Development and Implementation of a System to Calculate Optimal Strategies for Finite Two-Person Zero-Sum Games

Jeni Prout

BSc (Hons) in Computer Science

2005

Development and Implementation of a System to Calculate Optimal Strategies for Finite Two-Person Zero-Sum Games

submitted by Jeni Prout

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed ……

Signed ……

Abstract

Game theory is a large topic that was initially recognised in the eighteenth century but more widely introduced by von Neumann, Borel and Zormelo in the 1920s. It was then further developed by von Neumann, Morgenstern and Nash in the 1940s and beyond. This dissertation briefly discusses the history of game theory and those responsible for its development, as well as its scope and applications in today's world. It then goes on to explain the definitions of game theory including different types of games, payoffs, and pure and mixed strategies. The methods for solving finite two-person zero-sum games of different sizes; in other words finding equilibrium situations over the pure or mixed strategies, are investigated in more depth. A system to solve such games is developed and implemented, and the possibilities of improving this system or extending it to cover a wider area in the extensive field of game theory are discussed.

Acknowledgments

Firstly I would like to thank my supervisor Dr. Nicolai Vorobjov for his constant help and instruction throughout the project. I would also like to thank my mother Hilary Prout, and Nicola Louzado for proof reading. Further thanks must go to my mother and Nicola as well as the rest of my family and friends, especially Gary, for their never-ending support, encouragement and patience throughout this project.

Contents

1 INTRODUCTION

One dictionary defines a game as "a competitive activity … in which players contend with each other according to a set of rules" (Dictionary.com [online] 2004). There are many everyday situations which can be thought of as games – a pedestrian crossing a road is playing a game with the drivers of the cars and those drivers are also playing a game with each other. In fact any real life situation in which two or more people interact with each other can be considered a game. Of course these are trivial examples; there are much more serious situations which can be described as games in this sense, such as economic policies and warfare strategies.

As with many real life situations, games can be analysed using mathematical models. Game theory is concerned with such modelling and determining what is the best course of action for each player. There may not be a simple answer; in fact unless the situation has been contrived to give such a result a simple solution is extremely rare. Such modelling obviously requires that we make assumptions about the players and their environment and the validity of these assumptions will affect the validity of the solution.

It is obvious that although game theory can be applied to trivial situations such as crossing a road, there is little worthwhile to be gained from mathematically calculating the pedestrian's best strategy. Game theory can however be applied usefully in a wide range of applications: it is used extensively in economics to aid decision making, particularly in situations such as oligopoly and bargaining; philosophists may be interested in the differences between the way the game theory model predicts people should act and how they actually act; and computer software agents also use it to decide between different courses of action.

As will be discussed in more detail in the literature survey in section 2, game theory models games in normal form for two players as a matrix. Each column of the matrix represents a strategy of player II and each row a strategy of player I. The element in the i^{th} row and the j^{th} column of the matrix is a vector with two components, the first of which is the payoff that player I receives and the second of which is the payoff that player II receives, given that player I plays strategy i and player II plays strategy j. An optimal strategy for a player is the one that returns the greatest payoff on average, and this dissertation is concerned with the calculation of such optimal strategies for each player.

While very small games can be solved easily by simply manipulating the payoff matrix using known facts and constraints about the matrix and strategies, the solution of large games can become very complex. The problem is concerned with optimising the payoff to each player subject to rules about the strategies and the game, and can therefore be viewed as a standard optimisation problem. Such optimisation problems can be solved using linear programming methods, in particular the simplex method which is described in the literature survey. Although the simplex method is valid in n-dimensional space, this is hard to visualise so a similar method is first described for 2xn games which can be drawn on 2-dimensional paper, making the communication of the ideas to the reader much easier.

As mentioned above the literature survey in section 2 describes to the reader the structure of games and strategies in more detail. The notation and any assumptions that will be

used throughout the dissertation are defined and discussed, giving the reader a brief overview of the problems faced by game theory as it tries to model real world situations. The large field of game theory is considered and the scope of this dissertation within that field confined to a smaller area. Some current software solutions to the problem are also examined and their effectiveness discussed.

Having analysed the problem, section 3 identifies the requirements for the system that is to be developed for this project. Section 4 discusses the algorithms that the system will implement to calculate the optimal strategies, including how it will deal with more complex special cases that may arise in certain games. An overview of how these algorithms will be implemented in the system is given in section 5, including a discussion of the user defined data structures that will be used throughout the system to aid its implementation.

The different testing methods and some more interesting test results are discussed in section 6. A full listing of test cases and results can be found in the appendix A1. An analysis of these results with a discussion of the effectiveness of the system in solving the initial problem can be found in section 7 as well as some ideas for future improvements and extensions to the system.

A full code listing and an executable file is included on an attached CD. Instructions for running the system using the files on this CD can be found in the appendix section A2.

2 LITERATURE SURVEY

2.1 Introduction

The purpose of the literature survey is to provide a preliminary investigation into the problem, the current known methods for solving the problem and any existing systems that implement these solutions.

It is widely accepted that although some ideas had arisen in the eighteenth century the initial development of game theory can be attributed to the work carried out from the 1920s by the three mathematicians von Neumann, Borel and Zermelo. Various papers and journal articles were published before John von Neumann and Oskar Morgenstern's book *The Theory of Games and Economic Behaviour* was published in 1944, truly launching the concept of game theory and encouraging its development by others.

One of those who extended this work was John Nash, who introduced in his PhD thesis in 1949 an important concept in game theory called the Nash equilibrium, which will be discussed in detail later. Nash equilibrium had a great impact on the scope and application of game theory but this impact is outside the boundaries of this project and will not be discussed.

2.2 Structure of Games

Games are defined by each player's possible actions or strategies and the payoff received by each player following each combination of strategies. For an n-player game a payoff is an n-vector $W = (w_1, w_2, \ldots, w_n)$ such that each element $w_i \in P$ for $i = 1, 2, \ldots, n$ corresponds to the payoff to player I. Such an n-vector exists for all possible combinations of players' strategies so that for a two-person game where each player has k strategies there will be k^2 possible payoffs; for three players there will be k^3 and so on.

The combinations of strategies and their corresponding payoffs can be represented in an n-dimensional matrix *A* where n is the number of players. For simplicity a two-person game will be described since this can be represented in a two dimensional matrix but this can obviously be extended up to n players using an n-dimensional matrix.

		Player II				
		y_1	y_2	y_3	…	y_n
Player I	x_1	a_{11}	a_{12}	a_{13}	…	a_{1n}
	x_2	a_{21}	a_{22}	a_{23}	…	a_{2n}
	…	…	…	…	…	…
	x_m	a_{m1}	a_{m2}	a_{m3}	…	a_{mn}

Figure 2.1
A generic two-person game in normal form

In Figure 2.1 Player I's possible strategies are $x_1, x_2, \ldots, x_m$ and player II's possible strategies are $y_1, y_2, \ldots, y_n$. The element $a_{i,j} = (\pi_x, \pi_y)$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$

is the payoff vector that is achieved when player I plays strategy x_i and player II plays strategy y_j such that player I receives the payoff π_x and player II receives the payoff π_y.

This matrix representation is known as the *normal* or *strategic* form of a game. It is a good representation for games where players make one strategy choice each and play their strategies simultaneously with no prior knowledge of their opponent's strategy choice.

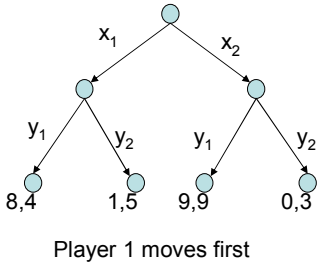
Obviously this is not always the case; sometimes players act in turns, for example in the game of chess. In this case the game should be represented in *extensive* form.

Extensive form is represented as a tree diagram. An example is shown in Figure 2.3. The nodes represent the different situations in which the game can be at each stage (that is, after an arbitrary number of moves) and the paths between the nodes represent the strategies each player can choose at each stage of the game. This representation is used for games where the players do not play their strategies simultaneously and the order in which the players play their strategies can affect the choices they make at each stage. Even if each player only moves once, knowing their opponent's strategy may affect their choice. The following example is similar to one taken from Hargreaves Heap and Varoufakis [2004], and shows the difference between representing the same game in normal or extensive form. We assume in both cases that each player wishes to maximise his own payoff and is not concerned with the payoff his opponent will receive.

	y_1	y_2
x_1	8,4	1,5
x_2	9,9	0,3

Figure 2.2
A two-person non zero-sum game in normal form

First we will look at the game in normal form as shown in Figure 2.2, assuming that both players choose their strategy simultaneously with no prior knowledge of their opponent's choice. If player II chooses strategy y_1 then player I's best response is to play x_2; because this yields a greater payoff than x_1. However if player II chooses strategy y_2 then player I would be better off playing strategy x_1. Similarly player II's best strategy choice will depend on player I's strategy. If player I chooses x_1 then player II should choose y_2, but if player I chooses x_2 then player II should choose y_1. Obviously in this game it would benefit each player to choose his strategy after his opponent's strategy is known. These different choices for the game are shown in Figure 2.3 (a) and (b).



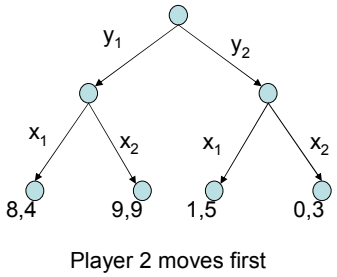


Figure 2.3(a) *Figure 2.3(b)*
A two-person non zero-sum game in extensive form

For this project only games that can be represented in normal form will be studied; that is, games where "players choose their strategies once, simultaneously and without

communicating with one other" (Hargreaves Heap and Varoufakis [2004]). Furthermore, only two-person games of this type will be studied.

Hence the scope of the project has been limited to two-person games in normal form. It will be further limited to strictly competitive games.

Strictly competitive games are also known as zero-sum games because the elements of each payoff vector sum to zero; in other words whatever is lost by one player must be won by another and vice versa. In a two-person game this means that player II's payoff is exactly the opposite of player I's, so if player I wins five points then player II loses five points; one could consider a player as 'paying' his opponent's winnings. Two-person zero-sum games can therefore be represented as a matrix where each element contains only a single value which is the payoff to player I. The payoff to player II is the negative of the value contained in the matrix.

So the two matrix games shown in Figure 2.4 represent the same game. In Figure 2.4(b) the payoff to player I ($\sigma_1(x_i,y_j)$) is shown in the matrix and the payoff to player II ($\sigma_2(x_i,y_j)$) is the negative of this value.

$a_{ij} = \sigma_1(x_i,y_j) = -\sigma_2(x_i,y_j)$ where a_{ij} are the elements of the matrix in Figure 2.4(b)

	y_1	y_2
x_1	-1,1	1,-1
x_2	1,-1	-1,1

Figure 2.4(a)

	y_1	y_2
x_1	-1	1
x_2	1	-1

Figure 2.4(b)

Two representations of a two-person zero-sum game in normal form

Game theory is an extremely large topic and limiting the scope of the project to this relatively small area will allow the project's development to reach a suitably advanced level and allow the author to acquire a greater depth of knowledge and understanding in a more specialised area.

The system that will be developed for the project will take a payoff matrix of the form shown in Figure 2.4(b) as its input, and will output for each player their best or optimal strategies. The derivation of these strategies is discussed in the following sections.

2.3 Assumptions and Notation

Some assumptions must be made about the players' behaviour and attitudes in order to derive their optimal strategies. It is difficult to precisely define an optimal strategy since two different players may not agree on what is the best strategy, depending on their different beliefs and attitudes to game playing. This is apparent in real life situations; one person may think it better to run across a busy road when there is a small gap whereas somebody else may think this is too risky and prefer to wait for a safer gap.

Firstly we assume that a player, let it be player I, believes that should his opponent, player II, discover his strategy, player II will always choose his strategy so as to minimise player I's payoff. This is not an unrealistic assumption to make in zero-sum games because by minimising his opponent's payoff, player II will also be maximising his own.

Secondly we assume that all players are conservative and not risk takers; that is, they would rather choose a strategy that guarantees receiving an outcome α than choose a strategy that risks receiving an outcome $\beta<\alpha$ even if the probability of receiving β is small and the probability of receiving $\gamma>\alpha$ with the same strategy is large.

Thirdly we assume that all players are rational and know that their opponents are rational. A player's beliefs about an opponent's rationality can greatly affect their choices. For example in the game described in Figure 2.5 a rational player II would always choose y_2 because he will always ensure a bigger payoff with this strategy. If player I believes that player II is rational and will choose y_2 then he will choose x_1 to maximise his payoff. If however player I believes that player II is irrational and will choose y_1 then he will choose x_2 because he will then secure a greater payoff. This concept of rationality was introduced by Bernheim [1984] and Pearce [1984].

	y_1	y_2
x_1	-1	-2
x_2	5	-3

Figure 2.5
A two-person zero-sum game in normal form

Some notation that will be used for the remainder of this document is now explained.

- Player I has m pure strategies $x_1, x_2, \dots, x_i, \dots, x_m$
- The set of all player I's pure strategies is denoted by x*
- Player II has n pure strategies $y_1, y_2, \dots, y_j, \dots, y_n$
- The set of all player II's pure strategies is denoted by y*
- The payoff matrix will be represented by a matrix A where each element a_{ij} is the payoff to player I when player I plays strategy x_i and player II plays strategy y_j.
- The payoff to player I of a strategy pair (x_i, y_j) is given by the function $\sigma(x_i, y_j)$
- The payoff to player II of a strategy pair (x_i, y_j) is $-\sigma(x_i, y_j)$; that is, the opposite of the payoff to player II

2.4 Pure Strategies

A pure strategy is a single strategy chosen unambiguously by a player each time the game is played (Hargreaves Heap and Varoufakis [2004]).

Take for example the game described in Figure 2.6. If player I chooses x_1 then player II will choose y_1 so as to minimise player I's payoff and maximise his own. If player I chooses x_2 then player II will still choose y_1. Player II's pure strategy for the game is therefore y_1 because it is the best response to all of player I's strategies.

	y_1	y_2
x_1	1	2
x_2	3	4

Figure 2.6
A two-person zero-sum game in normal form

Conversely if player II chooses y_1 then player I will choose x_2 so as to minimise player II's payoff and maximise his own. If player II chooses y_2 then player I will still choose x_2 because it continues to yield a greater payoff. Player I's pure strategy for the game is therefore x_2 because it is the best response to all of player II's strategies.

We can see that no matter how many times this game is played the players will always choose the strategy pair (x_2,y_1). Neither player will ever have any reason to regret their choice or change their strategy. The pair (x_2,y_1) is therefore an equilibrium situation for the game.

A strategy pair (x',y') is called an equilibrium situation if and only if the inequality shown in equation 2.1 is true. This inequality states that if player I chooses a different strategy $x_i \neq x'$ given that player II chooses y' then the payoff will be smaller and so player I has made a bad choice. If player II chooses a different strategy $y_i \neq y'$ given that player I chooses x' then the payoff to player I will be larger and so player II has made a bad choice.

$$\sigma(x_i,y') \leq \sigma(x',y') \leq \sigma(x',y_j) \; . \qquad (2.1)$$

The strategies x' and y' defined above are the optimal pure strategies for player I and player II respectively and the strategy pair (x',y') is called the Nash equilibrium or the saddle point for the game. The value of the element that is the Nash equilibrium is called the value *v* of the game. It should be noted that not all games have a Nash equilibrium over the pure strategies, whereas some have more than one. It is necessary in these cases to use mixed strategies which are discussed in the next section.

John von Neumann did a large amount of study on two-person zero-sum games and proved that they can all be solved in the same manner. His solution was mainly based on the first two assumptions stated in the previous section; that is, that each player will try to minimise his opponent's payoff and players will not take risks. Combining these assumptions he concluded that each player would aim to maximise the worst possible outcome for himself.

In the game described in Figure 2.6 the worst payoff to player I given that he chooses x_1 is 1 and the worst payoff given that he chooses x_2 is 3. Given this information player I will choose strategy x_2 because this yields the greater payoff in the worst case. This value is player I's *maximin* value and is called the *upper value* $\overline{v}$ of the game. Owen 1982 refers to this as player I's *gain floor* since it is the minimum payoff that player I can guarantee winning.

In the same game the worst payoff to player II given that he chooses y_1 is -3 and the worst payoff given that he chooses y_2 is -4, so player II will choose strategy y_1. This value is player II's *minimax* value and is called the *lower value* $\underline{v}$ of the game. Owen 1982 refers to this as player II's *loss ceiling* since player II can guarantee that this is the maximum amount that he will lose.

The method for finding these upper and lower values of a game was derived by von Neumann. In this game the maximin for player I and the minimax for player II give the same value (that is, a payoff to player I of three points) and form the strategy pair (x_2,y_1) which is the Nash equilibrium for the game. It can be proved that for all zero-sum games if the minimax and maximin are equal then the strategy pair they correspond to will be an equilibrium strategy; in other words von Neumann's minimax and maximin values

correspond to the Nash equilibrium. However the same is not true for non zero-sum games. An element in a finite zero-sum game is a Nash equilibrium if and only if it is simultaneously the smallest in its row and the largest in its column.

2.5 Mixed Strategies

Some games have no equilibrium situation over the pure strategies such as are described in the previous section, for example the game described in Figure 2.4. Alternatively some games have multiple equilibrium strategies such as the game described in Figure 2.7.

Neither player can be certain in either of these games which strategy they should choose; there is no single equilibrium situation that can be achieved using pure strategies as described in the previous section. The solution in such cases is to randomly choose a pure strategy each time the game is played. In reality the pure strategy will be selected using a probability distribution for each player over the set of all the player's pure strategies.

A mixed strategy is a probability vector such that player I's mixed strategy will be a vector $X = (p_1, p_2, \dots, p_i, \dots, p_m)$ where p_i is the probability that player I will choose pure strategy x_i and X satisfies the conditions shown in equations 2.2 and 2.3.

$$\sum_{i=0}^{m} p_i = 1 \quad . \tag{2.2}$$

$$p_i \geq 0 \quad \text{for } i = 1, 2, \dots, m \; . \tag{2.3}$$

The set of all player I's mixed strategies is $X^* = (X_1, X_2, \dots, X_m)$ which, since it is representing a complete range of probability distributions, will be an infinite set.

Similarly player II's mixed strategy will be a vector $Y= (q_1, q_2, \dots, q_j, \dots, q_n)$ where q_j is the probability that player II will choose pure strategy y_j and Y satisfies the conditions shown in equations 2.4 and 2.5.

$$\sum_{j=0}^{n} q_j = 1 \; . \tag{2.4}$$

$$q_j \geq 0 \quad \text{for } j = 1, 2, \dots, n \; . \tag{2.5}$$

The set of all player II's mixed strategies is $Y^* = (Y_1, Y_2, \dots, Y_N)$. Again this will be an infinite set.

The expected payoff using a mixed strategy is the sum of the product of each probability pair and the payoff given by the corresponding pure strategies as shown in equation 2.6. This is equivalent to the matrix multiplication of the two probability vectors X, Y and the matrix form of the game A as shown in equation 2.7

$$\tau(X,Y) = \sum_{i=0}^{m}\sum_{j=0}^{n} p_i \sigma(x_i, y_j) q_j = \sum_{i=0}^{m}\sum_{j=0}^{n} p_i a_{i,j} q_j \tag{2.6}$$

$$XAY^T = \begin{bmatrix} p_1 & p_2 & \dots & p_m \end{bmatrix} \times \begin{bmatrix} \sigma(x_1, y_1) & \sigma(x_1, y_2) & \dots & \sigma(x_1, y_n) \\ \sigma(x_2, y_1) & \sigma(x_2, y_2) & \dots & \sigma(x_2, y_n) \\ \dots & \dots & \dots & \dots \\ \sigma(x_m, y_1) & \sigma(x_m, y_2) & \dots & \sigma(x_m, y_n) \end{bmatrix} \times \begin{bmatrix} q_1 \\ q_2 \\ \dots \\ q_n \end{bmatrix} \quad (2.7)$$

The game described in Figure 2.7 describes the game of matching pennies where both players choose either heads or tails. If their strategies match; that is, they both choose heads or they both choose tails, then player I wins one unit. If they choose different strategies then player II wins one unit. Intuitively we know that both players should choose each strategy with probability ½ so that their opponent will not guess what they are going to play and they have a 50% chance of winning.

	y_1	y_2
x_1	1	-1
x_2	-1	1

Figure 2.7
'Matching pennies' - a two-person zero-sum game in normal form

The proof of this is taken from Osborne [2004].

The mixed strategies for player I and player II are $X = (p_1,p_2)$ and $Y = (q_1,q_2)$ respectively. Because these are probability vectors and must therefore sum to one they can be represented with a single probability value for each so that $X = (p, 1-p)$ and $Y = (q, 1-q)$.

If player I chooses pure strategy x_1 then his expected payoff will be

$$\begin{aligned} \tau(x_1, Y) &= [1 * q] + [-1 * (1-q)] \\ &= q - 1 + q \\ &= 2q - 1 \end{aligned}$$

Similarly if he chooses x_2 his expected payoff will be

$$\begin{aligned} \tau(x_2, Y) &= [-1 * q] + [1 * (1-q)] \\ &= 1 - 2q \end{aligned}$$

Player II wants to choose q so that the payoff is the same regardless of player I's strategy choice; that is, he wants equation 2.8 to hold. This equation can be solved to give a value for q of 0.5. By examining the payoff for different values of q we can verify that if q is less than 0.5, then $\tau(x_1, Y) < \tau(x_2, Y)$ so player I could increase his payoff by choosing x_1; in other words setting p=0. If q is greater than 0.5 then $\tau(x_1, Y) > \tau(x_2, Y)$ so player I could again increase his payoff, this time by choosing x_2; in other words setting p=1. If however q is 0.5 then $\tau(x_1, Y) = \tau(x_2, Y)$ so any combination of x_1 and x_2 will result in the same expected outcome. Therefore p can take any value in the interval p=[0,1].

$$2q-1 = 1-2q \quad (2.8)$$

A similar analysis can be carried out for player I to show that the optimal value for p, which yields the same payoff regardless of player II's strategy choice, is 0.5. This information can be represented on a graph as shown in Figure 2.8.

For player I (shown in green),

if q<½ then p=0
if q=½ then p=½
if q>½ then p=1

For player II (shown in red)

if p<½ then q=1
if p=½ then q=½
if p>½ then q=0

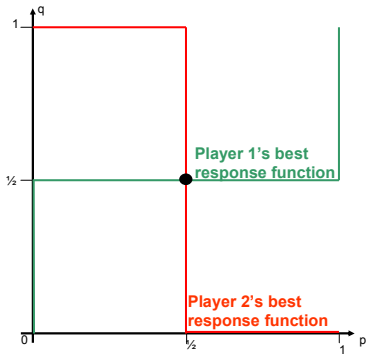


Figure 2.8
Graph of player I and player II's best response functions for the game of matching pennies

The mixed strategy pair (X', Y') is called an equilibrium situation if and only if the inequality in equation 2.9 is true. This inequality states that if player I chooses a different mixed strategy $X_i \neq X'$ given that player II chooses Y' then the expected payoff will be smaller and so player I has made a bad choice. If player II chooses a different strategy $Y_i \neq Y'$ given that player I chooses X' then the payoff to player I will be larger and so player II has made a bad choice.

$$\tau(X_i,Y') \leq \tau(X',Y') \leq \sigma(X',Y_j) \,. \qquad (2.9)$$

The strategies X' and Y' defined above are the optimal mixed strategies for player I and player II respectively and the strategy pair (X', Y') is called the mixed strategy Nash equilibrium for the game. The intersection points of the two best response curves on a graph such as Figure 2.8 show the Nash equilibria for a game. In this game there is a single Nash equilibrium over the mixed strategies but it is possible to have more than one just as it is possible to have more than one over the pure strategies. This simply means that there are multiple optimal mixed strategies, which will be discussed in more depth in later sections.

The upper and lower values of the game that were discussed in section 2.4 can also be extended to the mixed strategies. The derivations below are taken from Owen [1982].

Player I's expected gain floor assuming he uses mixed strategy X will be

$$v(X) = \min_{Y \in Y^*} XAY^T$$

where Y is player II's mixed strategy that will give the minimum payoff if player I uses X.

The mixed strategy X can be thought of as a weighted average of the expected payoffs for player I against each of player II's pure strategies. In other words for each j, X is a weighted average of the values in A_j (the j^{th} column of the payoff matrix A).

Since player II will have to eventually choose a pure strategy, to minimise the expected payoff she should choose the pure strategy y_j that gives the minimum of these weighted averages

$$v(X) = \min_j XA_j$$

Player I should therefore choose X so as to maximise the minimum of these weighted averages

$$\bar{v} = \max_{X \in X^*} \min_j XA_j$$

Similarly for player II we can find $\underline{v} = \min_{Y \in Y^*} \max_i A_i Y^T$

So $\bar{v}$ is player I's minimax strategy (the upper value of the game) and $\underline{v}$ is player II's maximin strategy (the lower value of the game).

von Neumann's minimax theorem states that $\bar{v} = \underline{v}$ for all finite zero-sum games. For this project the theorem will simply be accepted but the reader may refer to Owen [1982] for the proof or to Binmore [1992] for a less mathematically demanding proof.

2.6 Calculating Optimal Mixed Strategies

When one player has two pure strategies the method for finding the optimal mixed strategy (assuming there is no Nash equilibrium over the pure strategies) can be described graphically.

Let us assume that player I has two pure strategies and player II has n pure strategies, then the normal form of the game will be a 2xn matrix *A* of the form shown in Figure 2.9

	y_1	y_2	…	y_n
x_1	a_{11}	a_{12}	…	a_{1n}
x_2	a_{21}	a_{22}	…	a_{2n}

Figure 2.9
A generic two-person game in normal form where player I has two strategies

When the game is played player II will choose some pure strategy y_j. Player I has a mixed strategy $X = (p_1, p_2)$ but since $p_1 + p_2 = 1$ and both p_1 and p_2 are non negative (as defined in equations 2.2 and 2.3) this can be rewritten as X = (p, 1-p)

The expected payoff for player I playing mixed strategy X against player II's pure strategy y_j is:

$$\begin{aligned} \tau(X,y_j) &= p * a_{1j} + (1-p)a_{2j} \\ &= p * a_{1j} + a_{2j} - p * a_{2j} \\ &= p(a_{1j} - a_{2j}) + a_{2j} \end{aligned}$$

This is the form of a straight line with gradient $(a_{1j} - a_{2j})$ intercepting the vertical axis at a_{2j}. These straight lines of the expected payoffs to player I using X against each of player

II's pure strategies can be plotted. Each line will go through the points $(0, a_{2j})$ and $(1, a_{1j})$ as shown in Figure 2.10.

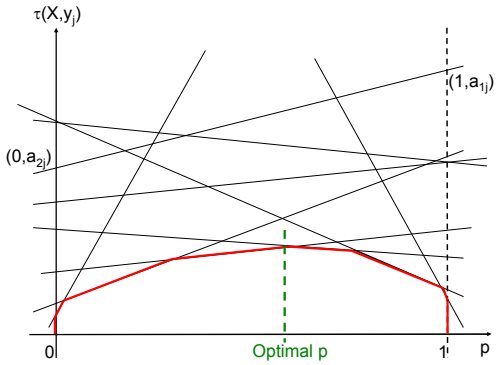


Figure 2.10
Graph showing the payoff generated by each of player II's pure strategies against each value for p

The worst case for any value of p is the value of the payoff given by the lowest line at p. This creates a piecewise line for p which is shown in red in Figure 2.10. The optimal value of p is the maximum peak of this red line indicated in the diagram by the dashed green line. This peak does not necessarily have to be a point; it could be an interval meaning that any value for p between the limits of the interval would generate an optimal mixed strategy.

A similar algorithm can be used when player II has two pure strategies and player I has m pure strategies. In this case the normal form of the game would be the matrix *A* shown in Figure 2.11.

	y_1	y_2
x_1	a_{11}	a_{12}
x_2	a_{21}	a_{22}
…	…	…
x_n	a_{n1}	a_{n2}

Figure 2.11
A generic two-person game in normal form where player II has two strategies

The expected payoff to player I when player II uses the mixed strategy Y = (q, 1-q) is $\tau(x_i,Y) = q(a_{i1}-a_{i2})+a_{i2}$. Again this is a family of straight lines which each go through the points $(0, a_{i2})$ and $(1,a_{i1})$ as shown in Figure 2.12.

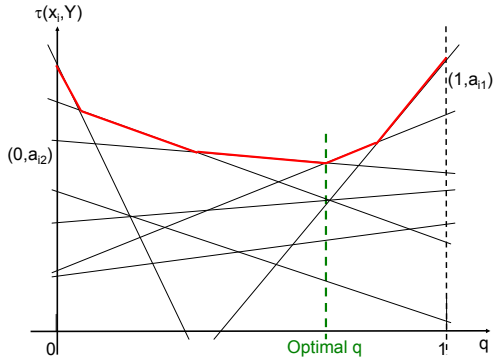


Figure 2.12
Graph showing the payoff generated by each of player I's pure strategies against each value for q

Player II wants to minimize the worst case payoff so instead of the peak of the minimal line, it is the lowest point of the maximal line that gives the optimal value for q.

If player I has two pure strategies then the set of player II's pure strategies contributing to the graph can be reduced to the two strategies that generate the maximal peak of the minimal line. The game has thus been reduced to a 2x2 game so both players' optimal mixed strategies can be found using the methods described above. The reader should note that this conclusion assumes that only two lines pass through the maximal vertex on the graph used to calculate the optimal value for p. The case of more than two lines passing through this vertex is discussed in section 4.4.3.

One algorithm that would implement this method to find the optimal mixed strategy for player I, where player I has two pure strategies, is as follows:

1. Find the equations for all straight lines on the graph corresponding to all the pure strategies of player II.
2. Find all the points of intersection between these lines; there will be maximum n^2 points.
3. Discard all intersection points which lie outside the interval p=[0,1].
4. Loop through each intersection point finding the points that lie on the minimal line.
5. Find the maximum point of those found in step 4.

This is an inefficient algorithm since it involves examining every intersection point. A more efficient algorithm is detailed below.

1. Reduce player II's choice of strategies so that there are none where a single different pure strategy is a better response to both of player I's pure strategies.
2. Find the equations for the straight lines corresponding to all the remaining strategies.
3. Find the line that is minimum at p=0.
4. Calculate this line's intersection with all other lines.
5. Find the intersection where p is the minimum over all the intersection points with p>0, and find the line it corresponds to.
    a. Let p_c be the value of p at this intersection.
    b. Let l_c be the equation of the second line at this intersection.
6. Calculate the value of p at the intersection of l_c with all other lines, excluding those that have already been used. In other words calculate p_i for every other line l_i such that $l_c(p_i) = l_i(p_i)$.
7. Find the intersection with the smallest value for p_i where $p_i > p_c$ and find the line it corresponds to.
    a. If there is no intersection with $p_i > p_c$ then the current line is the final section of the piecewise minimal line; proceed to step 9.
    b. Otherwise
        i. Let p_c be the value of p at this intersection.
        ii. Let l_c be the equation of the second line at this intersection.
8. If $p_c < 1$ and not all lines have been used then return to step 6.
9. The vertices on the piecewise minimal line are the points $l_c(p_c)$.
    a. Find the maximum of these points; that is, the point giving the maximum payoff not the point with the maximum value for p.

Depending on the outcome of step 1 this could still involve examining every intersection point. However this would be the worst case scenario and step one could significantly reduce the number of points that need to be examined.

When one player has three strategies the method is similar but the graph is extended to three dimensions with intersecting planes instead of intersecting lines. This is difficult but not impossible to draw however when both players have more than three pure strategies it becomes extremely complex to depict graphically. The same ideas still apply in the higher dimensions; there will be intersecting multidimensional simplices representing the payoffs yielded by using different mixed strategies against an opponent's pure strategies. The algorithms above can be extended to find the maximum point of the polytope generated by the minimum values of these intersecting simplices

2.7 Linear Programming and the Simplex Method

Linear programming is concerned with maximizing or minimizing a function subject to a set of constraints. There are specific conditions which must be adhered to for the program to be strictly in standard form but these will not be discussed in this project. For further information the reader may refer to Fourer [2004] which gives a brief overview of linear programming and references many suitable further sources of information on the topic.

Essentially a linear program has a linear function called the objective function in a set of variables, and a set of equalities or inequalities on these variables that must be satisfied. The linear program calculates either the maximum or minimum value that the objective function can attain to while simultaneously adhering to all of the constraints.

One of the main types of linear programming solutions and the method that is most suitable for calculating optimal strategies is the simplex method invented by George Dantzig in 1947. Fourer [2004] describes concisely how this method works:

> "Basic solutions represent extreme boundary points of the feasible region … the simplex method can be viewed as moving from one such point to another along the edges of the boundary."

This is essentially the same algorithm as described for 2xn games in a higher dimension; it involves moving along the vertices of a polytope until it finds the most optimal vertex or vertices.

For the purposes of game theory the objective function is the expected payoff to player I which should be maximized for player I and minimized for player II. The constraints are the conditions that must be satisfied for the mixed strategy to be a probability vector. The generation of a polytope that contains the optimal strategy by intersecting simplices is described for 2xn games in the previous section.

The 'basic solutions' described above by Fourer [2004] are the vertices of this polytope. The simplex method starts with one of these vertices and moves to whichever adjacent vertex is optimal for the objective function; that is, whichever vertex is larger for maximization or smaller for minimization. A pivot method is used to decide which of the adjacent vertices is optimal; there are various different pivot methods, one of which is described below. The optimum must be a vertex of the polytope so the interior points can be ignored. It is possible for two vertices to be optimal and thus both of these vertices and the edge between them would form the set of optimal solutions. Indeed it is possible for many vertices to be optimal and the set of optimal solutions would be the convex hull of

these vertices. This is discussed in greater depth in section 4.4.3. When none of the adjacent vertices are more optimal, then the current vertex must be the optimal solution.

It is the algorithm for the pivot method that determines the complexity of the various simplex methods. Theoretical analysis of the worst case complexity of the simplex algorithm shows that it could visit all 2^n vertices of the polytope making its complexity exponential time. In 1972 Klee and Minty gave an example where this is the case and such examples have been found for all pivot methods. However the algorithm does guarantee reaching the correct value and in practice the average case complexity is extremely efficient. It is not yet known whether there is a pivot method with polynomial time worst case complexity.

2.8 Using the Simplex Method to Calculate Optimal Mixed Strategies

The description of the simplex method below and the method of using it to calculate optimal mixed strategies is taken from Ferguson [2004].

Player I wants to choose X = (p_1, p2, …, p_m) so as to

maximize $\min_{1 \le j \le n} \sum_{i=1}^{m} p_i a_{i,j}$

subject to $p_1 + p_2 + \ldots + p_m = 1$

$p_i \geq 0$ for i = 1, 2, …, m

This is not a linear program because the $\min_{1 \le j \le n}$ in the objective function prevents it from being linear in p. However it is possible to force it to be linear by using a false variable v where $v \le \min_{1 \le j \le n} \sum_{i=1}^{m} p_i a_{i,j}$ and then the objective function becomes

maximize v

subject to

$$v \le \sum_{i=1}^{m} p_i a_{i,1}$$

$$v \le \sum_{i=1}^{m} p_i a_{i,2}$$

...

$$v \le \sum_{i=1}^{m} p_i a_{i,n}$$

$p_1 + p_2 + \ldots + p_m = 1$

$p_i \geq 0$ for i = 1, 2, …, m

Similarly the objective function for player II becomes

minimize w

subject to

$$w \geq \sum_{j=1}^{n} a_{1,j} q_j$$

$$w \geq \sum_{j=1}^{n} a_{2,j} q_j$$

...

$$w \geq \sum_{j=1}^{n} a_{m,j} q_j$$

$q_1 + q_2 + \ldots + q_n = 1$

$q_j \geq 0$ for j = 1, 2, …, n

The theory of duality and the Duality Theorem essentially state that if one program is aiming to maximize a function and another is aiming to minimize the negative of the same function then they are called dual programs and will return the same value. This situation precisely describes the two programs above so it is only necessary to optimize one of the objective functions.

The optimization can be simplified if the value of the game is known to be positive. Assume $v \geq 0$ and let $r_i = \frac{p_i}{v}$ so $r_1 + r_2 + \ldots + r_m = \frac{1}{v}$

To maximize v we want to minimize $\frac{1}{v}$ so the program becomes

minimise $r_1 + r_2 + \ldots + r_m$

subject to

$$1 \leq \sum_{i=1}^{m} r_i a_{i,1}$$

$$1 \leq \sum_{i=1}^{m} r_i a_{i,2}$$

...

$$1 \leq \sum_{i=1}^{m} r_i a_{i,n}$$

$r_i \geq 0$ for i = 1, 2, …, m

When this is solved the value of the game $v = \frac{1}{r_1 + r_2 + \ldots + r_m}$ and the elements of the probability vector X are $p_i = v x_i$ for i = 1, 2, …, m

The pivot method below details how to calculate the value of a game and both players optimal mixed strategies.

1. Add a constant k to all values of the payoff matrix to ensure that the value of the game is positive (this k must be subtracted from the value returned by the algorithm to ensure the correct result).
2. Create a tableau using the payoff matrix, the players' strategies, a border column and row of 1's and -1's respectively along the right side and bottom of the matrix, and a zero in the bottom right corner as shown below:

	y_1	y_2	…	y_n	
x_1	a_{11}	a_{12}	…	a_{1n}	1
x_2	a_{21}	a_{22}	…	a_{2n}	1
…	…	…	…	…	1
x_m	a_{m1}	a_{m2}	…	a_{mn}	1
	-1	-1	-1	-1	0

Figure 2.15
A generic simplex tableau

3. Pick a pivot in the interior of the tableau, let it be element $a_{f,h}$.
    a. The border value at the bottom of column h must be negative
    b. The pivot value must be positive
    c. The row must be chosen so that the ratio between the border value at the right of row f and the pivot is the smallest for all elements in column h
4. Pivot the matrix on this element.
    a. Replace each element that is not in the same column or row as the pivot with $a_{i,j} - \frac{a_{f,j} \times a_{i,h}}{a_{f,h}}$
    b. Replace each element that is in the same row as the pivot except the pivot itself with $\frac{a_{i,j}}{a_{f,h}}$
    c. Replace each element that is in the same column as the pivot except the pivot itself with $-\frac{a_{i,j}}{a_{f,h}}$
    d. Replace the pivot with its reciprocal $\frac{1}{a_{f,h}}$
5. Exchange the label on the left of the pivot row with the label on the top of the pivot column.
6. If there are any negative numbers in the bottom row go back to step 3.
7. The solution for the game can now be found:
    a. The value of the game is the reciprocal of the bottom right element. If a constant k was added in step one then it must be subtracted here to get the true value of the original game
    b. Player I's optimal strategy:
        i. If x_i is somewhere in the leftmost column then $p_i = 0$
        ii. If x_i is somewhere in the top row then p_i is the value at the bottom of its column divided by the bottom right element
    c. Player II's optimal strategy:
        i. If y_i is somewhere in the top row then $q_i = 0$
        ii. If y_i is somewhere in the leftmost column then q_i is the value at the end of its row divided by the bottom right element

This returns one optimal mixed strategy. A further pivot step on a pivot where either the element in the border row or border column is zero, so that none of the border elements become negative, will return another strategy.

2.9 Existing Solutions

The mathematics software package Maple has a simplex package that uses an implementation of the simplex algorithm to solve linear programs.

There are various applets available on the internet for calculating optimal mixed strategies for two-person games which are of the same form as this project. One particular applet found at Warner and Costenoble [1997b] allows the user to input the normal form of a two-person zero-sum game and outputs, among other things, the optimal strategies for both players. The algorithm used to implement this can be found at Warner and Costenoble [1997a] but is essentially the pivot method described above.

Savani [2004] gives a solution that uses a reverse search algorithm to find solutions to bimatrix games. It can be used to find solutions to zero-sum games but the output is in a less comprehensive form to Warner and Costenobole's solution.

Most other solutions required the user to input the standard form of the linear program. Since this system will be required to solve matrix games, to make the system more usable the user will only be expected to input the payoff matrix. The user is not required to translate the payoff matrix into an objective function and variable constraints, which would require knowledge of linear programming as well as knowledge of game theory.

Almost all of the solutions found are tools used to illustrate the concepts of game theory, linear programming and the simplex method. Although the wide scope of game theory has extensive applications in today's world, most of these involve much more complex games than those falling within the project's scope. Therefore it is apparent that the system that will be developed for this project cannot be used by those wishing to model and solve a real life situation using game theory. The system will only be used by those wishing to find the optimal strategies for those simple games that the system will accept. These users may wish to find such strategies for educational purposes; to test another system's solutions or methods, or to illustrate some point about game theory or certain methods.

3 REQUIREMENTS

3.1 Introduction

The aim of this dissertation is to develop a computer system that allows a user to input a matrix representing the payoff function of a two-person zero-sum finite game, then calculates and outputs the optimal pure or mixed strategies for each player and the value of the game.

The literature survey gave a brief overview of the known algorithms and methods used to calculate these mixed strategies for each player, as well as examining and analyzing the existing software solutions to the problem and the adequacy of these solutions.

This requirements specification aims to define the features and functionality required by the system, as well as any constraints on the project and additional non essential functionality that would be desirable in the system. The system developed through this dissertation will be designed only to provide the functionality that would be used in a final end user system. The design and implementation of the 'front end' for the system, in other words a usable interface, could be the subject of a separate dissertation in itself, and therefore the user interface will be a basic command line interface allowing the input of the matrix game in a more basic way.

The scope of the system is limited to solving finite two-person zero-sum games; therefore the ways in which a user can use the system is limited to solving these games. The single top level requirement of a user is to be able to solve such games. Owing to this fact the requirements elicitation process has been limited to discussions with the project tutor about the requirements and constraints for different types of input, output and computation.

The solution of mxn games can be found using linear programming methods as discussed in the previous section. There is code available that implements the simplex method to solve these problems, and some of this code can be obtained free of charge. It has been decided that this code would not be suitable for the purposes of this project as it would contain a lot of excess code not required for the project and would inhibit the learning and understanding acquired through the project as less insight into the solution would be gained.

3.2 Project requirements

The project is constrained by time, having only a few months from conception to completion including the design, implementation, testing and writing of the report (this dissertation). The system must therefore be designed so that it can realistically be finished within this time whilst also allowing for the other academic commitments of the author.

1. Core functionality should be identified and the system designed so that this can be implemented first and additional features added incrementally if time allows.
2. The system should initially be developed so that it runs reliably and correctly on the University of Bath's BUCS computers. It can be adapted to allow for portability to other platforms if there is sufficient time at the end of the project

3. The functionality of the system should be specified and implemented first. If there is sufficient time after the implementation of this functionality then an investigation into the requirements and possible implementation of a graphical user interface to make the system more user friendly can be made.
4. The entire project, including the implementation of the system and the writing of this dissertation, must be complete before 16th May 2005.
   4.1. The dissertation must conform to the specification laid out by the Department of Computer Science, University of Bath as defined in "The Project Dissertation" (Barry [2004]).

3.3 System requirements

Somerville [2001] separates system requirements into three main categories.

- Functional requirements "describe the functionality or services that the system is expected to provide" (Sommerville [2001]). They should explicitly define what the system should or should not do, including definitions of the acceptable inputs and outputs, and how it should handle exceptions.

- Non functional requirements refer to the emergent properties of the system such as reliability or safety, and any constraints on time or memory that the system must adhere to.

- Domain requirements include functional and non functional requirements and depend on the environment or domain in which the system will be used. They may not be required for the system to function correctly but are needed for the system to be used in the required domain, for example legal or safety requirements arising from the intended environment.

The system will be developed incrementally so that core functionality can be implemented first. Designing and developing the system in this incremental way will encourage modularity in the final system and therefore increase its maintainability as well as allowing the project to reach intermediate end points before the final completion of the system. If problems are encountered that delay the project so that it cannot be completed in the allocated time then it should still be at some intermediate state which can be considered complete for the purpose of writing this dissertation.

To allow this intermediate end point the system will first be developed to solve only games where at least one of the players has only two strategies. Once this functionality has been successfully implemented the system can be extended to solve larger games where both players have more than two strategies. The requirements for both stages of the system will be almost identical, the only difference being that requirement 1.3 below should be modified for the intermediate end point to state that:

The user should be able to enter any 2x2, 2xn or mx2 game where $m>2$ and $n>2$.

The scope of the system is strictly limited to two-person zero-sum finite games. It will not at any stage consider n-person games, dynamic games or non zero-sum games.

Requirements will be written in the form:

1. "The system *must* …" indicating a core requirement which is essential to the system and must be implemented

2. "The system *should* ..." indicating a preferred requirement that should be implemented but not at the expense of any of the core requirements
3. "The system *may* …" indicating a desirable requirement that is not required for the functionality of the system but will increase its scope and/or usability. These should only be considered if all core functionality has been successfully implemented

3.3.1 Functional Requirements

1. Data input:
   1.1. The user must be able to enter the payoff matrix from a command line prompt in the form $[[a_{1,1}, a_{1,2}], [a_{2,1}, a_{2,2}], \dots]$
      1.1.1. The user should be able to enter white spaces between the elements or rows to make the input more readable
   1.2. The user should be able to enter the payoff matrix from a file by entering $progName :fileName at the command line
      1.2.1. The file should have the same format as the command line input
      1.2.2. The user should be able to enter white spaces between the elements or rows (including carriage returns) to make the input more readable.
   1.3. The user must be able to enter any mxn matrix where m>1 and n>1
   1.4. The user must be able to enter positive and negative elements in the payoff matrix
   1.5. The user must be able to enter integer or floating point values in the payoff matrix
   1.6. The user may be able to enter fractions in the payoff matrix
2. Data output:
   2.1. If the user entered the payoff matrix from the command line prompt then the output must be written to the screen below the command line input
   2.2. If the user entered the payoff matrix from a file then the output should be appended to that file
   2.3. The system must output an optimal strategy for each player and the value of the game in the following format

        Player 1: $[p_1, p_2, \dots p_m]$
        Player 2: $[q_1, q_2, \dots q_n]$
        Game value = v

   2.4. The system should output any further strategies that have been calculated by the system for each player and the value of the game in the following format

        Strategy 1:
        Player 1: $[p_{11}, p_{12}, \dots p_{1m}]$
        Player 2: $[q_{11}, q_{12}, \dots q_{1n}]$
        Game value = v
        Strategy 2:
        Player 1: $[p_{21}, p_{22}, \dots p_{2m}]$
        Player 2: $[q_{21}, q_{22}, \dots q_{2n}]$
        Game value = v
        …
        Strategy k:
        Player 1: $[p_{k1}, p_{k2}, \dots p_{km}]$
        Player 2: $[q_{k1}, q_{k2}, \dots q_{kn}]$
        Game value = v

   2.5. If interval strategies exist for 2xn or mx2 games then these should be represented in the following form where the probability that Player I chooses strategy 1 can range from p_{1a} to p_{1b}, and $p_{2a} = 1-p_{1a}$ and $p_{2b} = 1-p_{1b}$

        Strategy 1:

Player 1: $[(p_{1a} - p_{1b}), (p_{2a} - p_{2b})]$
Player 2: $[q_1, q_2, \ldots q_n]$
Game value = *v*

Since a 2xn or mx2 game can be reduced to a 2x2 game, the optimal strategy for Player II containing an interval can be represented in the following form, where the probability that Player II chooses strategy i ranges from q_{ia} to q_{ib} and the probability that he chooses strategy j ranges from q_{ja} to q_{jb} accordingly. Values for q_k are zero where $1 \leq k \leq n$, $k \neq i, j$.

Strategy 1:
Player 1: $[p_1, p_2]$
Player 2: $[q_1, (q_{ia} - q_{ib}), \ldots, (q_{ja} - q_{jb}), \ldots q_n]$
Game value = *v*

2.6. With reference to requirements 2.3-2.5
- 2.6.1. Each p_i and q_i must be able to be either an integer or floating point number within the interval [0,1]
- 2.6.2. Each p_i and q_i may be a fraction within the interval [0,1]
- 2.6.3. The following must be true for every strategy

$$\sum_{i=1}^{m} p_i = 1 \qquad \sum_{j=1}^{n} q_j = 1$$

- 2.6.4. The value of the game *v* must be able to be either an integer or floating point number
- 2.6.5. The value of the game *v* may be a fraction
- 2.6.6. The value of the game *v* must be able to be positive, negative or zero

3.3.2 Non Functional Requirements

3. The output must be returned within a reasonable time
   3.1. The output must be returned within 2 seconds
4. Any errors should be reported to the user and the program terminated
   4.1. System errors encountered such as memory allocation errors or file handling errors should be reported to the screen, even if the payoff matrix was input from a file, so that the user is immediately aware of them.
   4.2. Errors due to incorrect data entry such as entering an invalid matrix should be output to either the screen or input file, depending on the input method. This will allow the user to examine the error message and the erroneous input simultaneously.

3.3.3 Domain Requirements

5. If the user does not enter the matrix in the required format (for example there are missing or additional operators or numbers) then the system must report the error and allow the user to enter a new matrix.

4 DESIGN

4.1 Introduction

The literature survey gave a brief description of the algorithms used to solve the matrix games that the system is concerned with. This section will describe these algorithms in a much greater depth and discuss the practical implications of any special cases that might arise and how these will be dealt with.

4.2 Overall Structure of the System

If the matrix game is smaller than 2x2 then the system will not deal with it since it is too trivial. One player has only one strategy therefore has no choice over which strategy to play. His opponent chooses the strategy that yields the optimal payoff for himself. Therefore if the matrix game input has m<2 or n<2 then an error message will be reported to the user informing him that the game is too small to be solved by the system.

If the matrix game is an mx2 game with m>2 then it must be transformed into an equivalent 2xn game; the algorithm for transforming the game is described in section 4.5. It can then be solved using the algorithm for solving 2xn games and the result transformed to correspond to the original mx2 game.

If the matrix game is a 2x2 game then the system first checks for a saddle point over the pure strategies. If no saddle point exists then the system will find the optimal mixed strategy for the game.

If the matrix game is a 2xn game with n>2 or an mx2 game with m>2 that has been transformed into its equivalent 2xn game then it must be solved to find the optimal strategies and the value of the game. The algorithm for solving this finds both pure and mixed strategies, and is described in detail in section 4.3.

If the matrix game is an mxn game with both n>2 and m>2 then the simplex method must be used to find the optimal strategies. mxn games are solved separately in this way to allow the solution of 2xn and mx2 games to be implemented first. This incremental development will allow the system to reach an end point where it solves mxn games where either m or n is equal to two.

4.3 Solving 2x2 Games

4.3.1 Finding Pure Strategies

The system must try to find the saddle point of the game $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$

The system first finds player I's maximin value by finding the minimum element in each row, then finding the maximum of these two minimums. It then finds player II's minimax value by finding the maximum element in each row, then finding the minimum of these two maximums. If these two elements are in fact the same element then it is a saddle point. If the elements are not the same or it was impossible to find the minimax or

maximin due to the strategies yielding equal numbers, then there is no saddle point and the system must find the optimal mixed strategies for the players.

$$A=\begin{bmatrix}1 & 0\\ 0 & 2\end{bmatrix} \qquad A=\begin{bmatrix}2 & 0\\ 1 & 3\end{bmatrix} \qquad A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$$

Figure 4.1(a) *Figure 4.1(b)* *Figure 4.1(c)*

Examples of two-person 2x2 games with and without saddle points

In Figure 4.1(a) player I has no maximin because the minimum value in the first row is the same as the minimum value in the second row. It is therefore impossible to define a maximum between these two values, so player I has no maximin value and thus there is no saddle point.

Player I's maximin element in Figure 4.1(b) is a_{21}=1. However player II's minimax element is a_{11}=2. These two elements are different therefore there is no saddle point for this game. The system is therefore required to calculate the optimal mixed strategies for the games described in Figures 4.1(a) and 4.1(b).

In Figure 4.1(c) player I's maximin element and player II's minimax element are both a_{21}=3. Since player I's maximin and player II's minimax element are the same, the game has a saddle point which is this element. Therefore the optimal strategies for this game are pure strategies x_2 for player I and y_1 for player II.

4.3.2 Finding Mixed Strategies

This method for finding optimal mixed strategies for 2x2 games is taken from Owen [1982]. The game is solved using known facts about matrices and constraints on the optimal strategies to manipulate the matrix and find the strategies and value of the game.

The system must solve the matrix game A for X, Y and v where

$$A=\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}$$

$X = (p_1, p_2)$ is the optimal mixed strategy for Player I
$Y = (q_1, q_2)$ is the optimal mixed strategy for Player II
$v = XAY^T$ is the value of the game

Let $J=\begin{bmatrix}1 & 1\end{bmatrix}$, then the following facts are known:

$$p_1+p_2=1 \tag{8.1}$$

$$q_1+q_2=1 \tag{8.2}$$

$$XA=\begin{bmatrix}v & v\end{bmatrix}=vJ \qquad \Rightarrow X=vJA^{-1}. \tag{8.3}$$

$$AY^T=\begin{bmatrix}v\\ v\end{bmatrix}=vJ^T \qquad \Rightarrow Y=A^{-1}vJ^T. \tag{8.4}$$

The sum of the components of $X = XJ^T = 1$. Combining this with equation 8.3 we can obtain values for v and X in terms of A and J as shown in equation 8.5 and 8.6 respectively. A similar analysis with equation 8.4 leads to the value for Y shown in equation 8.7.

$$XJ^T = vJA^{-1}J^T$$
$$\Rightarrow v = \frac{1}{JA^{-1}J^T}. \tag{8.5}$$

$$X = vJA^{-1}$$
$$\Rightarrow X = \frac{JA^{-1}}{JA^{-1}J^T}. \tag{8.6}$$

$$Y = \frac{A^{-1}J^T}{JA^{-1}J^T} \tag{8.7}$$

The above assumes that A is non singular; that is, it assumes that A has an inverse. It is a trivial observation that if A is singular then A^{-1} in the above formulas can be replaced with A^*, the adjoint of A. This still holds when A is non singular.

So for a matrix game A with no saddle point, equations 8.5, 8.6 and 8.7 give equations 8.8, 8.9 and 8.10 respectively, where A^* is the adjoint of A and $|A|$ is the determinant of A.

$$X = \frac{JA^*}{JA^*J^T} \tag{8.8}$$

$$v = \frac{|A|}{JA^*J^T} \tag{8.9}$$

$$Y = \frac{A^*J^T}{JA^*J^T} \tag{8.10}$$

Since we know the individual elements of A, we know that $A^* = \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$ and $|A| = a_{11}a_{22} - a_{12}a_{21}$. By substituting these facts into equation 8.8 we can calculate X in terms of the individual elements of A as shown below, leading to a final value for X in equation 8.9.

$$X = \frac{\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}a_{22} & -a_{12}\\ -a_{21} & a_{11}\end{bmatrix}}{\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}a_{22} & -a_{12}\\ -a_{21} & a_{11}\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}}$$

$$\Rightarrow X = \frac{\begin{bmatrix}a_{22}-a_{21} & a_{11}-a_{12}\end{bmatrix}}{(a_{22}-a_{21})+(a_{11}-a_{12})}$$

$$\Rightarrow X = \begin{bmatrix}\frac{a_{22}-a_{21}}{a_{22}-a_{21}+a_{11}-a_{12}} & \frac{a_{11}-a_{12}}{a_{22}-a_{21}+a_{11}-a_{12}}\end{bmatrix} \quad (8.11)$$

Similar substitutions into equations 8.9 and 8.10 lead to final values for *Y* and *v,* as shown in equations 8.12 and 8.13. These final values can simply be coded into the system as functions of the elements of the payoff matrix input by the user.

$$Y = \begin{bmatrix}\frac{a_{22}-a_{12}}{a_{22}-a_{21}+a_{11}-a_{12}} & \frac{a_{11}-a_{21}}{a_{22}-a_{21}+a_{11}-a_{12}}\end{bmatrix} \quad (8.12)$$

$$v = \frac{a_{11}a_{22}-a_{12}a_{21}}{a_{22}-a_{21}+a_{11}-a_{12}} \quad (8.13)$$

4.4 Solving 2xn Games

2xn games can be solved using the simplex method. However as noted in the requirements section it is necessary to allow for different end points for the system, allowing it to be in some complete state if problems arise preventing it from being completed to the final specification. For this reason a different algorithm will be used and separate functions will be programmed to solve 2xn and mx2 games. This also allows deeper insight into the method of solving games, and greater understanding of how the simplex method works at a lower dimension.

4.4.1 Player I's Optimal Strategy

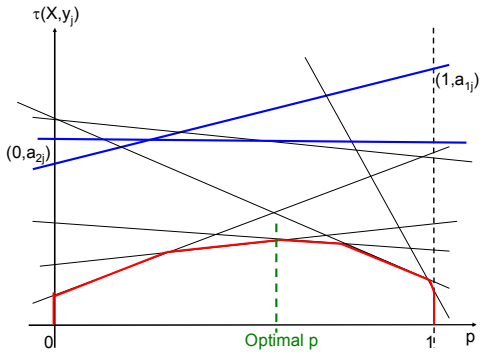


Figure 4.4
Graph showing the payoff generated by each of player II's pure strategies against each value for p

For player I's mixed strategy X=(p, 1-p), Figure 4.4 shows the payoff of each of player II's pure strategies against every value of p; each line on the diagram represents one of Player II's pure strategies. The lower piecewise red line shows the minimum value for the payoff for every value of p. The system is aiming to find the value for p where this red line is highest. Because the line is convex, this maximum point will clearly be the point where two lines intersect such that the line on the left has a positive gradient and the line on the right has a negative gradient. If the line on the right has a gradient equal to zero then there will be an interval for the optimal mixed strategy, since both vertices at the limits of the interval are optimal.

The algorithm starts on the red line at p=0 and works along the lower line until it finds the junction where the gradients on each side satisfy the conditions above. To give the algorithm a starting vertex on the lower red line from which it can search for the peak, the algorithm is required to first find the line which has the minimum value over all the lines at p=0. Any lines with higher values at both zero and one, such as those shown on Figure 4.4 in blue, can be discarded. These lines will not intersect this first line within the interval p=[0,1] and therefore will not appear on the lower red line.

If the gradient of this first line is less than zero then the optimal value for p is zero; player I will therefore have the optimal pure strategy x_2. Player II will also have a pure strategy which will be the column in the matrix that generates this line.

If the gradient of this line is zero then the optimal value for p is an interval from zero to the next junction on the red line, which will be the next intersection between this and some other line. If there is no such junction where $p \leq 1$ then the interval for optimal p is the entire interval [0,1], since the horizontal line makes up the red line for the whole of this interval. Player II will have a pure strategy which will be the column in the matrix that generates this horizontal line.

If the gradient of this line is greater than zero, then the intersection between this and all other lines is calculated. The line that intersects at the smallest value for p between this intersection (in the first case this will be zero) and one is the next line to be considered. If no lines intersect within this interval then this line is the last line to be considered. In this case the value for p will therefore be one, giving player I a pure strategy x_1 and player II the pure strategy that is the column that generates this line.

If however another line does intersect within this interval and the gradient of this new line is less than zero, then the optimal value for p is the value at which these two lines intersect. These two lines can be plotted on a graph for player II to calculate his optimal strategy Y as shown in section 4.4.2.

If the gradient of this new line is equal to zero then the optimal value for p is an interval from this junction to the next on the lower red line. If there is no such junction with $p \leq 1$, then the end point of this interval for optimal p is one. Player II will have a pure strategy which will be the column in the matrix that generates this horizontal line.

If the gradient of this new line is greater than zero then the above steps must be repeated until either a junction where the second line has a gradient less than or equal to zero is found, or the value for p at the junction is greater than one.

4.4.2 Player II's Optimal Strategy

As described above if player I has a pure strategy then player II also has a pure strategy which is the best response to player I's. There may be cases where player I has a pure strategy and player II has an interval strategy but such cases can become extremely complex to calculate as shown in section 4.4.3 and will not be considered for this system.

If the steps above yield a single value for p where $0<p<1$; in other words player I has a single defined mixed strategy, not a pure or interval strategy, then the columns generating the two lines intersecting at that maximal vertex are the only strategies to have non zero probabilities within the mixed strategy probability distribution for player II.

The game is thus reduced to a 2x2 game and the value for q can be found in a similar way to p. Each row of the 2x2 matrix can be represented as a line on the graph with points at $(0,a_{i,2})$ and $(1,a_{i,1})$ as shown in Figure 4.5. The optimal value for q is the minimum value of the maximal piecewise line that is created by these lines. This piecewise line is shown on the diagram in red, while the optimal value for q is indicated with a green dashed line.

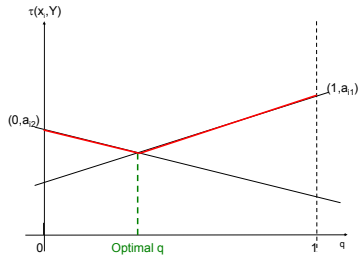


Figure 4.5
Graph showing the payoff generated by two of player I's pure strategies against each value of q

4.4.3 Special Cases

Many Lines Forming the Maximal Vertex

There will be cases where many lines intersect at the maximal vertex for player I, for example the game shown in Figure 4.6(a). Figure 4.6(b), which shows the payoff for each of player II's pure strategies against p, shows 3 lines intersecting at the maximum vertex of the lower line at p=0.4.

$$A = \begin{bmatrix} 15 & 9 & 0 \\ 0 & 4 & 10 \end{bmatrix}$$

Figure 4.6(a)
An example two-person 2x3 game

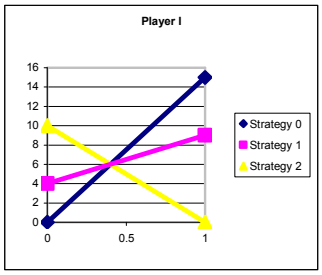


Figure 4.6(b)
Graph showing the payoff for each of player II's pure strategies against each value of p

Player I will therefore have the optimal mixed strategy X=[0.4,0.6] while player II can choose his strategy based on either of the pairs (y_1, y_2) or (y_0, y_2). The pair (y_0, y_1) cannot be used since the intersection of these lines does not form a maximal vertex in itself. If player II were to calculate an optimal strategy from these two pure strategies he would choose the pure strategy [0,1,0] as shown in Figure 4.7(c). However this would not be an optimal strategy for player II, since player I can increase his payoff against this strategy by using the strategy [1,0].

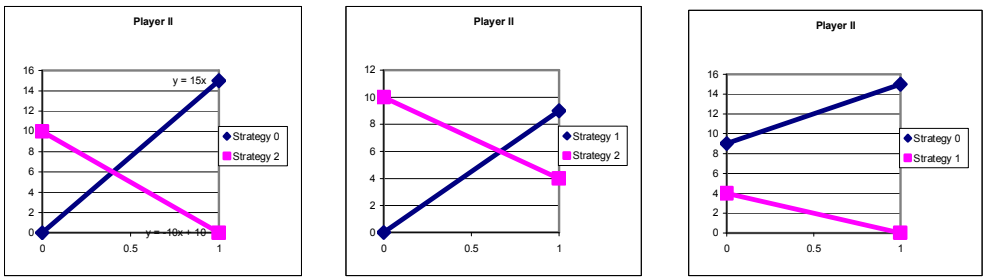


Figure 4.7(a) *Figure 4.7(b)* *Figure 4.7(c)*

Graph showing the payoff generated by each pair of pure strategies for player II

The two valid pairs give two possible optimal strategies for player II:

1. [0, ⅔, ⅓] from Figure 4.7(a)
2. [0.4, 0, 0.6] from Figure 4.7(b)

As explained in the literature survey, the simplex method examines adjacent vertices on a polytope and moves along the connecting edge to the more optimal adjacent vertex, if there is one. It was observed that if two vertices are both optimal then the edge connecting these vertices must also be optimal, giving an interval strategy.

Consider the set of all player II's pure strategies as a regular n-sided polytope with all vertices connected by an edge, where each vertex represents a pure strategy and the edge between two vertices represents a mixed strategy over only those two pure strategies that form its end points. Points within the polytope represent mixed strategies with varying probabilities over all the pure strategies, so that the central point is the strategy [$^1/_n$, $^1/_n$, $^1/_n$, …, $^1/_n$]

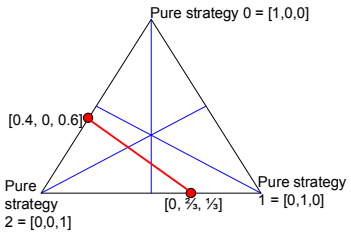


Figure 4.8

The convex hull of two optimal strategies

Figure 4.8 shows this for the example shown in Figure 4.6(a), with the three pure strategies labeled at the vertices. The blue lines show the line from the mid point of each edge to the opposite vertex. The two mixed strategies found in Figure 4.7(a) and 4.7(b) are displayed as red circles. The red line represents the convex hull of these mixed strategies which contains all of the optimal mixed strategies for player II. In this case the convex hull is simply a line since there are only two calculated optimal strategies from Figure 4.7, and any strategy that lies on the red line is also an optimal strategy.

$$A = \begin{bmatrix} 15 & 9 & 0 & 12 \\ 0 & 4 & 10 & 2 \end{bmatrix}$$

Figure 4.9(a)
An example two-person 2x4 game

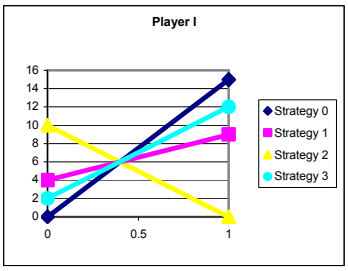


Figure 4.9(b)
Graph showing the payoff for each of player II's pure strategies against each value for p

Consider the larger 2x4 game described in Figure 4.9(a). From the graph in 4.9(b) which shows the payoff for each of player II's pure strategies against p as described above, we can see that player I has the optimal strategy [0.4, 0.6]. The strategies for player II can be found by drawing graphs for the pairs (y_0, y_2), (y_1, y_2), (y_2, y_3).
The optimal mixed strategies generated by these pairs are:

1. [0.4, 0, 0.6, 0] from the pair (y_0, y_2)
3. [0, ⅔, ⅓, 0] from the pair (y_1, y_2)
2. [0, 0, 0.5, 0.5] from the pair (y_2, y_3)

In this case the complete set of all mixed strategies for player II is a triangular based pyramid as shown in Figure 4.10. The three mixed strategies found above are marked on the edges of the pyramid with red circles, while the red triangle shows the convex hull of these three strategies and thus the entire set of player II's optimal mixed strategies.

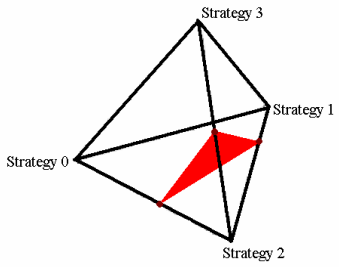


Figure 4.10
The convex hull of three optimal strategies

This theory for player II having multiple, and therefore interval, strategies can be explained in a different way by examining a 3xn game for player I. In this case the graph for calculating player I's optimal strategy increases from a 2xn game by one dimension. This means that the base of the graph is a triangle on a plane, rather than an interval on a straight line. Plotting each column on this graph again defines a plane rather than a line. It is possible for these planes to intersect so that the maximum of the lower piecewise plane is a point; for example if the planes are arranged in a pyramid shape, or the highest point is at an intersection with the walls bounding player I's base triangle. However the maximal intersection between these planes will more commonly be a line, and therefore generate an interval strategy.

Any method that finds more than one optimal strategy for a matrix game has only found the extreme cases. The complete set of all optimal strategies is the convex hull of these strategies. For 2xn games the system will, for simplicity, consider only the two lower

lines; that is, the two with the maximum and minimum gradient at the maximal intersection point.

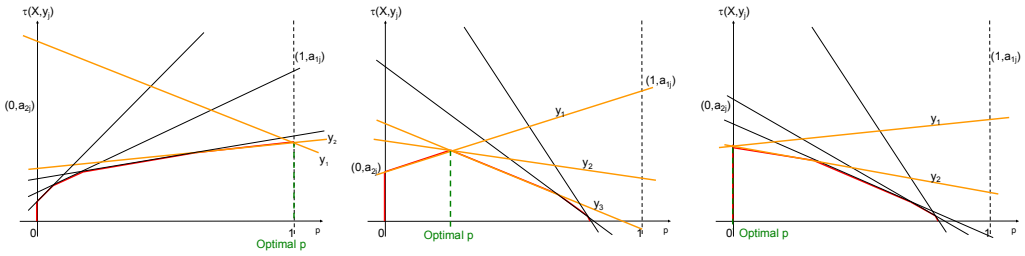


Figure 4.11(a) *Figure 4.11(b)* *Figure 4.11(c)*

Graphs showing different positions for optimal values for p, with multiple lines intersecting at the optimal vertex

In Figure 4.11(a) the two lines y_1 and y_2 (shown in orange) intersect at p=1. In this situation player I will use pure strategy x_1 while player II can use either pure strategy y_1 or y_2, or any mixed strategy that lies in the convex hull of these two pure strategies. The convex hull will be any mixed strategy where only y_1 and/or y_2 have probability greater than zero. This system will consider player II to use only the strategy that generates the line that is lowest within the interval p = [0,1] which in this case is y_2.

A similar solution is used for the situation shown in Figure 4.11(c) where two lines intersect at p=0. Again the system will consider player II to use the pure strategy that generates the line that is lowest in the interval p=[0,1] which in this case is again y_2.

In Figure 4.11(b) three lines intersect at the maximal point of the line at 0<p<1. In this situation player I will use the mixed strategy X=(p, 1-p) while player II's optimal strategy is calculated from the two lower lines; that is, the two with the maximum and minimum gradient which in this case are y_1 and y_3.

Lines Intersecting Outside the Interval [0,1]

When calculating player II's optimal strategy the game can be reduced to a 2x2 game by considering only those two lines that form the intersection point at optimal p on player I's graph as described in section 4.4.2. A graph of the payoff for player II from player I's pure strategies against q can then be drawn to calculate player II's optimal strategy. If the two lines intersect within the interval q=[0,1] as shown in Figure 4.12 then player II uses the first strategy (line 1) with probability q, and the second strategy (line 2) with probability (1-q)

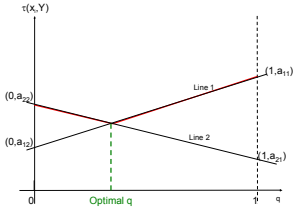


Figure 4.12

Graph showing optimal value for q against two of player I's pure strategies

There will be cases where the lines used to calculate player II's optimal strategy do not intersect within the interval q = [0,1]. This will also occur when the lines are parallel and

only intersect at infinity. In such cases it is necessary to find the upper line and examine the gradient to find the value for q.

This situation will never arise using the algorithm described in section 4.4.1. It could however occur if the system is extended to deal with multiple lines passing through the optimal vertex or interval, and it is therefore discussed here.

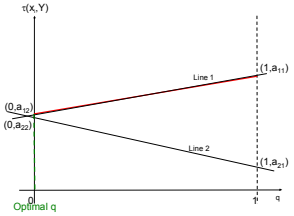


Figure 4.13(a)

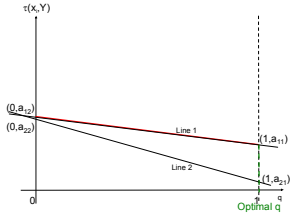


Figure 4.13(b)

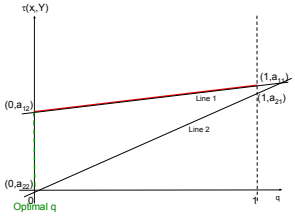


Figure 4.13(c)

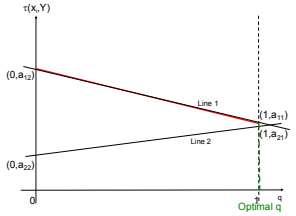


Figure 4.13(d)

Graphs showing different situations for optimal value for q of zero or one

If the two lines intersect at q≤0 or q≥1 then the optimal value for q depends on the gradient of the upper line. If the gradient is greater than zero such as in Figures 4.13(a) and 4.13(c) then the optimal value will be zero, so player II will have pure strategy two. If the gradient is less than zero such as in Figures 4.13(b) and 4.13(d) then the optimal value will be one, so player II will have pure strategy one.

If the two lines intersect at q≤0 such as in Figures 8.13(a) and 8.13(b) then the upper line is the one with the largest gradient. If they intersect at q≥0 such as in Figures 8.13(c) and 8.13(d) then the upper line is the one with the smallest gradient.

The case of either gradient being zero will only occur if player I has a pure strategy. In this case, it is clear that player II will also have a pure strategy, which is the best response to player I's. This will already have been found and therefore the situation of either line having a gradient of zero need not be considered.

4.5 Solving mx2 Games

Given an mx2 game it is possible to convert to an equivalent 2xn game by taking the transpose of the payoff matrix and multiplying each element by –1. The game can then be solved as a 2xn game as described above. After finding the solution the probability vectors X and Y should be swapped and the value of the game multiplied by –1. This means that instead of coding two very similar solutions for 2xn and mx2 games, a simpler function can be used to convert mx2 games into 2xn games, avoiding duplication of code and making the system more modular and easier to maintain.

So the mx2 matrix game $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ \dots & \dots \\ a_{m1} & a_{m2} \end{bmatrix}$ becomes $\begin{bmatrix} -a_{11} & -a_{21} & \dots & -a_{m1} \\ -a_{12} & -a_{22} & \dots & -a_{m2} \end{bmatrix}$

The optimal strategy for the second game

Player 1: $[p_1, p_2]$
Player 2: $[q_1, q_2, \dots q_m]$
Game value = v

becomes the following for the original game

Player 1: $[q_1, q_2, \dots q_m]$
Player 2: $[p_1, p_2]$
Game value = (-1)* v

4.6 Solving mxn Games

mxn games will be solved by implementing the simplex method described in section 2.8. This method may generate more than one pair of optimal strategies in which case it will be necessary to create and return an array of optimal strategies.

Initially the function will calculate the first pair of optimal strategies; that is, it will pivot until there are no negative values in the border row at the bottom of the simplex tableau. The strategy arising from this will be the first in the array.

Pivoting one step further from this tableau on a pivot element in the same row or column as a zero border element will generate a further strategy. If there are many zero border elements then each one gives one further optimal strategy which can be found by pivoting exactly one step from this matrix. The pivot steps should not be sequential; they should all start from the same matrix, so the tableau generating the first pair of optimal strategies must be stored so that it can be restored after each further pivot step. Each of these pairs will then be added to the array of optimal strategies.

5 IMPLEMENTATION

5.1 Introduction

The algorithms for solving the matrix games that the system will be presented with were examined in section 4. This section will discuss specifically how these algorithms are to be implemented, including a description of any data structures that must be defined and how these will be used. Other implementation decisions such as the choice of programming language and the reasons behind these decisions are also discussed in detail.

5.2 Programming language

There is only a limited amount of time available for the project and due to its complex mathematical nature a large proportion of this time has been spent learning and understanding the mathematics behind the algorithms before implementing them. The system should therefore be implemented in a programming language that is familiar to the developer so that a minimal amount of time is spent learning the language before implementation begins. It should be powerful enough to cope with the mathematical demands of the system, and dynamic memory allocation is desired so that the size of the games the user can input is not limited by some internal parameter. The familiarity of the project supervisor with the chosen programming language will greatly affect the amount of support available in the case of problems arising during implementation.

Taking into account the considerations above, the language chosen will be C. Both the developer and supervisor have some previous knowledge of the language, and pointers provide flexible dynamic memory allocation that can be used to implement varying size matrices. The only other viable alternative would be Java however this has less dynamic memory allocation. C is also a slightly faster language than Java since it compiles to the native machine language, rather than to an intermediate, portable byte code (Flanagan [1999]). Although this slower performance is less of an issue with later versions of Java, it could still be important for this system since there will be many computations to carry out. However this does mean that the system will suffer from portability problems and will need to be recompiled for different machine platforms.

5.3 Generating the payoff matrix

The user is expected to input the matrix in the following form:

$$m,n,[[a_{11}, a_{12}, \ldots, a_{1n}]\ [a_{21}, a_{22}, \ldots, a_{2n}]\ \ldots\ [a_{1m}, a_{1m}, \ldots, a_{mn}]]$$

or to enter data from a file by typing "*$progName :filename*" at the command prompt. The file is expected to be in the following form:

$m,$
$n,$
$[[a_{11}, a_{12}, \ldots, a_{1n}]$
$[a_{21}, a_{22}, \ldots, a_{2n}]$
$\ldots$
$[a_{1m}, a_{1m}, \ldots, a_{mn}]]$

Any additional white spaces (including tabs and carriage returns in the input file) that the user wishes to enter to make the input more readable will be accepted.

The system requires a function that will first determine whether the user is inputting the data from the command line or from a file. It must then read the input string or file and convert it into an mxn matrix of the form shown in Figure 5.1.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Figure 5.1
A generic payoff matrix

It is possible that the user may enter spaces between elements or rows instead of commas or brackets, or may forget to enter an element or row. In this case the system should report the error to the user and exit the system.

After determining where the user is inputting the data from, an output stream should be created for displaying data to the user. This will either be the screen or the file from which the data was read. If the system is outputting to a file the output will be appended to the existing data so that the file will show the input as well as the generated output.

Although C provides some support for multidimensional arrays, dynamically allocating two dimensional arrays can become complicated since they do not really exist in memory as the language makes them appear to the user.

One method is to define a pointer to pointers to doubles. Memory can then be allocated for a pointer to doubles for each row of the matrix and finally memory can be allocated for each row in the matrix. Figure 5.2 shows the code required to implement this while Figure 5.3 (Summit [2005]) depicts how the array would appear in memory.

```
double ** array
array = (double *)malloc(number_of_rows * sizeof(double *))
for (i=0; i<number_of_rows; i++) {
    array[i] = (double)malloc(number_of_columns * sizeof(double))
}
```

Figure 5.2
Code to generate a matrix

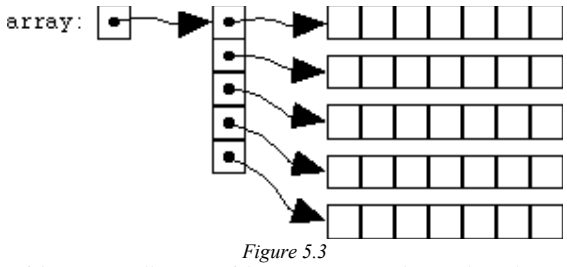


Figure 5.3
Diagram of the memory allocation of the matrix generated using the code in Figure 5.2

Each element in the matrix can now be indexed in the intuitive way so that element (i,j) is indexed by entering "array[i][j]". However this requires a lot of surplus memory, since memory is required for the pointer to each row, and accessing elements in the matrix is relatively slow. A different, faster method which is only slightly less intuitive is described below.

Instead of defining a two dimensional array, a standard one dimensional array is used. Each row of the matrix will be stored sequentially in memory, so the $(n+1)^{th}$ element (where n is the number of columns) will be the first element of the second row. Figure 5.4 shows the code required to implement this while Figure 5.5 depicts how the array would appear in memory. In Figure 5.5 m and n represent the variables number_of_columns and number_of_rows respectively.

```
double * array
array = (double)malloc(number_of_rows * number_of_columns * sizeof(double))
```

Figure 5.4
Code to generate a matrix

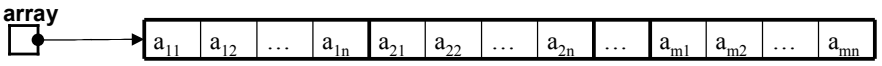


Figure 5.5
Diagram of the memory allocation of the matrix generated using the code in Figure 5.4

To index element (i,j), simply multiply i by the number of elements in each row, and add j. So "array[i][j]" becomes "array[i * number_of_columns + j]". This implementation is easily extendable up to an n-dimensional matrix; for example to access element (i,j,k) in a 3 dimensional matrix of size (d1, d2, d3), one would enter "array[i*d1*d2 + j*d1 + k]" and so on.

The second method accesses the matrix elements faster and is only slightly less intuitive to use than the first. It also requires less code to set up the matrix when it is created. It is difficult to expand the matrix; with the first method, memory can simply be reallocated for each row and new memory allocated for new rows, whereas using the second method memory must be reallocated for the whole matrix, and each element stored in its new index position. However once a matrix has been defined for the system it remains fairly static and such reallocation should not be required often. After considering the advantages and disadvantages of each method it has been decided that the second method will be used.

After regular use of the new index method any person reading or developing the code will become accustomed to translating the index method and it will become more intuitive. To aid this process, in places where either i, j, or number_of_columns is zero, the zero values will still be entered even though they make no mathematical difference to the index value. For example, the element (0, 5) will be written as "array[0 * number_of_columns + 5]" rather than "array[5]" since it is then easier to see exactly which element is being accessed when visualized as a two-dimensional array.

5.4 Data Structures

5.4.1 Doubly Linked List

A data structure will be required to store data about lines on the graph for 2xn games. An array of such data structures is not suitable since it may be necessary to discard some elements and it is not desirable to have gaps in an array where elements have been discarded, or to shift elements in an array to prevent such gaps.

Therefore the data structure will be a doubly linked list and it will require a library of functions. These functions should allow the addition and removal of items in the list as well as iterations through the list and retrieval of data stored in the list elements. The data structure will be called ColStrat and will contain all the required data for the lines as well as pointers to the previous and next elements in the list. The library has been extended from an implementation in Hall [2005].

It will contain the following variables

1. An integer *j* which will represent the line's column index in the payoff matrix.
2. A double *val0* which will represent the value of the line at p=0. This value is the value of the element in the second row of the column.
3. A double *grad* which will represent the gradient of the line. This value is the difference between the elements in the first and second rows of the column with attention paid to the sign of this value.
4. A pointer *prev* to a ColStrat item that is the previous element in the list. For the first element in the list this will be null.
5. A pointer *next* to a ColStrat item that is the next element in the list. For the last element in the list this will be null.

The following functions must be available

1. A function that creates a new, empty list. This should take no arguments and return a pointer to an empty list; that is, it returns a null ColStrat pointer.
2. A function that adds an item to the list. This should take a pointer to a ColStrat object and data required for a new ColStrat object as its arguments. A new ColStrat object containing the data provided will be inserted into the list after the ColStrat item that is passed as an argument. A pointer to the new element is returned.
3. A function that deletes an item from the list. This should take a pointer to a ColStrat object as its argument and remove it from its list. A pointer to the previous element in the list will be returned. If the deleted element was the first element in the list then a pointer to the next element is returned. If the deleted element was the only element in the list then a null pointer is returned. This will enable the deletion of the entire list from any point in the list without having to start from either end.
4. A function that moves to the next item in the list. This should take a pointer to a ColStrat object as its argument and return a pointer to the next element in the list. If the argument points to the last element in the list then it should return a null pointer.
5. A function that moves to the previous item in the list. This should take a pointer to a ColStrat object as its argument and return a pointer to the previous element in the list. If the argument points to the first element in the list then it should return a null pointer.
6. A function that moves to the first item in the list. This should take a pointer to a ColStrat object as its argument and return a pointer to the first element in the list.
7. A function that moves to the last item in the list. This should take a pointer to a ColStrat object as its argument and return a pointer to the last element in the list.

8. Functions that retrieve the data from an element. One such function will be required for each variable that may be retrieved from the data type. These functions should take a pointer to a ColStrat object as an argument and return the value of the data requested for that object.

5.4.2 Elements in the Simplex Tableau

The tableau that is used for solving mxn games using the simplex method contains many different types of elements. The top row contains text labels of player II's pure strategies, the left column contains text labels of player I's pure strategies, the bottom row and right column contain 'border values' which are floating point numbers, and the remaining interior of the tableau contains the elements of the payoff matrix. The top right, top left and bottom left elements are blank.

It is therefore necessary to have a composite data structure that can store and identify each of these types of elements. The simplex tableau will then be made up of these elements.

The data structure will contain the following variables:
1. An integer *playStrat* that identifies the type of element stored. The value will be:
   - -1 if the element is blank
   - 0 if it is an element from the payoff matrix
   - 1 if it is a label for one of player I's strategies
   - 2 if it is a label for one of player II's strategies
   - 3 if it is an element in the border row or column, including the element in the bottom right corner
2. An integer *index* that stores the index of the player's strategy. This will only be used if the value of the *playStrat* variable is 1 or 2.
3. A double *val* that stores the value of the payoff element or border element. This will only be used if the value of the *playStrat* variable is 0 or 3.

5.4.3 Returning Optimal Strategies From Functions

Many of the functions return the optimal strategies of both players and the value of the game. Since it is impossible using C to return multiple objects from a single function it is necessary to create a data structure OptStrat which will contain the optimal strategies of both players and the value of the game.

It will contain the following variables:
1. An array of doubles *XOpt* that stores player I's optimal strategy
2. An array of doubles *YOpt* that stores player II's optimal strategy
3. A double *value* that stores the value of the game
4. An integer *XInterval* that acts as a flag. The value is set to 1 if player I has an interval strategy, otherwise it is set to 0
5. An integer *YInterval* that acts as a flag. The value is set to 1 if player II has an interval strategy, otherwise it is set to 0
6. An array of doubles *XInt* that represents the end point of player I's interval values if player II has an interval strategy
7. An array of doubles *YInt* that represents the end point of player II's interval values if player II has an interval strategy

If a player has an interval strategy then his interval flag is set to 1, and his two arrays of doubles work as parallel arrays, with the *Opt* array containing the start of the interval and the *Int* array containing the end. If a player does not have an interval then his interval flag is set to 0 and only the *Opt* array is used to store the probability values.

5.5 Solving The Games

5.5.1 2x2 Games

There will be a function that accepts a 2x2 payoff matrix as its argument and returns an OptStrat object containing the optimal strategies and the value of the game. The function will first attempt to find a saddle point by implementing the algorithm described in section 4.3.1. If one exists then the pure strategies and the resulting value of the game will be returned. If no saddle point over the pure strategies exists then the function will calculate and return the optimal mixed strategy for each player by implementing the algorithm described in section 4.3.2.

5.5.2 2xn Games

There will be a function that accepts a 2xn payoff matrix as its argument and returns the optimal pure or mixed strategy for each player and the value of the game in an OptStrat object. The strategies will be found by implementing the algorithms described in section 4.3.

5.5.3 mx2 Games

Transforming the mx2 Game into the 2xn Game

Once the payoff matrix has been created it is easy to transpose it. There will be a function that accepts an mx2 payoff matrix as its argument and returns a 2xn matrix, with n=m, which is the negative transpose of the original.

Transforming the 2xn Solution into the mx2 Solution

There will be a function that accepts a pair of optimal strategies and the value of the game for a 2xn game. It will swap the two strategies and multiply the value of the game by -1. The result will be the pair of optimal strategies and the value of the game for the corresponding mx2 game with n=m that was transposed using the function above.

5.5.4 mxn Games

If there are any negative values in the payoff matrix then the absolute value of the smallest element should be added to all values of the payoff matrix. This will guarantee that all elements are positive and therefore that the value of the game is positive. This will not affect the calculation of the optimal strategies but it will increase the value of the game by the amount added. The value added must therefore be subtracted from the calculated value of the game before the output is displayed to the user.

The simplex tableau created as described in section 2.8 contains different types of elements and data types. It is therefore necessary to define a compound data structure to store all of the required information and build the tableau from this structure as described in section 5.4.2. The tableau can then be created using the payoff matrix, the players'

strategies, a border column and row of 1's and -1's respectively along the right side and bottom of the tableau, and a zero in the bottom right corner as shown in Figure 2.15.

The system will examine the border row at the bottom of the tableau for a column with a negative element. Once such a column has been found, the positive element in this column that gives the smallest ratio between itself and the value in the border column at the end of its row is found. This will be the pivot element.

The values of the tableau must be stored in a temporary tableau before the pivot is carried out so that the new elements are calculated using the old values. Once the tableau has been pivoted as described in section 2.8 step 4, this temporary tableau can be deleted.

The label on the left of the pivot row must then be swapped with the label on the top of the pivot column.

This pivoting is repeated until there are no negative elements in the border row at the bottom of the tableau. Once there are no negative values in this row a solution for the game can be found as described in section 2.8 step 7.

If there are any elements in the border row or column that are zero then further strategies can be found by pivoting one step further on an element in that column or row respectively. The pivot element must be positive and give the smallest ratio between itself and the value in the border column at the end of its row over all such ratios for elements in its column. Pivoting with this element must also not cause any elements in the border row or column to become negative.

5.6 Main Function

The main function will call the functions above to solve the game. The first step is to generate the payoff matrix and store the values for m and n. If the game is an mx2 game then it will call the function to transform it into a 2xn game.

A function to solve the game is now called depending on the values of m and n. After the game has been solved, if the original game was an mx2 game, the function is called to reverse the conversion to a 2xn game by swapping the strategies and multiplying the value by -1.

The strategies are then printed to the screen or input file.

To make the main program more readable and maintainable as well as to allow for future extensions, each of the functions for solving the game will return a pointer to an OptStrat object rather than returning the object itself. This is only really necessary for the function for solving mxn games since this is the only one that may return more than one pair of strategies However it allows each function to have a similar interface as well as making it easier to return arrays of OptStrat objects if these functions are modified to find multiple optimal strategies.

6 TESTING

6.1 Introduction

The testing can be divided into two main sections.

White box testing involves examining the code and performing code walkthroughs to verify that it is correct. This is clearly a time consuming process as it is difficult to examine every path through the code, therefore it is only suitable for small sections of code or specific areas of testing such as memory allocation as described in section 6.2.2.

Black box testing involves using test data to verify that the program gives the correct output for a given input. It does not verify that the output was generated correctly; therefore it could miss some errors that only arise in certain situations when the wrong path through the code is executed. It is therefore important to create test cases that generate extreme or boundary situations to test the limits of the system.

Appendix section A1.1 details which section and where appropriate which test criteria within that section test each requirement.

6.2 White Box Testing

6.2.1 Creating The Doubly Linked List

A code walkthrough was used to verify that the generation of the linked list using the addItem() function was correct.

Adding the first item to an empty list

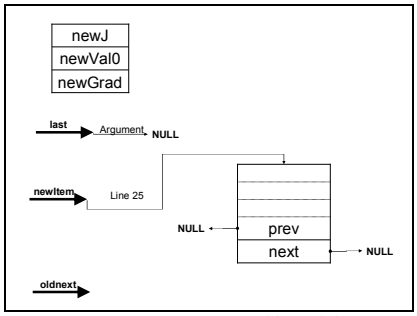


Figure 6.1(a). Demonstrating addItem() line 25

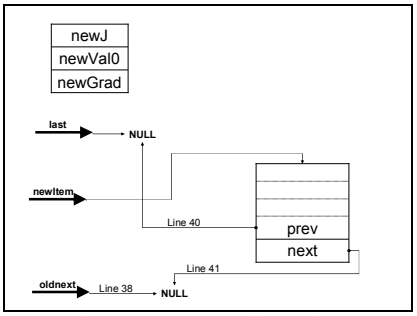


Figure 6.1(b). Demonstrating addItem() lines 26-44

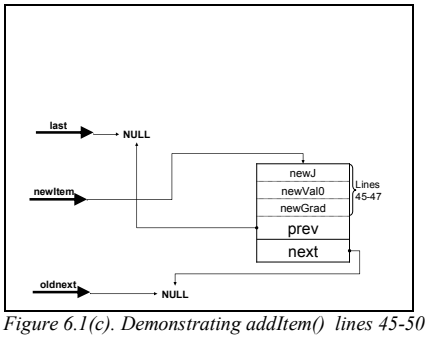


Figure 6.1(c). Demonstrating addItem() lines 45-50

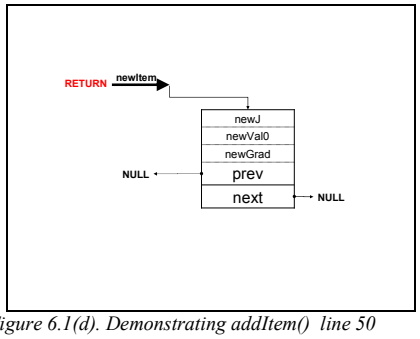


Figure 6.1(d). Demonstrating addItem() line 50

In Figure 6.1(a), the first time addItem() is called, the pointer *last* is a null pointer. In line 25 a new item is created that holds no data.

Due to the *last* pointer pointing to NULL, lines 31-36 are not executed. Instead, line 38 sets the *oldnext* pointer to NULL as shown by Figure 6.1(b). This diagram also shows line 40 which sets the *prev* pointer of the new item to point to the object that *last* is pointing to, which is NULL in this case. Line 41 sets the *next* pointer of the new item to point to the object that *oldnext* is pointing to, which again is NULL in this case.

Figure 6.1(c) shows lines 45-47 which set the data for the new item from the data passed to the function as arguments.

In line 50 the function returns a pointer to the same object that *newItem* is pointing to as shown by Figure 6.1(d).

Adding the second item to the end of the list

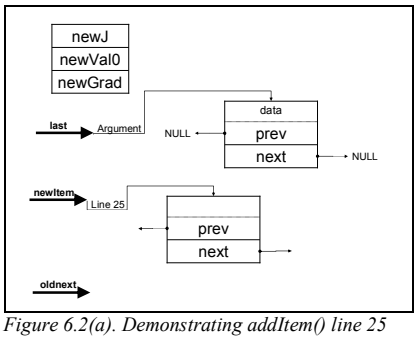


Figure 6.2(a). Demonstrating addItem() line 25

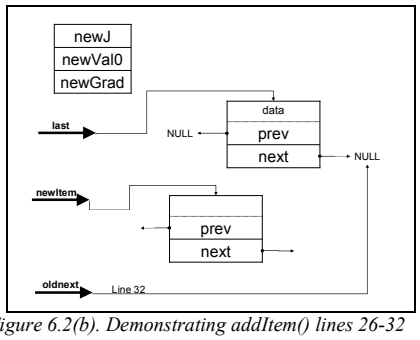


Figure 6.2(b). Demonstrating addItem() lines 26-32

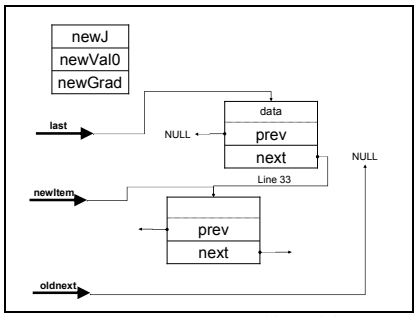


Figure 6.2(c). Demonstrating addItem() line 33

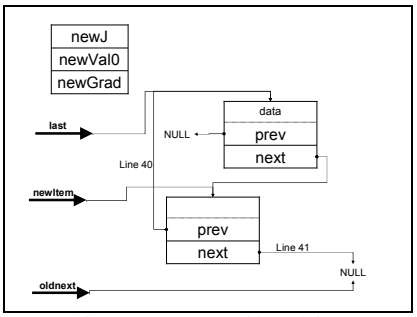


Figure 6.2(d). Demonstrating addItem() lines 34-44

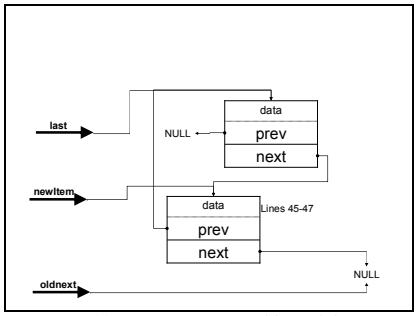


Figure 6.2(e). Demonstrating addItem() lines 45-49

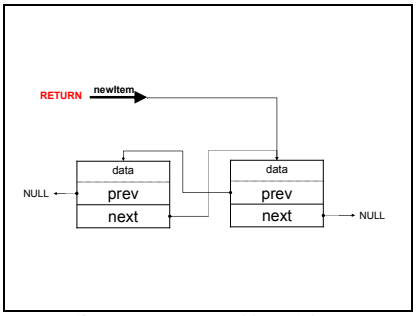


Figure 6.2(f). Demonstrating addItem() line 50

The second time addItem() is called, *last* points to the only item in the list. In line 25 a new item is created that holds no data as shown by Figure 6.2(a).

This time *last* is not pointing to NULL so lines 31-36 are executed. Figure 6.2(b) displays line 32 in which *oldnext* is set to point to the object pointed to by the *next* pointer of the object pointed to by *last*, which in this case is NULL.

Figure 6.2(c) shows line 33 where the *next* pointer of the object pointed to by *last* is set to point to the new item.

Because *oldnext* points to NULL, line 36 is not executed. Figure 6.2(d) shows line 40 setting the *prev* pointer of the new item to point to the object that *last* is pointing to, which is the only other element in the list. It also shows line 41 setting the *next* pointer of the new item to point to the object that *oldnext* is pointing to, which is still NULL in this case.

Figure 6.2(e) shows lines 45-47 which set the data for the new item from the data passed to the function as arguments.

Line 50 returns a pointer to the same object that *newItem* is pointing to, which is the item that has just been added to the list as shown in Figure 6.2(f).

Adding an item to the middle of a list

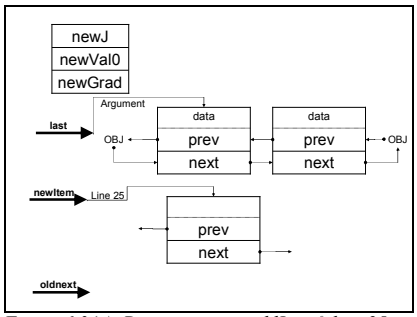


Figure 6.3(a). Demonstrating addItem() line 25

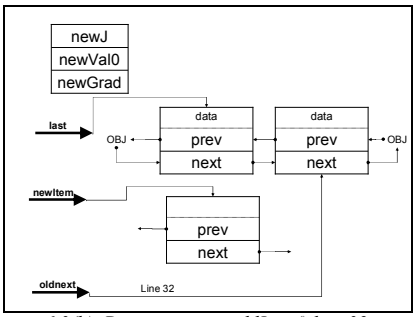


Figure 6.3(b). Demonstrating addItem() line 32

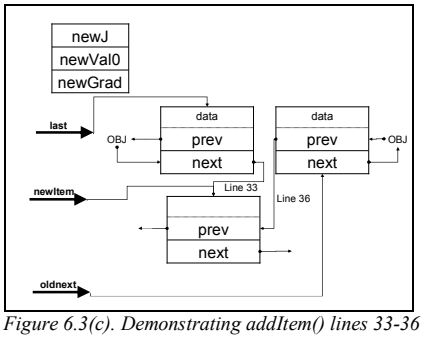


Figure 6.3(c). Demonstrating addItem() lines 33-36

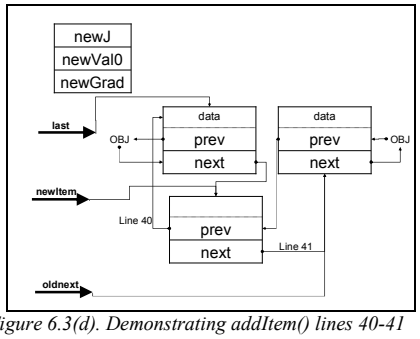


Figure 6.3(d). Demonstrating addItem() lines 40-41

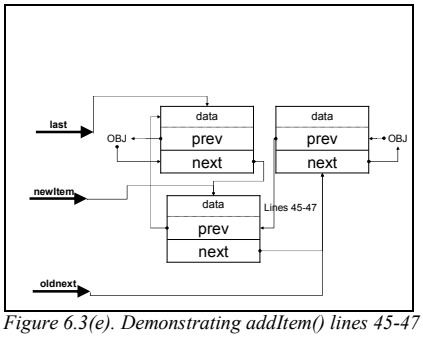


Figure 6.3(e). Demonstrating addItem() lines 45-47

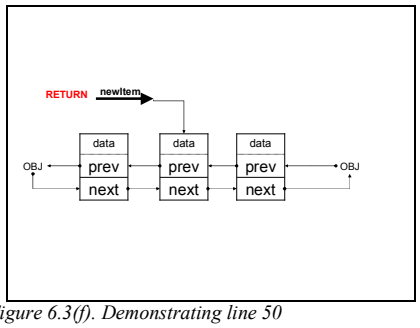


Figure 6.3(f). Demonstrating line 50

Suppose we want to add an item in the middle of a list. The pointer *last* points to the object after which we want to place the new object. In line 25 a new item is created that holds no data as shown in Figure 6.3(a).

Figure 6.3(b) shows line 32 where *oldnext* is set to point to the object pointed to by the *next* pointer of the object pointed to by *last*, which in this case is the object that will follow the new object once it has been placed in the list.

In line 33 the next pointer of the object pointed to by *last* is set to point to the new item as shown in Figure 6.3(c). *Oldnext* does not point to null, so line 36 is executed. It sets the *prev* pointer of the object pointed to by *oldnext* to point to the new item, again shown in Figure 6.3(c).

Figure 6.3(d) shows line 40 which sets the *prev* pointer of the new item to point to the object that *last* is pointing to; this is the object that will precede the new item in the list. This diagram also shows line 41 which sets the *next* pointer of the new item to point to the object that *oldnext* is pointing to; this is the object that will follow the new item in the list.

Lines 45-47 set the data for the new item from the data passed to the function as arguments as shown in Figure 6.3(e).

Line 50 returns a pointer to the same object that newItem is pointing, to which is the item that has just been added to the list as shown in Figure 6.3(f). The item has been successfully added between the two items, following the object that was originally passed to the function as the *last* argument

6.2.2 Memory Allocation

The code was examined to verify that every reservation of memory using either the calloc() or malloc() statements was accompanied by a free() statement as soon as the memory block was no longer required, including immediately before any intermediate exits from functions due to unforeseen errors. The results of this code examination can be found in the appendix section A1.2

6.2.3 Reporting of System Errors

In accordance with requirement 4.1, while the report of errors generated due to the incorrect input of the payoff matrix by the user are output to the output stream, system errors such as memory allocation errors should always be reported to the screen, enabling the user to see system errors immediately. Test cases that generate such problems cannot be formulated but code inspection verifies that all error messages generated due to system errors are printed to the screen using *printf* rather than to the output stream using *fprintf*.

6.2.4 Function Testing

Each function was tested throughout implementation by outputting statements and calculated values to the screen at different stages of the functions' computations. Examining the output together with the code enabled the author to verify that the system was taking the correct path through the code and that the computations were correct.

6.3 Black Box Testing

All black box testing that involves inputting a valid mxn game where $m \leq 5$ and $n \leq 5$ will be tested against an online game theory utility (Warner and Costenoble [1997b]). In the case where there are multiple optimal strategies this utility may return a different optimal strategy to this system. If this is the case then it should first be verified that both systems give the same value for the game. If this test does not fail further examination of the optimal strategy calculated by the system will be required to verify that is satisfies the inequality shown in equation 2.9.

6.3.1 Testing the Linked List Library

Test code was written to test the linked list library functions. This involved creating a list using the newList() and addItem() functions; iterating forward and backward through the list using the moveNext(), movePrev(), moveFirst() and moveLast() functions; and deleting items using the discItem() function. The status of the list was printed at various stages of the computation to verify that it was correct. All functions were found to work correctly. The source code for this can be found in the appendix section A3.10 and the output in A1.3. The source code and the executable file can also be found on the attached CD in the folder "Testing/Section 6-3-1".

6.3.2 Testing the Input Method

The system should be able to receive input from the command line or from a file. All output should be directed to the place that the input came from.

Several games were entered from both the command line and a file. The output to the screen and file in each case was examined and verified to be identical. The complete

results can be found in the appendix section A1.4. A copy of all the input files can be found on the attached CD, in the folder "Testing\Section 6-3-2\Input Files", and a corresponding copy of all the output files in the folder "Testing\Section 6-3-2\Output Files".

Since this test verified that entering data from files generated identical output to entering data from the command line, all further tests that required entering test data were input from files. This allowed the input and output in each case to be stored easily and examined together to verify its correctness.

6.3.3 Testing Invalid Inputs

Different invalid games were entered to verify that any invalid matrix game would not be accepted, and a useful error message would be reported to the user. Tests were devised for each of the following different ways in which the matrix can be invalid. The games and their generated output can be found in the appendix section A1.5. All tests generated a useful error message which was appended to the file containing the input as can be seen in the appendix section A1.5. A copy of all the input files can be found on the attached CD, in the folder "Testing\Section 6-3-3\Input Files", and a corresponding copy of all the output files in the folder "Testing\Section 6-3-3\Output Files".

The criteria tested were:

1. Entering an incorrect value for m
    1.1. A negative value
    1.2. A floating point number
    1.3. A valid, incorrect value for m (incorrect for the actual matrix)
2. Entering an incorrect value for n
    2.1. A negative value
    2.2. A floating point number
    2.3. A valid, incorrect value for n (incorrect for the actual matrix)
3. Omitting the required commas and brackets
4. Omitting single elements from the matrix (Omission of whole rows or columns is the same as entering valid, incorrect value for m or n respectively)

6.3.4 Testing Different Types of Games

Different types of solutions; for example games with saddle points in various positions, mixed strategies, interval strategies and so on, will be tested for each type of game size; that is 2x2, 2xn, mx2 and mxn games. The games and their generated output can be found in the appendix section A1.6. A copy of all the input files can be found on the attached CD, in the folder "Testing\Section 6-3-4\Input Files", and a corresponding copy of all the output files in the folder "Testing\Section 6-3-4\Output Files".

All tests generated the expected output, with only one unusual result found in test 6. The input and output for this test is shown in Figure 6.4.

Input:

```
2,2,[
[-1.5,0.5]
[1.5,-0.5]]
```

Output:

```
Payoff matrix :=
[-1.500000, 0.500000]
[1.500000, -0.500000]

There is 1 strategy
Strategy 1:
Player I  Optimal Strategy: [0.500000, 0.500000]
Player II Optimal Strategy: [0.250000, 0.750000]
Value of the game: -0.000000
```

Figure 6.4
Input and output from A1.6 test 6

The value of the game is shown to be -0.000000. This is probably caused by a rounding anomaly due to the use of floating point values. Rounding errors such as these are also present in existing software solutions; when the game shown in Figure 6.5 is entered into the online utility used to verify test results (Warner and Costenoble [1997a]) it returns the output shown in Figure 6.6. The rounding error on the column strategy (player II's strategy) has caused a rounding error on the value of the game. The correct output that this utility should return is shown alongside Figure 6.6 in Figure 6.7

$$A = \begin{bmatrix} 15 & 9 & 0 \\ 0 & 4 & 10 \end{bmatrix}$$

Figure 6.5
2x3 game generating an incorrect result on the online utility

```
The optimal column strategy is:
[0 0.66666666 0.33333334 0 0 ]
The optimal row strategy is:
[0.4 0.6 0 0 0 ]
The value of this game is 5.99999999.
```

Figure 6.6
The result generated on the online utility from the game in Figure 6.5

```
The optimal column strategy is:
[0 0.66666667 0.33333333 0 0 ]
The optimal row strategy is:
[0.4 0.6 0 0 0 ]
The value of this game is 6.
```

Figure 6.7
The correct result from the game in Figure 6.5

The criteria to be tested are listed below:

2x2 Games

1. Test with games that have a saddle point in each position.
2. Test with games that have optimal strategies X = (p, 1-p) and Y = (q, 1-q) where p and q have the values shown in Table 6.1.

Test	p	q
2.1.	< 0.5	< 0.5
2.2.	= 0.5	< 0.5
2.3.	> 0.5	< 0.5
2.4.	< 0.5	= 0.5
2.5.	= 0.5	= 0.5
2.6.	> 0.5	= 0.5
2.7.	< 0.5	> 0.5
2.8.	= 0.5	> 0.5
2.9.	> 0.5	> 0.5

Table 6.1
Testing combinations of p and q

2xn Games

3. Test with games that have saddle points in the four corners of the matrix and in the middle.
4. Test with games where Player II has a pure strategy at various different column positions including the two end columns, and Player I has a mixed strategy. (Note that Player I's mixed strategy must be an interval strategy for Player II to have a pure strategy in response to it.)

Games where Player II has a mixed strategy and Player I has a pure strategy will not be found with this system due to the algorithm used. For further details see section 4.4.

5. Test with games where both players have mixed strategies, with Player II's distributed across various different pure strategies.

mx2 Games

Test all games used for 2xn games in their equivalent mx2 form. Verify that the output for these games is equivalent to the equivalent 2xn game.

6. Test with games that have saddle points in the four corners of the matrix and in the middle.
7. Test with games where Player I has a pure strategy at various different row positions including the two end rows, and Player II has a mixed strategy. (Note that Player II's mixed strategy must be an interval strategy for Player I to have a pure strategy in response to it.)
8. Test with games where both players have mixed strategies, with Player I's distributed across various different pure strategies.

mxn games

9. Test with games that have saddle points in the four corners of the matrix and in the middle.
10. Test with games where both players have mixed strategies across various different pure strategies.
11. Test with games where there are multiple optimal strategies.

All games

12. Test games with values that are
    12.1. positive
    12.2. negative
    12.3. zero
13. Test games with values that are
    13.1. integer
    13.2. floating point
14. Test games with elements that are
    14.1. positive
    14.2. negative
    14.3. zero
15. Test games with elements that are
    15.1. integer
    15.2. floating point

7 CONCLUSION

7.1 Analysis of System and Areas for Further Development

The system meets all of the essential requirements defined in section 4 as well as some of the desirable, non essential ones. In this sense the system has been successful however there are many areas in which it can be improved.

The development of my own algorithm for solving 2xn games allowed the project to be developed incrementally and have a possible earlier end point, should unforeseen events have meant that the project could not be finished. However the algorithm was not perfected and subjected to rigorous mathematical analysis by those accustomed to developing algorithms before its implementation. Therefore problems were encountered during implementation due to the algorithm being not so finely 'polished' as one that has been analysed and criticised by leaders in the field. Were I to do the project again I would implement a minimax method to search all games, including mxn, for saddle points before implementing the simplex method for all games greater than 2x2 if no saddle point were found. This would allow as an earlier end point a system that only returns a saddle point, if one exists for the game. However the development of the algorithm did increase my knowledge and understanding of the problem. It also made it easier to learn and understand the simplex method for larger games and was therefore still a worthwhile part of the project.

It is misleading to state there are a finite number of strategies when there are multiple strategies. The total set of optimal strategies is the convex hull of these strategies, therefore the system should be developed to output this information and ensure that the user can interpret the results so that he can select an optimal strategy from this set. It is assumed that the user has some knowledge of game theory and the structure of optimal strategies; however this is more advanced knowledge and therefore it cannot reasonably be assumed that the user is familiar with it. It may be possible to incorporate some sort of explanation of this concept into a graphical user interface, so that the user could correctly interpret the finite list of optimal strategies.

The statement regarding the number of optimal strategies would still be misleading even if the user knew how to interpret the case of multiple optimal strategies. For 2x2, 2xn and mx2 games the system is only designed to find one optimal strategy, although for 2xn and mx2 games this could include an interval strategy for one player. The output states that "*There is 1 strategy"* when there may be more than one; the system has just not found them. More strategies could be found for 2xn and mx2 games if the simplex algorithm were implemented to find these strategies, rather than using a separate algorithm. However even the simplex algorithm does not find all optimal strategies as discussed below.

The simplex method does return multiple optimal strategies however it does not always calculate all of the optimal strategies that one would expect. Any strategy is an optimal strategy for the game described in Figure 7.1; each player can play any strategy with any probability and the value of the game will always be one. The expected output from the system would be every combination of both players' pure strategies; that is, 25 strategy pairs. The user should then interpret the total set of optimal strategies as the convex hull

of these 25 strategies. However Figure 7.2, which shows the output from the system for this game, shows that while the system returns all of player II's pure strategies, only one of player I's is shown; making only five strategy pairs in total.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 7.1
A game that does not generate the complete output from the system

```
amos $ solveGame 5,5,[[1,1,1,1,1][1,1,1,1,1][1,1,1,1,1][1,1,1,1,1][1,1,1,1,1]]

Payoff matrix :=
      [1.000000, 1.000000, 1.000000, 1.000000, 1.000000]
      [1.000000, 1.000000, 1.000000, 1.000000, 1.000000]
      [1.000000, 1.000000, 1.000000, 1.000000, 1.000000]
      [1.000000, 1.000000, 1.000000, 1.000000, 1.000000]
      [1.000000, 1.000000, 1.000000, 1.000000, 1.000000]

There are 5 strategies
Strategy 1:
  Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000]
  Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000]
  Value of the game: 1.000000

Strategy 2:
  Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000]
  Player II Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000, 0.000000]
  Value of the game: 1.000000

Strategy 3:
  Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000]
  Player II Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000, 0.000000]
  Value of the game: 1.000000

Strategy 4:
  Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000]
  Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000, 0.000000]
  Value of the game: 1.000000

Strategy 5:
  Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000]
  Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000, 1.000000]
  Value of the game: 1.000000
```

Figure 7.2
Input and output for the system, for the game in Figure 7.1

Rounding or other internal C errors sometimes mean that zero is displayed as -0.0000 even if the true figure is exactly zero, as shown in test 6 of section 6.3.4. This caused problems when implementing the simplex method, since to find multiple strategies it is necessary to test for zero values in the border row and column of the simplex tableau. This was solved by testing within an interval rather than testing for an exact value of zero in the function findZeroBorder() in the file simplex.c, which can be found in the appendix

section A3.6. The size of the interval was determined by the accuracy of the printed numbers; the interval was bound by $1E^{-6} \pm \frac{1E^{-6}}{2}$ because the resolution of the printed numbers was $1E^{-6}$. This could be rectified in some cases if the system was developed to accept fractional numbers as stated in requirement 1.6.

Even without extending the scope of the system to non zero-sum or dynamic games, there are several areas in which the system could be extended. An obvious area for extension is the calculation of all optimal vertices on the polytope which represents the strategies for 2xn and mx2 games. This would involve implementing the case for 2xn games where multiple lines intersect at the maximal vertex on player I's graph as discussed in section 4.4.3. A method would also need to be investigated for communicating to the user the nature of the set of solutions in these cases. The simplex method implemented for solving mxn games must also be refined so that all strategies forming optimal vertices on the polytope for mxn games are discovered.

Provisions have been made for these extensions to 2xn and mx2 games; all functions for solving games can return an array of OptStrat objects rather than just a single one so the interface of these functions with the main function would not be required to change. The function for reversing the transformation of an mx2 game to a 2xn game is also written so that it can change an array of strategies rather than just a single one.

The current method of inputting the payoff matrix is awkward and prone to errors. It is difficult to see the structure of the matrix when inputting from the command line although this is solved to a certain extent by inputting the matrix from a file. A more user friendly graphical user interface could be designed for the system so that it would be easier to view the matrix in its standard form; that is, in the form shown in Figure 7.3(a) rather than that shown in Figure 7.3(b). A good design could also make it impossible to omit or enter erroneous operators such as square brackets and commas, but should still allow the user to enter any arbitrary size matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}$$

Figure 7.3(a)
Desired form of user input for future systems

$$A = m, n, [[a_{11}, a_{12}, \dots, a_{1m}][a_{21}, a_{22}, \dots, a_{2m}]..[a_{n1}, a_{n2}, \dots, a_{nm}]]$$

Figure 7.3(b)
Current form of user input

7.2 Personal Conclusion

The system perhaps suffered from a lack of planning in some sections; this was due mainly to the unrefined algorithm for solving 2xn games, as discussed above. In hindsight I can also see that I had a relatively poor knowledge of some of the finer points of the programming language. Although I began the project with a basic knowledge of C and good programming ability in general, I feel that my awareness and understanding of the

language has increased tremendously throughout the project as I became more familiar with it, to the point that I began to predict how functions such as *fprintf()* could be used to output to the screen, rather than using an *if* statement to decide whether to use *printf()* or *fprintf()* to output to the screen or file.

In general I think my time management was good. The requirement to have finished the literature survey before Christmas ensured the project got off to a good start and was not neglected until closer to the deadline. There was possibly more spare time that could have been utilised in the first semester of the course which would have relieved some of the pressure nearer the end of the project. However I did not fall too far behind my project plan, and managed to keep within my own set timescale which had accounted for some contingency time.

Through the project I have gained a lot of technical knowledge in the area of game theory, in particular finite two-person zero-sum games. I have also improved my practical abilities in designing and programming a working system, and am much more familiar with the C programming language. Areas of my working style that are good or those that need improvement have also been brought to my attention, and I will be able to apply this in future projects at university and at work.

8 REFERENCES

BARRY, A. M. 2004. *The Project Dissertation*. Bath, Somerset: University of Bath. Available from: http://www.cs.bath.ac.uk/~amb/CM30076/dissertations/ProjectDissertation.pdf [Accessed May 2005]

BERNHEIM, D. 1984. Rationalizable strategic behaviour. *Econometrica* 52:1007-1028

BINMORE, K. 1992. *Fun and Games: A Text on Game Theory*. Lexington: D. C. Heath and Company.

CAMERER, C. 2003. *Behavioural Game Theory.* New Jersey: Princeton University Press.

DELORIE, D.J. 2001. *The GNU C Library* [online]. Boston, MA: Free Software Foundation, Inc. Available from: http://www.delorie.com/gnu/docs/glibc/libc.html [Accessed April 2005]

Dictionary.com [online]. Los Angeles, CA: Lexico Publishing Group. Available from: http://dictionary.reference.com/search?q=game [Accessed November 2004]

FERGUSON, T. *Game Theory* [online]. Los Angeles, CA: University of California. Available from: http://www.math.ucla.edu/~tom/Game_Theory/Contents.html [Accessed November 2004]

FLANAGAN, D. 1999. *JAVA IN A NUTSHELL*. 3rd ed. Sebastopol, CA: O'Reilly & Associates, Inc.

FOURER, R. 2000. *Linear Programming Frequently Asked Questions* [online]. Evanstan, IL: Optimization Technology Center of Northwestern University and Argonne National Laboratory. Available from:http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html [Accessed Nov 2004]

FUDENBURG, D. and TIROLE, J. 1991. *Game Theory*. Cambridge, MA: MIT Press

HALL, P.M. 2005. *notes* [online]. Bath, Somerset: University of Bath. Available from http://staff.bath.ac.uk/maspmh/comp0001/notes.pdf. [Accessed January 2005]

HARGREAVES HEAP, S. and VAROUFAKIS, Y. 2004. *Game Theory: A critical text*. 2nd ed. London: Routeledge.

HARSANYI, J. 1973. Oddness of the number of equilibrium points: A new proof. *International Journal of Game Theory* 2:235-250

NASH, J. 1950. Equilibrium points in *n*-person games. *Proceedings of the National Academy of Sciences* 36:48-49

OSBORNE, J. 2004. *an introduction to Game Theory*. New York: Oxford University Press.

OWEN, G., 1982. *Game Theory.* 2nd ed. London: Academic Press.

PEARCE, D. 1984. Rationalizable strategic behaviour and the problem of perfection. *Econometrica* 52:1029-1050

PFEIFER, G and JOHNSON, J. 2005. *GCC Home Page – GNU Project – Free Software Foundation (FSF)* [online]. Boston, MA:Free Software Foundation, Inc. Available from: http://gcc.gnu.org/ [Accessed April 2005]

REVELIOTIS, S. *An Introduction to Linear Programming and the Simplex Algorithm* [online]. Atlanta, GA: Georgia Institute of Technology. Available from: http://www.isye.gatech.edu/~spyros/LP/LP.html. [Accessed Nov 2004]

SAVANI, R. *Solving a Bimatrix Game* [online]. London: The London School of Economics and Political Science. Available from: http://banach.lse.ac.uk/form.html [Accessed November 2004]

SHOR, M. 2003. *Game Theory and Business Strategy: Normal Form game Solver Applet* [online]. Nashville, TN: Vanderbilt University. Available from: http://www.gametheory.net/Mike/applets/NormalForm/NormalForm.html [Accessed Nov 2004]

SUMMIT, S. 1995. *C Programming Notes*[online]. Montevideo, Uraguay: Universidad de la Republica. Available from: http://www.eumus.edu.uy/eme/c/c-notes_summit/intermediate/sx9.html [Accessed April 2005]

WARNER, S and COSTENOBLE, S. 1997a. *Summary of Chapter Topic:Game Theory* [online]. Available from: http://people.hofstra.edu/faculty/Stefan_Waner/RealWorld/Summary9.html [Accessed April 2005]

WARNER, S and COSTENOBLE, S. 1997b. *Game Theory Simulation* [online]. Hempstead, NY: Hofstra University. Available from: http://people.hofstra.edu/faculty/Stefan_Waner/RealWorld/gametheory/games.html [Accessed April 2005]

Wikipedia, The Free Encyclopedia[online]. Hempstead, NY: Hofstra University. Available from: http://en.wikipedia.org/wiki/Main_Page [Accessed November 2004]

APPENDICES

A1. TESTING

A1.1 Mapping of Requirements to Test Plan

This section details which subsection of section 6 and, if appropriate, which test criteria within that subsection are used to verify that each requirement has been met

Requirement number	Test section/criteria	Requirement met?
1.1	Section 6.3.3	✓
1.2		✓
1.3	All test cases that input a valid payoff matrix	✓
1.4	Section 6.3.5, criteria 14	✓
1.5	Section 6.3.5, criteria 15	✓
1.6	Not implemented	×
2.1	Section 6.3.3	✓
2.2		✓
2.3	All test cases that input a valid payoff matrix	✓
2.4	Section 6.3.5, criteria 11	✓
2.5	Section 6.3.5, criteria 4, 7	✓
2.6.1	Section 6.3.5, criteria 1,3,6, 9 for integers; criteria 2,4,5,7,8,10 for floating point numbers	✓
2.6.2	Not implemented	×
2.6.3	All tests that input a valid payoff matrix	✓
2.6.4	Section 6.3.5, criteria 13	✓
2.6.5	Not implemented	×
2.6.6	Section 6.3.5, criteria 12	✓
3	All tests that input a valid payoff matrix	✓
4.1	Section 6.3.2	✓
4.2		✓
5	Section 6.3.4, criteria 1,2,3,4	✓

A1.2 Memory Allocation Code Walkthrough

Due to the flexibility of the system there are many stages at which memory is allocated, reallocated and freed. To ensure that the system does not reserve too much memory while it is running it is necessary to ensure that memory is freed once it is no longer needed.

To verify this involves examining the code and ensuring that each block of memory that is allocated is freed once it is no longer needed. The table below matches each allocation of memory with its 'free' statement.

In some cases the memory will be allocated and freed in separate functions. In this case the allocation of memory in the first function is matched with the freeing of the memory in the second function, and the calling of the first function to allocate the memory is matched with the calling of the second function to free the memory. It is also possible that an abnormal exit from a function can occur before the memory is freed. In this case it is necessary to free any memory that will not be required due to the abnormal exit; such 'intermediate exits' are listed in the far right column of the table.

The reader may note that the variable returnValue is not freed after the possible intermediate exit of the function addStrategy() in simplex.c on lines 305 and 309. This function adds another strategy to the end of the array, so there may be other strategies that have been successfully added previously. It is therefore unadvisable to free this memory, since a higher function may still wish to access the previous strategies.

Allocation of Memory				Freeing of Memory				Intermediate exits	
Filename	**Function name**	**Variable name**	**Line allocated**	**Filename**	**Function name**	**Variable name**	**Line freed**	**Filename**	**Line freed**
LinkListLib.c	addItem()	newItem	25	LinkListLib.c	discItem()	old	71		
strategies.c	get2by2Strat()	returnValue	119	functions.c	freeOptStratArray()	(array+i)	210	strategies.c	125, 130
strategies.c	get2by2Strat()	returnValue.XOpt	123	functions.c	freeOptStratArray()	(array+i)->XOpt	206	strategies.c	131
strategies.c	get2by2Strat()	returnValue.YOpt	128	functions.c	freeOptStratArray()	(array+i)->YOpt	207		
strategies.c	get2byNStrat()	returnValue	275	functions.c	freeOptStratArray()	(array+i)	210	strategies.c	281, 287, 294, 302, 469
strategies.c	get2byNStrat()	returnValue.XOpt	279	functions.c	freeOptStratArray()	(array+i)->XOpt	206	strategies.c	286,292, 299, 465
strategies.c	get2byNStrat()	returnValue.YOpt	284	functions.c	freeOptStratArray()	(array+i)->YOpt	207	strategies.c	293, 300, 466
strategies.c	get2byNStrat()	returnValue.XInt	290	functions.c	freeOptStratArray()	(array+i)->XInt	208	strategies.c	301, 467
strategies.c	get2byNStrat()	returnValue.YInt	297	functions.c	freeOptStratArray()	(array+i)->YInt	209	strategies.c	468
strategies.c	get2byNStrat()	list	310	strategies.c	get2byNStrat()	list	337, 514	strategies.c	472
simplex.c	pivotMatrix()	tempMatrix	178	simplex.c	pivotMatrix()	tempMatrix	222		-
simplex.c	addStrategy()	returnValue	296, 301	functions.c	freeOptStratArray()	(array+i)	210		
simplex.c	addStrategy()	newValue.XOpt	308	functions.c	freeOptStratArray()	(array+i)->XOpt	206	simplex.c	314
simplex.c	addStrategy()	newValue.YOpt	312	functions.c	freeOptStratArray()	(array+i)->YOpt	207		
simplex.c	getMbyNStrat()	matrix	366	simplex.c	getMbyNStrat()	matrix	447	simplex.c	433

Allocation of Memory				Freeing of Memory				Intermediate exits	
Filename	**Function name**	**Variable name**	**Line allocated**	**Filename**	**Function name**	**Variable name**	**Line freed**	**Filename**	**Line freed**
simplex.c	getMbyNStrat()	tempMatrix	431	simplex.c	getMbyNStrat()	tempMatrix	443		
simplex.c	getMbyNStrat()	returnValue	404, 440	Dealt with by function calling getMbyNStrat()					
functions.c	getFileName()	filename	13	Dealt with by function calling getFileName()					
functions.c	makePayoffMatrix()	filename	38	functions.c	makePayoffMatrix()	filename	59	functions.c	43, 48
functions.c	makePayoffMatrix()	input	46, 68	functions.c	makePayoffMatrix()	input	188	functions.c	86, 96, 108, 118, 127, 140, 153, 165, 179
functions.c	makePayoffMatrix()	payoff	116	Dealt with by function calling getFileName()				functions.c	128, 141, 154, 166, 180
functions.c	transposeMatrix()	tempPayoff	225	functions.c	transposeMatrix()	tempPayoff	247		
functions.c	swapStrat()	temp.XOpt	259	functions.c	swapStrat()	temp.XOpt	306	functions.c	265, 270
functions.c	swapStrat()	temp.YOpt	263	functions.c	swapStrat()	temp.YOpt	306	functions.c	271
functions.c	swapStrat()	temp.YInt	268	functions.c	swapStrat()	temp.YInt	306		
solveGame.c	main()	payoff	21	solveGame.c	main()	payoff	144	solveGame.c	29, 36, 58, 91, 101

Allocation of Memory				Freeing of Memory				Intermediate exits	
Filename	**Function name**	**Variable name**	**Line allocated**	**Filename**	**Function name**	**Variable name**	**Line freed**	**Filename**	**Line freed**
solveGame.c	main()	filename	28	solveGame.c	main()	filename	39	solveGame.c	35
solveGame.c	main()	mixed	72, 78, 84	solveGame.c	main()	mixed	145	solveGame.c	102

A1.3 Testing the Linked List Library

The function LinkListTest.c was written to test the functions in the file LinkListLib.c. The source code for both of these files can both be found in appendix A3. LinkListTest.c creates an empty list, adds elements to the list, discards an item from the list and finally deletes the list. All of the functions newList(), addItem(), discItem(), getJ(), getVal0(), getGrad(), moveFirst(), moveLast(), moveNext() and movePrev() are called. A copy of the source code and the executable file for LinkListTest can also be found on the attached CD in the folder "Testing\Section 6-3-1". A copy of the source code for LinkListLib.c can be found on the same CD in the folder "System Code and Executables\Source Code".

Output

```
mary $ LinkListTest
Creating new empty list
Adding items to the list
        Item 1: j = 1; val0 = 1.500000; grad = 2.500000
        Item 2: j = 2; val0 = 3.000000; grad = 5.000000
        Item 3: j = 3; val0 = 4.500000; grad = 7.500000
        Item 4: j = 4; val0 = 6.000000; grad = 10.000000
        Item 5: j = 5; val0 = 7.500000; grad = 12.500000
Printing the list
        Item 1: j = 1; val0 = 1.500000; grad = 2.500000
        Item 2: j = 2; val0 = 3.000000; grad = 5.000000
        Item 3: j = 3; val0 = 4.500000; grad = 7.500000
        Item 4: j = 4; val0 = 6.000000; grad = 10.000000
        Item 5: j = 5; val0 = 7.500000; grad = 12.500000
Discarding element 2 from the list
Printing the list
        Item 1: j = 1; val0 = 1.500000; grad = 2.500000
        Item 2: j = 3; val0 = 4.500000; grad = 7.500000
        Item 3: j = 4; val0 = 6.000000; grad = 10.000000
        Item 4: j = 5; val0 = 7.500000; grad = 12.500000
Printing the last item in the list
        Last item: j = 5; val0 = 7.500000; grad = 12.500000
Printing the previous item in the list
        Last item: j = 4; val0 = 6.000000; grad = 10.000000
Deleting the list
```

A1.4 Testing the Input Method

Seven test cases were input to the system using both methods of input. The output for each method from each test case was examined and verified to be identical. A copy of all the input files can be found on the attached CD, in the folder "Testing\Section 6-3-2\Input Files", and a corresponding copy of all the output files in the folder "Testing\Section 6-3-2\Output Files".

Input		
Test	**Screen**	**File**

1	`-2,4,[[1,2,3,4][4,5,6,7]]`	`-2,4,[`
		`[1,2,3,4]`
		`[4,5,6,7]]`

2	`2,2,[12][3,4]]`	`2,2,[`
		`[12]`
		`[3,4]]`

3	`2,2,[[4,3][2,1]]`	`2,2,[`
		`[4,3]`
		`[2,1]]`

4	`2,4,[[0,1,2,3][-3,-2,-1,0]]`	`2,4,[`
		`[0,1,2,3]`
		`[-3,-2,-1,0]]`

5	`4, 2, [[0, 3][-1, 2][-2, 1][-3, 0]]`	`4, 2, [`
		`[0, 3]`
		`[-1, 2]`
		`[-2, 1]`
		`[-3, 0]`
		
		`]`

6	`5, 5,[[4.5, 3.5, 7.5, 4.5, 2.5][7.5, 1.5, 1.5, 6.5, 0.5][1.5, -0.5, 2.5, -1.5, -2.5][-1.5, 4.5, -2.5, 1.5, -0.5][0.5, 2.5, 4.5, 0.5, 1.5]]`	`5, 5,[`
		`[4.5, 3.5, 7.5, 4.5, 2.5]`
		`[7.5, 1.5, 1.5, 6.5, 0.5]`
		`[1.5, -0.5, 2.5, -1.5, -2.5]`
		`[-1.5, 4.5, -2.5, 1.5, -0.5]`
		`[0.5, 2.5, 4.5, 0.5, 1.5]`
		
		`]`

Input		
Test	**Screen**	**File**

7	`4,4,[[1, 2, 3, 4][2, 3, 4, 5][5, 4, 3, 2][4, 3, 2, 1]]`	`4,4,[`
		`[1, 2, 3, 4]`
		`[2, 3, 4, 5]`
		`[5, 4, 3, 2]`
		`[4, 3, 2, 1]`
		`]`

Output		
Test	**Screen**	**File**
1	`Invalid entry for m (number of rows)`	`Invalid entry for m (number of rows)`
2	`Invalid entry at element[0,1], each element must be separated by ','`	`Invalid entry at element[0,1], each element must be separated by ','`

3	`Payoff matrix :`
	`[4.000000, 3.000000]`
	`[2.000000, 1.000000]`
	
	`There is 1 strategy`
	`Strategy 1:`
	`Player I Optimal Strategy: [1.000000, 0.000000]`
	`Player II Optimal Strategy: [0.000000, 1.000000]`
	`Value of the game: 3.000000`
		`Payoff matrix :`
		`[4.000000, 3.000000]`
		`[2.000000, 1.000000]`
		
		`There is 1 strategy`
		`Strategy 1:`
		`Player I Optimal Strategy: [1.000000, 0.000000]`
		`Player II Optimal Strategy: [0.000000, 1.000000]`
		`Value of the game: 3.000000`

4	`Payoff matrix :`
	`[0.000000, 1.000000, 2.000000, 3.000000]`
	`[-3.000000, -2.000000, -1.000000, 0.000000]`
	
	`There is 1 strategy`
	`Strategy 1:`
	`Player I Optimal Strategy: [1.000000, 0.000000]`
	`Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]`
	`Value of the game: 0.000000`
		`Payoff matrix :`
		`[0.000000, 1.000000, 2.000000, 3.000000]`
		`[-3.000000, -2.000000, -1.000000, 0.000000]`
		
		`There is 1 strategy`
		`Strategy 1:`
		`Player I Optimal Strategy: [1.000000, 0.000000]`
		`Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]`
		`Value of the game: 0.000000`

Output		
Test	**Screen**	**File**
5	Payoff matrix := [0.000000, 3.000000] [-1.000000, 2.000000] [-2.000000, 1.000000] [-3.000000, 0.000000] There is 1 strategy Strategy 1: Player I Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000] Player II Optimal Strategy: [1.000000, 0.000000] Value of the game: -0.000000	Payoff matrix := [0.000000, 3.000000] [-1.000000, 2.000000] [-2.000000, 1.000000] [-3.000000, 0.000000] There is 1 strategy Strategy 1: Player I Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000] Player II Optimal Strategy: [1.000000, 0.000000] Value of the game: -0.000000
6	Payoff matrix := [4.500000, 3.500000, 7.500000, 4.500000, 2.500000] [7.500000, 1.500000, 1.500000, 6.500000, 0.500000] [1.500000, -0.500000, 2.500000, -1.500000, -2.500000] [-1.500000, 4.500000, -2.500000, 1.500000, -0.500000] [0.500000, 2.500000, 4.500000, 0.500000, 1.500000] There is 1 strategy Strategy 1: Player I Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000] Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000, 1.000000] Value of the game: 2.500000	Payoff matrix := [4.500000, 3.500000, 7.500000, 4.500000, 2.500000] [7.500000, 1.500000, 1.500000, 6.500000, 0.500000] [1.500000, -0.500000, 2.500000, -1.500000, -2.500000] [-1.500000, 4.500000, -2.500000, 1.500000, -0.500000] [0.500000, 2.500000, 4.500000, 0.500000, 1.500000] There is 1 strategy Strategy 1: Player I Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000, 0.000000] Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000, 1.000000] Value of the game: 2.500000

Output		
Test	**Screen**	**File**
7	Payoff matrix := [1.000000, 2.000000, 3.000000, 4.000000] [2.000000, 3.000000, 4.000000, 5.000000] [5.000000, 4.000000, 3.000000, 2.000000] [4.000000, 3.000000, 2.000000, 1.000000] There are 3 strategies Strategy 1: Player I Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Player II Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Value of the game: 3.500000 Strategy 2: Player I Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Player II Optimal Strategy: [0.250000, 0.000000, 0.750000, 0.000000] Value of the game: 3.500000 Strategy 3: Player I Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Player II Optimal Strategy: [0.000000, 0.750000, 0.000000, 0.250000] Value of the game: 3.500000	Payoff matrix := [1.000000, 2.000000, 3.000000, 4.000000] [2.000000, 3.000000, 4.000000, 5.000000] [5.000000, 4.000000, 3.000000, 2.000000] [4.000000, 3.000000, 2.000000, 1.000000] There are 3 strategies Strategy 1: Player I Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Player II Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Value of the game: 3.500000 Strategy 2: Player I Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Player II Optimal Strategy: [0.250000, 0.000000, 0.750000, 0.000000] Value of the game: 3.500000 Strategy 3: Player I Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000] Player II Optimal Strategy: [0.000000, 0.750000, 0.000000, 0.250000] Value of the game: 3.500000

A1.5 Testing Invalid Inputs

A1.5.1 Mapping of Criteria to Tests

Each of the criteria listed in section 6.3.3 are tested using test cases. The table below shows which tests map to which criteria.

Test criteria	Test Number	Result
1.1	1	Pass
1.2	2	Pass
1.3	3	Pass
2.1	4	Pass
2.2	5	Pass
2.3	6	Pass
3	7,8,9,10	Pass
4	9,11	Pass

A1.5.2 Test results

A copy of all the input files can be found on the attached CD, in the folder "Testing\Section 6-3-3\Input Files", and a corresponding copy of all the output files in the folder "Testing\Section 6-3-3\Output Files". Each test is presented below as it appears in the file which was used to input the data. The input to the system is shown first and is followed by the output that was appended to the file. For example in Test 1 the input is:

```
-2,4,[
[1,2,3,4]
[4,5,6,7]]
```

and the output is:

```
Invalid entry for m (number of rows)


Test 1.
-2,4,[
[1,2,3,4]
[4,5,6,7]]

Invalid entry for m (number of rows)

Test 2.
2.5,4,[
[1,2,3,4]
[4,5,6,7]]

No comma after m (number of rows)

Test 3.
3,4,[
[1,2,3,4]
[4,5,6,7]]

Invalid entry, each row must start with '['

Test 4.
2,-4,[
[1,2,3,4]
```

```
[4,5,6,7]]

Invalid entry for n (number of columns)
```

Test 5.

```
2,4.5,[
[1,2,3,4]
[4,5,6,7]]

Invalid start to payoff matrix. Require comma, double open square bracket
```

Test 6.

```
2,3,[
[1,2,3,4]
[4,5,6,7]]

Invalid entry, each row must end with ']'
```

Test 7.

```
2,2,[
[,1,2]
[3,4]]

Invalid data entry for element [0,0]
```

Test 8.

```
2,2,
[1,2]
[3,4]]

Invalid entry, each row must start with '['
```

Test 9.

```
2,2,[
[12]
[3,4]]

Invalid entry at element[0,1], each element must be separated by ','
```

Test 10.

```
2,2,[
[1.2]
[3,4]]

Invalid entry at element[0,1], each element must be separated by ','
```

Test 11.

```
2,2,[
[,2]
[3,4]]

Invalid data entry for element [0,0]
```

A1.6 Testing Different Types of Games

A1.6.1 Mapping of Criteria to Tests

Each of the criteria listed in section 6.3.4 are tested using test cases. The table below shows which tests map to which criteria.

Test criteria	Test Number	Result
1	1,2,3,4,	Pass
2.1	5,	Pass
2.2	6,	Pass
2.3	7,	Pass
2.4	8,	Pass
2.5	9,	Pass
2.6	10,	Pass
2.7	11,	Pass
2.8	12,	Pass
2.9	13,	Pass
3	14,15,16,17,18,19,	Pass
4	20,21,22,23,	Pass
5	24,25,26,27,	Pass
6	28,29,30,31,32,33,	Pass
7	34,35,36,37,	Pass
8	38,39,40,41,	Pass
9	42,43,44,45,46,47,48,	Pass
10	49,50,51,52,	Pass
11	53,54,55,	Pass
12.1	1,2,3,4,5,7,8,9,11,12,13,14,17,18, 19, 20, 21,22,23, 24,25,26,27, 30, 42, 43, 45,46,47,48, 49,50,51,52, 53,54,55,	Pass
12.2	10, 16,28, 31, 32, 33, 34,35,36,37, 38,39,40,41,	Pass
12.3	6,15,29, 44,	Pass
13.1	1,2,3,4,6,14,15, 16,17, 18, 19, 20, 21,22,23, 28,29, 30, 32, 33, 34,35,36,37, 42, 44, 45,46,47,48,	Pass
13.2	5,7,8,9,10,11,12,13, 24,25,26,27, 31, 38,39,40,41, 43, 49,50,51,52, 53,54,55,	Pass
14.1	1,2,3,4,5,6,7,8,9,11,12,13,14,15, 17, 18, 19, 20, 21,22,23, 24,25,26,27, 29, 30, 34,35,36,37, 38,39,40,41, 42, 43, 44, 45,46,47,48, 49,50,51,52, 53,54,55,	Pass
14.2	6,10, 15, 16,20, 28,29, 31, 32, 33, 34,35,36,37, 38,39,40,41, 42, 43, 44, 45,46,47,48, 49,50,51,52,	Pass
14.3	1,2,3,4,9,10, 15, 16,20, 21,22,23,29, 34,35,36,37, 42, 44, 45,46,47,48, 49,50,51,52,	Pass
15.1	1,2,3,4,5,7,8,9,10,11,12,13,14, 15, 16, 18, 19, 20, 21,22,23, 24,25,26,27, 29, 30, 34,35,36,37, 38,39,40,41, 42, 44, 45,46,47,48, 49,50,51,52,	Pass
15.2	6, 17, 24,25,26,27, 28, 31, 32, 33, 38,39,40,41, 43, 53,54,55,	Pass

A1.6.2 Test results

A copy of all the input files can be found on the attached CD, in the folder "Testing\Section 6-3-4\Input Files", and a corresponding copy of all the output files in the

folder “Testing\Section 6-3-4\Output Files”. Each test is presented below as it appears in the file which was used to input the data. The input to the system is shown first and is followed by the output that was appended to the file. For example in Test 1 the input is:

```
2,2,[
[3,4]
[1,2]]
```

and the output is:

```
Payoff matrix :=
       [3.000000, 4.000000]
       [1.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000]
   Player II Optimal Strategy: [1.000000, 0.000000]
   Value of the game: 3.000000
```

Test 1.

```
2,2,[
[3,4]
[1,2]]

Payoff matrix :=
       [3.000000, 4.000000]
       [1.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000]
   Player II Optimal Strategy: [1.000000, 0.000000]
   Value of the game: 3.000000
```

Test 2.

```
2,2,[
[4,3]
[2,1]]

Payoff matrix :=
       [4.000000, 3.000000]
       [2.000000, 1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000]
   Player II Optimal Strategy: [0.000000, 1.000000]
   Value of the game: 3.000000
```

Test 3.

```
2,2,[
[2,1]
[4,3]]

Payoff matrix :=
       [2.000000, 1.000000]
       [4.000000, 3.000000]

There is 1 strategy
Strategy 1:
```

```
   Player I  Optimal Strategy: [0.000000, 1.000000]
   Player II Optimal Strategy: [0.000000, 1.000000]
   Value of the game: 3.000000
```

Test 4.

```
2,2,[
[1,2]
[3,4]]

Payoff matrix :=
       [1.000000, 2.000000]
       [3.000000, 4.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000]
   Player II Optimal Strategy: [1.000000, 0.000000]
   Value of the game: 3.000000
```

Test 5.

```
2,2,[
[4,1]
[1,3]]

Payoff matrix :=
       [4.000000, 1.000000]
       [1.000000, 3.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.400000, 0.600000]
   Player II Optimal Strategy: [0.400000, 0.600000]
   Value of the game: 2.200000
```

Test 6.

```
2,2,[
[-1.5,0.5]
[1.5,-0.5]]

Payoff matrix :=
       [-1.500000, 0.500000]
       [1.500000, -0.500000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.500000, 0.500000]
   Player II Optimal Strategy: [0.250000, 0.750000]
   Value of the game: -0.000000
```

Test 7.

```
2,2,[
[1,3]
[4,1]]

Payoff matrix :=
       [1.000000, 3.000000]
       [4.000000, 1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.600000, 0.400000]
```

```
   Player II Optimal Strategy: [0.400000, 0.600000]
   Value of the game: 2.200000
```

Test 8.

```
2,2,[
[1,4]
[3,2]]

Payoff matrix :=
       [1.000000, 4.000000]
       [3.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.250000, 0.750000]
   Player II Optimal Strategy: [0.500000, 0.500000]
   Value of the game: 2.500000
```

Test 9.

```
2,2,[
[1,0]
[0,1]]

Payoff matrix :=
       [1.000000, 0.000000]
       [0.000000, 1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.500000, 0.500000]
   Player II Optimal Strategy: [0.500000, 0.500000]
   Value of the game: 0.500000
```

Test 10.

```
2,2,[
[-1,-2]
[-3,0]]

Payoff matrix :=
       [-1.000000, -2.000000]
       [-3.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.750000, 0.250000]
   Player II Optimal Strategy: [0.500000, 0.500000]
   Value of the game: -1.500000
```

Test 11.

```
2,2,[
[1,4]
[3,1]]

Payoff matrix :=
       [1.000000, 4.000000]
       [3.000000, 1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.400000, 0.600000]
   Player II Optimal Strategy: [0.600000, 0.400000]
```

```
   Value of the game: 2.200000
```

Test 12.

```
2,2,[
[3,1]
[2,4]]

Payoff matrix :=
       [3.000000, 1.000000]
       [2.000000, 4.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.500000, 0.500000]
   Player II Optimal Strategy: [0.750000, 0.250000]
   Value of the game: 2.500000
```

Test 13.

```
2,2,[
[3,1]
[1,4]]

Payoff matrix :=
       [3.000000, 1.000000]
       [1.000000, 4.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.600000, 0.400000]
   Player II Optimal Strategy: [0.600000, 0.400000]
   Value of the game: 2.200000
```

Test 14.

```
2,4,[
[1,2,3,4]
[4,5,6,7]]

Payoff matrix :=
       [1.000000, 2.000000, 3.000000, 4.000000]
       [4.000000, 5.000000, 6.000000, 7.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000]
   Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]
   Value of the game: 4.000000
```

Test 15.

```
2,4,[
[0,1,2,3]
[-3,-2,-1,0]]

Payoff matrix :=
       [0.000000, 1.000000, 2.000000, 3.000000]
       [-3.000000, -2.000000, -1.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000]
   Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]
   Value of the game: 0.000000
```

Test 16.

```
2,4,[
[0,-2,-1,-3]
[-3,-5,-4,-6]]

Payoff matrix :=
       [0.000000, -2.000000, -1.000000, -3.000000]
       [-3.000000, -5.000000, -4.000000, -6.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000]
   Value of the game: -3.000000
```

Test 17.

```
2,4,[
[1.5,2.5,3.5,4.5]
[7.5,6.5,5.5,5]
]

Payoff matrix :=
       [1.500000, 2.500000, 3.500000, 4.500000]
       [7.500000, 6.500000, 5.500000, 5.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000]
   Value of the game: 5.000000
```

Test 18.

```
2,4,[
[1.5,2.5,4.5,3.5]
[7.5,6.5,5,5.5]
]

Payoff matrix :=
       [1.500000, 2.500000, 4.500000, 3.500000]
       [7.500000, 6.500000, 5.000000, 5.500000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000]
   Value of the game: 5.000000
```

Test 19.

```
2,4,[
[7.5, 5, 6.5, 5.5]
[1.5, 4.5, 2.5, 3.5]
]

Payoff matrix :=
       [7.500000, 5.000000, 6.500000, 5.500000]
       [1.500000, 4.500000, 2.500000, 3.500000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000]
```

```
   Player II Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000]
   Value of the game: 5.000000
```

Test 20.

```
2,4,[
[-1, 6, 8, 2]
[7, 7, 0, 2]
]

Payoff matrix :=
       [-1.000000, 6.000000, 8.000000, 2.000000]
       [7.000000, 7.000000, 0.000000, 2.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [(0.250000-0.625000), (0.750000-0.375000)]
   Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000]
   Value of the game: 2.000000
```

Test 21.

```
2,4,[
[-1, 6, 2, 8]
[7, 7, 2, 0]
]

Payoff matrix :=
       [-1.000000, 6.000000, 2.000000, 8.000000]
       [7.000000, 7.000000, 2.000000, 0.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [(0.250000-0.625000), (0.750000-0.375000)]
   Player II Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000]
   Value of the game: 2.000000
```

Test 22.

```
2,4,[
[7, 2, 7, 0]
[-1, 2, 6, 8]
]

Payoff matrix :=
       [7.000000, 2.000000, 7.000000, 0.000000]
       [-1.000000, 2.000000, 6.000000, 8.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [(0.375000-1.000000), (0.625000-0.000000)]
   Player II Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000]
   Value of the game: 2.000000
```

Test 23.

```
2,4,[
[2, 7, 7, 0]
[2,-1,  6, 8]
]

Payoff matrix :=
       [2.000000, 7.000000, 7.000000, 0.000000]
       [2.000000, -1.000000, 6.000000, 8.000000]
```

```
There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [(0.375000-1.000000), (0.625000-0.000000)]
   Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]
   Value of the game: 2.000000
```

Test 24.

```
2,4,[
[2, 6, 3, 4]
[6, 0.5, 2, 1]
]

Payoff matrix :=
       [2.000000, 6.000000, 3.000000, 4.000000]
       [6.000000, 0.500000, 2.000000, 1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.800000, 0.200000]
   Player II Optimal Strategy: [0.200000, 0.000000, 0.800000, 0.000000]
   Value of the game: 2.800000
```

Test 25.

```
2,4,[
[2, 6, 4, 3]
[6, 0.5, 1, 2]
]

Payoff matrix :=
       [2.000000, 6.000000, 4.000000, 3.000000]
       [6.000000, 0.500000, 1.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.800000, 0.200000]
   Player II Optimal Strategy: [0.200000, 0.000000, 0.000000, 0.800000]
   Value of the game: 2.800000
```

Test 26.

```
2,4,[
[0.5, 6, 2, 1]
[6, 2, 3, 4]
]

Payoff matrix :=
       [0.500000, 6.000000, 2.000000, 1.000000]
       [6.000000, 2.000000, 3.000000, 4.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.200000, 0.800000]
   Player II Optimal Strategy: [0.000000, 0.200000, 0.800000, 0.000000]
   Value of the game: 2.800000
```

Test 27.

```
2,4,[
[0.5, 6, 1, 2]
[6, 2, 4, 3]
]

Payoff matrix :=
```

```
       [0.500000, 6.000000, 1.000000, 2.000000]
       [6.000000, 2.000000, 4.000000, 3.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.200000, 0.800000]
   Player II Optimal Strategy: [0.000000, 0.200000, 0.000000, 0.800000]
   Value of the game: 2.800000
```

Test 28.

```
4, 2, [
[-1,-4]
[-2,-5]
[-3,-6]
[-4,-7]
]

Payoff matrix :=
       [-1.000000, -4.000000]
       [-2.000000, -5.000000]
       [-3.000000, -6.000000]
       [-4.000000, -7.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]
   Player II Optimal Strategy: [0.000000, 1.000000]
   Value of the game: -4.000000
```

Test 29.

```
4, 2, [
[0, 3]
[-1, 2]
[-2, 1]
[-3, 0]
]

Payoff matrix :=
       [0.000000, 3.000000]
       [-1.000000, 2.000000]
       [-2.000000, 1.000000]
       [-3.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]
   Player II Optimal Strategy: [1.000000, 0.000000]
   Value of the game: -0.000000
```

Test 30.

```
4, 2, [
[0, 6]
[2, 5]
[1, 4]
[3, 4]
]

Payoff matrix :=
       [0.000000, 6.000000]
       [2.000000, 5.000000]
       [1.000000, 4.000000]
       [3.000000, 4.000000]
```

```
There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000]
   Player II Optimal Strategy: [1.000000, 0.000000]
   Value of the game: 3.000000
```

Test 31.

```
4, 2, [
[-1.5, -7.5]
[-2.5, -6.5]
[-3.5, -5.5]
[-4.5, -5]
]

Payoff matrix :=
       [-1.500000, -7.500000]
       [-2.500000, -6.500000]
       [-3.500000, -5.500000]
       [-4.500000, -5.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000]
   Player II Optimal Strategy: [0.000000, 1.000000]
   Value of the game: -5.000000
```

Test 32.

```
4, 2, [
[-1.5, -7.5]
[-2.5, -6.5]
[-4.5, -5]
[-3.5, -5.5]
]

Payoff matrix :=
       [-1.500000, -7.500000]
       [-2.500000, -6.500000]
       [-4.500000, -5.000000]
       [-3.500000, -5.500000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000]
   Player II Optimal Strategy: [0.000000, 1.000000]
   Value of the game: -5.000000
```

Test 33.

```
4, 2, [
[-7.5, -1.5]
[-5, -4.5]
[-6.5, -2.5]
[-5.5, -3.5]
]

Payoff matrix :=
       [-7.500000, -1.500000]
       [-5.000000, -4.500000]
       [-6.500000, -2.500000]
       [-5.500000, -3.500000]

There is 1 strategy
```

```
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000]
   Player II Optimal Strategy: [1.000000, 0.000000]
   Value of the game: -5.000000
```

Test 34.

```
4, 2, [
[1, -7]
[-6, -7]
[-8, 0]
[-2, -2]
]

Payoff matrix :=
       [1.000000, -7.000000]
       [-6.000000, -7.000000]
       [-8.000000, 0.000000]
       [-2.000000, -2.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000]
   Player II Optimal Strategy: [(0.250000-0.625000), (0.750000-0.375000)]
   Value of the game: -2.000000
```

Test 35.

```
4, 2, [
[1, -7]
[-6, -7]
[-2, -2]
[-8, 0]
]

Payoff matrix :=
       [1.000000, -7.000000]
       [-6.000000, -7.000000]
       [-2.000000, -2.000000]
       [-8.000000, 0.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000]
   Player II Optimal Strategy: [(0.250000-0.625000), (0.750000-0.375000)]
   Value of the game: -2.000000
```

Test 36.

```
4, 2, [
[-7, 1]
[-2, -2]
[-7, -6]
[0, -8]
]

Payoff matrix :=
       [-7.000000, 1.000000]
       [-2.000000, -2.000000]
       [-7.000000, -6.000000]
       [0.000000, -8.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000]
```

```
   Player II Optimal Strategy: [(0.375000-1.000000), (0.625000-0.000000)]
   Value of the game: -2.000000
```

Test 37.

```
4, 2, [
[-2, -2]
[-7, 1]
[-7, -6]
[0, -8]
]

Payoff matrix :=
       [-2.000000, -2.000000]
       [-7.000000, 1.000000]
       [-7.000000, -6.000000]
       [0.000000, -8.000000]

There are interval strategies
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000]
   Player II Optimal Strategy: [(0.375000-1.000000), (0.625000-0.000000)]
   Value of the game: -2.000000
```

Test 38.

```
4, 2, [
[-2,-6]
[-6, 0.5]
[-3, -2]
[-4, -1]
]

Payoff matrix :=
       [-2.000000, -6.000000]
       [-6.000000, 0.500000]
       [-3.000000, -2.000000]
       [-4.000000, -1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.200000, 0.000000, 0.800000, 0.000000]
   Player II Optimal Strategy: [0.800000, 0.200000]
   Value of the game: -2.800000
```

Test 39.

```
4, 2, [
[-2,-6]
[-6, -0.5]
[-4, -1]
[-3, -2]
]

Payoff matrix :=
       [-2.000000, -6.000000]
       [-6.000000, -0.500000]
       [-4.000000, -1.000000]
       [-3.000000, -2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.200000, 0.000000, 0.000000, 0.800000]
   Player II Optimal Strategy: [0.800000, 0.200000]
   Value of the game: -2.800000
```

Test 40.

```
4, 2, [
[-0.5, -6]
[-6, -2]
[-2, -3]
[-1, -4]
]

Payoff matrix :=
       [-0.500000, -6.000000]
       [-6.000000, -2.000000]
       [-2.000000, -3.000000]
       [-1.000000, -4.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.200000, 0.800000, 0.000000]
   Player II Optimal Strategy: [0.200000, 0.800000]
   Value of the game: -2.800000
```

Test 41.

```
4, 2, [
[-0.5, -6]
[-6, -2]
[-1, -4]
[-2, -3]
]

Payoff matrix :=
       [-0.500000, -6.000000]
       [-6.000000, -2.000000]
       [-1.000000, -4.000000]
       [-2.000000, -3.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.200000, 0.000000, 0.800000]
   Player II Optimal Strategy: [0.200000, 0.800000]
   Value of the game: -2.800000
```

Test 42.

```
5, 5,[
[2, 4, 3, 7, 4]
[0, 7, 1, 1, 6]
[-3, 1, -1, 2, -2]
[-1, -2, 4, -3, 1]
[1, 0, 2, 4, 0]
]

Payoff matrix :=
       [2.000000, 4.000000, 3.000000, 7.000000, 4.000000]
       [0.000000, 7.000000, 1.000000, 1.000000, 6.000000]
       [-3.000000, 1.000000, -1.000000, 2.000000, -2.000000]
       [-1.000000, -2.000000, 4.000000, -3.000000, 1.000000]
       [1.000000, 0.000000, 2.000000, 4.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000,
0.000000]
```

```
   Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000,
0.000000]
   Value of the game: 2.000000
```

Test 43.

```
5, 5,[
[4.5, 3.5, 7.5, 4.5, 2.5]
[7.5, 1.5, 1.5, 6.5, 0.5]
[1.5, -0.5, 2.5, -1.5, -2.5]
[-1.5, 4.5, -2.5, 1.5, -0.5]
[0.5, 2.5, 4.5, 0.5, 1.5]
]

Payoff matrix :=
       [4.500000, 3.500000, 7.500000, 4.500000, 2.500000]
       [7.500000, 1.500000, 1.500000, 6.500000, 0.500000]
       [1.500000, -0.500000, 2.500000, -1.500000, -2.500000]
       [-1.500000, 4.500000, -2.500000, 1.500000, -0.500000]
       [0.500000, 2.500000, 4.500000, 0.500000, 1.500000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000,
0.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000,
1.000000]
   Value of the game: 2.500000
```

Test 44.

```
5, 5,[
[-2, 5, -1, -1, 4]
[-5, -1, -3, 0, -4]
[-3, -4, 2, -1, -1]
[-1, -2, 0, 2, -2]
[0, 2, 1, 5, 2]
]

Payoff matrix :=
       [-2.000000, 5.000000, -1.000000, -1.000000, 4.000000]
       [-5.000000, -1.000000, -3.000000, 0.000000, -4.000000]
       [-3.000000, -4.000000, 2.000000, -1.000000, -1.000000]
       [-1.000000, -2.000000, 0.000000, 2.000000, -2.000000]
       [0.000000, 2.000000, 1.000000, 5.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000,
1.000000]
   Player II Optimal Strategy: [1.000000, 0.000000, 0.000000, 0.000000,
0.000000]
   Value of the game: 0.000000
```

Test 45.

```
5, 5,[
[7, 1, 1, 6, 0]
[1, -1, 2, -2, -3]
[-2, 4, -3, 1, -1]
[0, 2, 4, 0, 1]
[4, 3, 7, 4, 2]
]

Payoff matrix :=
```

```
       [7.000000, 1.000000, 1.000000, 6.000000, 0.000000]
       [1.000000, -1.000000, 2.000000, -2.000000, -3.000000]
       [-2.000000, 4.000000, -3.000000, 1.000000, -1.000000]
       [0.000000, 2.000000, 4.000000, 0.000000, 1.000000]
       [4.000000, 3.000000, 7.000000, 4.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000,
1.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 0.000000,
1.000000]
   Value of the game: 2.000000
```

Test 46.

```
5, 5,[
[7, 1, 0, 1, 6]
[1, -1, -3, 2, -2]
[4, 3, 2, 7, 4]
[-2, 4, -1, -3, 1]
[0, 2, 1, 4, 0]
]

Payoff matrix :=
       [7.000000, 1.000000, 0.000000, 1.000000, 6.000000]
       [1.000000, -1.000000, -3.000000, 2.000000, -2.000000]
       [4.000000, 3.000000, 2.000000, 7.000000, 4.000000]
       [-2.000000, 4.000000, -1.000000, -3.000000, 1.000000]
       [0.000000, 2.000000, 1.000000, 4.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000,
0.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 1.000000, 0.000000,
0.000000]
   Value of the game: 2.000000
```

Test 47.

```
5, 5,[
[7, 0, 1, 1, 6]
[4, 2, 3, 7, 4]
[1, -3, -1, 2, -2]
[-2, -1, 4, -3, 1]
[0, 1, 2, 4, 0]
]

Payoff matrix :=
       [7.000000, 0.000000, 1.000000, 1.000000, 6.000000]
       [4.000000, 2.000000, 3.000000, 7.000000, 4.000000]
       [1.000000, -3.000000, -1.000000, 2.000000, -2.000000]
       [-2.000000, -1.000000, 4.000000, -3.000000, 1.000000]
       [0.000000, 1.000000, 2.000000, 4.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000,
0.000000]
   Player II Optimal Strategy: [0.000000, 1.000000, 0.000000, 0.000000,
0.000000]
   Value of the game: 2.000000
```

Test 48.

```
5, 5,[
[7, 1, 1, 0, 6]
[1, -1, 2, -3, -2]
[-2, 4, -3, -1, 1]
[4, 3, 7, 2, 4]
[0, 2, 4, 1, 0]
]

Payoff matrix :=
       [7.000000, 1.000000, 1.000000, 0.000000, 6.000000]
       [1.000000, -1.000000, 2.000000, -3.000000, -2.000000]
       [-2.000000, 4.000000, -3.000000, -1.000000, 1.000000]
       [4.000000, 3.000000, 7.000000, 2.000000, 4.000000]
       [0.000000, 2.000000, 4.000000, 1.000000, 0.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000,
0.000000]
   Player II Optimal Strategy: [0.000000, 0.000000, 0.000000, 1.000000,
0.000000]
   Value of the game: 2.000000
```

Test 49.

```
5, 5,[
[6, 7, 5, 3, 7]
[2, 1, 0, 10, 5]
[2, 4, 8, 9, 4]
[4, 0, 7, 4, 1]
[7, 4, 9, 3, 2]
]

Payoff matrix :=
       [6.000000, 7.000000, 5.000000, 3.000000, 7.000000]
       [2.000000, 1.000000, 0.000000, 10.000000, 5.000000]
       [2.000000, 4.000000, 8.000000, 9.000000, 4.000000]
       [4.000000, 0.000000, 7.000000, 4.000000, 1.000000]
       [7.000000, 4.000000, 9.000000, 3.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.449541, 0.119266, 0.192661, 0.000000,
0.238532]
   Player II Optimal Strategy: [0.467890, 0.133028, 0.000000, 0.385321,
0.013761]
   Value of the game: 4.990826
```

Test 50.

```
5, 5,[
[6, 5, 3, 7, 7]
[4, 7, 4, 0, 1]
[7, 9, 3, 4, 2]
[2, 0, 10, 1, 5]
[2, 8, 9, 4, 4]
]

Payoff matrix :=
       [6.000000, 5.000000, 3.000000, 7.000000, 7.000000]
       [4.000000, 7.000000, 4.000000, 0.000000, 1.000000]
       [7.000000, 9.000000, 3.000000, 4.000000, 2.000000]
       [2.000000, 0.000000, 10.000000, 1.000000, 5.000000]
       [2.000000, 8.000000, 9.000000, 4.000000, 4.000000]
```

```
There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.449541, 0.000000, 0.238532, 0.119266,
0.192661]
   Player II Optimal Strategy: [0.467890, 0.000000, 0.385321, 0.133028,
0.013761]
   Value of the game: 4.990826
```

Test 51.

```
5, 5,[
[5, 6, 7, 3, 7]
[0, 2, 1, 10, 5]
[8, 2, 4, 9, 4]
[9, 7, 4, 3, 2]
[7, 4, 0, 4, 1]
]

Payoff matrix :=
       [5.000000, 6.000000, 7.000000, 3.000000, 7.000000]
       [0.000000, 2.000000, 1.000000, 10.000000, 5.000000]
       [8.000000, 2.000000, 4.000000, 9.000000, 4.000000]
       [9.000000, 7.000000, 4.000000, 3.000000, 2.000000]
       [7.000000, 4.000000, 0.000000, 4.000000, 1.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.449541, 0.119266, 0.192661, 0.238532,
0.000000]
   Player II Optimal Strategy: [0.000000, 0.467890, 0.133028, 0.385321,
0.013761]
   Value of the game: 4.990826
```

Test 52.

```
5, 5,[
[7, 4, 0, 4, 1]
[5, 6, 7, 3, 7]
[0, 2, 1, 10, 5]
[8, 2, 4, 9, 4]
[9, 7, 4, 3, 2]
]

Payoff matrix :=
       [7.000000, 4.000000, 0.000000, 4.000000, 1.000000]
       [5.000000, 6.000000, 7.000000, 3.000000, 7.000000]
       [0.000000, 2.000000, 1.000000, 10.000000, 5.000000]
       [8.000000, 2.000000, 4.000000, 9.000000, 4.000000]
       [9.000000, 7.000000, 4.000000, 3.000000, 2.000000]

There is 1 strategy
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.449541, 0.119266, 0.192661,
0.238532]
   Player II Optimal Strategy: [0.000000, 0.467890, 0.133028, 0.385321,
0.013761]
   Value of the game: 4.990826
```

Test 53.

```
4,4,[
[1, 2, 3, 4]
[2, 3, 4, 5]
[5, 4, 3, 2]
```

```
[4, 3, 2, 1]
]

Payoff matrix :=
       [1.000000, 2.000000, 3.000000, 4.000000]
       [2.000000, 3.000000, 4.000000, 5.000000]
       [5.000000, 4.000000, 3.000000, 2.000000]
       [4.000000, 3.000000, 2.000000, 1.000000]

There are 3 strategies
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000]
   Player II Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000]
   Value of the game: 3.500000

Strategy 2:
   Player I  Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000]
   Player II Optimal Strategy: [0.250000, 0.000000, 0.750000, 0.000000]
   Value of the game: 3.500000

Strategy 3:
   Player I  Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000]
   Player II Optimal Strategy: [0.000000, 0.750000, 0.000000, 0.250000]
   Value of the game: 3.500000
```

Test 54.

```
4,4,[
[1, 2, 3, 4]
[4, 3, 2, 1]
[2, 3, 4, 5]
[5, 4, 3, 2]
]

Payoff matrix :=
       [1.000000, 2.000000, 3.000000, 4.000000]
       [4.000000, 3.000000, 2.000000, 1.000000]
       [2.000000, 3.000000, 4.000000, 5.000000]
       [5.000000, 4.000000, 3.000000, 2.000000]

There are 3 strategies
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.500000, 0.500000]
   Player II Optimal Strategy: [0.000000, 0.500000, 0.500000, 0.000000]
   Value of the game: 3.500000

Strategy 2:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.500000, 0.500000]
   Player II Optimal Strategy: [0.250000, 0.000000, 0.750000, 0.000000]
   Value of the game: 3.500000

Strategy 3:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.500000, 0.500000]
   Player II Optimal Strategy: [0.000000, 0.750000, 0.000000, 0.250000]
   Value of the game: 3.500000
```

Test 55.

```
4,4,[
[2, 3, 4, 1]
[3, 2, 1, 4]
[3, 4, 5, 2]
[4, 3, 2, 5]
]
```

```
Payoff matrix :=
       [2.000000, 3.000000, 4.000000, 1.000000]
       [3.000000, 2.000000, 1.000000, 4.000000]
       [3.000000, 4.000000, 5.000000, 2.000000]
       [4.000000, 3.000000, 2.000000, 5.000000]

There are 3 strategies
Strategy 1:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.500000, 0.500000]
   Player II Optimal Strategy: [0.500000, 0.500000, 0.000000, 0.000000]
   Value of the game: 3.500000

Strategy 2:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.500000, 0.500000]
   Player II Optimal Strategy: [0.750000, 0.000000, 0.250000, 0.000000]
   Value of the game: 3.500000

Strategy 3:
   Player I  Optimal Strategy: [0.000000, 0.000000, 0.500000, 0.500000]
   Player II Optimal Strategy: [0.000000, 0.750000, 0.000000, 0.250000]
   Value of the game: 3.500000
```

A2. RUNNING THE SYSTEM

The source code for the system as well as an executable file, all test files and all test results can be found on the CD attached to this dissertation.

A2.1 Installing the Executable File

The executable file "solveGame" can be found on the CD in the file "System Code and Executables\Executable". It was compiled on the University of Bath's BUCS machines. C is compiled to the machine code of the machine it is compiled on; therefore the executable file will not be portable to other platforms.

If the machine that the executable file is to be run on is the same platform, copy the executable file solveGame (note that this is not the file solveGame.c) to a folder on your machine. Ensure that you are viewing this folder from the command line. At the command line type "solveGame " followed by your input (see sections A2.3 and A2.4).

If the machine that the executable file is to be run on is not the same platform, the code will need to be recompiled. See section A2.2 for details on how to recompile the code, then run the file from the command line as described above.

A2.2 Modifying and Recompiling the Source Code

The source code can be found on the CD in the file "System Code and Executables\Source Code". It may for some reason be desirable to recompile the code; for example after changes have been made. The developer of this system used GCC, the GNU Compiler Collection, which is available free of charge from the GCC Home Page (Pfeifer and Johnson [2005]). Instructions for the recompilation are shown below:

1. Copy the source code from the CD to a folder on the designated machine.
    a. The files required for the system are:
        i. LinkListLib.h
        ii. LinkListLib.c
        iii. strategies.h
        iv. strategies.c
        v. simplex.h
        vi. simplex.c
        vii. functions.h
        viii. functions.c
        ix. solveGame.c
2. Modify the required files.
3. Ensure that the command line is viewing the correct folder (that is, the one containing the modified source code)
4. At the command line type:
   gcc LinkListLib.c strategies.c simplex.c functions.c solveGame.c -o solveGame
5. The system is now ready to run.

A2.3 Inputting a Game From the Command Line

To input data from a file, type "solveGame " followed by the payoff matrix in the form shown below, where m is the number of rows, n is the number of columns, and a_{ij} is the element in the i^{th} row and the j^{th} column. Additional spaces are allowed.

The format should be:

m,n,[[a_{11}, a_{12}, …, a_{1n}] [a_{21}, a_{22}, …, a_{2n}] … [a_{1m}, a_{1m}, …, a_{mn}]]

For example, to input the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ type "solveGame 2,3,[[1,2,3][4,5,6]]".

The output will be displayed at the command line interface below the input.

A2.4 Inputting a Game From a File

To input data from the command line, type "solveGame :" followed by the name of the file containing the input.

For example if the input is stored in the file "testFile", type "solveGame :testFile". The output will be appended to the file "testFile"

The format of the matrix in the file should be the same as for entering the matrix from the command line, although additional white spaces including tabs and carriage returns are permitted to make the input more readable.

For example, to input the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ the data in the file could read:

"2,3,[[1,2,3][4,5,6]]"

or "2,3,[
[1,2,3]
[4,5,6]
]
and so on.

A3. SOURCE CODE

A3.1 LinkListLib.h

```
/* header file for LinkListLib.c */


/* Define the data structure ColStrat */
typedef struct _col {
    int j;
    double val0;
    double grad;
    struct _col * next;
    struct _col * prev;
} ColStrat;


/* Declare the functions */
extern ColStrat * newList();

extern ColStrat * addItem(ColStrat * last, int newJ, double newVal0, double newGrad);

extern ColStrat * discItem(ColStrat * old);

extern int getJ(ColStrat * col);

extern double getVal0(ColStrat * col);

extern double getGrad(ColStrat * col);

extern ColStrat * moveNext(ColStrat * col);

extern ColStrat * movePrev(ColStrat * col);

extern ColStrat * moveFirst(ColStrat * col);

extern ColStrat * moveLast(ColStrat * col);
```

A3.2 LinkListLib.c

```
/* Library file containing functions for the doubly linked list data

 * structure ColStrat
 */

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "LinkListLib.h"

/* Function that creates a new,empty list
 * Returns a null ColStrat pointer
 */
ColStrat * newList() {
    return NULL;
}

/* Function that adds an item to the list, after the item pointed to by the
 * argument last
 */
ColStrat *addItem(ColStrat * last, int newJ, double newVal0, double newGrad){
    ColStrat * newItem;
    ColStrat * oldnext;

    // Reserve the memory for the new item
    newItem = (ColStrat *)malloc(sizeof(ColStrat));

    /* adjust the pointers so the new item lies after the item pointed to by
     * last
     */
    if(last) {

        oldnext = last->next;
        last->next = newItem;

        if (oldnext)
            oldnext->prev = newItem;
    } else
        oldnext = (ColStrat *)NULL;

    newItem->prev = last;
    newItem->next = oldnext;


    //set the data for the new item
    newItem->j = newJ;
    newItem->val0= newVal0;
    newItem->grad=newGrad;

    //return a pointer to the new item
    return newItem;
}

/* Function that discards the item pointed to by the argument last from the
 * list and returns a pointer to the next item in the list if there is one,
 * or the previous item if it is the last item, or a null pointer if it is
 * the only item in the list
 */
ColStrat * discItem(ColStrat * old) {

    /* Adjust the pointers of the previous and next item so that there is no
     * gap in the list
     */
    if (old) {
        if(old->next)
            old->next->prev = old->prev;

        if (old->prev)
            old->prev->next = old->next;

        //free the memory
        free(old);
```

```
    }

    //return a pointer to the next item if there is one
    if (old->next)
        return old->next;
    //if there is no following item, return a pointer to the preceding item
    else if (old->prev)
        return old->prev;
    //if there are no other items return a null pointer
    else
        return NULL;

}

/* Function that retrieves the value of j for the item pointed to by the
 * argument col
 */
int getJ(ColStrat * col) {
    //catch any null pointers
    if (col)
        //return the value
        return col->j;
}

/* Function that retrieves the value of val0 for the item pointed to by the
 * argument col
 */
double getVal0(ColStrat * col) {

    //catch any null pointers
    if (col)
        //return the value
        return col->val0;
}

/* Function that retrieves the value of grad for the item pointed to by the
 * argument col
 */
double getGrad(ColStrat * col) {
    //catch any null pointers
    if (col)
        //return the value
        return col->grad;
}

/* Function that moves to the next item in the list after the item pointed to
 * by the argument col
 */
ColStrat * moveNext(ColStrat * col) {
    //catch any null pointers
    if (col)
        //move to the next item
        col = col->next;

    /* return the next item. If col was a null pointer then the return value
     * will be null
     */
    return col;
}

/* Function that moves to the previous item in the list before the item
 * pointed to by the argument col
 */
ColStrat * movePrev(ColStrat * col) {
    //catch any null pointers
    if (col)
        //move to the previous item
        col = col->prev;

    /* return the previous item. If col was a null pointer then the return
     * value will be null
     */
    return col;
}


/* Function that moves to the first item in the list
 */
ColStrat * moveFirst(ColStrat * col) {
    //catch any null pointers
    if (col) {
        //move to the first item in the list
        while (col->prev) {
            col = movePrev(col);
        }
    }

    /* return the first item. If col was a null pointer then the return value
     * will be null
     */
    return col;
}

/* Function that moves to the last item in the list
 */
ColStrat * moveLast(ColStrat * col) {
    //catch any null pointers
    if (col) {
        //move to the last item in the list
        while (col->next) {
            col = moveNext(col);
        }
    }

    /* return the last item. If col was a null pointer then the return value
     * will be null
     */
    return col;
}
```

A3.3 strategies.h

```
/* header file for strategies.c */

#include "LinkListLib.h"

/* Define the data structure OptStrat */
typedef struct _optStrat {
    double *XOpt;
    double *YOpt;
    double value;
```

```
    int XInterval;
    int YInterval;
    double *XInt;
    double *YInt;
} OptStrat;

/* Declare the functions */
extern double calcXInt(ColStrat * col1, ColStrat * col2);

extern double getXinterval(ColStrat * newJ, double x1, ColStrat * list);

int getYint(double *payoff, int n, int j1, int j2, double *y1);

extern OptStrat * get2by2Strat(double * payoff, OptStrat * returnValue);

extern OptStrat * get2byNStrat(double * payoff, int cols, OptStrat * returnValue, int *
xInterval);
```

A3.4 strategies.c

```
/* Library file containing functions for solving 2x2 and 2xN matrix games */

#include "strategies.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

/* calculate the x value of the intercept between two lines
 */
double calcXInt(ColStrat * col1, ColStrat * col2) {
    return (getVal0(col1)-getVal0(col2))/(getGrad(col2)-getGrad(col1));

}

/* Function that calculates the end point of an interval for player I given
 * the start point and the line
 */
double getXinterval(ColStrat * newJ, double x1, ColStrat * list) {
    double tempX, xInt;

    /* find the next p intercept p2 that is less than 1 and greater than
     * p1
     */
    list = moveFirst(list);
    tempX = 1;
    while (list->next) {
        if (list != newJ) {
            xInt = calcXInt(newJ, list);
            if (xInt > x1 && xInt < tempX) {
                tempX = xInt;
            }
        }
        list = moveNext(list);
    }

    /* return the next intercept, or if there are no intercepts before p=1
     * return 1
     */
    return tempX;
}
```

```
/* Function that calculates the value for q for player II given the index of
 * the columns that are the two pure strategies the mixed strategy is
 * distributed over
 */
int getYint(double *payoff, int n, int j1, int j2, double *y1) {

    double yInt, row1Grad, row2Grad, row1_0, row1_1, row2_0, row2_1;

    /* create the 2x2 game that the original 2xN game has been reduced to
     */
    double a11 = payoff[0*n+j1];
    double a12 = payoff[0*n+j2];
    double a21 = payoff[1*n+j1];
    double a22 = payoff[1*n+j2];

    /* Find the value of the intercept between the two lines generated by the
     * rows of the matrix
     */
    yInt = (a22-a12)/((a11-a12)-(a21-a22));

    /* Calculate the value of the gradients of the lines and their values at
     * 0 and 1
     */
    row1Grad = a11-a12;
    row2Grad = a21-a22;
    row1_0 = a12; row1_1 = a11;
    row2_0 = a22; row2_1 = a21;

    /* If the two lines intercept outside the interval y=[0,1] then it is
     * necessary to examine the gradients of the lines to determine which
     * generates the upper line and whether it generates a value for y of 0 or
     * 1. It is not possible for the gradient of either of the lines to be 0
     * since this interval strategy for player II would mean a pure strategy
     * for player I. If this was the case then a pure strategy for player II
     * would already have been found
     */
    if (yInt<=0) {
        if (row1Grad > row2Grad) {
            if (row1Grad > 0) {
                *y1=0;
            } else if (row1Grad < 0) {
                *y1=1;
            }
        } else {
            if (row2Grad > 0) {
                *y1=0;
            } else if (row2Grad < 0) {
                *y1=1;
            }
        }
    } else if (yInt >= 1) {
        if (row1Grad < row2Grad) {
            if (row1Grad > 0) {
                *y1=0;
            } else if (row1Grad < 0) {
                *y1=1;
            }
        }
    } else {
        *y1 = yInt;
    }
```

```
    // The function was successful, return 1
    return 1;
}

/* Function that calculates and returns the saddle point or optimal mixed
 * strategies for a 2x2 game
 */
OptStrat * get2by2Strat(double * payoff, OptStrat * returnValue) {
    int m = 2, n=2;
    int a1, a2, b1, b2, c, x1, y1;
    int d1, d2, e1, e2, f, x2, y2;
    int saddle=1;
    double divisor;
    /* Allocate the memory for the returnValue object */
    if (!(returnValue = (OptStrat *)malloc(sizeof(OptStrat)))) {
        printf("Error allocating memory\n");
        return NULL;
    }
    if (!(returnValue->XOpt = (double *)malloc(sizeof(double)*2))) {
        printf("Error allocating memory\n");
        free(returnValue);
        return NULL;
    }
    if (!(returnValue->YOpt = (double *)malloc(sizeof(double)*2))) {
        printf("Error allocating memory\n");
        free(returnValue);
        free(returnValue->XOpt);
        return NULL;
    }

    /* find player 1's maximin
     * Set a1 and b1 to be -1 if the minimum values in rows 0 and 1 cannot be
     * found, 0 if they are in column 0, 1 if they are in column 1
     */
    a1 = payoff[0*n+0] < payoff[0*n+1] ? 0 : (payoff[0*n+1] < payoff[0*n+0] ? 1 : -1 );
    b1 = payoff[1*n+0] < payoff[1*n+1] ? 0 : (payoff[1*n+1] < payoff[1*n+0] ? 1 : -1 );

    /* set a2 and b2 to be 1 if there is no minimum in the row, or a1 and b1
     * if there is
     */
    a2 = a1 == -1 ? 1 : a1;
    b2 = b1 == -1 ? 1 : b1;

    /* set c to be player 1's maximin
     */
    c = payoff[0*n+a2] > payoff[1*n+b2] ? 0 : (payoff[1*n+b2] > payoff[0*n+a2] ? 1 : -1);
    switch(c) {
        case -1:
            //Player 1 has no maximin; no pure strategies
            saddle = 0;
            break;
        case 0:
            if (a1 == -1) {
                //Player 1 has no maximin; no pure strategies
                saddle = 0;
            } else {
                //saddle point for player 1 is [c,a1] so x1 = c, y1 = a1
                x1 = c; y1 = a1;
                saddle = 1;
            }
            break;
        case 1:
            if (b1 == -1) {
                //Player 1 has no maximin; no pure strategies
                saddle = 0;
            } else {
                //saddle point for player 1 is [c,b1] so x1 = c, y1 = b1
                x1 = c, y1 = b1;
                saddle = 1;
            }
            break;
        default:
            printf("Error calculating saddle point\n");
            saddle = 0;
            break;
    }

    if (saddle) {
        /* find player 2's minimax
         * Set d1 and e1 to be -1 if the maximum values in cols 0 and 1 cannot be
         * found, 0 if they are in row 0, 1 if they are in row 1
         */
        d1 = payoff[0*n+0] > payoff[1*n+0] ? 0 : (payoff[1*n+0] > payoff[0*n+0] ? 1 : -1);
        e1 = payoff[0*n+1] > payoff[1*n+1] ? 0 : (payoff[1*n+1] > payoff[0*n+1] ? 1 : -1);

        /* set d2 and e2 to be 1 if there is no maximum in the col, or d1 and e1
         * if there is
         */
        d2 = d1 == -1 ? 1 : d1;
        e2 = e1 == -1 ? 1 : e1;

        /* set f to be player II's maximin
         */
        f = payoff[d2*n+0] < payoff[e2*n+1] ? 0 : (payoff[e2*n+1] < payoff[d2*n+0] ? 1 : -1);

        switch (f) {
            case -1:
                //Player II has no minimax; no pure strategies
                saddle = 0;
                break;
            case 0:
                if (d1 == -1) {
                    //Player II has no minimax; no pure strategies
                    saddle = 0;
                } else {
                    //saddle point for player 2 is [d1,f] so x2 = d1, y2 = f
                    x2 = d1; y2 = f;
                    saddle = 1;
                }
                break;
            case 1:
                if (e1 == -1) {
                    //Player II has no minimax; no pure strategies
                    saddle = 0;
                } else {
                    //saddle point for player 2 is [e1,f] so x2 = e1, y2 = f
                    x2 = e1; y2 = f;
                    saddle = 1;
                }
                break;
            default:
                printf("Error calculating saddle point\n");
                saddle = 0;
                break;
        }
    }
```

```
    /* check that player 1's maximin is the same as player 2's minimax over
     * the pure srategies. If it is then this is the saddle point.
     */
    if ((x1 == x2) && (y1 == y2) && saddle) {
        //set strategies and value of the game and return the OptStrat object
        returnValue->XOpt[x1] = 1;
        returnValue->YOpt[y1] = 1;
        returnValue->value = payoff[x1*2+y1];
        returnValue->XInt = 0;
        returnValue->YInt = 0;
        return returnValue;
    } else {
        /* if there is no saddle point then calculate the optimal mixed
         * strategies for each player, set the value of the game and return the
         * OptStrat object
         */
        divisor = (payoff[1*n+1]-payoff[1*n+0]+payoff[0*n+0]-payoff[0*n+1]);
        returnValue->XOpt[0] = (payoff[1*n+1]-payoff[1*n+0])/divisor;
        returnValue->XOpt[1] = (payoff[0*n+0]-payoff[0*n+1])/divisor;
        returnValue->YOpt[0] = (payoff[1*n+1]-payoff[0*n+1])/divisor;
        returnValue->YOpt[1] = (payoff[0*n+0]-payoff[1*n+0])/divisor;
        returnValue->value = (payoff[0*n+0]*payoff[1*n+1]-payoff[0*n+1]*payoff[1*n+0])/divisor;

        returnValue->XInt = 0;
        returnValue->YInt = 0;
        return returnValue;
    }
}

/* Function that calculates and returns the optimal mixed strategies for a
 * 2xN game
 */
OptStrat * get2byNStrat(double * payoff, int cols, OptStrat * returnValue, int * xInterval) {
    ColStrat * list, *newJ, *prevJ, *firstJ, *listStart;
    double
        val0, grad, minVal0,
        x, tempMinX, xInt, tempVal0, tempGrad,
        x1, x2, y1, y2, value;
    int j, i, j1, j2;
    int found;

    /* Allocate the memory for the returnValue object including space for
     * interval strategies
     */
    if (!(returnValue = (OptStrat *)malloc(sizeof(OptStrat)))) {
        printf("Error allocating memory\n");
        return NULL;
    }
    if (!(returnValue->XOpt = (double *)malloc(sizeof(double)*2))) {
        printf("Error allocating memory\n");
        free(returnValue);
        return NULL;
    }
    if (!(returnValue->YOpt = (double *)malloc(sizeof(double)*cols))) {
        printf("Error allocating memory\n");
        free(returnValue->XOpt);
        free(returnValue);
        return NULL;
    }
    if (!(returnValue->XInt = (double *)malloc(sizeof(double)*2))) {
        printf("Error allocating memory\n");
        free(returnValue->XOpt);
        free(returnValue->YOpt);
        free(returnValue);
        return NULL;
    }
    if (!(returnValue->YInt = (double *)malloc(sizeof(double)*cols))) {
        printf("Error allocating memory\n");
        free(returnValue->XOpt);
        free(returnValue->YOpt);
        free(returnValue->XInt);
        free(returnValue);
        return NULL;
    }

    /* create a list to hold the straight lines generated by each column of
     * the 2xN matrix
     */
    //create an empty list
    list = newList();
    //add each column to the list
    for (j=0; j<cols; j++) {
        val0 = payoff[1*cols+j];
        grad = payoff[0*cols+j]-payoff[1*cols+j];
        list = addItem(list, j, val0, grad);
    }

    /* find the column with the minimum value at x=0
     */
    //initialise the variables
    list = moveFirst(list);
    minVal0=getVal0(list);
    grad = getGrad(list);
    firstJ = list;

    //find column with minimum value at x=0
    while (list->next) {
        list = moveNext(list);
        tempVal0 = getVal0(list);
        tempGrad = getGrad(list);
        if  ((tempVal0 < minVal0) || (tempVal0 == minVal0 && tempGrad < grad)) {
            firstJ = list;
            minVal0 = tempVal0;
            grad = tempGrad;
        } else if ((tempVal0>minVal0)&&((tempGrad+tempVal0)>(grad+minVal0)) ) {
            //discard lines which are above another line for the interval [0,1]
            list = discItem(list);
            if (list->prev)
                list = movePrev(list);
        }
    }

    /* calculate the optimal value for x
     */
    //initialise the variables
    list = moveFirst(list);
    x=0; tempMinX=1; found=0; prevJ=firstJ; newJ=firstJ;
    /* If the gradient of the first line is negative then the optimal value
     * for x is 1
     */
    if (grad < 0) {
        found = 1;
        tempMinX = 0;
    /* If the gradient of the first line is zero then the optimal value for x
     * is an interval on this line. Calculate the next intercept along this
     * line to find the end point of the interval and set the interval flag
```

```
     */
    } else if (grad == 0) {
        x1 = 0;
        x2 = getXinterval(newJ, x1, list);
        * xInterval = 1;
        found = 1;
    }
    /* If the optimal value for x has not yet been found (ie the gradient of
     * the line is greater than zero) then find the two lines that intersect
     * at the highest point
     */
    while (!(found)) {
        int ok=0;
        if (list) ok=1;
        list = moveFirst(list);
        while (ok) {
            if (list != prevJ) {
                xInt = calcXInt(prevJ, list);
                if ((xInt<tempMinX)&& (getGrad(list)<getGrad(prevJ))&& (xInt>x)){
                    tempMinX = xInt;
                    newJ = list;
                }
            }
            if (list->next) {
                list = moveNext(list);
            } else {
                ok=0;
            }
        }

        /* If there is no intercept before x=1 then the optimal value for x is
         * 1. Player II's pure strategy is the line prevJ/newJ
         */
        if (prevJ==newJ && x==1) {
            found = 1;

        /* If the gradient of the new line is zero then the optimal value for x
         * is an interval on this line. Calculate the next intercept along this
         * line to find the end point of the interval and set the interval flag
         * Player II's pure strategy is the line newJ
         */
        } else if (getGrad(newJ) == 0) {
            x1 = tempMinX;
            x2 = getXinterval(newJ, x1, list);
            * xInterval = 1;
            found = 1;

        /* If the gradient of the new line is zero then the optimal value for x
         * is the intercept just calculated. The two lines prevJ and newJ are
         * the two pure strategies that contribute to player II's mixed
         * strategy
         */
        } else if (getGrad(newJ) < 0) {
            found = 1;

        /* If the gradient of the new line is greater than zero and the
         * intercept is less than 1, continue finding the next intercept
         */
        } else {
            prevJ = discItem(prevJ);
            prevJ = newJ;
            x = tempMinX;
            tempMinX = 1;
        }
    }

    if ((* xInterval)) {
        /* have values for x1, x2 for player I's interval, and the columm
         * number for player II's pure strategy
         */
        returnValue->XInterval = 1;
        returnValue->XOpt[0] = x1;
        returnValue->XOpt[1] = 1-x1;
        returnValue->XInt[0] = x2;
        returnValue->XInt[1] = 1-x2;
        for (i=0; i<cols; i++) {
            if (i==getJ(newJ)) {
                returnValue->YOpt[i] = 1;
            } else {
                returnValue->YOpt[i] = 0;
            }
        }
    } else if (tempMinX == 0 || tempMinX == 1) {
        /* Player I has a pure strategy, therefore Player II must also have a
         * pure strategy ~ it will be the best response to player I's pure
         * strategy
         */
        returnValue->XOpt[0] = tempMinX;
        returnValue->XOpt[1] = 1-tempMinX;
        for (i=0; i<cols; i++) {
            if (i==getJ(newJ)) {
                returnValue->YOpt[i] = 1;
            } else {
                returnValue->YOpt[i] = 0;
            }
        }

        returnValue->XInt = 0;
        returnValue->YInt = 0;
    } else {
        /* the two lines intersecting at the maximum have been found. Use these
         * to determine player II's optimal strategy
         */
        j1 = getJ(newJ);
        j2 = getJ(prevJ);
        //(* yInterval) = 0; y1=-1; y2=-1;
        if (!(getYint(payoff, cols, j1,j2, &y1))) {
            printf("Error calculating player II's strategy");
            free(returnValue->XOpt);
            free(returnValue->YOpt);
            free(returnValue->XInt);
            free(returnValue->YInt);
            free(returnValue);
            list = moveFirst(list);
            while (list) {
                list = discItem(list);
            }
            return NULL;
        }

        /* set the values of the optimal strategy to return
         */
        //set player I's optimal strategy
        returnValue->XOpt[0]= tempMinX;
        returnValue->XOpt[1]= 1-tempMinX;
```

```
        //set player II's optimal strategy
        for (i=0; i<cols; i++) {
            if (i==j1) {
                returnValue->YOpt[i] = y1;
                //if ((* yInterval)) { returnValue->YInt[i] = y2; }
            } else if (i == j2) {
                returnValue->YOpt[i] = 1 - y1;
                //if (* yInterval) { returnValue->YInt[i] = 1-y2; }
            } else {
                returnValue->YOpt[i] = 0;
                //if (* yInterval) { returnValue->YInt[i] = 0; }
            }
        }

        returnValue->XInt = 0;
        returnValue->YInt = 0;
    }

    //set the value of the game
    value = 0;
    for (i = 0; i<2; i++) {
        for (j=0; j<cols; j++) {
            value = value + returnValue->XOpt[i]*payoff[i*cols+j]*returnValue->YOpt[j];
        }
    }
    returnValue->value = value;

    /* free the memory
     */
    list = moveFirst(list);
    while (list) {
        list = discItem(list);
    }

    /* return the OptStrat object
     */
    return returnValue;
}
```

A3.5 simplex.h

```
/* header file for simplex.c*/


#include "strategies.h"

/* Define the data structure simpEl which will be an element in the simplex
 * tableau
 */
typedef struct simplex_matrix_element {
    /* an integer that is 0 if payoff element, 1 if player I strategy,
     * 2 if player II strategy, 3 if border value
     */
    int playStrat;

    /* an integer that stores strategy index */
    int index;

    /* a double that stores payoff matrix value */
    double val;
} simpEl;
```

```
/* Declare the functions */
extern int findNegCol(int f, simpEl * matrix, int m, int n);

extern int findZeroBorder(int pvtRC, simpEl * matrix, int m, int n);

extern int findMinRatInCol(int col, simpEl * matrix, int m, int n);

extern int findMinRatInRow(int row, simpEl * matrix, int m, int n);

extern int swap(int f, int h, simpEl * matrix, int n);

extern int testPivot(simpEl * matrix, int m, int n, int f, int h);

extern int pivotMatrix(simpEl * matrix, int m, int n, int f, int h);

extern int createSimpMat(double * payoff, simpEl * matrix, int m, int n, double k);

extern OptStrat * addStrategy(simpEl * matrix, int m, int n, int k, OptStrat * returnValue, int
* noStrat);

extern OptStrat * getMbyNStrat(double * payoff,int m, int n, OptStrat * returnValue, int *
noStrat);
```

A3.6 simplex.c

```
/* File containing functions to solve mxn games using the simplex method

 */

#include "simplex.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

/* function that returns the column index of the first column after column
 * f with a negative value in the border row. Returns 0 if there are no such
 * columns
 */
int findNegCol(int h, simpEl * matrix, int m, int n) {
    int j;
    for (j = h+1; j<=n;j++) {
        if (matrix[(m+1)*(n+2)+j].val < 0) {
            return j;
        }
    }
    return 0;
}

/* Column indices are 1->n, row indices are (n+1)->(m+n)
 * The function returns the index of the first column/row after
 * index pvtRC with zero in the bottom/right element.
 */
int findZeroBorder(int pvtRC, simpEl * matrix, int m, int n) {
    int i, j,start;
    double borderVal;

    /* loop through cols, checking the border row at the bottom of the matrix.
     * If pvtRC refers to a row then it will be greater than or equal to n so
     * this code will not be executed
```

```
     */
    for (j=pvtRC+1; j<=n; j++) {
        borderVal = matrix[(m+1)*(n+2)+j].val;
        /* due to rounding errors it is necessary to define an interval to test
         * rather than testing for an absolute value of zero. The size of the
         * interval is defined by the resolution of the numbers output
         */
        if ( borderVal < 0.0000005 && borderVal >= -0.0000005) {
            //If a zero is found then return the column index
            return j;
        }
    }

    /* If no zero was found in the border row or pvtRC referred to a row then
     * loop through rows, checking the border column at the right of the
     * matrix
     */
    start = pvtRC > n ? pvtRC - n + 1 : 1;
    for (i=start; i<=m; i++) {
        borderVal = matrix[i*(n+2)+(m+1)].val;
        if ( borderVal < 0.0000005 && borderVal >= -0.0000005) {
            //If a zero is found then return the row index + n
            return (i+n);
        }
    }

    /* if no zeroes are found then return 0 */
    return 0;
}


/* function that returns the row index i of the row with the minimum ratio
 * between the pivot [i,col] and border value at the right of the row [i,n+1]
 */
int findMinRatInCol(int col, simpEl * matrix, int m, int n) {
    int minRow, i;
    double minValue, newVal, pivotVal;

    minRow= 0;
    for (i=1; i<=m; i++) {
        pivotVal = matrix[i*(n+2)+col].val;
        if ( pivotVal > 0)
            newVal = matrix[i*(n+2)+(n+1)].val/pivotVal;
        if ((minRow==0 && pivotVal>0) || (minRow !=0 && newVal < minValue) ) {
            minValue = newVal;
            minRow = i;
        }
    }
    return minRow;
}

/* function that returns the column index j of the column with the minimum
 * ration between the pivot [row,j] and border value at the right of the row
 * [row, n+1]
 */
int findMinRatInRow(int row, simpEl * matrix, int m, int n) {
    int minCol, j;
    double minValue, newVal, pivotVal;

    minCol = 0;
    for (j=1; j<=n; j++) {
        pivotVal = matrix[row*(n+2)+j].val;
        if ( pivotVal > 0)
            newVal = matrix[row*(n+2)+(n+1)].val/pivotVal;
        if ((minCol==0 && pivotVal > 0) || (minCol !=0 && newVal < minValue)) {
            minValue = newVal;
            minCol = j;
        }
    }
    return minCol;
}



/* function that exchanges the label on the left of the pivot row with the
 * label on the top of the pivot column
 */
int swap(int f, int h, simpEl * matrix, int n) {
    simpEl temp = matrix[0*(n+2)+h];
    matrix[0*(n+2)+h] = matrix[f*(n+2)+0];
    matrix[f*(n+2)+0] = temp;
    return 1;
}

/* function that returns 0 if a pivot of matrix about the element matrix[f,h]
 * will result in negative values in either the bottom border row or the
 * right border column
 */
int testPivot(simpEl * matrix, int m, int n, int f, int h) {
    int i, j;

    /* Loop through cols, checking the border row at the bottom of the matrix
     * If any are negative then return 0
     */
    for (j=1; j<=n; j++) {

        if (j==h) {
            //same column as pivot
            if ((-1 * matrix[(m+1)*(n+2)+j].val / matrix[f*(n+2)+h].val) < 0) {
                return 0;
            }
        } else {
            //different row and column to pivot
            if ((matrix[(m+1)*(n+2)+j].val - (matrix[f*(n+2)+j].val * matrix[(m+1)*(n+2)+h].val
/ matrix[f*(n+2)+h].val)) < 0) {
                return 0;
            }
        }
    }

    /* Loop through rows, checking the border col at the right of the matrix
     * If any are negative then return 0
     */
    for (i=1; i<=m; i++) {
        if (i == f) {
            //same row as pivot
            if ((matrix[i*(n+2)+(n+1)].val / matrix[f*(n+2)+h].val) < 0) {
                return 0;
            }
        } else {
            //different row and column to pivot
            if (matrix[i*(n+2)+(n+1)].val - (matrix[f*(n+2)+(n+1)].val * matrix[i*(n+2)+h].val
/ matrix[f*(n+2)+h].val) < 0) {

                return 0;
            }
```

```
        }
    }

    /* no negative values have been found. Return 1
     */
    return 1;
}

/* function that pivots a simplex matrix about a given element
 */
int pivotMatrix(simpEl * matrix, int m, int n, int f, int h) {
    simpEl * tempMatrix;
    int i,j;
    double pivot, newVal;

    /* set the value of the pivot */
    pivot = matrix[f*(n+2)+h].val;

    /* store the old values in a temporary matrix so that the new values are
     * not calculated using values that have already been pivoted
     */
    if (!(tempMatrix = (simpEl *)malloc(sizeof(simpEl)*(m+2)*(n+2)))) {
        printf("Error allocating memory\n");
        return 0;
    }
    for (i=0; i<(m+2)*(n+2); i++) {
        tempMatrix[i].playStrat = matrix[i].playStrat;
        tempMatrix[i].index = matrix[i].index;
        tempMatrix[i].val = matrix[i].val;
    }

    //pivot each element
    for (i=1; i<=m+1; i++) {
        for (j=1; j<=n+1; j++) {
            if (i==f && j==h) {
                //the pivot point
                newVal = 1/pivot;
                if (newVal < 0.0000005 && newVal >= -0.0000005)
                    newVal = 0;
                matrix[i*(n+2)+j].val = newVal;
            } else if (j==h) {
                //same column as pivot point
                newVal = -1 * tempMatrix[i*(n+2)+j].val / pivot;
                if (newVal < 0.0000005 && newVal >= -0.0000005)
                    newVal = 0;
                matrix[i*(n+2)+j].val = newVal;
            } else if (i == f) {
                //same row as pivot point
                newVal = tempMatrix[i*(n+2)+j].val / pivot;
                if (newVal < 0.0000005 && newVal >= -0.0000005)
                    newVal = 0;
                matrix[i*(n+2)+j].val = newVal;
            } else {
                //all other payoff elements in different row and column
                newVal = tempMatrix[i*(n+2)+j].val - (tempMatrix[f*(n+2)+j].val *
tempMatrix[i*(n+2)+h].val / pivot);
                if (newVal < 0.0000005 && newVal >= -0.0000005)
                    newVal = 0;
                matrix[i*(n+2)+j].val = newVal;
            }
        }
    }

    //swap the labels on the far left and top of the pivot element
    swap(f,h,matrix,n);

    free(tempMatrix);

}

/* function that creates the simplex matrix from the payoff matrix
 */
int createSimpMat(double * payoff, simpEl * matrix, int m, int n, double k) {
    simpEl temp;
    int i, j;

    /* create a default simpEl element and fill in the blank values (the top
     * left, top right and bottom left corners) with this element
     */
    temp.playStrat = -1; temp.index = -1; temp.val = -1;
    matrix[0*(n+2)+0] = temp;
    matrix[0*(n+2)+(n+1)]= temp;
    matrix[(m+1)*(n+2)+0] = temp;

    //create the player I strategy column on the far left
    for (i=1; i<=m; i++) {
        temp.playStrat = 1;
        temp.index = i-1;
        matrix[i*(n+2)+0] = temp;
    }
    temp.playStrat = -1; temp.index = -1; temp.val = -1;

    //create the player II strategy row on the top
    for (j=1; j<=n; j++) {
        temp.playStrat = 2;
        temp.index = j-1;
        matrix[0*(n+2)+j] = temp;
    }
    temp.playStrat = -1; temp.index = -1; temp.val = -1;

    //create the border column of 1 down the right hand side
    for (i=1; i<=m; i++) {
        temp.playStrat = 3;
        temp.val = 1;
        matrix[i*(n+2)+(n+1)] = temp;
    }
    temp.playStrat = -1; temp.index = -1; temp.val = -1;

    //create the border row of -1 along the bottom
    for (j=1;j<=n; j++) {
        temp.playStrat = 3;
        temp.val = -1;
        matrix[(m+1)*(n+2)+j] = temp;
    }
    temp.playStrat = -1; temp.index = -1; temp.val = -1;

    //set the bottom right cell to 0
    matrix[(m+1)*(n+2)+(n+1)].playStrat = 3;
    matrix[(m+1)*(n+2)+(n+1)].index = -1;
    matrix[(m+1)*(n+2)+(n+1)].val = 0;

    //enter the payoff matrix values
    for (i=1; i<=m; i++) {
        for (j=1; j<=n; j++) {
            temp.playStrat = 0;
            temp.val = payoff[(i-1)*n+(j-1)]+k;
```

```
            matrix[i*(n+2)+j] = temp;
        }
    }
}

/* function that adds a new optimal strategy to an array of optimal
 * strategies
 */
OptStrat * addStrategy(simpEl * matrix, int m, int n, int k, OptStrat * returnValue, int *
noStrat) {
    OptStrat newValue, * tempArray;
    int i, j;

    /* add an empty space to the end of the array */
    if ((*noStrat)==0) {
        if (!(returnValue = (OptStrat *)malloc(((*noStrat)+1)*sizeof(OptStrat)))) {
            printf("Error allocating memory\n");
            return NULL;
        }
    } else {
        if (!(returnValue = (OptStrat *)realloc(returnValue, ((*noStrat)+1)*sizeof(OptStrat))))
{
            printf("Error allocating memory\n");
            return NULL;
        }
    }

    /* create the new strategy */
    if (!(newValue.XOpt = (double *)malloc(sizeof(double)*m))) {
        printf("Error allocating memory\n");
        return NULL;
    }
    if (!(newValue.YOpt = (double *)malloc(sizeof(double)*n))) {
        printf("Error allocating memory\n");
        free(newValue.XOpt);
        return NULL;
    }

    //There are no intervals with the simplex method
    newValue.XInterval=0;
    newValue.YInterval=0;

    //Set the value of the game

    newValue.value = 1 / matrix[(m+1)*(n+2)+(n+1)].val - k;

    /* Set player I's strategy*/
    //initialise XOpt to 0
    for (i=0; i<m; i++) {
        newValue.XOpt[i] = 0;
    }
    //enter probabilities for each strategy that is in the top row
    for (j=1; j<n+1; j++) {
        if (matrix[0*(n+2)+j].playStrat==1)
            newValue.XOpt[matrix[0*(n+2)+j].index] =
matrix[(m+1)*(n+2)+j].val/matrix[(m+1)*(n+2)+(n+1)].val;
    }

    /* Set player II's strategy */
    //initialise YOpt to 0
    for (j=0; j<n; j++) {
        newValue.YOpt[j] = 0;
    }
    //enter probabilities for each strategy that is in the left column
    for (i=1; i<m+1; i++) {
        if (matrix[i*(n+2)+0].playStrat==2)
            newValue.YOpt[matrix[i*(n+2)+0].index] =
matrix[i*(n+2)+(n+1)].val/matrix[(m+1)*(n+2)+(n+1)].val;
    }

    /* add the new value to the end of the array */
    returnValue[(*noStrat)] = newValue;

    /* increment the number of strategies */
    (*noStrat)++;

    /* return the array of strategies */
    return returnValue;
}


/* function that calculates and returns the optimal mixed strategies for an
 * m x n game using the simplex method
 */
OptStrat * getMbyNStrat(double * payoff,int m, int n, OptStrat * returnValue, int * noStrat) {
    double k;
    simpEl * matrix, * tempMatrix;
    int i,j,f,h,pvtRC, whileCounter=0;

    if (!(matrix = (simpEl *)malloc(sizeof(simpEl)*(m+2)*(n+2)))) {
        printf("Error allocating memory\n");
        return NULL;
    }

    /* calculate the value of k that is needed to be added to each element
     * of the payoff matrix so that each payoff element in the simplex
     * matrix is positive
     */
    k = 0;
    for (i=0; i<(m*n); i++) {
        if (payoff[i] < k)
            k = payoff[i];
    }
    k=abs(k);

    /* create simplex matrix */
    createSimpMat(payoff, matrix, m, n,k);

    /* Perform the required number of pivots to find the first optimal
     * strategy
     */
    while (findNegCol(0, matrix, m, n) != 0) {
        whileCounter++;

        h = 0;

        /* find a valid pivot value */
        do {
            h = findNegCol(h, matrix, m, n);
            f = findMinRatInCol(h, matrix, m, n);
        } while (!f);

        /* pivot the matrix */
        pivotMatrix(matrix, m, n, f, h);

    }
```

```
    /* add the strategy to the list */
    returnValue = addStrategy(matrix, m, n, k, returnValue, noStrat);

    /* find any other optimal strategies */
    pvtRC = 0;
    while (findZeroBorder(pvtRC, matrix, m, n) != 0) {
        whileCounter++;

        /* find a valid pivot that does not cause any elements in the border
         * row/column to be zero
         */
        do {
            pvtRC = findZeroBorder(pvtRC, matrix, m, n);
            if (pvtRC <= m) {
                h = pvtRC;
                f = findMinRatInCol(h, matrix, m, n);
            } else {
                f = pvtRC;
                h = findMinRatInRow(f, matrix, m, n);
            }
        } while(!testPivot(matrix, m, n, f, h));

        /* check that f and h are in valid rows/columns */
        if (f<1 || f>m || h<1 || h>n) {
            break;
        }

        //just pivot a temporary matrix
        if (!(tempMatrix = (simpEl *)malloc(sizeof(simpEl) * ((m+2)*(n+2))))) {
            printf("Error allocating memory\n");
            free(matrix);
            return NULL;
        }
        tempMatrix = (simpEl *)memcpy(tempMatrix, matrix, ((m+2)*(n+2))*sizeof(simpEl));
        pivotMatrix(tempMatrix, m, n, f, h);

        /* add the strategy to the list */
        returnValue = addStrategy(tempMatrix, m, n, k, returnValue, noStrat);

        /* free the memory for the temporary matrix */
        free(tempMatrix);
    }

    /* free the memory for the simplex tableau */
    free(matrix);

    /* return the array of strategies */
    return returnValue;
}
```

A3.7 functions.h

```
/* header file for functions.c */


#include "simplex.h"

/* Declare the functions */
extern char * getFileName(int argc, char **argv);

extern double * makePayoffMatrix(double * payoff, int argc, char **argv, int *rows, int *cols,
int * file);


extern void freeOptStratArray(OptStrat * array, int noStrat);

extern int transposeMatrix(double * payoff, int * rows, int * cols);

extern int swapStrat(OptStrat * mixed, int * rows, int * cols,int *xInterval, int *yInterval);
```

A3.8 functions.c

```
/* Library file containing miscellaneous functions that do not belong in any

 * other modules
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "functions.h"

char * getFileName(int argc, char **argv) {
    char * fileName;
    int i;

    if (!(fileName = (char *)malloc(sizeof(char)*(strlen(argv[1]-1))))) {
        printf("Error allocating memory\n");
        return NULL;
    }
    for (i=1; i<=(strlen(argv[1])); i++) {
        fileName[i-1] = argv[1][i];
    }

    return fileName;
}

/* create and store the payoff matrix from the input
 */
double * makePayoffMatrix(double * payoff, int argc, char **argv, int *rows, int *cols, int *
file) {
    int i,j,m,n;
    char * input,* temp, *temp2, w;
    double k;
    FILE *fp_in;
    char * fileName;

    input="";

    /* if the first character is : then read from file */
    if (argv[1][0] == ':') {
        (*file)=1;

        if (!(fileName = getFileName(argc, argv))) {
            return NULL;
        }
        if (!(fp_in = fopen(fileName, "a+"))) {
            printf("Error opening file\n");
            free(fileName);
            return NULL;
        }
        if (!(input = (char *)malloc(sizeof(fp_in)))) {
```

```
        printf("Error allocating memory\n");
        free(fileName);
        if (fp_in != stdout) fclose(fp_in);
        return NULL;
    }
    i=0;
    while((w=fgetc(fp_in)) != EOF) {
        if (!isspace(w)) {
            input[i] = w;
            i++;
        }
    }
    free(fileName);

} else {
    (*file)=0;
    fp_in = stdout;

    /* create single word of arguments with no whitespace*/
    for (i=1; i<argc; i++) {
        if (i==1)
            input = (char *)malloc(sizeof(char)*strlen(argv[i]));
        else
            input = (char *)realloc(input, sizeof(char)*(strlen(input)+strlen(argv[i])));

        input = (char *)strcat(input, argv[i]);
    }
}
temp = input;

/*separate the input from the output */
fprintf(fp_in, "\n");

//assign m
m = strtol(temp, &temp2, 10);
*rows = m;
if (temp == temp2 || m<= 0) {
    //couldn't assign m. report error and quit
    fprintf(fp_in, "Invalid entry for m (number of rows)\n");
    free(input);
    if (fp_in != stdout) fclose(fp_in);
    return NULL;
} else {
    temp = temp2;
}

//check for comma
if (temp[0] != ',') {
    fprintf(fp_in, "No comma after m (number of rows)\n");
    free(input);
    if (fp_in != stdout) fclose(fp_in);
    return NULL;
} else {
    temp++;
}
//assign n
n = strtol(temp, &temp2, 10);
*cols = n;
if (temp == temp2 || n<= 0) {
    //couldnt assign n. report error and quit
    fprintf(fp_in, "Invalid entry for n (number of columns)\n");
    free(input);
    if (fp_in != stdout) fclose(fp_in);
        return NULL;
    } else {
        temp = temp2;
    }

    /* allocate the memory for the payoff matrix */
    if (!(payoff = (double *)malloc(sizeof(double)*m*n))) {
        printf("Error allocating memory\n");
        free(input);
        if (fp_in != stdout) fclose(fp_in);
        return NULL;
    }


    //check for comma, open square bracket
    if (temp[0] != ',' || temp[1] != '[' ) {
        fprintf(fp_in, "Invalid start to payoff matrix. Require comma, double open square
bracket\n");
        free(input);
        free(payoff);
        if (fp_in != stdout) fclose(fp_in);
        return NULL;
    } else {
        temp = temp+2;
    }

    //loop through each row
    for (i=0; i<m; i++) {
        //each row must start with '['
        if (temp[0] != '['){
            fprintf(fp_in, "Invalid entry, each row must start with '['\n");
            free(input);
            free(payoff);
            if (fp_in != stdout) fclose(fp_in);
            return NULL;
        } else {
            temp++;

            //loop through columns
            for (j=0; j<n; j++) {
                //if not first element, should be preceded by comma
                if (j > 0) {
                    if (temp[0] != ',') {
                        fprintf(fp_in, "Invalid entry at element[%d,%d], each element must be
separated by ','\n", i,j);
                            free(input);
                            free(payoff);
                            if (fp_in != stdout) fclose(fp_in);
                            return NULL;
                    } else {
                        temp++;
                    }
                }
                k = strtod(temp, &temp2);
                if (temp == temp2) {
                    //couldn't convert to double. report error and quit
                    fprintf(fp_in, "Invalid data entry for element [%d,%d]\n", i,j);
                        free(input);
                        free(payoff);
                        if (fp_in != stdout) fclose(fp_in);
                        return NULL;
                } else {
                    temp = temp2;
```

```
                    payoff[i*n+j] = k;
                }
            }
        }

        //each row must end with ']'
        if (temp[0] != ']') {
            fprintf(fp_in, "Invalid entry, each row must end with ']'\n");
            free(input);
            free(payoff);
            if (fp_in != stdout) fclose(fp_in);
            return NULL;
        } else {
            temp++;
        }
    }
    //free the input array
    free(input);

    /* close the output file */
    if (fp_in != stdout) fclose(fp_in);

    //return the matrix
    return payoff;
}

/* Function that frees the memory reserved for the array of optimal
 * strategies
 */
void freeOptStratArray(OptStrat * array, int noStrat) {
    int i;

    //loop through the array freeing all reserved memory
    if (noStrat > 0) {
        for (i=0; i<noStrat; i++) {
            free((array+i)->XOpt);
            free((array+i)->YOpt);
            if ((array+i)->XInterval) free((array+i)->XInt);
            if ((array+i)->YInterval) free((array+i)->YInt);
            free((array+i));
        }
        free(array);
    }
}

/* Function that takes a Mx2 payoff matrix and converts it into an equivalent
 * 2xN payoff matrix, ie the negative transpose,
 */
int transposeMatrix(double * payoff, int * rows, int * cols) {
    double * tempPayoff;
    int temp,i,j;
    int m = *rows, n = *cols;

    //create a temporary structure to store the new matrix
    if (!(tempPayoff = (double *)malloc(sizeof(double)*m*n))) {
        return 0;
    }

    //swap the elements
    for (i=0; i<m; i++) {
        for (j=0; j<n; j++) {
            tempPayoff[j*m+i] = -1 * payoff[i*n+j];
        }
    }

    //store the swapped elements in the original matrix
    for (i=0;i<m*n; i++) {
        payoff[i] = tempPayoff[i];
    }

    //swap the number of rows and columns
    temp = *rows;
    *rows= *cols;
    *cols = temp;

    //free the temporary matrix
    free(tempPayoff);
}

/* Function that takes an OptStrat object for a 2xN game and converts it to
 * an OptStrat object for the equivalent Mx2 game
 */
int swapStrat(OptStrat * mixed, int * rows, int * cols,int *xInterval, int *yInterval) {
    OptStrat temp;
    int tempInt;
    int m = *rows, n = *cols;

    //allocate the memory for the temporary variable
    if (!(temp.XOpt = (double *)malloc(sizeof(double)*n))) {
        printf("Error allocating memory\n");
        return 0;
    }
    if (!(temp.YOpt = (double *)malloc(sizeof(double)*m))) {
        printf("Error allocating memory\n");
        free(temp.XOpt);
        return 0;
    }
    if (!(temp.YInt = (double *)malloc(sizeof(double)*m))) {
        printf("Error allocating memory\n");
        free(temp.XOpt);
        free(temp.YOpt);
        return 0;
    }

    //move the values to the new section of memory
    temp.XOpt = (double *)memmove(temp.XOpt, mixed->YOpt, sizeof(double)*n);
    temp.YOpt = (double *)memmove(temp.YOpt, mixed->XOpt, sizeof(double)*m);
    if ((*xInterval)) temp.YInt = (double *)memmove(temp.YInt, mixed->XInt, sizeof(double)*m);
    temp.value = -1*mixed->value;

    //reallocate the original memory
    mixed->XOpt = (double *)realloc(mixed->XOpt, sizeof(double)*n);
    free(mixed->XInt);
    mixed->YOpt = (double *)realloc(mixed->YOpt, sizeof(double)*m);
    if ((*xInterval)) mixed->YInt = (double *)realloc(mixed->YInt, sizeof(double)*m);

    //move the new values to the original memory
    mixed->XOpt = (double *)memmove(mixed->XOpt, temp.XOpt, sizeof(double)*n);
    mixed->YOpt = (double *)memmove(mixed->YOpt, temp.YOpt, sizeof(double)*m);
    if ((*xInterval)) mixed->YInt = (double *)memmove(mixed->YInt, temp.YInt, sizeof(double)*m);

    //swap the number of rows and columns
    tempInt = *rows;
    *rows= *cols;
    *cols = tempInt;
```

```
    //swap the interval flags
    tempInt = *xInterval;
    *xInterval = *yInterval;
    *yInterval = tempInt;

    //multiply the value of the game by -1
    mixed->value= mixed->value * -1;

    //free the memory allocated for temp
    free(temp.XOpt); free(temp.YOpt); free(temp.XInt); free(temp.YInt);
}
```

A3.9 solveGame.c

```
/* main function to solve any mxn two-person zero-sum matrix game by
 * calling functions in other modules
 */

#include "functions.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>


main(int argc, char **argv) {

    double * payoff;
    int matrixTransposed = 0;
    OptStrat * mixed;
    int m,n,i,j, saddle, noStrat, xInterval=0, yInterval=0, file;
    char * fileName;
    FILE * fp_out;

    /* Generate the payoff matrix */
    if (!(payoff = makePayoffMatrix(payoff, argc, argv, &m, &n, &file))) {
        exit(0);
    }

    /* Set the output stream depending on the input method - file or screen */
    if (file) {
        /* open the file */
        if (!(fileName = getFileName(argc, argv))) {
            free(payoff);
            exit(0);
        }

        if (!(fp_out = fopen(fileName, "a"))) {
            printf("Error opening file\n");
            free(fileName);
            free(payoff);
            exit(0);
        }
        free(fileName);
    } else {
        fp_out = stdout;
    }
    /* append the matrix to the file */
    fprintf(fp_out, "\nPayoff matrix :=\n");
    for (i=0; i<m; i++) {
        fprintf(fp_out, "\t[%lf", payoff[i*n+0]);
        for (j=1; j<n; j++) {
            fprintf(fp_out, ", %lf", payoff[i*n+j]);
        }
        fprintf(fp_out, "]\n");
    }
    fprintf(fp_out, "\n");

    /*for mx2 games transpose the matrix */
    if (m>2 && n==2) {
        //transpose matrix, multiply all values by -1 and set the flag
        if (!transposeMatrix(payoff, &m, &n)) {
            free(payoff);
            exit(0);
        } else {
            //set matrixTransposed
            matrixTransposed = 1;
        }
    }

    if (m<2 || n<2) {
        //matrix is too small
        fprintf(fp_out, "Matrix is too small for this system\n");
        mixed = NULL;
    } else if (m==2 && n==2) {
        //call function to solve the 2x2 game. There will be only one strategy
        mixed = get2by2Strat(payoff, mixed);
        noStrat=1;
    } else if (m == 2 && n > 2) {
        /* call function to solve the 2xN game. There will be only one strategy
         * although this may be an interval strategy
         */
        mixed = get2byNStrat(payoff, n, mixed, &xInterval);
        noStrat=1;
    } else if (m>2 && n>2) {
        /* call function to solve the MxN game. There may be more than one
         * strategy
         */
        mixed = getMbyNStrat(payoff, m, n, mixed, &noStrat);
    }

    /* If there was an error calculating the optimal strategies then free the
     * memory and exit the system
     */
    if (!mixed) {
        free(payoff);
        exit(0);
    }

    /* If matrixTransposed is set, swap the XOpt and YOpt strategies and
     * multiply value by -1
     */
    if (matrixTransposed) {
        for (i=0; i<noStrat; i++) {
            if (!swapStrat(mixed, &m, &n, &xInterval, &yInterval)) {
                free(payoff);
                freeOptStratArray(mixed, noStrat);
                exit(0);
            }
        }
    }

    /* append the output to the file */
    if (xInterval || yInterval) {
        fprintf(fp_out, "There are interval strategies\n");
```

```
    } else if (noStrat == 1) {
        fprintf(fp_out, "There is 1 strategy\n");
    } else {
        fprintf(fp_out, "There are %d strategies\n", noStrat);
    }
    for (i=0; i<noStrat; i++) {
        fprintf(fp_out, "Strategy %d:\n",i+1);
        fprintf(fp_out, "   Player I  Optimal Strategy: [");
        for (j=0; j<m; j++) {
            if (j != 0) fprintf(fp_out, ", ");
            if (xInterval && ((mixed+i)->XOpt[j] != (mixed+i)->XInt[j])) {
                fprintf(fp_out, "(%lf-%lf)", (mixed+i)->XOpt[j], (mixed+i)->XInt[j]);
            } else {
                fprintf(fp_out, "%lf", (mixed+i)->XOpt[j]);
            }
        }
        fprintf(fp_out, "]\n");
        fprintf(fp_out, "   Player II Optimal Strategy: [");
        for (j=0; j<n; j++) {
            if (j != 0) fprintf(fp_out, ", ");
            if (yInterval && ((mixed+i)->YOpt[j] != (mixed+i)->YInt[j])) {
                fprintf(fp_out, "(%lf-%lf)", (mixed+i)->YOpt[j], (mixed+i)->YInt[j]);
            } else {
                fprintf(fp_out, "%lf", (mixed+i)->YOpt[j]);
            }
        }
        fprintf(fp_out, "]\n");
        fprintf(fp_out, "   Value of the game: %lf\n\n", (mixed+i)->value);
    }

    if (file) fclose(fp_out);

    //free the memory
    free(payoff);
    freeOptStratArray(mixed, noStrat);
}
```

A3.10 LinkListTest.c

```
/*File to test the linked list library

 */

#include "LinkListLib.h"

main() {
    ColStrat * list;
    int i, j;

    //create a new empty list
    printf("Creating new empty list\n");
    list = newList();

    //add some items to the new list
    printf("Adding items to the list\n");
    for(i=1; i<=5; i++){
        list = addItem(list, i, 1.5*(double)i, 2.5*(double)i);
        printf("\tItem %d: j = %d; val0 = %lf; grad = %lf\n", i, getJ(list), getVal0(list),
getGrad(list));
    }

    //print the items in the list
    printf("Printing the list\n", getJ(list));
    list = moveFirst(list);
    for(i=1; i<=5; i++) {
        printf("\tItem %d: j = %d; val0 = %lf; grad = %lf\n", i, getJ(list), getVal0(list),
getGrad(list));
        if (list->next) list = moveNext(list);
    }

    //discard item j from the list
    j=2;
    printf("Discarding element %d from the list\n", j);
    list = moveFirst(list);
    for (i=1; i<j; i++) {
        list = moveNext(list);
    }
    list = discItem(list);

    //print the items in the list
    printf("Printing the list\n");
    list = moveFirst(list);
    for(i=1; i<=4; i++) {
        printf("\tItem %d: j = %d; val0 = %lf; grad = %lf\n", i, getJ(list), getVal0(list),
getGrad(list));
        if (list->next) list = moveNext(list);
    }

    //print the last item in the list
    printf("Printing the last item in the list\n");
    list = moveLast(list);
    printf("\tLast item: j = %d; val0 = %lf; grad = %lf\n", getJ(list), getVal0(list),
getGrad(list));

    //printf the previous item in the list
    printf("Printing the previous item in the list\n");
    list = movePrev(list);
    printf("\tLast item: j = %d; val0 = %lf; grad = %lf\n", getJ(list), getVal0(list),
getGrad(list));

    //delete the list
    printf("Deleting the list\n");
    list = moveFirst(list);
    while (list) {
        list = discItem(list);

    }
}
```