shortest distance to files
                                                try {
                                                write("C:\\Users\\Alamri\\Desktop\\D_Positions.txt",
D_Positions);
                                                write("C:\\Users\\Alamri\\Desktop\\MrX_Moves.txt",X_ToNodes);
                                                write2D("C:\\Users\\Alamri\\Desktop\\Shortest.txt", shortest);
                                                }
                                                catch(Exception e) {System.out.println("Exception
Handled_Writing");}
                                        // System call for the c++ code to get the best route for every
detective
                                                try{MyCode();}
                                                catch(Exception e) {System.out.println("Exception
handled_Waitfor()");}
                                        // Read the generated file from the C++ code save the read file in an
array "matched"
                                        try {readC(matched);        }
                                        catch(Exception e) {System.out.println("Exception Handled_Reading");}
                                        if (detectives[0].canMove(board)) {
                                                //get the current detective possible moves
                                                TreeSet<Move> mo = detectives[0].getPossibleMoves(board);
                                                //convert the treeset to array
                                                Move [] moves_D = new Move [mo.size()];
                                                moves_D = mo.toArray(moves_D);
                                                //convert the array to integers positions
                                                int[] Int_moves = new int [moves_D.length];
                                                for(int i = 0; i < moves_D.length ; i++) {
                                                        Int_moves[i] = moves_D[i].getNode();
                                                }
                                                        int D1 = 100; // used to hold the value of the
shortest distance
                                                        int D1_i = 0;
                                                        int Mrx_pos1 = 0;
                                                        if(matched[1] == 7)
                                                                Mrx_pos1 = X_ToNodes[0];
                                                        else if (matched[1] == 8)
                                                                Mrx_pos1 = X_ToNodes[1];
                                                        else if (matched[1] == 9)
                                                                Mrx_pos1 = X_ToNodes[2];
                                                        else if (matched[1] == 10)
                                                                Mrx_pos1 = X_ToNodes[3];
                                                        else if (matched[1] == 11)
                                                                Mrx_pos1 = X_ToNodes[4];
                                                        else if (matched[1] == 12)
                                                                Mrx_pos1 = X_ToNodes[5];
                                                        else if (matched[1] == 13)
                                                                Mrx_pos1 = X_ToNodes[6];
                                                        else if (matched[1] == 14)
                                                                Mrx_pos1 = X_ToNodes[7];
                                                        else if (matched[1] == 15)
                                                                Mrx_pos1 = X_ToNodes[8];
                                                        else if (matched[1] == 16)
                                                                Mrx_pos1 = X_ToNodes[9];
                                                        else if (matched[1] == 17)
                                                                Mrx_pos1 = X_ToNodes[10];
                                                        else if (matched[1] == 18)
                                                                Mrx_pos1 = X_ToNodes[11];
                                                        else if (matched[1] == 19)
                                                                Mrx_pos1 = X_ToNodes[12];
                                                        // To get the shortest path Node which is determined
by C++ code
                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos1];
                                                        if(x < D1) {
```

```
                                                D1 = x;
                                                D1_i = i;
                                        }

                                        }
                                        D_moves[0] = moves_D[D1_i]; // Detective_1 move
                                        canMove[0] = true;
                                }
                        else {
                                detectives[0].setStaticState();
                                msg.setText(I18n.tr("MsgDetectiveStranded", 0 + 1));
                                getMove.setVisible(false);
                                canMove[0] = false;
                        }


                        if (detectives[1].canMove(board)) {
                                //get the current detective possible moves
                                TreeSet<Move> mo = detectives[1].getPossibleMoves(board);
                                //convert the treeset to array
                                Move [] moves_D = new Move [mo.size()];
                                moves_D = mo.toArray(moves_D);
                                //convert the array to integers positions
                                int[] Int_moves = new int [moves_D.length];
                                for(int i = 0; i < moves_D.length ; i++) {
                                        Int_moves[i] = moves_D[i].getNode();
                                }
                                        int D2 = 100; // used to hold the value of the
shortest distance index
                                        int D2_i = 0;
                                        int Mrx_pos2 = 0;
                                        if(matched[3] == 7)
                                                Mrx_pos2 = X_ToNodes[0];
                                        else if (matched[3] == 8)
                                                Mrx_pos2 = X_ToNodes[1];
                                        else if (matched[3] == 9)
                                                Mrx_pos2 = X_ToNodes[2];
                                        else if (matched[3] == 10)
                                                Mrx_pos2 = X_ToNodes[3];
                                        else if (matched[3] == 11)
                                                Mrx_pos2 = X_ToNodes[4];
                                        else if (matched[3] == 12)
                                                Mrx_pos2 = X_ToNodes[5];
                                        else if (matched[3] == 13)
                                                Mrx_pos2 = X_ToNodes[6];
                                        else if (matched[3] == 14)
                                                Mrx_pos2 = X_ToNodes[7];
                                        else if (matched[3] == 15)
                                                Mrx_pos2 = X_ToNodes[8];
                                        else if (matched[3] == 16)
                                                Mrx_pos2 = X_ToNodes[9];
                                        else if (matched[3] == 17)
                                                Mrx_pos2 = X_ToNodes[10];
                                        else if (matched[3] == 18)
                                                Mrx_pos2 = X_ToNodes[11];
                                        else if (matched[3] == 19)
                                                Mrx_pos2 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos2];

                                                if(x < D2) {
                                                        D2 = x;
                                                        D2_i = i;}
                                        }
                                        D_moves[1] = moves_D [D2_i];
                                        canMove[1] = true;
                                }

                        else {
                                detectives[1].setStaticState();
                                msg.setText(I18n.tr("MsgDetectiveStranded", 1 + 1));
                                getMove.setVisible(false);
                                canMove[1] = false;
                        }
```

```
                                if (detectives[2].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[2].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D3 = 100; // used to hold the value of the
shortest distance index
                                                int D3_i = 0;
                                                int Mrx_pos3 = 0;
                                                if(matched[5] == 7)
                                                        Mrx_pos3 = X_ToNodes[0];
                                                else if (matched[5] == 8)
                                                        Mrx_pos3 = X_ToNodes[1];
                                                else if (matched[5] == 9)
                                                        Mrx_pos3 = X_ToNodes[2];
                                                else if (matched[5] == 10)
                                                        Mrx_pos3 = X_ToNodes[3];
                                                else if (matched[5] == 11)
                                                        Mrx_pos3 = X_ToNodes[4];
                                                else if (matched[5] == 12)
                                                        Mrx_pos3 = X_ToNodes[5];
                                                else if (matched[5] == 13)
                                                        Mrx_pos3 = X_ToNodes[6];
                                                else if (matched[5] == 14)
                                                        Mrx_pos3 = X_ToNodes[7];
                                                else if (matched[5] == 15)
                                                        Mrx_pos3 = X_ToNodes[8];
                                                else if (matched[5] == 16)
                                                        Mrx_pos3 = X_ToNodes[9];
                                                else if (matched[5] == 17)
                                                        Mrx_pos3 = X_ToNodes[10];
                                                else if (matched[5] == 18)
                                                        Mrx_pos3 = X_ToNodes[11];
                                                else if (matched[5] == 19)
                                                        Mrx_pos3 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos3];
                                                        if(x < D3) {
                                                                D3 = x;
                                                                D3_i = i;}
                                                }
                                                D_moves[2] = moves_D [D3_i];
                                                canMove[2] = true;
                                        }
                                else {
                                        detectives[2].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 2 + 1));
                                        getMove.setVisible(false);
                                        canMove[2] = false;
                                }
                                if (detectives[3].canMove(board)) {
                                        //get the current detective possible moves
                                        TreeSet<Move> mo = detectives[3].getPossibleMoves(board);
                                        //convert the treeset to array
                                        Move [] moves_D = new Move [mo.size()];
                                        moves_D = mo.toArray(moves_D);
                                        //convert the array to integers positions
                                        int[] Int_moves = new int [moves_D.length];
                                        for(int i = 0; i < moves_D.length ; i++) {
                                                Int_moves[i] = moves_D[i].getNode();
                                        }
                                                int D4 = 100; // used to hold the value of the
shortest distance index
                                                int D4_i = 0;
                                                int Mrx_pos4 = 0;
                                                if(matched[7] == 7)
                                                        Mrx_pos4 = X_ToNodes[0];
                                                else if (matched[7] == 8)
```

```
                                                  Mrx_pos4 = X_ToNodes[1];
                                        else if (matched[7] == 9)
                                                  Mrx_pos4 = X_ToNodes[2];
                                        else if (matched[7] == 10)
                                                  Mrx_pos4 = X_ToNodes[3];
                                        else if (matched[7] == 11)
                                                  Mrx_pos4 = X_ToNodes[4];
                                        else if (matched[7] == 12)
                                                  Mrx_pos4 = X_ToNodes[5];
                                        else if (matched[7] == 13)
                                                  Mrx_pos4 = X_ToNodes[6];
                                        else if (matched[7] == 14)
                                                  Mrx_pos4 = X_ToNodes[7];
                                        else if (matched[7] == 15)
                                                  Mrx_pos4 = X_ToNodes[8];
                                        else if (matched[7] == 16)
                                                  Mrx_pos4 = X_ToNodes[9];
                                        else if (matched[7] == 17)
                                                  Mrx_pos4 = X_ToNodes[10];
                                        else if (matched[7] == 18)
                                                  Mrx_pos4 = X_ToNodes[11];
                                        else if (matched[7] == 19)
                                                  Mrx_pos4 = X_ToNodes[12];

                                        for(int i = 0; i < Int_moves.length ; i++) {
                                                  int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos4];

                                                  if(x < D4) {
                                                            D4 = x;
                                                            D4_i = i;}
                                        }
                                        D_moves[3] = moves_D [D4_i];
                                        canMove[3] = true;
                              }
                    else {
                              detectives[3].setStaticState();
                              msg.setText(I18n.tr("MsgDetectiveStranded", 3 + 1));
                              getMove.setVisible(false);
                              canMove[3] = false;
                    }
                    if (detectives[4].canMove(board)) {
                              //get the current detective possible moves
                              TreeSet<Move> mo = detectives[4].getPossibleMoves(board);
                              //convert the treeset to array
                              Move [] moves_D = new Move [mo.size()];
                              moves_D = mo.toArray(moves_D);
                              //convert the array to integers positions
                              int[] Int_moves = new int [moves_D.length];
                              for(int i = 0; i < moves_D.length ; i++) {
                                        Int_moves[i] = moves_D[i].getNode();
                              }
                                        int D5 = 100; // used to hold the value of the
shortest distance index
                                        int D5_i = 0;
                                        int Mrx_pos5 = 0;
                                        if(matched[9] == 7)
                                                  Mrx_pos5 = X_ToNodes[0];
                                        else if (matched[9] == 8)
                                                  Mrx_pos5 = X_ToNodes[1];
                                        else if (matched[9] == 9)
                                                  Mrx_pos5 = X_ToNodes[2];
                                        else if (matched[9] == 10)
                                                  Mrx_pos5 = X_ToNodes[3];
                                        else if (matched[9] == 11)
                                                  Mrx_pos5 = X_ToNodes[4];
                                        else if (matched[9] == 12)
                                                  Mrx_pos5 = X_ToNodes[5];
                                        else if (matched[9] == 13)
                                                  Mrx_pos5 = X_ToNodes[6];
                                        else if (matched[9] == 14)
                                                  Mrx_pos5 = X_ToNodes[7];
                                        else if (matched[9] == 15)
                                                  Mrx_pos5 = X_ToNodes[8];
                                        else if (matched[9] == 16)
                                                  Mrx_pos5 = X_ToNodes[9];
                                        else if (matched[9] == 17)
```

```
                                                        Mrx_pos5 = X_ToNodes[10];
                                                else if (matched[9] == 18)
                                                        Mrx_pos5 = X_ToNodes[11];
                                                else if (matched[9] == 19)
                                                        Mrx_pos5 = X_ToNodes[12];

                                                for(int i = 0; i < Int_moves.length ; i++) {
                                                        int x =
TestBoard.shortestDistance[Int_moves[i]][Mrx_pos5];

                                                        if(x < D5) {
                                                                D5 = x;
                                                                D5_i = i;
                                                                }
                                                }
                                                D_moves[4] = moves_D [D5_i];
                                                canMove[4] = true;
                                        }
                                else {
                                        detectives[4].setStaticState();
                                        msg.setText(I18n.tr("MsgDetectiveStranded", 4 + 1));
                                        getMove.setVisible(false);
                                        canMove[4] = false;
                                }
                                done.doClick();
                                getMove.repaint();
                                repaint();
                                parentFrame.setVisible(true);
                                        }

                        }
}

        public static void readC(int [] x) throws IOException {
                Scanner scanner = new Scanner(new File("C:\\Users\\Alamri\\Desktop\\matched.txt"));
                int i = 0;
                while(scanner.hasNextInt()){
                   x[i++] = scanner.nextInt();
                           }
                scanner.close();
                 }


        public static void MyCode() throws InterruptedException{
                try
                {
                        Process
process=Runtime.getRuntime().exec("C:\\Users\\Alamri\\source\\repos\\Project2\\Debug\\Project2",
                                        null,new File("C:\\Users\\Alamri\\source\\repos\\Project2\\Debug"));
                        InputStream stdout = process.getInputStream();
                        while(stdout.read() >=0 ) {
                                process.waitFor();
                        }
                } catch (IOException e)
                {
                    e.printStackTrace();
                }

        }

        public static void write (String filename, int[]x) throws IOException{
                    BufferedWriter outputWriter = null;
                    outputWriter = new BufferedWriter(new FileWriter(filename));
                    for (int i = 0; i < x.length; i++) {
                      outputWriter.write(Integer.toString(x[i]));
                      outputWriter.newLine();
                    }
                    outputWriter.flush();
                    outputWriter.close();
                }

        public static void write2D (String filename, int[][]x) throws IOException{
                    BufferedWriter outputWriter = null;
                    outputWriter = new BufferedWriter(new FileWriter(filename));
                    for (int i = 0; i < x.length; i++) {
                             for(int j = 0 ; j < x[i].length ; j++) {
                                        outputWriter.write(Integer.toString(x[i][j]));
```

```
                                        outputWriter.newLine();
                                }
                        }
                        outputWriter.flush();
                        outputWriter.close();
                }
        /**
         * Called when the item in the JComboBox is changed
         *
         * @param ie
         *            the ItemEvent object which has the details of this event
         */
        public void itemStateChanged(ItemEvent ie) {
                String move = (String) ie.getItem();
                recentMove = new Move(move);
        }

        /**
         * This is the main method. Called when run as an application.
         */
        public static void main(String args[]) {
                JFrame f = new JFrame(I18n.tr("WindowTitle"));
                PlayGame game = new PlayGame();
                game.buildUI(f);
                f.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent we) {
                                System.exit(0);
                        }
                });

                f.setSize(new Dimension(1024, 700));
                f.setVisible(true);
                f.setResizable(true);
        }

public static Node[] removeTheElement(Node[] arr,
            int index)
{

// If the array is empty
// or the index is not in array range
// return the original array
if (arr == null
|| index < 0
|| index >= arr.length) {

return arr;
}

// Create another array of size one less
Node[] anotherArray = new Node[arr.length - 1];

// Copy the elements from starting till index
// from original array to the other array
System.arraycopy(arr, 0, anotherArray, 0, index);

// Copy the elements from index + 1 till end
// from original array to the other array
System.arraycopy(arr, index + 1,
anotherArray, index,
arr.length - index - 1);

// return the resultant array
return anotherArray;
}

        /**
         * This method displays the previous positions of the detectives and Mr. X on the
         * map after the game is over.
         */
        @Override
        public void valueChanged(ListSelectionEvent evt) {
                if (!gameStarted) {
                        DefaultListSelectionModel table = (DefaultListSelectionModel) evt.getSource();
                        int index = table.getMaxSelectionIndex();
                        for (int i = 0; i <= NO_OF_DETECTIVES; i++) {
```

```
                                LinkedList<Move> prevPos;
                                if (i == 0) {
                                        prevPos = board.getMrX().getPrevPos();
                                } else {
                                        prevPos = board.getDetectives()[i - 1].getPrevPos();
                                }

                                int pos = (prevPos.size() <= index + 1 ? prevPos.getLast().getNode() :
prevPos.get(
                                                index + 1).getNode());
                                the_map.setPlayerPos(i, board.getPoint(pos));
                                the_map.setCurrentPlayer(-1);
                        }
                }
        }
}
```

TestBoard.java

```
package game;

import i18n.I18n;

import java.awt.Point;

import java.io.File;
import java.io.RandomAccessFile;
import java.io.IOException;

import java.util.Arrays;
import java.util.TreeSet;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.StringTokenizer;
import java.util.concurrent.atomic.AtomicBoolean;

import javax.swing.JOptionPane;

/**
 * Defines the board for the game. The main workhorse of the game.
 *
 * @author Shashi Mittal
 * @version 2.4 (19-APR-2010)
 */
public class TestBoard implements Transport, Comparator<Object>, Comparable<Object> {
	static private final int NO_OF_MOVES = 23;
	static private final int NO_OF_DETECTIVES = 5;
	static private final int CHECK_POINTS = 5;
	static private final int INF = 100;
	static private final int WIN = 200;
	static private final int LOSE = -200;
	static private final int NBEST = 1000000000;
	static private final int BLACK_TICKETS = 5;

	static private int DEPTH = 2;
	static public int[][] shortestDistance;
	static private Node[] nodes;
	static private Point[] nodePositions;
	static private int[] checkPoints;
	static private int noOfNodes;

	private int currentMoves;
	private Detective[] detectives;
	private Fugitive MrX;

	/**
	 * This constructor initializes the board
	 */
	TestBoard() {
		currentMoves = 0;
		checkPoints = new int[CHECK_POINTS];
		for (int i = 0; i < CHECK_POINTS; i++)
			checkPoints[i] = 3 + 5*i;
		readFile();
		readPosFile();
		detectives = new Detective[NO_OF_DETECTIVES];
		int partition = nodes.length / NO_OF_DETECTIVES - 1;
		int part = 1;
		for (int i = 0; i < NO_OF_DETECTIVES; i++) {
			int rnd = (int) (partition * Math.random());
			int pos = rnd + part;
			detectives[i] = new Detective(nodes[pos]);
			part += partition;
		}
		int xPos = 0;
		boolean done = true;
		do {
			xPos = 1 + (int) (noOfNodes * Math.random());
			for (int i = 0; i < NO_OF_DETECTIVES; i++)
				if (xPos == detectives[i].getPosition().getPosition())
					done = false;
```

```
                } while (!done);
                MrX = new Fugitive(nodes[xPos]);
                MrX.setBlackTickets(BLACK_TICKETS);
                shortestDistance = new int[nodes.length][nodes.length];
                for (int i = 0; i < nodes.length; i++)
                        for (int j = 0; j < nodes.length; j++)
                                shortestDistance[i][j] = weight(nodes[i], nodes[j]);
                shortestDistance = getShortestDistanceMatrix(shortestDistance, 1);
        }

        /**
         * This constructor make a copy of the TestBoard board
         *
         * @param board
         *            the TestBoard whose copy has to be made
         */
        private TestBoard(TestBoard board) {
                this.currentMoves = board.currentMoves;
                detectives = new Detective[board.detectives.length];
                for (int i = 0; i < detectives.length; i++) {
                        detectives[i] = new Detective(board.detectives[i]);
                }
                Node n = board.MrX.getPosition();
                MrX = new Fugitive(n);
        }

        /**
         * This method changes the difficulty level for the game
         *
         * @param d
         *            the required difficulty level
         */
        public void setDepth(int d) {
                DEPTH = d;
        }

        /**
         * This method reads the text file which contains the map
         */
        private void readFile() {
                String fileName = "./SCOTMAP.TXT";
                try {
                        File f = new File(fileName);
                        if (!f.exists())
                                throw new IOException();
                        RandomAccessFile map = new RandomAccessFile(f, "r");
                        String buffer = map.readLine();
                        StringTokenizer token;
                        token = new StringTokenizer(buffer);
                        noOfNodes = Integer.parseInt(token.nextToken());
                        nodes = new Node[noOfNodes];
                        for (int i = 0; i < nodes.length; i++)
                                nodes[i] = new Node(i);
                        int lks = Integer.parseInt(token.nextToken());
                        for (int i = 0; i < lks; i++) {
                                buffer = map.readLine();
                                token = new StringTokenizer(buffer);
                                int node1 = Integer.parseInt(token.nextToken());
                                int node2 = Integer.parseInt(token.nextToken());
                                String strType = token.nextToken();
                                int type = INF;
                                if (strType.equals("T"))
                                        type = TAXI;
                                if (strType.equals("B"))
                                        type = BUS;
                                if (strType.equals("U"))
                                        type = UG;
                                if (strType.equals("F"))
                                        type = FERRY;
                                nodes[node1].addLink(nodes[node2], type);
                                nodes[node2].addLink(nodes[node1], type);
                        }
                } catch (Exception e) {
                         JOptionPane.showMessageDialog(null, I18n.tr("ErrorFileNotFound", fileName),
                                        I18n.tr("ErrorTitle"), JOptionPane.ERROR_MESSAGE);
                        System.exit(1);
```

```
                }
        }

        /**
         * This method reads the text file which contains the positions of the nodes
         * on the map
         */
        private void readPosFile() {
       String fileName = "./SCOTPOS.TXT";
                try {
                        File f = new File(fileName);
                        if (!f.exists())
                                throw new IOException();
                        RandomAccessFile map = new RandomAccessFile(f, "r");
                        String buffer = map.readLine();
                        StringTokenizer token = new StringTokenizer(buffer);
                        noOfNodes = Integer.parseInt(token.nextToken());
                        nodePositions = new Point[noOfNodes];

                        for (int i = 0; i < noOfNodes; i++) {
                                buffer = map.readLine();
                                token = new StringTokenizer(buffer);
                                int node = Integer.parseInt(token.nextToken());
                                int x = Integer.parseInt(token.nextToken());
                                int y = Integer.parseInt(token.nextToken());

                                nodePositions[node] = new Point(x, y);
                        }
                } catch (Exception e) {
                        JOptionPane.showMessageDialog(null, I18n.tr("ErrorFileNotFound", fileName),
                                        I18n.tr("ErrorTitle"), JOptionPane.ERROR_MESSAGE);
                        System.exit(1);
                }
        }

        /**
         * This method prints the board i.e what are the nodes what are the links.
         * It is useful for testing if the map which has been entered is correct or
         * not
         */
        public void printBoard() {
                for (int i = 0; i < nodes.length; i++) {
                        System.out.print("\nNode:" + nodes[i].getPosition() + "\t");
                        Link[] links = nodes[i].getLinks();
                        for (int j = 0; j < links.length; j++) {
                                Node to = links[j].getToNode();
                                System.out.print("\t" + to.getPosition());
                                System.out.println("\t" + links[j].getType());
                        }
                }
        }

        /**
         * This method returns the distance between the two nodes
         *
         * @param x
         *            ,y the two nodes given
         * @return 0 if x and y are the same nodes, 100 if they are not adjacent
         *         nodes, 50 if they are part of a ferry route, 1 if the two nodes
         *         are connected by any other transport mode.
         */
        public int weight(Node x, Node y) {
                if (x.equals(y))
                        return 0;
                int weight = INF;
                Link[] lk = x.getLinks();
                for (int i = 0; i < lk.length; i++) {
                        Node n = lk[i].getToNode();
                        if (n.equals(y))
                                weight = lk[i].getType();
                }
                if (weight != INF && weight != FERRY)
                        weight = 1;
                return weight;
        }
```

```
/**
 * This method evaluates the shortest distance between all the possible
 * nodes. It uses the Floyd-Warshall's Algorithm
 */
private int[][] getShortestDistanceMatrix(int[][] mat, int k) {
        if (k == nodes.length - 1)
                return mat;
        int newMat[][] = new int[nodes.length][nodes.length];
        {
                for (int i = 0; i < nodes.length; i++)
                        for (int j = 0; j < nodes.length; j++)
                                newMat[i][j] = Math.min(mat[i][j], mat[i][k] + mat[k][j]);
        }
        k = k + 1;
        return getShortestDistanceMatrix(newMat, k);
}

/**
 * Prints the shortest distance matrix
 */
public void test() {
        for (int i = 0; i < nodes.length; i++)
                for (int j = 0; j < nodes.length; j++)
                        System.out.println("Weight function for " + i + " " + j + " = "
                                        + shortestDistance[i][j]);
}

public static int getNumberOfDetectives() {
        return TestBoard.NO_OF_DETECTIVES;
}

/**
 * Checks if the move to along the link l is legal or not for the fugitive
 *
 * @param l
 *            the link to be tested
 *@return true if the move is legal otherwise it returns false
 */
private boolean isLegalMove(Link l) {
        if (l.getType() == FERRY && MrX.getBlackTickets() <= 0)
                return false;
        boolean canMove = true;
        for (int i = 0; i < detectives.length; i++) {
                Node n = detectives[i].getPosition();
                if (n.getPosition() == l.getToNode().getPosition())
                        canMove = false;
        }
        return canMove;
}

/**
 * This method returns all the possible moves for a given detective
 *
 * @param i
 *            the detective index of the detective whose possible moves we
 *            want
 *@return the possible moves of this detective in a TreeSet. If this
 *                          detective cannot move, it returns null.
 */
public TreeSet<Move> getDetectivePossibleMoves(int i) {
        return detectives[i].getPossibleMoves(this);
}



/**
 * This method is used to change the position of a detective
 *
 * @param i
 *            the index of the detective whose position we want to change
 *@param move
 *            the new move for this detective
 */
public void changeDetectivePosition(int i, Move move) {
        detectives[i].changePosition(nodes[move.getNode()], move.getType());
}
```

```
	/**
	 * Checks whether the machine has won
	 *
	 * @return true if machine has won,otherwise false
	 */
	public boolean isMachineWin() {
		boolean noneCanMove = true;
		for (int i = 0; i < NO_OF_DETECTIVES; i++)
			if (detectives[i].canMove(this))
				noneCanMove = false;
		return ((currentMoves == NO_OF_MOVES) || noneCanMove);
	}

	/**
	 * Checks if the user has won
	 *
	 * @return true if the user has won, otherwise false
	 */
	public boolean isUserWin() {
		Link[] xLinks = MrX.getPosition().getLinks();
		boolean isBlocked = true;
		for (int i = 0; i < xLinks.length; i++) {
			Node xNode = xLinks[i].getToNode();
			boolean thisIsOccupied = false;
			for (int j = 0; j < NO_OF_DETECTIVES; j++) {
				Node dNode = detectives[j].getPosition();
				if (dNode.equals(xNode))
					thisIsOccupied = true;
			}
			if (!thisIsOccupied)
				isBlocked = false;
		}
		boolean isCaptured = false;
		for (int i = 0; i < NO_OF_DETECTIVES; i++) {
			Node detNode = detectives[i].getPosition();
			if (detNode.equals(MrX.getPosition()))
				isCaptured = true;
		}
		return (isBlocked || isCaptured);
	}

	/**
	 * This method makes a random move for the MrX
	 *
	 * @return a random legal Node position for the MrX
	 */
	@SuppressWarnings("unused")
	private Node randomMove() {
		Node n = MrX.getPosition();
		Node toNode = n;
		Link[] lk = n.getLinks();
		boolean done = false;
		while (!done) {
			int rnd = (int) (lk.length * Math.random());
			if (isLegalMove(lk[rnd]))
				done = true;
		}
		return toNode;
	}

	/**
	 * This is the most important method of this class. It returns the best
	 * possible move by calling the evaluate() method.
	 *
	 * @return the best node position of MrX
	 */
	private Node bestMove(AtomicBoolean useBlackTicket) {
		Node n = MrX.getPosition();
		Link[] lk = n.getLinks();
		int score[] = new int[lk.length];
		Node possibleNodes[] = new Node[lk.length];
		int legalMoves = 0;
		int beta = LOSE;
		for (int i = 0; i < lk.length; i++) {
			if (isLegalMove(lk[i])) {
```

```
                                        if (legalMoves > 0 && beta < score[legalMoves - 1])
                                                beta = score[legalMoves - 1];
                                        TestBoard board = new TestBoard(this);
                                        board.MrX.change(lk[i].getToNode());
                                        possibleNodes[legalMoves] = lk[i].getToNode();
                                        score[legalMoves] = evaluateMove(board, false, 0, beta);
                                        legalMoves++;
                                }
                        }
                        Node toNode = possibleNodes[0];
                        int max = score[0];
                        //System.out.println("Scores at Node " + n.getPosition());
                        for (int i = 0; i < legalMoves; i++) {
                                //System.out.println("Position " + possibleNodes[i].getPosition() + " Score " +
score[i]);
                                if (max < score[i]) {
                                        toNode = possibleNodes[i];
                                        max = score[i];
                                }
                        }
                        //System.out.println("");

                        useBlackTicket.set(isBlackTicketUsable(n, possibleNodes, legalMoves, score));
                        return toNode;
                }

                /**
                 * Prints all the possible detective moves for the current board positions
                 * of the detectives, together with the corresponding scores for each new
                 * positions of the detectives.
                 */
                public void printPossibleDetectiveMoves() {
                        System.out.println("Generating detective moves:");
                        LinkedList<TestBoard> possibleMoves = new LinkedList<TestBoard>();
                        generateDetectiveMoves(this, possibleMoves, 0);
                        Object[] moves = possibleMoves.toArray();
                        Arrays.sort(moves);
                        int size = moves.length;
                        for (int count = 0; count < size && count <= NBEST; count++) {
                                TestBoard b = (TestBoard) moves[count];
                                for (int i = 0; i < NO_OF_DETECTIVES; i++)
                                        System.out.print(b.detectives[i].getPosition().getPosition() + " ");
                                System.out.println("Score " + scoreBoard(b) + "");
                        }
                        possibleMoves = null;
                        moves = null;
                        System.gc();
                }

                /**
                 * This method evaluates the position of the Node using depth first shallow
                 * search algorithm
                 *
                 * @param b
                 *            the initial TestBoard passed by the user
                 *@param depth
                 *            the current depth of the recursion tree (in this version,
                 *            depth is set to 0)
                 *@param isMachineMove
                 *            true if the next move is of the machine,else returns false
                 */
                private int evaluateMove(TestBoard b, boolean isMachineMove, int depth, int alphabeta) {
                        if (depth == DEPTH)
                                return scoreBoard(b);
                        else if (isMachineMove) {
                                Node dPos = b.MrX.getPosition();
                                Link[] lk = dPos.getLinks();
                                int score[] = new int[lk.length];
                                int legalMoves = 0;
                                int beta = LOSE;
                                for (int i = 0; i < lk.length; i++) {
                                        Node newPos = lk[i].getToNode();
                                        if (!b.isLegalMove(lk[i]))
                                                continue;

                                        if (legalMoves > 0 && beta < score[legalMoves - 1])
```

```
                                        beta = score[legalMoves - 1];
                                TestBoard board = new TestBoard(b);
                                board.MrX.change(newPos);
                                if (board.isUserWin())
                                        score[legalMoves] = LOSE;
                                else if (board.isMachineWin())
                                        score[legalMoves] = WIN;
                                else
                                        score[legalMoves] = evaluateMove(board, false, depth + 1,
beta);

                                // alpha cutoff here
                                if (score[legalMoves] > alphabeta) {
                                        return score[legalMoves];
                                }
                                legalMoves++;
                        }
                        int max = score[0];
                        for (int i = 0; i < legalMoves; i++)
                                if (score[i] > max)
                                        max = score[i];
                        return max;
                } else {
                        LinkedList<TestBoard> possibleMoves = new LinkedList<TestBoard>();
                        generateDetectiveMoves(b, possibleMoves, 0);
                        Object[] possibleMovesArray = possibleMoves.toArray();
                        int score[] = new int[possibleMovesArray.length];
                        int alpha = WIN;
                        for (int i = 0; i < score.length && i < NBEST; i++) {
                                if (i > 0 && alpha > score[i - 1])
                                        alpha = score[i - 1];
                                TestBoard board = (TestBoard) possibleMovesArray[i];
                                if (board.isUserWin())
                                        score[i] = LOSE;
                                else if (board.isMachineWin())
                                        score[i] = WIN;
                                else
                                        score[i] = evaluateMove(board, true, depth + 1, alpha);

                                // beta cutoff here
                                if (score[i] < alphabeta) {
                                        return score[i];
                                }
                        }
                        possibleMoves = null;
                        possibleMovesArray = null;
                        // Call the garbage collector to reclaim unused memory
                        System.gc();
                        int min = score[0];
                        for (int i = 0; i < score.length && i < NBEST; i++)
                                if (score[i] < min)
                                        min = score[i];
                        return min;
                }
        }

        /**
         * This method recursively generates all the possible detective moves, given
         * a board as the input.
         *
         * @param board
         *            The board from which to generate all the possible detective
         *            moves.
         * @param possibleMoves
         *            The linked list in which all the detective moves are stored.
         * @param detIndex
         *            the detective index - to be set to 0 when first called.
         */
        private void generateDetectiveMoves(TestBoard board, LinkedList<TestBoard> possibleMoves,
                        int detIndex) {
                if (!board.detectives[detIndex].canMove(board)) {
                        // keep the position of this detective the same
                        // i.e. do nothing
                        if (detIndex == NO_OF_DETECTIVES - 1)
                                possibleMoves.add(board);
                        else
```

```
                                        generateDetectiveMoves(board, possibleMoves, detIndex + 1);
                        } else {
                                TreeSet<Move> m = board.detectives[detIndex].getPossibleMoves(board);
                                int size = m.size();
                                for (int i = 0; i < size; i++) {
                                        Move l = m.pollFirst();
                                        TestBoard temp = new TestBoard(board);
                                        temp.detectives[detIndex].change(nodes[l.getNode()], l.getType());
                                        if (detIndex == NO_OF_DETECTIVES - 1) {
                                                possibleMoves.add(temp);
                                        } else
                                                generateDetectiveMoves(temp, possibleMoves, detIndex + 1);
                                }
                                m = null;
                        }
                }

                /**
                 * This method recursively generates all the possible detective moves, given
                 * initial board b This method is supposed to return a trimmed list of the
                 * possible detective moves: Only those moves are returned, which the human
                 * player will make logically.
                 */
                @SuppressWarnings("unused")
                private void generatePrunedDetectiveMoves(TestBoard board, LinkedList<TestBoard> possibleMoves,
                                int detIndex) {
                        if (!board.detectives[detIndex].canMove(board)) {
                                // keep the position of this detective the same
                                // i.e. do nothing
                                if (detIndex == NO_OF_DETECTIVES - 1)
                                        possibleMoves.add(board);
                                else
                                        generateDetectiveMoves(board, possibleMoves, detIndex + 1);
                        } else {
                                TreeSet<Move> m = board.detectives[detIndex].getPossibleMoves(board);
                                int size = m.size();
                                for (int i = 0; i < size; i++) {
                                        Move l = m.pollFirst();
                                        TestBoard temp = new TestBoard(board);
                                        temp.detectives[detIndex].change(nodes[l.getNode()], l.getType());
                                        if (detIndex == NO_OF_DETECTIVES - 1) {
                                                possibleMoves.add(temp);
                                        } else
                                                generatePrunedDetectiveMoves(temp, possibleMoves, detIndex +
1);
                                }
                                m = null;
                        }
                }

                /**
                 * This method checks if Mr. X has to reveal his position at this move or not.
                 *
                 * @return true if Mr. X has to reveal himself at this move, false
                 *         otherwise.
                 */
                public boolean isCheckPoint() {
                        boolean isCheckPoint = false;
                        for (int i = 0; i < checkPoints.length; i++)
                                if (currentMoves == checkPoints[i])
                                        isCheckPoint = true;
                        return isCheckPoint;
                }

                /**
                 * This method checks if Mr. X has to reveal his position at a given move or not
                 *
                 * @param move
                 *             the index of the move for which we want to check if it is a
                 *             checkpoint
                 * @return true if move is a checkpoint, false otherwise
                 */
                public boolean isCheckPoint(int move) {
                        boolean isCheckPoint = false;
                        for (int i = 0; i < checkPoints.length; i++)
                                if (move == checkPoints[i])
```

```
                                isCheckPoint = true;
                return isCheckPoint;
        }

        /**
         * This method is called to see if it makes sense to use a black ticket to
         * go from the source node to the target node for Mr. X. The black ticket
         * will NOT be used in the following circumstances: 1. Mr. X will reveal its
         * position in this move. 2. The from node has transportation of one type
         * only 3. There is only one possible move for Mr. X
         *
         * @param from
         *            the source node
         * @param to
         *            the target node
         * @return true if black ticket should be used for this move, false
         *         otherwise
         */
        private boolean isBlackTicketUsable(Node from, Node[] possibleNodes, int legalMoves, int[] score) {
                // First find all the valid moves from the source node
                // This is a code duplication, should be fixed later!
                Link[] links = from.getLinks();

                // If only one legal move from this move then obviously using
                // black ticket makes no sense
                if (legalMoves == 1)
                        return false;

                // If no black tickets, then obviously Mr. X cannot use them
                if (MrX.getBlackTickets() <= 0)
                        return false;

                // If Mr. X is going to reveal itself in this move or move after this,
                // then no need of using black ticket
                if (isCheckPoint(currentMoves + 1) || isCheckPoint(currentMoves + 2))
                        return false;

                // check if there are multiple destinations in possibleNodes. If not,
                // then don't use the black ticket.
                int pos1 = possibleNodes[0].getPosition();
                boolean multi = false;
                for (int i = 1; i < legalMoves; i++)
                        if (possibleNodes[i].getPosition() != pos1)
                                multi = true;
                if (!multi)
                        return false;

                // check if there are multiple transports available from the source.
                // If not, then don't use the black ticket.
                int taxiLink = 0;
                int busLink = 0;
                int ugLink = 0;
                int ferryLink = 0;
                for (int i = 0; i < links.length; i++) {
                        boolean canVisit = false;
                        for (int j = 0; j < legalMoves; j++)
                                if (possibleNodes[j] == links[i].getToNode() && score[j] > 0)
                                        canVisit = true;
                        if (canVisit && links[i].getType() == TAXI)
                                taxiLink = 1;
                        if (canVisit && links[i].getType() == BUS)
                                busLink = 1;
                        if (canVisit && links[i].getType() == UG)
                                ugLink = 1;
                        if (canVisit && links[i].getType() == FERRY)
                                ferryLink = 1;
                }
                if (taxiLink + busLink + ugLink + ferryLink <= 1)
                        return false;

                // Don't use consecutive black tickets
                if (MrX.prevPositions.getLast().getType() == BLACK)
                        return false;

                // OK, the false rules end here. Now the true rules: when to use black
                // tickets
```

```
                // if revealed in this move or the previous one, use black ticket
                if (isCheckPoint() || isCheckPoint(currentMoves - 1))
                        return true;

                return false;
        }

        /**
         * This method evaluates the given board
         *
         * @param board
         *            the given board
         *@return the score for this board
         */
        private static int scoreBoard(TestBoard board) {
                Detective[] det = board.detectives;
                Fugitive mrx = board.MrX;
                int minDistance = INF;
                int totalMobility = 0;
                for (int count = 0; count < NO_OF_DETECTIVES; count++) {
                        int distance = shortestDistance[det[count].getPosition().getPosition()][mrx
                                        .getPosition().getPosition()];
                        if (distance < minDistance)
                                minDistance = distance;
                        // totalMobility -= (det[count].mobility() / 3);
                }

                Node n = board.MrX.getPosition();
                Link[] lk = n.getLinks();
                for (int i = 0; i < lk.length; i++)
                        if (board.isLegalMove(lk[i]))
                                totalMobility++;

                int score = 20 * minDistance + totalMobility;
                return score;
        }

        /**
         * This method compares two objects of this class depending on the score of
         * the boards
         *
         * @param b1
         *            the first board
         *@param b2
         *            the second board
         *@return negative if score of b1 less then score of b2 positive otherwise
         */
        public int compare(Object b1, Object b2) {
                TestBoard board1 = (TestBoard) b1;
                TestBoard board2 = (TestBoard) b2;
                Detective[] det1 = board1.detectives;
                Detective[] det2 = board2.detectives;

                int[] d1 = new int[NO_OF_DETECTIVES];
                int[] d2 = new int[NO_OF_DETECTIVES];
                for (int i = 0; i < NO_OF_DETECTIVES; i++) {
                        d1[i] =
shortestDistance[det1[i].getPosition().getPosition()][board1.MrX.getPosition()
                                        .getPosition()];
                        d2[i] =
shortestDistance[det2[i].getPosition().getPosition()][board2.MrX.getPosition()
                                        .getPosition()];
                }
                Arrays.sort(d1);
                Arrays.sort(d2);
                int count = 0;
                while ((count < NO_OF_DETECTIVES) && (d1[count] == d2[count]))
                        count++;

                if (count >= NO_OF_DETECTIVES)
                        return 0;
                else
                        return d1[count] - d2[count];
        }
```

```java
/**
 * This method checks whether two boards are equal
 *
 * @param b1
 *            the first board
 *@param b2
 *            the second board
 *@return true if the boards are equal,false otherwise
 */
public boolean equal(TestBoard b1, TestBoard b2) {
        return (compare(b1, b2) == 0);
}

/**
 * This method compares this board to another board o
 *
 * @param o
 *            the board with which this is to be compared
 *@return similar to the compare() method
 */
public int compareTo(Object o) {
        TestBoard b = (TestBoard) o;
        return compare(this, b);
}

/**
 * This method computes a move for Mr. X and returns that move.
 *
 * @return The move computed for Mr. X.
 */
public Move moveMrX() {
        AtomicBoolean useBlackTicket = new AtomicBoolean();
        Node bestNode = bestMove(useBlackTicket);
        int type = MrX.changePosition(bestNode);
        int pos = MrX.getPosition().getPosition();
        if (useBlackTicket.get()) {
                MrX.useBlackTicket();
                type = BLACK;
        }
        currentMoves++;
        return (new Move(pos, type));
}

/**
 * This method is used to get the detectives of this board.
 *
 * @return the array containing the detectives of the current game.
 */
public Detective[] getDetectives() {
        return detectives;
}

/**
 * This method is used to get the MrX of this object.
 *
 * @return the MrX of this object.
 */
public Fugitive getMrX() {
        return MrX;
}

/**
 * This method returns the currentMoves of this object.
 *
 * @return the currentMoves of this object
 */
public int getCurrentMoves() {
        return currentMoves;
}

/**
 * String representation of this board.
 *
 * @return the score of this board in String form.
 */
public String toString() {
```

```
                return "" + scoreBoard(this);
        }

        /**
         * Returns the pixel coordinates of the specified node on the map.
         */
        public Point getPoint(int nodeIndex) {
                return nodePositions[nodeIndex];
        }
```

Transport.java

```java
package game;

/**
 * This interface defines constants associated with the various modes of
 * transport
 *
 * @author Shashi Mittal
 * @version 2.4 (19-APR-2010)
 */
public interface Transport {
          int NONE = 0;
          int TAXI = 1;
          int BUS = 2;
          int UG = 3;
          int FERRY = 50;
          int BLACK = 60;
          int INF = 100;
}
```

Appendix B. The Maximum Flow Minimum Cost Code

```
#include <iostream>
#include "mcmf.h"
#include <string.h>
#include <fstream>
#include <string>
#include <array>

using namespace std;

////////////////////////////////////////////////////////////////////////////
//
// MCMF_CS2
//
////////////////////////////////////////////////////////////////////////////

#define WHITE 0
#define GREY  1
#define BLACK 2
#define OPEN( a ) ( a->rez_capacity() > 0 )
#define CLOSED( a ) ( a->rez_capacity() <= 0 )
#define REDUCED_COST( i, j, a ) ( i->price() + a->cost() - j->price() )
#define FEASIBLE( i, j, a ) ( i->price() + a->cost() < j->price() )
#define ADMISSIBLE( i, j, a ) ( OPEN( a ) && FEASIBLE( i, j, a ) )
#define SUSPENDED( i, a ) ( a < i->first() )


#define REMOVE_FROM_EXCESS_Q( i )                                       \
            {
                        i           = _excq_first;                                          \
                        _excq_first = i -> q_next();                                \
                        i ->set_q_next( _sentinel_node );               \
            }

#define STACKQ_POP( i )                                                                     \
            {
                        i           = _excq_first;                                          \
                        _excq_first = i -> q_next();                                \
                        i ->set_q_next( _sentinel_node );               \
            }

#define GET_FROM_BUCKET( i, b )                                                     \
            {
                        i    = ( b -> p_first() );                                  \
                        b ->set_p_first( i -> b_next() );               \
            }
#define REMOVE_FROM_BUCKET( i, b )
            {
                        if ( i == ( b -> p_first() ) )
                                    b ->set_p_first( i -> b_next() );
                        else
                                    {
                                                ( i -> b_prev() )->set_b_next( i -> b_next() ); \
                                                ( i -> b_next() )->set_b_prev( i -> b_prev() ); \
                                    }
            }


void MCMF_CS2::err_end( int cc)
{
            // abnormal finish
            printf ("\nError %d\n", cc );
            // 2 - problem is unfeasible
            // 5 - allocation fault
            // 6 - price overflow
            exit( cc);
}

void MCMF_CS2::allocate_arrays()
{
            // (1) allocate memory for 'nodes', 'arcs' and internal arrays;

            _nodes = (NODE*) calloc ( _n+2,   sizeof(NODE) );
            _arcs = (ARC*)  calloc ( 2*_m+1, sizeof(ARC) );
```

```
            _cap = (long*) calloc ( 2*_m,   sizeof(long) );

            _arc_tail = (long*) calloc ( 2*_m,   sizeof(long) );
            _arc_first = (long*) calloc ( _n+2,   sizeof(long) );
            // arc_first [ 0 .. n+1 ] = 0 - initialized by calloc;

            for ( NODE *in = _nodes; in <= _nodes + _n; in ++ ) {
                        in->set_excess( 0);
            }
            if ( _nodes == NULL || _arcs == NULL || _arc_first == NULL || _arc_tail == NULL) {
                        printf("Error:  Memory allocation problem inside CS2\n");
                        exit( 1);
            }

            // (2) resets;
            _pos_current = 0;
            _arc_current = _arcs; // set "current" pointer to the first arc
            _node_max = 0;
            _node_min = _n;
            _max_cost = 0;
            _total_p = _total_n = 0;
            // at this moment we are ready to add arcs and build the network,
            // by using set_arc()...
}

void MCMF_CS2::deallocate_arrays()
{
            if ( _arcs) free ( _arcs );
            if ( _dnode) delete _dnode;
            if ( _cap) free ( _cap );
            if ( _buckets) free ( _buckets );
            if ( _check_solution == true) free ( _node_balance );
            if ( _nodes) {
                        _nodes = _nodes - _node_min;
                        free ( _nodes );
            }
}

void MCMF_CS2::set_arc( long tail_node_id, long head_node_id,

            long low_bound, long up_bound, // up_bound is basically capacity;
                                                                        price_t cost)
{
            // DIMACS format:
            // c arc has <tail> <head> <capacity l.b.> <capacity u.b> <cost>

            if ( tail_node_id < 0 || tail_node_id > _n ||
                         head_node_id < 0 || head_node_id > _n ) {
                        printf("Error:  Arc with head or tail out of bounds inside CS2\n");
                        exit( 1);
            }
            if ( up_bound < 0 ) {
                        up_bound = MAX_32;
                        printf("Warning:  Infinite capacity replaced by BIGGEST_FLOW\n");
            }
            if ( low_bound < 0 || low_bound > up_bound ) {
                        printf("Error:  Wrong capacity bounds inside CS2\n");
                        exit( 1);
            }

            // no of arcs incident to node i is placed in _arc_first[i+1]
            _arc_first[tail_node_id + 1] ++;
            _arc_first[head_node_id + 1] ++;
            _i_node = _nodes + tail_node_id;
            _j_node = _nodes + head_node_id;

            // store information about the arc
            _arc_tail[_pos_current]   = tail_node_id;
            _arc_tail[_pos_current+1] = head_node_id;
            _arc_current->set_head( _j_node );
            _arc_current->set_rez_capacity( up_bound - low_bound );
            _cap[_pos_current] = up_bound;
            _arc_current->set_cost( cost );
            _arc_current->set_sister( _arc_current + 1 );
            ( _arc_current + 1 )->set_head( _nodes + tail_node_id );
            ( _arc_current + 1 )->set_rez_capacity( 0 );
```

```
            _cap[_pos_current+1] = 0;
            ( _arc_current + 1 )->set_cost( -cost );
            ( _arc_current + 1 )->set_sister( _arc_current );

            _i_node->dec_excess( low_bound );
            _j_node->inc_excess( low_bound );

            // searching for minimum and maximum node
            if ( head_node_id < _node_min ) _node_min = head_node_id;
            if ( tail_node_id < _node_min ) _node_min = tail_node_id;
            if ( head_node_id > _node_max ) _node_max = head_node_id;
            if ( tail_node_id > _node_max ) _node_max = tail_node_id;

            if ( cost < 0 ) cost = -cost;
            if ( cost > _max_cost && up_bound > 0 ) _max_cost = cost;

            // prepare for next arc to be added;
            _arc_current += 2;
            _pos_current += 2;
}

void MCMF_CS2::set_supply_demand_of_node( long id, long excess)
{
            // set supply and demand of nodes; not used for transhipment nodes;
            if ( id < 0 || id > _n ) {
                        printf("Error:  Unbalanced problem inside CS2\n");
                        exit( 1);
            }
            (_nodes + id)->set_excess( excess);
            if ( excess > 0) _total_p += excess;
            if ( excess < 0) _total_n -= excess;
}

void MCMF_CS2::pre_processing()
{
            // called after the arcs were just added and before run_cs2();
            // ordering arcs - linear time algorithm;
            long i;
            long last, arc_num, arc_new_num;;
            long tail_node_id;
            NODE *head_p;
            ARC *arc_new, *arc_tmp;
            long up_bound;
            price_t cost; // arc cost;
            excess_t cap_out; // sum of outgoing capacities
            excess_t cap_in; // sum of incoming capacities

            if ( ABS( _total_p - _total_n ) > 0.5 ) {
                        printf("Error:  Unbalanced problem inside CS2\n");
                        exit( 1);
            }

            // first arc from the first node
            ( _nodes + _node_min )->set_first( _arcs );

            // before below loop arc_first[i+1] is the number of arcs outgoing from i;
            // after this loop arc_first[i] is the position of the first
            // outgoing from node i arcs after they would be ordered;
            // this value is transformed to pointer and written to node.first[i]
            for ( i = _node_min + 1; i <= _node_max + 1; i ++ ) {
                        _arc_first[i] += _arc_first[i-1];
                        ( _nodes + i )->set_first( _arcs + _arc_first[i] );
            }

            // scanning all the nodes except the last
            for ( i = _node_min; i < _node_max; i ++ ) {

                        last = ( ( _nodes + i + 1 )->first() ) - _arcs;
                        // arcs outgoing from i must be cited
                        // from position arc_first[i] to the position
                        // equal to initial value of arc_first[i+1]-1

                        for ( arc_num = _arc_first[i]; arc_num < last; arc_num ++ ) {
                                    tail_node_id = _arc_tail[arc_num];

                                    while ( tail_node_id != i ) {
```

```
            // the arc no  arc_num  is not in place because arc cited here
                                                // must go out from i;
                                                // we'll put it to its place and continue this process
                                                // until an arc in this position would go out from i

                                                arc_new_num = _arc_first[tail_node_id];
                                                _arc_current = _arcs + arc_num;
                                                arc_new = _arcs + arc_new_num;

                                                // arc_current must be cited in the position arc_new
                                                // swapping these arcs:

                                                head_p = arc_new->head();
                                                arc_new->set_head( _arc_current->head() );
                                                _arc_current->set_head( head_p );

                                                up_bound          = _cap[arc_new_num];
                                                _cap[arc_new_num] = _cap[arc_num];
                                                _cap[arc_num]     = up_bound;

                                                up_bound = arc_new->rez_capacity();
                                                arc_new->set_rez_capacity( _arc_current->rez_capacity() );
                                                _arc_current->set_rez_capacity( up_bound) ;

                                                cost = arc_new->cost();
                                                arc_new->set_cost( _arc_current->cost() );
                                                _arc_current->set_cost( cost );

                                                if ( arc_new != _arc_current->sister() ) {
                                                            arc_tmp = arc_new->sister();
                                                            arc_new->set_sister( _arc_current->sister() );
                                                            _arc_current->set_sister( arc_tmp );

                                                            _arc_current->sister()-
>set_sister( _arc_current );
                                                            arc_new->sister()->set_sister( arc_new );
                                                }

                                                _arc_tail[arc_num] = _arc_tail[arc_new_num];
                                                _arc_tail[arc_new_num] = tail_node_id;

                                                // we increase arc_first[tail_node_id]
                                                _arc_first[tail_node_id] ++ ;

                                                tail_node_id = _arc_tail[arc_num];
                                    }
                        }
                        // all arcs outgoing from  i  are in place
            }
            // arcs are ordered by now!


            // testing network for possible excess overflow
            for ( NODE *ndp = _nodes + _node_min; ndp <= _nodes + _node_max; ndp ++ ) {
                        cap_in  =   ( ndp->excess() );
                        cap_out = - ( ndp->excess() );
                        for ( _arc_current = ndp->first(); _arc_current != (ndp+1)->first();
                                                _arc_current ++ ) {
                                    arc_num = _arc_current - _arcs;
                                    if ( _cap[arc_num] > 0 ) cap_out += _cap[arc_num];
                                    if ( _cap[arc_num] == 0 )
                                                cap_in += _cap[ _arc_current->sister() - _arcs ];
                        }
            }
            if ( _node_min < 0 || _node_min > 1 ) {
                        printf("Error:  Node ids must start from 0 or 1 inside CS2\n");
                        exit( 1);
            }

            // adjustments due to nodes' ids being between _node_min - _node_max;
            _n = _node_max - _node_min + 1;
            _nodes = _nodes + _node_min;

            // () free internal memory, not needed anymore inside CS2;
            free ( _arc_first );
```

```
            free ( _arc_tail );
}

void MCMF_CS2::cs2_initialize()
{
            // initialization;
            // called after allocate_arrays() and all nodes and arcs have been inputed;

            NODE *i; // current node
            ARC *a; // current arc
            ARC *a_stop;
            BUCKET *b; // current bucket
            long df;

            _f_scale = (long) SCALE_DEFAULT;
            _sentinel_node = _nodes + _n;
            _sentinel_arc  = _arcs + _m;

            for ( i = _nodes; i != _sentinel_node; i ++ ) {
                        i->set_price( 0);
                        i->set_suspended( i->first());
                        i->set_q_next( _sentinel_node);
            }

            _sentinel_node->set_first( _sentinel_arc);
            _sentinel_node->set_suspended( _sentinel_arc);

            // saturate negative arcs, e.g. in the circulation problem case
            for ( i = _nodes; i != _sentinel_node; i ++ ) {
                        for ( a = i->first(), a_stop = (i + 1)->suspended(); a != a_stop; a ++ ) {
                                    if ( a->cost() < 0) {
                                                if ( ( df = a->rez_capacity()) > 0) {
                                                            increase_flow( i, a->head(), a, df);
                                                }
                                    }
                        }
            }

            _dn = _n + 1;
            if ( _no_zero_cycles == true) { // NO_ZERO_CYCLES
                        _dn = 2 * _dn;
            }

            for ( a = _arcs; a != _sentinel_arc; a ++ ) {
                        a->multiply_cost( _dn);
            }

            if ( _no_zero_cycles == true) { // NO_ZERO_CYCLES
                        for ( a = _arcs; a != _sentinel_arc; a ++ ) {
                                    if ((a->cost() == 0) && (a->sister()->cost() == 0)) {
                                                a->set_cost( 1);
                                                a->sister()->set_cost( -1);
                                    }
                        }
            }

            if ((double) _max_cost * (double) _dn > MAX_64) {
                        printf("Warning:  Arc lengths too large, overflow possible\n");
            }
            _mmc = _max_cost * _dn;

            _linf = (long) (_dn * ceil(_f_scale) + 2);

            _buckets = (BUCKET*) calloc ( _linf, sizeof(BUCKET));
            if ( _buckets == NULL )
                        err_end( ALLOCATION_FAULT);

            _l_bucket = _buckets + _linf;

            _dnode = new NODE; // used as reference;

            for ( b = _buckets; b != _l_bucket; b ++ ) {
                        reset_bucket( b);
            }

            _epsilon = _mmc;
```

```
            if ( _epsilon < 1) {
                        _epsilon = 1;
            }

            _price_min = -PRICE_MAX;

            _cut_off_factor = CUT_OFF_COEF * pow( (double)_n, CUT_OFF_POWER);

            _cut_off_factor = MAX( _cut_off_factor, CUT_OFF_MIN);

            _n_ref = 0;

            _flag_price = 0;

            _dummy_node = &_d_node;

            _excq_first = NULL;

            //print_graph(); // debug;
}

void MCMF_CS2::up_node_scan( NODE *i)
{
            NODE *j; // opposite node
            ARC *a; // (i, j)
            ARC *a_stop; // first arc from the next node
            ARC *ra; // (j, i)
            BUCKET *b_old; // old bucket contained j
            BUCKET *b_new; // new bucket for j
            long i_rank;
            long j_rank; // ranks of nodes
            long j_new_rank;
            price_t rc; // reduced cost of (j, i)
            price_t dr; // rank difference

            _n_scan ++;

            i_rank = i->rank();

            // scanning arcs;
            for ( a = i->first(), a_stop = (i + 1)->suspended(); a != a_stop; a ++ ) {

                        ra = a->sister();

                        if ( OPEN ( ra ) ) {
                                    j = a->head();
                                    j_rank = j->rank();

                                    if ( j_rank > i_rank ) {
                                                if ( ( rc = REDUCED_COST( j, i, ra ) ) < 0 ) {
                                                            j_new_rank = i_rank;
                                                } else {
                                                            dr = rc / _epsilon;

            j_new_rank = ( dr < _linf ) ? i_rank + (long)dr + 1 : _linf;
                                                }

                                                if ( j_rank > j_new_rank ) {
                                                            j->set_rank( j_new_rank);
                                                            j->set_current( ra);

                                                            if ( j_rank < _linf ) {
                                                                        b_old = _buckets + j_rank;
                                                                        REMOVE_FROM_BUCKET( j, b_old );
                                                            }

                                                            b_new = _buckets + j_new_rank;
                                                            insert_to_bucket( j, b_new );
                                                }
                                    }
                        }
            }

            i->dec_price( i_rank * _epsilon);
            i->set_rank( -1);
}
```

```
void MCMF_CS2::price_update()
{
            register NODE *i;
            excess_t remain;
            // total excess of unscanned nodes with positive excess;
            BUCKET *b; // current bucket;
            price_t dp; // amount to be subtracted from prices;

            _n_update ++;

            for ( i = _nodes; i != _sentinel_node; i ++ ) {
                        if ( i->excess() < 0 ) {
                                    insert_to_bucket( i, _buckets );
                                    i->set_rank( 0);
                        } else {
                                    i->set_rank( _linf);
                        }
            }

            remain = _total_excess;
            if ( remain < 0.5 ) return;

            // scanning buckets, main loop;
            for ( b = _buckets; b != _l_bucket; b ++ ) {

                        while ( nonempty_bucket( b) ) {

                                    GET_FROM_BUCKET( i, b );
                                    up_node_scan( i );

                                    if ( i ->excess() > 0 ) {
                                                remain -= ( i->excess());
                                                if ( remain <= 0 ) break;
                                    }
                        }
                        if ( remain <= 0 ) break;
            }

            if ( remain > 0.5 ) _flag_updt = 1;

            // finishup
            // changing prices for nodes which were not scanned during main loop;
            dp = ( b - _buckets ) * _epsilon;

            for ( i = _nodes; i != _sentinel_node; i ++ ) {

                        if ( i->rank() >= 0 ) {
                                    if ( i->rank() < _linf ) {
                                                REMOVE_FROM_BUCKET( i, ( _buckets + i->rank()) );
                                    }
                                    if ( i->price() > _price_min ) {
                                                i->dec_price( dp);
                                    }
                        }
            }
}

int MCMF_CS2::relabel( NODE *i)
{
            register ARC *a; // current arc from i
            register ARC *a_stop; // first arc from the next node
            register ARC *a_max; // arc which provides maximum price
            register price_t p_max; // current maximal price
            register price_t i_price; // price of node  i
            register price_t dp; // current arc partial residual cost

            p_max = _price_min;
            i_price = i->price();

            a_max = NULL;

            // 1/2 arcs are scanned;
            for ( a = i->current() + 1, a_stop = (i + 1)->suspended(); a != a_stop; a ++ ) {

                        if ( OPEN(a) && ( (dp = (a->head()->price() - a->cost())) > p_max ) ) {
```

```
                                        if ( i_price < dp ) {
                                                    i->set_current( a);
                                                    return ( 1);
                                        }
                                        p_max = dp;
                                        a_max = a;
                                }
                    }

                    // 2/2 arcs are scanned;
                    for ( a = i->first(), a_stop = i->current() + 1; a != a_stop; a ++ ) {
                                if ( OPEN( a) && ( (dp = (a->head()->price() - a->cost())) > p_max ) ) {
                                        if ( i_price < dp ) {
                                                    i->set_current( a);
                                                    return ( 1);
                                        }
                                        p_max = dp;
                                        a_max = a;
                                }
                    }

                    // finishup
                    if ( p_max != _price_min ) {
                                i->set_price( p_max - _epsilon);
                                i->set_current( a_max);
                    }
                    else { // node can't be relabelled;
                                if ( i->suspended() == i->first() ) {
                                        if ( i->excess() == 0 ) {
                                                    i->set_price( _price_min);
                                        } else {
                                                    if ( _n_ref == 1 ) {
                                                                err_end( UNFEASIBLE );
                                                    } else {
                                                                err_end( PRICE_OFL );
                                                    }
                                        }
                                } else { // node can't be relabelled because of suspended arcs;
                                        _flag_price = 1;
                                }
                    }

                    _n_relabel ++;
                    _n_rel ++;
                    return ( 0);
}

void MCMF_CS2::discharge( NODE *i)
{
                    register ARC *a;// an arc from i
                    register NODE *j; // head of a
                    register long df; // amoumt of flow to be pushed through a
                    excess_t j_exc; // former excess of j

                    _n_discharge ++;

                    a = i->current();
                    j = a->head();

                    if ( !ADMISSIBLE( i, j, a ) ) {
                                relabel( i );
                                a = i->current();
                                j = a->head();
                    }

                    while ( 1 ) {

                                j_exc = j->excess();
                                if ( j_exc >= 0 ) {

                                        df = MIN( i->excess(), a->rez_capacity() );
                                        if ( j_exc == 0) _n_src++;
                                        increase_flow( i, j, a, df ); // INCREASE_FLOW
                                        _n_push ++;

                                        if ( out_of_excess_q( j ) ) {
```

```
                                        insert_to_excess_q( j );
                                }
                        }
                        else { // j_exc < 0;

                                df = MIN( i->excess(), a->rez_capacity() );
                                increase_flow( i, j, a, df ); // INCREASE_FLOW
                                _n_push ++;

                                if ( j->excess() >= 0 ) {
                                        if ( j->excess() > 0 ) {
                                                _n_src ++;
                                                relabel( j );
                                                insert_to_excess_q( j );
                                        }
                                        _total_excess += j_exc;
                                }
                                else {
                                        _total_excess -= df;
                                }
                        }

                        if ( i->excess() <= 0) _n_src --;
                        if ( i->excess() <= 0 || _flag_price ) break;

                        relabel( i );

                        a = i->current();
                        j = a->head();
                }

                i->set_current( a);
}

int MCMF_CS2::price_in()
{
                NODE *i; // current node
                NODE *j;
                ARC *a; // current arc from i
                ARC *a_stop; // first arc from the next node
                ARC *b; // arc to be exchanged with suspended
                ARC *ra; // opposite to a
                ARC *rb; // opposite to b
                price_t rc; // reduced cost
                int n_in_bad; // number of priced_in arcs with negative reduced cost
                int bad_found; // if 1 we are at the second scan if 0 we are at the first scan
                excess_t i_exc; // excess of i
                excess_t df; // an amount to increase flow


                bad_found = 0;
                n_in_bad = 0;

 restart:

                for ( i = _nodes; i != _sentinel_node; i ++ ) {

                        for ( a = i->first() - 1, a_stop = i->suspended() - 1; a != a_stop; a -- ) {

                                rc = REDUCED_COST( i, a->head(), a );
                                if ( ( rc < 0) && ( a->rez_capacity() > 0) ) { // bad case;
                                        if ( bad_found == 0 ) {
                                                bad_found = 1;
                                                update_cut_off();
                                                goto restart;
                                        }
                                        df = a->rez_capacity();
                                        increase_flow( i, a->head(), a, df );

                                        ra = a->sister();
                                        j  = a->head();

                                        i->dec_first();
                                        b = i->first();
                                        exchange( a, b );
```

```
                                        if ( SUSPENDED( j, ra ) ) {
                                                j->dec_first();
                                                rb = j->first();
                                                exchange( ra, rb );
                                        }

                                        n_in_bad ++;
                                }
                                else {
                                        if ( ( rc < _cut_on ) && ( rc > -_cut_on ) ) {
                                                i->dec_first();
                                                b = i->first();
                                                exchange( a, b );
                                        }
                                }
                        }
                }


                if ( n_in_bad != 0 ) {

                        _n_bad_pricein ++;

                        // recalculating excess queue;
                        _total_excess = 0;
                        _n_src = 0;
                        reset_excess_q();

                        for ( i = _nodes; i != _sentinel_node; i ++ ) {
                                i->set_current( i->first());
                                i_exc = i->excess();
                                if ( i_exc > 0 ) { // i is a source;
                                        _total_excess += i_exc;
                                        _n_src ++;
                                        insert_to_excess_q( i );
                                }
                        }

                        insert_to_excess_q( _dummy_node );
                }

                if ( _time_for_price_in == TIME_FOR_PRICE_IN2)
                        _time_for_price_in = TIME_FOR_PRICE_IN3;
                if ( _time_for_price_in == TIME_FOR_PRICE_IN1)
                        _time_for_price_in = TIME_FOR_PRICE_IN2;

                return ( n_in_bad);
}

void MCMF_CS2::refine()
{
                NODE *i; // current node
                excess_t i_exc; // excess of i
                long np, nr, ns; // variables for additional print
                int pr_in_int; // current number of updates between price_in

                np = _n_push;
                nr = _n_relabel;
                ns = _n_scan;

                _n_refine ++;
                _n_ref ++;
                _n_rel = 0;
                pr_in_int = 0;

                // initialize;
                _total_excess = 0;
                _n_src = 0;
                reset_excess_q();

                _time_for_price_in = TIME_FOR_PRICE_IN1;

                for ( i = _nodes; i != _sentinel_node; i ++ ) {
                        i->set_current( i->first());
                        i_exc = i->excess();
                        if ( i_exc > 0 ) { // i  is a source
```

```
                                    _total_excess += i_exc;
                                    _n_src++;
                                    insert_to_excess_q( i );
                          }
                }

                if ( _total_excess <= 0 ) return;

                // (2) main loop

                while ( 1 ) {

                          if ( empty_excess_q() ) {
                                    if ( _n_ref > PRICE_OUT_START ) {
                                              pr_in_int = 0;
                                              price_in();
                                    }

                                    if ( empty_excess_q() ) break;
                          }

                          REMOVE_FROM_EXCESS_Q( i );

                          // push all excess out of i
                          if ( i->excess() > 0 ) {
                                    discharge( i );

                                    if ( time_for_update() || _flag_price ) {
                                              if ( i->excess() > 0 ) {
                                                        insert_to_excess_q( i );
                                              }

                                              if ( _flag_price && ( _n_ref > PRICE_OUT_START ) ) {
                                                        pr_in_int = 0;
                                                        price_in();
                                                        _flag_price = 0;
                                              }

                                              price_update();

                                              while ( _flag_updt ) {
                                                        if ( _n_ref == 1 ) {
                                                                  err_end( UNFEASIBLE );
                                                        } else {
                                                                  _flag_updt = 0;
                                                                  update_cut_off();
                                                                  _n_bad_relabel ++;
                                                                  pr_in_int = 0;
                                                                  price_in();
                                                                  price_update();
                                                        }
                                              }
                                              _n_rel = 0;


                if ( _n_ref > PRICE_OUT_START && (pr_in_int ++ > _time_for_price_in) ) {
                                                        pr_in_int = 0;
                                                        price_in();
                                              }
                                    }
                          }
                }

                return;
}

int MCMF_CS2::price_refine()
{
                NODE *i; // current node
                NODE *j; // opposite node
                NODE *ir; // nodes for passing over the negative cycle
                NODE *is;
                ARC *a; // arc (i,j)
                ARC *a_stop; // first arc from the next node
                ARC *ar;
                long bmax;            // number of farest nonempty bucket
```

```
            long i_rank;          // rank of node i
            long j_rank;         // rank of node j
            long j_new_rank;      // new rank of node j
            BUCKET *b;              // current bucket
            BUCKET *b_old;          // old and new buckets of current node
            BUCKET *b_new;
            price_t rc = 0; // reduced cost of a
            price_t dr; // ranks difference
            price_t dp;
            int cc;
            // return code: 1 - flow is epsilon optimal
            // 0 - refine is needed
            long df; // cycle capacity
            int nnc; // number of negative cycles cancelled during one iteration
            int snc; // total number of negative cycle cancelled

            _n_prefine ++;

            cc = 1;
            snc = 0;

            _snc_max = ( _n_ref >= START_CYCLE_CANCEL) ? MAX_CYCLES_CANCELLED : 0;


            // (1) main loop
            // while negative cycle is found or eps-optimal solution is constructed
            while ( 1 ) {

                        nnc = 0;
                        for ( i = _nodes; i != _sentinel_node; i ++ ) {
                                    i->set_rank( 0);
                                    i->set_inp( WHITE);
                                    i->set_current( i->first());
                        }
                        reset_stackq();

                        for ( i = _nodes; i != _sentinel_node; i ++ ) {
                                    if ( i->inp() == BLACK ) continue;

                                    i->set_b_next( NULL);

                                    // deapth first search
                                    while ( 1 ) {
                                                i->set_inp( GREY);

                                                // scanning arcs from node i starting from current
                                                for ( a = i->current(), a_stop = (i + 1)-
>suspended(); a != a_stop; a ++) {
                                                            if ( OPEN( a ) ) {
                                                                        j = a->head();
                                                                        if ( REDUCED_COST ( i, j, a ) < 0 ) {
                                                                                    if ( j-
>inp() == WHITE ) { // fresh node  - step forward
                                                                                                i-
>set_current( a);
                                                                                                j-
>set_b_next( i);
                                                                                                i = j;
                                                                                                a = j-
>current();

            a_stop = (j+1)->suspended();
                                                                                                break;
                                                                                    }

                                                                                    if ( j-
>inp() == GREY ) { // cycle detected
                                                                                                cc = 0;
                                                                                                nnc ++;
                                                                                                i-
>set_current( a);
                                                                                                is = ir = i;
                                                                                                df = MAX_32;

                                                                                                while ( 1 ) {
```

```
            ar = ir->current();
            if ( ar->rez_capacity() <= df ) {
                                                                                    }
            if ( ir == j ) break;
            ir = ir->b_next();
                                                                        }
                                                                        ir = i;
                                                                        while ( 1 ) {
            ar = ir->current();
            increase_flow( ir, ar->head(), ar, df);
            if ( ir == j ) break;
            ir = ir->b_next();
                                                                        }

            if ( is != i ) {
            for ( ir = i; ir != is; ir = ir->b_next() ) {
                                                                                    }

            i = is;
            a = is->current() + 1;
            a_stop = (is+1)->suspended();
            break;
                                                                        }
                                                                    }
                                                                }
                                                                // if j-color is BLACK -
continue search from i
                                                        }
                                                } // all arcs from i are scanned

                                                if ( a == a_stop ) {
                                                        // step back
                                                        i->set_inp( BLACK);
                                                        _n_prscan1 ++;
                                                        j = i->b_next();
                                                        stackq_push( i );
                                                        if ( j == NULL ) break;
                                                        i = j;
                                                        i->inc_current();
                                                }

                                        } // end of deapth first search
                        } // all nodes are scanned


                        // () no negative cycle
                        // computing longest paths with eps-precision

                        snc += nnc;
                        if ( snc < _snc_max ) cc = 1;
                        if ( cc == 0 ) break;
                        bmax = 0;

                        while ( nonempty_stackq() ) {

                                        _n_prscan2 ++;
                                        STACKQ_POP( i );
                                        i_rank = i->rank();
                                        for ( a = i->first(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {
```

```
                                        if ( OPEN( a ) ) {
                                                j  = a->head();
                                                rc = REDUCED_COST( i, j, a );

                                                if ( rc < 0 ) { // admissible arc;
                                                        dr = (price_t) (( - rc -
 0.5 ) / _epsilon);

            if (( j_rank = dr + i_rank ) < _linf ) {
                                                                        if ( j_rank > j->rank() )
                                                                                j-
>set_rank( j_rank);
                                                        }
                                                }
                                        }
                                } // all arcs from i are scanned

                                if ( i_rank > 0 ) {
                                        if ( i_rank > bmax ) bmax = i_rank;
                                        b = _buckets + i_rank;
                                        insert_to_bucket( i, b );
                                }
                        } // end of while-cycle: all nodes are scanned - longest distancess are computed;


                        if ( bmax == 0 ) // preflow is eps-optimal;
                                { break; }


                        for ( b = _buckets + bmax; b != _buckets; b -- ) {
                                i_rank = b - _buckets;
                                dp = i_rank * _epsilon;

                                while ( nonempty_bucket( b) ) {
                                        GET_FROM_BUCKET( i, b );
                                        _n_prscan ++;

                                        for ( a = i->first(), a_stop = (i + 1)-
>suspended(); a != a_stop; a ++) {
                                                if ( OPEN( a ) ) {
                                                        j = a->head();
                                                        j_rank = j->rank();
                                                        if ( j_rank < i_rank ) {

            rc = REDUCED_COST( i, j, a );
                                                                        if ( rc < 0 ) {

            j_new_rank = i_rank;
                                                                        } else {

            dr = rc / _epsilon;

            j_new_rank = ( dr < _linf ) ? i_rank - ( (long)dr + 1 ) : 0;
                                                                        }

            if ( j_rank < j_new_rank ) {

            if ( cc == 1 ) {
                                                                                j-
>set_rank( j_new_rank);

            if ( j_rank > 0 ) {


                                                                                }

            b_new = _buckets + j_new_rank;

            insert_to_bucket( j, b_new );
                                                                                }
                                                                                else {

            df = a->rez_capacity();

            increase_flow( i, j, a, df );
```

```
                                                                                                        }
                                                                                                }
                                                                                        }
                                                                                } // end if opened arc
                                                                        } // all arcs are scanned

                                                                        i->dec_price( dp);

                                                                } // end of while-cycle: the bucket is scanned
                                                } // end of for-cycle: all buckets are scanned

                                                if ( cc == 0 ) break;

                                        } // end of main loop



                                        // (2) finish
                                        // if refine needed - saturate non-epsilon-optimal arcs;

                                        if ( cc == 0 ) {
                                                for ( i = _nodes; i != _sentinel_node; i ++) {
                                                        for ( a = i->first(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {
                                                                if ( REDUCED_COST( i, a->head(), a ) < - _epsilon ) {
                                                                        if ( ( df = a->rez_capacity() ) > 0 ) {
                                                                                increase_flow( i, a->head(), a, df );
                                                                        }
                                                                }
                                                        }
                                                }
                                        }

                                        return ( cc );
}

void MCMF_CS2::compute_prices()
{
                                        NODE *i; // current node
                                        NODE *j; // opposite node
                                        ARC *a; // arc (i,j)
                                        ARC *a_stop; // first arc from the next node
                                        long bmax; // number of farest nonempty bucket
                                        long i_rank; // rank of node i
                                        long j_rank; // rank of node j
                                        long j_new_rank; // new rank of node j
                                        BUCKET *b; // current bucket
                                        BUCKET *b_old; // old and new buckets of current node
                                        BUCKET *b_new;
                                        price_t rc; // reduced cost of a
                                        price_t dr; // ranks difference
                                        price_t dp;
                                        int cc; // return code: 1 - flow is epsilon optimal 0 - refine is needed

                                        _n_prefine ++;
                                        cc = 1;

                                        // (1) main loop
                                        // while negative cycle is found or eps-optimal solution is constructed
                                        while ( 1 ) {

                                                for ( i = _nodes; i != _sentinel_node; i ++) {
                                                        i->set_rank( 0);
                                                        i->set_inp( WHITE);
                                                        i->set_current( i->first());
                                                }
                                                reset_stackq();

                                                for ( i = _nodes; i != _sentinel_node; i ++ ) {
                                                        if ( i->inp() == BLACK ) continue;

                                                        i->set_b_next( NULL);
                                                        // depth first search
                                                        while ( 1 ) {
                                                                i->set_inp( GREY);

                                                                // scanning arcs from node i
```

```
                                        for ( a = i->suspended(), a_stop = (i + 1)-
>suspended(); a != a_stop; a ++) {
                                                if ( OPEN( a ) ) {
                                                        j = a->head();
                                                        if ( REDUCED_COST( i, j, a ) < 0 ) {
                                                                if ( j-
>inp() == WHITE ) { // fresh node  - step forward
                                                                        i-
>set_current( a);
                                                                        j-
>set_b_next( i);
                                                                        i = j;
                                                                        a = j-
>current();

            a_stop = (j+1)->suspended();
                                                                        break;
                                                                }

                                                                if ( j-
>inp() == GREY ) { // cycle detected; should not happen
                                                                        cc = 0;
                                                                }
                                                        }
                                                        // if j-color is BLACK -
 continue search from i
                                                }
                                        } // all arcs from i are scanned

                                        if ( a == a_stop ) {
                                                // step back
                                                i->set_inp( BLACK);
                                                _n_prscan1 ++;
                                                j = i->b_next();
                                                stackq_push( i );
                                                if ( j == NULL ) break;
                                                i = j;
                                                i->inc_current();
                                        }

                                } // end of deapth first search
                        } // all nodes are scanned


                        // no negative cycle
                        // computing longest paths

                        if ( cc == 0 ) break;
                        bmax = 0;

                        while ( nonempty_stackq() ) {
                                _n_prscan2 ++;
                                STACKQ_POP( i );
                                i_rank = i->rank();
                                for ( a = i->suspended(), a_stop = (i + 1)-
>suspended(); a != a_stop; a ++) {
                                        if ( OPEN( a ) ) {
                                                j  = a->head();
                                                rc = REDUCED_COST( i, j, a );


                                                if ( rc < 0 ) {// admissible arc
                                                        dr = - rc;

            if (( j_rank = dr + i_rank ) < _linf ) {
                                                                if ( j_rank > j->rank() )
                                                                        j-
>set_rank( j_rank);
                                                        }
                                                }
                                        }
                                } // all arcs from i are scanned

                                if ( i_rank > 0 ) {
                                        if ( i_rank > bmax ) bmax = i_rank;
                                        b = _buckets + i_rank;
```

```
                                        insert_to_bucket( i, b );
                                }
                        } // end of while-cycle: all nodes are scanned - longest distancess are computed;

                        if ( bmax == 0 )
                                { break; }

                        for ( b = _buckets + bmax; b != _buckets; b -- ) {
                                i_rank = b - _buckets;
                                dp = i_rank;

                                while ( nonempty_bucket( b) ) {
                                        GET_FROM_BUCKET( i, b );
                                        _n_prscan ++;

                                        for ( a = i->suspended(), a_stop = (i + 1)-
>suspended(); a != a_stop; a ++) {
                                                if ( OPEN( a ) ) {
                                                        j = a->head();
                                                        j_rank = j->rank();
                                                        if ( j_rank < i_rank ) {

            rc = REDUCED_COST( i, j, a );

                                                                        if ( rc < 0 ) {

            j_new_rank = i_rank;
                                                                        } else {
                                                                                dr = rc;

            j_new_rank = ( dr < _linf ) ? i_rank - ( (long)dr + 1 ) : 0;
                                                                        }

            if ( j_rank < j_new_rank ) {

            if ( cc == 1 ) {
                                                                                                j-
>set_rank( j_new_rank);

            if ( j_rank > 0 ) {


                                                                                                }

            b_new = _buckets + j_new_rank;

            insert_to_bucket( j, b_new );
                                                                                        }
                                                                        }
                                                                }
                                                        } // end if opened arc
                                        } // all arcs are scanned

                                        i->dec_price( dp);

                                } // end of while-cycle: the bucket is scanned
                        } // end of for-cycle: all buckets are scanned

                        if ( cc == 0 ) break;

            } // end of main loop
}

void MCMF_CS2::price_out()
{
            NODE *i; // current node
            ARC *a; // current arc from i
            ARC *a_stop; // first arc from the next node
            ARC *b; // arc to be exchanged with suspended
            double n_cut_off; // -cut_off
            double rc; // reduced cost

            n_cut_off = - _cut_off;

            for ( i = _nodes; i != _sentinel_node; i ++) {
                        for ( a = i->first(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {
```

```
                                    rc = REDUCED_COST( i, a->head(), a );
                                    if ( ( rc > _cut_off && CLOSED(a->sister()) ) ||
                                                ( rc < n_cut_off && CLOSED(a) ) ) { // suspend the arc

                                                b = i->first();
                                                i->inc_first();
                                                exchange( a, b );
                                    }
                        }
            }
}

int MCMF_CS2::update_epsilon()
{
            // decrease epsilon after epsilon-optimal flow is constructed;
            if ( _epsilon <= 1 ) return ( 1 );

            _epsilon = (price_t) (ceil ( (double) _epsilon / _f_scale ));
            _cut_off = _cut_off_factor * _epsilon;
            _cut_on = _cut_off * CUT_OFF_GAP;

            return ( 0 );
}

int MCMF_CS2::check_feas()
{
            if ( _check_solution == false)
                        return ( 0);

            NODE *i;
            ARC *a, *a_stop;
            long fa;
            int ans = 1;

            for ( i = _nodes; i != _sentinel_node; i ++) {
                        for ( a = i->suspended(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {
                                    if ( _cap[ N_ARC(a) ] > 0) {
                                                fa = _cap[ N_ARC(a) ] - a->rez_capacity();
                                                if ( fa < 0) {
                                                            ans = 0;
                                                            break;
                                                }
                                                _node_balance[ i - _nodes ] -= fa;
                                                _node_balance[ a->head() - _nodes ] += fa;
                                    }
                        }
            }

            for ( i = _nodes; i != _sentinel_node; i ++) {
                        if ( _node_balance[ i - _nodes ] != 0) {
                                    ans = 0;
                                    break;
                        }
            }

            return ( ans);
}

int MCMF_CS2::check_cs()
{
            // check complimentary slackness;
            NODE *i;
            ARC *a, *a_stop;

            for ( i = _nodes; i != _sentinel_node; i ++) {
                        for ( a = i->suspended(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {

                                    if ( OPEN(a) && (REDUCED_COST(i, a->head(), a) < 0) ) {
                                                return ( 0);
                                    }
                        }
            }
            return(1);
}
```

```
int MCMF_CS2::check_eps_opt()
{
            NODE *i;
            ARC *a, *a_stop;

            for ( i = _nodes; i != _sentinel_node; i ++) {
                        for ( a = i->suspended(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {

                                    if ( OPEN(a) && (REDUCED_COST(i, a->head(), a) < - _epsilon) ) {
                                                return ( 0);
                                    }
                        }
            }
            return(1);
}

void MCMF_CS2::init_solution()
{
            ARC *a; // current arc (i,j)
            NODE *i; // tail of a
            NODE *j; // head of a
            long df; // residual capacity

            for ( a = _arcs; a != _sentinel_arc; a ++ ) {
                        if ( a->rez_capacity() > 0 && a->cost() < 0 ) {
                                    df = a->rez_capacity();
                                    i  = a->sister()->head();
                                    j  = a->head();
                                    increase_flow( i, j, a, df );
                        }
            }
}

void MCMF_CS2::cs_cost_reinit()
{
            if ( _cost_restart == false)
                        return;

            NODE *i; // current node
            ARC *a;          // current arc
            ARC *a_stop;
            BUCKET *b; // current bucket
            price_t rc, minc, sum;


            for ( b = _buckets; b != _l_bucket; b ++) {
                        reset_bucket( b);
            }

            rc = 0;
            for ( i = _nodes; i != _sentinel_node; i ++) {
                        rc = MIN(rc, i->price());
                        i->set_first( i->suspended());
                        i->set_current( i->first());
                        i->set_q_next( _sentinel_node);
            }

            // make prices nonnegative and multiply
            for ( i = _nodes; i != _sentinel_node; i ++) {
                        i->set_price( (i->price() - rc) * _dn);
            }

            // multiply arc costs
            for (a = _arcs; a != _sentinel_arc; a ++) {
                        a->multiply_cost( _dn);
            }

            sum = 0;
            for ( i = _nodes; i != _sentinel_node; i ++) {
                        minc = 0;
                        for ( a = i->first(), a_stop = (i + 1)->suspended(); a != a_stop; a ++) {
                                    if ( (OPEN(a) && ((rc = REDUCED_COST(i, a->head(), a)) < 0)) )
                                                minc = MAX( _epsilon, -rc);
                        }
                        sum += minc;
            }
```

```
            _epsilon = ceil(sum / _dn);

            _cut_off_factor = CUT_OFF_COEF * pow((double)_n, CUT_OFF_POWER);

            _cut_off_factor = MAX( _cut_off_factor, CUT_OFF_MIN);

            _n_ref = 0;

            _n_refine = _n_discharge = _n_push = _n_relabel = 0;
            _n_update = _n_scan = _n_prefine = _n_prscan = _n_prscan1 =
                        _n_bad_pricein = _n_bad_relabel = 0;

            _flag_price = 0;

            _excq_first = NULL;
}

void MCMF_CS2::cs2_cost_restart( double *objective_cost)
{
            // restart after a cost update;
            if ( _cost_restart == false)
                        return;

            int cc; // for storing return code;

            printf("c \nc ****************************\n");
            printf("c Restarting after a cost update\n");
            printf("c ****************************\nc\n");

            cs_cost_reinit();

            printf ("c Init. epsilon = %6.0f\n", _epsilon);
            cc = update_epsilon();

            if (cc != 0) {
                        printf("c Old solution is optimal\n");
            }
            else {
                        do { // scaling loop
                                    while ( 1 ) {
                                                if ( ! price_refine() )
                                                            break;

                                                if ( _n_ref >= PRICE_OUT_START ) {
                                                            if ( price_in() )
                                                                        break;
                                                }
                                                if ((cc = update_epsilon ()))
                                                            break;
                                    }
                                    if (cc) break;
                                    refine();
                                    if ( _n_ref >= PRICE_OUT_START ) {
                                                price_out();
                                    }
                                    if ( update_epsilon() )
                                                break;
                        } while ( cc == 0 );
            }

            finishup( objective_cost );
}

void MCMF_CS2::print_solution()
{
            if ( _print_ans == false)
                        return;

            NODE *i;
            ARC *a;
            long ni;
            price_t cost ;
//          printf ("c\ns %.0l\n", cost );
            ofstream file;
            file.open("C:\\Users\\Alamri\\Desktop\\matched.txt");
```

```
            for ( i = _nodes; i < _nodes + _n; i ++ ) {
                        ni = N_NODE( i );
                        for ( a = i->suspended(); a != (i+1)->suspended(); a ++) {
                                    //if ( _cap[ N_ARC (a) ]  > 0 ) {
                                    if (_cap[N_ARC(a)] - a->rez_capacity() > 0 && ni > 1 && ni < 7 ) {
                                                file << ni << endl << N_NODE(a-
>head()) << endl ;//, _cap[ N_ARC(a) ] - a->rez_capacity());
                                    }
                        }
    }

            // COMP_DUALS?
            if ( _comp_duals == true) { // find minimum price;
                        cost = MAX_32;
                        for ( i = _nodes; i != _sentinel_node; i ++) {
                                    cost = MIN(cost, i->price());
                        }
                        for ( i = _nodes; i != _sentinel_node; i ++) {
                                    printf("p %7ld %7.2lld\n", N_NODE(i), i->price() - cost);
                        }
            }

            //printf("c\n");
}

void MCMF_CS2::print_graph()
{
            NODE *i;
            ARC *a;
            long ni, na;
            printf ("\nGraph: %d\n", _n);
            for ( i = _nodes; i < _nodes + _n; i ++ ) {
                        ni = N_NODE( i );
                        printf("\nNode %d", ni);
                        for ( a = i->suspended(); a != (i+1)->suspended(); a ++) {
                                    na = N_ARC( a );
                                    printf("\n {%d} %d -> %d  cap: %d  cost: %d", na,
                                                ni, N_NODE(a->head()), _cap[N_ARC(a)], a->cost());
                        }
    }
}

void MCMF_CS2::finishup( double *objective_cost)
{
            ARC *a; // current arc
            long na; // corresponding position in capacity array
            double obj_internal = 0; // objective
            price_t cs; // actual arc cost
            long flow; // flow through an arc
            NODE *i;

            // (1) NO_ZERO_CYCLES?
            if ( _no_zero_cycles == true) {
                        for ( a = _arcs; a != _sentinel_arc; a ++ ) {
                                    if ( a->cost() == 1) {
                                                assert( a->sister()->cost() == -1);
                                                a->set_cost( 0);
                                                a->sister()->set_cost( 0);
                                    }
                        }
            }

            // (2)
            for ( a = _arcs, na = 0; a != _sentinel_arc ; a ++, na ++ ) {
                        cs = a->cost() / _dn;
                        if ( _cap[na]  > 0 && (flow = _cap[na] - a->rez_capacity()) != 0 )
                                    obj_internal += (double) cs * (double) flow;
                        a->set_cost( cs);
            }

            for ( i = _nodes; i != _sentinel_node; i ++) {
                        i->set_price( (i->price() / _dn));
            }

            // (3) COMP_DUALS?
            if ( _comp_duals == true) {
```

```
                        compute_prices();
            }

            *objective_cost = obj_internal;
}

void MCMF_CS2::cs2( double *objective_cost)
{
            // the main calling function;
            int cc = 0; // for storing return code;


            // (1) update epsilon first;
            update_epsilon();


            // (2) scaling loop;
            do {
                        refine();

                        if ( _n_ref >= PRICE_OUT_START )
                                    price_out();

                        if ( update_epsilon() )
                                    break;

                        while (1) {
                                    if ( ! price_refine() )
                                                break;

                                    if ( _n_ref >= PRICE_OUT_START ) {
                                                if ( price_in() ) break;
                                                if ( (cc = update_epsilon()) ) break;
                                    }
                        }
            } while ( cc == 0 );


            // (3) finishup;
            finishup( objective_cost );
}

int MCMF_CS2::run_cs2()
{
            // example of flow network in DIMACS format:
            //
            //"p min 6 8
            //c min-cost flow problem with 6 nodes and 8 arcs
            //n 1 10
            //c supply of 10 at node 1
            //n 6 -10
            //c demand of 10 at node 6
            //c arc list follows
            //c arc has <tail> <head> <capacity l.b.> <capacity u.b> <cost>
            //a 1 2 0 4 1
            //a 1 3 0 8 5
            //a 2 3 0 5 0
            //a 3 5 0 10 1
            //a 5 4 0 8 0
            //a 5 6 0 8 9
            //a 4 2 0 8 1
            //a 4 6 0 8 1"
            //
            // in order to solve this flow problem we have to follow these steps:
            // 1. ctor of MCMF_CS2 // sets num of nodes and arcs
            //                     // it also calls allocate_arrays()
            // 2. call set_arc() for each arc
            // 3. call set_supply_demand_of_node() for non-transhipment nodes
            // 4. pre_processing()
            // 5. cs2_initialize()
            // 6. cs2()
            // 7. retreive results
            //
            // this function is basically a wrapper to implement steps 4, 5, 6;

            double objective_cost;
```

```
            // (4) ordering, etc.;
            pre_processing();


            // () CHECK_SOLUTION?
            if ( _check_solution == true) {
                        _node_balance = (long long int *) calloc (_n+1, sizeof(long long int));
                        for ( NODE *i = _nodes; i < _nodes + _n; i ++ ) {
                                    _node_balance[i - _nodes] = i->excess();
                        }
            }


            // (5) initializations;
            _m = 2 * _m;
            cs2_initialize(); // works already with 2*m;
            print_graph(); // exit(1); // debug;

            printf("\nc CS 4.3\n");
            printf("c nodes: %ld  arcs: %ld\n", _n, _m/2 );
            printf("c scale-factor: %f  cut-off-factor: %f\nc\n",
                        _f_scale, _cut_off_factor);


            // (6) run CS2;
            cs2( &objective_cost );
            double t = 0.0;

            printf("c time:         %15.2f    cost:       %15.0f\n", t, objective_cost);
            printf("c refines:      %10ld     discharges: %10ld\n", _n_refine, _n_discharge);
            printf("c pushes:       %10ld     relabels:   %10ld\n", _n_push, _n_relabel);
            printf("c updates:      %10ld     u-scans:    %10ld\n", _n_update, _n_scan);
            printf("c p-refines:    %10ld     r-scans:    %10ld\n", _n_prefine, _n_prscan);
            printf("c dfs-scans:    %10ld     bad-in:     %4ld  + %2ld\n",
                        _n_prscan1, _n_bad_pricein, _n_bad_relabel);


            // () CHECK_SOLUTION?
            if ( _check_solution == true ) {
                        printf("c checking feasibility...\n");
                        if ( check_feas() )
                                    printf("c ...OK\n");
                        else
                                    printf("c ERROR: solution infeasible\n");
                        printf("c computing prices and checking CS...\n");
                        compute_prices();
                        if ( check_cs() )
                                    printf("c ...OK\n");
                        else
                                    printf("ERROR: CS violation\n");
            }

            // () PRINT_ANS?
            if ( _print_ans == true ) {
                        print_solution();
            }

            // () cleanup;
            deallocate_arrays();
            return 0;
}
//This method reads the file that contains the positions of the detectives
// Return the number of detective(this is important because along with the number of possible moves of MrX,
//it represent the number of nodes in my graph
int read_D_Pos(ifstream& file, int x[]) {
            int cnt = 0;
            string line = "";
            file.open("C:\\Users\\Alamri\\Desktop\\D_Positions.txt");
            if (!file.is_open())
                        printf("faild to open");
            else {
                        while (getline(file, line)) {
                                    x[cnt++] = stoi(line);
                        }
```

```
            }
            file.close();

            return cnt;
}
//This method reads the file that contains the possible moves of MrX
// Return the number of possible moves(this is important because along with the number of detectives ,
//it represent the number of nodes in my graph
int read_X_Pos(ifstream& file, int x []) {
            int cnt = 0;
            string line = "";
            file.open("C:\\Users\\Alamri\\Desktop\\MrX_Moves.txt");
            if (!file.is_open())
                        printf("faild to open");
            else {
                        while (getline(file, line)) {
                                    x[cnt++] = stoi(line);
                        }
            }
            file.close();
            return cnt;
}
// This method reads the shortest path between each detective and every possible move of Mrx,
//the file is formatted where every node to node is in one line.
// Returns the number of line (which represent my number of edges on the graph of course with
//adding the edges from source and SInk
int read_Shortest(ifstream& file, int x[]) {
            int cnt = 0;
            string line = "";
            file.open("C:\\Users\\Alamri\\Desktop\\Shortest.txt");
            if (!file.is_open())
                        printf("faild to open");
            else {
                        while (getline(file, line)) {
                                    x[cnt++] = stoi(line);
                        }
            }
            file.close();
            return cnt;
}

////////////////////////////////////////////////////////////////////////////
//
// main
//
////////////////////////////////////////////////////////////////////////////

int main(int argc, char *argv[])
{
            //"p min 6 8
            //c min-cost flow problem with 6 nodes and 8 arcs
            //n 1 10
            //c supply of 10 at node 1
            //n 6 -10
            //c demand of 10 at node 6
            //c arc list follows
            //c arc has <tail> <head> <capacity l.b.> <capacity u.b> <cost>
            //a 1 2 0 4 1
            //a 1 3 0 8 5
            //a 2 3 0 5 0
            //a 3 5 0 10 1
            //a 5 4 0 8 0
            //a 5 6 0 8 9
            //a 4 2 0 8 1
            //a 4 6 0 8 1"
            //int num_nodes = 6;
            //int num_arcs = 8;
            int s = 0;
            int x[20];
            ifstream file;
            int n_of_Dpos = read_D_Pos(file, x);
            cout << n_of_Dpos ;
            int y[20];
            int n_of_Xpos= read_X_Pos(file, y);
            cout << "+";
            cout << n_of_Xpos ;
```

```
            int z[60];
            int shortest = read_Shortest(file, z);
            cout << "+";
            cout << shortest;
            int num_nodes = n_of_Dpos + n_of_Xpos + 2;
            int num_arcs = n_of_Dpos + shortest + n_of_Xpos;
            cout << "+";
            cout << num_nodes;
            cout << "+";
            cout << num_arcs;
            MCMF_CS2 my_mcmf_problem( num_nodes, num_arcs);
            // first loop for the imaginary source node
            for (int i = 0; i < n_of_Dpos; i++) {
                        my_mcmf_problem.set_arc(1, i + 2, 0, 1, 1);
            }
            //second loop for the edges between the detectives and  MrXpos
            for (int i = 2; i < n_of_Dpos + 2; i++) {
                        for (int j = n_of_Dpos + 2; j < n_of_Dpos + 2 + n_of_Xpos; j++) {
                                    my_mcmf_problem.set_arc(i, j, 0, 1, z[s]);
                                    s = s + 1;
                        }
            }
            //Third loop for the edges from MrXpos to the imaginary sink
            for (int i = 0; i < n_of_Xpos; i++) {

            //The if statement is for handling if the number of detectives is more than the moves of MrX

            // We simply direct the extra detectives to the last position of the possible moves of MrX
                        if (n_of_Dpos > n_of_Xpos ) {
                                    if (i != n_of_Xpos - 1) {

            my_mcmf_problem.set_arc(n_of_Dpos + 2 + i, num_nodes, 0, 1, 1);
                                    }
                                    else

            my_mcmf_problem.set_arc(n_of_Dpos + 2 + i, num_nodes, 0, 1 + n_of_Dpos - n_of_Xpos, 1);

                        }
                        else
                                    my_mcmf_problem.set_arc(n_of_Dpos + 2 + i, num_nodes, 0, 1, 1);
            }

            my_mcmf_problem.set_supply_demand_of_node(1, n_of_Dpos);
            my_mcmf_problem.set_supply_demand_of_node(num_nodes, -n_of_Dpos);

            my_mcmf_problem.run_cs2();
            return 0;
}
```

Appendix C. Floyd-Warshall's Algorithm

```
/**
         * This method evaluates the shortest distance between all the possible
         * nodes. It uses the Floyd-Warshall's Algorithm
         */
        private int[][] getShortestDistanceMatrix(int[][] mat, int k) {
                if (k == nodes.length - 1)
                        return mat;
                int newMat[][] = new int[nodes.length][nodes.length];
                {
                        for (int i = 0; i < nodes.length; i++)
                                for (int j = 0; j < nodes.length; j++)
                                        newMat[i][j] = Math.min(mat[i][j], mat[i][k] +
mat[k][j]);
                }
                k = k + 1;
                return getShortestDistanceMatrix(newMat, k);
        }
```

Bibliography

[1] D. Monett, C. W. P. Lewis, and K. R. Th, "Journal of Artificial General Intelligence," vol. 11, no. 2, 2020.

[2] V. B. (Kisan) Pandit, "Artificial Intelligence and Expert Systems: A Technology Update," 1994.

[3] R. Myerson, *Game Theory: Analysis of Conflict*. Harvard University Press, 1991.

[4] P. Nijssen and M. H. M. Winands, "Monte carlo tree search for the game of Scotland Yard," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 4, pp. 282–294, 2012.

[5] M. L. Ginsberg, "GIB: Steps toward an expert-level bridge-playing program," *IJCAI Int. Jt. Conf. Artif. Intell.*, vol. 1, pp. 584–589, 1999.

[6] R. Zhang, H. Qiu, Y. Wang, Y. Xiao, and J. Wang, "Design and Development of Bridge AI bid Program based on Double-hand Solver," *Proc. 31st Chinese Control Decis. Conf. CCDC 2019*, pp. 6281–6286, 2019.

[7] I. Frank and D. Basin, "Search in games with incomplete information: A case study using Bridge card play," *Artif. Intell.*, vol. 100, no. 1–2, pp. 87–123, 1998.

[8] X. Liu, J. Wu, and S. Wei, "Design and implementation of military chess game algorithm based on probability model and situation evaluation," *Proc. 31st Chinese Control Decis. Conf. CCDC 2019*, pp. 6310–6314, 2019.

[9] X. Wang, J. Zhang, X. Xu, and Z. Xu, "Risk dominance strategy in imperfect information multi-player game," *Proc. - 8th Int. Conf. Intell. Syst. Des. Appl. ISDA 2008*, vol. 2, pp. 596–601, 2008.

[10] John C. Harsanyi and Reinhard Selten, *A General Theory of Equilibrium Selection*

in Games. MIT Press, 1988.

[11] D. Billings, L. Pena, J. Schaeffer, and D. Szafron, "Using probabilistic knowledge and simulation to play poker," *Proc. Natl. Conf. Artif. Intell.*, pp. 697–703, 1999.

[12] T. Cazenave, "Monte Carlo Beam Search," *IEEE Trans. Comput. Intell. AI GAMES*, vol. 4, no. 1, pp. 68–72, 2012.

[13] P. Ciancarini and G. P. Favini, "Monte Carlo tree search techniques in the game of Kriegspiel," *IJCAI Int. Jt. Conf. Artif. Intell.*, pp. 474–479, 2009.

[14] H. Finnsson, "C ADIA P LAYER : A Simulation-Based General Game Player," vol. 1, no. 1, pp. 4–15, 2009.

[15] X. Rui, X. Rui, and Y. He, "UCT-RAVE algorithm applied to multi-player games with imperfect information," *Proc. - 2011 6th IEEE Jt. Int. Inf. Technol. Artif. Intell. Conf. ITAIC 2011*, vol. 1, pp. 312–315, 2011.

[16] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," *Twent. Annu. Conf. Neural Inf. Process. Syst. NIPS 2006*, 2006.

[17] N. Sturtevant, "Last-branch and speculative pruning algorithms for max," *IJCAI Int. Jt. Conf. Artif. Intell.*, pp. 669–675, 2003.

[18] C. Luckhart and K. Irani, "An Algorithmic Solution of N-Person Games.," *Aaai*, pp. 158–162, 1986.

[19] R. E. Korf, "Multi-player alpha-beta pruning," vol. 48, pp. 99–111, 1991.

[20] N. R. Sturtevant and R. E. Korf, "On Pruning Techniques for Multi-Player Games," *Proc. Seventeenth Natl. Conf. Artif. Intell. Twelfth Conf. Innov. Appl. Artif. Intell.*, pp. 201–207, 2000.

[21] C. Ababei, "C++ Implementation of Goldberg's CS2 Scaling Minimum-Cost Flow

Algorithm," 2009. [Online]. Available: http://www.ece.ndsu.nodak.edu/~cris/software.html.

[22] A. V. Goldberg, "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm," *J. Algorithms*, vol. 22, no. 1, pp. 1–29, 1997.

[23] Stanley, J., 2019. How to win at Scotland Yard. [Blog] *James Stanley Blog*, Available at: <https://incoherency.co.uk/blog/stories/scotland-yard.html> [Accessed 28 December 2019].

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-10-2020	Master's Thesis	Aug 2018 - Oct 2020

4. TITLE AND SUBTITLE

Artificial Intelligence in Pursuit-Evasion Games, Specifically in the Scotland Yard Game

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Alamri, Arif, Capt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-20-S-003

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research provides a heuristic algorithm for the detectives, who try to collectively capture a criminal known as Mr. X, in the Scotland Yard pursuer-evasion game. In Scotland Yard, a team of detectives attempts to converge on and capture a criminal known as Mr. X. The heuristic algorithm developed in this thesis is designed to emulate human strategies when playing the game. The algorithm uses the current state of the board at each time step, including the current positions of the detectives as well as the last known position of Mr. X. The heuristic algorithm then analyses all of the possible options. The heuristic algorithm then uses a process of elimination to detemine the best possible detective moves by running an appropriately constructed minimum cost flow maximum flow instance. The heuristic algorithm was tested in a series of experiments, in which the algorithm achieved a 57% win rate. This win rate was achieved using a random starting position for each of the pursuer detectives as well as for the evader, Mr. X. When Mr. X started at an easily accessible location, namely position 146, the pursuing detectives were able to capture him 62% of the time.

15. SUBJECT TERMS

^Pursuit-evasion games
Artificial Intelligence
Heuristic algorithm

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Kenneth M. Hopkinson, AFIT/ENG
U	U	U	UU	137	19b. TELEPHONE NUMBER *(Include area code)* 312-785-3636 x4579 Kenneth.Hopkinson@afit.edu

Reset

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18