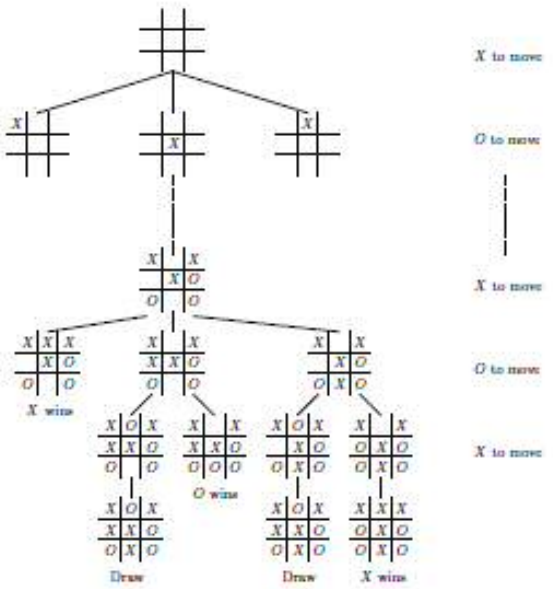
The Theory of Games

Andrea Schalk
A.Schalk@manchester.ac.uk
School of Computer Science
University of Manchester



Preface

A note to students of COMP34111 from 2021 onward from Jonathan Shapiro

What follows is the 2013 version of a set of lecture notes which was produced for the first semester (or more specifically weeks 2 – 5) of COMP34120 by Andrea Schalk. It is based on previous versions, and also is a shortened and slightly refocused version of a set of notes for a full-semester course on the theory of games which Dr. Schalk used to teach. The course COMP34120 was devised by Andrea, Xiao-Jun Zeng, and me, and these notes were written by Andrea to support her part of the course.

Now Andrea has moved on to other teaching, and I have taken over her part of the course. Andrea has very generously agreed to allow me to continue to use her lecture notes, and as I think they are extremely good, and very appropriate for this course, I intend to use them as a textbook. COMP34120 has spit into two one-semester courses, and this course, COMP34111, is related to the first semester of COMP34120.

The section listed in the Table of Contents as "About this course" has been largely removed. This contained a schedule of the course and other information specific to that course, but irrelevant to this one. What remains is pages 2 – 4 "What is a game?". Under the section "Making the notes better", I have stricken through the text on improving the notes, because Andrea no longer maintains a web-page for this course. Any errors should be brought to *my* attention, `jonathan.l.shapiro@manchester.ac.uk`. View this as a textbook, not necessarily written for this course.

It is also worth noting that these notes were lasted updated in 2013. The field and Andrea's knowledge of the field has developed. This set of notes might not be what she would produce today. Nonetheless, this set of notes develops the foundations of Game Theory in a way which is well-informed and continues to be valid. I cannot find a text which combines game theory with practical game playing in such an excellent manner. It was written before the explosion of machine learning in game-playing intelligence, so a separate set of notes will support that topic.

Jonathan Shapiro
September 26, 2021

This page intentionally left blank.

What is a game?

A technical definition of a game appears in the next chapter, here we just want to give an intuition. Every time people (or, more generally, agents) interact with each other and jointly influence an outcome we can think of it as a game. There are lots of examples and everybody will have encountered at least some of these:

- Somebody has an old mp3-player that he know longer wants. He's offering it to his friend and will give it to the person who offers him the most in return. How should his friends decide how much to bid? What rules should he set so that he gets the largest amount?
- In a shared student flat the kitchen and the bathroom need to be cleaned regularly to stay in a decent condition—else the landlord is going to make trouble (and most people don't enjoy living in a filthy place). Does game theory tell us something about what the likely behaviour of the inhabitants is going to be, and does it help explaining it?
- On a network there is a limited capacity for transporting data. How should the individual machines on it be configured to ensure that
  - the capacity is shared fairly and
  - the existing capacity is fully exploited whenever there is sufficient demand?

  Bear in mind that a machine sending information will have no information about what the other machines on the network are doing.
- We can think of the buying and the selling of shares on the stock market as a game (lots of entities interact, and the outcome is defined by money and shares changing hands). If we could predict how the market is going to behave in the future we could make a lot of money...

There are, of course, also the better known (and expected) examples of playing a board or card game.

What all these 'games' have in common is that they can be defined more formally in such a way that they have the following properties:

- A number of agents interact with each other. These are known as the *players*.
- At the end of their interactions there is an outcome that can be described numerically—each player receives a *pay-off* which may be negative.
- There are clearly defined rules as to what decisions each agent can make at which point in time and once all such decisions have been made the outcome is uniquely determined.

These are fairly broad requirements, which are met by many situations in the real world. That is the reason games have been so successful at modelling a great variety of issues, and they are used in areas such as computer science, sociology, economics, biology, and mathematics.

In some of these subjects it is quite difficult to come up with a mathematical model that allows a worthwhile analysis of some situation, and in many cases the theory of games supplies by far the most productive one. Rather than trying to describe something through a set of equations all that is required here is the codification of the rules, and putting a number to the potential outcomes that somehow measures its 'benefit' (or detriment) to the participants. Game theory allows for an analysis of the resulting model. In some cases, where many similar transactions occur, it can be quite easy to describe the rules for the interaction of two individuals, and the complexity arises from the fact that many such interactions occur over a period of time. We see in the chapters on modelling that this can give rise to systems which are very expressive, but which are still only partly understood.

Games have also been successfully studied as models of conflict, for example in biology (animals or plants competing for resources or mating partners) as well as in sociology (people competing for resources, or those with conflicting aims). In particular in the early days of the theory of games a lot of work was funded by the military.

Two basic assumptions are being made here which are not completely unproblematic:

- It is assumed that the numeric value associated with a particular outcome (that is, the pay-off for each player) adequately reflects the worth of that outcome to all the players.
- Each player is concerned with maximizing his own outcome (this is sometimes known as 'rational behaviour') without regard of the outcome for the other players.

We do not spend much time in questioning these assumptions, but you should be aware of them. Game theory cannot say anything about situations where these don't apply. In the real world people do not always behave rationally.

To give a practical example, assume you are given a coin and, when observing it being thrown, you notice that it shows heads about 75% of the time, and tails the remaining 25%. When asked to bet on such a coin, a player's chances are maximized by betting on heads *every single time*. It turns out, however, that people typically bet on heads 75% of the time only!

One has to make some kind of assumption on how people will behave in a given situation in order to model it. In the absence of the above rationality assumption it might be possible instead to describe 'typical' behaviour and use that (this is often done in economy) but we do not pursue that idea here.

We make one further assumption for the sake of this course unit:

- Players are not allowed to pool their pay-off and cooperate in affecting the outcome.

Game theory can model this kind of situation, but the resulting mathematics is considerably more complex and we deem it beyond the scope of this course. In technical parlance we only look at *non-cooperative* games here.

What has to be stressed is that the only available language to analyse games is that of *mathematics*. Consequently there is a lot of mathematical notation in the notes and there are various theorems, some of them with proofs. While I do not expect you to become so proficient in this language that you can easily express your thoughts in it it is vital that you understand the concepts and results given here. To help with this informal descriptions accompany each mathematical definition, and the various results are also explained informally, and you will have to be able to argue at this level. You should be aware that even informal statements are fairly precise in their meaning, and that you have pay great attention to what you are saying to ensure that you get it right. There are some subtleties involved and if you don't appreciate these you may lose marks in an exam. This level of understanding cannot be gained in a short period of time and therefore it is vital that you engage with the material while it is being taught.

In this part of the course we aim to give answers to the following questions.

- What is a game?
- How can we describe a game plan for a player?
- What does it mean to play a game well?
- How do we find good game plans?
- What are some of the situations that can be modelled using games?

Help making the notes better

~~I would appreciate the readers' help in order to eliminate any remaining errors and to improve the notes. If you spot something that seems wrong, or doubtful, or if you can think of a better way of explaining something then please let me know by sending email to A.Schalk@cs.man.ac.uk. I will keep a list of known errors (and their corrections) available on the above webpage. Please note that while these notes are based on those for the old course unit COMP30192 they have been substantially rewritten so I fear there are a number of typos, and possibly worse errors.~~

I would like to thank David MacIntosh, Isaac Wilcox, Robert Isenberg, Roy Schestowitz, Andrew Burgess, Xue Qing Huang, Cassie Wong, Florian Stürmer, Mark Daniel, Steve Hughes, Kevin Patel, Mousa Shaikh-Soltan, and Ian Worthington for helping me improve the course material.

Literature

I don't know of any text book that presents all the material contained in these notes. Within the text I give references regarding the sources for each chapter to allow you to explore the area further by seeking those out.

Contents

Chapter 1

Games and strategies

1.1 So what's a game?

In every-day language, 'game' is quite a common word which seems to apply to a variety of activities (a game of chess, badminton, solitaire, poker, quake), and if we consider the act of 'playing' as something that applies to a game, then we get an even more varied range (playing the guitar, the lottery, the stock market). In this course unit we only consider some of these games.

The game tree

In the introduction to these notes three requirements are given, which we repeat here.

- A number of agents interact with each other. These are known as the *players*.
- At the end of their interactions there is an outcome that can be described numerically—each player receives a *pay-off* which may be negative. We assume that the pay-offs are given as real numbers (in many cases these will be natural or at least rational numbers).
- There are clearly defined rules as to what actions each agent can take at which point in time and once all such decisions have been made the outcome is uniquely determined.

While this may seem like a good specification it is not precise enough for our purposes here. To study games we have to give a precise definition of what we are talking about. How can we do that in a way that encapsulates the above requirements? The key to the general idea is given in the third requirement: We know at any time which player can make a decision, and what their options are at this stage. We can therefore think of these *decision points* as governing the game. Figure 1.1 gives a graphical presentation of this idea.

Example 1.1. Noughts and crosses. Part of a game tree for noughts and crosses (also known as tic-tac-toe) is given in Figure 1.1.

A bit more thought shows that we may think of a game satisfying the three requirements above as a tree: The nodes of the tree correspond to the decision points in the game, and the connections between these nodes indicate that there is a choice that takes the players to the next decision point.

The root of the tree is the starting point of the game, when nothing has happened yet.

The connections represent what we consider the *moves* of the game, these are understood to describe *actions* taken by the player. In the tree on page 7 the actions are given implicitly by looking at the images representing the nodes, and often this is sufficient. These connections are the *edges* of the game tree.

Trees are not allowed to contain circles, and so we can always draw a game tree with the root on top, and then every node naturally can be drawn on a level corresponding to the number of moves it takes to get there. The nodes that have no moves out of them (we call

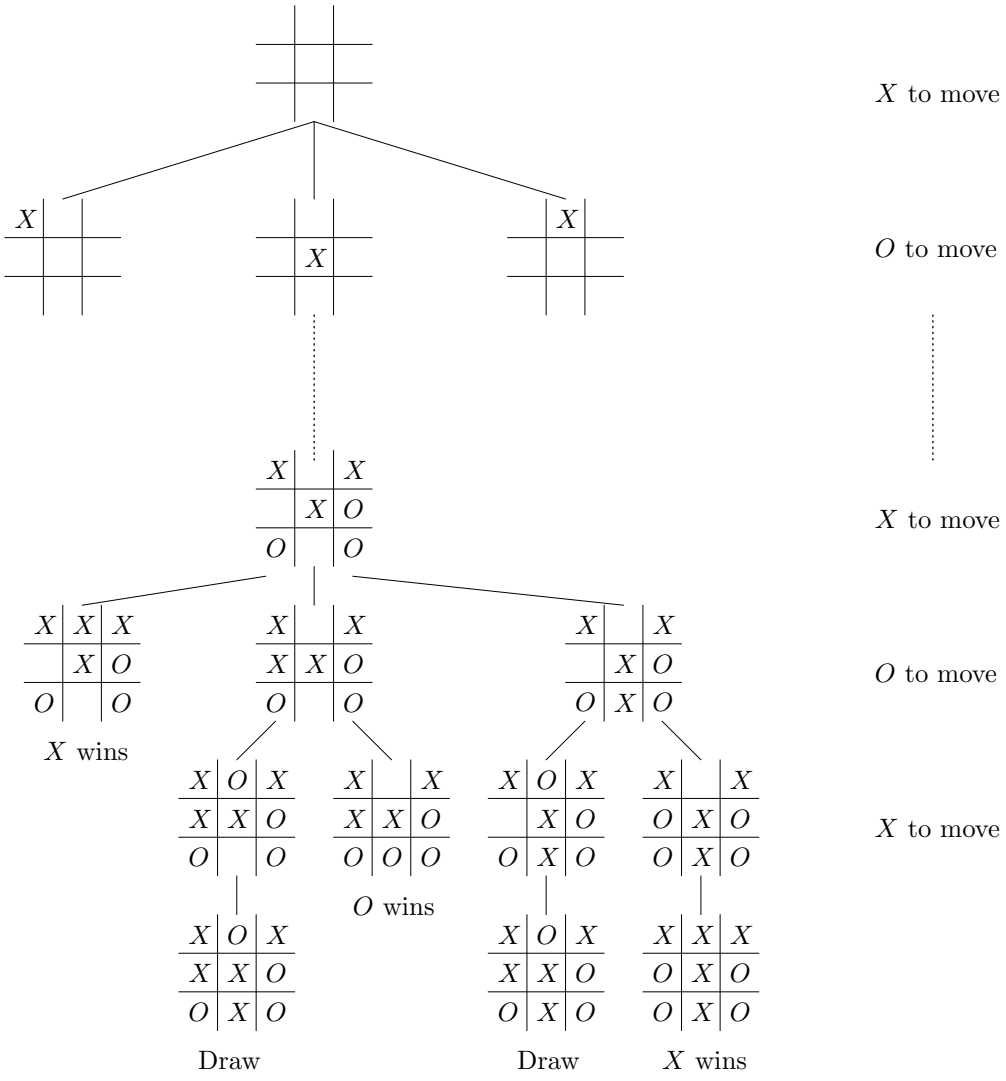


Figure 1.1: Part of a game tree for noughts and crosses

these the *leaves of the tree*) correspond to the various outcomes of the game. Strictly speaking they are not decision points of the game since no further decisions are being made but this poses no problem. These nodes are also known as the *final* nodes in the tree.

To fully describe a game we have to specify who the players are, for each decision point who owns the decision it represents (the player 'whose turn it is'), and for each leaf we have to give the outcome it represents. In Figure 1.1 we have named the players as X and O, for each decision point we have given the owner to the right and for each leaf we have specified who wins, or if there is a draw.

However, to satisfy the second requirement above we need to turn this into a number for each player, their *pay-off*. It is customary, but not the only possible choice, in a game like noughts and crosses to assign 1 to the winner, -1 to the loser, and 0 to each in the case of a draw. Usually the pay-off is written as a tuple where the first component gives the pay-off for Player 1 and the second component that for Player 2, and so on.

Often the players are simply assigned numbers, and in all our formal decisions that is how we refer to them. If there are only two players, 1 and 2, we use the convention that 'he' refers to Player 1 while 'she' always means Player 2.

Note that every time the game is played the players collectively follow one path in the game tree that leads from the root to a leaf. At each decision point the player whose turn it is chooses one of the available options until no further options are available. We call such a path a *play of the game*. Also note that every node uniquely determines the path that leads

to it from the root. In other words, every node *encodes the complete history of the game so far*. This has advantages and disadvantages, and we comment on these later in this chapter. Further note that every node defines a *subtree* of the original tree, namely the one whose root is that node. This gives rise to the notion of a *subgame* of the original game.

Here is another simple example.

Example 1.2. Chomp. Consider the following game. Two players have a bar of chocolate with $m \times n$ squares. The square in the top left corner is known to be poisonous. The players play in turn, where the rules are as follows: A player chooses one of the (remaining) squares of chocolate. He then eats this together with all the pieces which are below and/or to the right of the chosen one. In other words, he removes the entire rectangle of chocolate whose top left corner is given by the piece he chose. (Obviously) the player who has to eat the poisonous piece loses.

Figure 1.2 shows a game tree for (2×2)-chomp.

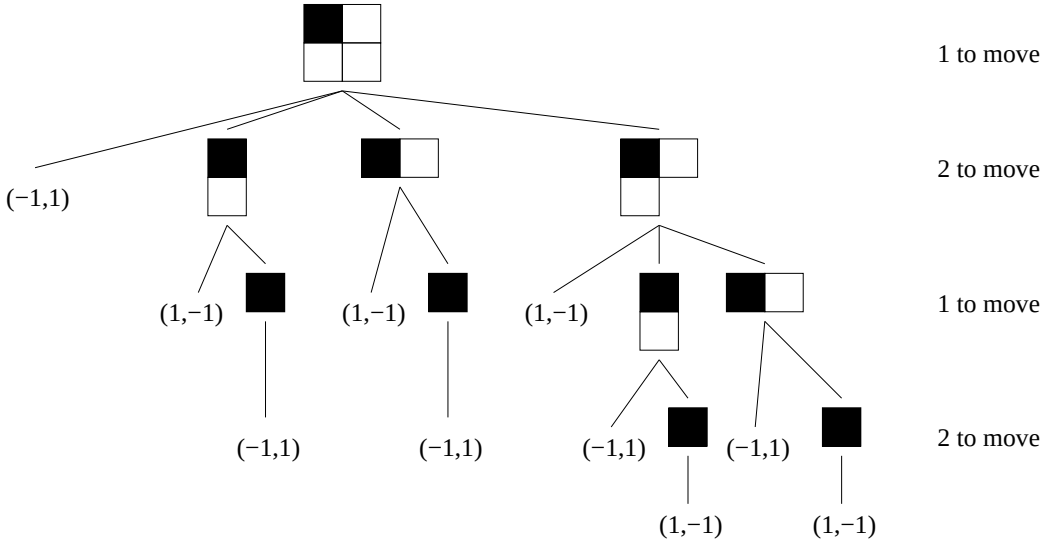


Figure 1.2: A game tree for (2×2)-chomp

This is a good time to practise drawing your own game trees.

B Exercise 1. (a) **Nim.** This is a game between two players who have a (finite) number of piles of matches in front of them. A valid move consists of choosing a pile and removing as many matches from that as the player chooses as long as it is at least one. The player who has to take the last match loses. (There is also a version where the player who takes the last match wins.) Draw a game tree for nim with two piles of two matches each. This is known as $(2, 2)$-nim. (If we had one pile of one match, two piles of two matches and one pile of three matches, it would be $(1, 2, 2, 3)$-nim.)

(b) Draw a game tree for 2×3-chomp.

Question 1. (a) Could you (in principle, don't mind the size) draw a game tree for Backgammon, or Snakes-and-Ladders? If not, why not?

(b) Could you draw a game tree for paper-stone-scissors? If not, why not?

(c) Consider the following simple game between two players: Player 1 has a coin which he hides under his hand, having first decided whether it should show head or tail. Player 2 guesses which of these has been chosen. If she guesses correctly, Player 1 pays her 1 quid, otherwise she has to pay the same amount to him. Could you draw a game tree for this game? If not why not?

There are some features a game might have which we cannot express in the game tree as we have thought of it so far. Hence the notion has to be expanded to cover such games.

- **Chance.** There might be situations when moves depend on chance, for example the throwing of a die, or the drawing of a card. In that case, the control over which move will be made does not entirely rest with the player whose turn it is at the time.
- **Simultaneous moves.** We take care of those through the same mechanism that we use to solve the following problem.
- **Imperfect information.** The players may not know where exactly in the game tree they are (although they have to be able to tell which moves are valid at any given time!). This often occurs in card games (which also typically contain elements of chance), where one player does not know what cards the other players hold, or when the game allows for 'hidden' moves whose consequences are not immediately clear.

Handling chance

So how do we go about adding chance elements to our game? The accepted method for doing so is to consider somebody called *Nature* who controls all the moves that involve an element of chance. (But Nature is not assigned a pay-off at the end, and when it comes to looking at players maximizing their pay-offs we exclude Nature.) In the game tree, all we do is to add nodes

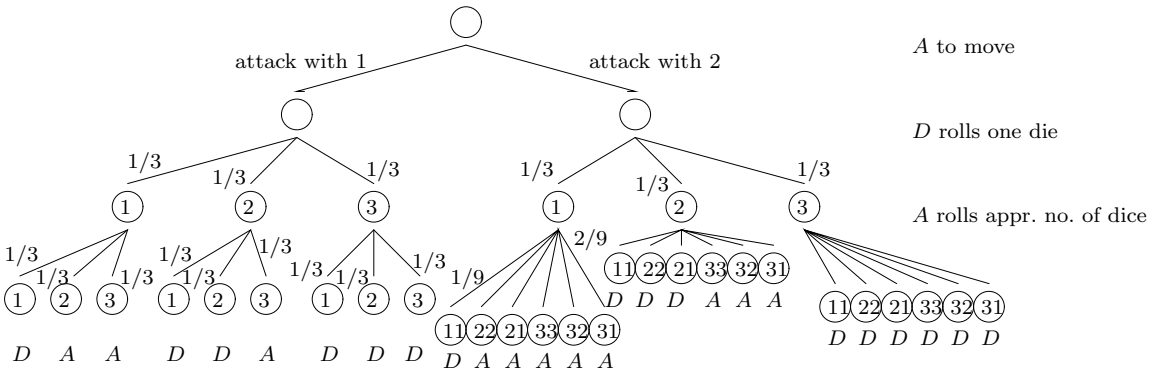
- where it is nobody's turn and
- where the branches from that node are labelled with the *probability* of the corresponding move occurring.

This does not just allow for the incorporation of chance devices, such as the throwing of coins and the rolling of dice, but also for situations with an otherwise uncertain outcome. In battle simulations, for example, it is often assumed that in certain situations (for example, defender *versus* aggressor), we have some idea of what is going to happen based on statistics (for example, in seven out of ten cases, defender wins). This is a somewhat crude way of modelling such things since it does not take into account the particular circumstances of a specific encounter (for example the personality and experience of those involved, the influence of geographical features, the quality of the defender's position (bunkers, suitable terrain, supplies, or the like)), but it still allows us to make a reasonable prediction regarding the outcome. A somewhat crude model is often better than none at all.

Example 1.3. Risk. In the board game Risk players have 'armies' which can defend or conquer territory on a map (which forms the board). Assume a defender has one army left in some country. An attacker can choose (by placing his armies) how many he or she might want to attack with. We limit the choices here to attacking with one or two armies. Both players then roll as many dice as they have armies in the bout (here, one or two). In the case where two dice are rolled against one, only the higher of the results of the throw of the two dice counts. If the defender's throw is at least as high as the attacker's then defender wins. In other words, for attacker to win his highest throw has to be higher than defender's. To keep the size of the game tree reasonable, we assume that instead of using ordinary dice we use ones which produce the numbers 1, 2 and 3 only, with equal probability.

The outcome of such a bout is shown in Figure 1.3. To keep the clutter down we only put the winner for each outcome. We say that the defender D wins if he successfully defends his territory, and that the attacker A wins if he invades the territory. The winner is marked for each final node of Figure 1.3.

If you can't see where the probabilities in Figure 1.3 come from, or if you feel a bit rusty regarding probabilities, then do the following exercise.



Probabilities for throwing two dice: 1/9 for each branch where the two numbers agree, 2/9 where they differ.

Figure 1.3: An excerpt from a game of Risk

B Exercise 2. (a) Draw a game tree where a player throws two dice one after the other. Assume that these dice show 1, 2, or 3 with equal probability. Use it to calculate the probability for each possible outcome and use them to explain the subtree of Figure 1.3 where A rolls two dice. You may want to read on a bit if you are unsure how to deal with probabilities.

(b) Draw a tree for the game where two players get one card each out of a deck of three (consisting, say, of J, Q and K). Count the number of different deals, and then the number where Player 1 has the higher card. If Player 2 wins in the cases where she has the Q, or where she has the K and Player 1 has the J, what is the probability that she wins the game?

So how much 'better' is it for the attacker to use two armies? For this we want to calculate the probability that A wins in either case. How do we do that?

From Figure 1.3 we can see that if he attacks with one army, there are 3 final nodes (out of 9) (corresponding to one play each) where A wins. We have to *add up* those probabilities.

To calculate the probabilities for a final node, we have to *multiply* all probabilities mentioned along the path from the root that leads to it.

So the probability that D throws 1 while A throws 2 is $1/3 \times 1/3 = 1/9$. Similarly for the other two nodes where A wins (namely where A throws 3 while D throws 1 or 2), so the probability that A wins if he attacks with one army is

$$1/9 + 1/9 + 1/9 = 3/9 = 1/3 \approx 0.33.$$

Secondly we consider the case where A attacks with two armies. Now we have eight cases (out of 18) where A wins. The probabilities we have to add up are (from left to right) as follows.

$$1/27 + 2/27 + 1/27 + 2/27 + 2/27 + 1/27 + 2/27 + 2/27 = 13/27 \approx 0.48.$$

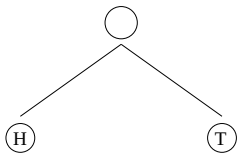
The principles regarding calculating probabilities for complex outcomes are important, so we repeat them in more general terms. To calculate the probability of a particular result do the following:

- Determine all the leaves in the game tree that are part of the desired result.
- For each of these leaves calculate its probability by multiplying all the probabilities along the unique path that connects it with the root.
- Add up all the probabilities from the previous step.

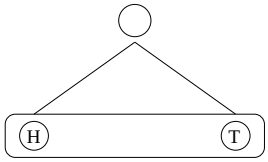
Handling imperfect information

Card games in particular often involve chance as well as imperfect information, because no player knows the cards the other players hold. This information has to be built into the game tree.

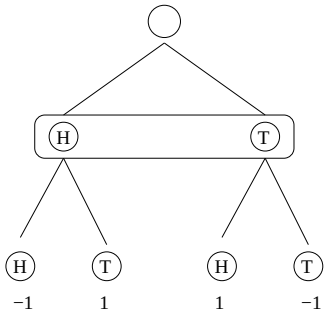
Example 1.4. We go back to the game described in Question 1 (c). Player 1 makes a move that is hidden. The game tree so far looks like this.



However, Player 2 does not know which of the two nodes on the bottom level they are in. She has to decide which move to make *despite of this*. This is important information regarding the rules of the game and we have to include it in the game tree. We do this by including the nodes she may be confused about in the same 'bubble'. This indicates that Player 2 has to choose between the actions H and T without knowing which of those two nodes she is in.



The full game tree then looks as follows.



We refer to nodes of the tree in the same bubble as *belonging to the same information set*. The player whose turn it is at such a point has to make a decision regarding which action to take without knowing which of the nodes in that set the game is in at present. To draw a game tree all information sets have to be given. If both players lack knowledge about some of the nodes the tree can become quite complicated and then drawing all the required bubbles can be very messy. In such cases we instead connect all the nodes in the same information set with a dotted line instead. Note that the information set tells us *about the knowledge of the player whose turn it is*. Some people find it easier to draw information sets for all the players at all levels of the game tree. In that case it is best to use different colours to distinguish between the players.

Example 1.5. Simplified poker. There are two players, each of whom has to pay one pound to enter a game (the *ante*). They then are dealt a hand of one card each from a deck containing three cards, labelled J, Q and K. The players then have the choice between either betting one pound or passing. The game ends when

- either a player passes after the other has bet, in which case the better takes the money on the table (the *pot*),

- or there are two successive passes or bets, in which case the player with the higher card (K beats Q beats J) wins the pot.

The game tree for this game is given on page 13.

B Exercise 3. Alice and Bob play a different form of simplified poker. There are three cards, J, Q and K, which are ranked as in the above example. Each player puts an ante of one pound into the pot, and Alice is then dealt a card face down. She looks at it and announces 'high' or 'low'. To go 'high' costs her 2 pounds paid into the pot, to go 'low' just 1. Next Bob is dealt one of the remaining cards face down. He looks at it and then has the option to 'fold' or 'see'. If he folds the pot goes to Alice. If he wants to see he first has to match Alice's bet. If Alice bet 'high' the pot goes to the holder of the higher, if she bet 'low' it goes to the holder of the lower card. Draw the game tree for this game, including the information sets.

The concept of some information being 'hidden' can also be used to cope with games where two players move simultaneously.

Example 1.6. Paper-stone-scissors. This is a two player game. At a command, both players hold out their right hand in one of three ways, indicating whether they have chosen paper, stone, or scissors. Paper beats stone which beats scissors which beats paper. To draw a game tree for this game we let Player 1 move first, but demand that Player 2 be unable to tell which choice Player 1 has made. Figure 1.4 gives the corresponding game tree, where P is for choice 'paper', R is for choice 'stone' (think 'rock') and S is for choice 'scissors'. The result is marked as 1 if Player 1 wins, 2 if Player 2 is successful and D for a draw.

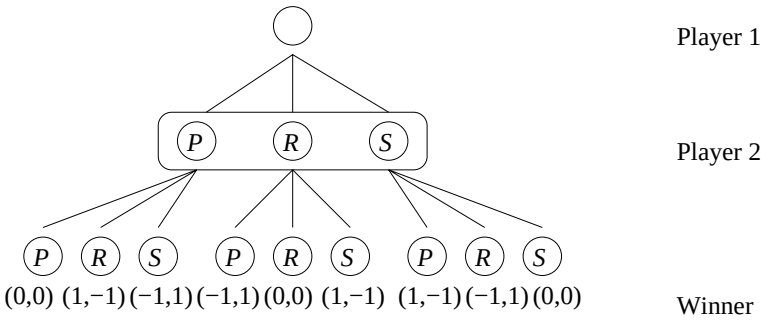
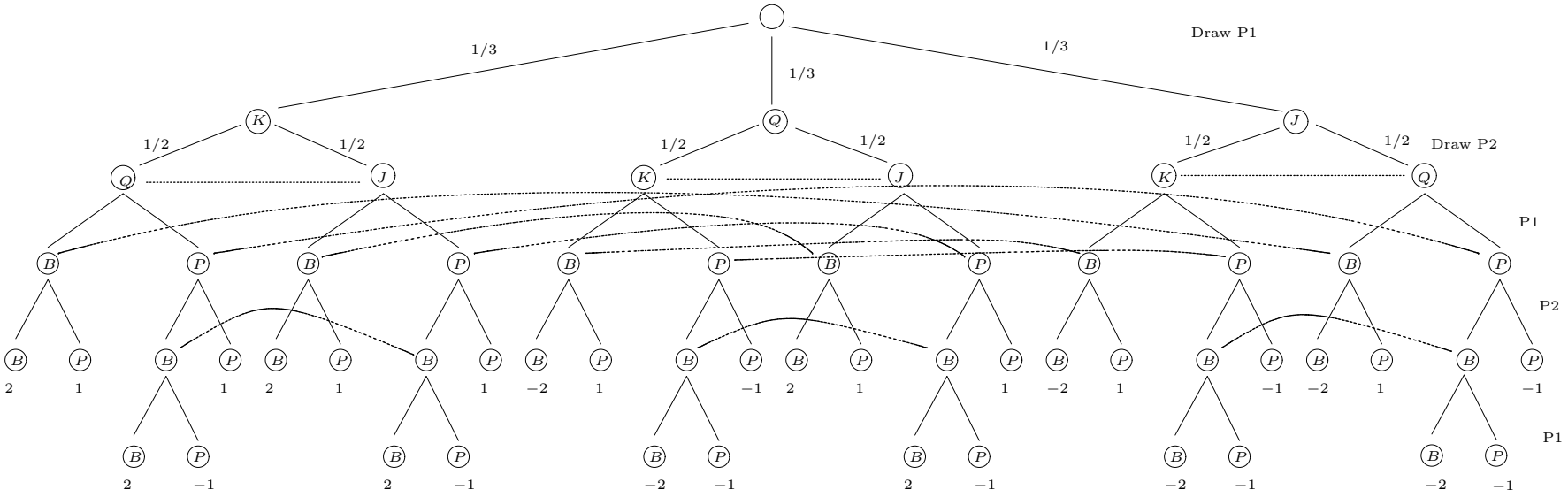


Figure 1.4: Paper-stone-scissors

Note that for nodes to be members of the same information set it *must be* the case that the actions that can be taken from each of those nodes are precisely the same. This is necessary so that the player whose turn it is at that point cannot find out whether or not a node really belongs to the information set by trying to play a move which is possible only for some of the nodes in the set.


Draw P1
1/3
1/3
1/3
K
Q
J
1/2
1/2
1/2
1/2
1/2
1/2
Draw P2
Q
J
K
J
K
Q
P1
B
P
P2
P1

A game formally

We are finally ready to formally define a game.

Definition 1. *A* **game** *is given by*

- *a finite set of* players*,*
- *a tree,*
- *for each node of the tree, a player who owns that node (the player whose turn it is at that node—this might be the extra player Nature),*
- *for each edge of the tree an* action *that labels it (actions taken by Nature must include the probability with which they occur),*
- *for each player the corresponding* information sets*; these are sets of decision nodes for that player such that for every two node in some information set for Player i it is the case that*
  - *the actions taken by Player i to reach the nodes are identical and*
  - *the actions labelling the edges out of the nodes are identical.*
- *for each leaf node and each player a pay-off.*

Games defined like this are often called *games in extensive form* in the literature. That implies that there is another form of describing games. The *normal form of a game* is defined later in this chapter. Much of the original development of the theory of games has been restricted to games in normal form, but from the point of view of implementing games (and from the point of view of learning in games) the extensive form is by far the more important one.

There are a few groups of games that are important enough to have their own names, and we introduce those here.

Definition 2. *(a) A game is a* **2-player game** *if there are two players (not counting Nature).*

(b) A game is **of perfect information** *if all information sets in the game tree have size one—in other words, all players know at all times which node they are in. Games that do not satisfy this condition are* **of imperfect information***.*

(c) A game is **zero-sum** *if at every leaf of the game tree the sum of the pay-off for all the players is* 0*.*

(d) A game is **without chance** *if no node in the game tree is controlled by Nature, otherwise it is a* **game with chance***.*

Below we list all the properties from above that the given game has.

Example 1.7. (a) Poker is a zero-sum[1] game of imperfect information with chance.

(b) Chess, go, draughts, reversi (also known as othello) are 2-person zero-sum games of perfect information without chance.

(c) Backgammon is a 2-person zero-sum game of perfect information with chance.

(d) Rummy (with a lot of other card games) is a game of imperfect information with chance.

(e) Risk is a game of perfect information with chance, as is snakes-and-ladders.

[1] Unless the players have to pay to join, or a casino takes a cut.

(f) Tossing a coin can be described as a two-person zero-sum game of perfect information with chance.

(g) Paper-stone-scissors is a 2-person zero-sum game without chance.

Question 2. Which other games do you know? Can you categorize them?

When talking about 2-person zero-sum games it is customary to only give pay-offs for Player 1 since the ones for Player 2 can be derived from those. We do so in our game trees from now on.

It turns out that game theoretic methods are most useful for 2-person zero-sum games, and those are also the games for which computing good game plans is easiest. We see how and why in subsequent chapters.

Consequences of this definition

It is worth pointing out that a game tree distinguishes between positions that might be considered the same: There are several ways of getting to the board position pictured in the third line of Figure 1.1. Player X might start with a move into the centre, or a corner, and similarly for Player O. Hence this position comes up several times in the game tree. This may seem inefficient since it seems to blow up the game tree unnecessarily, but it is the accepted way of analysing a game.

If we wanted to identify those nodes that correspond to 'the same position', say for a board game, we would get a 'game graph' rather than a game tree. This would make it more difficult to keep track of other things. We might, for example want to represent a chess position by the current position of all the pieces on the board. Then two positions which 'look' the same to an observer would be the same. However, even in chess, that information is not sufficient. For example, we would still have to keep track of whose turn it is, and we would have to know which of the two sides is still allowed to castle. Hence at least in chess some information (beyond a picture of the board) is required to determine the valid moves in a given position.

More importantly, by using this game graph we would make the implicit assumption that any player would always play in the same way from any of the nodes corresponding to 'the same' position. This is a restrictive assumption since the play that got us to that position might give us some idea of how our opponents operate, and may lead us to wanting to make different choices.

With the game tree, every node of the tree comes with the entire history of moves that led to it. As a consequence, when following moves from the start node (root), possibilities may divide but they can never reunite. In that sense, the game tree makes the *maximal number of distinctions* between positions. This allows us to consider a larger number of *strategies* for each player. Tree traversal is easily implemented, which helps.

However, all this means that our notion of strategy assumes that a player can remember the entire play so far, and one might argue that this is not always warranted. We see in subsequent chapters how one can cut down the number of strategies, in particular in games which are repeated a number of time.

There is another issue with looking at games in this way: How do we find a pay-off value for games that do not naturally come with numbers describing the outcomes? For example in the situation regarding the kitchen in a shared student flat, how do we assign numbers to the effect of having to live with a dirty one? How does that compare with the value of not having to do any cleaning? This is a very complex subject, but for the purposes of this course we assume that the choice has been made already. If one wants to use game theory for example to manage risk it is extremely important, however, that one gets these values right (for example in a 'game' where a player risks his life the pay-off where they lose it must be an enormously high negative number when compared to the other pay-offs, or the analysis will not give a result that matches the true evaluation of one's decisions).

But even if there are numbers readily at hand, they may not describe the value of something to the individual: Buying a lottery ticket is something that nobody should do given that the expected pay-off is negative. Yet people do it—presumably because the thought that they have a chance of winning is worth more to them than the risk of losing their stake.

1.2 Strategies

If we want to talk about what it takes to 'play a game well' we need a formal notion of a *game plan*. We can then compare game plans to each other and come up with criteria for when one is more desirable than another, and more generally which game plan one might want to go for.

Note that a player's actions in a game are uniquely determined if for each node of the game tree they have made a decision about which action to take if play reaches that node.

Question 3. What can you say about the number of game plans for a player if you know how many decision points he has?

Definition 3. *A* **fully specified strategy for Player** *i in a game is given by choosing*

for each of i's nodes an action allowed by the game tree such that

for all nodes in the same information set the same action is chosen.

Note that when people talk about strategies in every-day life they usually have something much vaguer in mind—maybe just a fairly general plan. Whenever we talk about a strategy in this course unit we have a fully specified game plan in mind.

Figure 1.5 gives an example for such a fully specified strategy in (2×2)-chomp. The dashed lines indicate the full game tree, while the ones drawn without interruption show the choice specified by this strategy for each of Player 1's nodes.

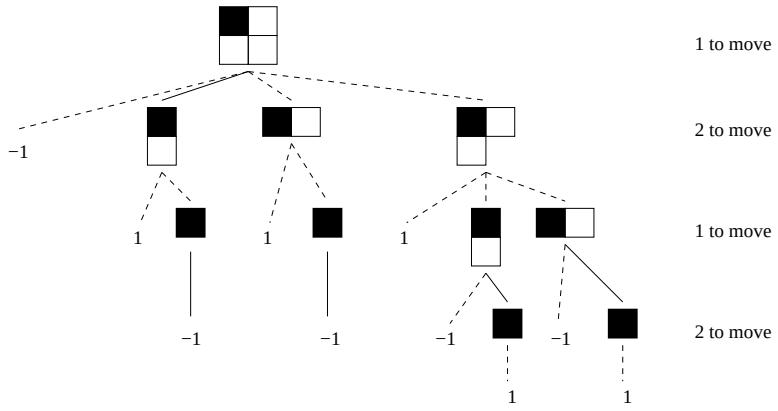


Figure 1.5: A fully specified strategy for (2×2)-chomp

Question 4. How many fully specified strategies for Player 1 are there in (2×2)-chomp? How about Player 2?

We get the number of fully specified strategies for a player by doing the following:

- for each decision point of the player, count the number of choices and then
- multiply all the numbers gained in the previous step.

B Exercise 4. (a) Describe all fully specified strategies for both players in (2×2)-chomp.

(b) Do the same for $(2, 2)$-nim.

Question 5. Can you tell how many possible outcomes there are if Player 1 follows this strategy?

If we look more closely at fully specified strategies we realize that they actually *over-specify* a player's action. Consider the following example and compare it with Figure 1.5

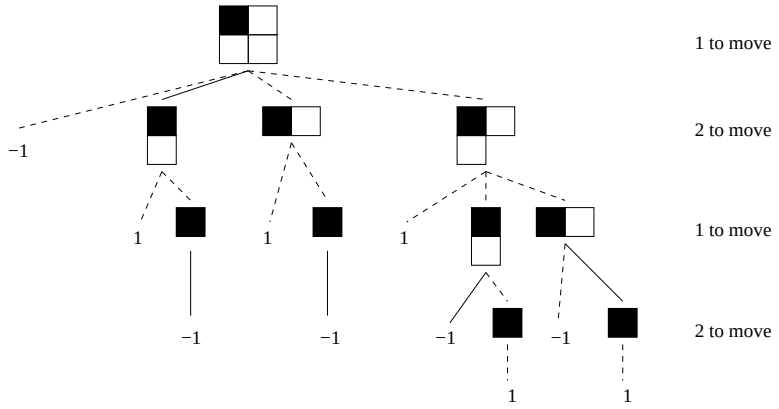


Figure 1.6: Another fully specified strategy for (2×2)-chomp

When playing, in either one of these strategies Player 1's first choice takes the game down the second branch. Hence any choices that are specified when following the fourth branch from the root are never carried out.

Put differently, Player 1 behaves in precisely the same way, no matter which of these two game plans he follows. By describing all the fully specified strategies for a player we are describing more game plans than anybody can distinguish between! This is the reason why we define another notion of game plan below. However, we see in subsequent sections why the notion of a fully-specified strategy is still useful. While it does create more game plans than one might want it is also a less complicated one than the one we arrive at below.

Figure 1.7 gives all the game plans for Player 1 in (2×2)-chomp which do not overspecify his choices. The game tree is now given in a stylized form only, but it should be clear what is described. The solid lines give the decisions of Player 1 in the obvious way. We have also drawn in a solid line those moves for Player 2 that might still be possible if Player 1 plays in accordance with the given game plan.

Note that there are seven strategies for Player 1 in this game while there are sixteen fully specified ones.

Closer inspection of the example is that in this way we are describing a game plan that is naturally a subtree of the game tree with certain properties: Whenever a position is reachable based on the choices made 'so far'

- if it is the chosen player's turn, precisely one of the available moves is chosen;
- if it is not the chosen player's turn, all available moves are chosen.

If we follow these rules we can generate a strategy, making sure that we only make a decision when we have to (that is, we do not worry about unreachable positions). We can now define formally what a strategy is. Note that we demand that a choice be made for every position where it is the chosen player's turn; we do not allow bringing the game to a halt by refusing to continue. If we want to give the player the option of resigning we should make that an explicit move in the game tree.

The following is a recursive definition.

Definition 4. *A (pure)* **strategy for Player** *i is given by a subtree of the game tree with the following properties:*

- *The root of the game tree belongs to the strategy;*

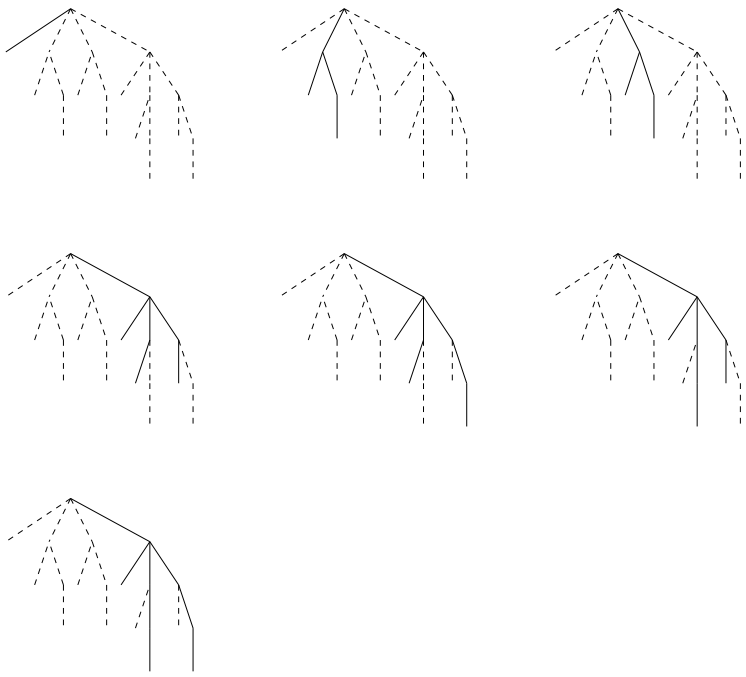


Figure 1.7: All the strategies in (2×2)-chomp for Player 1

- *whenever it is Player i's turn at a node that belongs to the subtree,*
  - *exactly one of the available moves belongs to the subtree and*
  - *for all nodes in the same information set for Player i the same action is chosen;*
- *whenever it is not Player i's turn at a node that belongs to the subtree, all of the available moves belong to the subtree.*

Note that as a consequence of using trees rather than graphs to give the rules of the game, we are allowing *more* strategies: We are allowed to take into account all the moves made so far, not merely the position reached on the board, say. This more generous notion can be justified by pointing out that the history that led to the current position might have given us an insight into the other players' ability in playing the game in question.

B Exercise 5. (a) How many strategies are there for Player 2 in 2×2-chomp?

(b) How many strategies for simplified poker (see Exercise 3) are there for both players?

So what happens if we have a game which includes elements of chance? Actually, the definition we have just given still works. We merely have to treat Nature as just another player.

Let's have a closer look at a game of incomplete information, Example 1.6, paper-stone-scissors. What the definition says is that there are only three strategies for Player 2—he (or she) is not allowed to try to take into account something he does not know, namely the first move made by Player 1. All valid strategies for Player 2 are given in Figure 1.8.

B Exercise 6. (a) Give all the strategies for $(2, 2)$-nim (for both players). For this it is useful if your game tree takes symmetry into account to make it smaller!

(b) Give three different strategies for simplified poker (confer Exercise 3).

Note that given a fully specified strategy we can produce a strategy for the same player, say i, that shows precisely the same behaviour by doing the following: Follow the recursive definition. Whenever we reach a node where it is player i's turn, follow the given game plan.

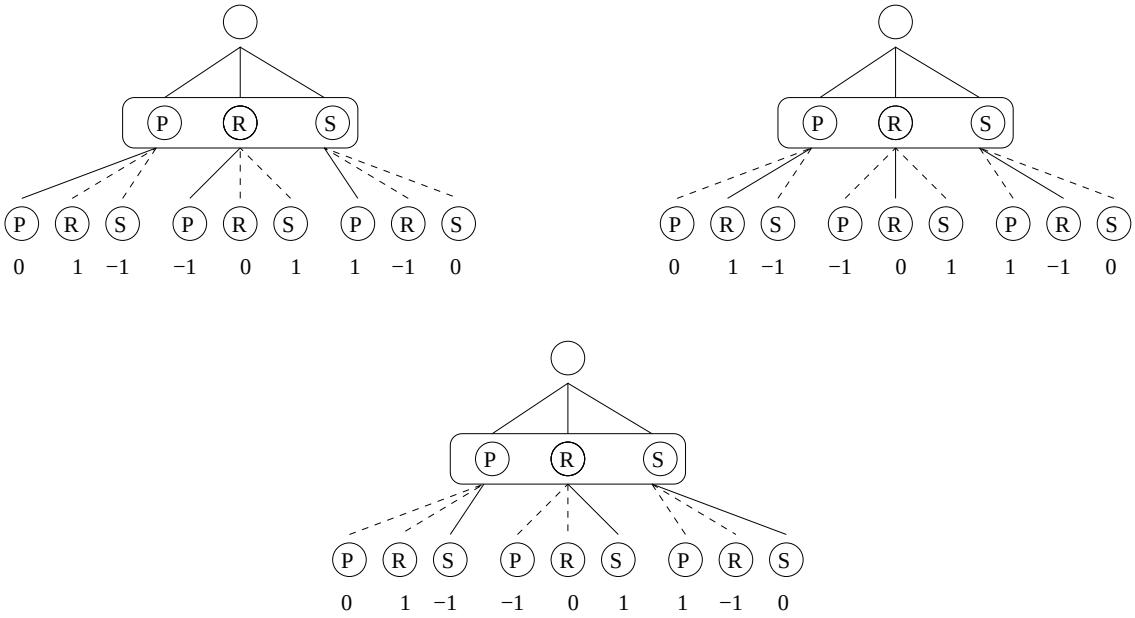


Figure 1.8: All the strategies for Player 2 in paper-stone-scissors

What this does is to remove the decision made for all nodes that cannot be reached in any play in which Player i follows the given fully specified strategy. Hence there is a surjective function from the set of all fully specified strategies for Player i to that of all strategies for the same player.

Generating all strategies

Generating all the strategies for some player, say i, can be performed recursively as follows. When searching for the strategies for Player i in a game tree t, we assume that we already know the strategies of the sub-games $t_1, \ldots, t_n$, which follow after the first move has been played (see Figure 1.9). At the same time we count the **number of strategies $N_i(t)$ for Player i**. However, this only works for games of *perfection information*! (It has to be refined and becomes rather more complex for games of imperfect information, but the same idea still works.) Note that counting strategies is rather more difficult than counting fully specified ones.

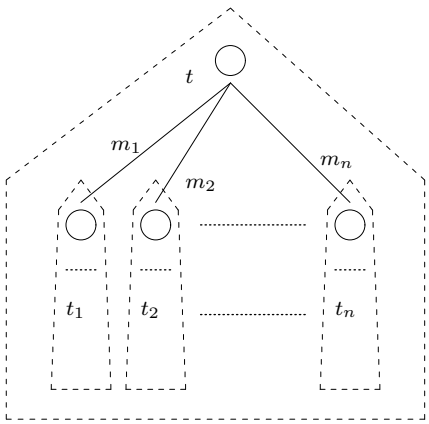


Figure 1.9: Counting strategies and immediate sub-games

- A game with a game tree of height zero has one strategy for each player ('do nothing').

- To find all the strategies for Player i when the first move is Player i's: Player i has to choose one of the available first moves, m_1, m_2, $\ldots m_n$. Once that move, say m_j, has been played the game proceeds as per the game tree t_j. Hence every first move m_j, combined with some possible strategy for the corresponding sub-game t_j, gives a valid strategy for the game t. Therefore in this situation,

$$N_i(t) = N_i(t_1) + N_i(t_2) + \cdots + N_i(t_n).$$

- To find all the strategies for Player i when the first move is not Player i's, Player i needs a reply for all the possible first moves in the game (which are somebody else's choice). So a strategy for the game t for Player i consists of picking a strategy for this player in each of the games t_1, t_2, $\ldots$, t_n. All the combinations arising in this way are counted by

$$N_i(t) = N_i(t_1) \times N_i(t_2) \times \cdots \times N_i(t_n).$$

Think about how the above would have to be changed for games of *imperfect information*.

1.3 The normal form of a game

One way of thinking about playing a game is the following: Let each player choose one of their available strategies at the start. Let each of them play according to that strategy. Now playing becomes a purely mechanical act: All a player has to do is to look up what to do whenever it is his turn. Arguably, that makes playing a game a fairly boring activity, but we see below why this sometimes is a useful point of view. Indeed, it leads to another way of describing a game.

A simple example is Paper-Scissors-Stone. Each player has three strategies, which may be conveniently labelled by P, R and S

If we list the strategies for Player 1 in a column, and those for Player 2 in a row we can fill in the result of playing a strategy for the one against a strategy for the other in the form of a table.

The players get the following pay-offs:

Player 1

		2		
		P	R	S
	P	0	1	-1
1	R	-1	0	1
	S	1	-1	0

Player 2

		2		
		P	R	S
	P	0	-1	1
1	R	1	0	-1
	S	-1	1	0

In cases where there are only two players the pay-offs are often given in a combined table whose entries are pairs with the first component giving the pay-off for Player 1 and the second that for Player 2. In the above example that would look as follows.

		2		
		P	R	S
	P	$(0,0)$	$(1,-1)$	$(-1,1)$
1	R	$(-1,1)$	$(0,0)$	$(1,-1)$
	S	$(1,-1)$	$(-1,1)$	$(0,0)$

When we don't care about the name of the strategies we can give that table without labelling the rows and columns, and that is the standard presentation:

$$\begin{vmatrix} (0,0) & (1,-1) & (-1,1) \\ (-1,1) & (0,0) & (1,-1) \\ (1,-1) & (-1,1) & (0,0) \end{vmatrix}$$

Determining the outcome when playing strategies against each other is done as follows. If no elements of chance are involved then one simply follow the unique play (that is, path through the game tree) that the strategies under consideration (one for each player) determine collectively until a final position has been reached. Graphically one can think of it as overlaying all the subtrees defined by the various strategies—there is precisely one path where they all overlap. At that position the pay-off for each player can be read off and put into a table as above.

B Exercise 7. (a) Give the normal form for $(2, 2)$-nim.

(b) Do the same for the game from Question 1 (c) whose game tree is constructed in Example 1.4.

We can also calculate the normal form of a game by using the fully specified strategies by both players (of which there are typically more). The resulting table has *duplicate* rows and/or columns: Every fully specified strategy generates the same row/column as the strategy it corresponds to.

B Exercise 8. Calculate the normal forms based on fully specified strategies for both games from the previous exercise (cf. Exercise 4).

Hence if our purpose is to calculate the normal form of a game it is sufficient to generate the strategies; doing this with the fully specified ones makes the pay-off tables larger but it does not add any information.

If elements of chance are involved then the subtrees corresponding to the various strategies may have more than one path in common. What one has to do then is the following. To calculate the expected pay-off for Player i in that situation:

- For each final node that is reachable in this way
  - calculate the probability of reaching that node by multiplying all the probabilities that appear along the path to that node and
  - multiply it with the pay-off for Player i at that node.
  - Now add up all the numbers calculated in the previous step.

Example 1.8. Consider the following game between two players. Player 1 rolls a a three-faced die (compare Example 1.3). If he throws 1 he pays two units to Player 2. If he throws 2 or 3, Player 2 has a choice. She can either choose to pay one unit to Player 1 (she stops the game) or she can throw the die. If she repeats Player 1's throw, he has to pay her two units. Otherwise she pays him one unit. The game tree is given in Figure 1.10, with the pay-off being given for Player 1.

Player 1 has only one strategy (he never gets a choice) whereas Player 2 has four strategies (she can choose to throw the die or not, and is allowed to make that dependent on Player 1's throw). We can encode her strategies by saying what she does if Player 1 throws 2, and what she does if Player 1 throws 3, stop (S) or throw the die (T). So $S|T$ means that she stops if he throws 2, but throws if he throws 3. The pay-off table for Player 1 has the following shape:

	2			
	$S\|S$	$S\|T$	$T\|S$	$T\|T$
1				

What are the expected pay-offs for the outcome of these strategies? We first consider the case $S|S$. We calculate the expected pay-off as follows: For each of the possible outcomes of playing this strategy (combined with Player 1's only strategy), take the probability that it occurs and multiply that with the pay-off, then add all these up. Hence we get

$$(1/3 \times -2) + (1/3 \times 1) + (1/3 \times 1) = 0.$$

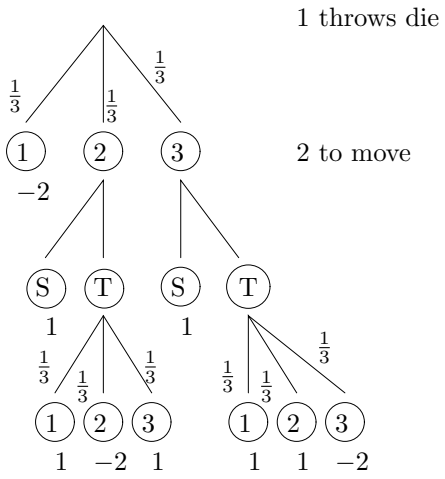


Figure 1.10: A game of dice

Now for the more interesting case of $S|T$:

$$(1/3 \times -2) + (1/3 \times 1) + (1/9 \times 1) + (1/9 \times 1) + (1/9 \times -2) = -3/9 = -1/3.$$

The case of $T|S$ is symmetric and therefore yields the same pay-off. Finally for the case $T|T$.

$$(1/3 \times -2) + (1/9 \times 1) + (1/9 \times 1) + (1/9 \times -2) + (1/9 \times 1) + (1/9 \times 1) + (1/9 \times -2) = -2/3.$$

The complete pay-off table for Player 1 looks like this:

	2			
	$S\|S$	$S\|T$	$T\|S$	$T\|T$
1	0	$-1/3$	$-1/3$	$-2/3$

Question 6. Which player would you rather be in this game?

B Exercise 9. (a) Take the game tree where one player throws two dice in succession (see Exercise 2). Assume that the recorded outcome this time is the sum of the two thrown dice. For all numbers from 2 to 6, calculate how likely they are to occur. Then calculate the expected value of the sum.

(b) Take the game from Example 1.8, but change the pay-off if Player 2 decides to throw a die. If Player 1 and Player 2's throws add up to an odd number then Player 1 pays Player 2 one unit, otherwise she pays him one unit. Produce the matrix version of this game.

So once each player has chosen a strategy, we can read off the result from these tables, or matrices, without bothering with going through the motion of playing the actual game. We have therefore given an *alternative description* of the original game—one which makes playing it rather boring. From the game-theoretic point of view, however it is totally irrelevant how exactly the result is reached, just what it is. If a game is given *via* its strategies and a pay-off table for each player we say that the game is in **normal form**. In the literature this is also sometimes referred to as the **matrix form** of the **strategic form** of the game. In the next section we discuss which form of a game is more suitable for which circumstances.

In the important case of a 2-person zero-sum game it is sufficient to give the pay-off table for Player 1 only: We can read off that for Player 2 by negating every entry in that table. (Compare the two tables for paper-stone-scissors.) Hence 2-person zero-sum games in normal form are described by one pay-off table, just as it is customary to only give the pay-off for Player 1 in the corresponding game tree.

If there are several players then referring to the pay-off tables can be tedious. If we have l players we assume that for each player, say i, there is a pay-off function p_i such that

$$p_i(s_1, s_2, \ldots, s_l)$$

gives the pay-off for Player i if

- Player 1 plays strategy s_1,
- Player 2 plays strategy s_2,

...

- Player l plays strategy s_l.

Formally speaking the normal form of a game is given by the following ingredients.

Definition 5. *A* **game in normal form** *is given by the following:*

- *A (finite) list of players, 1, ..., l;*
- *for each player a list of valid strategies for the player, numbered 1, ..., n_i for Player i;*
- *for each player i a pay-off function p_i which maps the space of consisting of all combination of strategies the players may play*

$$\prod_{1 \leq j \leq l} \{1, \ldots, n_j\}$$

to the real numbers. In other words

$$p_j \colon \prod_{1 \leq j \leq l} \{1, \ldots, n_j\} \longrightarrow \mathbb{R}.$$

If there are only two players we typically write the pay-off function in the form of a table as in the example of paper-stone-scissors above. If the game is 2-person zero-sum we usually only give a table for Player 1.

This formal definition looks scarier than it is. Here is a concrete example with three players.

Example 1.9. Consider the following three person game. Each player pays an ante of one. On a signal, all the players hold up one or two fingers. If the number of fingers held up is divisible by 3, Player 3 gets the pot. If the remainder when dividing is 1, Player 1 gets it, otherwise Player 2 is the lucky one.

Question 7. Do you think that this game is likely to be 'fair', in the sense of giving all the players an even chance to win? Which player would you prefer to be?

Each player has two strategies: Holding up one finger or holding up two fingers. We number them as 1 and 2 (in that order). Hence the space of all strategies is

$$\begin{aligned} \prod_{1 \leq j \leq 3} \{1, 2\} &= \{1, 2\} \times \{1, 2\} \times \{1, 2\} \\ &= \{(1,1,1), (1,1,2), (1,2,1), (1,2,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2)\}. \end{aligned}$$

It has $2 \times 2 \times 2 = 8$ elements. The three pay-off functions are given as follows (this takes into account that the players have all paid the ante, that is they either lose one unit or they get 3, one of which they put in themselves, making for a win of 2):

	p_1	p_2	p_3
$(1,1,1)$	-1	-1	2
$(1,1,2)$	2	-1	-1
$(1,2,1)$	2	-1	-1
$(1,2,2)$	-1	2	-1
$(2,1,1)$	2	-1	-1
$(2,1,2)$	-1	2	-1
$(2,2,1)$	-1	2	-1
$(2,2,2)$	-1	-1	2

Assume that Player 1 chooses to show 1 finger, that is his strategy 1, Player 2 and Player 3 show 2 fingers, that is their strategy 2. Then the pay-off for Player 2 is $p_2(1,2,2) = 2$. You will notice that there are three cases in which Player 1 wins, and also three cases in which Player 2 wins, but only two in which Player 3 is fortunate. Hence Player 3 seems to be at a disadvantage.

B Exercise 10. (a) Consider the following game for three players. Each player places a bet on the outcome (1 or 2) of a throw of a die without knowing what the others are betting. Then the die is thrown. If the number showing is odd we record the result as 1, otherwise as 2. A player gets a pay-off of ten points if he is the only one to bet on the correct result, if two of them do so they each get four points, and if all three are successful they get two points each. Describe the normal form of this game.

(b) Consider the following game for three players. Player 1 announces whether he chooses left (L) or right (R), then Player 2 does the same, and lastly Player 3. The pay-off for each player is calculated as follows: If all players make the same choice, they each get 1 point if that choice is L, and they each lose 1 point if that choice is R. If two choose R while one chooses L then the two players choosing R obtain 2 points each while the sole supporter of L loses 2 points, and if two choose L while only one chooses R then the person choosing R gets 3 points while the other two get nothing, but don't have to pay anything either. How many strategies are there for each player in the game?

1.4 Extensive or normal form?

Every game (of the kind we consider) can be given in either extensive or normal form, at least in principle. So which form should one study?

The answer is 'it depends', as is so often the case with interesting questions. Much of the mathematical work on games has been for games in normal form—for a mathematician the idea that the two are equivalent is sufficient reason to concentrate on the one more amenable to the available methods. However, there has recently been much interest in computational properties of games, and for various reasons that has led to extensive forms of games being studied in more detail.

Keeping it small

It is always worth trying to keep the description of a game as small as possible—that makes it easier to analyse. With careful consideration it is sometimes possible to leave out possibilities that are equivalent to others.

At first sight, the game tree in Example 1.1 has fewer opening moves than it should have. There are nine opening moves: X might move into the middle square, or he might move into one of the four corners, or into one of the four remaining fields. But for the purposes of the game *it does not make any difference* which corner is chosen, so we replace those four moves by just one, and similar for the remaining four moves. The reason for this is that the game board has a lot of symmetries (taking its mirror image or rotating it by 90 degrees gives us

another game board, and every play on a board, say, rotated by 90 degrees corresponds to one on the original board).

This allows to cut down on the number of options we have to consider. For the full game tree, there is an obvious mapping of plays into our reduced game tree. A similar situation arises for $(2, 2)$-nim: We begin with two piles of two matches each, and it does not matter whether the first player takes matches from the first pile or the second pile. We can cut the size of the game tree in half if we only consider one situation. Obviously it is vital when drawing a reduced game tree to ensure that all options are captured in some way, but it does pay off to keep things small.

If a game is in normal form then there is no point in keeping duplicate rows or columns, so we usually remove those.

Normal form

The advantage of using the normal form is that the decisions have been stratified: Each player has to pick a strategy, and that's all the choice he has to make. The complexity of the game tree has been subsumed in the number of available strategies. The pay-offs are very easy to compute once the strategies have been chosen—they just have to be looked up in a table. Working out which strategies to choose under which circumstances is a problem that has been long studied, and it is this form for which we currently have the most information.

However, there are a number of disadvantages that come with choosing this approach:

- Size. Computing the normal form of a game is not always feasible.
- Modelling. If it is our aim to model behaviour (and learning of behaviour) does it really make sense to start with the normal form? This is not how people play games, nor how they learn about them.

So far we have not worried about size at all, so we begin with some basic considerations.

Question 8. How does the number of positions grow with the height of the game tree?

If some game tree has at least two choices at each decision point[2] then the number of positions of a tree of height m is at least $2^{m+1} - 1$. In other words the number of positions is *exponential* in the height of the game tree. To put it differently, with the height the overall size of a tree grows very quickly.

As a result even constructing the full game tree is not feasible for all games. In fact, games played by people often have fairly compact rules (see for example chess, draughts, go), whereas the full game tree requires much space to be represented.

For the game of chess, for example, there are 20 opening moves for White (the eight pawns may each move one or two fields, and the knights have two possible moves each), and as many for Black's first move. Hence on the second level of the game tree we already have $20 \times 20 = 400$ positions (note how the possibilities are *multiplied* by each other). This is why most game rules are specified in a way so as to allow the players to derive the valid moves in *any given position*. In this sense the game tree is a *theoretic device* which allows us to reason about a game, but which may not be of much use when playing the game.

Question 9. How many plays are there for noughts and crosses? If you can't give the precise number, can you give an upper bound?

But even for a game where we could hold the whole game tree in memory calculating all the strategies may still be a problem.

Question 10. How does the number of strategies (added up over all players) grow with the size of the game tree?

[2]It would be sufficient for that to be the case at *most* decision points.

Estimating the number of strategies is more difficult than doing so for the positions in a game tree, because their number depends on how many decision points there are for each player. In the extreme case where all the decisions are made by the same player then the number of strategies in a game tree of height m is the same as the number of final positions, that is 2^m. But this is not very realistic, and if the decision points are distributed evenly then the number grows much more quickly.

As an example here is a table counting the number of positions where we assume that

- the game tree is a complete binary tree, that is, at every decision point there are precisely two choices and
- there are two players who move strictly alternating, starting with Player 1.

height	no pos.	strats for P1	strats for P2	all strats
0	1	1	1	2
1	3	2	1	3
2	7	2	4	6
3	15	8	4	12
4	31	8	64	72
5	63	128	64	192
6	127	128	16384	16512
7	255	32768	16384	49152
8	511	32768	1073741824	1073774592

S Exercise 11. (a) Take a complete binary tree of height 3. How many decision points does it have? Try different ways of assigning those to two players and count the number of strategies for each.

(b) In the table above, how can one calculate the number of strategies for each player from the previous entries?

Because of the exponential growth, generating all strategies is expensive. With a large game tree there are also problems with storing strategies efficiently—they would take up a good deal of space if stored as subtrees of the game tree. In the chapter on game-playing programs we study the issue of finding 'good' strategies without having to list *all* all of them.

We see from these considerations that calculating the normal form of a game is not feasible for many interesting games. Those certainly have to be studied in another way. Usually the extensive form plays a role in that.

Extensive form

In order to study a game in extensive form we do not have to have the entire game tree in memory: It is sufficient if we can *create it as we need it.* Hence we can mimic the natural procedure of giving the rules of a game by specifying what moves are legal in any position, and then dynamically create more of the game tree as we require it.

In addition, for some games this form is much more compact than any other: If we repeat a simple game over and over (examples of this are often studied when using games to model behaviour) then describing the game is very easy; we describe the simple game and a way of determining how often it is repeated. The number of strategies for each player for the full game can easily be enormous!

As well as making it feasible to cope with large games computationally using the extensive forms also has the advantage that it is closer to actually playing a game, which is useful when it comes to modelling behaviours.

The main disadvantage of using this form is that reasoning about it can be very cumbersome. Its complex structure makes that inevitable. Sometimes it is possible to get proofs by a simple induction on the height of the game tree, and an example of this appears in the next section. We encounter the minimax and the alpha-beta pruning algorithm for trees in Chapter 3, but beyond that most tasks are quite difficult.

Which form?

Which form of a game one should use depends on the size and nature of the game, and on what one wishes to do. In these notes we develop the two theories side by side as far as possible, concentrating on only one of them where that is inevitable.

Infinity

Many mathematical results we study here only hold for finite games, that is games with a finite number of players where the game tree consists of a finite number of positions.

There are very few cases where an infinite number of players is ever considered; this usually occurs in evolutionary games for strictly theoretical purposes—clearly any implementation has to be satisfied with only finitely many of them.

Game trees can be infinite in two different ways:

- There is at least one play that can be continued at any stage.
- There is at least one decision point where a player has an infinite number of choices.

Most games are designed to prevent the first case: The rule for chess which states that a player can claim a draw if a position is repeated three times is to ensure that the game does not continue forever, as are the rules that the game finishes in a draw if neither side can possibly force a checkmate. Games that may continue forever are rarely studied. The only exception I am aware of are of a theoretical nature when looking at games that may be repeated either an infinite or an indefinite number of times. These are sometimes used as theoretical devices when studying games that model behaviour. All the results in Chapter 2 assume that the game tree has finite height, that is there is no path of infinite length in the game tree.

Game theory has something to say about games where players may have an infinite number of choices in some decision points. However the mathematical analysis of these situations often requires differential equations, and most computer science students are not familiar with these. Hence we do not have much to say about this situation in these notes. You will encounter *Stackelberg games* in the second semester—they are games with infinitely many choices for the players which are simple enough to be analysed.

1.5 Simple two person games

We can state and prove a first result.

Definition 6. *In a 2-person zero-sum game a strategy for a player*

- *is* **winning** *if, when following that strategy, the player always gets a pay-off greater than* 0 *(no matter what the other player does);*
- **ensures a draw** *if, when following that strategy, the player always receives a pay-off greater than or equal to* 0 *(no matter what the other player does).*

Because in a 2-person zero-sum game the pay-offs for both players have to add up to 0 in all situations the above definition agrees with our usual notion of 'winning', and it makes sense to declare such a game a draw if both players receive a pay-off of 0.

Theorem 1.10. *Consider a 2-person zero-sum game with of perfect information and without chance such that every play ends after a finite number of moves. Then one of the following is the case:*

(i) Player 1 has a winning strategy;

(ii) Player 2 has a winning strategy;

(iii) Player 1 and 2 both have strategies which ensure a draw.

Proof. The proof proceeds by induction over the height of the game tree. The base case is given by a game of height 0, that is a game without moves. Clearly in this situation we merely have to read off the pay-off to see which of the three situation it falls under.

Assume that the statement is true for all games of height at most n. Consider a game of height $n+1$. This game can be considered as being constructed as follows: From the root, there are a number of moves (say k many) leading to game trees of height at most n.

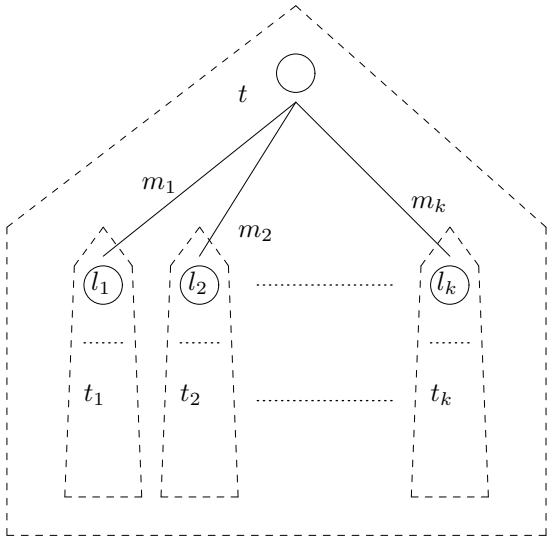


Figure 1.11: First moves and sub-games

By the induction hypothesis we can label the roots of these game trees with a number (say l_j for tree t_j) as follows:

- it bears label $l_j = 1$ if Player 1 wins the game rooted there;
- it bears label $l_j = -1$ if Player 2 wins the game rooted there;
- it bears label $l_j = 0$ if the game rooted there is such that either side can enforce (at least) a draw.

Now if the first move of the game is made by Player 1, then there are the following cases to be considered:

- There is a child of the root labelled with 1, that is there is j in $\{1, 2, \ldots, k\}$ such that $l_j = 1$. Then Player 1 can choose m_j as his opening move, and combine it with the winning strategy for the game rooted at that child. This results in a winning strategy for the whole game and case (i) is met.
- None of the children of the root is labelled with 1, (that is $l_j \neq 1$ for all $1 \leq j \leq k$) but there is at least one j with $l_j = 0$. Then by choosing m_j as his first move, Player 1 can ensure that game t_j is now played out where he can enforce a draw since $l_j = 0$. Hence Player 1 can enforce a draw in the overall game. To ensure that case (iii) is met we have to show that Player 2 also can enforce at least a draw. But all the games rooted at a child of the root of the overall game have label 0 or -1, so Player 2 can enforce at least a draw in all of them. Hence she can enforce at least a draw in the overall game.
- None of the children of the root is labelled with 1 or 0. That means that for all $1 \leq j \leq k$, $l_j = -1$ and Player 2 can enforce a win for all t_j. That means she has a winning strategy for the overall game, no matter which first move Player 1 chooses. Hence case (ii) is met.

The case where the first move of this game is made by Player 2 is symmetric to the one just discussed. □

A slightly more general statement (involving chance) was first made by Zermelo and later proved by John von Neumann. We give more general results below which subsume this one. Note that in order to actually find a winning strategy the entire game tree has to be searched if one is to follow the method given in the proof. We see in Chapter 3 how the idea underlying this proof can be turned into an algorithm that finds these strategies in a game of perfect information, and finds other useful strategies in the case of imperfect information.

Note that this result means that games like chess or go are intrinsically boring in that one of those three statements has to be true for each of them. The games are so large, however, that we currently are nowhere near deciding which of the three cases applies (though for chess it is usually assumed that case (i) or (ii) holds), and so we still find it worthwhile to play them. Contrast this with the game of noughts and crosses, where the third case applies. Children typically discover this after a having played that game a few times and discard it as a pastime thereafter.

B Exercise 12. (a) Take the game (2×2)-chomp from Example 1.2. Does either player have a winning strategy, and if yes, who, or can they both force a draw?

(b) Answer the same question for the game of $(2, 2)$-nim from Exercises 1 (a) 6 (a).

Summary of Chapter 1

- Games can be represented in their *extensional form*using a *game tree*. Typically, a position in such a tree contains more information than just what would be shown on the board.
- Elements of *chance* are then modelled by having a player called Nature, with probabilities labelling such moves. *Imperfect information* is modelled by including in the game tree information about any nodes which cannot be distinguished by the player about to move.
- The *pay-off function* for a player assigns a value to each of the possible outcomes possible in the game.
- A *strategy* for a player is a complete game plan for that player. It can either be *fully specified* or be made not to contain redundant information. It provides a choice of move for *every* situation in which the player might find himself.
- Any game can be transferred into its *normal form*, which means it is described as a simultaneous one-move game: Each player picks one of his available strategies, and there is a pay-off function for each player for each combination of choices. Computing the normal form is not feasible for games above a certain size.
- Whether one should use the extensional or normal form of a game depends on the situation.
- In 2-player zero-sum games of perfect information without chance either one of the players can force a win, or they can both force a draw.

Sources for this section. Many textbooks start with the normal form of a game and hardly mention the extensive one. Even those that do take the transformation between the two for granted. While I have used various sources to help with the examples I do not know of an account which covers the same ground.

Chapter 2

Equilibria

In this chapter we seek to answer the question of what it means to play a game well. However, the most studied solution has some drawbacks which we also cover.

The answer to our question comes in the form of a suggestion which strategies to play, and what to expect from doing so. It is important to note that while we refer to strategies here there is no underlying assumption that the game is in normal form. For much of what we do there is no need to calculate all strategies, and so the observations we make regarding particular strategies hold generally. However, finding strategies that have been identified as desirable is another issue. We look at that issue both, in this chapter and also the following ones.

2.1 Best response

Assume we are playing a game, and we know which strategy each of the other players is going to employ. Assume we are Player 1 and that there are also Players 2 to l. If we know that Player 2 plays a specific strategy, say s_2, Player 3 plays s_3 and so on, with s_l being employed by Player l we can ask the following question:

Which of our strategies gives us the highest pay-off in that situation? In other words, which strategy s_1^* has the property that

$$\text{for all strategies } s_1 \qquad p_1(s_1^*, s_2, \ldots, s_l) \geq p_1(s_1, s_2, \ldots, s_l)?$$

Note that there might be more than one strategy with that property. We call such a strategy **the best response for Player 1 to** $(;s_2, s_3, \ldots, s_l)$. If we know what the other players are going to do then any of our best response strategies in that situation will give us the maximal pay-off we can hope for. So when trying to maximize our pay-off (which every player is trying to do) we are looking for best response strategies. Of course, these considerations apply to all the players in a game, not just Player 1—for each player i, given a choice of strategy for all the other players we can look for strategies maximizing p_i.

It is possible to calculate the best response to a given choice of strategies for the other players (and some learning algorithms are concerned with learning such best responses). However, this assumes that one knows in advance what the other players are going to do!

One might hope to find a best response strategy that works for *every* situation, that is for example a strategy s_1^* for Player 1 with the property that

- for all strategies $s_2, \ldots, s_l$ for Players 2,..., l respectively and
- for all strategies s_1 for Player 1 it is the case that

$$p_1(s_1^*, s_2, \ldots, s_l) \geq p_1(s_1, s_2, \ldots, s_l).$$

Note that since we are now quantifying over all $s_2, \ldots, s_l$ this is a much stronger requirement than before. Unfortunately most games do not have strategies that are best responses to *all* strategy choices for the other players at the same time.

Proposition 2.1. *If a player has a winning strategy in a 2-person zero-sum game with pay-offs in* $\{-1, 0, 1\}$ *then it is a best response to all the the other player's strategies.*

A Exercise 13. Prove this proposition.

It is worth remembering that the pay-off functions the players are trying to maximize in games of chance only cover the *expected pay-off*—if there is chance involved then these pay-offs are not guaranteed, they are merely the 'expected' ones. In other words they are the average of what can be expected when playing the game a large number of times.

Question 11. Assume we are playing a game where we throw a coin, and one of us bets on heads while the other bets on tails. If you win, you have to pay me a million pounds, otherwise I pay you a million pounds. What is the expected pay-off in this game? Would you be willing to play a round?

This shows once again that pay-off functions have to be carefully chosen. If we merely use the monetary payment involved in the game in Question 11 and then just look at the *expected pay-off* it seems a harmless game to play—on average neither of us will lose (or win) anything. In practice, this doesn't cover all the considerations each of us would take into account before playing this game. One solution to this might be not to merely use the monetary payment as a pay-off, but rather make it clear that neither of us could afford to pay out that sort of money. If, for example, we set the pay-off for losing a million pound to $-100,000,000,000,000$, or the like, and kept 1,000,000 for the win of the million, then the 'expected result' would better reflect our real opinion.

The theory of equilibria is the study of choices of strategies $(s_1, s_2, \ldots, s_l)$ with the property that each s_i is the best response to the remaining strategies.

2.2 Equilibria

The idea is simple: If each player is seeking to play his best response to the choice of strategy by the other players then we must be particularly interested in situations where every player has found a strategy that is the best response to the others present. We can think of the players as having found a point of balance: If any one of the players unilaterally decides to change his strategy then he's risking moving away from a best response to all the other strategies, but that means his pay-off can never get better, it can only get worse. Hence each player has an incentive to stay with such an equilibrium point.

Definition 7. *For a game with* l *players we say that* $(s_1^*, s_2^*, \ldots, s_l^*)$ *is a* **(Nash) equilibrium point** *for the game if*

- *each* s_i^* *is a strategy for Player* i *and*
- *each strategy* s_i^* *is the best response to* $(s_1^*, s_2^*, \ldots, s_{i-1}^*, \cdot, s_{i+1}^*, \ldots, s_l^*)$*, that is for all strategies* s_i *for Player* i

$$p_i(s_1^*, \ldots, s_{i-1}^*, s_i^*, s_{i+1}^*, \ldots, s_l^*) \geq p_i(s_1^*, \ldots, s_{i-1}^*, s_i, s_{i+1}^*, \ldots, s_l^*).$$

We say that the s_i^* *are* **simultaneous best responses to each other**.

Note that even if all but one player announce their choice to play their part in an equilibrium point then the remaining player is best off also playing his—it is, after all, a best response to the collective choice of the others. This is a particular property of equilibria.

These equilibria are often referred to as *Nash equilibria* in the literature, after John Nash. He is a mathematician who won the Nobel prize (for economy) for his work in game theory in 1994, 45 years after his ground-breaking paper on the subject first appeared. Nash suffered from schizophrenia for decades, but recovered in the 1990s. If you want to find out more about him, `http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Nash.html` gives

a brief history. You are also encouraged to read the acclaimed biography *A Beautiful Mind*, by Sylvia Nasar (Simon & Schuster, 1998—there's also a paperback edition by Faber and Faber, 1999). Or, if you can't stomach a long book you can always watch the award-winning film of the same title.

There are other notions of equilibria but most of them are rather more complicated and we do not study them here.

Question 12. What can you say about the equilibria for a 2-person zero-sum game of perfect information without chance? Hint: Consider whether Theorem 1.10 helps here.

Proposition 2.2. *For every 2-person zero-sum game of complete information without chance and with pay-offs in* $\{-1, 0, 1\}$ *exactly one of the following holds:*

- *There is an equilibrium point that contains a winning strategy for Player 1 or*
- *there is an equilibrium point that contains a winning strategy for Player 2 or*
- *there is an equilibrium point that consists of two strategies ensuring draws.*

To illustrate the idea of an equilibrium we have a look at a particular example.

Example 2.3. Camping holiday. Let us assume there is a couple heading for a camping holiday in the American Rockies. They both love being out of doors, there is just one conflict that keeps cropping up. Amelia[1] appreciates being high up at night so as to enjoy cool air which means she will sleep a lot better. Scottie[1] on the other hand has a problem with the thin air and would prefer sleeping at a lower altitude, even if that means it's warm and muggy. In order to come to a decision they've decided upon the following: The area has four fire roads running from east to west and the same number from north to south. They have decided that they will camp near a crossing, with Scottie choosing a north-south road while Amelia decides on an east-west one, independently from each other. The height (in thousands of feet) of these crossings (with the roads numbered from east to west and north to south) is given by the following table.

		Scottie			
		1	2	3	4
Amelia	1	7	2	5	1
	2	2	2	3	4
	3	5	3	4	4
	4	3	2	1	6

Are there any equilibria in this game?

Let's consider the situation from Amelia's point of view. If she chooses Road 1 (with a potential nice and airy 7000 feet) then Scottie's best response Road 4 pushes her down as far as 1000 feet. If she decides on her Road 2 then Scottie can decide between his Roads 1 and 2, and they'll still be sleeping in a hot and humid 2000 feet. In other words to calculate Scottie's best response to each of her choices, she finds the *minimum* of each row. Note that Scott's best responses are the same as her 'worst case'. We can summarize these considerations in this table.

Road No	min. height
1	1
2	2
3	3
4	1

Her best response to Scottie's best responses is Road 3. If she chooses that then they will sleep at a guaranteed 3000 feet, no matter what Scottie does. In all other cases he can push

[1]Yes, there are some very obscure references hidden in those names (one for each). A chocolate bar to anybody who can figure those out! Hint: Look back a few decades.

her further down. So she chooses the *maximum* of the entries in the new table by choosing her Road 3.

Let's now look at the situation from Scottie's point of view. If he chooses Road 1 then Amelia's best response is her Road 1, leaving him at a scary 7000 feet, the mere thought of which makes him feel somewhat nauseous. If he chooses Road 2, on the other hand, then at Amelia's best response can push him up to 3000 feet. In order to calculate her best response to each of his choices he looks for the *maximal* entry in each column. This is also the worst that can happen to him in each case. The result is summarized in the following table.

Road No	max. height
1	7
2	3
3	5
4	6

His best response to her best responses is to choose the strategy which gives him the least of these numbers, guaranteeing that he will not have to sleep above 3000 feet. Hence he goes for his Road 2.

The situation is pictured in Figure 2.1. Here the dashed lines are the roads which lead from north to south and the solid lines are those running from east to west.[2]

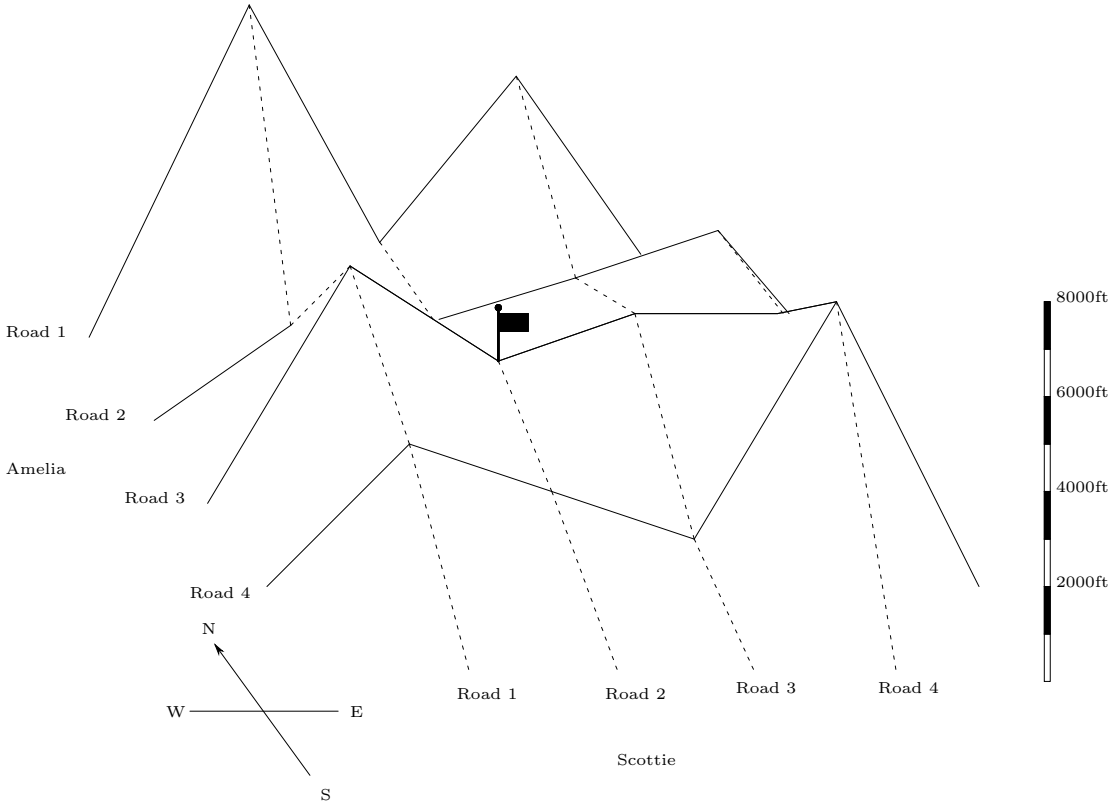


Figure 2.1: A landscape

The flag indicates where they will eventually camp.

Question 13. What happens if Amelia changes her mind while Scottie sticks with his choice? What if it is the other way round, that is, Amelia keeps her choice while Scottie changes his?

We can now check that their choices—Road 3 for Amelia, Road 2 for Scottie—is indeed an equilibrium point leading to a 'pay-off' of 3000ft. In fact, the proof is included in the discussion above, but here we make it more explicit.

It is worth pointing out that from Amelia's point of view

[2]Note that this 'landscape' shows the terrain in terms of the roads only.

- if she changes her mind, but Scottie doesn't, then the situation will worsen as far as she's concerned, that is they will camp at a lower site.

From Scottie's point of view, on the other hand, it is the case that

- if he changes his mind while Amelia doesn't then the situation will worsen as far as he's concerned, that is they will stay even higher up.

In other words if we cut along Scottie's choice of roads (which corresponds to Amelia changing her mind while Scottie sticks to his choice) then the point they choose lies on the top of a hill (see Figure 2.2)—if she changes her mind, they will end up lower down. If, on the other hand, we cut along her choice (which corresponds to Scottie changing his mind) then their site lies in a valley (see Figure 2.3). Such points are known as **saddle points**.

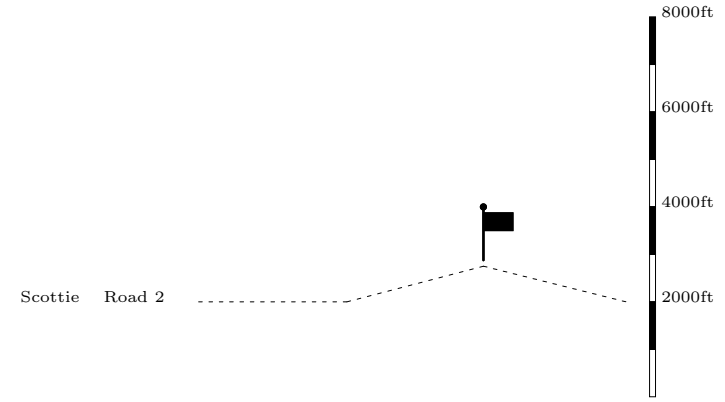


Figure 2.2: Road 2 (north-south), Scottie's choice

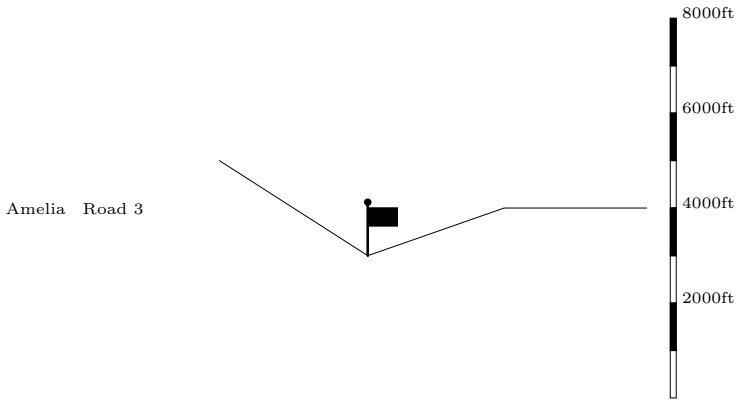


Figure 2.3: Road 3 (east-west), Amelia's choice

However, more is true about these points: In fact, there is no point on Road 3 which is below the chosen one, nor is there a point above it on Road 2. This is a stronger condition since Road 3, for example, might dip into a valley, then go up, and then go very far down again—leaving a saddle point as before, but violating this new observation.

Hence the choices $(3, 2)$ are best responses to each other and this is indeed an equilibrium point of the game (if we assume that Scottie's pay-off is, say, the negative of Amanda's, which is given in terms of the height).

As we see below, however, the way in which Amelia and Scottie arrived at their equilibrium point choice does not work for all games in normal form, even if those are 2-person zero-sum games.

B Exercise 14. Find the equilibria for the following games. Do so by checking for pairs of strategies which are best responses to each other.

(a) $\left|\begin{array}{cc} (-10,5) & (2,-2) \\ (1,-1) & (-1,1) \end{array}\right|$ (b) $\left|\begin{array}{cc} (1,2) & (0,0) \\ (0,0) & (2,1) \end{array}\right|$

The principle underlying the notion of an equilibrium point is that it is a 'point of balance' for all players of the game because if only one of them moves away from it they risk a worse pay-off than they are currently getting—after all, they are all playing the best response to everybody else's collective choice. In the next section we study the question whether this is always a sensible assumption to make. The reason this notion works so well in 2-person zero-sum games (evidence for that is yet to come) is that what is good for one player is automatically bad for the other, and *vice versa*.

If we consider fully-specified strategies we can consider a stronger notion of equilibrium point.

Definition 8. *Let a* **sub-game** *of a given game be obtained by choosing any position in the game tree and considering it as the root of a game tree, namely the one given by the part of the original tree which is below it.*

A **sub-game equilibrium point** *in a game is given by a tuple of fully-specified strategies, one for each player, such that this tuple gives an equilibrium point for each sub-game of the game, even for sub-games that wouldn't be reached when playing these strategies.*

Question 14. Why doesn't it make sense to think of a sub-game equilibrium point as given by a tuple of strategies?

Clearly every sub-game equilibrium point is an equilibrium of the original game. The converse is false. We can think of a sub-game equilibrium point as a particularly 'perfect' equilibrium: No matter where in the game we might start off the strategies involved they will give us the equilibrium play from that point.

This notion becomes particularly important when studying games that are played repeatedly over lots of rounds. We do not look at it closer here but this seems a sensible point for introducing it.

2.3 Equilibria in non zero-sum games

We look at non-zero sum games to get an idea of what equilibria can arise, and what properties they might have.

Consider the pay-off table given by

$$\left|\begin{array}{cc} (-20,-20) & (15,-15) \\ (-15,15) & (10,10) \end{array}\right|$$

It has two equilibrium points at $(1,2)$ and $(2,1)$, each of them being preferred by one of the players. Clearly, once they have settled on one of these, each player risks the utterly undesirable outcome of $(-20,-20)$ when unilaterally moving away from the equilibrium point. But how should they decide to settle on a particular equilibrium point? Remember that they are not allowed to cooperate, for example by agreeing to switch between the two on a regular basis.

The option $(2,2)$ is certainly a compromise of some sort, but how can the players get there if they are not allowed to communicate with each other? And wouldn't it be tempting for either of them to switch strategies to increase the pay-off from 10 to 15? That's why the solution $(2,2)$ which leads to a pay-off of 10 for each player cannot be considered stable. But certainly there is a question whether we can consider either equilibrium point a 'solution' for this game.

Example 2.4. The prisoner's dilemma. Two dark figures, Fred and Joe, have been caught by the police and are now being questioned—in separate rooms. The police have the problem that they do not have firm evidence that the two have committed a major crime—if they both keep mum then the best the police can expect is that they each get two years. The police are therefore interested in bluffing them into making a confession, offering that if one of them turns crown witness he will get off scot-free while the other will face 10 years in prison. (It is not stressed that if they both confess they will each face 8 years, two years having been deducted due to the confession.)

Maybe somewhat surprisingly, the players in this game are the two prisoners. Each of them faces a choice: to confess or to stay quiet. It seems tempting at first not to say anything—after all, if Joe only does the same then Fred will get away with just two years. On the other hand, two years in gaol is a long time. If he talks he might walk away a free man. And, of course, can he really trust Joe to keep quiet? If Joe shops him he's looking at ten years while Joe is out. Surely it's better to at least have the confirmation that Joe is suffering as well for his treachery.

Here is the pay-off table (in years spent in prison, with a negative number to make it clear that 10 years in prison are worse than 2, and so to fit our interpretation of the pay-off functions) for the situation. The number on the left of each pair shows the pay-off for Fred, the number on the right that for Joe.

		Joe	
		talk	don't talk
Fred	talk	$(-8,-8)$	$(0,-10)$
	don't talk	$(-10,0)$	$(-2,-2)$

This game has an equilibrium point at (talk, talk), since for both, Joe and Fred, the situation will get worse if they shift away from that strategy. This is the only choice of strategies that are best responses to each other.

Hence from the game theory point of view, the 'solution' to this game is for each of them to talk and spend 8 years in prison. Clearly, this is not a particularly good outcome. If one takes their collective situation into account, it is very clear that what they should both do is to remain silent (much to the regret of the police!).

Question 15. What would you do in a situation like Joe and Fred? You don't have to picture yourself as a prisoner to come into a similar dilemma. For example, assume somebody is offering goods for sale on the Internet and you're interested in buying. Neither of you wants to pay/send the goods first, so you decide you'll both send your contribution to the other party at the same time. Isn't it tempting to let the other guy send you the goods without paying? Would you change your mind if you wanted to do business with this person again (which amounts to playing the game again)?

There are, in fact, a large number of 'prisoners' dilemma type' situations. We may study these when we look at using games to model a number of situations.

Here is another example. Douglas R. Hofstadter once sent a postcard with the following text to twenty of his friends:[3]

'. . . Each of you is to give me a single letter: 'C' or 'D' standing for 'cooperate' or 'defect'. This will be used as your move in a Prisoner's Dilemma with *each* of the nineteen other players. The pay-off table I am using for the Prisoners' Dilemma is given in the diagram.

		Player B	
		C	D
Player A	C	$(3,3)$	$(0,5)$
	D	$(5,0)$	$(1,1)$

[3]He summarized this in his column in the *Scientific American, June 1983.* These columns, with a few extra articles, can be found in his book *Metamagical Themas.*

Thus if everyone sends in 'C', everyone will get \$57, while if everyone sends in 'D', everyone will get \$19. You can't lose! And, of course, anyone who sends in a 'D' will get at least as much as everyone else will. If, for example, 11 people send in 'C' and 9 send in 'D', then the 11 C-ers will get \$3 apiece for each of the other C-ers, (making \$30), and zero for the D-ers. So C-ers will get \$30 each. The D-ers, by contrast, will pick up \$5 apiece for each of the C-ers, making \$55, and \$1 each for the other D-ers, making \$8, for a grand total of \$63.

...You are not aiming at maximizing the total number of dollars *Scientific American* shells out, only maximizing the number that come to *you*!

...I want all answers by telephone (call collect, please) the day you receive this letter.

It is to be understood (it *almost* goes without saying, but not quite) that you are not to try to get in touch with and consult with others who you guess have been asked to participate. In fact, please consult with no one at all. The purpose is to see what people will do on their own, in isolation. ...'

Question 16. What would you do if you received such a letter? And why? How many people do you think chose 'C', how many 'D'?

Hofstadter had hoped for twenty 'C's.

Question 17. Can you think of a way in which a clever person could have convinced himself that everybody should go for 'C'?

In fact, he got 14 'D's and 6 'C's, so the 'defectors' each received \$43 while the 'cooperators' had to make do with \$15 each.

Question 18. What would game theory have to tell these people? What would have happened if they had all applied this theory? On the other hand, how do you think this sort of game would develop if it were played repeatedly, say once a week over a year?

B Exercise 15. Discuss the relative merits of the 'solutions' given by the equilibria for the following non-cooperative games. What if the pay-off is in pound sterling, and you are the player having to make a decision?

(a)
$$\left|\begin{array}{cc} (4,-300) & (10,6) \\ (8,8) & (5,4) \end{array}\right| \qquad \text{(b)} \qquad \left|\begin{array}{cc} (4,-300) & (10,6) \\ (12,8) & (5,4) \end{array}\right|$$

If you think that the above examples are artificial in nature, maybe the following two will be more to your taste. If nothing else they are taken from the area of computing.

Example 2.5. TCP/IP Congestion Control Protocol. All TCP implementations are required to support the algorithm known as *slow-start*. It works as follows:

- Start with a batch size of one packet;
- keep doubling the number of packets in a batch until a predefined threshold size is reached;
- from then on increase the batch size by one with each transmission;
- when congestion occurs (which is detected by a time-out occurring when waiting for an acknowledgement), reset the threshold value to half the current batch size and start over (with a batch size of one packet).

The idea of the algorithm is to adapt the batch size to what is currently supportable within the network, and it is fair since everybody follows the same rules, thus trying to split resources evenly between all users.

For an individual, however, it is tempting to employ a 'greedier' algorithm than the above slow start: For example jump straight back to the new threshold value (rather than starting with a batch size of 1). (It doesn't make sense to go for a bigger size since in all likelihood, that will just result in more time-outs.) But if everybody does that, the network will become

very congested. This is a multi-player prisoners' dilemma-type situation: The community is best off if everybody exercises restraint. But a small number of people could get away with getting a better return than the average (and, indeed, the slow-start algorithm strategy does not lead to an equilibrium even when everybody employs it). Of course, this only works if only a very few people try to cheat.

Example 2.6. ALOHA Network Algorithm. An early, elegant (and very successful) algorithm for allocating a multiple-access channel (initially designed for ground-based radio broadcasting) works as follows:

- Send a data packet as soon as it becomes available for sending;
- if a collision occurs (because somebody else accessed the channel at the same time), wait a random amount of time and try again.

Using a random delay means that packets are unlikely to collide repeatedly. Again, the algorithm is fair because everybody uses the same algorithm, and thus on average has the same amount of waiting time.

Once more it is very tempting for the individual to try to get away with retransmitting a packet immediately—and if there's just one such individual, he's likely to get away with it. Again this is a multi-player prisoners' dilemma-type situation.

Question 19. What happens if two players try to get away with immediate retransmissions?

While both the situations given in Examples 2.6 and 2.5 are games which are carried out repeatedly, it is impossible for the other players to tell whether any of the others is cheating. Hence any incentive to behave oneself that might exist due to social pressure is lost. There has been a lot of recent interest in using games to come up with algorithms that help with the fair use of resources such as a channel or bandwidth, while maximising throughput. An important issue here is that a great number of agents are trying to get their messages across the network, and coming up with ways for each of them to behave to achieve the two targets is not easy. There has been particular attention given to the learnability of such solutions (one might decide not to go for an optimal solution but one that is only slightly worse, but more easily achievable). However, much of the material is too advanced for this course unit. Some examples can be found in *Algorithmic Game Theory*.

As we can see from this section considering equilibria as 'solutions' to non-zero sum games is somewhat problematic. While there are sometimes compelling reasons that individuals will choose equilibria (for example in the prisoners' dilemma game) these often do not constitute the optimal outcome for both players. Moreover, when observing behaviours 'in the real world' one finds that often people do not behave so selfishly.

There are different approaches to addressing these issues.

- Instead of concentrating on individual one can look at the system arising from a number of agents playing such games. One might then ask whether there are stable states for such systems. That gives rise to a different notion of solution.
- Rather than focussing solely on pay-offs gained one can consider different ways of measuring success in a game. There is a notion of *regret*, typically measured by the difference between the actual pay-off and that which one might have achieved by making a different decision. In some cases minimizing the regret (over time) can lead to different solutions than calculating an equilibrium point.
- One can allow the agents playing the game to negotiate with each other rather than forcing them to make a decision without consultation.

B Exercise 16. (a) Find all equilibria in the game from Exercise 10 (a).

(b) Do the same for the game from Example 1.9, whose normal form is given on page 24.

A Exercise 17. For the game in Exercise 10 (b) can you find a path in the game tree that leads to an equilibrium point pay-off? (It is possible to do so without writing out the normal form, although it might be helpful to draw a game tree first.) How many strategies lead to this pay-off, and how many equilibrium points exist? *T*his part goes beyond a simple exercise in places.

2.4 Properties of equilibria in 2-person zero-sum games

When we only consider 2-person zero-sum games then the idea of considering the equilibria of a game as its solutions becomes much more compelling. We give here some of the properties enjoyed by such solutions to show why this is so.

We begin by looking at a simple way of finding some equilibria.

We need some formal notation. Let us assume we have a matrix (or table) with elements $a_{i,j}$, where i indicates the row and j indicate the column.

$$\begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{vmatrix}$$

We say that this is an $(m \times n)$ **matrix**. A 2-person zero-sum game in normal form is described by an $(m \times n)$-matrix where m is the number of strategies for Player 1 and n plays the same role for Player 2.

Proposition 2.7. *For a 2-person zero-sum game in normal form the strategy pair (i, j) is an equilibrium point if and only if the corresponding pay-off is maximal in its column and minimal in its row.*

Proof. Assume that (i, j) is an equilibrium point. Since i is a best response to strategy j it has to be one of the maximal pay-offs possible when playing against strategy j. These pay-offs are given by the column corresponding to j (which varies Player 1's strategy while that of Player 2 stays fixed). Since j is a best response to i (and Player 2 is trying to minimize the pay-offs as given in the matrix) it similarly has to be the case that it is minimal it its row.

If on the other hand $a_{i,j}$ is maximal in its column then i is a best response to j, and if it is also minimal in its row then j is a best response to i. □

This result gives us a very convenient algorithm for finding equilibria: We merely have to check whether any given value in the matrix is maximal in its column and minimal in its row.

B Exercise 18. Find the equilibria in the 2-person zero-sum games given by the following matrices:

(a) $$\begin{vmatrix} 4 & 3 & 1 & 1 \\ 3 & 2 & 2 & 2 \\ 4 & 4 & 2 & 2 \\ 3 & 3 & 1 & 2 \end{vmatrix}$$ (b) $$\begin{vmatrix} 2 & -3 & 1 & -4 \\ 6 & -4 & 1 & -5 \\ 4 & 3 & 3 & 2 \\ 2 & -3 & 2 & -4 \end{vmatrix}$$

We now look at some other properties of equilibria. To illustrate the next point we go back to the example of Amelia and Scottie.

Amelia's first step consisted of calculating, for a fixed row, that is for a fixed i, the *minimum* of all the $a_{i,j}$ where j ranges over the number of columns (here $1 \leq j \leq 4$). That is, she determined, for each $1 \leq i \leq 4$,

$$\min_{1 \leq j \leq 4} a_{i,j}.$$

And then she calculated the largest one of those to make the corresponding road her choice, that is she computed

$$\max_{1 \leq i \leq 4} \min_{1 \leq j \leq 4} a_{i,j}.$$

Scottie, on the other hand, first computed, for a fixed column, that is for a fixed j, the maximum of the $a_{i,j}$, that is

$$\max_{1\le i\le 4} a_{i,j}.$$

Then he took the least of those and chose accordingly, that is he looked at

$$\min_{1\le j\le 4} \max_{1\le i\le 4} a_{i,j}.$$

We can summarize this information in the original table of heights as follows.

		Scottie				
		1	2	3	4	min of row
Amelia	1	7	2	5	1	1
	2	2	2	3	4	2
	3	5	3	4	4	3
	4	3	2	1	6	1
	max of col.	7	3	5	6	3\3

So Amelia calculated

$$\max_i \min_j a_{i,j}$$

and matching strategies

while Scottie looked at

$$\min_j \max_i a_{i,j}$$

and associated strategies.

Note that in a general 2-person zero-sum game given by a matrix as above we have the following.

The value

$$\max_i \min_j a_{i,j}$$

is the **largest expected pay-off Player 1 can guarantee for himself**;

the value

$$\min_j \max_i a_{i,j}$$

is the **largest expected pay-off Player 2 can guarantee for herself**.

We can see that this is true by the following consideration. From Player 1's point of view, for her strategy i the worst that can happen is the pay-off $\min_j a_{i,j}$. What she does is to maximize the worst case that can occur. For Player 2 the situation works similarly, we just have to remember that Player 2 is trying to *minimize* the pay-offs as given by the matrix (because they are the pay-offs for Player 1).

In the example of Amelia and Scottie the two numbers are equal, and the pay-off is that of the (only) equilibrium point of the game. This is no coincidence.

B Exercise 19. For the zero-sum games in normal form given below, calculate

$$\max_{1\le i\le m} \min_{1\le j\le n} a_{i,j} \quad \text{and} \quad \min_{1\le j\le n} \max_{1\le i\le m} a_{i,j}:$$

(a)
$$\begin{vmatrix} 4 & 3 & 1 & 1 \\ 3 & 2 & 2 & 2 \\ 4 & 4 & 2 & 2 \\ 3 & 3 & 1 & 2 \end{vmatrix}$$

(b)
$$\begin{vmatrix} 2 & 3 & 4 & 1 \\ 4 & 2 & 3 & 2 \\ 1 & 2 & 3 & 2 \\ 3 & 1 & 2 & 3 \end{vmatrix}$$

Informally the following is the case: If we have an equilibrium point for a game then these two numbers have to be equal, and they give the pay-off at this equilibrium point. On the other hand, if these two numbers are equal then we can find an equilibrium point.

Proposition 2.8. *Let (i^*, j^*) be an equilibrium point for a 2-person zero-sum game in normal form with m strategies for Player 1 (rows) and n strategies for Player 2 (columns) and pay-off matrix*

$$\begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{vmatrix}.$$

Then it is the case that

$$a_{i^*,j^*} = \min_{1\le j\le n} \max_{1\le i\le m} a_{i,j} = \max_{1\le i\le m} \min_{1\le j\le n} a_{i,j}.$$

If on the other hand

$$\min_{1\le j\le n} \max_{1\le i\le m} a_{i,j} = \max_{1\le i\le m} \min_{1\le j\le n} a_{i,j},$$

then the game has an equilibrium point.

Proof. Let us first assume that the game has an equilibrium point. Since a_{i^*,j^*} is the maximum of its column, that is $a_{i^*,j^*} = \max_{1\le i\le m} a_{i,j^*}$, it is the case that

$$\begin{aligned} a_{i^*,j^*} &= \max_{1\le i\le m} a_{i,j^*} \\ &\ge \min_{1\le j\le n} \max_{1\le i\le m} a_{i,j}. \end{aligned}$$

Since it is also the case that a_{i^*,j^*} is the minimum of its row we can calculate

$$\begin{aligned} a_{i^*,j^*} &= \min_{1\le j\le n} a_{i^*,j} \\ &\le \max_{1\le i\le m} \min_{1\le j\le n} a_{i,j}. \end{aligned}$$

Hence

$$\max_{1\le i\le m} \min_{1\le j\le n} a_{i,j} \ge a_{i^*,j^*} \ge \min_{1\le j\le n} \max_{1\le i\le m} a_{i,j}.$$

We next show that

$$\max_{1\le i\le m} \min_{1\le j\le n} a_{i,j} \le \min_{1\le j\le n} \max_{1\le i\le m} a_{i,j}$$

We know that for a fixed $1 \le i \le m$ and all $1 \le j \le n$ it is the case that

$$\min_{1\le j\le n} a_{i,j} \le a_{i,j}.$$

Now for all $1 \le j \le n$ we can take the maximum over all such i to get

$$\max_{1\le i\le m} \min_{1\le j\le n} a_{i,j} \le \max_{1\le i\le m} a_{i,j}.$$

But this is true for *all* j, and therefore it must also be true for the smallest such term, that is

$$\max_{1\le i\le m} \min_{1\le j\le n} a_{i,j} \le \min_{1\le j\le n} \max_{1\le i\le m} a_{i,j}.$$

We are done with this direction.

Let us now assume that the equation

$$\min_{1\le j\le n} \max_{1\le i\le m} a_{i,j} = \max_{1\le i\le m} \min_{1\le j\le n} a_{i,j}$$

holds. We use v for this number. We can now choose i^* in $\{1, \ldots, m\}$ such that

$$\min_{1 \leq j \leq n} a_{i^*,j} = v$$

and j^* in $\{1, \ldots, n\}$ such that

$$\max_{1 \leq i \leq m} a_{i,j^*} = v.$$

We claim that (i^*, j') is an equilibrium point.

We note that

$$v = \min_{1 \leq j \leq n} a_{i^*,j} \leq a_{i^*,j^*} \leq \max_{1 \leq i \leq m} a_{i,j'} = v$$

and so all these numbers are equal. In particular $\min_{1 \leq j \leq n} a_{i^*,j} = a_{i^*,j^*}$ says that a_{i^*,j^*} is minimal in its row, and $\max_{1 \leq i \leq m} a_{i,j'} = a_{i^*,j^*}$ says that a_{i^*,j^*} is maximal in its column, so it is indeed an equilibrium point. □

We get an immediate consequence:

Corollary 2.9. *All equilibrium points in a 2-person zero-sum game lead to the same pay-off.*

Question 20. Can you see why this result immediately follows from the previous one?

This means that some of the issues discussed in the previous section do not arise in the case of 2-person zero-sum games: All equilibria for such a game lead to the same pay-off, so it cannot happen that there are several equilibria each preferred by one of the players.

It also means we can refine our algorithm for finding equilibria: When we have found one equilibrium we only have to look at those entries in the matrix which are equal to the pay-off at the first equilibrium.

Definition 9. *For a 2-person zero-sum game the (unique) pay-off at an equilibrium point is known as the* **value of the game**.

If we have a 2-person game with several equilibria then something interesting happens.

Proposition 2.10. *Let (i, j) and (i', j') be equilibria for a 2-person zero-sum game. Then (i, j') and (i', j) are also equilibria for that game.*

Proof. The proof of this result is much easier if we assume that the game is in normal form. First of all note that we have established already that the pay-offs at the given equilibria have to be equal, that is $a_{i,j} = a_{i',j'}$. Now consider the pay-off at (i, j'), $a_{i,j'}$. It is in the same column as $a_{i',j'}$, and since that value is maximal in its column it has to be the case that $a_{i,j'} \leq a_{i'.j'}$. Similarly $a_{i,j}$ is minimal in its row and so $a_{i,j} \leq a_{i,j'}$. But $a_{i,j} = a_{i',j'}$ and so $a_{i,j'}$ has to be equal to that number. But that means it is equal to the maximal value in its column and also equal to the minimal value in its row and hence it is the pay-off at an equilibrium point. The proof that (i', j) is also an equilibrium point is much the same. □

So if we have several equilibria we can often combine them to give even more. Moreover, it is possible to pair any strategy for Player 1 that appears in some equilibrium point with any strategy for Player 2 that appears in some equilibrium point and one gets another equilibrium. Hence it makes sense to speak of **an equilibrium point strategy for a player** (some sources also speak of an *optimal* strategy). Note that this only makes sense in the 2-person zero-sum case.

In the various discussions above we have established a result regarding the value of the game:

Proposition 2.11. *A 2-person zero-sum game has an equilibrium point if and only if there exists a value $v \in \mathbb{R}$ such that*

- *v is the highest pay-off Player 1 can guarantee for himself;*

- *$-v$ is the highest pay-off Player 2 can ensure for herself.*

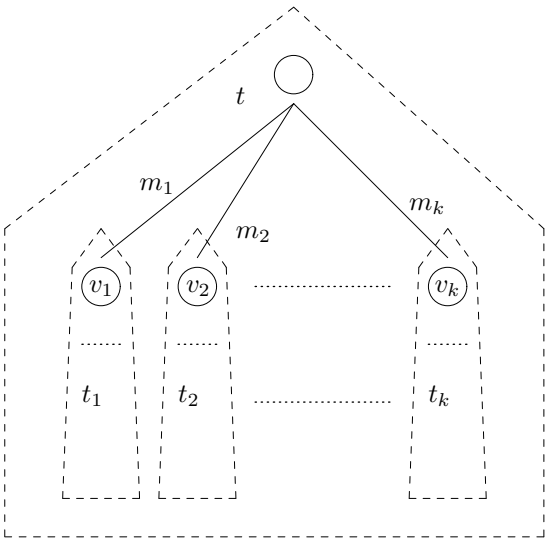
This number v is the value of the game.

A Exercise 20. Find the proof of this result in the discussion so far.

Note that to formulate the previous two results we do not have to assume that the game is given it normal form. There is one final result we can establish at this point.

Proposition 2.12. *Every 2-person zero-sum game of perfect information in which every play is finite has at least one equilibrium point.*

Proof. The proof of this result is similar to that of Theorem 1.10. A few changes have to be made, however. In that Theorem we assumed we only tracked outcomes regarding who was winning by using numbers 1, -1, or 0 (it is okay to think of these as pay-offs). Now we want to keep track of guaranteed pay-offs for both players. The induction hypothesis changes to the assumption that each game of height at most n has a value $v \in \mathbb{R}$, which we write down at its root.



The second adaptation is somewhat more complicated: In Theorem 1.10, we did not allow the game to include any elements of chance. So apart from nodes where it is Player 1's or Player 2's turn, we now have to include nodes where a random choice occurs.

Clearly every game of no moves has an equilibrium point—each player has precisely one strategy so they have to be best responses to each other. Thus every game of height 0 has a value, and we assume that this is true for games whose game tree is of height at most n.

Now consider a game tree of height $n+1$. As argued in Theorem 1.10 we can view this game as starting at the root with each possible first move m_j leading to a game t_j of height at most n which is subsequently played out. We assume that at the root of each of these sub-games its value is given. There are three cases to consider.

Let us first assume that the first move is made by Player 1. Then Player 1 can ensure that his pay-off is the maximum

$$v = \max_{1 \le j \le k} v_j$$

of the values v_j of the sub-games reached after the first move. Player 2, on the other hand, can ensure that *her* pay-off is at least $-v$: No matter which first move Player 1 chooses, the worst case for Player 2 is that where she gets $-v$. Hence v is indeed the value of the overall game.

Let us now assume that the first move is made by Player 2. This case is almost the same as the one we have just discussed, the only difference being that values are given referring

to Player 1's pay-off. Hence Player 2 will be looking to maximize $-v$, which is equivalent to minimizing v, so she looks for the *least*

$$v = \min_{1 \leq j \leq k} v_j$$

of the values labelling the sub-games. The argument that this v is the value of the game is similar to the one before, only that the roles of Player 1 and Player 2 are reversed.

Finally we have to consider the case where the first move is a chance move. Then the highest pay-off Player 1 can hope for is the *expected pay-off*

$$\sum_{1 \leq j \leq k} q_j v_j,$$

which is calculated by taking the probability q_j that a particular move m_j occurs times the value v_j of the subsequent game, and summing those up over all possible first moves. But that is precisely the best pay-off that Player 2 can expect. □

Note that there is an algorithm hidden in this proof, and that this algorithm calculates a number v with the properties given in Proposition 2.11. We look at a slight adaptation of this in the form of the *minimax algorithm* which is described in Chapter 3. Note that this result does not assume that the game is in normal form, nor does its proof.

However, not every 2-person zero-sum game has an equilibrium point. Let us look at paper-stone-scissors whose pay-off matrix we repeat here.

$$\begin{vmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{vmatrix}$$

None of the entries is maximal in its column *and* minimal in its row at the same time. If we think about how the game is played this is not surprising: If there was an equilibrium point then there would have to be two strategies that are best responses to each other. But for each choice there is one that betters it, so this cannot exist. The following section looks at a more general approach.

2.5 Mixed strategies

Even when a game has no equilibrium point when playing it just one game theory can help us play such games better. The answer is to work out 'good play' if the game is repeated *many times*. This appears somewhat paradoxical: if we can't even say how to play the game well once, why should we be any better at trying to play it a number of times?

Mixing strategies

The answer is that that allows us to come up with something other than *a single strategy* in our answer to the question of how best to play the game. We can make statements such us 'when playing paper-stone-scissors one should aim to use all three strategies with equal frequency'. More formally what we do is assign a probability to each of the strategies we might play.

Definition 10. *For a game in normal form a* **mixed strategy** *for a player consists of a probability distribution for all the strategies available to that player.*

This means we have to assign a probability to each of the player's strategies so that these probabilities add up to one. If the player has strategies numbered $\{1, \ldots, m\}$ then we represent a mixed strategy by an m-tuple

$$(q_1, \ldots, q_m) \qquad \text{where}$$

- q_i is the probability that strategy i will be employed and
- all probabilities add up to 1, that is $q_1 + q_2 + \cdots + q_m = 1$.

So how do we play according to a mixed strategy? Before the game starts, we employ a device which will give us a number from 1 to m, such that the result will be number i with probability q_i. We then use the strategy thus decided upon. A mixed strategy literally allows us to mix our available strategies in any proportion we like.

Note that we have not lost the ability to just pick one strategy: If we always want to use strategy i we can do so by employing the mixed strategy $(0, \ldots, 0, 1, 0, \ldots, 0)$, where the 1 occurs at the ith entry. We sometimes abbreviate this strategy as i, as we did when we were only considering 'pure' strategies. In contrast with mixed strategies some people refer to our strategies as 'pure strategies'—they are mixed strategies that are made up purely of one strategy.

Consider a game like Paper-Stone-Scissors. Assume you are playing it many times against your best friend. Clearly, if you decided to *always* play Paper then your friend would soon catch on and start to always play Scissors, thus winning every single time. Hence employing a pure strategy in this game is not a very good idea. What would be a good answer instead? Intuitively it is clear that all strategies are equivalent, and that if we remove one of them from our considerations we give the other player an advantage. In fact, *everything* that makes it easier for him to predict what we are going to do next will give him an advantage. Hence the intuitive solution to this game is that all three strategies should be employed with equal probability, meaning that the mixed strategy $(1/3, 1/3, 1/3)$ should come out as best.

This first of all raises the question of how we measure the performance of a mixed strategy, but that is easily solved: We just use the *expected pay-off* when playing according to this strategy. This makes it easy to assign a number to the outcome of playing a mixed strategy against the pure strategy of an opponent: Assume that the opponent has n strategies, and that our strategy i playing against his strategy j gives a pay-off of $p_1(i, j)$. Then the mixed strategy $(q_1, \ldots, q_m)$ employed against strategy j will result in the expected pay-off

$$q_1 p_1(1, j) + q_2 p_1(2, j) + \cdots + q_m p_1(m, j).$$

Let us consider our mixed strategy $(q_1, \ldots, q_m)$ against his mixed strategy $(r_1, \ldots, r_n)$, which will result in the expected pay-off for Player 1 (which we again refer to as p_1):

$$\begin{aligned} p_1((q_1, \ldots, q_m), (r_1, \ldots, r_n)) = & \quad q_1 r_1 p_1(1,1) + q_1 r_2 p_1(1,2) + \cdots + q_1 r_n p_1(1,n) \\ + & \quad q_2 r_1 p_1(2,1) + q_2 r_2 p_1(2,2) + \cdots + q_2 r_n p_1(2,n) \\ + & \quad \cdots \\ + & \quad q_m r_1 p_1(m,1) + q_m r_2 p_1(m,2) + \cdots + q_m r_n p_1(m,n) \\ = & \quad \sum_{i=1}^{m} \sum_{j=1}^{n} q_i r_j p_1(i,j). \end{aligned}$$

We can visualize what happens by looking at the following table.

		2			
		r_1	r_2	$\ldots$	r_n
	q_1	$p_1(1,1)$	$p_1(1,2)$	$\ldots$	$p_1(1,n)$
1	q_2	$p_1(2,1)$	$p_1(2,2)$	$\ldots$	$p_1(2,n)$
		$\vdots$	$\vdots$		$\vdots$
	q_m	$p_1(m,1)$	$p_1(m,2)$	$\ldots$	$p_1(m,n)$

The probability that the pay-off $p(i, j)$ is reached is the probability q_i that Player 1 uses strategy i times the probability r_j that Player 2 uses strategy j. These are the probabilities marking the row/column of $p(i, j)$ respective. Hence what we have to do to calculate the expected pay-off is

- multiply each entry in the table above with its row and column probabilities and
- add up all these numbers.

If there are n players then we can still use the same principle.

So what happens is that we can derive pay-offs when using mixed strategies from the normal form of the game, and no additional information is required. If the game is given in extensive form the situation changes slightly.

Mixed strategies for games in extensive form

If we have a game given in extensive form then we have to rethink the notion of a mixed strategy. If we do not know what all our available strategies are then it makes no sense of wanting to mix them.

What we can do instead is *mix our choice at the various decision points.*

Definition 11. *For a game in extensive form a* **mixed strategy for Player** *i is given by a probability distribution over the available choices for each decision point belonging to i in such a way that all the decision points in the same information set have matching probabilities for all the available actions.*

For example in the game of simplified poker whose game tree is given on page 13, a mixed strategy for Player 1 consists of giving a probability for passing or betting for each choice of card he might get, as well as probabilities for these choices in the case where Player 1 has passed on the first move but Player 2 has bet. Note that these probabilities must not depend on Player 2's card. An example is given on page 47. Here probabilities arising from a chance move are given in the form 1/3, say, while those indicating Player 1's mixed strategy appear as $\frac{1}{3}$.

We can now calculate the probability that a particular outcome occurs when playing in accordance with that strategy by treating these probabilities in precisely the same way as we would treat the player Nature (sometimes called Chance), see page 10.

But how do such strategies relate to the mixed strategies in Definition 10?

If we have a mixed strategy $(q_1, q_2, \ldots, q_m)$ then we can turn it into a mixed strategy on the extensive game as follows:

- Pick a decision point for the player in question.
  - For each possible action at that point find all the strategies which choose that particular action.
  - Add up all the probabilities q_i for the strategies found in the previous step.
- We now have probabilities assigned to each action from our decision point but these may add up to less than 1. Scale the probabilities for all the actions by the same factor so that they add up to precisely 1.

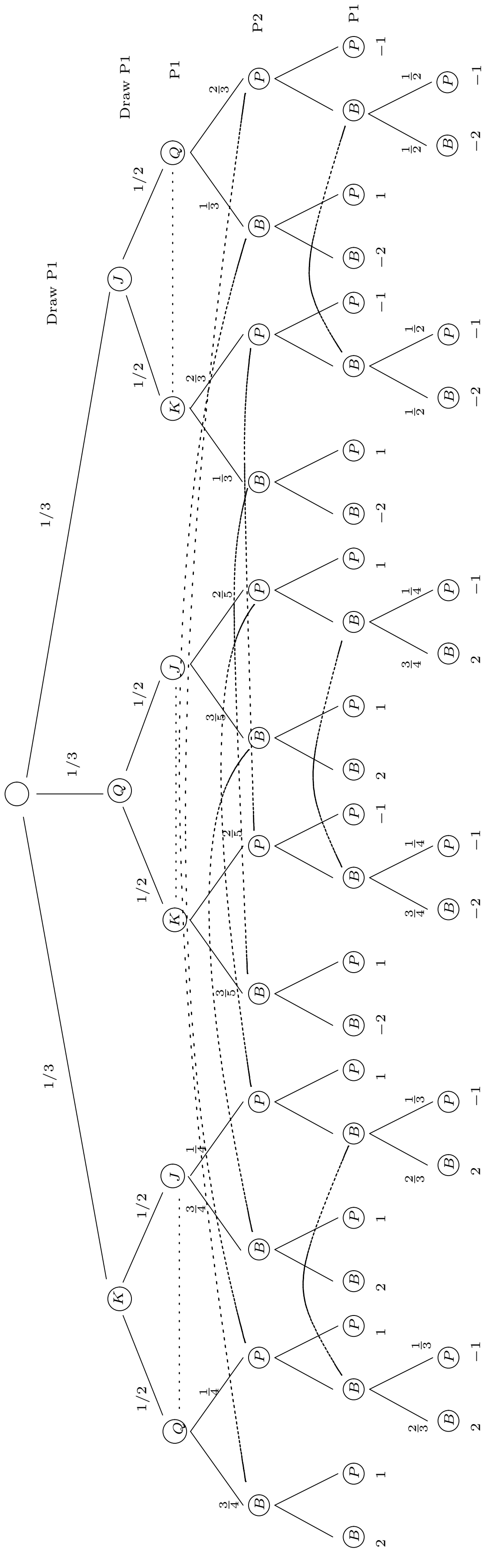
It is also not difficult to make a connection that goes in the other direction: Every mixed strategy for a game in extensive form corresponds to a mixture of the *fully specified* strategy for the same player.

Question 21. Why is this statement is true?

In other words fully specified strategies become a useful concept if we want to consider mixed strategies in the extensive form of a game. What is less obvious is the fact that actually every mixed strategy for the extensive form of a game corresponds to a mixture of strategies.[4]

[4] You may find statements to the contrary in the literature. This is true for games where it is allowed to have information sets containing nodes that have differing histories visible to the player who owns the information set. Game trees of this kind are sometimes allowed to model the players not having *perfect recall*, that is being unable to remember everything that happened in the course of a game.

In summary, any distribution of probabilities over all the outcomes that can be achieved with the one form of mixed strategy can be achieved with the other. As a consequence we can consider mixed strategies in either setting, that of games given in normal or extensive form.



Equilibria

What happens to our notion of equilibrium if we allow mixed strategies? The answer is, nothing—we can still use our original definition. In other words we still want to have choices of (mixed) strategies such that each strategy is a best response for the remaining strategies in the equilibrium point.

However we now have a problem: Even if there are only finitely many pure strategies in a game, as soon as one player has at least two strategies there are infinitely many mixed strategies for at least one of the players. How do we check whether we have an equilibrium point? Previously we only had to check against finitely many alternative strategies whether something was a best response. Fortunately we do not have to carry out infinitely many checks.

Verifying mixed strategy equilibria

It seems difficult at first sight to determine that a given tuple of mixed strategies is an equilibrium point for a game, but fortunately there is a proposition which tells us that we do not, in fact, have to match it against all the mixed strategies for all the players. Doing that with the pure ones suffices.

We establish this by first proving a lemma that turns out to be useful for a different purpose.

Lemma 2.13. *Given a tuple of mixed strategies $(s_1, \ldots, s_l)$ for some game, if Player i strategy s_i gives as least as high a pay-off as can be achieved by any of his pure strategies then the strategy s_i is a best response for the given strategies played by the other players.*

Proof. Let $s = (q_1, q_2, \ldots, q_{n_i})$ be a mixed strategy for Player i. We have to show that the pay-off to Player i when playing this mixed strategy is at most as high as that when playing the given strategy s_i. It is the case that

$$\begin{aligned} p_i(s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_l) &= \sum_{k=1}^{n_i} q_k p_i(s_1, \ldots, s_{i-1}, k, s_{i+1}, \ldots, s_l) \\ &\leq \sum_{k=1}^{n_i} q_k p_i(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_l) \\ &= p_i(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_l) \sum_{k=1}^{n_i} q_k \\ &= p_i(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_l). \end{aligned}$$

as required. □

Proposition 2.14. *A tuple of mixed strategies $(s_1, \ldots, s_l)$ is an equilibrium point for a game if and only if for every player i it is the case that i's pay-off at the the given tuple is as least as high as i can achieve by using any of his pure strategies against the other given strategies.*

Proof. Clearly this is a necessary condition for being an equilibrium point since strategy s_i is a best response to the given strategies for the other players it must give at least as good a pay-off as any of i's pure strategies.

It is also sufficient by the previous lemma. □

So we can *check* whether or not a given mixed strategy tuple gives an equilibrium point by performing finitely many calculations: We just have to check what happens for every pure strategy for every player.

We return to our example of Paper-Stone-Scissors. We have claimed that the mixed strategies $(1/3, 1/3, 1/3)$ and $(1/3, 1/3, 1/3)$[5] together define an equilibrium point. We first

[5]Since the game is entirely symmetric it should come as no surprise that the solution is symmetric as well.

calculate the expected pay-off which is

$$\begin{aligned} p_1((1/3,1/3,1/3),(1/3,1/3,1/3)) = & \\ & (1/9\times 0+1/9\times 1+1/9\times(-1))+(1/9\times(-1)+1/9\times 0+1/9\times 1) \\ & +(1/9\times 1+1/9\times(-1)+1/9\times 0) \\ = \;& 3\times(1/9\times 1)+3\times(1/9\times(-1)) \\ = \;& 0. \end{aligned}$$

We then compare this mixed strategy to each of the pure strategies for Player 1.

$$p_1(1,(1/3,1/3,1/3)) \;=\; 0\times 1/3+1\times 1/3-1\times 1/3=0.$$

Because all the strategies are equivalent, $p_1(2,(1/3,1/3,1/3))$ and $p_1(3,(1/3,1/3,1/3))$ evaluate to the same value. For symmetry reasons this argument also applies to comparing this mixed strategy to Player 2's pure strategies. By Proposition 2.14, we have indeed found an equilibrium point.

B Exercise 21. (a) Show that the game with the pay-off matrix given below has the mixed strategy equilibrium $((1/2,0,0,1/2),(1/4,1/4,1/2))$.

$$\begin{vmatrix} -3 & -3 & 2 \\ -1 & 3 & -2 \\ 3 & -1 & -2 \\ 2 & 2 & -3 \end{vmatrix}$$

(b) Consider the following game. Alice has an Ace and a Queen, while Bob has a King and a Joker. It is assumed that the Ace beats the King which beats the Queen, whereas the Joker is somewhat special. Both players pay an ante of one pound into the pot. Then they select a card, each from his or her hand, which they reveal simultaneously. If Bob selects the King then the highest card chosen wins the pot and the game ends. If Bob chooses the Joker and Alice the Queen they split the pot and the game ends. If Bob chooses the Joker and Alice the Ace then Alice may either resign (so that Bob gets the pot) or demand a replay. If a replay occurs they each pay another pound into the pot and they play again, only this time Alice does not get the chance to demand a replay (so Bob gets the pot if he chooses the Joker and Alice the Ace).

Draw a game tree for this game and then bring it into matrix form. If you have your strategies in the same order as I do then you can show that an equilibrium is given by Alice's mixed strategy $(0,1/8,1/4,5/8)$ and Bob's mixed strategy $(1/4,1/4,1/2)$. To get your strategies in the same order I suggest

- list Alice choosing the Ace before choosing the Queen and consider her ending the game before demanding a reply and
- Bob choosing the King before the Joker.

Who has the advantage in the game? *Strictly speaking, the requirement to order the strategies correctly to prove the equilibrium point is more than I'd expect in an exam, but the individual calculations for game tree, normal form, and checking equilibria I consider basic.*

Properties of mixed strategy equilibria

The introduction of mixed strategies solves the problem of non-existence of equilibrium points.

Theorem 2.15 (Nash)**.** *Every game with finitely many pure strategies for each player has at least one mixed strategy equilibrium point.*

Proof. This proof requires too much mathematical background to be appropriate for this course. □

This is quite surprising in many ways! Even if we're not sure what to make of equilibria as solutions to games which aren't 2-person zero-sum at least we know that such a 'solution' always exists.

Section 2.4 establishes a number of properties of equilibria for 2-person zero-sum games. These extend to mixed strategies where this makes sense.

Corollary 2.9 remains true, even if we allow mixed strategies to be employed. Hence the notion of the *value* of such a game still makes sense. However the proof we used previously only takes pure strategies into account, so we restate the result and prove it in its full generality.

Proposition 2.16. *For a 2-person zero-sum game all equilibrium points, whether they consist of pure or mixed strategies, lead to the same pay-off, which we call the* value *of the game.*

Proof. One possible proof of this result is similar to that of Proposition 2.8, just using the definition of the pay-off function for mixed strategies. There is an alternative proof which we present here. Let (s,t) and (s',t') be equilibrium points for the game under consideration. (If there are any pure strategies involved we still think of them as represented as mixed strategies, with a probability of 0 being given to all but the chosen strategy.) These lead to pay-offs $p(s,t)$ and $p(s',t')$ respectively for Player 1. By the definition of an equilibrium point, we know that if one player changes away from an equilibrium point, his or her pay-off can only decrease. Hence

$$p(s,t) \geq p(s',t) \geq p(s',t') \geq p(s,t') \geq p(s,t).$$

The first inequality is Player 1 changing away from (s,t), the second is Player 2 changing away from (s',t'), the third Player 1 changing away from (s',t'), and the last Player 2 changing away from (s,t). But the above chain of inequalities means that all these numbers have to be equal, in particular $p(s,t) = p(s',t')$. □

Propositions 2.11 and 2.10 also remain true. The former tells us that the value of the game still maximizes their pay-off in the worst case. The latter tells us that we can pair up equilibrium strategies for the players in any way we like, and still gain an equilibrium point. Hence the notion of 'optimal' strategies for both players still makes sense, although we need to remember when using that term that these strategies are only optimal under the assumption that the other player is also playing optimally!

There is also a version of Proposition 2.8 that is true, which really means reformulating Proposition 2.11 in terms of pay-off functions. It is the case that

$$\max_s \min_t p_1(s,t) = \min_t \max_s p_1(s,t)$$

where s and t vary over all mixed strategies for Player 1 and 2 respectively.

It is also still the case that any equilibrium strategy for Player 1 paired with any equilibrium strategy for Player 2 gives an equilibrium point of the game—in fact, any mixed strategy which assigns non-zero probabilities only to pure equilibrium strategies is another equilibrium strategy.

There is quite a bit more that can be said about the nature of mixed strategy equilibria and the strategies that appear in it with non-zero probabilities. For example, in a 2-person zero-sum game any mixture of pure equilibrium strategies for either player is another equilibrium strategy for that player. More generally, if $((q_1,\ldots,q_m),(r_1,\ldots,r_n)$ is an equilibrium point of a 2-person game then if $q_i \neq 0$ it is the case that pure strategy i is a best response for Player 1 against $(r_1,\ldots,r_n)$. Some of these considerations are useful when one generally wants to tackle the problem of computing equilibria, but a detailed study is beyond the scope of this course.

However even for 2-person zero-sum games there is one drawback with considering only equilibria when deciding how to play a game: Equilibria work well if both players know

enough to play their optimal strategies. If one player is not capable of doing this then there are usually opportunities for the other player to exploit that weakness. For example, one optimal strategy might be better at this than another—or, alternatively, there might be a non-optimal strategy that is the best response to the other player's behaviour.

2.6 Finding equilibria

Having identified a notion of 'solution' to a game in the form of equilibria the question arises how one can find these. This is not at all easy.

One issue we have to decide on is the following:

- Do we only want to find one equilibrium?
- Do we want to find *all* equilibria?

Depending on which task we seek to carry out we have to employ quite different strategies. For 2-person zero-sum games finding one equilibrium point is sufficient to tell us how to play the game: If we employ any one of our optimal strategies we'll guarantee the best pay-off for ourselves, provided the other player does so too.

For non zero-sum games pay-offs at equilibria can vary vastly. There finding one equilibrium is unlikely to lead to good play.

Reducing the size—dominance

In Chapter 1 it is stressed that one should try to keep the description of the game as small as possible. There we looked at issues of symmetry that allowed us to reduce the size of the game tree. If we are only interested in finding *one* equilibrium point then we can take this idea lot further.

Definition 12. *We say that* **a strategy s for Player i is dominated by a strategy s' for the same player** *if*

- *for all choices of strategies by the other players it is the case that*
  - *the pay-off for Player i when playing strategy s is less then or equal to that when playing strategy s'.*

It is very important to note the universal ('for all') quantification in this definition.

What this definition tells us is that in *all situations* the strategy s is outperformed by the strategy s' (or at least the strategy s' will give the player at least as good a pay-off). We conclude that there is no reason for the player ever to use strategy s because strategy s' is preferable. In other words we can remove dominated strategies from consideration.

By Lemma 2.13 to check that a strategy is dominated it is sufficient to do so against all the *pure strategy choices* for the other players.

Here is an example. Consider the 2-person zero-sum game given by the following matrix.

$$\begin{vmatrix} 1 & -1 & 2 \\ -1 & 1 & 3 \\ -3 & -2 & 4 \end{vmatrix}$$

Player 2 wishes to minimize her pay-out to Player 1, and from her point of view her strategy 3 is particularly undesirable: If she compares it point by point with her strategy 1 she notices that if Player 1 chooses his strategy 1 then she will only have to pay 1 if she chooses her strategy 1 as opposed to 2 if she selects her strategy 3. If Player 1 goes for his strategy 2, the values become a win of 1 *versus* a pay-out of 3, and for Player 1's strategy 3 a win of 3 compared to a pay-out of 4. In other words, Player 2's strategy 3 is dominated by her strategy 1.

If we remove strategy 3 for Player 2 from consideration we obtain the following game matrix.

$$\begin{vmatrix} 1 & -1 \\ -1 & 1 \\ -3 & -2 \end{vmatrix}$$

Now it is the case that Player 1's strategy 3 is dominated by both, strategies 1 and 2, so out it goes, leaving us with the following table.

$$\begin{vmatrix} 1 & -1 \\ -1 & 1 \end{vmatrix}$$

Note that we could not remove Player 1's strategy until we had removed a strategy for Player 2.

We now have a symmetric matrix game with a particularly simple solution: the mixed strategies $(1/2, 1/2)$, $(1/2, 1/2)$ define an equilibrium point. If we wish to formulate this solution in terms of the original game, all we have to do is to turn those into mixed strategies $(1/2, 1/2, 0)$ and $(1/2, 1/2, 0)$.

In the prisoners' dilemma game Example 2.4 the 'talk' strategy dominates the 'don't talk' one: No matter what the other player does, the 'talk' strategy always gives a better pay-off than the 'don't talk' one does in the same situation. Hence we can reduce that game by removing dominated strategies to a (1×1)-matrix giving the pay-off at the equilibrium point (talk, talk).

B Exercise 22. Reduce the games given by the matrices below via dominance consideration as far as possible.

(a)
$$\begin{vmatrix} 2 & 4 & 0 & -2 \\ 4 & 8 & 2 & 6 \\ -2 & 0 & 4 & 2 \\ -4 & -2 & -2 & 0 \end{vmatrix}$$

(b)
$$\begin{vmatrix} 2 & -3 & 1 & -4 \\ 6 & -4 & 1 & -5 \\ 4 & 3 & 3 & 2 \\ 2 & -3 & 2 & -4 \end{vmatrix}$$

Note that in the prisoner's dilemma game Example 2.4 the 'talk' strategy dominates the 'don't talk' strategy for both players, so removing those gives us the equilibrium point where both talk.

However, often we cannot get very far with removing strategies in this way even for 2-person zero-sum games. Note that our definition of dominance does *not* refer to pure strategies—it also applies to mixed strategies. This gives us a more complicated method that allows us to find more dominated strategies.

Consider the following game as an example.

$$\begin{vmatrix} -1 & 2 \\ 2 & -1 \\ 0 & 0 \end{vmatrix}$$

No (pure) strategy for Player 1 is dominated by any other. Similarly for Player 2's two strategies.

Again we're faced with the problem of finding out whether we can remove a strategy for either player by using the idea of dominance. Who should we start with, Player 1 or Player 2? As a rule of thumb, if one player has more strategies than the other, he or she is a good target, and so we will start with Player 1.

So which of Player 1's three strategies might be dominated by a mix of the other two?

Strategy 1 has a potential pay-off of 2 which is maximal in its column, and so cannot be outperformed by the remaining entries. Similarly, strategy 2 has the unique highest value in the first column. That leaves the only feasible target of strategy 3.

The question becomes: can we find a probability λ such that strategy 3 is dominated by the mixed strategy

$$(\lambda, 1 - \lambda, 0)?$$

In order to find such a λ we have to calculate the pay-offs for playing that mixed strategy *versus* playing the pure strategy 3 for each of the choices that Player 2 can make. We need a λ with the property that the following inequalities are true simultaneously.

$$\begin{array}{ll} \text{Player 2 plays Strategy 1} & 0 \leq -\lambda + 2(1-\lambda) = 2 - 3\lambda \\ \text{Player 2 plays Strategy 2} & 0 \leq 2\lambda - (1-\lambda) \quad = 3\lambda - 1 \end{array}$$

The former is equivalent to

$$\lambda \leq \frac{2}{3}$$

and the latter to

$$\lambda \geq \frac{1}{3},$$

so $\lambda = 1/3$ will do the job. Hence Player 1's strategy 3 is dominated by his mixed strategy $(1/3, 2/3, 0)$.

It is also possible to remove dominated strategies from consideration even if one hasn't calculated the matrix of the game—instead one can remove strategies from consideration looking at the extensive form of the game. One then produces an already reduced matrix for the game. A concrete example of this technique appears in Section 2.7.

Making the game smaller in this way is only safe if we know that a solution to the reduced game can be turned into a solution of the original game.

Proposition 2.17. *Let G' be a game that results from the game G by removing a pure strategy j for Player i which is dominated by some other strategy. Any equilibrium point for the game G' can be turned into an equilibrium point for the game G by putting the probability that Player i plays strategy j to be* 0.

The proof of this result is lengthy but not too difficult, and we omit it here.

Hence any equilibrium we find for the smaller game 'is' an equilibrium for the original one. In some games one can get a fairly long way by reducing the size:

- Sometimes one can reduce the game to a (1×1)-matrix, that is, there is only one strategy left for each player. An example for this is the prisoners' dilemma game. In that case applying the above considerations leads directly to the solution in the form of an equilibrium point. However, this may not be the only equilibrium point.
- Sometimes one can reduce the game to a (2×2)-matrix. If the game cannot be reduced further there is an equilibrium point which is a mixture of two strategies by each player. In the case of a 2-person zero-sum game this equilibrium can be calculated by solving two systems of two linear equations: It uses the fact that we know that in the equilibrium point it must be the case that

  $$\min_t \max_s p_1(s,t) = \max_s \min_t p_1(s,t)$$

  where s ranges over all strategies for Player 1 and t over those for Player 2.

In all other cases reducing the size is merely a preparatory step before applying methods outlined below. Note that if we remove strategies in this way we may remove equilibria. There is an adjusted technique that removes fewer strategies but which keeps all equilibria which we do not discuss in detail.

Calculating equilibria

For most cases calculating equilibria is a hard, and for some cases there is no known algorithm. We here give a summary of what is known. The following statements concern *games in normal form*. Games in extensive form have only recently attracted more attention. We study ways of approaching these in Chapters 3 and 4.

2-person zero-sum. In the case of such games there is an algorithm requiring polynomial time that finds all equilibria. The problem of finding equilibria in this case can be reformulated so that it belongs to a family known as 'linear programming problems'. At least three different algorithms are known, and how good they are depends on the specific problem in question. There are nonetheless still open problems in this area regarding the precise question of how hard a problem this is to solve.

2-person general sum. There is a known algorithm, the *Lemke-Howson* algorithm, for finding one equilibrium point of a given game. In the worst case it runs in exponential time. This algorithm belongs to a complexity class between 'P' (can be solved in polynomial time) and 'NP'. For those doing the algorithms course it might be interesting to know that there are a number of NP-complete problems related with finding equilibria in these games. We only give some of those.

- Are there at least two equilibria?
- Is there an equilibrium where Player 1's pay-off is at least some given number?
- Is there an equilibrium where the combined pay-off of the two players is at least some given number?

n-person games. This case is even more difficult than the previous one and not much is known here at all. There is no known algorithm that works for all such games. This is particularly problematic since many of the applications are for multi-agent games (for example routing problems in a network).

Since the case of finding even one equilibrium point is quite hard in general sum games people have come up with alternatives to equilibria that are easier to calculate.

- There is a notion of an *approximate equilibrium* which guarantees that the pay-off for all players is at most a given (small) number ε below the pay-off at an actual equilibrium point.
- There is a notion of a *correlated equilibrium* where players are allowed to correlate their choices with each other.

There's quite a lot of literature regarding these notions and it is beyond the scope of these notes to cover it in more detail.

Learning equilibria

If it is difficult to calculate equilibria can one set up systems of agents that will learn these? This is a sensible question, and the issue of learning in games will be considered in detail in Semester 2.

Here we only want to stress some of the issues that have to be decided when setting up a learning situation. Note that in this case there has been quite a bit of work on games in extensive form: It is certainly quite intuitive to consider agents that are learning about a game as it is played in the usual manner.

- Is there only one learning agent in an environment where the other agents are static? In this case we are not really treating the full problem.
- If the other agents in the system also vary, do they apply a learning algorithm? Is it the same one as that employed by 'our' agent?

- What exactly is our agent expected to learn? Is it to find a best response to the other agents? How are we going to define that if the other agents employ variable strategies?
- What information does our agent get? Does it find out the pay-off for the other agents? What does it learn about moves made by the other agents? (For example in games like poker, does it find out the cards the other player had?)
- What should we model explicitly in our agent (the other agents or just the rules of the game as they apply to our agent)?
- Even if we have a system of agents all employing the same learning algorithm can we expect them to move towards an equilibrium point (that is a situation where they play best responses to one another)? Will they reach it?

This is an area with a lot of on-going research, some of which is carried out in this department. For the moment we just want to point out that there are a lot of decisions to be made when it comes to addressing this issue, and for a few of those we do not have all the answers just yet. This is even true for the 2-person zero-sum case.

2.7 Extended example: simplified poker

This section is for the most part taken from A.J. Jones book on game theory (see the list of sources), and she attributes the study to H.W. Kuhn.[6]

Poker as it is actually played (it does not particularly matter which variant we are talking about) is much too big for a detailed analysis. We radically simplify it by fixing the number of players at 2, and by assuming there are only 3 cards (which each player being dealt a hand of 1). While this would not be a very interesting game to play in practice, it turns out that many issues of playing 'real' poker appear when analysing this simple game, for example, that of bluffing, or folding with a hand which appears quite good. The game is as described in Example 3. We repeat the rules here for convenience.

Each of the two players has to pay one unit to enter a game (the *ante*). They then are dealt a hand of one card each from a deck containing three cards, labelled J, Q and K. The players then have the choice between either betting one unit or passing. The game ends when

- either a player passes after the other has bet, in which case the better takes the money on the table (the *pot*),
- or there are two successive passes or bets, in which case the player with the higher card (K beats Q beats J) wins the pot.

The full game tree appears on page 13, but it is rather large and unwieldy. We can also visualize what happens in the following way.

There are six possible deals (Player 1 might get one of three cards, and Player 2 will get one of the two remaining cards, making $3 \times 2 = 6$ possibilities). For each of these deals, the subsequent game tree is given in Figure 2.4, and we briefly explain the notation used there. The players' options in each case are to bet (B) or to pass (P). The result is expressed by giving the winner (1 is Player 1, 2 is Player 2 and H is the holder of the higher card) and the amount he or she will win (where we have deducted the player's own contribution to the pay-off), that is 1:1 means that Player 1 wins 1 unit.

If we consider the available strategies for Player 1 we find that he has to make a decision whether to pass or to bet, depending on his card, for round 1 and a potential round 2. We record his strategies in tables, where for example

J	Q	K
PB	PP	B

[6]The full article is: H.W. Kuhn, **A simplified two-person poker.** In: *Contributions to the Theory of Games*, I, Ann. Math. Studies No. 24, 105–116, 1950.

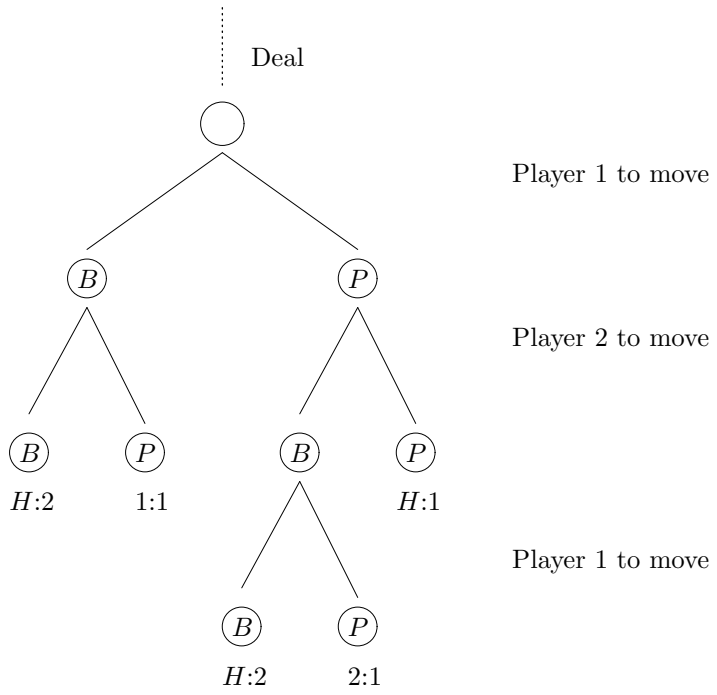


Figure 2.4: Simplified poker

means that if Player 1 has the Jack (J) he will pass in the first round and bet in the second (if they get that far), if he has the Queen (Q) he will always pass, and if he has the King (K), he will bet in the first round. But we do not really need a table to gather all the information: After all, the order J, Q, K is constant. Hence we can use a triple (PB, PP, B) to denote the same information. Since Player 1 has three choices, PP, PB, and B, for each of the three cards he might possibly be dealt, he has $3 \times 3 \times 3 = 3^3 = 27$ (pure) strategies.

Now for Player 2. Her strategies may depend on her own card (one of three) *as well as Player 1's move in the first round* (one of two). For each of her potential cards, there are four strategies (giving an answer for the two first moves Player 1 might have made), giving 4^3 strategies altogether. Again we encode these as triples which tell her how to make a choice based on her card. However, her potential moves depend on Player 1's first move as well, and therefore we need a different encoding. We use the four options $P|P$, $P|B$, $B|P$, and $B|B$, where the first component says what to do if Player 1 bets, and the second what to do if he passes. So $P|B$, for example means to pass if Player 1 bets and to bet if Player 1 passes.

We wish to solve this game, but at the moment it is still somewhat large. Hence we try to make it smaller by removing from consideration those strategies which are dominated by others. Clearly, when confronted with a bet, a player should bet if he or she has the K, and pass if he or she has the J: the former guarantees a win, and the latter will result in a loss of 1 rather than 2 units.

Let us go through this argument in more detail. We start with Player 1. If Player 1 has the J, and the course of play so far has been P, B then if Player 1 decides to bet, he will lose 2 units rather than just the 1 he will lose if he passes in that situation. Hence we remove all strategies which have PB in their first component (which describes what to do if one's card is the J). If Player 1 has the K, on the other hand, then if the course of play so far has been P, B he should definitely bet, guaranteeing a win of 2 rather than just 1 unit. Therefore we remove all strategies which have PP in their last component (which describes what to do if one's card is the K). That leaves us with $2 \times 3 \times 2 = 12$ strategies for Player 1. Note that these considerations do *not* say that Player 1 should be on the first move if has the K!)

Now we apply similar considerations to Player 2. If she is faced with a bet, that is the course of play so far has been B, and she holds the J, then she will lose 1 unit if she passes, and 2 if she bets, so passing is the better option. Therefore we remove all strategies that have

$B|P$ or $B|B$ in their first component. If Player 2 has the K, on the other hand, she should *never* pass, that would only decrease her potential winnings. When faced with a course of play so far of P, then if she passes too she will win 1 unit, whereas she might win 2 if she were to bet. When faced with a course of play so far of B then by betting in return she will win 2 rather than lose 1. Hence we insist that all her strategies have $B|B$ as their last component. That leaves $2 \times 4 \times 1 = 8$ strategies for her.

Now that we have removed some strategies from consideration we can use this information to get rid of more: We may use the fact that the strategies we have removed already are out of consideration. If Player 1 holds the Q, he should pass in the first round and only bet if Player 2 bets in her first go: If Player 2 has the K she will definitely bet no matter what (as we have just ruled out all other strategies). Hence Player 1 will lose 2 if he bets in the first round, and the same amount if he instead first passes and then bets in the second round. If Player 2 has the J, on the other hand, then she will pass when confronted with a bet (we have ruled out all her other strategies in the previous paragraph). This gives the following possibilities when comparing the strategies B and PB under the assumption that Player 1 has the Q.

P1 has the Q					
P1's strategy	B		PB		
P1's first move	B		P		
P2's card	J	K	J		K
P2's move	P	B	B	P	B
P1's snd move			B		B
pay-off for P1	1	-2	2	1	-2

If we compare the possible outcomes of playing B *versus* playing PB when Player 1 holds the Q, we find that he can only improve his situation when choosing PB. The only difference occurs when Player 2 holds the J: she might be tempted into betting when facing a pass (in which case there is a chance of Player 1 winning 2 units rather than 1) but she certainly won't do so when faced with a bet. Hence we remove all the strategies for Player 1 which have a B in the second component. That leaves $2 \times 2 \times 2 = 8$ strategies for Player 1.

Finally we look at the case of Player 2 holding the Q and facing a pass. It must be the case that Player 1 holds either the J or the K, and we only have to consider those components of Player 1's strategies. This leaves us with four strategies for Player 1 (combining PP or B for the first component with PB or B for the last component), and the four possibilities for the middle component of Player 2's strategies: $P|P$, $B|P$, $P|B$, and $B|B$.

To fill in the resulting table we have to work out how a strategy for Player 2 with the given middle component would do against a strategy for Player 1 with the given first and third component. Since either case occurs with the same probability and we are only interested in dominance we do not have to take probability factors into account.

So if a strategy for Player 1 of the form $(PP, , PB)$ faces a strategy for Player 2 of the form $(, P|P,)$ then, under the assumption that she has the Q and he has either the J or the K, there are two cases which are equally likely.

Player 1 has the J: The course of play will be P, P and the pay-off will be -1 for Player 1.

Player 1 has the K: The course of play will be P, P and the pay-off will be 1 for Player 1.

Hence we can summarize this case by saying that the expected pay-off will be 0. Treating all the other cases in the same way we obtain the following full table.

	Second component			
	$P\|P$	$B\|P$	$P\|B$	$B\|B$
$(PP, , PB)$	0	0	1	1
$(PP, , B)$	0	1	0	1
$(B, , PB)$	2	-1	3	0
$(B, , B)$	2	0	2	0

We notice that strategy 3 for Player 2 is dominated by strategy 1 and that strategy 4 for the same player is dominated by strategy 2. This means that having $P|B$ or $B|B$ in the second component is something that Player 2 should avoid.[7]

We can argue the same case without looking at a matrix. Assume Player 2 holds the Q and play so far consists of a pass P by Player 1. If she bets then if Player 1 holds the J he will pass (because we have excluded his other strategies) and she will win 1, and if he holds the K he will bet (again because we have excluded his other strategies) and she will lose 2. If, on the other hand, she passes then she will once again win 1 if Player 1 holds the J, but only lose 1 if he holds the K.

Hence we we strike all her strategies which have $P|B$ or $B|B$ in the second component. That leaves $2 \times 2 \times 1 = 4$ strategies for Player 2 ($P|P$ or $B|P$ in the first component, the same for the second, and $B|B$ in the third).

Discarding all these strategies makes it feasible to look at the full matrix of the remaining strategies for this game. To calculate the pay-off function for playing, say, (PP, PP, PB) against $(P|P, P|P, B|B)$, we have to calculate the expected pay-off. For this we must consider all possible deals. We give these as pairs, with the first component the card of Player 1, and the second component that of Player 2. Each of these deals occurs with the probability 1/6.

$$(PP, PP, PB) \qquad \textit{versus} \qquad (P|P, P|P, B|B) :$$

$(\mathbf{J}, \mathbf{Q})$: The moves played are P, P and Player 1 gets -1.

$(\mathbf{J}, \mathbf{K})$: The moves played are P, B, P and Player 1 gets -1.

$(\mathbf{Q}, \mathbf{J})$: The moves played are P, P and Player 1 gets 1.

$(\mathbf{Q}, \mathbf{K})$: The moves played are P, B, P and Player 1 gets -1.

$(\mathbf{K}, \mathbf{J})$: The moves played are P, P and Player 1 gets 1.

$(\mathbf{K}, \mathbf{Q})$: The moves played are P, P and Player 1 gets 1.

Hence the expected pay-off when playing (PP, PP, PB) against $(P|P, P|P, B|B)$ is

$$(1/6 \times (-1)) + (1/6 \times (-1)) + (1/6 \times 1) + (1/6 \times (-1)) + (1/6 \times 1) + (1/6 \times 1) = 0.$$

We give the full matrix, but to make it easier to compare the entries, we multiply all of them by 6. So the true game matrix is 1/6 times the one given below.

	$(P\|P,$ $P\|P,$ $B\|B)$	$(P\|P,$ $B\|P,$ $B\|B)$	$(P\|B,$ $P\|P,$ $B\|B)$	$(P\|B,$ $B\|P,$ $B\|B)$
(PP, PP, PB)	0	0	-1	-1
(PP, PP, B)	0	1	-2	-1
(PP, PB, PB)	-1	-1	1	1
(PP, PB, B)	-1	0	0	1
(B, PP, PB)	1	-2	0	-3
(B, PP, B)	1	-1	-1	-3
(B, PB, PB)	0	-3	2	-1
(B, PB, B)	0	-2	1	-1

One can easily verify that the following 12 mixed strategies are equilibrium point strategies for Player 1 for this matrix (and thus for simplified poker):

[7] It should be pointed out that we could argue as we did only because all strategies consist of independent components.

$$
\begin{array}{rlcrlcrl}
2/3 & (PP,PP,PB) & + & 1/3 & (PP,PB,PB) & & & \\
1/3 & (PP,PP,PB) & + & 1/2 & (PP,PB,B) & + & 1/6 & (B,PP,PB) \\
5/9 & (PP,PP,PB) & + & 1/3 & (PP,PB,B) & + & 1/9 & (B,PB,PB) \\
1/2 & (PP,PP,PB) & + & 1/3 & (PP,PB,B) & + & 1/6 & (B,PB,B) \\
2/5 & (PP,PP,B) & + & 7/15 & (PP,PB,PB) & + & 2/15 & (B,PP,PB) \\
1/3 & (PP,PP,B) & + & 1/2 & (PP,PB,PB) & + & 1/6 & (B,PP,B) \\
1/2 & (PP,PP,B) & + & 1/3 & (PP,PB,PB) & + & 1/6 & (B,PB,PB) \\
4/9 & (PP,PP,B) & + & 1/3 & (PP,PB,PB) & + & 2/9 & (B,PB,B) \\
1/6 & (PP,PP,B) & + & 7/12 & (PP,PB,B) & + & 1/4 & (B,PP,PB) \\
5/12 & (PP,PP,B) & + & 1/3 & (PP,PB,B) & + & 1/4 & (B,PB,PB) \\
1/3 & (PP,PP,B) & + & 1/3 & (PP,PB,B) & + & 1/3 & (B,PB,PB) \\
2/3 & (PP,PB,B) & + & 1/3 & (B,PP,B) & & &
\end{array}
$$

These are also *all* the equilibrium point strategies for that player. Player 2 has far fewer such strategies, namely just the following two

$$
\begin{array}{rlcrlcrl}
1/3 & (P|P,P|P,B|B) & + & 1/3 & (P|P,B|P,B|B) & + & 1/3 & (P|B,P|P,B|B) \\
2/3 & (P|P,P|P,B|B) & + & 1/3 & (P|B,B|P,B|B). & & &
\end{array}
$$

There are ways of categorizing these strategies using *behavioural parameters*, but we will not look at those here. The interested reader is encouraged to look up the treatment in Jones's version or to go back to the original article.

It is worth pointing out that two practices employed by experienced poker players play a role in Player 1's arsenal of optimal strategies, namely *bluffing* and *underbidding*.

Bluffing means betting with a J (a card which is bound to lose if a comparison is forced) and underbidding refers to passing (at least initially) with a K (a card that is bound to win under any circumstances). Almost all Player 1's equilibrium strategies involve *both* those. Similarly, all such strategies for Player 2's involve bluffing. Bluffing is not really an option for her, due to the specific rules of our game.[8]

When looking at ordinary poker it is commonly observed that the average player will

- not bluff often enough—people generally have the feeling that they should be able to win a 'show-down' before betting on their cards;
- not underbid often enough. This case is slightly more complicated. In ordinary poker (as opposed to our 'baby' version here), underbidding is usually more varied. When people have a reasonably good hand, they will almost inevitably bet fairly highly on it, assuming erroneously that their having an above-average hand must mean that nobody else can have one (let alone one which beats theirs). While our example is too simple to show this effect, we get at least its shadow.

The value of simplified poker is $-1/18$, so Player 1 will lose on average. In this simplified version of the game having to take the initiative is a disadvantage. In other such variants the opposite is true. The lesson of this section is that game theory can indeed help us to improve our game at the kinds of games people play as a pastime. If the actual game is too large to analyse, looking at simplified versions can produce useful guidelines.

A Exercise 23. Alice and Bob play a different form of simplified poker. There are three cards, J, Q and K, which are ranked as in the above example. Each player puts an ante of one pound into the pot, and Alice is then dealt a card face down. She looks at it and announces 'high' or 'low'. To go 'high' costs her 2 pounds paid into the pot, to go 'low' just 1. Next Bob is dealt one of the remaining cards face down. He looks at it and then has the option to 'fold' or 'see'. If he folds the pot goes to Alice. If he wants to see he first has to match Alice's bet. If Alice bet 'high' the pot goes to the holder of the higher, if she bet 'low' it goes to the holder of the lower card.

Draw the game tree for this game, indicating the information sets. Convince yourself that

[8] If one assumes that both, the ante as well as the amount players can place on a bet, are real numbers, then one can play with these parameters and see their effect on the equilibrium strategies. Some of these variants rule out bluffing. Again, see Jones for a discussion of this.

Alice has 8 (pure) strategies and that Bob has 64. Discard as many of these strategies you can by arguing that there are better alternatives. You should be able to get the game down to 2 strategies for Alice and 4 for Bob. Find the matrix for the reduced game and solve it.[9]

Summary of Chapter 2

- The solution to a game is often taken to be given by *equilibrium points*.
- For zero-sum games equilibrium points make sense, but for other games it is not obvious that these can be considered solutions.
- In order to guarantee the existence of an equilibrium point we may have to switch to *mixed strategies*, meaning that pure strategies are chosen with certain probabilities. We can also define mixed strategies for games in extensive form.
- Equilibria have the property that if any one player moves away from them unilaterally, his pay-off can only get worse.
- Every (non-cooperative) game has at least one equilibrium point (of mixed strategies).
- In 2-person zero-sum games, all equilibrium points lead to the same pay-off, the *value of the game*. The value is the least pay-off that Player 1 can guarantee for himself, while Player 2 can ensure that it is the highest amount she may have to pay to Player 1. In such a game, it makes sense to talk of equilibrium strategies as *optimal* for the respective player. If the game is one of perfect information, then an equilibrium point consisting of pure strategies exist.
- In practice, we can make a game matrix smaller by removing *dominated strategy*. However, finding equilibria is a hard problem in general. Only in the 2-person zero-sum case do we have a polynomial-time algorithm.

Sources for this section

Antonia J. Jones. **Game Theory: Mathematical models of conflict.** *Horwood Publishing*, Chichester, 2000.

Melvin Dresher. **Games of Strategy: Theory and Applications.** *Prentice-Hall International, Inc.* 1961.

L. C. Thomas. **Games, Theory and Applications.** *Dover Publications, Inc.* 2003.

A. Dixit and S. Skeath. **Games of Strategy.** *W. W. Norton & Company*, 2004.

Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. **Algorithmic Game Theory.** *Cambridge University Press*, 2007.

J. D. Williams. **The Compleat Strategyst.** *McGraw-Hill Book Company, Inc.* 1954.

J. F. Nash. **Non-cooperative games.** In: *Annals of Math.*, 54, 286–295, 1951.

J. von Neumann and O. Morgenstern. **Theory of Games and Economic Behaviour.** *Princeton University Press*, 1947.

M.J. Osborne and A. Rubinstein. **A Course in Game Theory.** *MIT Press*, 1994.

D.R. Hofstadter, *Dilemmas for Superrational Thinkers, Leading up to a Luring Lottery.* In: **Metamagical Themas.** *Basic Books*, 1986.

F. Moller, private communication.

[9] This problem isn't easy—and there will not be anything like it in the exam. It is a good exercise in reasoning about strategies, which is why I left it in the notes.

Chapter 3

Two important algorithms

In the previous chapter our remarks regarding calculating equilibria were mostly directed at games in normal form. As is explained in Chapter 1 many games of interest are too large to be presented in normal form. Recall that the size of the game tree is exponential in its height and that the number of strategies is roughly exponential in the size of the game tree.

In order therefore to play games such as chess, checkers, go and many others we cannot rely on having the game in normal form. This chapter is concerned with two algorithms that allow us to find certain solutions (which are not necessarily equilibria) to games in extensive form. Moreover these algorithms are also used to find 'good' strategies in situations when precise solutions cannot be calculated because the game in question is too big.

Our starting point is the proof of Theorem 1.10 and Proposition 2.12. In the former there is a hint of an algorithm which there determines whether or not a player can force a win or ensure a draw. In the latter the same idea is applied to calculating the highest pay-off a player can guarantee for himself, no matter what the other player does.

We extract that algorithm and consider what it calculates in more general situations. The result is the **minimax** algorithm. It can be refined in a way that makes it faster by carefully considering whether the entire game tree has to be searched. The result of this is known as **alpha-beta search** or **alpha-beta pruning**. Even in situations where the game tree is too large to be traversed fully these algorithms can be employed, and the chapter on game-playing program describes their vital role for that situation.

3.1 The minimax algorithm

In the proof of Proposition 2.12 an algorithm is described that finds the *value* of a 2-person zero-sum game. It does so by recursively determining the value of each sub-game. In a 2-person zero-sum game the value is the largest expected pay-off Player 1 can guarantee for himself (and its negative is the largest expected pay-off Player 2 can guarantee for herself).

Looking at the algorithm again we note the following:

- There is no reason to restrict the algorithm to two players.
- If we run it in the non zero-sum case we have to run it separately for each player.
- In that case when carrying out the algorithm for Player i we should at each step calculate
  - $\max v_j$ if it is Player i's turn at the current node and
  - $\min v_j$ if it is another player's turn and
  - $q_j v_j$ if it is a chance move and q_j is the probability for choosing the jth sub-tree,

  where v_j is the value already determined for the jth sub-tree.
- For each sub-tree this algorithm calculates recursively the largest expected pay-off Player i can guarantee for himself (even in the worst case). We call this **the value of the game for Player i.**

- If for each node where it is Player i's turn we remember which choice(s) lead to the maximal pay-off we can determine the strategies that guarantee this pay-off.

In the case of a 2-person zero-sum game the number determined for Player 1 is the value of the game. This algorithm has to be altered if we are dealing with games of *imperfect information*.

Question 22. How could the algorithm be adapted to cope with a game of imperfect information?

However, in the general case it is important to note that the number calculated for each player does not necessarily correspond to the pay-off at an equilibrium point. All we are doing is calculating the 'best choice that maximizes the pay-off in the worst case'. We hence implicitly assume that the other players will play in a manner that does the worst for Player i when in reality these players are all concerned with maximizing their own pay-off. Only in the 2-person zero-sum case do the two objectives coincide.

One could try an alternative algorithm: Instead of looking for the smallest v_j if it is another player's turn one could attempt to pick the v_j which belongs to the sub-tree that maximizes the pay-off for the player whose turn it is. However, there might be several such moves leading to the same pay-off for that other player, but to different pay-offs for Player i, so this is not a deterministic algorithm any longer! Clearly there is no sensible way of picking one of several such subtrees.

Hence we have an algorithm that can calculate the value of the game for each player which we can think of as *minimal expected pay-off for each player* (in the sense that it is the pay-off that a player can ensure for himself, no matter what the other players do). In the case of a 2-person zero-sum game

- the value for Player 1 is the value of the game while
- the value for Player 2 is the negative of the value of the game.

Question 23. What are the advantages and disadvantages of following the play suggested by the algorithm in a 2-person general sum game?

What we have described here is a *recursive algorithm*: To find the value of a position (and its corresponding sub-tree), we need to know the values of all its immediate successors, and to calculate those we need to know the values of *their* immediate successors, and so forth, until we reach a final position, where we can read off the value from the pay-off function.

But how do we calculate this value in practice? So far we have thought of it in a *bottom-up* approach, that is, one starts at the final positions and work one's way up from there. This approach is demonstrated in Figure 3.1.

But in practice the game tree is typically given by

- the root and
  - links to all the immediate sub-tees or
  - a way of generating all the immediate sub-trees.

(Sometimes a node is linked to its siblings, but usually there are no additional links available.)

Hence in order to get to a leaf we have to work our way down from the root. If we found all the leafs first in order to follow the bottom-up approach we would have to traverse a lot of the tree several times. Many positions and their values would have to be kept in memory at the same time (or generated repeatedly).

Instead this algorithm is typically programmed using a *depth-first* approach. An example of calculating the value for Player 1 for the root of the game tree (and thus for all the positions) is given in Figures 3.2 and 3.3. The only values that are known at the start are those of the leaves (once the leaves have been found/generated).

This procedure is called the **minimax algorithm**, because minima and maxima (over the values of successor positions) are taken as it proceeds. Figures 3.2 and 3.3 assume that from each node, we can only move to its children and its parent. If the chosen data structure knows about siblings then it is not necessary to return to the parent between visiting two siblings,

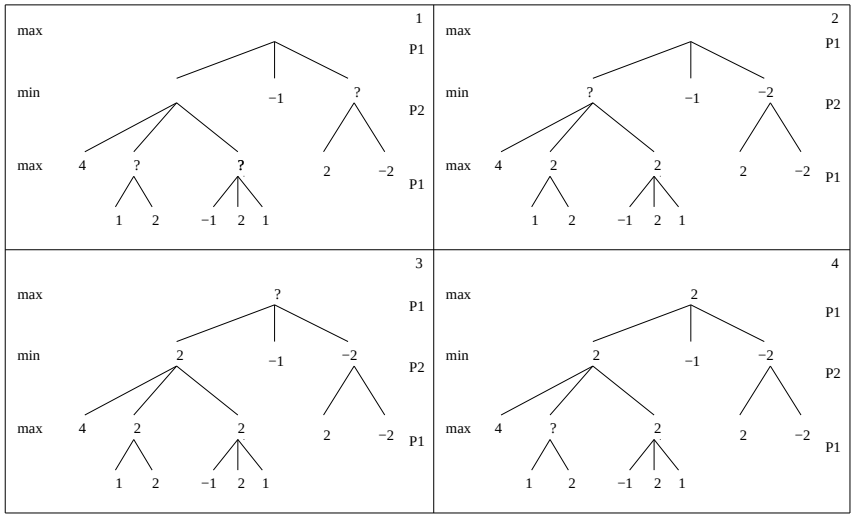


Figure 3.1: A bottom-up approach to determining the value

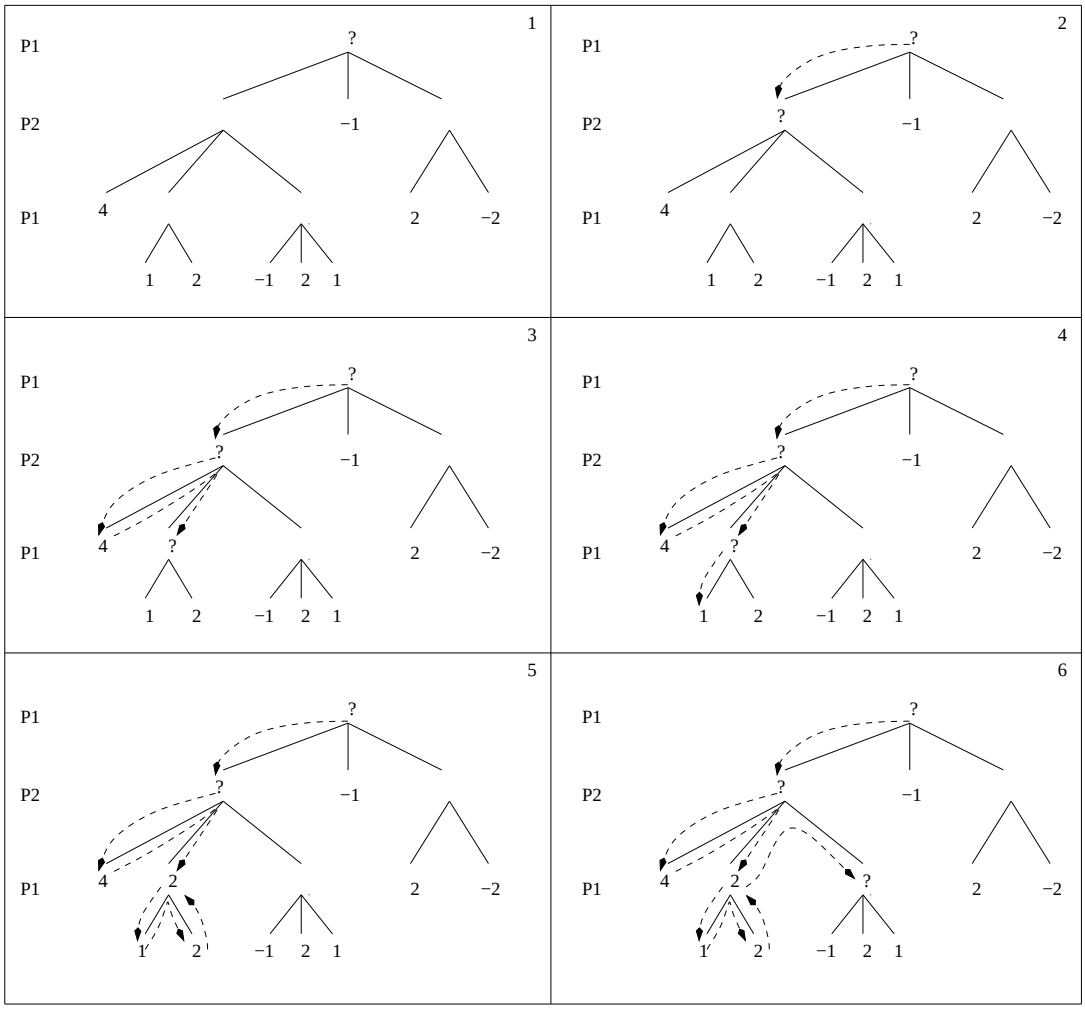


Figure 3.2: A depth-first approach—the minimax algorithm

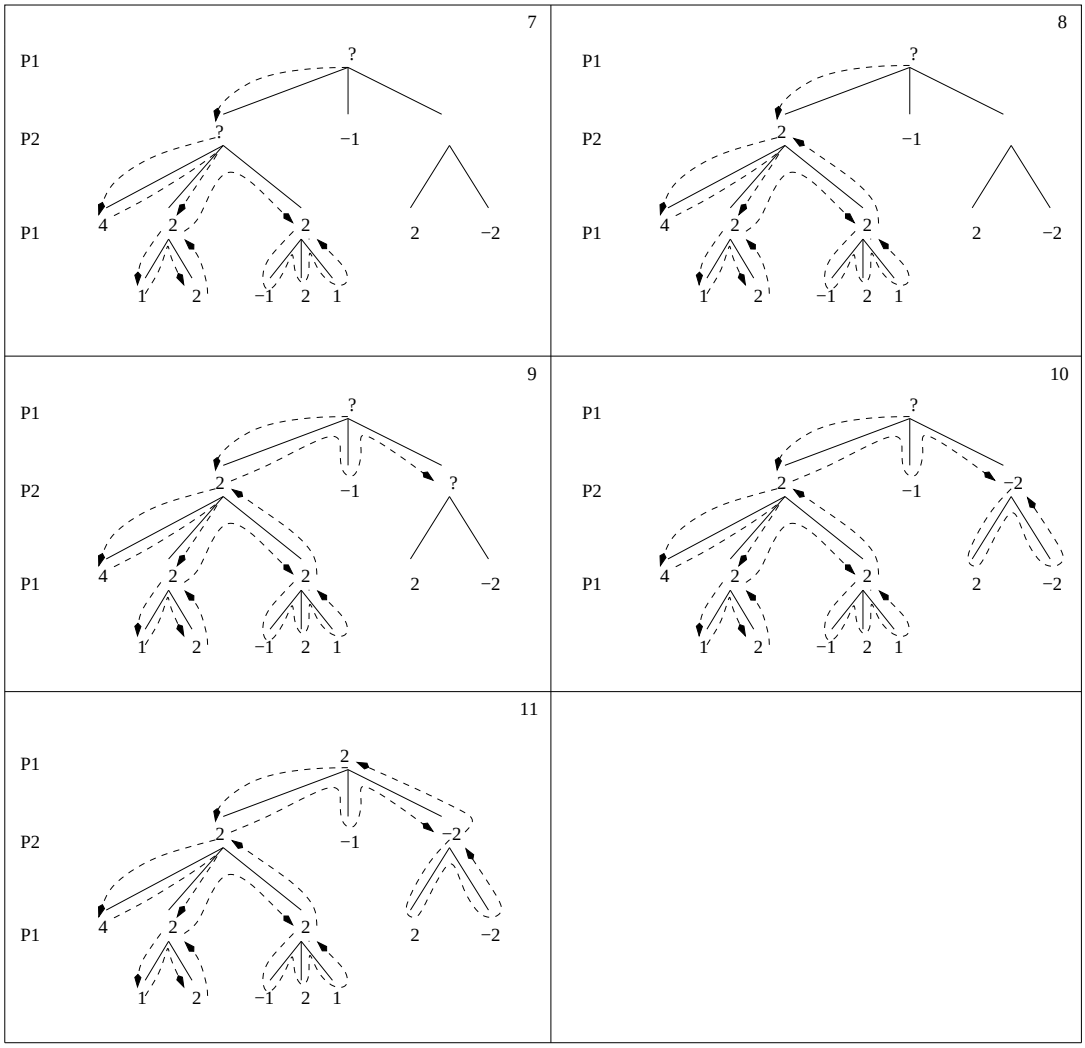


Figure 3.3: A depth-first approach—the minimax algorithm

but this speed is gained at some cost to the space needed to store the currently required part of the game tree.

Hence we have identified a way of finding equilibria point for a 2-person zero-sum game of perfect information which does *not* require that all the strategies be identified first (provided we record all the decisions that lead to the identified value). In the next section we discuss ways of speeding up the minimax algorithm (so that we can deal with even larger games).

B Exercise 24. (a) Carry out the minimax algorithm for each player in the game pictured in Figure 3.4. What are the values for the root of the game, which are the corresponding strategies, and what happens if they play these strategies against each other?

(b) Apply the minimax algorithm to the game from Exercise 10 (b).

It has to be pointed out here that the minimax algorithm as described does *not* apply to games which are of imperfect information. This should come as no surprise: While all the positions in the same information set must have the same immediate moves, 'down the line' the game trees will become different, and in particular they will lead to different pay-offs. A typical example are card games: A player only knows his own hand, and his moves are the same initially no matter what cards the other players hold. But eventually the pay-off will depend on everybody's hand, so there cannot be a good decision procedure which makes do without that information (the game trees for card games are typically such that the potential

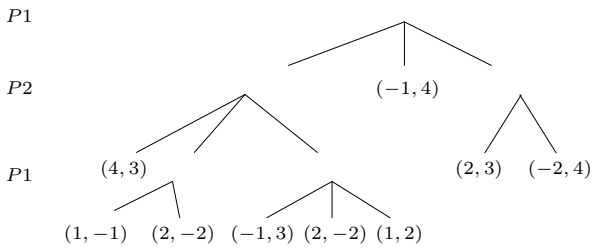


Figure 3.4: A non-zero sum game

difference in pay-offs for positions in the same information set is huge). Compare Simplified Poker, Chapter 2.7. One can adjust the algorithm to cope with that to some extent, but the result is not necessarily as useful:

If the uncertainty regarding the current position is a result of

- a chance move then one could take the expected pay-off for the Player i in question;
- another player's move then one could take the worst-case pay-off for Player i.

However, in many practical examples the uncertainty arises from a combination of the two, requiring a rather more careful analysis. Otherwise the algorithm will no longer calculate the value as defined.

3.2 Alpha-beta pruning

The minimax algorithm allows us to find strategies maximizing the worst case pay-off without having to generate all strategies first. It does so by traversing the tree, visiting each leaf node precisely once. This looks at first sight as if it is an optimal solution. But, in fact, it can be improved upon. In the small games typically drawn in this notes the difference may seem trifling, but with larger games (which can easily be held in a computer) the difference can be enormous. Most game-playing programs for chess, go and many other games use the algorithm we introduce below. They would be much less effective without it. We see in the following chapter how it is typically used.

The idea behind alpha-beta pruning is to use the values calculated for the sub-games *so far* to make decision on whether or not it is worthwhile to look at other sub-games. Consider the situation in Figure 3.5.

At the root, we have as yet no idea at all what the real value is. All we know is that it is a real number, so we know it is somewhere in $(-\infty, \infty)$. In order to find out its value we need to search the children of the root. As we pass down to them we pass on what we know already in terms of the possible values. So far, that is *nothing*—that is, the values we are interested in are all those in $(-\infty, \infty)$.

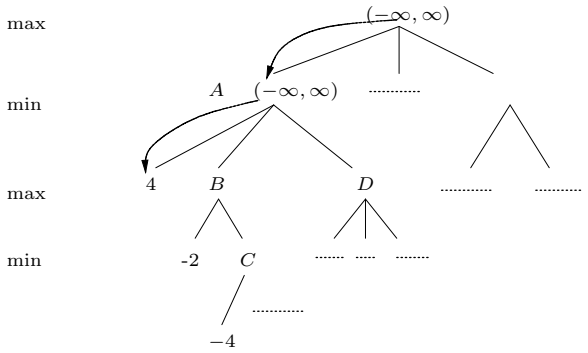


Figure 3.5: A minimax calculation passing down relevant ranges—alpha-beta pruning

Then we reach a position for which we can just read off the value, namely 4. Once we know this we know that the parent of that leaf, the node marked A, being a min node, will have a value of *at most* the value of this child, 4. So when we return to A we update the available information: The true value of the note is in the interval $(-\infty, 4]$. When we look at other children of A we are only interested in their values provided that they are in the given range. Values larger than 4 will not contribute to our calculation of the value of A. Hence when we descend to the next child, the node B we pass along that we are only interested in values in $(-\infty, 4]$.

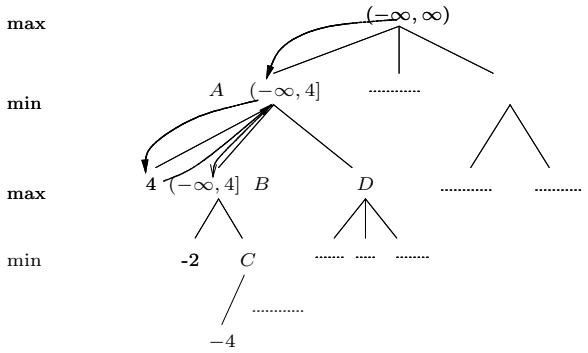


Figure 3.6: A minimax calculation passing down relevant ranges—alpha-beta pruning

Passing down to the first child of B we can read off another value, namely -2. That value is returned to B and we now know that the value of B is at least -2 since it is a max node. We therefore now have

- a lower bound -2 for the value of B and
- an upper bound 4 for the value of A which gives us an upper bound for those numbers we are interested in when it comes to determining the value of B.

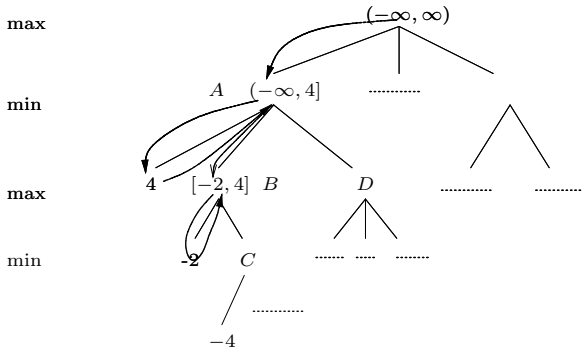


Figure 3.7: A minimax calculation passing down relevant ranges—alpha-beta pruning

We visit the node C, again passing along the range of values we are interested in. We find a value -4 which is *outside* that range.

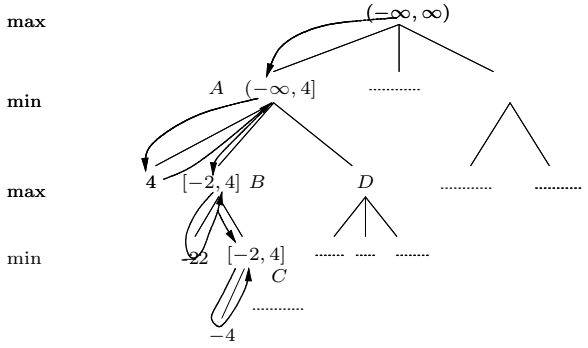


Figure 3.8: A minimax calculation passing down relevant ranges—alpha-beta pruning

We now know the following:

- In order to determine the value of A we only need to consider values of B which are below 4.
- The value of B is at least -2 and so only need to look at values for its children if these are above that number
- The value of C is at most -4, hence it will not contribute to calculating the value of B.

Hence we do *not have to determine the value of* C *to find the value of either* B *or* A! As a consequence we do not have to look at any of the other children of C. We have 'pruned' part of the tree and do not have to visit any nodes in that part.

When we return to the node B we *know* that its value is -2 too, and we report that value upwards to *its* parent A. We can now update what we know about the value of A–since it is a min node its value will be at most -2.

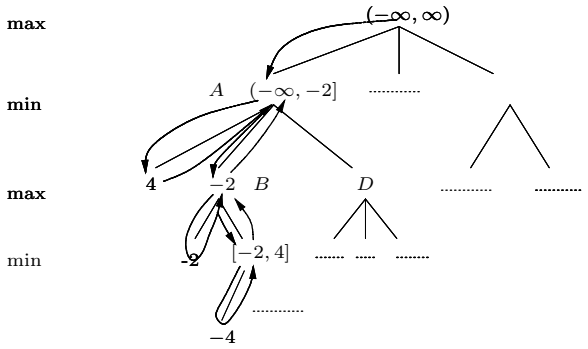


Figure 3.9: A minimax calculation passing down relevant ranges—alpha-beta pruning

We move to the node D and then to its first child. Let us assume that we eventually establish a value of 1 for it as indicated in Figure 3.10. Then we can once more stop searching the tree below D: We know that a value for D below -2 contribute to the value of A. We have just discovered that the value of D is at least 1, so the true value of D is not relevant to calculating the value of A. We have no need then to look at any other children of D to determine the value of A and we can prune part of the tree once again.

The fact that 1 is not in our range of interest is equivalent with the fact that it will not be of relevance to the value of nodes higher above it. This establishes a value of -2 for the node A.

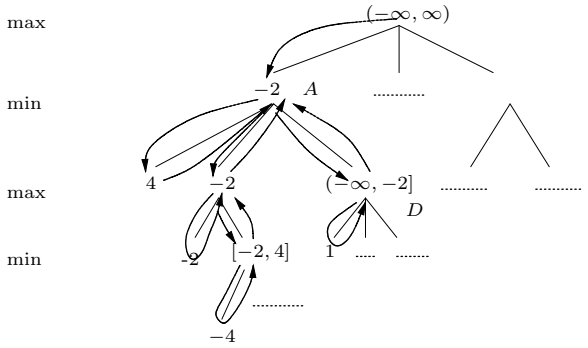


Figure 3.10: A minimax calculation passing down relevant ranges—alpha-beta pruning

Note that in order to find this value we *did not have to search the entire tree rooted there*! By passing down the relevant ranges as we recursively move through the tree in a depth-first manner, we may cut off the search at various points. That is what alpha-beta search is all about. We continue with our example.

Returning to the root for the first time we can finally give a smaller possible range. The value we will obtain for it is *at least* -2 since the max player can ensure that by moving to the first child. Let us assume that further search eventually returns a value of 1 for the second child of the root.

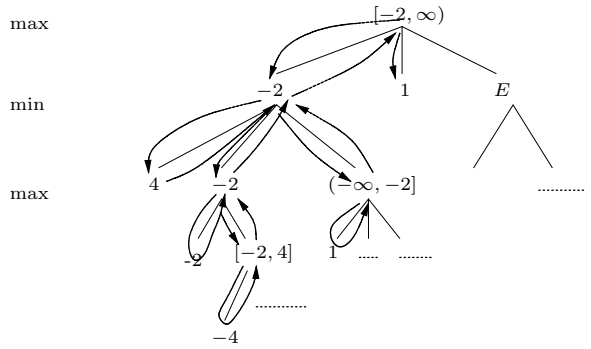


Figure 3.11: A minimax calculation passing down relevant ranges—alpha-beta pruning

Again this value is reported to the root and leads to an update of the relevant range—we now know that the value of the root is at least 1, which the max player can ensure by moving to the second child of the root.

We descend to the third child of the root E, again passing along the relevant information.

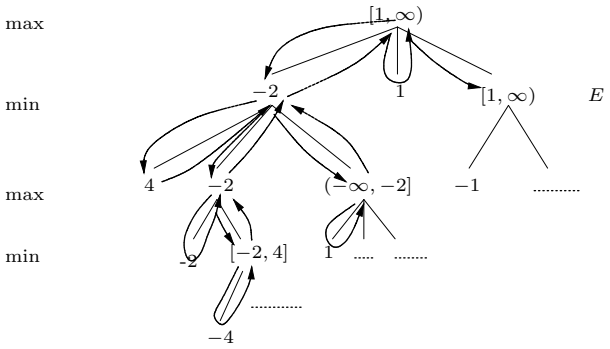


Figure 3.12: A minimax calculation passing down relevant ranges—alpha-beta pruning

If eventually we find a value of -1 for the first child of E as in Figure 3.13, it is again outside of the relevant range, and once more we may stop our search without traversing the tree further. We now know that the value of the root is 1.

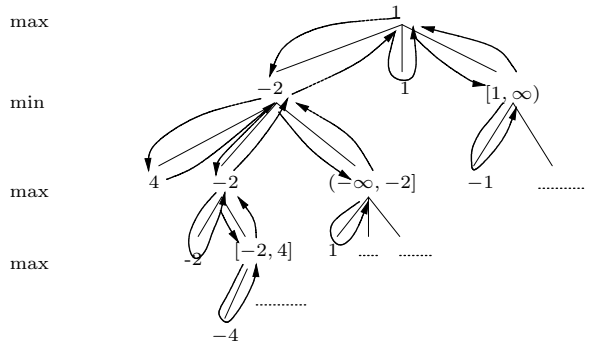


Figure 3.13: A minimax calculation passing down relevant ranges—alpha-beta pruning

Let us summarize how we did this. On each recursive call of the procedure that determines the value, two parameters (the 'alpha' and 'beta' in 'alpha-beta pruning', or 'alpha-beta search') are passed along to indicate the relevant range of values. At the start, these are set to $-\infty$ and ∞ respectively. When we get to a new node we know that are only interested in values between α and β. When we now visit the children of that node we (eventually) find a value v for it. We then do the following.

Value v is	Node is of type	
	max	min
below or equal to α	ignore v and move on to next child	value of current node is irrelevant; return to parent
between α and β	set α to v and move on to next child	set β to v and move on to next child
above or equal to β	value of current node is irrelevant; return to parent	ignore v and move on to next child

It should be clear where the 'pruning' in alpha-beta pruning comes from: By finding that we do not have to explore a particular subtree we effectively cut it off from the tree that we will have to traverse to find the value of the root. In other words, we 'prune' the tree under consideration by cutting off irrelevant parts.

We note that alpha-beta pruning is particularly effective if the *first* child investigated is the best move for the Player who makes it:

The final value of a node is the value of that particular child which corresponds to the best move for the Player whose turn it is. Hence by investigating that child first we guarantee that we will never get a *better* value coming from another child, which means that *all* the other children will allow some pruning!

When carrying out alpha-beta pruning it is therefore advantageous to try promising moves first. How to find 'promising' moves is a science in itself which clearly has to be adapted to the game under consideration. We look at this issue in the section on game-playing programs. Alpha-beta pruning can be applied to find values for a game for all players, and strategies which ensure that a player will get a pay-off at least as high as that value.

This works provided that the game is small enough so that the game tree can be created as required. For many interesting games this is not the case. In that situation something that is typically is done to have a *heuristic* that tries to 'guess' the true value of a node. Alpha-beta pruning is then carried out by searching to a given depth and then using the heuristic value *as if it were the actual one.* Clearly if the heuristic is far from reality then this leads to poor results.

In a 2-person zero-sum game may use alpha-beta pruning to find equilibrium strategies as well as the pay-off at the equilibrium points. In particular that means if there is a winning strategy for one of the players then this algorithm will find it.

S Exercise 25. Find a winning strategy in the following games using alpha-beta pruning. Try to do so without first creating the game tree, and make use of symmetry whenever you can. Which player can force a win? For the player who isn't so lucky, can you find a 'good' strategy that allows him to win if the other player makes mistakes? Can you find a way of generalizing your strategy to larger Chomp/Nim games?

(a) 2×3-Chomp;

(b) $(3, 3, 3)$-Nim.

Summary of Chapter 3

- For general games we can define a *value for each player* which is the minimum pay-off the player can secure for himself.
- For 2-person zero-sum game the value for Player 1 agrees with the one introduced in Chapter 2.
- If a game is small enough that its tree can be fully traversed then the *minimax algorithm* can be used to find these values, as well as strategies guaranteeing these as the worst possible pay-off.
- This can be improved upon by *alpha-beta pruning*, where information regarding relevant values is passed down in the search, so that parts of the tree can be discarded as irrelevant. This makes finding good strategies faster, and it allows applying the idea to larger games.

Sources for this chapter

Most of the material in this section is well known, although this presentation is original. I did have a look, however, to see how other people present it, and found:

David Eppstein's notes on *Strategy and board game programming* at the University of California at Irvine, `http://www.ics.uci.edu/~eppstein/180a/`.

A.N. Walker's notes on his course in *Game Theory* (with a part on alpha-beta pruning) at the University of Nottingham, `http://www.maths.nott.ac.uk/personal/anw/G13GAM/`.

Pui Yee Chan, Hiu Yin Choi and Zhifeng Xiao's webpage on *Game trees and alpha beta search* at McGill University, `http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11/`.

Chapter 4

Game-playing programs

The previous chapter concerns games that are small enough that alpha-beta pruning can be applied to determine values as well as strategies guaranteeing them. In particular in the case of 2-person zero-sum games we can use it to find equilibria. There are, however, many games whose game trees are too large to be amenable to these methods. This chapter is about establishing how such games still find good moves to make (and so determine good strategies). The methods described here underlie every available commercial chess program as well as many others. We now only consider 2-person zero-sum games.

4.1 Writing game-playing programs

We introduce the various tasks that need to be carried out to write a game-playing programs and then discuss them in some detail. Examples of the kinds of games that we are interested here are chess, go, othello (also known as reversi), hex, go-moku, connect-4, checkers, and backgammon. Note, however, that we do not make many comments about chance elements.

The first task that has to be performed is to implement a way of **representing the current position** and **generating the legal moves for a position.** The latter is necessary so that the game tree can be generated as needed.

All successful programs for the kind of games we are interested in involve some version of a minimax algorithm with alpha-beta pruning. Since we are interested here in finding a good next move, for the remainder of this chapter we refer to this as **alpha-beta search**.

But how can we employ this algorithm in situations where it is not feasible to work one's way to the final positions of the game in order to calculate values for the various nodes? What these programs typically do is the following:

- Use alpha-beta pruning to search the game tree to a specified depth.
- When a node at the given depth is reached use a *heuristic* to determine an approximate value for that node without searching the game tree below it.
- Use the approximate value as if it were the real one.

This idea has proved to be very successful in finding good strategies.

Heuristics like this are referred to as **evaluation functions**. They assigns a score to a given position based on various criteria which depend on the game in question. This is a tough job: Imagine you had to look at a chess board and say somehow how good a position this is! Typically there is *no one right answer* to the question—what is required of a good evaluation function is that it gives a *decent approximation*, and in particular that it can *compare one position to another*. To put it bluntly, an evaluation function is *guessing* the true value of a position, and it is the job of the programmer to make it the *most educated guess possible*. In particular if the evaluation function assigns a higher value to some position than to another then it ought to be the case that it is easier to win from the former than from the latter!

For such a program to work well

- the evaluation function must be as accurate as possible and
- the program must search the game tree to the highest depth possible (using alpha-beta search).

The limiting factor regarding the latter is speed—the program needs to return a move within a given amount of time. Hence programmers typically aim to optimize the processes of

- generating the game tree (which requires finding, doing, and undoing moves) and
- writing the evaluation function in a form that is quick to calculate for a given position.

In competitions programs typically have a limited amount of time, either per move or for a given number of moves. Good programs will search the game tree as deeply as possible given time constraints while also keeping track of the best move found so far. If the search has to be aborted for time reasons this allows the program to play at least a sensible move.

To see a graphical presentation of the way a particular chess programme searches among positions, go to `http://turbulence.org/spotlight/thinking/chess.html`.

One amusing phenomenon in computers playing games is that every now and then they spot lines, due to carrying out an exhaustive search, which are so complicated that human opponents are unlikely to see them. One program, playing a Grandmaster, suddenly seemed to offer a rook for capture for no reason that the assembled experts could discern. After the game was over they made the machine go back to that position and asked it what would happen if it had made the move judged 'obviously better' by the audience. The machine pointed out an intricate mate which it was trying to avoid. Arguably, it would have been better off leaving the rook alone and just hoping that the opponent wouldn't see the mate!

Another phenomenon that comes up with machines playing games is that if there is a mistake in the program, for example it has a weakness if a particular line or position comes up, then this weakness can be explored over and over again, because most programs are incapable of learning. Many tournaments between various programs seemed to be more about who could discover whose built-in faults, rather than whose program genuinely played best!

In this chapter we discuss the following:

- possible solutions to the problem of representing the board and finding valid moves,
- some potential criteria for finding reasonable evaluation functions,
- variations on carrying out alpha-beta search and
- some of the pitfalls that typically arise, as well as other components that are employed by some programs.

4.2 Representing positions and moves

Speed is the one thing that is at the bottom of every decision made when designing a game playing program. Hence even the internal presentation of positions and moves is important: Speeding up the program's ability to generate, do and undo moves makes the whole search process faster, and since the program will spend a lot of time doing that, getting this right can make a real difference. Game programmers quite often use clever encodings which allow bit-wise operations to make their programs faster.

Let us consider the example of chess. Obviously the current position on the board will have to be represented in the program. But there also has to be a way of denoting whose turn it is, whether a player can still castle, and whether a capture *en passant* is possible. Worse, there are rules about *repeating previous positions* in chess (which will lead to a draw), so the program has to have a way of remembering those! Clearly, whichever format is used

to represent a board position, saving all previously seen positions would be expensive, and searching them a nightmare.

Chess programs typically use a large hash table to keep track of positions that have occurred in play. This table can also be used to keep track of what is currently known about a position that has been searched previously: When we search to depth, say, 8 then every position we may be in after we have made a move, followed by one of the other player, has already been searched to a depth of 6. If we can remember something about that search we can use the information

- to pick an order in which to examine the available moves (remember that the better the first searched move is the more we can prune the tree)
- provide a 'fallback-move' in case carrying out the current search to the desired depth takes too long.

Something else that should be easily possible in the program is to *undo* moves. This is not so much in case a (human) opponent wishes to cheat by reconsidering a move made, but because in the course of generating the game tree and carrying out alpha-beta search, moves have to be made and undone. That means that the program will have to remember where a given piece came from, and which, if any, piece was captured by the moves.

A fairly obvious presentation of the game board is as an 8×8 *array*, with each element of the array containing the code for one (or none) of the pieces. To generate valid moves, a program then has to loop over the array to pick up one piece after the other. The moves of knights are easy in the sense that all it takes for a given move to be valid is that the field where the piece ends is not already occupied by a figure of the same colour—that's just one look-up operation. For a king, not only does the field it goes to have to be vacant of own pieces, it also must not be a field any of the enemy pieces may go to in one move, so the program also has to figure out the valid moves for the opponent. For the other pieces, rook, bishop, queen and pawn, the program has to make sure that all the fields *on the way* to the new one have to the new position are empty, generating many more look-up operations.[1] Finally the program has to ensure that the move would end on a valid field and not go beyond the borders. When a move has been made the position is updated by changing the entries in source and target fields. Even with a simple figure like a pawn, there are four possible moves: move one field forward, move two fields forward, capture on the left and capture on the right (including the possibility of capture *en passant*).

An alternative to this presentation is to give each square of the board a number (a single byte), where the high four bits decode the row and the low four bits the column, leading to a table like this:

		a	b	c	d	e	f	g	h	
		0000	0001	0010	0011	0100	0101	0110	0111	low bits
8	0111	112	113	114	115	116	117	118	119	
7	0110	96	97	98	99	100	101	102	103	
6	0101	80	81	82	83	84	85	86	87	
5	0100	64	65	66	67	68	69	70	71	
4	0011	48	49	50	51	52	53	54	55	
3	0010	32	33	34	35	36	37	38	39	
2	0001	16	17	18	19	20	21	22	23	
1	0000	0	1	2	3	4	5	6	7	
	high bits									

To move one field to the left or right, just subtract or add one. To move up a row, add 16, to move down a row, subtract 16. The whole board is then represented as an array with 128 entries, only 64 of which correspond to actual fields on the chess board. At first sight,

[1] At least the program should be clever enough to remember fields previously looked up when trying to move a bishop, rook or queen one field beyond the one previously tried.

this is puzzling—why use this particular presentation?[2] The answer is that checking whether a number describes a valid field is very easy: It does if the number i satisfies $i\&0x88 == 0$ (where & is a bitwise operation and $0x88$ is the hexadecimal representation of the number 136). It is the reason why this presentation is sometimes referred to as $0x88$. It provides a sufficient speed-up over our first one that it is implemented in a number of chess playing programs.

Another popular representation uses *bitboards*. Rather than use an array where entries stand for a square on the board and hold the type of piece on that square, the approach here is, for each piece, to hold a presentation of the board indicating where to find such a piece.

The advantage of this is that, given a particular piece, for each square of the board we require only one bit to state whether or not that piece can be found on that square. That means that for every piece we can store the required information in a 64-bit word (or two 32-bit words). Then many operations can be carried out using bit-wise Boolean operations, speeding them up considerably. Let us consider an example.

Figure 4.1: A chess position

In the board position given in Figure 4.1, the following bitboard describes the position of the white pawns.

$$\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

The bitboard representing all the fields occupied by black pieces is then merely the bit-wise 'or' of all the bitboards for the black pieces. Bit-wise operations are extremely fast, which can be used to good effect here. Similarly we can compute a bitboard for all the occupied positions, or just those occupied by a white piece. A move of a piece by one row consists of shifting the corresponding bitboard by 8. If a bitboard of empty fields is required, a bit-wise

[2] My source claims that performing a look-up operation in a one-dimensional array is a tad faster than in a two-dimensional one, but I'm not sure about that.

negation of the bitboard of occupied fields will suffice—this allows a quick test of whether the intended move is valid. If required, *all* the legal moves of pawns by one field can be stored in a bitboard, and similarly for all legal moves of pawns by two fields (carrying out a bit-wise 'and' operation with the board which contains '1's on the fourth row and '0's everywhere else makes sure only the pawns which are allowed to move two fields will be considered—and constant bitboards can be prepared at compile time to be available in a library). Pawn captures are equally quick to calculate (shifting the bitboard by 7 or 9 and bit-wise 'and'ing it with the bitboard for pieces of the opposite colour).

The code required when using bitboards is more complicated than that for using arrays. However, it has the following advantages:

- bit-wise operations are fast;
- bitboards required more than once only have to be computed once (compare that with checking whether a field is occupied in the array representation);
- several moves can be generated at the same time.

A disadvantages is that it is more complicated to turn a bitboard of possible moves into a list of such moves. Many tricks are used to make such operations (for example finding the non-zero bits in a bitboard) fast.

No matter which representation is chosen, undoing moves should be done by having a stack of moves made with sufficient information to undo them—using a stack of positions is a lot slower.

In many ways, generating moves (and representing the board internally) is the easiest of the tasks we have identified. Once it has been implemented, it can typically be left alone, and no changes will be required down the line provided that sufficient attention has been paid in the design process. However alpha-beta search works fastest if the best move is considered first. This is typically done by using results of previous searches, but that information has to be integrated in the part of the program that generates the moves to be investigated.

4.3 Evaluation functions

Whereas we are mostly concerned with speed when it comes to the internal representation of moves and position, finding a good evaluation function is a matter of implementing *knowledge* about the game in question.

The basic idea for this is fairly simple. Since we cannot expect to work our way through the entire game tree for chess, or go, we have to have a way of turning positions into values *without further searching the game tree*. That is the purpose of the evaluation function. It provides us with a provisional value for a position, which by force has to be crude. After all, it is a static process which does not take into account how the game might develop from a given position. Instead, it is an attempt to assign a value by merely looking at *what is currently on the board*. If this part goes wrong, the alpha-beta search will pick moves which may not lead to good positions, which can then be exploited by the opponent. There is no way of guaranteeing one has 'the right' evaluation function (such a thing certainly does not have to be unique), and a big part of writing game playing programs consists of watching the program play and fine-tuning the evaluation function accordingly. Apart from a few common sense ideas, evaluation functions are therefore mostly based on *heuristics*.

There is one aspect regarding evaluation functions which concerns *speed*: Evaluating a position can be quite a complicated process, with various aspects of the position requiring scoring. Therefore calculating such an estimated value for each position separately will inevitably repeat some of the work done previously, and hence be fairly slow.

By forcing an evaluation function to provide a value by looking only at the current position it is difficult to take into account, for example, whether the piece in question is protected against capture by another piece, or whether in case of a capture a recapture would make the

exchange worthwhile. Some evaluation functions allow a limited further search of the game tree to provide a more accurate value by taking these issues into account.

Here are some aspects that might be relevant to evaluating a position. Just how important they are will vary from game to game. To judge a position it is typically important to do these evaluations for *both* players—having many pieces on the board does not give White any advantage if Black is about to checkmate him!

- **Material.** In chess, that would be the number of pieces, where each piece gets its own value, in go, it would be a count of pieces on the board, and similarly in, say, othello. This is not equally useful in all games, however: In othello, for example, it is not really the number of pieces in one's own colour that is important, but whether one holds specific fields, for example corner positions. Quite often the player with the better position will have *fewer* pieces on the board. There are other games where the number of pieces may be irrelevant.

- **Space.** In some games it is possible to divide the board into areas of influence, where a given player controls a number of fields. This is particularly relevant for go. In chess, one can count the number of fields threatened by one player for this purpose, and in othello the number of pieces which cannot be taken by the opponent (a connected group of pieces surrounding a corner). One could just calculate the size of these regions, or attach some sort of weight to them if not all fields are equally important.

- **Mobility.** Having many different moves available can be an advantage in a game, othello being a particular example. For chess there is some doubt as to whether or not this is a useful measure—some people have tried and discarded it while others have retained the principle.

- **Tempo.** In games such as go there is a question of which player has the *initiative*, that is the ability to make moves which advance his own agenda (as opposed to having to make defensive moves whose main purpose is damage limitation). Often having the initiative means that in reply, the other player has to make a defensive move to avoid worse, leaving the initiative with the original player. In other games, some sort of parity argument works: there are positions which lead to a win for the player whose turn it is (or sometimes for the player whose turn it is not), and that often merely depends on numbers easy to evaluate (in nim and connect-4 these arise quite often).

- **Threats.** Can one of the players capture (or threaten to capture) a piece? In Connect-4, or go-moku, does one of the players have a number of pieces lined up already? In othello, is a player threatening to take a corner?

- **Shape.** This is really about various pieces on the board relating to each other. In chess, for example, a line of pawns advancing is much stronger than, say, pawns sharing a column. In go, shape is about 'territory to be'—a few well-placed stones outline a territory which the player can defend when threatened.[3]Judging shape can be very difficult, and typically shape is formed by a number of moves being made, where every one such move improves the position only incrementally, but where the resulting position can be a lot stronger. Shape is also typically a *long-term target*. An evaluation function partially based on shape will have to be based on something other than the simple addition of piece-based evaluation functions we discussed above.

- **Known patterns.** In many games there are patterns which come up over and over again. This is particularly true for go, where there are many libraries of sequences of moves concerning a small area (such a sequence is known as a *joseki*, where players following such an established line can maintain balance). In chess, a bishop capturing

[3]This is an over-simplification, really, but then go is the game which to date most stubbornly defies programmers.

a pawn on the border is often trapped. In othello, it is sometimes advantageous to sacrifice one of the corners if one can then force ownership of another corner. It might be worthwhile to program such things explicitly in order to avoid making a bad move, or to follow the moves from a library if certain constellations are reached. What is typically difficult is to reliably recognize positions where such patterns should be applied, and to adjust the moves identified to the current situation.

The above criteria result in a variety of components that might make up the actual evaluation function. Typically these components are weighted and then added up, where the weights are determined based on heuristics. The reason for this is that summation is a simple process for combining numbers, which is fairly fast. There certainly is a question of whether one could not assign probabilities (for the other player to choose various moves, for example) to help with defining weights, but game programmers typically do not use such an analysis.

Typically an evaluation function consists of a number of components each with a weight, that is, given a position p we have

$$e(p) = w_1 e_1(p) + w_2 e_2(p) + \cdots + w_n e_n(p),$$

where

- the w_i are numbers known as 'weights' and
- the e_i implement one of the measures suggested above.

One can think of the weights as allowing us to give varying importance to the different parts of the evaluation function.

Typically to improve an evaluation function there are then two alternatives:

- Make the evaluation function more sophisticated by introducing further functions e_i.
- Adjust the weights w_i to get better results.

Much time is typically spent on the second of these. Here are a few ideas that can be used to fine-tune the weights.

- **Deducing constraints.** In games such as chess, every piece is given a material value. Clearly a rook, say, is more powerful than a pawn, and the material value should reflect that. By analysing typical games, it can be possible to deduce constraints that these values should satisfy. chess players know, for example, that it is usually advantageous to exchange a rook for a two pawns and a bishop, or two pawns and a knight, but a disadvantage if there is only one pawn involved. Hence the weight of a rook should be below that of two pawns and a bishop, but above that of one pawn and a bishop. That drastically reduces the numbers one might have to try.
- **Hand tweaking.** This is what happens most often in practice. Programmers watch their implementation play and then try to judge which parameters should be changed, and how. They perform the change and watch again. This produces reasonable results fairly quickly, but requires that the programmer knows enough about the game to analyse what is going wrong.
- **Optimization techniques.** Rather than use human judgement to tweak any parameters involved, one can use general optimization techniques. One example for these is 'hill climbing': Small changes are made to the parameters, and changes are only kept if they improve the performance. This requires some sort of measure to judge performance, for example the percentage of won games against some opponent. This tends to be slow and risks being stuck in positions where each small change makes the performance worse, but where a big change might bring huge gains (such situations are known as 'local optima'). This algorithm can be modified by randomly sticking with some changes which

do not improve performance in the hope of avoiding this problem. The 'randomness' is controlled by some probabilities which typically should start out fairly high and then become smaller as a good value is approached. This adjusted method is slower than the original, but can get good values.

- **Learning.** When the first game playing programs were written it was envisaged that machine-based learning would be the most important aspect in developing them. This faith in Artificial Intelligence has not proved appropriate in practice. All world-class game-playing programs use other principles foremost. Examples of approaches based on learning involve genetic algorithms and neural networks. Both are in practice rather slow methods, and their main advantage is that they do not require much 'human intelligence' in the form of knowledge relevant to the game in question. The reason they can be very slow is that the number of test games required is typically very high (commercial game programmers who have worked with these approaches tried about 3000 matches to allow the program to learn about the game, and that was not sufficient to perform better than hand tweaking). Another problem is that if the program plays against an opponent that is too good it will lose all the time and never start learning. As a result learning is more often applied to smaller games.

Almost all these methods require some sort of measure for the performance of the evaluation function which results from a particular choice of parameters. One way of doing so is to run the program on a large suit of test positions which come, for example, from high-quality human games, and see whether the program can follow the winner's actions. (This is actually useful to just test whether one's program plays reasonably well, for example by trying it on 'checkmate in two' kind of positions.) This method is typically combined with letting the program play a large number of matches against a known opponent, such as another program, or even a version of itself which has different weights, so that the two can be compared to each other. The problem with the latter is that playing what is, for the most part, the same program against itself will often lead to the same lines being explored over and over. To avoid this one might want to start the program(s) from positions a few moves into a game, or introduce a random factor for the opening few moves.

4.4 Alpha-beta search

As outlined above, a game playing program will apply the minimax algorithm making use of alpha-beta pruning. Rather than explore the game tree all the way down to the final positions it will stop at a pre-programmed depth. It will there use the value given by the evaluation function as an estimate for the real value, and otherwise use the algorithm as described in Section 3.2. You can find a nice illustration of the number of moves a program might search by going to `http://turbulence.org/spotlight/thinking/chess.html` (you must have Java enabled in your browser for this to work).

In this section we look at some variations of alpha-beta search.

Iterative deepening. One of the problems with searching to a pre-defined depth is that time constraints may mean that not all moves can be explored. Also when applying alpha-beta search, the *order* in which moves are searched becomes vital—the better the first move, the more pruning can occur. That is why many programs first carry out shallow searches, deepening the level one by one. This sounds inefficient, but shallow searches are fast. Compared with an exhaustive search to a higher depth, it does not really amount to much. This technique is often combined with others which make use of the information gained from the shallow search. But, if nothing else, we can use this information to decide the order in which we consider moves in the alpha-beta search, and it ensures that we've got a decent move to make if we should run out of time while searching to a greater depth. And if we use a hash table as outlined above to keep track of positions already searched then any position we encounter on the board will already have been searched to some depth before we made

our previous move, so that sort of information should already be on hand, so that we don't have to start from scratch.

Modified alpha-beta search. When we do ordinary alpha-beta search as described in Section 3.2 we have no preconceived idea what the value of the root of the tree might be. As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for the value if the current node is a max node, this value is usually called α (the 'alpha' from 'alpha-beta');
- successively decreasing upper bounds for the value if the current node is a min node, this value is typically called β the 'beta' from 'alpha-beta').

As we descend into the tree we keep track of the current values of α and β by passing them down and updating them as appropriate.

- If the current node is a max node we only consider moves which lead to a value of at least α, because we know that we have found a move guaranteeing it, and thus are only interested in finding better ones. If we find a move with a better value we adjust α accordingly.

  If we find a value of above β then we have discovered a part of the tree that is irrelevant for our search, and we return to the parent without adjusting α or β.

- If the current node is a min node we only consider moves which lead to a value of at most β, because we know that we have found a move limiting us to this, and thus are only interested in finding better moves from the opposing player's point of view. If we find a move with a lower value we adjust β accordingly.

  If we find a value of below α we know that we have found a value which is irrelevant for our search, and we return to the parent without adjusting α or β.

It is worth mentioning at least that there is no need to program an 'alpha-beta for max nodes' and an 'alpha-beta for min nodes': By using the negative of existing values and exchanging α and β when moving down one level we can ensure that the same code works for both kinds of nodes.

Iteratively deepening search provides us with a provisional value, say v, for a position we want to search to a higher depth now. One can then *pretend* that one already has an upper and a lower bound for the possible score. We thus use a range from α to β with

$$\alpha \leq v \leq \beta$$

to achieve further pruning as follows. Carry out the alpha-beta search algorithm to the required depth, but on a max node

- only consider moves which lead to a value at least α (this allows more pruning)—this is as before, only then α was a value we could guarantee as a minimum;
- if you find a value w above β, stop the search and report w back.

On a min node

- only consider moves which lead to a value of at most β (again, this allows more pruning)—again this is as before, only then β was a value we could guarantee as being the maximum achievable;
- if you find a value w below α, stop the search and report w back.

Whereas before we could be sure that values above β (respective below α) resulted from descending into irrelevant parts of the tree this is no longer true with our guessed parameters, so we have to keep track of this.

The following cases may then arise

- The search returns a value in the given range from α to β. This will be the correct value.
- The search returns a value w larger than β. That means that our preliminary value was too pessimistic. We have to adjust our preliminary value to w, and might consider allowing a larger range. This is known as 'failing high'.
- The search returns a value w below α. That means that our preliminary value was overly optimistic. Again we have to start over with the adjusted preliminary value w in the place of v, and again we may want to allow a larger range. This is known as 'failing low'.

This technique is known as 'aspiration search'. In the very best case (where the best move is explored first, and the considered range always contains the correct value) the total size of the tree searched is reduced to $(\sqrt{b})^d$, where b is the branching factor of the tree and d is the depth. That means that using this algorithm, one can search *twice* as deeply in the same time (at least in the best case). This explains why such variants of alpha-beta pruning are employed in almost all commercially available game playing programs.

When combining this alpha-beta search algorithm with the hashing of positions as described on page 74 one has to be careful to store enough information for the hash table to be really useful.[4] It is typically a good idea to store the best approximation of a value so far, together with upper (alpha) and lower (beta) bounds, which may be useful when that position is revisited.

Move ordering. As stated above, in order for the alpha-beta search algorithm to perform at its best, that is to prune as often as possible, it is vital that the good moves are explored first. Of course, the whole point of alpha-beta search is to find good moves, but we can still use some of the clues we gain along the way to speed the process up as far as possible. For one, if we have searched a position before, even at a lower depth, we have some idea of which moves lead to good positions. (This kind of information can come either from a hash table or from employing an iteratively deepening search.) Secondly, we may have some ideas about which moves are typically good for the game in question (for example capturing moves in chess), and lastly we may have found a good move in a similar position (a sibling in the game tree) which may still be valid. Using these criteria, one should sort the available moves by expected quality, and then do the search in that order. And quite often (when pruning can be applied) it is good enough to just order the first few moves because the others may never be explored. Hence it makes sense to apply a sorting algorithm like SelectionSort or HeapSort which deliver the sorted items one by one. An obvious choice for a move to search first are known as 'killer moves'—moves which literally end the game. In chess these are captures of big pieces (in particular by small ones), checks and promotions.

When a good such sorting principle is in place and the best move is fairly often explored first, it can pay off to *reduce* the range for alpha-beta search on siblings of this expected best move. If this search fails, a normal search can still be applied. Since rather more positions are pruned when doing this, it can speed up the program considerably. This technique is known as 'PVS', short for 'principal variation search', because everything is compared against this principal variation (the first move searched). A related idea is discussed below.

Selective extension. Many game playing programs today do not search to a given depth no matter what the position is. Instead, they are selective about that and search to a greater depth whenever

- there is reason to believe that the current value for a position is inaccurate or
- when the current line of play is particularly important.

[4] In fact, one of my sources claims that the proper interaction between the two is not easy to achieve, and many bugs occur in this area.

A popular choice is to look at the deepest positions to be evaluated (that is, after reaching the current search horizon) and extend all lines which start by moves which are likely to change the evaluation of the position considerably. In chess such moves are capturing moves (and that is true for other games such as checkers). This is also known as 'quiescent search'. Alternatives are to extend the depth of search whenever the line of play under consideration contains a capturing move (this would be good enough to work for the example in Figure 4.3), or maybe a check. Clearly this has to be used in a limited fashion or the tree searched might expand hugely, possibly even infinitely, so some sort of termination mechanism has to be implemented. But at least this avoids making moves towards a position where, say, the opponent can capture our queen but our evaluation function thought that our pieces were placed well enough to give us an advantage over the opponent!

Many programs (for example Deep Blue, see the section on chess) apply an extended search to moves which they identify as the 'principal line', because they start with a move that is much better (on current knowledge) than its siblings, thus trying to ensure that when they choose this move no ugly surprises lurk below the horizon. There are also tricks when instead of increasing the depth of some lines, the depth is decreased on 'obviously bad' lines.

4.5 Further issues

Only looking a limited number of moves ahead can lead to some surprising problems. We discuss two of them here and also provide ideas for solving them. We also look at other program components used in some commercial software.

Winning positions which don't lead to wins

One of the problems with alpha-beta search is a situation which seems almost paradox. When programmed naively, some winning positions may not lead to a win! The reason for this is not that 'the algorithm is wrong' but that it may need some encouragement to *force progress*. The reason for that is this: Say we give each winning position (that is, one we know we can win from) a value of 1000. When looking for the next move to make this will ensure that from a winning position we will always move to another winning position. That sounds quite good, but it's not good enough: It does not ensure that the move we have made leads to an actual win. Figure 4.2 gives a situation where this idea might fail.

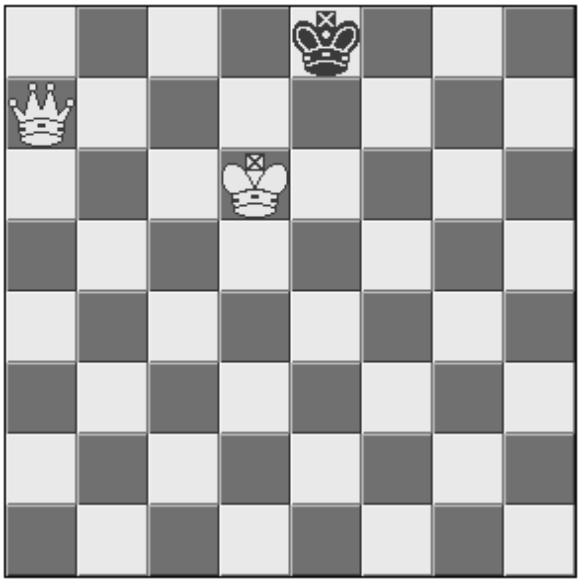


Figure 4.2: Another chess position

If White moves the king to $e6$ (one field to the right) then he is still in a winning position, with Black's only valid moves being to $d8$ and $f8$. Let's assume Black moves to $d8$. Then moving the king back to $d6$ again gives White a winning position. But if Black now moves back to $e8$, we are back where we started and our program might go into a loop. This will lead to a draw since there are rules about repeating the same position.

We can avoid falling into this trap by assigning a slightly adjusted value to a winning position, say 1000 minus the number of moves required to get to the win. Then alpha-beta search will indeed ensure that the program wins when it finds itself in a winning position.[5]

The horizon effect

One way of formulating the problem explained in the previous item is that while a win can be forced, it stays forever below the horizon (and the program is happy with moving to a position from where it can (still) win, because we have not told it otherwise). There is also the opposite effect.

A program as we have defined it so far is quite happy with bad moves *as long as the consequences lie below the current search horizon*. Clearly, there is no way the program can know about these consequences. We do get an effect when something 'bad' is about to happen, say the opponent might capture one of the program's pieces. Then the program will often try to avoid this capture (which might be inevitable) and thus will play a sequence of pointless moves to keep the event so long that it moves below the horizon, and so effectively can't be seen by the program.

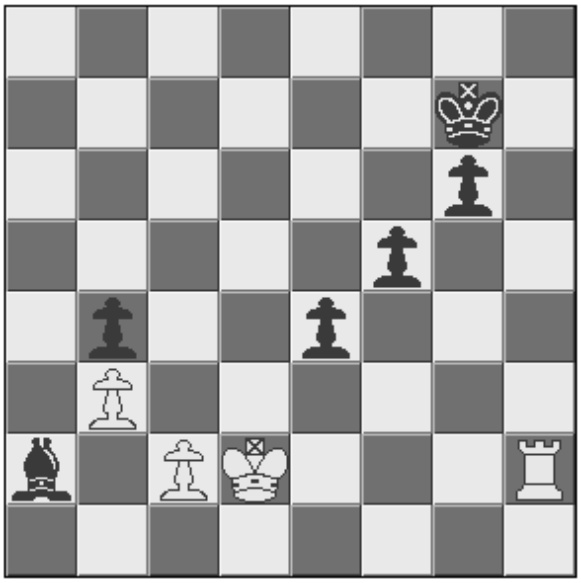


Figure 4.3: Yet another chess position

In the situation depicted in Figure 4.3 the black bishop is trapped by the white pawns. Eventually it will be taken, for example by the white rook moving from h2 to h1, a1 and finally a2. This would therefore occur after six moves. Assume the program playing Black searches six moves ahead. An accepted 'good' line for Black in this situation is to trade off the bishop against a pawn by capturing the pawn on b3 and being taken by the pawn on c2. The three connected pawns then might be good enough to win or draw against the rook. A program searching six moves will, however, typically move the black pawn e4 forward to e3, checking the king, a move which requires White to move the king (possibly capturing the

[5]Clearly there are some programs, such as othello, where this is not required since this game ends after at most 60 moves.

attacking pawn). That delays the taking of the bishop long enough so that the program can't see it any longer, and it thinks the bishop is safe. A program might thus throw away all its pawns to delay what is inevitable—setting itself up for a loss as a result.

There are several ways of trying to overcome this problem. One might try adding knowledge to the program so that it can detect when one of its pieces is trapped. This is typically rather difficult to achieve. Another way is to try and increase the overall depth of search in the hope that pushing back the horizon will make this sort of situation less likely. But probably the best tactics to employ is to instead selectively search deeper in situations like this, where a piece (like the pawn here) is being sacrificed. We discuss this as the next item.

Additional program components

Apart from the components introduced above many commercially available game-playing programs make use of libraries of some kind so that some moves can be read off rather than having to be calculated.

Opening libraries. In particular chess playing programs will contain libraries that know about the typical sequence of opening moves which have been found over time to be strong. They only begin applying alpha-beta search after the first few moves.

Well-known sequences. Some games allow us to identify situations that are likely to recur in a number of games, such as *joseki* in go. Some programs contain software that recognizes such situations, and a library regarding which moves to play for the next few moves.

Endgame. Once the game comes towards the end the situation is somewhat simplified since there are typically fewer moves available. Some games such as chess have *endgame libraries*. It is known, for example, that a king and a queen can force a win against a king and a rook, and how to go about winning in such a situation. One can put this information into a library for the program to use. Alternatively, in the last stages of the game the game tree typically has a much smaller *branching factor*, that is, number of actions at any one node. It might be possible to search the full game tree once the endgame has been reached.

4.6 The history of chess programs

In this section we look at chess as an example to show how game playing programs developed. Originally it was thought that one might be able to write programs that mimic human decision making, but it turns out that most of the improvement in game playing programs over time has come *through the ability to search through more positions in the game tree.* This history also illustrates how many of the ideas introduced in previous sections were developed.

Of all the game playing programs those playing chess have had the most time and man power invested in them, and as a result they are the most sophisticated. Just how sophisticated up-to-date programs currently are is described below. In the remainder of this section, when we talk about 'level of search', or the like, we typically count a move of the program followed by a move of the opponent as one level. chess players speak of a 'ply' when they mean what we call a move—for them a move consists of a move by White followed by one by Black.

In 1950 Claude Shannon (probably known best for his contributions to information theory) described principles for a program that could play chess. He suggested that each position should have a value (or score), to be calculated from the number of the various pieces (that is our 'material' criterion), each with an appropriate weight (so that a pawn is worth less than a bishop, say), their mobility and with special values for good 'pawn formations'. The program was then to search the game tree, and Shannon suggested two alternatives:

- The program might search to a given depth, the same everywhere; and he called that the 'fixed depth' method.

- The program might search to a variable depth depending on the 'type' of a position. Thus it should decide that if a move was 'obviously bad' there was no point in searching the game tree below it. There would have to be some notion of what he called 'stability' to decide at which step to stop. He called this the 'variable depth' method.

The program was then to apply a depth-first minimax algorithm to adjust the value of some given position. As discussed above it clearly is not realistic to wait for a program to calculate the *actual* value of some position, hence the need to fix a depth for the search in some way. His outline is still the basis for most game playing programs, although employing alpha-beta pruning is typically applied to increase speed. Shannon advertised this idea for one to find some application for computers, but also to gain insights into playing games, and thus into making intelligent decisions.

In 1951, Alan Turing[6] created the first algorithm for computer chess here in Manchester. Turing had worked at Bletchley Park during the war and where he was a central figure in the breaking of the German Enigma codes; computing machines were used there in order to try many different combinations quickly, thus helping with the decoding of messages. In the thirties he wrote a paper introducing the first formal notion of 'computability' with the help of *Turing machines* which were named after him. He originally came to Manchester to help build the first computer here, the 'Baby', but was actually a member of the Maths Department. His chess algorithm was not very good, searching to a low depth and with a very crude evaluation function. It was meant to be carried out by hand, and he did, in fact, play games employing this algorithm—making Manchester the place where the first chess playing program was designed and 'executed'.

Soon after that the first 'real' chess programs appeared, and in 1966 the first match between a Soviet and a US American program took place, the former winning 3 to 1. The rules for the match gave a time-out of 1 hour for every 20 moves, where each player was allowed to bring 'saved time' forward into the next 20 moves. That meant that searches had to be cut off, and typically a program would allow 3 minutes per move, that is, the same time for all moves (without trying to take into account how complicated the current situation was). The decision of which moves to explore first (and thus which moves might not be explored at all) was fairly random.

In 1974 the first world computer chess championships took place, which were repeated every three years thereafter. By the late eighties, the following improvements had been made.

- In order to avoid exploring the same position twice programs employed hash tables to remember which positions had already been searched, and what value had been discovered for them. This technique is known as employing *transposition tables*. However, programs typically failed to update this value if in the course of the game a different one was found.

- Most programs had *opening libraries* so that at the beginning, programs would just follow 'approved opening lines' (and thus not make catastrophic moves early on).[7]

- Programs employed hash tables to store 'tricky positions' for future play, thus implementing some sort of 'learning'.

- Rather than doing a pure depth-first search for the next move, many programs switched to 'iteratively deepening search'. A shallow search is carried out for *all* possible moves, and the result might be used to decide in which order to explore these moves to the desired depth. No pruning should be carried out at this point, since moves which might look bad if one only looks one step ahead (such as sacrifices) might become very good

[6] See A. Hodges, Alan Turing: The Enigma, Vintage, 1992, or visit `http://www.turing.org.uk/turing/`.

[7] It was much later that *endgame libraries* were used for the first time. When both those are employed the really interesting part of a chess program is how it does in the middle game—and that's also the most interesting part of a game between human players.

moves if the game tree is explored to a higher depth. Many of the early programs fell into this trap of avoiding to lose pieces at all cost (since such moves were classified as 'obviously bad'), whereas they are an integral part of playing chess.

- Instead of searching to a given depth for all positions attempts were made to make this more dynamic, depending on whether the current situation is 'tricky' or straightforward. In practice this is achieved by setting time-limits (low for 'simple' positions, high for 'tricky' ones), and then carrying out complete searches of iterative depths (first all moves on level 1, then all moves on level 2, and so on) until time runs out. Then the best move found during that time is made.

Some of the techniques learned over time were useful for other areas of computing, for example the 'iteratively deepening search' technique is successfully applied in automated theorem proving. In fact, one of the reasons for putting that much effort into writing chess playing programs was that the lessons learned would be applicable to other situations where a space had to be searched.

In the seventies, when Artificial Intelligence was assumed to be 'just around the corner' (at least by some people), chess playing programs were taken to provide a prime example of what might be done using machine learning, and a lot of research went into that. The only contribution to chess programming made by this approach was its use in solving various endgames in the late seventies and early eighties. The real power of chess programs consists in the speed with which they can search the game tree. While doing this cleverly with good evaluation functions requires some real knowledge, everything else is raw computing power!

The first 'serious' man-computer match occurred in 1978—man won. In the late eighties, AT&T developed the first 'chess circuitry' which lead to a program (called Deep Thought[8]) which dominated the computer chess scene for years. As a result of its special circuits it could generate moves very quickly and so search more positions in a given time frame than other programs.

A further development in the eighties was the inclusion of entire opening databases into programs, thus making whole books of openings available to the computer. Similarly endgame databases were developed, and all five piece endgames were solved in that period. Again, the computer's raw power allowed the finding of solutions which had eluded people so far. As a result, the official chess rules were changed—the previous version did not allow for some of the checkmates to be played out (due to a restriction in the number of moves the players were allowed to make without the capture of a piece or a 'check' occurring).

To illustrate the development of chess playing programs, Figure 4.4[9] shows the number of positions examined in three minutes by a program *versus* its rating according to the chess rating system. For a computer, this is a 'brute force' chart connecting playing ability and computing power. Note the logarithmic scale along the horizontal axis! Certainly humans look at far fewer positions as machines. They have a much better way of weeding out 'obviously bad moves', and research has shown that very good players do recognize patterns of some form which allow them to only consider a few candidates for the next move to make. We do not currently have any abstract way of describing such patterns, and chess players are not really aware of *how* they do what they do. Understanding this would certainly do a lot to advance a number of different subjects.

Another interesting question regarding Figure 4.4 is that of where exactly 'perfect play' would come in. Clearly machines have improved a lot over time, but how far off is it? And does the curve suggested really approach it, or does it have a lower limit?

Another issue worth examining is how much difference it makes to increase the search depth, that is the number of moves considered before making a decision. Figure 4.5 shows the results of playing the same program (Belle) against itself, using different depths to which alpha-beta search is carried out. The results are shown as 'won games out of 20', and are

[8] Anybody want to guess more obscure references?

[9] All tables and figures in the remainder of this section are taken from Newborn's book, see sources.

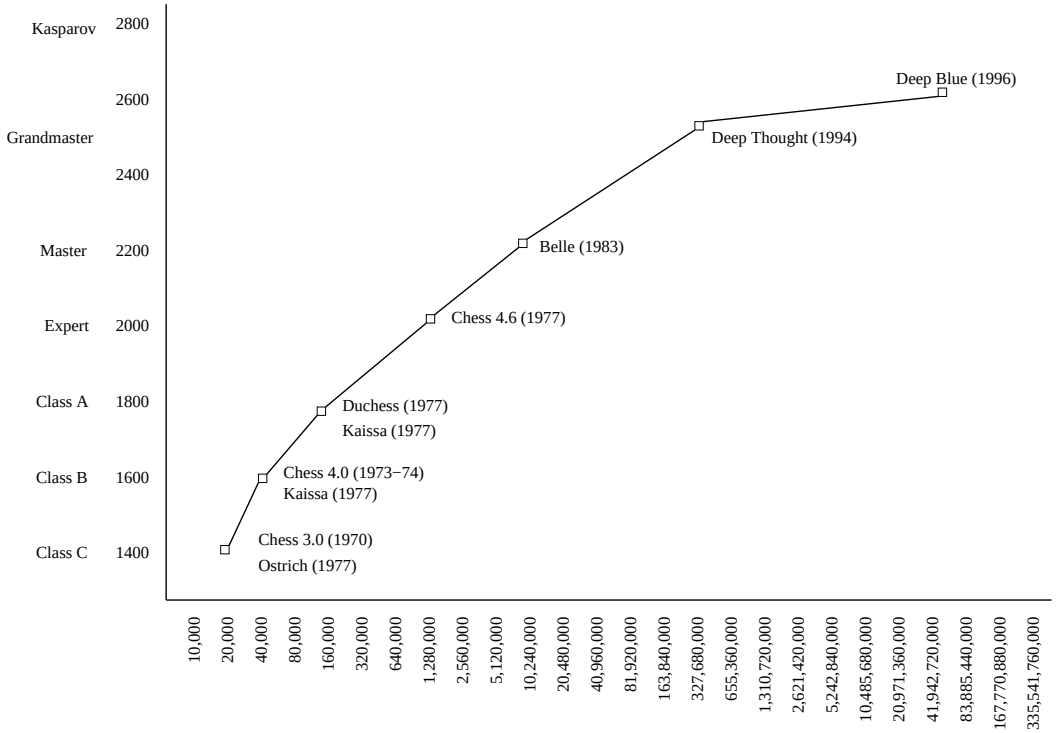


Figure 4.4: The level of play of programs

based on two experiments carried out by Thompson, the creator of the Belle program, in the late 70s.

	3	4	5	6	7	8	rating
3		4					1091
4	16		5.5				1332
5		14.5		4.5			1500
6			15.5		2.5		1714
7				17.5		3.5	2052
8					16.5		2320

	4	5	6	7	8	9	rating
4		5	.5	0	0	0	1235
5	15		3.5	3	.5	0	1570
6	19.5	16.5		4	1.5	1.5	1826
7	20	17	16		5	4	2031
8	20	19.5	18.5	15		5.5	2208
9	20	20	18.5	16	14.5		2328

Figure 4.5: Two experiments studying the effect of increasing depth

Figure 4.5 shows quite convincingly what a big difference searching three or four levels more can make—that of totally outclassing the opponent. The one constraining factor for searching to greater and greater depth is time, and computer programs have made great strides not only because a lot of time has been spent on making them better but because today's machines are so much faster and thus allow searching to greater depth. Of course, every bit of time gained by being 'clever' (for example, early on abandoning search on hopeless moves) can then be spent on going even deeper on promising moves.

The other interesting observation to be made regarding Figure 4.5 is the relation between

depth of search and the program's rating. While Figure 4.4 seems to suggest a linear connection between depth of search (logarithmic in number of positions searched means roughly linear in depth) and rating until about 2500, Thompson's experiments suggest that linearity only applies to level 2000. His data appears somewhat more convincing since no other improvements than depth of search have been made in his case, whereas Figure 4.4 compares very different programs with each other. One might tentatively state that the improvement beyond that which might be expected from adding depth to the search must result from other improvements made. After all, 25 years separate the early programs in Figure 4.4 from Deep Blue.

An alternative way of measuring improvement when increasing the search of depth is to compare how often the more thorough (and more expensive) search actually pays off. The following table attempts to measure that by looking at the number of times searching to a higher depth resulted in a different move being picked.

level	percentage of moves picked different from predecessor	approximate rating
4	33.1	1300
5	33.1	1570
6	27.7	1796
7	29.5	2037
8	26.0	2249
9	22.6	2433
10	17.7	2577
11	18.1	2725

Hence different moves are indeed picked quite frequently when searching to a higher depth, which presumably explains the results in Figure 4.5. The table also shows that the return of this is diminishing with increasing depth.

The late eighties saw the first time that Grandmasters[10] were beaten by programs, and in tournaments rather than in display matches. The first program to be rated Grandmaster was Deep Thought. But playing against the world champion in 1989 it was defeated in only 41 moves. In 1993 it managed to beat the youngest Grandmaster ever, Judit Polgar.

The main development from the late eighties onwards is the development of more specialized hardware to speed up the generation of moves, with considerable use of parallel machines. Programs that have been build to play at the highest level are therefore increasingly hardware dependent and cannot run on other machines. Computers participate increasingly in tournaments, in particular in the US (for example state championships). The following table gives some idea of the hardware and computing power thrown at this problem over time.

Name	Year	Description
Ostrich	1981	5-processor Data General system
Ostrich	1982	8-processor Data General system
Cray Blitz	1983	2-processor Cray XMP
Cray Blitz	1984	4-processor Cray XMP
Sun Phoenix	1986	Network of 20 VAXs and Suns
chess Challenger	1986	20 8086 microprocessors
Waycool[11]	1986	64-processor N/Cube system
Waycool	1988	256-processor N/Cube system
Deep Thought	1989	3 2-processor VLSI chess circuits
Star Tech	1993	512-processor Connection Machine
Star Socrates	1995	1,824-processor Intel Paragon
Zugzwang	1995	96-processor GC-Powerplus distributed system (based on the PowerPC)
Deep Blue	1996	32-processor IBM RS/6000 SP with 6 VLSI chess circuits per processor

[10]Bent Larsen, beaten by Deep Thought.

Until the early nineties, writing chess programs on this level was entirely an academic effort, and commercial programs available for sale typically played on a much weaker level.[12]

It was only then that IBM entered the scene and created Deep Blue, probably the best-known chess program, based on Deep Thought. In 1996 a 6-game match was arranged between it and the reigning world champion, Gary Kasparov. Deep Blue won the first game, but lost the match 2 to 4. In 1997 a rematch occurred which was won by the machine 3.5 to 2.5. Kasparov made a mistake in the deciding match, leading to his loss of the series. You may therefore be surprised to find Deep Blue ranking below Kasparov in Figure 4.4. The reason for this is that Deep Blue was very much fine-tuned to play against this one opponent. It had whole books on lines which Kasparov liked to play, and others on lines which Kasparov was known to avoid (in the hope that he did not know them well enough to play that strongly when forced into them by Deep Blue). Arguably the human player learns from playing against the machine and can then exploit its weaknesses. However, Deep Blue had access to hundreds of games that Kasparov had played, whereas the creators of the program were very reluctant to let him have access to games it had played prior to the match. Wikipedia has a nice entry on Deep Blue.

In summary the history of chess programs shows that currently, game programming is not really about mimicking the way human beings reason and make decisions. Instead it became a case study in applying the speed at which computers can carry out instructions to searching a given space. In particular it has shown us something about the relation between greater depth of search and reaching better results. As far as other games are concerned: In 1982 a program called IAGO was assessed as playing othello (also known as reversi) at world championship level, but didn't take part in any tournaments. In 1994 a program called Chinook became world checkers champion, but the reigning world champion had to forfeit the match due to illness. Go playing programs currently are many levels below even good amateurs, let alone professionals. The people who created Chinook solved the game in 2007, using the program: Either side can enforce a draw. Go to `http://webdocs.cs.ualberta.ca/ chinook/` for a lot of information on it.

[11] This serves to show that this expression goes quite some way back!

[12] Presumably it would be harder to sell a program which requires its own hardware, and expensive such a thing would be too.

Summary of Chapter 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search.
- The program carries out alpha-beta search to a given depth and then uses the evaluation function for the nodes at that depth to get an approximate value which it uses as if it were the actual value to determine which move to make. Because of the use of approximate values this move may not be the best one available.
- All considerations are overshadowed by the *need for speed*.
- Board representations should make the generation of moves, doing and undoing them fast.
- Evaluation functions require knowledge about the game in question. They are an attempt to assign a value to a board position from just what is on the board, without further descending into the game tree.
- Alpha-beta search is concerned with assigning a value to a position by searching the game tree below it and eventually applying the evaluation function. Searching to greater depth will result in a better program, so any gain in speed goes into searching to a greater depth. There are many tricks one might try to employ in order to concentrate on searching the relevant parts of the game tree; in particular ordering moves to search the most promising ones first.
- Most effort so far has gone into creating chess-playing programs. They have profited from faster hardware, and many improvements have been made which are very chess-specific: better heuristics, opening and endgame libraries, and the like.

Sources for this chapter

The material in this section has been compiled from the following.

David Eppstein's notes on *Strategy and board game programming* at the University of California at Irvine, `http://www1.ics.uci.edu/~eppstein/180a/w99.html`.

A.N. Walker's notes for his course on *Game Theory* at the University of Nottingham, available at `http://www.maths.nott.ac.uk/personal/anw/G13GAM/`.

M. Newborn. **Kasparov versus Deep Blue: computer chess comes of age.** *Springer*, 1997.

The *Scientific American*'s account of the second match between Kasparov and Deep Blue at `http://www.sciam.com/explorations/042197chess/`.

IBM's account of the same match at `http://www.chess.ibm.com/`.

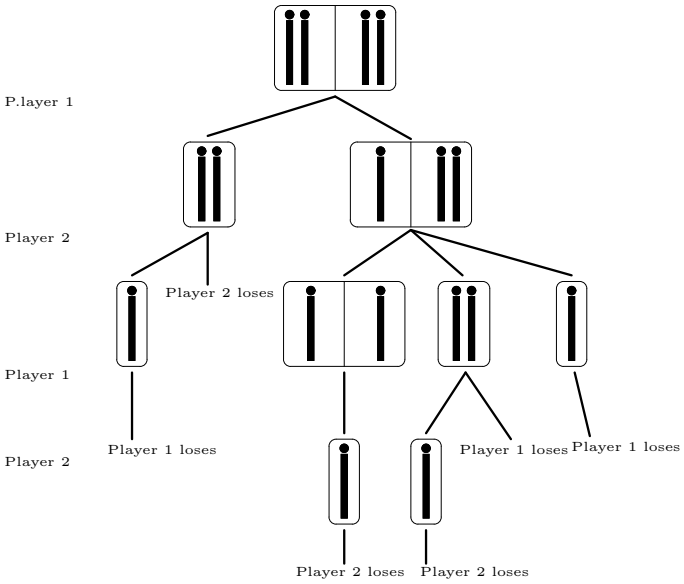
D. Levy and M. Newborn. **How Computers Play chess.** *Computer Science Press*, 1991.

The Chinook webpage (about checkers) at `http://www.cs.ualberta.ca/~chinook/`. This includes a link to the paper on the solving checkers.

Solutions for the exercises

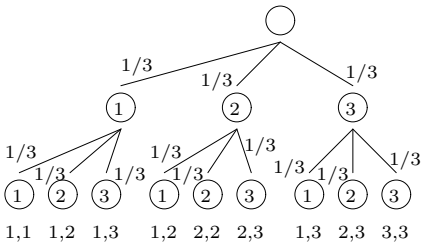
Note. The exercises in the notes are there to make sure you have understood the concepts as they are introduced. None of the exercises are particularly complex. Examples for the various tasks that need to be solved are given in the notes. Hence these model answers do not attempt to explain in detail how to solve an exercise but they provide a solution you can compare yours to.

Exercise 1. (a) Here is the game tree, pared down by symmetry considerations.



(b) The game tree is similar to Figure 2, just starting with a larger bar of chocolate. It is given in two parts in the solution to Exercise 25.

Exercise 2. (a) Here is the tree that corresponds to throwing two dice which show the numbers 1, 2 and 3 each with the same probability. At each leaf we give the two numbers thrown. The probability for each of the leaves being reached is $1/3 \times 1/3 = 1/9$. The



probability for throwing a 2 and a 3, for example, is the sum of all the probabilities of reaching a leaf labelled accordingly, that is

$$1/9 + 1/9 = 2/9,$$

because this label occurs for two leaves. The overall probabilities are summarized in the following table.

1,1	1,2	1,3	2,2	2,3	3,3
1/9	2/9	2/9	1/9	2/9	1/9

(b) The game tree is given in Figure 4.6.

There are 3 different cards that Player 1 may get, which leaves 2 possibilities for Player 2. All in all there are therefore 3×2 draws. Player 1 has the higher card if he has the K, which

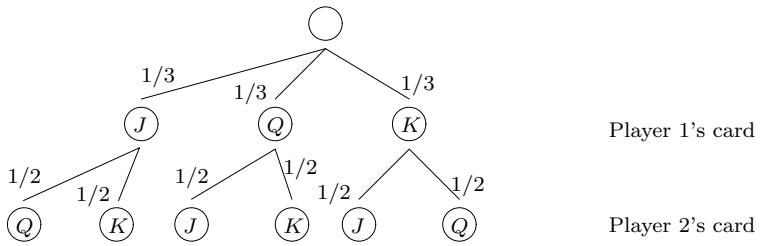
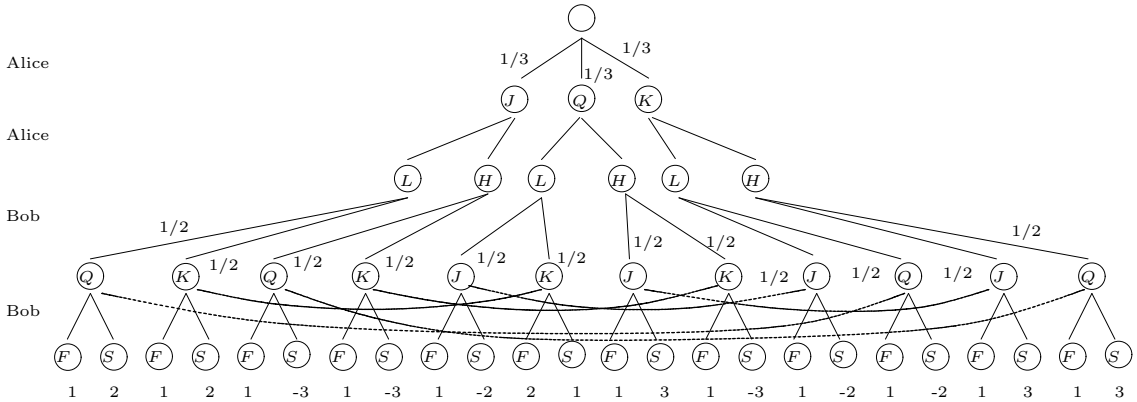


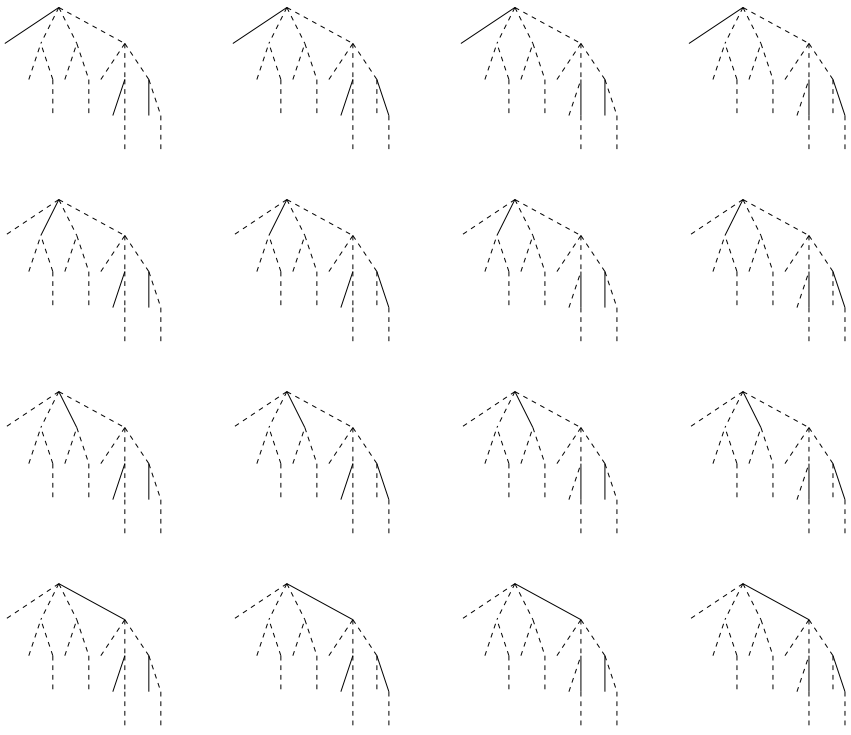
Figure 4.6: Dealing cards

occurs in two draws, and if he has the Q while Player 2 has the J. Hence in three of the draws Player 1 has the higher card—this is half of them, as you should have expected. The probability that Player 2 has the Q is $1/3$, and that she has the K while Player 1 has the J is $1/6$, so she wins in $1/3 + 1/6 = (2+1)/6 = 3/6 = 1/2$ of all draws.

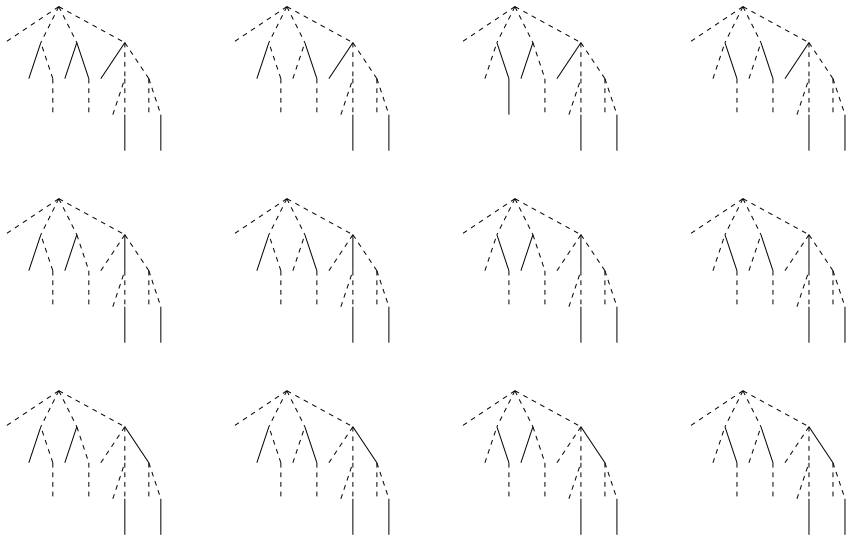
Exercise 3. Here is the game tree for this version of simplified poker. Nodes in the same information set are connected by dotted lines.



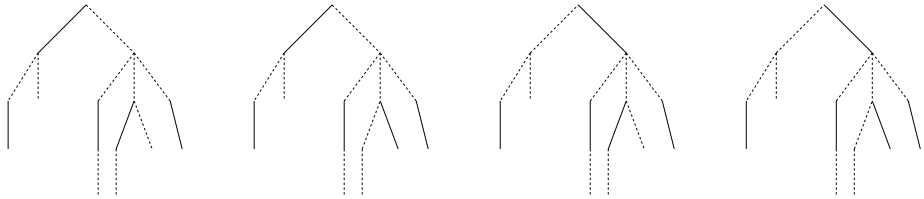
Exercise 4. (a) There are three decision points with more than one option for Player 1 in (2×2)-chomp (the root and the two decision points 'in round 2'). We have to combine all the possible choices with each other, which leads to $4 \times 2 \times 2$ fully specified strategies. They're all drawn in the following.



There are also three decision points with more than one option for Player 2, with two, two and three choices available respectively. Hence there are $2 \times 2 \times 3 = 12$ strategies for this player. Again we draw them.



(b) In $(2, 2)$-nim Player 1 has two decision points with more than one choice, they have two choices each. Hence there are four fully specified strategies for the player.

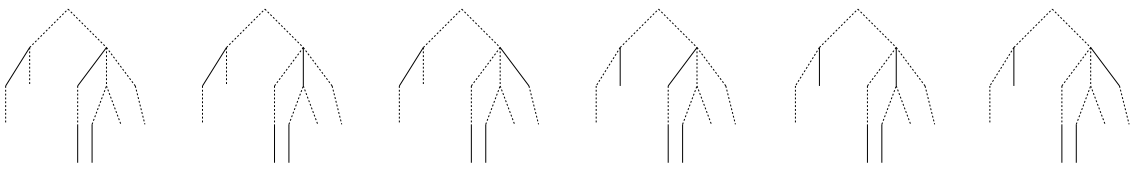


Alternatively one could describe the strategies verbally:

- For the first move, take two matches. If Player 2 takes one match in case we take one match for the first move, take one match on next move.
- For the first move, take two matches. If Player 2 takes one match in case we take one match for the first move, take two matches on next move.
- For the first move, take one match. If Player 2 takes one match on the next move, take one match on the next move.
- For the first move, take one match. If Player 2 takes one match on the next move, take two matches on the next move.

This formulation makes it obvious that the first two strategies are overspecified.

For Player 2 there are also two decision points which have more than one choice, one with two and one with three available options. This gives 6 strategies for the player.



Exercise 5. (a) Player 2 has

$$1 \times 2 \times 2 \times (1 + 1 + 1) = 2 \times 2 \times 3 = 12$$

strategies.

(b) The answer is given in Section 2.7 on simplified poker which starts on page 55.

Exercise 6. (a) For $(2,2)$-nim, compare the game tree in Exercise 1 (a), and the fully specified strategies in Exercise 4 (b). The first two fully specified strategies lead to the same behaviour and become just one strategy.

Alternatively we can look at the situation as follows: Player 1 has the choice of taking one or both matches from one of the two piles. If he takes two then he has no further choices to make as the game continues. If he takes just one match, then in case Player 2 chooses the middle move (which leaves one pile of two matches) in the next move Player 1 may choose between taking both matches or just one. We could encode his strategies as 2 (take 2 matches in the first move), $(1,1)$ (take one match in the first move, 1 match in the next move if there is a choice) and $(1,2)$ (take one match in the first move, two in the next move if the possibility arises).

For Player 2 the fully specified strategies are all different strategies since they do not specify play in situations that may not arise. To prepare the solution for the next exercise we explain our naming of strategies for Player 2. For Player 2, it all depends on which first move Player 1 makes. If Player 1 takes 2 matches in his first move, Player 2 may herself take one or two matches in her first move, and those are all the choices she ever makes. If Player 1 only takes one match, she may choose to take one match from the pile with 1 match, one match from the pile with 2 matches, or both matches from the pile with two matches. We can encode her choices as $(1|1(1))$, $(1|1(2))$, $(1|2(2))$, $(2|1(1))$, $(2|1(2))$, and $(2|2(2))$. Here before the $|$ we indicate what Player 2 will do if Player 1 takes two matches in his first move, and after the $|$ if Player 1 takes just one match in the first move. We indicate which pile to take a match from by (1) (the one with one match) and (2) (the one with two matches).

(b) Again, this is done in Section 3.6

Exercise 7. (a) We give a matrix with a pay-off of 1 or -1 for Player 1. For the names used for the various strategies see the previous exercise.

	$(1\|1(1))$	$(1\|1(2))$	$(1\|2(2))$	$(2\|1(1))$	$(2\|1(2))$	$(2\|2(2))$
2	-1	-1	-1	1	1	1
$(1,1)$	1	1	-1	1	1	-1
$(1,2)$	-1	1	-1	-1	1	-1

Note that $(1|2(2))$ is a winning strategy for Player 2: if she plays according to it, she will always win.

(b) We have Player 1 as the row player and Player 2 as the column player. The pay-off is given for Player 1, that for Player 2 is obtained by multiplying with -1.

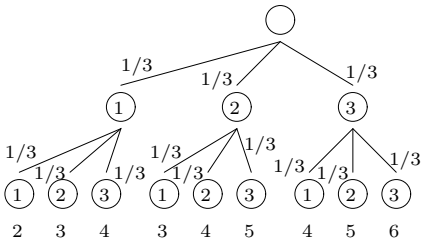
	H	T
H	-1	1
T	1	-1

Exercise 8. (a) For $(2,2)$-nim there are four fully specified strategies for Player 1 and six for Player 2. For Player 1, the strategy that takes 2 matches on the first move turns into two fully specified strategies. Using a similar notation as in the previous exercise we get the following pay-off matrix.

	$(1\|1(1))$	$(1\|1(2))$	$(1\|2(2))$	$(2\|1(1))$	$(2\|1(2))$	$(2\|2(2))$
$(2,1)$	-1	-1	-1	1	1	1
$(2,2)$	-1	-1	-1	1	1	1
$(1,1)$	1	1	-1	1	1	-1
$(1,2)$	-1	1	-1	-1	1	-1

(b) In this small and simple game the fully specified strategies are the same as the strategies, hence the normal form is as given for the previous exercise.

Exercise 9. (a) The game tree should be familiar. Here we give at each leaf the sum of the two numbers thrown. The probability for each of the leaves being reached is $1/3 \times 1/3 = 1/9$.



The probability for throwing a 5, for example, is the sum of all the probabilities of reaching a leaf labelled with a 5, that is

$$1/9 + 1/9 = 2/9,$$

because 5 occurs for two leaves. The overall probabilities are summarized in the following table.

2	3	4	5	6
1/9	2/9	3/9	2/9	1/9

The expected value is $2/9 + 6/9 + 12/9 + 10/9 + 6/9 = 36/9 = 4$.

(b) Here is the matrix for this game.

	$S\|S$	$S\|T$	$T\|S$	$T\|T$
	0	$-2/9$	$-4/9$	$-6/9$

Exercise 10. (a) The following table gives the pay-off function for the three players, $P1$, $P2$ and $P3$, where their strategies are given as 1 (bet one) and 2 (bet two).

	$P1$	$P2$	$P3$
$(1,1,1)$	1	1	1
$(1,1,2)$	2	2	5
$(1,2,1)$	2	5	2
$(1,2,2)$	5	2	2
$(2,1,1)$	5	2	2
$(2,1,2)$	2	5	2
$(2,2,1)$	2	2	5
$(2,2,2)$	1	1	1

(b) Figure 4.7 shows the game tree for this game.

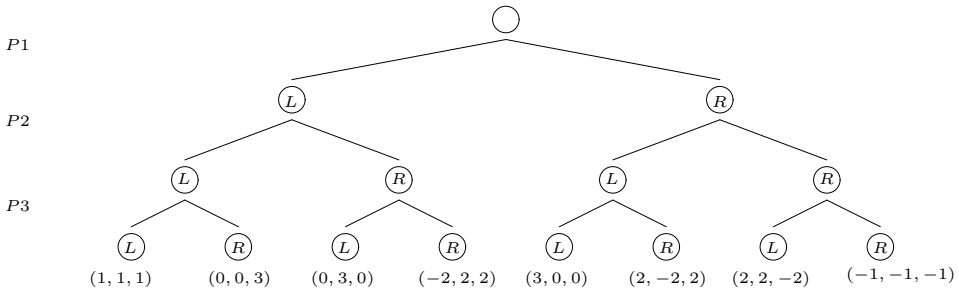


Figure 4.7: The left-right game

There are 2 strategies for Player 1, 4 for Player 2 and 16 for Player 3.

Exercise 11. (a) There are seven nodes in a full binary tree of height 3 which are not leaves, that is, where a decision is made. Clearly there are a number of ways of splitting those decision points between two players. We do not discuss them in detail.

(b) For height $n+1$: If $n+1$ is odd then the number of strategies for Player 1 is twice the number of strategies for Player 2 at height n. This is because the desired number of strategies can be calculated as follows: Player 1 has the choice between left and right at the (new) root of the tree, and thereafter he is either in the left or the right subtree. But the decision points there for him are precisely those Player 2 had in the game of height n. The number of strategies for Player 2 does not change because she still has the same number of decision points. If $n+1$ is even then the number of strategies for Player 1 is the same as at height n. But the number of strategies for Player 2 is the square of the number of strategies for Player 1 at height n. This is because for each of Player 1's choice of 'left' or 'right' she has to choose a strategy, one for the left and one for the right subtree. But her decision points there are those of Player 1 for a game of height n.

Exercise 12. (a) Player 1 has a winning strategy, the first move of which consists of eating one piece of chocolate. Now Player 2 has the choice of immediately eating the poisonous piece and losing, or eating either one of the two remaining non-poisonous pieces, leaving Player 1 to eat the other, and again requiring Player 2 to take the tainted one.

(b) Player 2 has a winning strategy. If Player 1 takes two matches on the first move, Player 2 may take one of the two remaining ones leaving Player 1 to take the last one and losing. If Player 1 takes just one match then Player 2 may take the pile of two matches, again leading to Player 1 having to pick the last match, and losing.

Exercise 13. A winning strategy will ensure the pay-off of 1 against every strategy employed by the other player. Without loss of generality we consider the case where Player 1 has a winning strategy (this means we don't have to mess around with indices). Hence for such a strategy s^* it is the case that for all strategies t for Player 2 we have

$$p_1(s^*, t) = 1.$$

But this is the highest pay-off possible for Player 1, so

$$p_1(s^*, t) = 1 \geq p_1(s, t)$$

for all strategies s for Player 1 and t for Player 2.

Exercise 14. (a) This game has no equilibrium point.

(b) In this game, both $(1, 1)$ and $(2, 2)$ are equilibrium points.

Exercise 15. (a) The equilibria are $(1, 2)$ and $(2, 1)$. Player 1 prefers the former (because it gives him the higher pay-off of 10 over 8) while Player 2 prefers the latter (because it gives her the higher pay-off of 8 over 6). But if Player 2 chooses her strategy 1 to aim for her preferred equilibrium point then if Player 2 chooses his strategy 1 to achieve *his* preferred equilibrium point she will get a pay-off of -300. It seems therefore much more prudent for her to 'play it safe' by choosing her strategy 2.

(b) The two equilibrium points are $(1, 2)$ and $(2, 1)$ where both players prefer the latter. This time it seems safe for Player 2 to choose her strategy 1 since it is unlikely that Player 1 will choose his strategy 1 which would lead to the worst case pay-off for her. If they are both rational (as we assume) they should choose this equilibrium point.

Exercise 16. (a) We have to go through each entry in the table, and check whether one of the three players changing their mind leads to an improvement of their pay-off. Here are two representative examples:

- Assume the players have chosen strategies $(1,1,1)$. If Player 1 changes his mind, they move to $(2,1,1)$, and Player 1's pay-off goes from 1 to 2, so Player 1 can improve his pay-off by unilaterally changing strategies, and this is not an equilibrium point.
- Assume the players have chosen strategies $(1,1,2)$.
  - If Player 1 changes his strategy they move to $(2,1,2)$, and his pay-off stays at 2.
  - If Player 2 changes his strategy they move to $(1,2,2)$, and his pay-off stays at 2.
  - If Player 3 changes his strategy htey move to $(1,1,1)$ and his pay-off goes down from 5 to 1.

  Hence this is an equilibrium point.

There are 6 equilibrium points, the only exceptions being $(1,1,1)$ and $(2,2,2)$.

(b) The mechanics are much the same as in the previous exercise. The equilibrium points are $(1,2,1)$, $(1,2,2)$. Note that neither of them is advantageous for Player 3!

Exercise 17. The game tree for this game is given above, see Exercise 10. When examining the various possibilities, we find (by inspection) that $(-1,-1,-1)$ is the only pay-off that leads to an equilibrium point (for all others, one of the players can improve his pay-off by changing his move). To reach this, Player 1 has to choose his strategy R. But for Player 2, any strategy that chooses R provided that Player 1 chose R will lead to this pay-off; it does not matter what Player 2 might do in the case where Player 1 chooses his strategy L. There are 2 such strategies which could be encoded as $(L|R)$ and $(R|R)$. Finally, for Player 3 it is sufficient that he chooses R provided that Players 1 and 2 have done so, and it does not matter in the least what he will do for his other 3 decision points. Hence there are $2^3 = 8$ such strategies for Player 3.

Exercise 18. (a) The equilibrium points are $(2,3)$, $(2,4)$, $(3,3)$ and $(3,4)$.

(b) The sole equilibrium point is $(3,4)$.

Exercise 19. (a) We collect the minimum of each row, and maximum of each column, in the following table.

4	3	1	1	1
3	2	2	2	2
4	4	2	2	2
3	3	1	2	1
4	4	2	2	$2\backslash 2$

So we find that $\max_{1\le i\le 4}\min_{1\le j\le 4} a_{i,j} = 2 = \min_{1\le j\le 4}\max_{1\le i\le 4} a_{i,j}$.

(b) We find that $\max_{1\le i\le 4}\min_{1\le j\le 4} a_{i,j} = 2$ but $\min_{1\le j\le 4}\max_{1\le i\le 4} a_{i,j} = 3$.

Exercise 20. It is important to note that 'the highest pay-off Player 1 can guarantee for himself' is $\max_i \min_j a_{i,j}$ while 'the highest pay-off Player 2 can ensure for herself' is $\min_j \max_i a_{i,j}$, where i ranges over the strategies for Player 1 and j over those for Player 2.

Hence the statement of the proposition amounts to a restating of Proposition 2.8.

Exercise 21. (a) If Player 1 chooses his mixed strategy $(1/2,0,0,1/2)$ his pay-offs against Player 2's pure strategies are as follows.

	1	2	3
$(1/2,0,0,1/2)$	$-1/2$	$-1/2$	$-1/2$

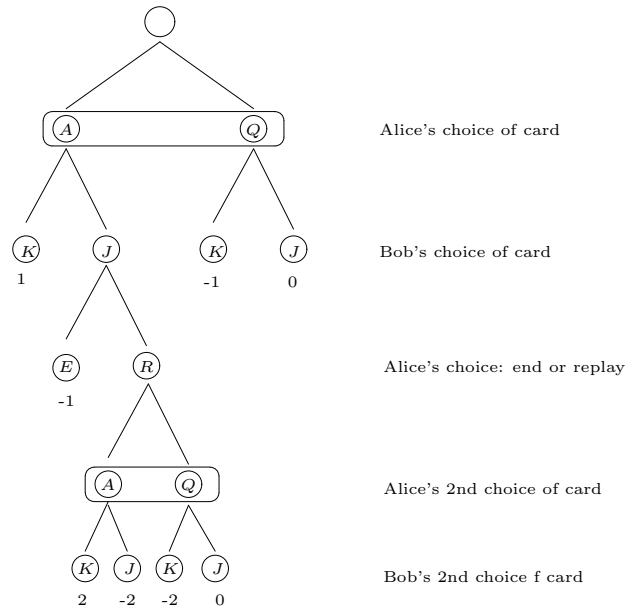
When he plays with that strategy against her mixed strategy $(1/4,1/4,1/2)$, his pay-off will be $-1/2$, which is at least as big as the numbers appearing in the above table.

When Player 2 matches her mixed strategy $(1/4,1/4,1/2)$ against Player 1's pure strategies the pay-offs are as follows.

	1	2	3	4
$(1/4, 1/4, 1/2)$	$-1/2$	$-1/2$	$-1/2$	$-1/2$

These pay-offs are at most as high as that of playing $(1/4, 1/4, 1/2)$ against the given mixed strategy for Player 1. Hence by Proposition 3.9 this is indeed an equilibrium point. Note that in the long run that means that Player 1 will lose 1/2 unit per game!

(b) The game tree is given below. Alice's strategies are in the first instance to choose either



A or Q. In the former case, she can either end the game (E) if Bob chose the J, or she can decide to go for a replay (R). In the latter case, she has to choose between A and Q again. Hence we can list her strategies as (A, E), (A, R, A), (A, R, Q) and (Q). Bob on the other hand has the choice between K and J as his first move. If he does choose the J he also has to decide what he is going to do if Alice should go for the replay. Hence his strategies can be listed as (K), (J, K), (J, J). We find the following game matrix

	K	J, K	J, J
A, E	1	-1	-1
A, R, A	1	2	-2
A, R, Q	1	-2	0
Q	-1	0	0

Showing that the given mixed strategies form an equilibrium point is done as in (a).

We find that

	A, E	A, R, A	A, R, Q	Q	$(0, 1/8, 1/4, 5/8)$
$(1/4, 1/4, 1/2)$	$-1/2$	$-1/4$	$-1/4$	$-1/4$	$-1/4$

All entries are below or equal to the final one as desired. We find for Player 1's mixed strategy that

	K	J, K	J, J
$(0, 1/8, 1/4, 5/8)$	$-1/4$	$-1/4$	$-1/4$

All these entries are bigger than or equal to the pay-off at the purported equilibrium point $-1/4$, so we have verified that it is indeed one. (So we were lucky with our choice of order of strategies.) Note that on average, Alice will lose 25 pence per game.

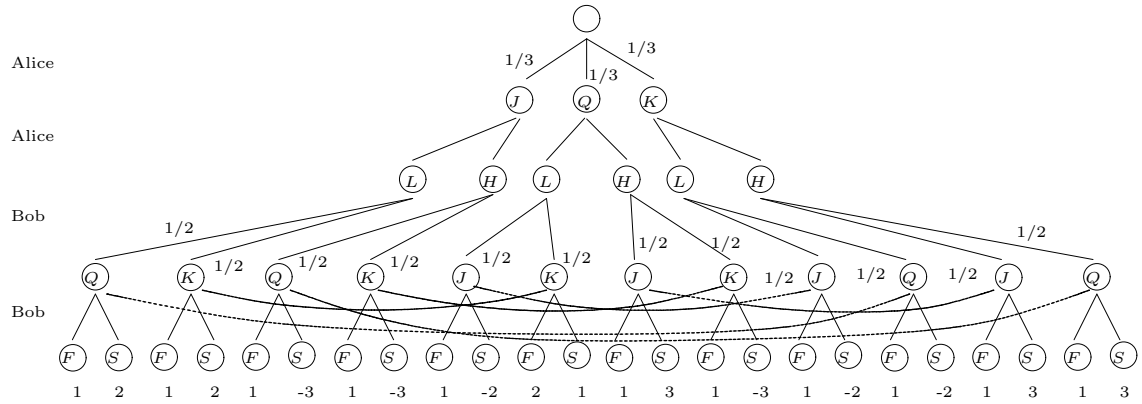
Exercise 22. (a) Strategy 2 for Player 1 dominates his Strategies 1 and 4 and Strategy 1 for Player 2 dominates her Strategy 2. That leaves the following matrix.

$$\begin{vmatrix} 4 & 2 & 6 \\ -2 & 4 & 2 \end{vmatrix}$$

The first strategy for Player 2 dominates the third one. That's as far as we can go here.

(b) From Player 2's point of view, strategy 4 dominates all the others, leaving us with a 4×1 matrix (the fourth column of the original matrix). It is now easy to read off that 2 is the value of the game and that its sole equilibrium point is $(3, 4)$.

Exercise 23. The game tree is given below. Note that nodes in the same information set are connected by dotted lines.



Alice gets one of three cards, and can choose to go L or H for each of those, giving her $2 \times 2 \times 2 = 8$ strategies. Bob knows that Alice has gone either L or H, and his own card (which multiply to 2×3 possible scenarios for which he needs a plan), and in each of these he can choose between F and S, giving him $2^6 = 64$ strategies. We can encode his strategies as triples where the first component gives his choice if he has the J, the second if he has the Q and the third if he has the K. For each of those, his choice will depend on whether Alice bet high or low. We represent such a choice as, for example, $S|F$, to mean S if Alice has gone L, and F if she has gone H.

We start eliminating strategies by looking at Bob's choices. If he has the J and Alice bet low, then he can only win the showdown, so he should opt for S. If, on the other hand, he has the J and Alice bet high he can only lose, so he should fold. If he has the K, the situation is dual: If Alice has bet low, he can only lose, so he should fold, but if she has bet high he can only win and should see. That leaves strategies of the form $(S|F, ?, F|S)$, of which there are four.

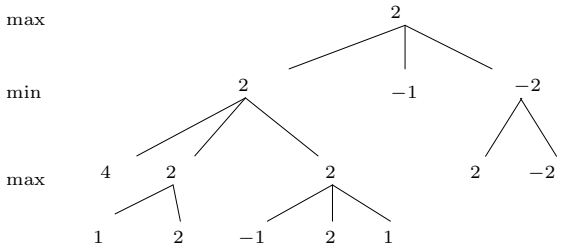
Alice, on the other hand, should bet low if she has the J, and high if she has the K. That leaves her only two strategies which will be of the form $(L, ?, H)$.

We now calculate the matrix for the game, where we only give Bob's choice in case he has the Q (in all other cases we have already determined what choice he will make). Because it is convenient we extract a factor of $1/6$ from each of the entries.

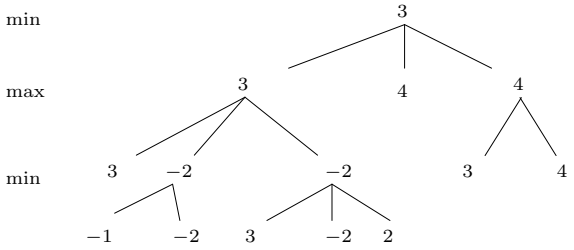
	$F\|F$	$F\|S$	$S\|F$	$S\|S$
(L, L, H)	3	5	4	6
(L, H, H)	2	4	3	5

We find that Alice's first strategy dominates her second, and that there is an equilibrium point at $(1, 1)$. The value of the game (here we need to bear in mind the factor of $1/6$ we dropped from the matrix!) is $3/6 = 1/2$.

Exercise 24. (a) The minimax algorithm for Player 1 leads to the following value:



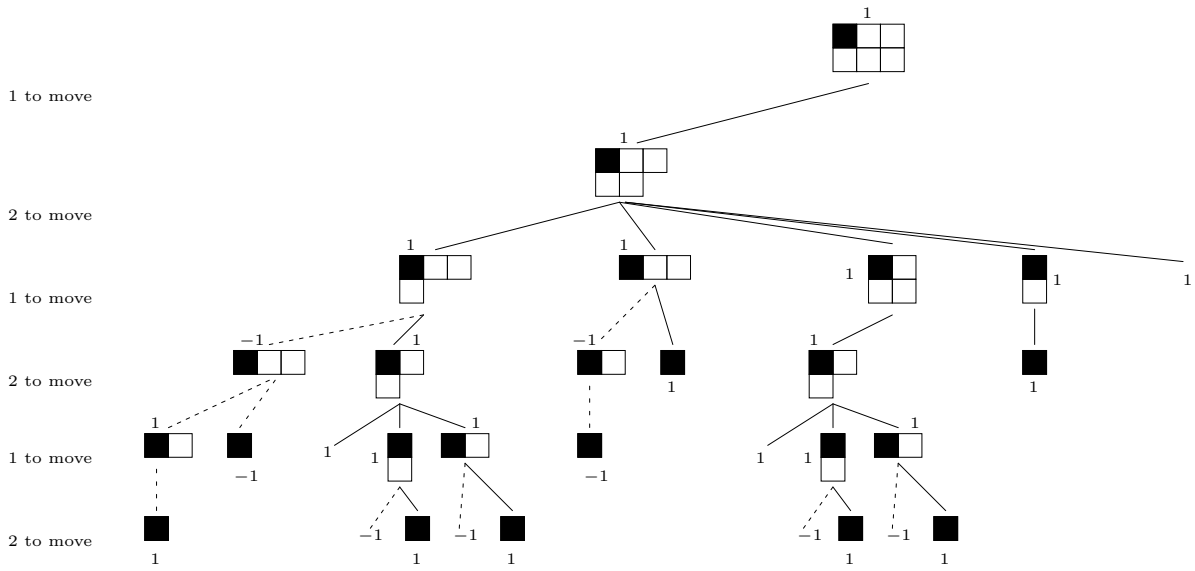
For Player 2 we get the following value:



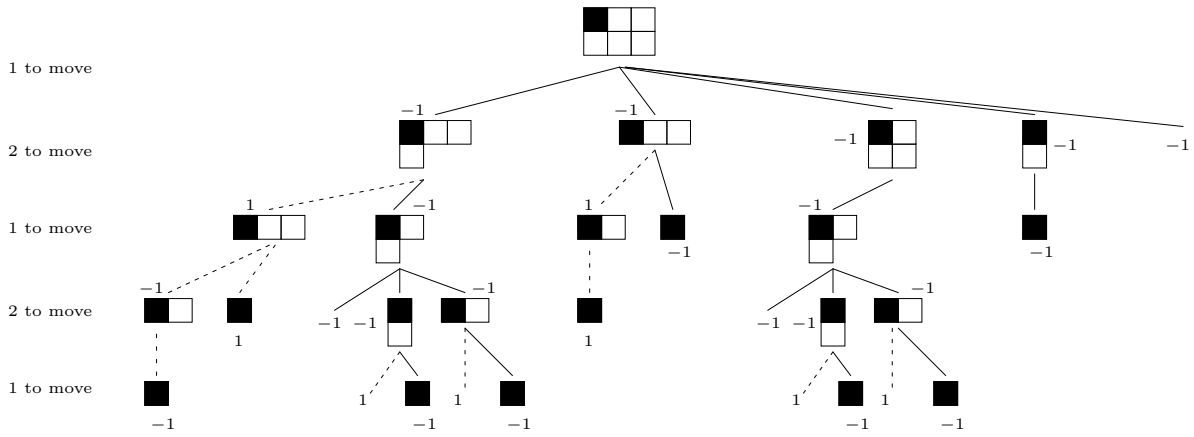
If the two players play these strategies against each other then they will go down the left-most branch of the tree, leading to a pay-off of 4 for Player 1 and 3 for Player 2. This is substantially better than the value of the game for Player 1, which is 2. Player 2, on the other hand, does not do better than her worst case analysis.

(b) The value for all three players is -1, and when they each play according to the strategy they found with the help of the minimax algorithm they end up with the equilibrium point outcome with a pay-off of -1 for each of them (see the solution to Exercise 17).

Exercise 25. (a) Player 1 has a winning strategy in this game which can be discovered using the minimax algorithm. It can be seen in the following excerpt from the game tree. (Note that in both cases some moves are omitted to keep the size down.)



If Player 1 picks any other opening move then Player 2 can force a win, as demonstrated below.



The game has been solved for arbitrary sizes of chocolate bars. I once set an extra difficult exam question (the students were warned that this would be a difficult question) challenging students to describe a winning strategy (and proving how that can be done) but that won't happen for this course. Descriptions of general solutions assuming varying amounts of mathematical theory are available online.

(b) Again Player 1 can force a win, by taking away an entire pile of three matches as his first move, leaving two piles with three matches each. Then

- if Player 2 takes three matches, take two matches from the only remaining pile;
- if Player 2 takes two matches, take the entire pile with three matches.
- if Player 2 takes one match, take one match from the pile with three matches, leaving $(2, 2)$-nim. We have seen already in Exercise 7 that the player going second (our Player 1) has a winning strategy in that game.

If he chooses any other first move then Player 2 can force a win. Can you see how to work out who has a winning strategy for nim of arbitrary size?