The complexity of Minesweeper and strategies for game playing

Kasper Pedersen

Department of Computer Science
University of Warwick
2003-2004

Supervisor: Dr Leslie A. Goldberg
Assessor: Dr Paul W. Goldberg

Abstract

To determine whether a Minesweeper configuration is consistent was proved **NP**-complete by Kaye. Here the complexity of Minesweeper is investigated and three game playing strategies are developed. The two decision problems: "Does a configuration have a unique solution?" and "Is a given move safe?" are proved complete in **DP**, and the problem of counting the number of solutions to a configuration complete in **#P**. Three Minesweeper strategies are presented, the best of which uses probability estimates to assist in guessing when required. This strategy is capable of winning 25% of games at expert level. The strategies are implemented in a framework developed specifically for automated Minesweeper playing.

Keywords: Minesweeper, complexity, **DP**-completeness, **#P**-completeness, game strategy development, probability estimation.

Author's assessment of the project

The following is an assessment of the project from the author's perspective.

- **Technical contribution of the project:** The project presents three complexity proofs about Minesweeper which further underpins the theoretical foundations of the game. Furthermore a technical description of game playing strategies is presented.
- **Relevance to Computer Science:** The project studies an instance of an **NP**-complete problem, and formulates related problems along with proving their relative complexity.
- **Use to others:** The theoretical results provide insight in what types of Minesweeper strategies are intractable, and the software is useful for implementing and analysing more strategies.
- **Reason for achievement:** The content of the project was technically challenging and the majority of the work presented is original. Furthermore, Minesweeper is not a commonly studied problem so only limited literature is available.
- **Weaknesses:** Several topics were covered so some aspects did not benefit from an in-depth enough analysis. It is felt that the project would benefit from a more detailed study of improving the probability estimation method used.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Minesweeper is a single player computer game which looks deceptively easy to play, but developing a method of winning every game is at least as hard as solving well-known computational problems like SAT and The Travelling Salesman. Richard Kaye of Birmingham University has recently proved that determining whether a Minesweeper configuration is consistent with the rules of the game is **NP**-complete [7]. This result means that the ability to play a perfect game of Minesweeper would solve an entire group of computationally hard problems along with one of the biggest open problems in contemporary mathematics, **P**=**NP**? This problem is one of the seven millennium problems that are intended to shape the direction of mathematical research in this century, and a solution to **P**=**NP**? would claim a $1000000 prize from the Clay Institute of Mathematics [17]. It is also exciting to know that a simple computer game like Minesweeper could hold the key to solving one of the most important open problems in modern mathematics and theoretical computer science.

This project uses Kaye's results as a platform for a study of some decision problems closely related to playing Minesweeper. Initially the configurations used by Kaye are studied and later new configurations, such as OR and XOR gates, are developed to suit specific needs in theorem proving such as ensuring that a configuration has a unique solution, a property not held by Kaye's AND

gate. The investigation of Minesweeper configurations and a study of Kaye's proof is presented in chapter 3.

The complexity of Minesweeper is explored in chapter 4, where it is proved that both attempting to identify a safe move on a configuration and to attempt determining whether a configuration has a unique solution are problems complete in the complexity class **DP**. Furthermore, the counting problem of finding the complete number of solutions to any Minesweeper configuration is proved to be **#P**-complete, a result which means that determining the exact probability of any square containing a mine is intractable.

Three Minesweeper strategies — single point, limited search, and limited search with probability estimation — are presented in chapter 5. The best strategy is limited search with probability estimation, which is capable of winning 92.5% of the games at beginner level, 67.7% at intermediate level and 25% at expert level. All the strategies have been developed independently of any other existing strategies, and only one strategy has been identified on the Internet that is capable of outperforming limited search with probability estimates on expert level. On both beginner and intermediate levels, no other strategy has been identified that perform better than both limited search and limited search with probability estimates.

It is noticeable that only two web pages have been identified concerned with automated Minesweeper playing and only one of those contain a framework for implementing strategies. Due to the lack of availability of Minesweeper frameworks capable of implementing an automated player, a Java application has been developed as part of this project to facilitate this. An abbreviated documentation of the software development is presented in chapter 6 along with a concise description of the API for implementing a Minesweeper strategy. Instructions for viewing the source code and executing the application are provided in appendix A.

Chapter 2

Background

2.1 Minesweeper

Minesweeper is a one-person board game played on a rectangular grid of size k by l. Let such a game be of size n where $n = kl$. Also choose m, such that $m < n$, to be the number of hidden mines on the grid. Initially all the squares on the grid are empty, and it is the aim of the game to uncover all the squares that do not contain a mine and mark (or leave blank) all the squares containing a mine. At each move the player must either choose an unlabeled square to 'probe' or mark a square as containing a mine. If the probed square contains a mine the game is over with a loss for the player; otherwise the number of mines immediately adjacent to it is revealed to the player in form of a number (0–8). This number will remain the label of the probed square for the remainder of the game. If the player uncovers the last square not containing a mine the game finishes with a win for the player.

2.2 The complexity hierarchy

Some background material on the complexity hierarchy is now presented. The initial topic of **NP** problems intentionally brief; a very readable introduction is found in Brassard & Bratley [2] and a more formal approach is

presented by Garey & Johnson [3], which also contains a very useful list of all known **NP**-complete problems known at the time of publication.

Definition 2.2.1 (Decision problem) *A decision problem is a computational problem requiring a yes/no answer.*

Traditionally, **NP** is the class of decision problems that can be solved in polynomial time using a non-deterministic algorithm,[1] but for the purposes of this project a more modern definition of **NP** is used. Informally, a problem is in **NP** if a succinct proof of membership exists and it can be verified in polynomial time. This definition is formalised as follows.

Definition 2.2.2 (NP) NP *is the class of decision problems X that admit a proof system $F \subseteq X \times Q$ such that there exists a polynomial $p(n)$ and a polynomial-time algorithm A such that*

- $\forall x \in X, \exists q \in Q.(x, q) \in F \wedge \mid q \mid \leq p(n)$*, where n is the size of x; and*
- $\forall (x, q) \in X \times Q$*, A can verify whether or not $(x, q) \in F$.*

A relatively small group of problems all belonging to **NP** is believed to contain the hardest problems in **NP**. This group of problems is called the **NP**-complete problems, and they have the property that if one problem can be solved in polynomial time then so can the rest. Each **NP**-complete problem can be reduced in polynomial time on a Turing machine to each other; such a reduction is known as a Turing reduction.

Definition 2.2.3 *A decision problem X is* **NP***-complete if*

- $X \in NP$*; and*
- $\forall Y \in \mathbf{NP}\ Y \leq_{\mathrm{T}}^{p} X$.

[1] See Hopcroft, Motwani & Ullman [4] for an introduction to the non-deterministic model of computation.

The original **NP**-complete problem is Boolean satisfiability (SAT) which asks: "Given a Boolean formula does it have a satisfying assignment?" This was proved **NP**-complete by Steven Cook in the early 1970's [2].

In order to prove **NP**-completeness of any decision problem it is thus enough to reduce a known **NP**-complete problem such as SAT to it as the following theorem shows.

Theorem 2.2.4 *Let X be an* **NP***-complete problem. Consider a decision problem $Z \in$* **NP** *such that $X \leq_{\mathrm{T}}^{p} Z$. Then Z is also* **NP***-complete.*

Proof. The proof is in Brassard & Bratley [2]. ■

In general, the complement of a decision problem in **NP** is in the complexity class **coNP**, which also contains complete problems. For example the decision problem "given a Boolean formula, does it have no satisfying assignments?" is the **coNP**-complete problem which complements SAT. The complexity class **coNP** thus contains problems which have a succinct disqualifier.

Definition 2.2.5 (coNP) *The set* **coNP** *contains those problems that have short disqualifications, i.e. a "no" instance of a problem in* **coNP** *possesses a short proof of its being a "no" instance; and only "no" instances have proofs. [12]*

A group of problems associated with counting problems rather than decision problems is also useful to consider.

Definition 2.2.6 (#P) *Let Q be a polynomially balanced, polynomial-time decidable binary relation. Its associated counting problem is as follows: Given x, how many y exists such that $(x, y) \in Q$? [12]*

Problems in **#P** ask questions about the number of solutions rather than merely the existence of a solution. The class also contains complete problems, however it is important to note that **#P**-complete problems are *much* harder

to solve than **NP**-complete problems. Another important issue to mention is for a problem to be complete in **#P**, the reduction from a known **#P**-complete problem has to be parsimonious, i.e. the number of solutions is preserved in the reduction. The class **#P** was introduced by Leslie Valiant in [19] and some **#P**-completeness results were presented in [20].

The final complexity class used in this project is the class **DP**, introduced by Christos Papadimitriou [11]. **DP** is primarily concerned with problems of determining whether a unique solution to a decision problem exists. **DP** is not a generally well-known complexity class, and it has been principally studied by Papadimitriou [12] although some special cases such as UNIQUE SAT are well documented in the literature [1]. **DP** is best defined as the intersection of two formal languages[2] as follows.

Definition 2.2.7 (DP) *A language L is in the class* **DP** *if and only if there are two languages $L_1 \in$* **NP** *and $L_2 \in$* **coNP** *such that $L_1 \cap L_2 = L$.*

It is important to note that in general $\mathbf{DP} \neq \mathbf{NP} \cap \mathbf{coNP}$, and that **DP** is a syntactic class hence containing *complete problems*, while the class $\mathbf{NP} \cap \mathbf{coNP}$ does not [12].

[2]See [4] for details on formal languages.

Chapter 3

Minesweeper configurations

In this chapter some Minesweeper configurations are presented. Initially the configurations used by Kaye to prove **NP**-completeness of Minesweeper are summarised and then some configurations developed during the course of the project are detailed. Finally a brief study of the importance of the dimension of a Minesweeper game is included.

3.1 Review of Kaye's work

Richard Kaye has recently proved that the following Minesweeper question, termed the "general Minesweeper problem" [7] is **NP**-complete by a reduction from Boolean satisfiability (SAT). In this section the proof of this claim is studied and used to introduce the exploration of Minesweeper configurations.

We start by defining the general Minesweeper consistency problem, which will henceforth be referred to as CONSISTENCY.

Definition 3.1.1 (Consistency) *Given a rectangular grid partially marked with numbers and/or mines, some squares being blank, determine if there is* some *pattern of mines in the blank squares that give rise to the numbers seen [7].*

CONSISTENCY is the decision problem Kaye proved **NP**-complete. We will show that the conditions for **NP**-completeness hold individually although this is not explicitly done by Kaye.

Theorem 3.1.2 CONSISTENCY $\in$ **NP**.

Proof. Take as certificate a total function $f : X \to L$ where X is the set of all board positions and $L = \{0, 1, \ldots, *\}$ where a number denotes a numeric label and * denotes an identified mine. Let $f(x) = i$ where $i \in \{0, \ldots, 8\}$ that position x is mine-free and has label i, and $f(x) = *$ denote that position x contains a mine. Since $|X| = n$ the certificate is polynomial in the input size and hence short.

Let the verification algorithm A proceed as follows:

1. Place mines and non-mines on the board as defined by f.

2. For each numerically labelled square, check that it has exactly the correct number of mines adjacent to it.

As A scans each square exactly once the worst case complexity of A is polynomial in n (in fact $A \in \Theta(n)$), so both conditions of **NP** membership are met. ■

It is worth noting that the proof is slightly stronger than the original statement, as it both checks the actual configuration for consistency, along with the empty squares as required by the definition of CONSISTENCY. Rather than an explicit proof of theorem 3.1.2 Kaye presented a reduction from an arbitrary Minesweeper configuration to SAT. This reduction is now summarised.

Consider a three-by-three block of squares as shown in figure 3.1. Let the predicate a_m denote "there is a mine at a," and for $0 \leq j \leq 8$ let a_j denote "there is no mine at a and exactly j mines adjacent to a." Define similar predicates for squares *b, c, . . . , i*. The rules of Minesweeper allow the following conditions for the centre square e to be deduced:

a	b	c
d	e	f
g	h	i

Figure 3.1: A three-by-three block.

.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	.
.	x'	x	1	x'	x	1	x'	x	1	x'	x	1	x'	x	.
.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	.

Figure 3.2: A Minesweeper wire, X, moving from left to right.

1. exactly one of e_m, e_1, ..., e_8 is true, and

2. for $k = 0, 1, \ldots, 8$, if e_k is true then exactly k of a_m, ..., d_m, f_m, ..., i_m are true.

Each instance of condition 2 can be expressed as a Boolean formula in 90 variables (a_m, a_0, ..., i_8). Conjoining these formulae to a single expression C it holds that the configuration is consistent if and only if there is a combination of inputs that satisfy C.

Since **NP**-membership of CONSISTENCY has been shown, completeness can be proved by exhibiting a polynomial Turing reduction from a known **NP**-complete problem to it.

The basic building block of the proof is the concept of a 'wire' illustrated in figure 3.2. It is easy to deduce that if x contains a mine then x' cannot contain a mine and vice versa; also either x or x' must contain a mine for the configuration to be consistent. A wire can 'carry' a Boolean value based on the position of the mines as defined in definition 3.1.3.

Definition 3.1.3 *The value represented by a wire is determined by the square immediately before (using an arbitrary predefined direction) the phase separating 1. If this square contains a mine then the value of the wire is true otherwise it is false.*

						1	1	1							
.	1	1	1	1	1	2	$*$	2	1	1	1	1	1	.
.	x'	x	1	x'	x	3	x'	3	x	x'	1	x	x'	.
.	1	1	1	1	1	2	$*$	2	1	1	1	1	1	.
						1	1	1							

Figure 3.3: A NOT gate.

	.	.	.																			
	1	1	1			1	2	2	1		1	1	1		1	1	1					
	1	u'	1			2	$*$	$*$	3	2	3	$*$	2	1	2	$*$	3	2	1			
	1	u	1	1	2	4	$*$	s	x	y	z	t'	3	t	t'	3	$*$	$*$	2			
1	2	2	1	1	$*$	$*$	4	$*$	3	2	3	$*$	2	1	1	2	t	$*$	2			
2	$*$	u'	2	2	4	s'	3	1	1	0	1	1	1	0	0	1	2	2	1			
2	$*$	$*$	3	u	u'	s	2	1	1	1	1	1	1	1	1	1	t'	1	1	1	1	.
2	4	5	$*$	4	$*$	**4**	t	t'	1	t	t'	1	t	t'	1	t	2	t	1	t'	t	.
2	$*$	$*$	3	v	v'	r	2	1	1	1	1	1	1	1	1	1	t'	1	1	1	1	.
2	$*$	v'	2	2	4	r'	3	1	1	0	1	1	1	0	0	1	2	2	1			
1	2	2	1	1	$*$	$*$	4	$*$	3	2	3	$*$	2	1	1	2	t	$*$	2			
	1	v	1	1	2	4	$*$	r	a	b	c	t'	3	t	t'	3	$*$	$*$	2			
	1	v'	1			2	$*$	$*$	3	2	3	$*$	2	1	2	$*$	3	2	1			
	1	1	1			1	2	2	1		1	1	1		1	1	1					
	.	.	.																			

Figure 3.4: An AND gate.

Kaye showed that wires can be bent, split, and negated; and two wires can be conjoined. In this summary only the negation and conjunction operators are included.

The negation of a wire is obtained by reversing the sequence of xs and x's as shown in figure 3.3. The NOT gate is based on the central square (highlighted) containing x', and the 3s adjacent to it ensuring that the two outside squares must be x.

The conjunction operator (figure 3.4) is based around the highlighted central square containing the label 4, where the two input wires U and V are crossed and the output T is initialised. The AND gate has two internal wires R and S which are mainly used to ensure consistency around the central square. Kaye demonstrated the correctness of the AND but for brevity the details are

omitted from this discussion. Note however that the AND gate does not have a unique solution since on the input case where both u and v are mine-free, and thus t must be mine-free, the configuration is consistent both when s, x, z, b, c contain mines and when r, a, c, x, y contain mines. Uniqueness is not required for **NP**-completeness and thus not considered by Kaye, but is important for later results in this paper.

The required logic gates have now been defined and Kaye's can be stated. **NP**-hardness is a direct result of the logic circuits and **NP**-completeness follows directly. The details are omitted but the interested reader should consult [7].

Theorem 3.1.4 CONSISTENCY *is* **NP**-*complete.*

Proof. See Kaye [7].

3.2 Other important configurations

The logic gates presented in the previous section were used by Kaye to prove that CONSISTENCY is **NP**-complete. Two additional logic gates will now be presented that have been invented independently of Kaye, although it was subsequently discovered that similar configurations are known to him [8]. The important aspect of the two new logic gates is that they both have the uniqueness property lagged by the AND gate which will be required for the derivation of later results. The following shorthand, $v(X) = [x \text{ contains a mine}]$ with reference to the standard representation of a wire in figure 3.2, will be used to justify the correctness and uniqueness of the two logic gates.

The first configuration is the disjunction of two wires shown in figure 3.5. The configuration is based around the highlighted square labelled 6, where the two input wires U and V are crossed. The OR gate has output wire T and an internal wire S which loops back onto T to ensure consistency. To prove that this configuration simulates an OR gate each of the three[1] possible input combinations are considered.

[1]Note that the two input combinations representing input wires with different truth-

				1	2	2	1		1	2	3	2	1				
.	1	1	1	2	$*$	$*$	3	1	1	$*$	$*$	$*$	1				
.	1	u'	u	3	u'	$*$	$*$	2	2	3	t'	3	2	1	1	1	.
.	2	2	3	3	$*$	**6**	t	t'	2	t	2	t	1	t'	t	1	.
.	1	v'	v	2	v'	$*$	s	$*$	5	4	t'	2	2	1	1	1	.
.	1	1	1	2	$*$	$*$	6	$*$	$*$	$*$	$*$	3	1				
				1	4	$*$	s'	5	b	c	t	$*$	2				
					2	$*$	$*$	a	4	4	$*$	$*$	2				
					1	3	$*$	$*$	$*$	2	2	2	1				
						1	2	3	2	1							

Figure 3.5: An OR gate.

Case 1: $v(U) = v(V) = true$. The squares labelled u and v contain mines and the ones labelled u' and v' do not. From the highlighted square both s and t must contain mines in this case. From the implied value of S it can be inferred that a and b must contain mines while c cannot, which implies that t must be a mine. This is consistent with its known value, and hence the configuration is consistent on this input.

Case 2: $v(U) = v(V) = false$. The squares u' and v' contain mines so s and t must be mine-free. From the value of S, b and c must contain mines, and thus t cannot contain a mine. Hence, the configuration is consistent on this input combination.

Case 3: $v(U) \neq (V)$. In this case we deduce from the highlighted square that either s or t — but not both — must contain a mine. If s contains a mine, then both a and b must contain mines. But this implies that t also contains a mine which is impossible. However, if t contains a mine then a and c contain mines which implies that s cannot contain a mine. Hence, the configuration is consistent and it simulates an OR gate correctly.

In showing the correctness of the OR gate it was evident that exactly one valid assignment of mines and non-mines to the squares labelled with letters existed for each input combination. This means that the OR gate is unique in the sense that given any input combination both the output and the state

values only constitute one input case when demonstrating the correctness of the configuration.

			1	2	2	1						
.	1	1	2	$*$	$*$	3	1					
.	1	v'	v	5	$*$	$*$	2	1	1			
.	1	1	3	$*$	v'	5	4	$*$	2	1	1	.
.	1	1	4	$*$	v	$*$	$*$	4	t'	t	1	.
.	1	u'	u	$*$	u'	**6**	t	$*$	3	1	1	.
.	1	1	3	4	$*$	s	s	$*$	2			
			2	$*$	6	$*$	$*$	4	2			
			3	$*$	s'	$*$	s'	$*$	2			
			2	$*$	6	s	6	$*$	2			
			1	2	$*$	$*$	$*$	2	1			
				1	2	3	2	1				

Figure 3.6: A XOR gate.

of all squares in the configuration is known.

The other logic gate configuration is the XOR gate shown in figure 3.6. The XOR gate is centred around the highlighted square labelled 6 and contains an internal wire S which loops back upon itself to ensure that either zero or two squares are mines around the central square. Due to the asymmetric nature of the configuration (v and u' border the central square) all four possible input cases need to be considered separately to prove correctness and uniqueness of the XOR gate.

Case 1: $v(U) = v(V) = true$. In this case v contributes one mine to the neighbourhood of the central square and u' contributes none, so two extra mines are required. Thus s must contain a mine and t is mine-free which is the correct value.

Case 2: $v(U) = v(V) = false$. In a similar fashion to case 1 u' contributes one mine to the neighbourhood of the central square and v contributes none, so two extra mines are required. Thus s must contain a mine and t is mine-free which is the correct value.

Case 3: $v(U) = true, v(V) = false$. In this case neither v or u' contain mines so three mines are required around the highlighted square. This is only satisfied when both s and t contain mines and thus the output is true as required.

Case 4: $v(U) = false, v(V) = true$. In this case both v and u' contain

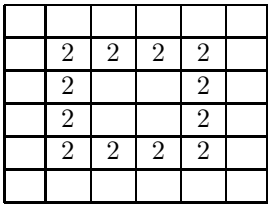


Figure 3.7: Determine the positions of all mines.

mines so only one mine can be placed adjacent to the highlighted square. Thus s must be mine-free while t contains a mine and the output is true as required.

Obviously the logic gate configurations are not very likely to occur when playing Minesweeper; they are merely theoretical configurations that provide insight into the complexity of the game. A configuration of which variations are more frequent will now be considered. Furthermore, it is a nice puzzle since it can be solved from the information provided, although not without some analysis. The configuration, which is also presented by Kaye [7], is shown in figure 3.7. One way to solve this configuration is to exhaustively search through all possibilities using a straightforward backtracking algorithm, which is feasible since the example is relatively small. Without further insight a mine could be placed at the upper left corner and then other mines and mine-free squares systematically placed around it, backtracking when an inconsistency arises. Within three iterations it is easily concluded that the top left square cannot contain a mine, and similar conclusions can be made regarding its neighbours, effectively solving the configuration. This configuration will be used at a later stage when testing our implemented strategies.

3.3 NP-completeness of n-dimensional Minesweeper

To conclude this chapter on Minesweeper configurations the effect of the dimension of the playing area on the **NP**-completeness of CONSISTENCY is

examined. The following definition is useful for this discussion.

Definition 3.3.1 (n-consistency) *Given an n-dimensional grid partially marked with numbers and/or mines, some squares being blank, determine if there is* some *pattern of mines in the blank squares that give rise to the numbers seen.*

It is intuitive that by adding extra dimensions to the playing area the game becomes harder, so it would be expected that 3-CONSISTENCY is **NP**-hard. In fact, n-CONSISTENCY is **NP**-complete for any $n \geq 2$. An interesting situation arises however when reducing the dimension to one, since it is not immediately clear how to construct logic gates as in the 2-dimensional variation. It turns out that 1-CONSISTENCY can be solved in polynomial time, and we present a Deterministic Finite Automata (DFA) that accomplishes this task.

3.3.1 n-dimensional Minesweeper is NP-complete

n-CONSISTENCY can be inductively reduced to $(n+1)$-CONSISTENCY. At a high level, the reduction is to construct a wire, an AND gate and a NOT gate by 'padding' the existing configuration with mines and increasing the labels appropriately to ensure consistency. Finally, the required labels around the outer layer of mines are added.

Theorem 3.3.2 *n-CONSISTENCY $\in$ **NP**, when $n \geq 2$.*

Proof. (Outline) The 2-dimensional case proved in theorem 3.1.2 generalises to n-dimensions. The certificate is an assignment of mines and mine-free squares to the n-dimensional board that is consistent with the configuration, and the same verification algorithm is correct. ■

Theorem 3.3.3 *n-CONSISTENCY is **NP**-hard, when $n \geq 2$.*

Proof. Induction on n. Base case $n = 2$. 2-CONSISTENCY is **NP**-hard by theorem 3.1.4.

Inductive step. Assume n-CONSISTENCY is **NP**-hard through a reduction to SAT via a simulation of logic gates. An extra dimension is added by extending each of the existing n-dimensional configurations as follows. For each cell x in the $(n+1)$st dimension that is adjacent to a labelled cell in the nth dimension a mine is placed on x and the label of each cell in the nth dimension adjacent to x is incremented by one. Then add appropriate label to all cells in the $(n+1)$st dimension neighbouring a mine, and by construction, the $(n+1)$-dimensional configuration is consistent and exhibits the same behaviour as the n-dimensional configuration. The reduction is polynomial in the size of the game board and constant in n. Hence, by induction n-CONSISTENCY is **NP**-hard for all $n \geq 2$. ■

Finally the two results are combined to prove **NP**-completeness.

Theorem 3.3.4 *n-CONSISTENCY is **NP**-complete for $n \geq 2$.*

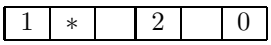
Proof. **NP**-membership is proved in theorem 3.3.2 and hardens in theorem 3.3.3. ■

3.3.2 1-dimensional Minesweeper can be solved in polynomial time

A slightly more interesting result is now presented, namely that 1-CONSISTENCY can be solved in polynomial time. A DFA will be used to demonstrate this result since its property of representing a finite number of states is required. The conversion into a Turing machine or an arbitrary high-level programming language using a set of variables is however implicit.

Before the DFA is described some properties of 1-dimensional Minesweeper are considered. Let a 1-dimensional Minesweeper game consist of a single string of input symbols encapsulated within the letter s, hence the input alphabet is $\Sigma = \{0, 1, 2, b, *, s\}$ where the numbers indicate labels on

Figure 3.8: The Minesweeper configuration corresponding to the string $s1{*}b2b0s$.



squares, b indicates a blank square and $*$ the presence of a mine. For an example of this notation see figure 3.8 which shows the Minesweeper configuration corresponding to the string $s1{*}b2b0s$. Since any square can have at most two neighbours, all the possible inconsistencies that could arise can be easily listed. A 'simple' inconsistency is a string from the alphabet $\Sigma' = \{0, 1, 2, *\}$ which is inconsistent with the rules of the game. It is clear that any inconsistency is at most three symbols long, and that all inconsistencies are symmetric. The simple inconsistencies (omitting symmetries) are: {02,12,22,0*,111,110,010,*1*}.

It is clear that purely detecting simple inconsistencies does not determine the consistency of a configuration. Consider the configuration $*1b2$ which is inconsistent but not directly detectable using the simple inconsistencies. However, when scanning the string from left to right it is clear that the b should be replaced either with a 0 or a 1 in order for the first size three sub string to be consistent, since replacing it with any other value would create one of the listed inconsistent sub strings. Suppose the b is replaced by a 0; this creates the string $*102$ which contains the simple inconsistency 02 as a sub string. Similarly, replacing the b with 1 would produce the inconsistent sub string 12. Seeing that sub strings of at most three symbols need to be recognised, only the previous two symbols and assignments to b need to be recorded.

We will keep track of the previously scanned symbols using the states of a DFA. Note that the symbols $0, 2$, and $*$ are absolute in the sense that information about the previous symbol is not required to determine whether the next symbol will cause an inconsistency. To avoid confusion between symbols of the alphabet and states, the absolute states above will be named Z, T, and M respectively. Special attention to the first and last symbol

scanned is required in order to determine start and end inconsistencies, but this is an easily implemented technicality. Once a simple inconsistency is detected, the DFA will move into a non-accepting state E from which all transitions are to that same state, and the DFA will eventually terminate and reject the input string. The DFA has exactly one accepting state OK which is only reached after scanning the second s; if more symbols are scanned following transition to OK the DFA will move into state E and reject the input.

The DFA is now described and its correctness proved. Let $\mathscr{A} = (Q, \Sigma, \delta, ST, F)$, where $\Sigma = \{s, 0, 1, 2, b, *\}$, $Q = \{ST, S, Sb, S1, Z, T, M, 01, *1, b1, bb, 0b, *b, E, OK\}$ and $F = \{OK\}$. The states are derived from the preceding discussion of possible sub strings, and represent the previous two scanned symbols with assignments to blank squares where appropriate. The transition function δ is defined from the previous discussion of state interdependency and is shown in table 3.1, as the schematic representation is more readable than the corresponding set notation. Note that in order to reduce the number of states, some semantically equivalent states are merged. For example if $\mathscr{A}$ is in state $*1$ and scans a 1 then the next symbol must be either b or $*$ for the input string to be consistent. But this is the behaviour implemented by the state 01 and thus introducing a new state is not required. Similarly if a b is scanned rather than a 1, it is clear that b cannot be assigned a mine since this would create the sub string $*1*$, thus the behaviour implemented in the state $0b$ is required.

The correctness of $\mathscr{A}$ is now proved and its complexity determined to justify the claim that 1-CONSISTENCY can be solved in polynomial time. The proof is an outline since a complete proof would need to consider each input case separately, but this is omitted for brevity.

Theorem 3.3.5 *$\mathscr{A}$ solves 1-*CONSISTENCY*.*

Proof. (Outline) Induction on the input size, n. Base case, $n = 2$. The only acceptable input string of length two is ss. By construction, the DFA will only accept this one string of length two.

Table 3.1: The state transition function δ.

δ	0	1	2	b	*	s
ST	E	E	E	E	E	S
S	Z	$S1$	E	Sb	M	OK
Sb	Z	$b1$	T	bb	M	OK
$S1$	E	E	E	M	M	E
Z	Z	01	E	$0b$	E	OK
T	E	E	E	M	M	E
M	E	$*1$	T	$*b$	M	OK
01	E	E	E	M	M	E
$*1$	Z	01	E	$0b$	E	OK
$b1$	Z	01	E	bb	M	OK
bb	Z	$b1$	T	bb	M	OK
$0b$	Z	01	E	bb	M	OK
$*b$	Z	$b1$	T	bb	M	OK
E	E	E	E	E	E	E
OK	E	E	E	E	E	E

Inductive step. Suppose the n-symbol string $sx_1x_2 \ldots x_{n-3}x_{n-2}s$ is consistent. Now consider the $(n+1)$-symbol input string $sx_1x_2 \ldots x_{n-3}x_{n-2}x_{n-1}s$. By the inductive hypothesis this string is consistent up to the sub string $x_{n-4}x_{n-3}x_{n-2}$ and so it can only be inconsistent if one or both of the sub strings $x_{n-3}x_{n-2}x_{n-1}$ and $x_{n-2}x_{n-1}s$ are inconsistent. The two cases are considered separately.

Case 1. Detecting an inconsistency in the sub string $x_{n-3}x_{n-2}x_{n-1}$ is done by construction, since all possible inconsistencies of size three sub strings are known. If an inconsistency is detected in this sub string, $\mathscr{A}$ will move into the non-accepting state, and hence the string is rejected.

Case 2. Detecting an inconsistency in the sub string $x_{n-2}x_{n-1}s$ is a technical variation of detecting an inconsistency in a 'normal' sub string. $\mathscr{A}$ has been constructed to detect these end-inconsistencies.

Hence, by construction $\mathscr{A}$ will only accept strings where all sub strings are consistent, and by induction it only accepts consistent input strings. ■

The worst-case performance of $\mathscr{A}$ is $\mathscr{A} \in \Theta(n)$ since it scans each input symbol exactly once and does no backtracking. A variation of CONSISTENCY that can be solved in polynomial time has thus been identified, but this variation is not known to be **NP**-complete.

Chapter 4

The complexity of Minesweeper

In this chapter Kaye's **NP**-completeness result for Minesweeper is extended by studying the inherent complexity of playing Minesweeper rather than simply the structure of the game. Some more natural questions about game playing will be asked and results will be presented in sequential order, continuously drawing from the results of the previous sections.

4.1 Basic definitions

Before considering the specific Minesweeper questions it is important to clearly define the notion of a configuration used in the previous chapter. Also recall the definition of consistency (definition 3.1.1).

Definition 4.1.1 (Configuration) *A Minesweeper configuration is a grid (usually rectangular) partially marked with numbers and/or mines and some squares remaining blank.*

Henceforth a solution to a configuration will be termed an explanation in order to obtain a more precise and natural terminology that will become useful.

Definition 4.1.2 (Explanation) *An explanation for a configuration B is an assignment of mines to the empty squares of the grid that gives rise to B.*

4.2 Finding a unique solution

The first Minesweeper question examined is a natural one that most Minesweeper players would consider on a regular basis: Is the information provided on the board sufficient to determine the positions of all remaining mines? Using the terminology of this paper we are interested in, given any configuration to determine whether it has a unique explanation. The decision problem SOLUTION encapsulates this question.

Definition 4.2.1 (Solution) *Input. A configuration B.*
Output. If there exists a unique explanation of B, output "yes". Else, output "no"

4.2.1 A counting argument

It is clear that SOLUTION is exactly equivalent to asking whether a configuration has exactly one explanation, and thus if the number of explanations could be determined efficiently, this would solve SOLUTION. The first attempt at determining the complexity of SOLUTION will make use of the following counting problem associated with CONSISTENCY.

Definition 4.2.2 (#Consistency) *Input. A configuration B.*
Output. The number of explanations of B.

We now prove that #CONSISTENCY is at least as hard as any problem in **#P**.

Theorem 4.2.3 #CONSISTENCY *is* **#P***-complete.*

Proof. Reduction from the **#P**-complete problem #SAT [12]. Since the disjunction and negation operator together are sufficient [9] to specify a logic language (i.e. all other logic operators can be obtained from combinations of OR and NOT) the following reduction applies:

1. Convert an instance S of #SAT to an equivalent instance S' containing only negation and disjunction of variables using identities such as $A \wedge B \equiv \neg(\neg A \vee \neg B)$.

2. Convert S' to a Minesweeper configuration using the gadgets in figure 3.3 and 3.5.

It was showed in the previous chapter that the OR gate is unique and hence preserves the number of solutions between a logic formula and a Minesweeper configuration. By inspection the same holds for the NOT gate so the reduction is parsimonious as required for **#P**-completeness.

A final technicality required is to ensure that the crossing of two wires preserves the number of solutions. Kaye showed that two wires can be crossed using three XOR gates [7] and since it was showed that the XOR gate in figure 3.6 is unique the number of solutions is also preserved when crossing wires.

Thus the number of solutions of the associated Minesweeper problem is the same as the number of satisfying assignments of S' and thus S. Hence #SAT can be solved in polynomial time if #CONSISTENCY can be solved in polynomial time. ■

The initial argument showed that SOLUTION is equivalent to determining whether the number of explanations of a configuration equals one, but by theorem 4.2.3 finding the number of explanations is **#P**-complete. It will be shown subsequently that SOLUTION is easier than #CONSISTENCY although it is still a hard problem. The **#P**-completeness result for #CONSISTENCY was included as it will be an important constraint when attempting to approximate the probabilities of each square containing a mine during the development of Minesweeper strategies.

4.2.2 A more direct approach

It will now be proved that SOLUTION is **DP**-complete and thus easier than #CONSISTENCY. From definition 2.2.7 two languages $L_1 \in \mathbf{NP}$ and $L_2 \in \mathbf{coNP}$ such that all "yes" instances of SOLUTION is $L_1 \cap L_2$ need to be exhibited to prove membership of **DP**.

Theorem 4.2.4 SOLUTION $\in$ **DP**.

Proof. Define L_1 and L_2 as follows: $L_1 = \{X|$ There exists an explanation of $X\}$ and $L_2 = \{X|X$ has at most 1 explanation$\}$. $L_1 \in \mathbf{NP}$ by theorem 3.1.2. To see that $L_2 \in \mathbf{coNP}$ note that it has a succinct disqualification: namely two different explanations of X. Since two explanations is only larger than a single explanation by a constant multiple the combination of two it remains polynomially short and thus $L_2 \in \mathbf{coNP}$.

Hence we have $L_1 \in \mathbf{NP}$ and $L_2 \in \mathbf{coNP}$. L_1 and L_2 were constructed such that their intersection is the language defined by SOLUTION since L_1 ensures that *at least* one explanation exists and L_2 that *at most* one explanation exists, i.e. $L_1 \cap L_2$ only contain configuration with exactly one explanation. Thus SOLUTION $\in$ DP. ■

The **DP**-complete problem UNIQUE SAT [12] is now used to show that SOLUTION is **DP**-complete. UNIQUE SAT is the following computational problem: Given a Boolean formula ϕ, is there exactly one satisfying assignment of truth values to the variables?

Theorem 4.2.5 SOLUTION *is* **DP***-complete.*

Proof. By theorem 4.2.4 SOLUTION $\in$ **DP**. UNIQUE SAT is reduced to SOLUTION using a method analogous to the reduction used in theorem 4.2.3. The technical details are similar so only an outline is given here.

1. Convert an instance S of UNIQUE SAT to an equivalent instance S' containing only negation and disjunction of variables using identities such as $A \wedge B \equiv \neg(\neg A \vee \neg B)$.

2. Convert S' to a Minesweeper configuration using the gadgets in figure 3.3 and 3.5.

It was proved in theorem 4.2.3 that this reduction is parsimonious which completes the proof. ■

Theorem 4.2.5 is an interesting result since it shows that SOLUTION is an easier problem than #CONSISTENCY but generally believed to be harder

than **NP**-complete problems like CONSISTENCY, although this last claim is not yet proved correct.[1]

4.3 Finding a strategy

In the previous section the question about directly determining the positions of the remaining mines on a configuration was considered. Another interesting — and perhaps more urgent in game playing — problem is considering whether a safe move exists. This problem can be used to find a sequence of moves resulting in solving the configuration by once a safe move has been found considering all the possible configurations that could arise from performing that move.

In order to reason formally about changing a configuration, a precise definition of a move is required.

Definition 4.3.1 (Move) *A move M is a pair $(x \in X, m \in \{0, 1\})$ where X is the set of all positions in a configuration such that*

- *$M = (x, 0)$ means 'probing' square x; and*
- *$M = (x, 1)$ means placing a mine on square x.*

Since a move on a given square is defined by a Boolean variable, the complement of a move can be easily defined.

Definition 4.3.2 (Move complement) *The complement of a move $M = (x, m)$ is $(x, 0)$ if $m = 1$ and $(x, 1)$ if $m = 0$.*

A natural question to ask when playing Minesweeper is thus whether any given move is safe, i.e. performing it cannot result in an immediate loss.

Definition 4.3.3 (Move safety) *Move safety is defined explicitly for the type of each move:*

[1] See [12] for a more in-depth account of the relative position of **DP**-complete problems in the complexity hierarchy.

- *A move $(x, 0)$ is safe from a configuration B if B has no explanations with a mine on x.*

- *A move $(x, 1)$ is safe from a configuration B if no explanations of B have x mine-free.*

The decision problem SAFETY can now be defined, and it is interestingly in the same complexity class as SOLUTION.

Definition 4.3.4 (Safety) *Input. A configuration B and a move M.*
Output. If M is safe from B return "yes", otherwise return "no".

Before **DP**-completeness of SAFETY can be proved the following interim theorem is required.

Theorem 4.3.5 *If $S = \{X|P_S\}$ and $R = \{X|P_R\}$ are languages in* **NP** *with associated verification algorithms A_S and A_R then $S \cup R \in$* **NP**.

Proof. Let $L = S \cup R$, thus $L = \{X|P_S \vee P_R\}$. Both S and R must emit a proof system to satisfy the definition of **NP**. Let these proof systems be $F_S \subseteq X \times Q_S$ and $F_R \subseteq X \times Q_R$ respectively. The proof system of L is then $F \subseteq X \times (Q_S \times Q_R)$. A certificate of L is by definition a pair of polynomial length certificates, and hence polynomial in the input size.

The verification algorithm A_L needs to determine whether $(x, (q_1, q_2)) \in F$ and is defined as follows: $A_L = (x, q_1) \in F_S \vee (x, q_2) \in F_R$. We can use A_S and A_R to determine the two terms of the disjunction in polynomial time by definition, so A_L must also be a polynomial time algorithm.

Thus, L emits a proof system consisting of succinct certificates and a polynomial time verification algorithm A_L. Hence, $L \in$ **NP**. ■

DP-completeness of SAFETY is now proved by a reduction from SOLUTION.

Theorem 4.3.6 SAFETY *is* **DP***-complete.*

Proof. Firstly membership of **DP** is required. Consider the languages $S = \{(B,(x,0))|B$ has an explanation with x mine-free$\}$ and $R = \{(B,(x,1))|B$ has an explanation which contains a mine at $x\}$. Both S and R are in **NP** since they both contain a short certificate of a "yes"-instance that can be efficiently verified, namely a configuration that satisfies the predicate part of the language definition. Define $L_1 = S \cup R$, thus $L_1 \in$ **NP** by theorem 4.3.5.

Now consider the language $L_2 = \{(B,M)|B$ is safe from $M\}$. $L_2 \in$ **coNP** since it has a succinct proof of a "no"-instance, and no "yes"-instances have such proofs. The disqualifier is an explanation of B where the complement move of M has been performed. To see that this does in fact disqualify an instance from being in L_2 note that if a move is guaranteed to be safe, then the complement of the move (i.e., putting a mine instead of 'probing' a square or vice versa) must produce an inconsistent configuration (i.e. not an explanation), since otherwise the original move would not be safe.

Now $L_1 \cap L_2$ is the set of configuration-move pairs (B,M) such that performing M results in a consistent configuration and M is guaranteed to be safe. This encapsulates the desired properties of SAFETY and hence SAFETY belongs to **DP**.

To prove **DP**-completeness of SAFETY we reduce to it the known **DP**-complete problem SOLUTION (definition 4.2.1 and theorem 4.2.5). The reduction proceeds as follows:

SOLUTION(B)
  for each unknown square $s \in B$
    b_1 = SAFETY$(B,(s,1))$
    b_2 = SAFETY$(B,(s,0))$
(1) **if** $b_1 = b_2$ **then**
      output "no"
(2) **else if** b_1 = "yes" **then**
      place a mine on square s
(3) **else**
      probe square s

end for
(4) **output** "yes"

This reduction is linear in the board size as it considers each unknown square exactly once. To complete the proof a number of issues need to be considered to show the correctness of the algorithm.

Firstly consider line (1). It is important to notice that both b_1 and b_2 cannot be "yes" since no configuration exists where it is safe to both probe and place a mine on the same square. Thus if $b_1 = b_2$ then both b_1 and b_2 must be "no", and thus from the definition of SAFETY the information provided on the entire board is not sufficient to find a safe move on square s. If a square s exists for which it cannot be determined whether s contains a mine or is mine-free this implies that although the configuration must have at least two explanations, one which contains a mine on s and one which has s mine-free. Thus the reduction correctly outputs "no" when such a square is identified.

If no square is identified by line (1) for which it cannot be determined whether or not it contains a mine it is clear that the value of all squares is known. In this case the only explanation of B has been identified (and the assignments are represented in B from lines (2) and (3)) and thus the reduction should return "yes" as correctly done on line (4). To see that this is in fact the correct behaviour notice that "yes" is only returned once all squares on the board have been checked, and no square found where the assignment of mine or mine-free is unknown.

Thus as the above argument shows the correctness of the reduction, the proof is complete and SAFETY is **DP**-complete. ■

Since questions of game playing have been the theme of this chapter it is interesting to note that the assignment of mines and non-mines in lines (2) and (3) of the reduction are not technically required, however they are included to provide an illustration of the possible implementation of a Minesweeper strategy. The complexity of both SOLUTION and SAFETY have

an important impact on the types of Minesweeper strategies that can reasonably be implemented and knowing these bounds is useful when designing strategies.

Chapter 5

Minesweeper strategies

This chapter covers the development of the Minesweeper strategies invented for this project. The three strategies, Single point, Limited search, and Limited search with probability estimation will be described in increasing order of success and a performance analysis of each will be undertaken.

5.1 General remarks

5.1.1 The consequences Minesweeper's complexity

In the last chapter it was proved that both finding the position of the remaining mines on the board and determining whether a given move is safe are **DP**-complete problems. The implication of the first result is clearly that determining the explanation of the configuration at any given time in an intractable problem and hence this cannot be implemented in a strategy. The second result means that the strategy of considering each square in turn and checking whether either placing a mine or leaving it mine-free is a safe move is also intractable. This hypothetical strategy would effectively be an implementation of the reduction from theorem 4.3.6 and would perform optimally, since whenever a safe move existed it would be identified and performed, and thus a guess would only be made when no safe move existed on the configuration. The complexity results thus imply that considering the entire game

board at each move is intractable and strategies methods for reducing the search area need to be considered if a complete search needs to be performed while remaining tractable.

5.1.2 The objectives of Minesweeper strategies

The intentions and objectives for the development of Minesweeper strategies are now clarified. To aid this discussion the notion of a *perfect game* is defined as a game in which the player takes no unnecessary risks by making a guess rather than utilising the available information to deduce a safe move. Clearly playing a perfect game is intractable since it requires using the **DP**-complete problem SAFETY and thus the developed strategies are not intended to play perfect games. Rather we aim to approximate perfection by requiring that each strategy is perfect *within its local area*, meaning the part of the game surface being considered. The objective thus becomes to develop strategies with maximal probability of winning an arbitrary game through improving the methods of utilising as much information as possible while maintaining tractability of the strategy.

An approach to solving the non-trivial configuration in figure 3.7 has been previously considered. For an experienced human player, there are several useful techniques that can improve the problem-solving performance[1] through limiting the interest area and recognising previously solved parts of a configuration.[2] Otherwise it may be inferred that the solution is symmetrical around the centre of the configuration, which would immediately indicate that the corner squares are all safe. This type of insight is extremely valuable for a human player of Minesweeper, however as it is due to the visual perception of the board and may not necessarily apply to different situations it is not attempted to model or implement any such insight. It is expected however that the best strategies will be able to successfully solve non-trivial

[1]For a human player the objective is often to obtain a solution in the least possible time while we are concerned with winning the most games on average.

[2]See [21] for more details of how to improve a human strategy.

configurations like figure 3.7.

5.1.3 Evaluation of strategies

The Minesweeper strategies will be evaluated mainly on their probability of winning an arbitrary game. To this end winning ratios of some existing Minesweeper strategies are required to use as a benchmark performance. A thorough search of the Internet revealed a severe lack of such information, as most serious Minesweeper sites (e.g. [10, 21]) are devoted to the development of better human strategies, motivated purely by minimising the playing time, generally at the cost of loosing more games on the average.

One web page devoted to automated Minesweeper playing, The Programmer's Minesweeper Page (PGMS) [15] that presents three strategies, was identified. Furthermore, Chris Studholme, a Ph.D. student at the university of Toronto, has implemented a very successful Minesweeper strategy based on reducing a configuration to a constraint satisfaction problem (CSPStrategy) [18]. Statistics of both his own strategy and the strategies available from the PGMS are presented by him. The statistics are win ratios at three different difficulty levels, namely a 10×10 board with 10 mines (beginner), a 16×16 board with 40 mines (intermediate), and a 30×16 board with 99 mines (expert). The same levels will be tested in this evaluation both to use Studholme's results as a benchmark, and because these levels are the standard levels distributed on most Minesweeper implementations with which the reader would be familiar. The benchmark performances are as follows: beginner 80%, intermediate 45%, and expert 34%.

Statistics will be collected by letting each strategy play 1000 sets of 100 games and recording the number of wins for each set. Using this method of data collection both the average number of wins and variance can be obtained along with the frequency distribution. Each strategy will also be tested on several non-trivial configurations such as figure 3.7, and it is expected that the best strategies will solve most of these.

5.1.4 The first move

At the beginning of a Minesweeper game the player is presented with a blank configuration, which means that the first move is forced to be a guess.[3] Our implementation of Minesweeper permits the player to select one of two rules for the first move: 'always safe' or 'can lose'. As the first-move-safe option is the most common variation of Minesweeper[4] it will be considered the default rule without making this explicitly clear; also the benchmark statistics are based on using this rule.

As the first move, a square containing the label "0" will ideally be uncovered, since that would allow the safe uncovering of all the neighbouring squares. We now show that the probability of uncovering a square with label "0" is maximal when by probing a corner square as the first move. Consider a Minesweeper board of size n which contains k hidden mines. Clearly there are $\binom{n-1}{k}$ possible explanations since the first move is safe and thus it can be assumed that the mines are only distributed after the first move has been performed. The first move can be categorised into three types depending on the number of neighbours it has: corner move (3 neighbours), border move (5 neighbours), other move (8 neighbours). The number of explanations to each type of move will be considered in turn.

Assume that the move is a corner move, then we need to fix the three adjacent squares and distribute the k mines in the remaining $(n-1)-3 = (n-4)$ squares. There are $C_{corner} = \binom{n-4}{k}$ ways of doing this. Similarly there are $C_{border} = \binom{n-6}{k}$ ways of securing a border square to yield a zero and $C_{other} = \binom{n-9}{k}$ ways if the initial move was not in a corner or on a border. The following probability

$$\max_{x \in C} \frac{C_x}{\binom{n}{k}} \tag{5.1}$$

needs to be maximised, which is done by selecting the largest C_x. It is

[3]Note that in the special case of playing a predefined configuration information is available on the board and a guess is not required.

[4]The Minesweeper game included in the Microsoft Windows operating system implements the first-move-safe rules.

intuitive that C_{corner} is the greatest, however for completeness an argument to justify this claim is included. Consider Pascal's triangle identity

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{5.2}$$

and note that in our case $k \geq 1$ and $n \geq 1$ since we need to place at least one mine on the board. Using this fact we find that

$$\binom{n-1}{k-1} > 0 \Rightarrow \binom{n-1}{k} < \binom{n}{k} \tag{5.3}$$

from equation 5.2. By transitivity

$$\binom{n-4}{k} > \binom{n-6}{k} > \binom{n-9}{k} \tag{5.4}$$

when $(n-9) > k$ which is generally the case since the total number of mines is usually a percentage of the number of squares (just over 20% at expert level). Hence probing a corner square will maximise the probability of finding a "0" label at the first move, and thus all the developed strategies will always probe the top left corner as the first move of the game.

5.2 Single point strategy

5.2.1 Background and motivation

At a high level, the single point strategy aims to identify safe moves based on the information provided by a completed move (x, b) and the squares immediately adjacent x. The motivation behind the single point strategy is the observation that it often is possible to deduce some safe moves using only the information from the probed square and its neighbours. The name was chosen mainly due to the fact that the strategy will only consider one square at a time, but also because the most elementary PGMS strategy has that name and has similar motivation. We point out that neither source code nor

any description of that strategy have been accessed both prior and after our strategy development.

5.2.2 Procedural detail

The single point strategy maintains an ordered set S of all identified safe moves. At each move the strategy checks whether $S \neq \{\}$, and if so removes the first move $M = (x, b), b \in \{0, 1\}$ from S and performs it[5], otherwise the algorithm will randomly select a move $M = (x, 0)$ where x is any square on the board that has not been probed. After performing M the algorithm checks for safe moves in the neighbourhood of x that can be added to $S \setminus \{M\}$. There are two cases in which safe moves can be deduced around x, namely if all neighbours are safe or all neighbours contain mines. Both of these cases can be easily detected provided the availability of the functions $\textsc{Neighbours}(x)$ and $\textsc{Label}(x)$, which return the set of board positions neighbouring x and the label of square x respectively.

To check whether the remaining squares around x are safe, the set N_x of x's neighbours is partitioned into two disjoint sets $F_x = \{x' | x' \in N_x \wedge \textsc{label}(x') \neq *\}$ and $U_x = \{x' | x' \in N_x \wedge \textsc{label}(x') = *\}$.[6] If $|U_x| = \textsc{Label}(x)$ then the remaining squares in N_x, namely F_x are safe and the set of moves $\{(x', 0) | x' \in F_x\}$ can be added to S.

Similarly to check whether the remaining squares around x must contain mines the following partition on $\mathbb{P}N_x$ is used. $F_x = \{x' | x' \in N_x \wedge \textsc{label}(x') = null\}$ and $U_x = \{x' | x' \in N_x \wedge \textsc{label}(x') = *\}$. Now, if $|U_x| + |F_x| = \textsc{Label}(x)$ then all the squares in F_x must contain a mine and hence the set of moves $\{(x', 1) | x' \in F_x\}$ is added to S.

Since the strategy only attempts to discover new moves immediately following a move (x, b) it is important that as much information about the neighbourhood of x as possible is available. A priority queue based on simple information theory, which states that knowing something that has a low

[5]Note that we use the notation introduced in the definition of a move (Definition 4.3.1).
[6]Recall that the asterisks symbol (*) is used to denote a mine label.

probability conveys more information than knowing something that has a high probability, is implemented. In general a square will have more mine-free neighbours than neighbours containing a mine, and hence it is considered more valuable to know the position of a mine rather than a free square, so moves identifying a mine are added to the front of the set. Algorithm 1 outlines the single point strategy.

Algorithm 1 Single point strategy.

```
S ← {}
while not lost do
  if S ≠ {} then
    (x, b) ← FIRST(S)
  else
    (x, b) ← (RANDOM, 0)
  end if
  DOMOVE(x, b)
  if LABEL(x) = 0 then
    S ← S ∪ NEIGHBOURS(x)
  else
    N ← NEIGHBOURS(x)
    for each n ∈ N do
      if LABEL(n) =ADJ_MINES(n) then
        S ← S ∪ NEIGHBOURS(n)
      end if
      if UNPROBED_ADJ(n)+ADJ_MINES(n) =LABEL(n) then
        S ← S ∪ NEIGHBOURS(n)
      end if
    end for
  end if
end while
```

5.2.3 Complexity

The worst-case complexity of the single point strategy is now briefly discussed. Let $S(n)$ denote the time single point strategy takes to play a Minesweeper game on a board of size n. Ironically, the worst case of the single point strategy is when it succeeds in winning a game. From algorithm

1 it appears that two nested loops determine the complexity, however the inner **for**-loop will have maximum eight iterations, since any square in 2-dimensional Minesweeper can have at most eight neighbours. Thus only the implementation of the set-union operator needs to be considered in order to determine the worst case cost of $S(n)$. It is clear that the set-union operator needs to compare the inserted elements to each element already in S which depends on the number of moves already performed. Say the move that has last been performed was move i, then the worst case is when S contains $n-i$ elements. The worst-case performance of the single point strategy is thus

$$\begin{aligned} S(n) &= \sum_{i=1}^{n} 8 \sum_{j=1}^{n-i} 1 && (5.5) \\ &= 8 \sum_{i=1}^{n} (n-i) && (5.6) \\ &= n^2 - \frac{n(n-1)}{2} && (5.7) \\ &\in O(n^2). && (5.8) \end{aligned}$$

The alternative implementation of S as an ordered binary tree would decrease the insertion time of an element to $\log n$ and a worst case of $S(n) \in O(n \log n)$ would be achieved. This implementation would however impose an external order on S based on the game board rather than the FIFO order which we require. The reason for this seemingly inefficient implementation of S is that a data structure with a queue-like behaviour but with the set property of no repeated elements is needed. Despite the quadratic worst-case performance of the single point strategy, we point out that it will be extremely unlikely that S contains sufficiently many elements to significantly affect the actual run-time of the implemented strategy, and thus the hidden constant would be small.

	Beginner	Intermediate	Expert
Avr. wins (%)	77.4	29.1	0.5
Win variance	17.15	20.47	0.48
First zero wins (%)	81.4	33.1	0.7
Avr. guesses	2.30	4.38	5.45
Guess variance	2.43	7.60	13.54

Table 5.1: Performance summary for the single point strategy.

5.2.4 Performance analysis

The results of the single point strategy are summarised in table 5.1 from which it is seen that it won 77.4% of the games at beginner level with a variance of 17.15, 29.1% at intermediate level with variance 20.47, and 0.5% at expert level with variance 0.48. In fact, the CSPStrategy only wins 80% of its beginner level games[18] at the obvious cost of a much more complicated algorithm. Furthermore, all PGMS strategies perform worse than the single point strategy at beginner level.

The single point strategy makes an average of 2.30 guesses per game with a variance of 2.43 at beginner level. Performing just over two guesses per game is relatively good considering that it is intended to be a novice strategy. Furthermore, the single point strategy increased its winning percentage to 81.4% in games where the first probe yielded a zero label, implying that the second move of the game was not a guess.

At intermediate level, the single point strategy was substantially worse than the CSPStrategy but still performed significantly better than the most elementary PGMS strategy (also single point) which only won about 4% of the games [18]. Unsurprisingly the average number of guesses per game increased significantly to 4.38 with variance 7.60. It was expected that as the mine density of the board increased (0.15625 at intermediate level) more guesses would have to be made, and the large variance indicates a relatively unstable strategy. Considering only games where the first probe yielded a zero, the single point strategy won 33.1% of the games at intermediate level.

The expert level performance was predictably poor, primarily due to frequent guessing resulting from the very limited scope the strategy has to make decisions on when a square is mine-free. The average number of guesses was only 5.45 with variance 13.54. While the average may seem small compared to the 4.38 at intermediate level, the increased mine density of 0.20625 means that less guesses are successful and hence a game is more likely to end at a loss with fewer guesses being performed. Note that the mine density at expert level implies that just over every fifth square contains a mine, and hence it is not unsurprising that the average number of guesses is just about five. From these statistics we feel that reducing the number of guesses made on expert level, and guessing more wisely is the principal area that needs improvement in more advanced strategies. Finally note that the single point strategy was not capable of solving the configuration from figure 3.7.

5.3 Limited search strategy

5.3.1 Background and motivation

As previously documented, the single point strategy performs poorly on the advanced level primarily due to its small search space and limited methods of reasoning about the board. It is clearly not realistic to attempt discovering the positions of all mines only by looking at the immediate neighbours of each square, however care must also be taken not to create an intractable problem by attempting to exhaustively search too great an area. The theoretical results presented indicate that in order to create a successful strategy, which remains computationally tractable, a method of significantly reducing the playing area without depriving the strategy of useful information is required. The limited search strategy comprises of a backtracking algorithm, which determines whether an initial assumption of placing a mine at a particular square implies a contradiction, which will be applied to a significantly reduced part of the playing area using a ‘zone of interest’.

The method for reducing the search space is from an idea, termed the

Figure 5.1: The zone of interest of the square X.

zone of interest, introduced by Peña and Wrobel in a paper exploring a multirelational learning method for playing Minesweeper [13]. The zone of interest is intended to be the set of labelled squares in the proximity of an unknown square that is most useful for determining whether the unknown square is safe.

Definition 5.3.1 *The zone of interest of an unknown square x is the set of squares labelled with numbers that are adjacent to x along with the set of labelled squares that share an unknown square with one of the labelled neighbours of x.*

Consider the configuration in figure 5.1 where the safety of X is required. Using the highlighted zone of interest it is clear that exactly one of the squares marked A must contain a mine implying that the left neighbour of X must also contain a mine, hence X is mine-free. We recognise that this example is contrived for the purpose of illustrating the usefulness of the zone of interest, however it illustrates that the zone of interest contains a good choice of squares that is neither so large that the search problem becomes intractable nor so small that the information required to solve the configuration is lost.

The limited search strategy considers each square in turn to determine whether a safe move can be performed on it. This is done by initially assum-

ing that the square in question, x, contains a mine and then exhaustively constructing the possible configurations with no unknown squares in the neighbourhood of the zone of interest. If a situation arises where no valid assignment can be made then it can be concluded that the previous assignment was erroneous and backtracking is required. The aim of this procedure is to backtrack all the way to the assignment of x, implying that no explanations of the configuration have the initial assignment choice. If this situation occurs it is clear that the initial assumption — x contains a mine — is false and hence x is mine-free. The same method is used with the assumption that x is mine-free if no contradiction was reached in the first search.

Limited search uses a simple backtracking algorithm based on depth-first-search, chosen primarily due to its minimal memory requirements despite having exponential complexity. One might have considered breadth-first-search better suited for the application since it avoids searching deep branches of the tree that do not contain an explanation, however by construction of the search problem all explanations will only be recognisable at the last level of the recursion and breadth-first-search would thus loose its performance advantage while having large memory requirements. Depth-first-search is thus most appropriate.

5.3.2 Procedural detail

A detailed description of the limited search strategy is now presented and the pseudo-code is detailed in algorithm 2. Assume that an unlabelled square x needs to be explored. Initially two sets, $Zone$ containing all the labelled squares that needed to explore x, and O containing all the unlabelled neighbours of all members of $Zone$, are created. An arbitrary order is imposed on O, with the only restriction that $O_1 = x$. Hence $O = \{s \in X | \exists s' \in Zone.s \in \text{ADJ}(s') \wedge \text{LABEL}(s) = null\}$.

$Zone$ is used to maintain two sets of variables $S_{Squares}$ which represents the set of unlabelled squares adjacent to z and S_{info} which contains the number

of unexplored mines[7] adjacent to z. $S_{Squares}$ and S_{info} will be maintained in parallel and are intended to be implemented as a joint data structure which we denote by the short hand S.

S and O are used to perform the actual search which is outlined in algorithm 3. If the search returns the Boolean value $true$, it indicates that a contradiction has been found and that the inverse move to the one attempted in the search can be safely performed. If, once the strategy has considered all the squares on the board no safe move has been performed, it concludes that a random guess is required.

The search algorithm is the main backbone of the limited search strategy, and its functionality will now be described. The search function SEARCH takes three arguments: S, O, and a Boolean value b which indicates whether the algorithm should place a mine on the next square in O or leave it mine-free. The next square to be considered is simply the next square in the ordered set O; note that the square in question x has been explicitly placed at O_1 in order to ensure that it is the first square to be considered in the search.

SEARCH achieves the desired backtracking by a recursive implementation of depth-first-search. Each time it is called it will first check the base case, namely whether O is empty in which case it will conclude that no contradiction has been made on the current branch and returns $false$. If O is not empty limited search will select the first square of O, $s = O_1$, and create a set $O' = O \setminus \{s\}$. Note at this point that $|O'| = |O| - 1$ ensuring that the size of O is decreased by one in each recursive call, implying that the recursion will terminate.

The new state of S will be represented by S', which is created using the value of b.

- If $b = true$ the algorithm will assign a mine to s and creates S' as follows. For each $z \in Zone$ if $s \in S_{Squares}(z)$ then $S'_{Squares}(z) = S_{Squares}(z) \setminus \{s\}$ and $S'_{info}(z) = S_{info}(z) - 1$, otherwise $S'(z) = S(z)$.

[7] unexplored mines = LABEL(z)− number of known adjacent mines to z.

Algorithm 2 An outline of the Limited search strategy.

```
while not lost do
  moved = false
  for each x ∈ X do
    S ← {}
    Z ←ZONEOFINTEREST(x)
    O_1 ← x
    for each z ∈ Z do
      O ← O∪ADJ_NOLABEL(z)
    end for
    for each z ∈ Z do
      S_Squares(z) ← ADJ_NOLABEL(z)
      S_info(z) ←LABEL(z) − #ADJ_MINE(z)
    end for
    if SEARCH(S, O, true) = true then
      PROBE(x)
      moved = true
    end if
    if SEARCH(S, O, false) = true then
      MARK_MINE(x)
      moved = true
    end if
  end for
  if moved = false then
    MAKE_RANDOM_MOVE
  end if
end while
```

Hence, if a mine is assigned to square s limited search will decrease the information counter associated with each element $z \in Zone$ whose current set of unknown neighbours contains the element s and then remove s from the set of unknown neighbours of z.

- If $b = false$ the algorithm will make s mine-free and create the set S' as follows. For each $z \in Zone$ if $s \in S_{Squares}(z)$ then $S'_{Squares}(z) = S_{Squares}(z) \setminus \{s\}$ and $S'_{info}(z) = S_{info}(z)$, otherwise $S'_z = S_z$. Hence, if s is assigned to be mine-free limited search will remove s from the set of unknown neighbours of z for each element $z \in Zone$ whose current set of unknown neighbours contains the element s without decreasing the information counter associated z.

After the creation of O' and S' limited search uses S' to detect whether the assignment of s was invalid. There are two cases of an invalid assignment and they can both be checked easily using the structure of S'. Observe that the cases are not mutually exclusive as situations can arise where both types of inconsistency occur from a single assignment to s. However, with respect to the algorithm it is irrelevant which inconsistency is detected since the required action is the same in both instances.

The most obvious case is when s has been assigned a mine, but all the mines adjacent to one of s's neighbours have already been identified. This situation can be detected by checking whether any $z \in Zone$ has a negative $S'_{info}(z)$, since $S'_{info}(z)$ is decremented each time a mine is placed adjacent to z.

The second case of inconsistency occurs when s has been assigned mine-free but as a result one of its labelled neighbours no longer has enough unlabelled neighbours to place the required number of mines on. Since s has been removed from $S'_{Squares}(z)$ for all z with out an associated decrease in $S'_{info}(z)$ this situation is detected by checking whether there exists any element $z \in Zone$ such that $|S'_{Squares}(z)| < S'_{info}(z)$, in which case z has less unknown neighbours than the required number of remaining mines adjacent to it.

Once an inconsistency has been detected SEARCH returns *true* without performing further search on the current branch of the tree. If no inconsistency is detected SEARCH will generate two new recursive calls, both with O' and S' as the parameters, and one with $b = true$ and one with $b = false$ in order to correctly check all possible assignments of values to the remaining squares in O'.

5.3.3 Complexity

The asymptotic worst-case performance of limited search will now be derived and the size of the hidden constant will be discussed since it will be useful in obtaining a reasonable estimate of the actual runtime. In a similar sense to the complexity of the single point strategy, the worst case of limited search is when it succeeds in winning a game. From algorithm 2 we see that it consists of two nested loops iterating over all squares on the board. Note that an inner **for**-loop also exists, but this loop will never iterate more than a constant number of steps and does thus not affect the worst-case performance of limited search. Considering the outer loop it is clear that it iterates as long as there are unknown squares left on the game board, and that at least one move will be made during each iteration. Thus, the worst case will occur when exactly one move is performed during each iteration of the outer loop, which would then be executed n times. The inner loop is executed once for each square left on the board, and each call to the function SEARCH can be used as a barometer to determine the complexity. Letting $S(N)$ be the worst case performance of SEARCH and $L(N)$ the time to perform the limited search strategy on a size N board it is clear from the above argument that

$$L(N) = \sum_{\substack{x \in X \\ \text{LABEL}(x)=null}} \sum_{\substack{x \in X \\ \text{LABEL}(x)=null}} S(N) \tag{5.9}$$

$$\leq \sum_{i=1}^{N} \sum_{i=1}^{N} S(N) \tag{5.10}$$

$$\in O(N^2 S(N)) \tag{5.11}$$

Algorithm 3 The searching algorithm used by the Limited search strategy.

function SEARCH(S, O, b)
  if $O = \{\}$ **then**
    return $false$
  else
    $p \leftarrow O_1$
    $O' \leftarrow O \setminus \{p\}$
    if $b = true$ **then**
      for each $z \in Zone$ **do**
        if $p \in S_{Squares}(z)$ **then**
          $S'_{info}(z) \leftarrow S_{info}(z) - 1$
          $S'_{Squares}(z) \leftarrow S_{Squares}(z) \setminus \{p\}$
        end if
      end for
    else
      for each $z \in Zone$ **do**
        if $p \in S_{Squares}(z)$ **then**
          $S'_{info}(z) \leftarrow S_{info}(z)$
          $S'_{Squares}(z) \leftarrow S_{Squares}(z) \setminus \{p\}$
        end if
      end for
    end if
    for each $z \in Zone$ **do**
      if $S'_{info}(z) < 0$ **then**
        return $true$
      end if
      if $|S'_{Squares}(z)| < S'_{info}(z)$ **then**
        return $true$
      end if
    end for
    return SEARCH($S', O', true$)$\wedge$SEARCH($S', O', false$)
  end if
end function

so the complexity of limited search depends on the complexity of the search algorithm.

From algorithm 3 each recursive call to SEARCH requires an amount of work linear in the size of S and generates two recursive calls to SEARCH with the size of O decreased by one. Let n and M denote the sizes of O and S respectively, noticing that M remains constant throughout the recursion. Define the recurrence relation $S(n)$ as follows:

$$\begin{aligned} S(n) &= 2S(n-1) + O(M) && (5.12)\\ S(0) &= 1 && (5.13) \end{aligned}$$

Dividing both sides by 2^n and substituting $T(n) = \frac{S(n)}{s^n}$ we get

$$T(n) = T(n-1) + \frac{O(M)}{2^n}. \tag{5.14}$$

But $S(0) = 1 \Rightarrow T(0) = 1$ so

$$\begin{aligned} T(n) &= \sum_{k=1}^{n} \frac{O(M)}{2^k} && (5.15)\\ &= O(M) \sum_{k=1}^{n} 2^{-k} && (5.16)\\ &= O(M)(1 - (\frac{1}{2})^n). && (5.17) \end{aligned}$$

Thus, substituting back for $S(n)$

$$\begin{aligned} S(n) &= 2^n O(M)(1 - (\frac{1}{2})^n) && (5.18)\\ &= O(M)(2^n - 1) && (5.19)\\ &= 2^n O(M) - O(M) && (5.20)\\ &\in O(M2^n). && (5.21) \end{aligned}$$

Hence, as intended, the search function has a worst-case complexity exponential in the size of the search area rather than the size of the entire

	Beginner	Intermediate	Expert
Avr. wins (%)	91.7	64.3	17.0
Win variance	7.09	22.11	14.35
First zero wins (%)	96.3	74.0	21.8
Avr. guesses	1.68	2.85	4.50
Guess variance	1.31	4.17	8.22

Table 5.2: The performance summary of the limited search strategy.

board. In general M is bounded by n and since the size of the search area is bounded from above by a constant, c_1 we technically have $M \leq c_2 n \leq c_1$ for $n > n_0$, which implies that $S(n) \in O(1)$. The complexity of limited search is thus $L(N) \in O(N^2)$ i.e., the same as the single point strategy.

Despite the complexity being the same in the worst case, the constant involved needs to be considered in order to obtain a reasonable estimate of the actual execution time. Assume for simplicity that the search area is exactly two squares in each direction from the square in question, and approximately one third of the squares in this area are unlabelled. This means that the search area is eight squares and the number of labelled squares is 17. Hence limited search will take approximately $17 \times 2^8 = 4352$ elementary operations in the inner loop compared to one in the single point strategy. As the size of the search area increases the inner constant increases exponentially, however as the search area is unaffected by the input size it is asymptotic.

5.3.4 Performance analysis

The results for limited search are summarised in table 5.2 and it is seen that it won 91.7% of the games played at beginner level with a variance of 7.09, 64.3% at intermediate level with variance 22.11, and 17.0% at expert level with variance 14.35. This is as expected a significant increase in performance over the single point strategy. Limited search is also significantly better at both beginner and intermediate levels than the benchmark performance, however at expert level it remains inferior.

*	*	*
3		3
1		1

Figure 5.2: A Minesweeper configuration that cannot be solved without making a 50-50 guess.

At beginner level the performance is now close to optimal. If only games in which a zero was yielded at the first move are considered the limited search strategy wins 96.3% of the games. This is significant since it implies that once the strategy has found an area in which it knows some safe moves it is able to keep extending that area to the remaining board. Obviously situations occasionally arise that cannot be solved explicitly by any algorithm; such as the configuration shown in figure 5.2, where there is a 50% chance of the one remaining mine being at either of the two blank squares. On the basis that unsolvable configurations can occur in Minesweeper games, we believe that a wining percentage of 96.3 is indeed close to optimal. Also note that the average number of guesses made per game has decreased to 1.68 with a variance of 1.31. Considering that each game must have at least one guess (the first move) this data is further evidence that the limited search strategy is capable of identifying virtually all the safe moves on the board.

The intermediate level performance has been increased significantly compared to the single point strategy, but unexpectedly at the cost of a slight increase in variance. The performance increase is mainly contributed to a significant decrease in the average number of guesses to 2.85 with variance 4.17. The fact that limited search uses one and a half guesses less on average shows that it is more successful in locating safe moves on this level. The variance of guesses is also significantly reduced which indicates that long sequences of guesses are uncommon in the limited search strategy. This is because a successful guess is more likely to provide useful information for *some* unlabelled square on the board, and a new sequence of safe moves can be initiated. Observe that if only games in which the first probe yielded a

zero are counted, limited search won 74.0% of the games. This again shows the importance for the limited search strategy to find an area where it can identify several safe moves and hence increase the size of that area.

The advanced level performance is where the largest performance increase has occurred, and again the low variance and average number of guesses are the main reasons for this. The average number of guesses at this level is 4.50 with variance 8.22 which is a substantial improvement from the single point strategy. While the average is only about one guess less, the much smaller variance implies that the strategy is more able to use the search function to explore the known area and thereby guess less frequently. It is also noteworthy that the winning percentage increases to 21.8 if only games where the first probe yielded a zero were counted. The increase is not as large as we might have hoped and it indicates that although a safe area is discovered at the beginning of a game, the mine density is so high that the strategy still needs to make several guesses in order to extend the safe area to the complete board. This means that the strategy needs to guess frequently in situations where the information on the board would be helpful in minimising the probability of hitting a mine. Limited search can therefore mainly be improved by improving the guessing strategy employed. Otherwise the results indicate that the search function is very successful in identifying safe moves and was capable of solving figure 3.7 without making a guess.

5.4 Adding probability estimates

5.4.1 Background and motivation

In the previous section it was seen that the limited search strategy performed relatively poorly on advanced level mainly due to frequent guessing. In this section limited search is extended to utilise the information provided when making a guess. This is done by estimating the probability of each square containing a mine, and selecting the square with the lowest probability. Again the previously presented theoretical results affect the approaches

that can reasonably be taken. It is clear that in order to calculate the exact probability of any square, say x, being a mine we need to count the number of explanations the configuration has with a mine on x and divide this number by the total number of explanations. However, counting the number of explanations a configuration has is exactly the problem #CONSISTENCY (see definition 4.2.2), which is **#P**-complete by theorem 4.2.3. This result restricts us to make use of estimates of the total probabilities for each square.

In order to estimate the probability of each square containing a mine, the search algorithm used in limited search is extended to count the number of explanations found, rather than simply returning whether or not any explanation was found. For each square x two searches will be performed, one with x being a mine and the other with x mine-free. The sum of the two search results will estimate the total number of explanations, which will be used as a heuristic. It is clear that sine the number of explanations of the configuration is required, the choice of the local area becomes significantly more important. Presently the zone-of-interest approach will be used, however other simple choices of local areas will be investigated subsequently.

5.4.2 Procedural detail

The SEARCH function of limited search can be easily extended to a search function PROB_SEARCH which counts explanations as follows. Each time SEARCH returns $true$, indicating that a contradiction has been found, PROB_SEARCH returns a 0, since no explanations were found on the current branch of the recursion. When SEARCH returns $false$, because it has reached the end of the set of squares without finding an inconsistent assignment, PROB_SEARCH returns a 1, indicating that it has identified one explanation of the search area. Finally, rather than conjoining the results of each recursive branch as done in SEARCH, PROB_SEARCH simply sums the total number of explanations found in each branch. The pseudo-code for PROB_SEARCH is shown in algorithm 4.

The main body of the limited search with probability estimates strategy

Algorithm 4 The extension of the search function to count explanations.

```
function SEARCH(S, O, b)
  if O = {} then
    return 1
  else
    p ← O_1
    O' ← O \ {p}
    if b = true then
      for each z ∈ Zone do
        if p ∈ S_Squares(z) then
          S'_info(z) ← S_info(z) − 1
          S'_Squares(z) ← S_Squares(z) \ {p}
        end if
      end for
    else
      for each z ∈ Zone do
        if p ∈ S_Squares(z) then
          S'_info(z) ← S_info(z)
          S'_Squares(z) ← S_Squares(z) \ {p}
        end if
      end for
    end if
    for each z ∈ Zone do
      if S'_info(z) < 0 then
        return 1
      end if
      if |S'_Squares(z)| < S'_info(z) then
        return 1
      end if
    end for
    return SEARCH(S', O', true)+SEARCH(S', O', false)
  end if
end function
```

differs only from that of limited search in its actions after receiving the search results. Let m be the returned number of explanations with x containing a mine, and f the number of explanations with x mine-free. Trivially, if $m = 0$ then x must be mine-free and can be probed immediately, similarly if $f = 0$ then x must contain a mine, which is placed immediately. Otherwise let $p = \frac{m}{m+f}$ be the estimated probability of x containing a mine and add the pair (x, p) to a set $SafeList$. Since at most one guess will be made following each iteration, only the squares with the lowest probability of containing a mine need to be stored. $SafeList$ is implemented as a set of pairs (s, q), such that for any two pairs $(s_1, q_1) \in SafeList$ and $(s_2, q_2) \in SafeList$ it holds $|q_1 - q_2| \leq \epsilon$ for some small value of ϵ. Although q_1 and q_2 are rational numbers by definition and can hence be tested for equality, the ϵ definition is used for implementation purposes in order to ensure that an incorrect set is not maintained due to representation errors. A suitable value for ϵ would be 10^{-5}. When a new pair (x, p) is added to $SafeList$ any element $(s, q) \in SafeList$ is selected and p and q compared.[8] The three outcomes are considered separately.

- $p - q > \epsilon$. If the new pair has a higher probability of containing a mine than all existing elements in $SafeList$ it is discarded and $SafeList$ remain unchanged; $SafeList \leftarrow SafeList$.

- $|p - q| \leq \epsilon$. If the new pair has the same probability of containing a mine as the elements already in $SafeList$ it is added; $SafeList \leftarrow SafeList \cup \{(x, p)\}$.

- $p - q < -\epsilon$. If the new pair has a lower probability of containing a mine than all elements in $SafeList$, all elements currently in $SafeList$ are discarded and $SafeList$ becomes the singleton containing only (x, p); $SafeList \leftarrow \{(x, p)\}$.

This implementation of $SafeList$ ensures that both adding and retrieving elements are constant time operations, rather than insertion time linear in

[8] By transitivity it is sufficient to only compare one existing pair with the new pair.

the board size if a list implementation storing all pairs was used.[9] This is significant since the insertion is performed n times during the inner loop in the worst case. If after considering each remaining square on the board no move has been made, a random element — known to be of minimal probability of containing a mine — is selected from $SafeList$ and probed. The pseudo-code for the limited search with probability estimates strategy is shown in algorithm 5.

5.4.3 Complexity

The worst-case complexity of limited search with probability estimates is the same as the basic limited search algorithm, as the probability estimates were obtained without modifying the general structure of the search algorithm. It was also showed that maintaining a list of the best probabilities can be done with both addition and retrieval as constant time operations, hence not creating extra significant process time to the strategy. There is however one issue concerning the average case performance of the PROB_SEARCH function worthy of a brief mention. The implementation of SEARCH used a feature of the Boolean result required, such that once the value $false$ was returned by one branch of the recursion no more recursive calls would be generated since at least one solution had been found.[10] This feature would reduce the average search time since only when a contradiction would be found or when the only explanation lies in the final recursive branch would the entire tree be searched. This slight gain in average case performance is not possible to obtain when counting the number of explanations since the number of explanations for each recursive branch needs to be counted. Thus, a slightly worse average case performance would be expected caused by the addition of probability estimates but there is no affect on the worst-case complexity.

[9] This could potentially be reduced to logarithmic time if $SafeList$ was implemented using an array but in that case space utilisation would not be optimal.

[10] This is due to the equivalence from Boolean algebra $false \wedge B \equiv false$.

Algorithm 5 An outline of the limited search with probability estimates strategy.

```
while not lost do
  SafeList ← {}
  moved = false
  for each x ∈ X do
    S ← {}
    Z ←ZONEOFINTEREST(x)
    O_1 ← x
    for each z ∈ Z do
      O ← O∪ADJ_NOLABEL(z)
    end for
    for each z ∈ Z do
      S_Squares(z) ← ADJ_NOLABEL(z)
      S_info(z) ←LABEL(z) − #ADJ_MINE(z)
    end for
    m = PROB_SEARCH(S, O, true)
    if m = 0 then
      PROBE(x)
      moved = true
    else
      f = PROB_SEARCH(S, O, false)
      if f = 0 then
        MARK_MINE(x)
        moved = true
      else
        ADD_MOVE(SafeList, (x, m/(m+f)))
      end if
    end if
  end for
  if moved = false then
    MAKE_BEST_MOVE(SafeList)
  end if
end while
```

	Beginner	Intermediate	Expert
Avr. wins (%)	92.5	67.7	25.0
Win variance	6.81	21.81	18.61
First zero wins (%)	97.1	77.8	30.0

Table 5.3: The performance summary of the limited search with probability estimates strategy.

5.4.4 Performance analysis

The results of testing the limited search with probability estimates strategy are summarised in table 5.3. Adding probability estimates to the limited search strategy achieves a slight increase in winning percentage to beginner level to 92.5% with a reduced variance of 6.81. At intermediate level the winning percentage increased to 67.7% with a variance of 21.81 and at expert level the winning percentage was increased by more than 40% to 25.0% with a variance of 18.61. The performance increase gained from adding probability estimates is evident, however at the advanced level our strategy remains inferior to the benchmark performance set by the CSPStrategy.

The relatively small performance increase at both beginner and intermediate level indicate that at those levels it is difficult to improve this strategy further. This is also suggested by the fact that the success rate at beginner level is 92.5%, which increases to 97.1% in games when the first probe yields a zero. The performance on both beginner and intermediate level remain significantly above the benchmark. In fact Studholme [18] conducted experiments where the ideal starting position (the square where the mine density is least) was provided by the game and failed to beat the success rates achieved by limited search with probability estimates, providing further evidence to support the conjecture that it is very difficult to improve the performance on these two levels.

Although adding probability estimates to the limited search increased the success rate at expert level by 40% we were disappointed to remain significantly short of the 34% success rate benchmark set by the CSPStrategy

Estimated probability	Trails	Sucessful	Observed probability
0.04	6337	5728	0.096
0.05	7292	6282	0.139
0.06	7764	7163	0.077
0.07	5208	4738	0.090
0.08	20860	18104	0.132
0.09	10233	9000	0.120
0.10	8903	7834	0.120
0.11	18033	15817	0.123
0.13	38223	33277	0.129
0.14	26462	22667	0.143
0.17	17278	14521	0.160
0.20	43800	35781	0.183
0.21	216323	192863	0.108
0.25	11359	9197	0.190
0.33	15981	11992	0.260
0.50	20206	10145	0.498

Table 5.4: Observed results for comparing estimated probability with observed probabilities. Only estimated probabilities with more than 5000 trails are included.

on the expert level. Furthermore, even when only considering games where the first probe yielded a zero the winning percentage remained less than the benchmark at 30%. For this reason we expect that it is possible to improve the probability estimation methods employed by the algorithm, although this may imply increasing the search area. Presently the data related to the correctness of the estimates will be analysed. Each time the strategy used the estimated probabilities to assist in guessing a move a counter associated with the estimated probability of finding a mine was incremented. If the guess was successful another counter associated with the same estimated probability was incremented as well. The results are summarised in table 5.4, where the observed probability of finding a mine is calculated as $1 - \frac{\text{successful trails}}{\text{total trails}}$. Only probabilities where more than 5000 trails were performed are included in order to let the observed probability stabilise.

Observe from table 5.4 that the estimated probabilities less than 0.10 are

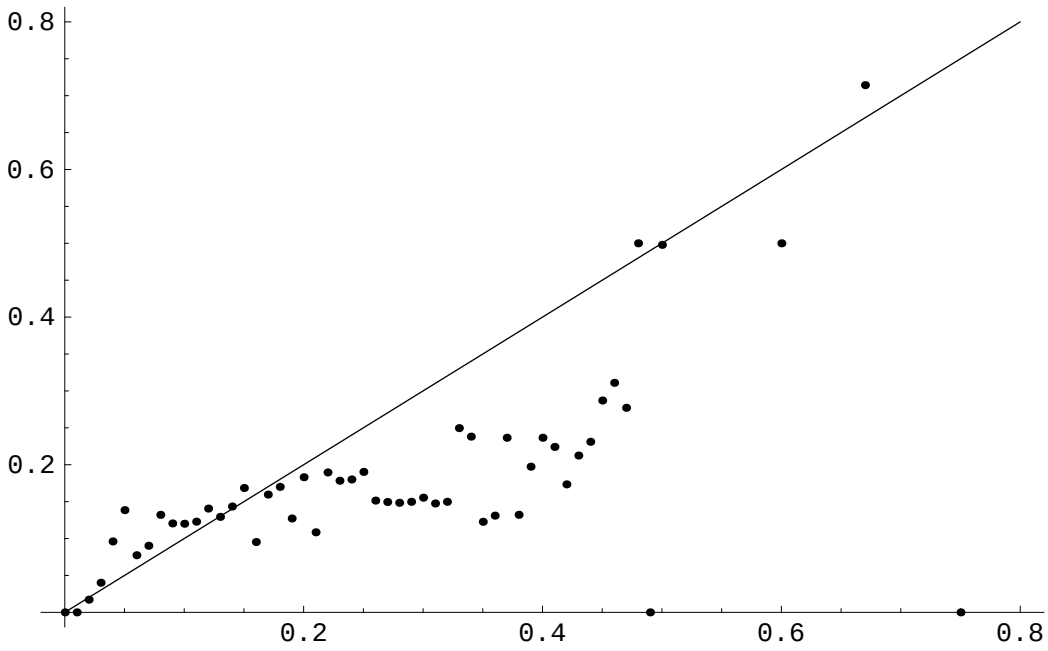


Figure 5.3: Estimation of probabilities with linear function.

generally too low, however the estimates in the region $0.10 \leq p \leq 0.20$ are close to the observed probabilities. It is interesting to note that the 0.21 estimated probability is almost twice as high as the corresponding observed probability. Also, this is the most frequently chosen probability, mainly because all squares initially have a probability of 0.20625 of containing a mine. For this reason, it is surprising that the estimated probability is so inaccurate, especially compared with the 0.13 case which corresponds to the situation where a one is revealed from the initial probe giving a 0.125 probability of finding a mine adjacent to it. Finally the 0.50 estimate is very close to the observed probability, a result we mainly contributed to the fact that once a fifty-fifty guess is the best possible option left on the board, close to total knowledge exists about the remaining squares. It is not uncommon for the last move of the game to be a random choice between two squares equally likely to contain the final mine on the board.

Further to considering the raw data only, figure 5.3 shows all the data

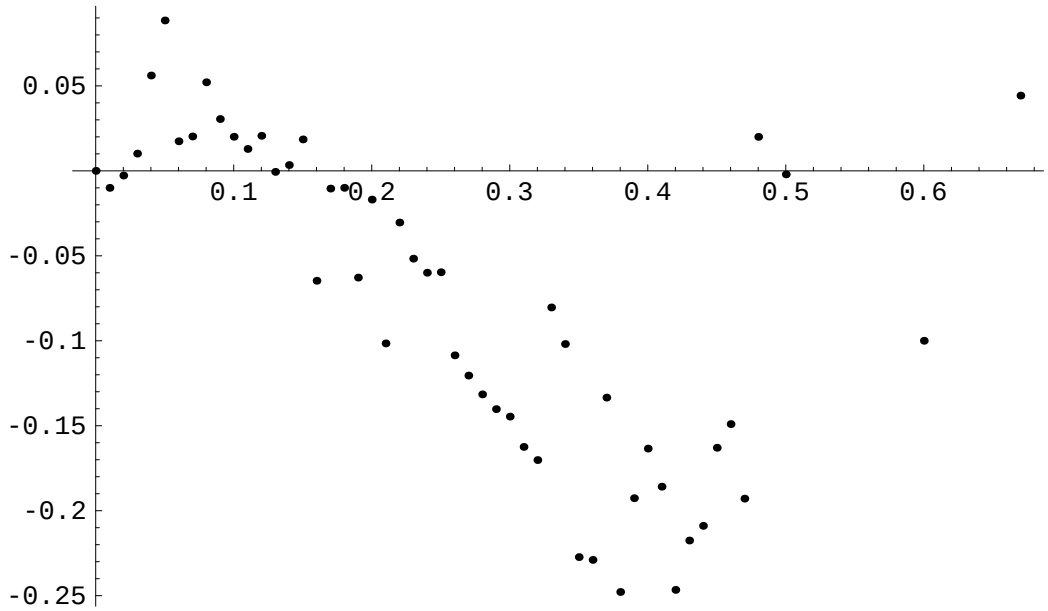


Figure 5.4: The difference between the observed probability and the estimated probabilities.

point plotted as the estimated probability versus the observed probability. To aid interpretation, $f(x) = x$ is plotted since ideally all the data points should represent that function. It is helpful to consider that all points above the line result from underestimating the probability and the points below the line from overestimating the probability. The line is a reasonable fit to the data, especially when not considering the two most noticeable outliers on the x-axis, which are both a result of very small number of successful trails (1 trail at $p = 0.49$ and 3 trails at $p = 0.75$). The closeness of the observed probabilities to the estimated probabilities is also shown in figure 5.4 which shows the estimated probability versus the difference between the observed probability and the estimated probability. The two outliers are removed from this figure, and the fit is relatively close to the x-axis. The region which is overestimated by approximately 0.2 is a result of relatively few observations for each data point.

5.5 Further experiments with strategies

Adding probability estimates to the limited search strategy resulted in the best Minesweeper strategy developed in this project. In this section the selection of the search area and its effect on the performance of the strategy will be analysed. We will experiment by gradually incrementing the search area as a square and note the effect this has on the strategy's win ratio. Other than the simple win ratio the distribution of the search size will be investigated in order to determine whether continually increasing the search area has any significant effect on the performance.

5.5.1 The size of the search area

Six square sizes were tested and will be identified by their 'radius', the shortest distance (number of squares) from the central square to the edge of the interest area. The squares ranged in radius from two to seven. New data was also collected for the zone of interest search area to enable comparison based on the same number of statistical trails. As the runtime is exponential in the size of the search area, only 1000 games were played by each strategy, although for $r = 7$ only 600 were managed. Prior to performing each search, the number of squares in the search area was recorded. It is important to note that the interest area contains both unknown and labelled squares, but since only unknown squares can be assigned a new value they constitute the size of the search area. Furthermore if an unknown square has no labelled neighbours, it will not be useful to consider this square a part of the search area. The recorded size of the search area thus only contains the squares that could usefully provide information about the board through assignments.

The results are displayed in table 5.5 which is sorted by the average size of the search area. It is important to note that only 1000 games were played implying a large variance of the win percentages, a fact that explains the slightly higher winning percentage of the zone of interest search here than previously presented. It is interesting to note that the zone of interest has

	Avr. size	Size variance	Games won (%)	Largest area
Zone	8.47	7.96	26.3	24
$r = 2$	9.04	9.37	24.8	26
$r = 3$	11.56	17.96	26.5	36
$r = 4$	14.06	29.56	27.6	44
$r = 5$	16.46	44.57	30.0	56
$r = 6$	18.76	61.38	27.2	64
$r = 7$	20.92	82.55	28.0	66

Table 5.5: The average size of the search area and winning percentages for playing 1000 games (only 600 for $r = 7$).

a smaller average search area than $r = 2$ but is more successful in winning games, a result we expected given the careful choice of search area employed by the zone of interest search. As expected, the average size increases gradually with the radius, and the variance becomes increasingly larger. Similarly the win percentage increases steadily despite not being completely monotonic as expected. Again we consider the small sample size and resulting large variance the reason for the slightly unexpected results. Note in particular that the $r = 5$ case twice won 37 of the 100 games played in a set which is the largest number of wins observed from any strategy. It is expected that the data would become monotonic when the sample size is increased sufficiently, and we take that as a strong indication that it is indeed possible to significantly improve the strategy by considering a larger area of the game board. Finally it is noted that the largest area that was searched was 66 squares causing a worst case of 7×10^{19} recursive calls.

While a significant performance improvement through expanding the search area was observed it is important to consider the time constraints involved. While playing 1000 games using the zone of interest is relatively fast (a matter of minutes), using $r = 5$ took approximately eight hours when running as the sole process on a Pentium III with 256 MB RAM and 1000MHz. As the search time increases exponentially with the size of the search area, the collection of the presented data took approximately 5 days of continuous pro-

cessing on the author's own PC along with multiple processes executing on a university server for part of the time. These rough estimates are sufficient evidence for us to consider algorithms that search a large area unsuitable for the collection of significant amounts of statistics, however for playing and viewing a single game they are often successful.

5.5.2 Searching the complete board

As a final experiment a set of games that considered the entire board at each move, hence playing a perfect game, was played. The only level where this is practically possible, judging from the results of the previous experiments, is the beginner level where it is reasonable to estimate that the search area will only uncommonly exceed 50 squares.

10000 games were played on a beginner level board with $r = 10$ and this failed to improve the performance from the zone of interest result previously presented winning 92.4% of the games with a variance of 6.91. Other than indicating that the zone of interest search is close to optimal at beginner level it also underlines our claim that it is not possible to create a strategy that wins *every* game since some configurations are not explicitly solvable. Despite not improving the performance of the strategy, the search done with $r = 10$ had an average search area of 16.44 squares compared with only 7.75 when using the zone of interest. The playing of 10000 games using $r = 10$ took close to eight hours, compared with a few minutes using the zone of interest. It is interesting to see that searching the entire board does not necessarily improve the performance, but due to its complexity it is very costly in terms of processing requirements.

Chapter 6

The Minesweeper analyser

This chapter covers the development of the 'Minesweeper analyser', an application used for implementing Minesweeper strategies. The project is not a Software Engineering project and as a result this chapter is not intended to document all design issues encountered rather it highlights the overall program structure and provides the necessary information for readers wishing to implement their own strategies.

6.1 Software requirements

The requirements for the Minesweeper analyser were detailed in the project specification document [14] and the most important features are summarised as follows:

- A graphical game board with the option to show or hide the mines,
- A summary panel showing the statistics for the current player,
- The ability for a human to create a game board graphically,
- Saving and loading created configurations,
- The possibility of to implementing automated players,
- An option to play several games without interaction, and

- A command line option to play several games without any graphical output and record the number of games won.

6.2 Design considerations

During the software design several issues were considered and decisions were made to tailor the application to comply most effectively with the requirements, while aiming to maintain both design consistency and implementation simplicity. A selection of design issues are now summarised but details of several trivial design considerations are omitted.

6.2.1 Language choice

The Minesweeper analyser is implemented as a Java application. Java was chosen to obtain the advantage of platform portability so both university servers and the author's personal equipment could be used for the collection of statistics. As the language choice was made prior to the software design, several design decisions were made considering implementation issues. Issues such as abstract classes, multithreading, events and exception handling will thus be considered during the software analysis and design.

6.2.2 Standardisation of strategy implementation

Providing a framework with an easy interface for implementing Minesweeper strategies was a top priority and received much consideration during software analysis. An informal template for a Minesweeper strategy was created from which the interaction methods a strategy would require of the game board, and the behaviours general enough to be abstracted out of each specific implementation could be extracted. The template was formalised to an abstract class, since its specified behaviour in situations such as statistics collection and communicating with a central control unit was general enough to be implemented independent of the game playing part of the strategy.

This approach led to an easily accessible abstract class, which is required to be extended by all Minesweeper strategies, hence enforcing a simple yet powerful framework of ensuring software standardisation. The template was also useful in designing a concise API for the game board component, the only class a Minesweeper strategy needs to interact with.

6.2.3 Safety board information

An interesting issue is the problem of reducing the amount of board information available to the developer of a strategy. Clearly the game board needs to contain the actual mine locations and that this information should not be accessible to a strategy but is required by the central control unit. It was noted that not only should the mine locations be hidden from a strategy, but also methods causing the board to be reset or otherwise modified.

This problem was solved by adopting a two-level game board design in which the game board along with the information intended to be visible to strategies made out the lower level, and the ability to alter the game board and obtain the location of the mines were placed at a higher level. Using this approach the game board could be encapsulated as two different classes depending on the intended purpose, and an author of a Minesweeper strategy would not be able to obtain extra information about the board. Furthermore, several methods of the abstracted player were marked with the `final` keyword in order to prevent overwriting of methods providing the statistical results.

6.2.4 No graphics option

An important feature of the software is the ability to play several games without invoking a GUI that would slow down the computation considerably. Since only the main class of the GUI is required to be held in reference by the control unit, this class was replaced by an `interface` containing signatures of all required methods. The graphics implantation would implement these

methods fully and thereby invoke a GUI, however a no graphics option was easily crated by implementing this interface with empty methods. Hence the user would be able to specify which implementing class was used by a command line option.

6.3 Design

The major design issues of the Minesweeper analyser are now covered. A high level design of the application is presented in the form of a class diagram, and some low level design issues involving exception handling important for understanding the requirements for implementing a Minesweeper strategy are discussed. The function of each major class in the framework will be summarised, however only the classes that require direct interaction or implementation when implementing a strategy will be described in significant detail.

6.3.1 Design overview

The software was designed with one centralised control component, the `MinesweeperEngine`, as shown in figure 6.1. The two most important classes are `Player` and `Board` representing an abstract player and the game board respectively. `Board` implements the rules of Minesweeper and provides the required methods for a `Player` to interact directly with it. Note that `MinesweeperEngine` holds reference to an `AdminBoard` which extends `Board`. `AdminBoard` contains methods for modifying the layout of the board and information about the locations of mines can be retrieved from this information. Hence it needs to be invisible to the `Player` which only holds reference to the actual game board. To allow the `MinesweeperEngine` to carry out tasks such as reacting to user inputs or updating the UI while a game is being played, `Player` is executed in its own thread implemented by the `PlayerRunner` class. The GUI is controlled by the `MinesweeperEngine`, which receives game information in the form of events from both `Board` and `Player` and uses them in conjunction with the

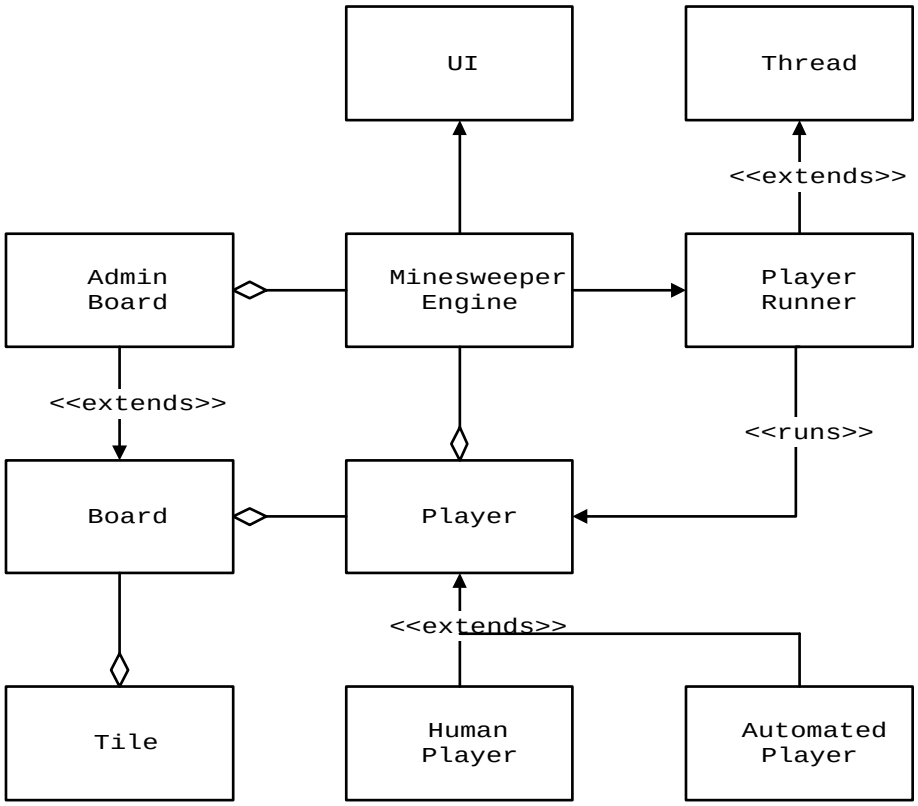


Figure 6.1: Class diagram for the Minesweeper analyser.

`UI` package.

6.3.2 The `Board` class

The `Board` class implements the rules of Minesweeper and maintains the state of the game board. The game board is a 2-dimensional array of `Tile` objects which contain information about each individual square of the board, such as its current label and whether it contains a mine. `Board` has a public method to retrieve the label of a particular square which is the only information available to a `Player` about a square. `Board` also contains public methods for obtaining the dimensions of the board, the total number of mines and the number of remaining mines.

The only form of interaction a `Player` can have with the `Board` is to

perform a move which can be either a 'probe' or placing a label on a square. `Board` has a method for each type of move, which takes the square as parameter and if it is a labelling move also the label. The `doMove` method throws four exceptions which need to be handled by the `Player` class. These are `GameWonException`, `GameLostException`, `GameInterruptedException` and `InvalidMoveException`. The first two are thrown when the game is completed with a loss or victory to the player as appropriate, the third is thrown when the user interrupts the game, and the `InvalidMoveException` is thrown when the strategy attempts to perform an invalid move such as probing an already probed square or specifying coordinates outside the game platform.

6.3.3 The `Player` class

`Player` is an abstract class representing a Minesweeper strategy. `Player` is mainly responsible for handling the two exceptions `GameWonException` and `GameLostException`, and use them to maintain statistics for the implemented strategy. A strategy is implemented by extending `AutomatedPlayer`, an abstract subclass of `Player` which handles user generated events. `AutomatedPlayer` contains one abstract method, `play`, which needs to be implemented as a strategy. `play` receives the game board as its only parameter and throws `GameWonException`, `GameLostException`, and `GameInterruptedException` which means that the `InvalidMoveException` needs to be handled by the implementing strategy.

6.4 Implementation and testing

This chapter is concluded by a discussion of some issues encountered during the implementation and testing of the software. Following the software design an iterative implementation strategy was employed. This method was preferred since the design has revealed several distinct modules that could be implemented and tested separately, hence ensuring that they functioned

properly before integrating them. The components that were developed first were the basic game playing frameworks such as `Board` and `Player`. For testing purposes a randomised player was implemented since it was very easy to program and adequately meet the needs of being able to test the basic functionality of the framework including a class extending `AutomatedPlayer`. A basic GUI was also implemented early, as the aid of a graphical realisation of the implemented strategies was an invaluable debugging tool for the development of game playing strategies. Following the basic framework, the specified features were implemented and with them the GUI functionality.

The source files were maintained using IntelliJ Idea. This provided a useful resource for maintaining consistency in following naming conventions and allowed easy access to several source files simultaneously for debugging purposes. The automatic suggestion of available methods and variables along with a continuous syntax checker were extremely useful in limiting typographic errors and hence greatly reduced the time spent on trivial debugging matters and allowed us to focus on the logical testing and debugging of the software.

Due to a thorough analysis and design, each module had well defined easily observable functionalities which made testing of each one fairly straight forward. An advantage of developing the framework of a game is that the functionality is easy to observe visually and hence the bugs were relatively easy to trace. The more abstract areas such as loading stored configurations where less easy to debug, although the end results were easily observable.

Chapter 7

Conclusions

The broad areas of the project — the complexity of Minesweeper (including configurations), the development of game playing strategies, and the design and implementation of the Minesweeper analyser — will be concluded separately and further work in each area will be mentioned. Finally the project as a whole will briefly be reflected upon.

7.1 The complexity of Minesweeper

In this project three important complexity results for Minesweeper were presented. Both the problems of determining whether a unique explanation to a configuration exists and whether a given move is safe are complete in **DP**, while counting the number of explanations is complete in **#P**. These results are significant restrictions of game playing strategies, and show that playing a perfect game of Minesweeper is intractable. Although Kaye's work provided the framework of the proofs, the work presented does not follow immediately from the **NP**-completeness of CONSISTENCY. For example is it important to realise that the configurations constructed by Kaye were not sufficient to prove **#P**-completeness since the AND does not have the required property of uniqueness to create a parsimonious reduction.

Although several game playing questions have been raised and answered

in this project there is still much interesting research that could be carried out on the complexity of Minesweeper. One area that would be particular interesting due to its very close connection with playing a game is an investigation of the complexity of similar problems to SOLUTION but with the restriction that the original configuration is consistent. In this case it would not be as intuitive to show membership of **DP** for the problem considered here.

Another area of further work, which is closely related to the development of Minesweeper strategies, would be a study of how small a search area can be selected whilst maintaining close to perfect information. In particular some theoretical bounds for the size of an optimal area would be useful, and it would be interesting to discover how close the zone of interest approach is to the optimal selection of game area. This is obviously a very challenging area of research, but we believe that the Minesweeper analyser would be useful for implementing approximation algorithms that could aid this investigation.

7.2 The development of Minesweeper strategies

Limited search with probability estimates was the best Minesweeper strategy developed. The performance at both beginner and intermediate level was significantly higher than the benchmark, although remaining inferior at expert level. While the complexity is quadratic in the board size, the runtime of the strategy is heavily influenced by the size of the search area. Although this is virtually a non-issue when playing a single game displayed graphically, the relatively slow runtime meant that collecting sizable statistics about the strategy was time consuming and required significant planning. We are generally pleased by the strategy development, especially considering that all strategies have been developed independently of any third party, and only at expert level does our best strategy not outperform the best third-party strategy we are aware of.

Despite of the successful development of Minesweeper strategies, we believe that this is a very interesting area in which to conduct further research. One obvious area that would benefit greatly from further research is the probability estimation, since it was clear from the statistics that it is not yet perfected. This is an interesting problem since it was proved **#P**-complete and hence approximate counting techniques would be required. It might also be interesting to use Minesweeper as a test platform for approximate counting algorithms for other **#P**-complete problems, since the estimated probabilities can be easily compared with observed probabilities. It would also be interesting to apply known SAT heuristics to a logic representation of a Minesweeper configuration and compare their performance with the strategies developed specifically for Minesweeper.

Finally, investigating further methods for improving Minesweeper strategies remain an interesting task in it self. One possible method to improve the average case performance of a strategy could be to investigate methods for discovering several safe moves during a single search or combining the constant time detection used in the single point player with a more advanced search strategy at each move. We feel that this is an area of algorithmic development that is both challenging and rewarding, since it allows us to attempt solving an **NP**-complete problem visualised as an entertaining game and it is rewarding to develop strategies more successful than one self at playing the game.

7.3 The Minesweeper analyser

The Minesweeper analyser proved a powerful application for the implementation and analysis of the developed Minesweeper strategies. It benefited from a simple interface, which allowed standardisation of strategy implementation. As the development of the application was a relatively low priority there are several ways in which it could be improved. These improvements are however more a way of improving the application into a program suitable for

distribution on The Internet rather than the foundation of future projects. The Minesweeper analyser could be improved in the following areas:

- Improve the functionality of the "save configuration" function to allow randomly generated configurations to be saved.
- Provide a plug-and-play interface to compile a strategy stored in a separate file and executing it without needing to update the source code of Minesweeper analyser.
- Improve the safety features implemented to further ensure that a strategy implementer cannot obtain vital board information or handle the end of game exceptions to manipulate the recorded result.
- Improve the usability features of the application.

7.4 General project conclusions

It is felt that the project has achieved the objectives initially defined as several results about the complexity of Minesweeper were obtained along with the completed development and implementation of three Minesweeper strategies and the game framework. It is worth pointing out again that all the work presented, with the exception of the definition of a zone of interest and Kaye's theorem, is original and that no other strategy has been identified that is more successful on both beginner and intermediate level than our best method.

Acknowledgements

I thank the group members James Bell, Chris Bright, and Alex deSousa for their interest and suggestions during the weekly meetings. I also thank Kati Kysenius for never ending support and Christian Romming for asking several difficult questions about my ideas, thereby helping me to better understand and explain the theoretical topics. Finally, I specially thank Dr Leslie Goldberg for supervising the project and in particular for checking the complexity proofs and providing technical suggestions for improvement or correction of the arguments.

Bibliography

[1] Andreas Blass and Yuri Gurevich. On the unique satisfiability problem. *Information and Control*, 55:80–88, 1982.

[2] Gilles Brassard and Paul Bratley. *Foundations of Algorithmics*. Prentice Hall, New Jersey, 1996.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories, Inc., 1978.

[4] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Longman, 2nd edition, 2001.

[5] Cay S. Horstmann and Gary Cornell. *Core JAVA2: Volume I-Fundementals*. Sun Microsystems, California, 2001.

[6] Cay S. Horstmann and Gary Cornell. *Core JAVA2: Volume II-Advanced Features*. Sun Microsystems, California, 2002.

[7] Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[8] Richard Kaye. Some Minesweeper configurations. Available from http://web.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm, August 2000.

[9] John Kelly. *The Essence of Logic*. Prentice Hall, 1997.

[10] Damien Moore. The authoritative minesweeper. http://metanoodle.com/minesweeper/. [Last accessed on March 30, 2004].

[11] C. H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences*, 28:244–259, 1984.

[12] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[13] Lourdes Peña Castillo and Stefan Wrobel. Learning minesweeper with multirelational learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 533–538. Academic Press, August 2003.

[14] Kasper Pedersen. The complexity of minesweeper and strategies for game playing: Project specification document. December 2003. Available at http://www.dcs.warwick.ac.uk/ csviq [Last accessed on April 20, 2004].

[15] John D. Ramsdell. Programmer's minesweeper. http://www.ccs.neu.edu/home/ramsdell/pgms/. [Last accessed on March 30, 2004].

[16] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, 2nd edition, 2003.

[17] Ian Stewart. Mathematical recreations: Be a Minesweeper millionaire. *Scientific American*, 283(4):94–95, October 2000.

[18] Chris Studholme. Minesweeper as a constraint satisfaction problem. April 2001. Available at http://www.cs.toronto.edu/ cvs/minesweeper [Last accessed on April 20, 2004].

[19] Leslie G. Valiant. The complexity computing the permanent. *Theroetical Computer Science*, 8(2):189–201, 1979.

[20] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3):410–421, August 1979.

[21] Frank Wester. The minesweeper page. http://www.frankwester.net/winmine.html. [Last accessed on March 30, 2004].

Appendix A

Instructions for using the CD

This is a copy of the README.TXT file found on the submitted CD.

A.1 CD overview

The CD contains two folders, source and compiled. The source folder contains the source code for the Minesweeper analyser and can be viewed with any text editor. The compiled folder contains the compiled version of the software and can be executed directly from the CD.

A.2 Executing the Minesweeper analyser

In order to execute the Minesweeper analyser in graphics mode, no command line options are required, i.e. the command is `java Minesweeper`. The `-verbose` option is optional and if used debug information will be written to the screen.

In order to play several games without invoking a graphics window, the `-g` option is used. The `-g` options takes six integer parameters: width, height, number of mines, number of games, strategy, rules. The strategy parameter is used as follows:

1. Single point strategy
2. Limited search strategy
3. Limited search with probability strategy
4. Limited search with probability strategy, r=2
5. Limited search with probability strategy, r=3

6. Limited search with probability strategy, r=4
7. Limited search with probability strategy, r=5
8. Limited search with probability strategy, r=6
9. Limited search with probability strategy, r=7
10. Limited search with probability strategy, r=10

The rules are coded such that 0 means that a loss can occur on the first move, and 1 means that the first move is safe.

Furthermore the option `-f` can be used to write the results to a file along with extra statistics collected such as the average size of the search area. `-f` takes the filename as parameter. By default the results are only written to file at the end of the computation, but the `-write` option forces results to be written after every 100th game. The options `-noResults` and `-noCount` prevent output of the outcome of each game and the number of the game currently played to be displayed on the screen.

For example the command `java Minesweeper -g 10 10 10 1000 1 -f testfile -noResults -noCount` will play 1000 games at beginner level with first move safe rules and write the results to `testfile`. No output will be generated in the command window.