Algorithmic Approaches to Playing Minesweeper

Algorithmic Approaches to Playing Minesweeper

A thesis presented by

David Becerra

to
Computer Science
in partial fulfillment of the honors requirements
for the degree of
Bachelor of Arts
Harvard College
Cambridge, Massachusetts

April 1, 2015

Abstract

This thesis explores the challenges associated with designing a Minesweeper solving algorithm. In particular, it considers how to best start a game, various heuristics for handling guesses, and different strategies for making deterministic deductions. The paper explores the single point approach and the constraint satisfaction problem model for playing Minesweeper. I present two novel implementations of both of these approaches called double set single point and connected components CSP. The paper concludes that the coupled subsets CSP model performs the best overall because of its sophisticated probabilistic guessing and its ability to find deterministic moves.

Contents

Chapter 1

Introduction

From soccer to Sudoku, Go to Connect Four, games have fascinated and challenged mathematicians and computer scientists for centuries. Oftentimes, a game may have a straightforward premise or a simple set of rules yet when scholars analyze it closer, a plethora of complex and intriguing problems emerge. It is this deceptive and sometimes unexpected depth that draws scholars to rigorously study games.

When researching games, scientists answer questions from a wide range of subject matters. For example, IBM's Watson is a supercomputer programmed to play Jeopardy. Designing Watson required sophisticated database management, optimized search algorithms, and complex language processing techniques. Integrating these features into a robust network of computers was a tremendous accomplishment and it provided researchers with valuable information.

Chess, on the other hand, has had a long legacy in computer science. For decades, programming a computer to successfully beat humans at chess was seen as the pinnacle of artificial intelligence. Now, there exist supercomputers which manage to beat the world champions of chess.

Minesweeper is another example of a game with a simple set of rules yet challenging implications. In fact, Minesweeper is in a class of mathematically difficult problems known as co-NP-complete. Therefore, understanding the complexity of Minesweeper and designing algorithms to solve it may prove useful to other related problems.

In this paper, I will analyze different approaches to designing an algorithm to play Minesweeper. I will first provide a detailed overview of the game followed by an introduction to key terminology in Chapter 2. Chapter 3 reviews various contributions to

the study of Minesweeper. In particular, I will consider the computational complexity of Minesweeper. I will describe the formal decision problem associated with Minesweeper and explain why it is co-NP-complete. Afterwards, I will explore specific groups of existing algorithms.

Chapter 4 will explore a few of the key design challenges a Minesweeper solver must consider. Furthermore, the chapter will define the various metrics utilized to evaluate a solver. In doing so, I will establish methods for determining a good solver.

Finally, this thesis will analyze in depth the single point and constraint satisfaction problem models to solving Minesweeper. I will provide the standard algorithms for each model before presenting my own implementation. In particular, I develop a double set method for the single point strategy which resolves an ordering problem associated with the standard algorithm. Additionally, I use the double set approach to further explore the design challenges discussed in Chapter 4. For the constraint satisfaction method, I provide an alternative technique to partitioning the search space which yields equivalent subproblems as the standard algorithm. Each solver will play 100000 Minesweeper games of various difficulties, sizes, and mine densities. Through these tests, I explain why the constraint satisfaction problem approach is successful.

Chapter 2

Background

2.1 What is Minesweeper?

Minesweeper is a single-player puzzle game available on several operating systems and GUIs. At the start of a game, the player receives an $n \times m$ rectangular grid of covered squares or cells. Each turn, the player may probe or uncover a square revealing either a mine or an integer. This integer represents the number of mines adjacent to that particular square. As such, the number on a cell ranges from 0 to 8 since a cell cannot have more than eight neighbors. Figure 2.1 provides a simple example of a numbered square and its covered neighbors. The game ends when the player probes a cell containing a mine. The objective of the game is to uncover every square that does not contain a mine.

FIGURE 2.1: An example of a numbered square. The **3** indicates that exactly three of the eight neighboring squares contain a mine.

Playing Minesweeper involves a fair amount of logic. A clever player will use the numbered cells to deduce the location of mines. For assistance, most implementations

of Minesweeper allow the player to mark or flag possible mine locations. However, this is simply for bookkeeping as the game does not validate any flagged squares. Higher difficulties of Minesweeper involve a greater degree of deductive reasoning as the mine density (number of mines over number of cells) increases. Oftentimes, mines cannot be deterministically located, and so the player must resort to guessing. As a player, guessing may seem frustrating; however, as this paper will explore, guessing leads to interesting challenges when designing a Minesweeper solver.

FIGURE 2.2: Two squares flagged by the player. Note: uncovered squares with a value of 0 are grayed out in this particular version of Minesweeper. Blue squares represent covered cells.

There are three difficulty levels for Minesweeper: beginner, intermediate, and expert. Beginner has a total of ten mines and the board size is either 8×8, 9×9, or 10×10. Intermediate has 40 mines and also varies in size between 13×15 and 16×16. Finally, expert has 99 mines and is always 16×30 (or 30×16). Typically in beginner, guessing is rarely necessary. The numbers on squares tend to stay in the ones and twos with the occasional three. As the difficulty increases, guessing becomes more common with expert configurations having numerous instances of guessing. Furthermore, higher numbered cells are more prevalent; however eights or sevens are still uncommon.

2.2 Basic Definitions

Throughout this paper, I will be using unique terminology to describe Minesweeper and its related problems. Therefore, this section will focus on defining key concepts

and vocabulary. Certain terms related to a specific idea may be explained as they are introduced.

Minesweeper boards are comprised of squares, cells, positions, or points. The board or grid itself may also be called a configuration.

Definition 2.1. (Configuration) *A Minesweeper configuration is a grid, typically rectangular, of possibly covered squares that may be partially labeled with numbers and/or mines.*

A configuration can be thought of as the state of a Minesweeper game, including all the numbers, marked mines, and covered squares. A ***solution*** to a configuration is an assignment of mines to the covered cells which gives rise to a consistent Minesweeper grid.

Definition 2.2. (Consistent) *A Minesweeper board is said to be consistent if there is some assignment of mines to the blank/covered squares that gives rise to the numbers shown.*

Determining if a configuration is consistent gives rise to a unique problem in Minesweeper known as CONSISTENCY, which will be discussed in Chapter 3.

If a player believes a square contains a mine, he/she is allowed to place an indicator on that cell. These are called **marked or flagged squares**. Throughout this paper, I will assume an ideal player; he/she only performs moves that he/she knows to be correct. In other words, I will assume that all flagged cells correctly identify squares with mines. Note from the previous section that in general this may not be true; players do not receive confirmation or feedback when they mark a position. In contrast, an **unmarked square** is one that has no flag or mark on it to denote it has a mine.

A **covered, unprobed, or unknown cell** is a position whose contents are unknown to the player. On the other hand, the contents of an **uncovered or probed cell** are available to the player. Lastly, a **mine-free** or simply **free cell** is one that does not contain a mine; it typically refers to a covered square that the player can safely click or probe.

The number on a free square is the position's **label**. Oftentimes an algorithm like a single point solver is concerned with the **effective label** of a point. As previously mentioned, the label of a cell refers to the number of mines adjacent to that cell. However,

Figure 2.3: The square with label 4 has three marked neighbors. Therefore, the effective label of this square is 1.

if neighboring positions have already been marked as mines, the number of mines still unaccounted for is the effective label. Mathematically, for some square x,

$$\mathsf{EffectiveLabel}(x) = \mathsf{Label}(x) - \mathsf{MarkedNeighbors}(x) \tag{2.1}$$

The effective label tends to be more useful when solving Minesweeper configurations since we are more concerned about what we have left to find.

Figure 2.4: A probed square and its second order neighbors.

Lastly, the neighbors of a cell x are the eight positions immediately adjacent to x. Certain algorithmic approaches, however, are concerned with squares in a larger neighborhood around a point. The **k-th order neighbors** of x are all the cells that are at most k squares away from x. For example, the second order neighbors of x would be the 24 squares that form a 5×5 grid centered at x.

Chapter 3

Related Work

3.1 Minesweeper Complexity

3.1.1 Consistency

In 2000, Richard Kaye suggested that Minesweeper was a computationally difficult game. In particular, Kaye analyzes a specific problem that players face called MINESWEEPER CONSISTENCY or simply CONSISTENCY in his paper "Minesweeper is NP-complete" [1]. CONSISTENCY is a decision problem which asks whether a Minesweeper configuration is consistent.

Kaye begins his paper by first demonstrating that CONSISTENCY is in the complexity class NP using the verifier-based definition. A certificate for an instance of CONSISTENCY is an assignment of mines to the covered cells. The verifier checks that these mines produce the numbers on the board. Performing this verification takes polynomial-time with respect to the input size (i.e. number of total squares in the grid). Therefore, CONSISTENCY is in NP.

Next, Kaye argues that CONSISTENCY is in fact NP-complete. He does so by creating a wire construct and, subsequently, Boolean logic gates from Minesweeper configurations. With a well-defined notion for logic gates, Kaye is able to provide a polynomial-time reduction from SAT to CONSISTENCY.

3.1.2 Inference

It is important to note that CONSISTENCY does not imply that Minesweeper itself is NP-complete, as suggested by the title Kaye's paper [1]. Although one could use the solution of CONSISTENCY to play a complete game of Minesweeper, this does not exclude the existence of more efficient strategies. Allan Scott et al. introduce a separate decision problem which is integral to playing Minesweeper: MINESWEEPER INFERENCE or INFERENCE for short [2]. INFERENCE asks whether there exists at least one covered square whose contents can be safely inferred given a consistent configuration. In other words, given an instance of the game, can one deduce with certainty the location of a mine or a free square? From this definition, it quickly follows that all deterministic attempts to play Minesweeper must try to solve INFERENCE; to make a move an agent must decide if a safe action can be made, resorting to a guess only if none exist.

INFERENCE is a promise problem as it only considers consistent grids. However, the solvers discussed in this paper only consider consistent game instances. Furthermore, players assume configurations are consistent when playing Minesweeper, so this is a reasonable restriction.

Nevertheless, Minesweeper is still a hard game to play. To arrive at this result, Allan Scot shows that INFERENCE is co-NP-complete [2]. He does so by first explaining how INFERENCE is in co-NP. Consider the No-instances of INFERENCE. A proof of a No-instance is a collection of two consistent boards for each covered square s: one where s is a mine and one where s is not a mine. The existence of these consistent boards with opposite assignments proves that a deterministic inference cannot be made since each covered square has multiple valid assignments. To verify this proof, one simply needs to iterate through each covered square s in each possible board and determine that s is consistent with its eight neighbors. Given a board of size $n \times m$, this takes $O(n^2m^2)$ time or polynomial time. Therefore, INFERENCE is co-NP.

In a similar fashion to Kaye's proof, Allan then proves that INFERENCE is in fact co-NP-complete by defining circuit constructs from partial Minesweeper tiles [2]. Allan creates wires, logical operators (AND and NOT gates), and terminals allowing him to create any Boolean circuit. Unlike Kaye, however, Allan must transform a Boolean formula F in such a way that F is unsatisfiable exactly when the final Minesweeper

circuit board $B_{F'}$ is a Yes-instance of INFERENCE. The reduction runs in polynomial-time since the board $B_{F'}$ is proportional to the number of literals in F. As a result, Allan is able to conclude that INFERENCE is co-NP-complete. Thus, despite not being NP-complete, solving a consistent game of Minesweeper is still computationally difficult.

3.2 Algorithmic Approaches

Early algorithms developed to solve Minesweeper focus on the deterministic deductions needed to uncover safe moves. Adamatzky modeled Minesweeper as a cellular automaton or CA [3]. The cell state is given two components. The first component indicates whether the cell is either covered, uncovered, or a mine. The second component, which is only available to the system if the cell is uncovered, is the number of mines adjacent to that particular cell. One limitation to the CA model is in the transition function which only accounts for two basic deductions in Minesweeper. However, its main weakness is its tendency to become stuck on particular configurations. Since the CA solver is deterministic, it never makes guesses or random moves, an occasionally necessary step in completing a game especially at harder difficulties. Adamatzky expressed the need for stochastic features necessary for future algorithms. Section 5.1 discusses this approach in more detail.

A single point algorithm was mentioned by Kasper Pederson [4], although it was known beforehand, specifically by the author of Programmer's Minesweeper or PGMS[1]. Although not mentioned here, section 5.2 will analyze the single point algorithm in PGMS in more detail. Nevertheless, Kasper's single point method determines safe moves deterministically by looking at individual squares, similar to CA. However, if no safe moves are discovered, a square is probed uniformly at random. This prevents the algorithm from becoming stuck. Nevertheless, this approach is still too naive which is underscored by the poor performance on harder difficulties where guessing becomes more prominent.

Kasper Pederson offers an alternative strategy called limited search. This method utilizes depth-first search and backtracking on a small zone of interest around uncertain squares, an idea suggested by Peña and Wrobel [5]. Since deductions often involve constraints placed on squares that are not direct neighbors, this technique can deduce more

[1] http://www.ccs.neu.edu/home/ramsdell/pgms/

safe moves than the single point counterpart. Furthermore, Pederson added probability estimations that allowed for "smart" guessing. By collecting solutions from the limited search, estimates could be made about the probability of mine locations. Squares determined to have a low probability of being a mine would then be chosen as smart guesses, an improvement from simply probing uniformly at random [4].

A slightly different approach models Minesweeper as a constraint satisfaction problem (CSP). In this model, each square is a variable with values 0 or 1, mine-free or mine, respectively. When a square is probed, it places a constraint on its neighbors. A solution to the constraint is an assignment of mines to the neighbors of a square. The CSP model maintains a set of constraints that reflects the current uncovered information from the board. Trivial constraints are identified right away which potentially reveals more information from the board. Once the algorithm has a set of non-trivial constraints, all the possible solutions for the constraints are aggregated. These solutions serve two main functions. One, they can identify new squares to probe or mark as mines. Second, they allow for probability estimations of mine locations, similar to limited search. Chapter 6 will discuss the CSP model of Minesweeper in more detail [6].

Other approaches to solvers involve modeling Minesweeper as a partially observable Markov Decision Process (POMDP). Several strategies that do so utilize a Monte-Carlo Tree Search technique formalized by Rémi Coulom [7]. In fact, one approach involves combining CSP solvers with Upper Confidence Trees (UCT) [8].

Chapter 4

Algorithmic Considerations

This chapter will examine two important design challenges a Minesweeper solving algorithm must consider to be successful. These features are applicable to all solvers as no assumptions are made about the underlying approach the player is using. Lastly, the final section will discuss the metrics that subsequent chapters use to assess an algorithm. In doing so, the section defines what a *good* solver is.

4.1 First Move

The first click of a Minesweeper game deserves special attention since it is rather unique. Namely, the opener is always a guess because the player starts with a covered board. In addition, the challenges associated with the opening move are unavoidable and are shared among all solvers. Thus, finding an optimal policy for dealing with the initial click will benefit every approach.

When successfull probing a square, the player reveals information about the contents of the probed cell as well as the constraint the cell places on its adjacent positions. For example, knowing cell x is a 4 not only tells you what x is, but it also suggests that x's neighbors can only have a total of four mines. Ideally, the player wants to maximize the information revealed on any given click.

Thus, consider the different outcomes of probing a square x assuming x is not a mine. Suppose x contains a number n such that $n \in \{1, \ldots, 8\}$. Therefore, exactly n of x's neighbors are mines. On the first click, simply knowing this constraint is insufficient

information to deduce any mines or safe squares. The player is either forced to guess or, in the case where x is not the first move, reason about another region of the board.

Suppose x is 0. In this scenario, the player knows for a fact that none of x's neighbors are mines. Therefore, the player can safely probe the neighbors of x revealing additional information about the board.

The desired outcome is to probe a cell that contains the number 0. The center or internal cells seem like the ideal candidates since these cells would expose the most board data per click. However, interior cells are the least likely to actually contain a 0. To see this, consider the conditions for a square to have a zero.

As mentioned above, the label of a position x_0 is 0 if and only if none of its k neighbors contain mines. As a probability, this statement becomes the following expression:

$$P(x_0 = 0) = P(x_0 \neq \text{mine}, \{x_i\}_{i=1}^{k} \neq \text{mine}), \tag{4.1}$$

where x_i is one of x's k neighbors.

Using the complement of an event, $P(x_i \neq \text{mine}) = 1 - P(x_i = \text{mine})$. Furthermore, $P(x_i = \text{mine}) = d$, where d is the mine density (i.e. the number of mines divided by the total number of covered squares), assuming a uniform distribution of mines on the board.

Equation 4.1 depends on the number of neighbors of x_0. Consider a concrete example where x_0 is a corner square and the number of neighbors k equals 3. Under this example, equation 4.1 becomes

$$P(x_0 = 0) = P(x_0 \neq \text{mine}, x_1 \neq \text{mine}, x_2 \neq \text{mine}, x_3 \neq \text{mine})$$

It is tempting to conclude that the above expression reduces to $P(x_0 = 0) = (1-d)^{3+1}$ where d is the mine density. However, this assumes that the probability of x_i being mine free is independent of x_{i-1}. This is not the case. Recall that $P(x_i = \text{mine})$ uses the total number of unknown squares. Therefore, once the player knows a particular position does not contain a mine, the total possible cells that could have the remaining mines decreases. The correct simplification is the following expression:

$$\begin{aligned} P(\bigcap_{i=0}^{3} x_i \neq \text{mine}) &= P(x_0 \neq \text{mine})P(x_1 \neq \text{mine}|x_0 \neq \text{mine})P(x_2 \neq \text{mine}|x_1 = x_0 \neq \text{mine}) \\ &\quad P(x_3 \neq \text{mine}|x_2 = x_1 = x_0 \neq \text{mine}) \\ &= \prod_{i=0}^{3}(1 - \frac{m}{s-i}), \end{aligned}$$

where m equals the number of mines remaining and s is the total number of covered squares left. More generally, the probability for a square x_0 to be zero given it has k neighbors is

$$P(x_0 = 0) = P(\bigcap_{i=0}^{k} x_i \neq \text{mine}) = \prod_{i=0}^{k}(1 - \frac{m}{s-i}) \tag{4.2}$$

Typically the total number of covered squares in a board is much greater than the number of neighbors a cell may have. So assuming $s \gg i$, equation 4.2 can be approximated by the following formula:

$$\begin{aligned} P(\bigcap_{i=0}^{k} x_i \neq \text{mine}) &\approx \prod_{i=0}^{k}(1 - \frac{m}{s}) \\ &\approx (1-d)^{k+1} \end{aligned} \tag{4.3}$$

Thus, assuming independent probabilities for $P(x_i = \text{mine})$ is not a terrible approximation.

It follows that k must be minimal to maximize the probability in equation 4.2. As a result, corners squares are the most likely to be zero since they have the lowest k value ($k = 3$).

Kasper Pederson offers a combinatorial explanation for the probability of a square being zero [4]. Define C_k as the number of ways of arranging m mines on a board with n total squares ensuring k specific squares are mine free. Mathematically, C_k is:

$$C_k = \binom{n-k}{m}$$

The probability of having a subset of cells be mine-free is exactly C_k divided by the total number of ways of distributing m mines on a board with n squares.

$$P(x_0 = 0) = P(\bigcap_{i=0}^{k} x_i \neq \text{mine}) = \frac{C_k}{\binom{n}{m}} \tag{4.4}$$

Once again, the above probability is maximal when k is the smallest.

Regardless of the method employed, corner squares have the highest probability of being a zero. Therefore, the corners are the safest location to start a game of Minesweeper. Nevertheless, any algorithm must consider the tradeoff between the likelihood of obtaining a zero and the amount of information revealed when considering the ideal first move.

4.2 Handling Guesses

Being able to make deterministic actions in Minesweeper is oftentimes insufficient for completing games. Eventually the player will be presented with too little information to make a concrete deduction, and so he/she will be forced to guess. Without guessing, any Minesweeper solver will face problems similar to Adamatzky's cellular automaton model which was unable to finish certain configurations. In fact, section 4.1 was a special case of guessing that occurs at the outset of all Minesweeper games.

Despite the mathematical formalism regarding how to handle the opener to a game, guessing in general is much more difficult. Similar to the first click, algorithms must consider whether to select a corner, edge, or internal cell. However, solvers need to additionally reason about the constraints probed cells place on unknown cells, the number of mines still in play, and the locations that will be the most advantageous to the player. There are several different heuristics that solvers utilize; we will only mention a few here.

The most naive approach for handling guessing is random selection. Random selection nondeterministically chooses an arbitrary covered cell and probes it. The benefit to this approach is that it is simple to implement. In addition, it is easy to convince oneself that this method will never get stuck.

On the other hand, random selection's main drawback is in its simplicity. Random selection is not informed by any knowledge that the player may have about the board state. Furthermore, section 4.1 illustrated how certain squares are better sites for guessing than others. As a result, random selection does not perform well especially at higher difficulties where guessing becomes increasingly more common.

Another heuristic involves enumerating all the consistent configurations associated with each possible assignment to a particular cell. Doing so allows the solver to calculate a probability estimate that the cell under consideration contains a mine. Chris Studholme takes this approach in his CSP model [6]. The main problem with this approach is that it relies on enumerating a potentially large search space. In addition, Chris Studholme also suggests that this method assumes particular configurations are equally likely, which may not be the case. In other words, this method may make assumptions about the underlying probability of each configuration which results in a loss in performance. Nonetheless, finding a probability estimate for mines in the covered cells yields better results than random selection as it takes advantage of the current state space.

Occasionally, there are instances where the player is forced between a 50-50 guess. To make matters worse, these "crap-shoot" scenarios, as Chris Studholme calls them [6], are typically unavoidable and no additional board data can resolve them. Some solvers attempt to identify these game instances and deal with them as soon as possible. The idea is to guess early so that the solver can move onto a new game quickly if need be.

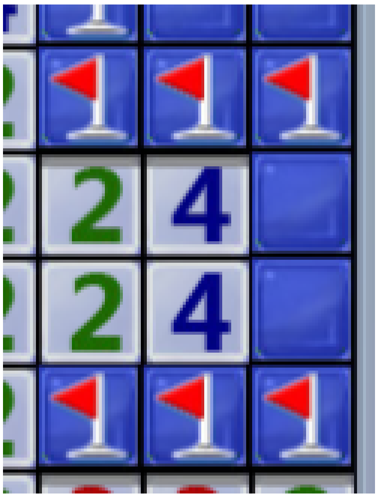

FIGURE 4.1: An example of a crap-shoot scenario. Each covered cell has an equal chance of being a mine. Since these squares have no more covered neighbors, the player will have to guess here.

4.3 Metrics for Evaluation

The main metric for evaluating an algorithm will be the percentage of games won, or win ratio. Each algorithm will play 1000 sets of 100 games on various board sizes

and difficulties. Having multiple sets allows for the computation of the average win percentage as well as the standard deviation. Win ratio is a straightforward measurement for determining how *good* a approach is. In particular, algorithms that can win games are more desirable since the objective of a solver is to successfully play Minesweeper.

Algorithms will be tested on three difficulty levels: beginner, intermediate, and expert. Beginner will be a 9×9 board with 10 mines. Intermediate will be a 16×16 board with 40 mines. Expert will be a 16×30 board with 99 mines. These specifications were chosen for beginner and intermediate because they reflect the Windows operating system's current distribution of Minesweeper which is one of the most popular implementations.

In addition, several tests will run on boards with a varying mine density. In other words, the number of mines will change while the board size remains constant. In particular, the board will have a constant 81 squares (9×9 configuration). Another set of tests will evaluate the opposite experiment: keeping the mine density constant (or as close to constant as possible) while altering the board size. The mine density for this group of tests will be approximately 0.16, the density of an intermediate configuration.

Tests will be utilizing *soft* rules for first clicks. Namely, the board generation will ensure that the solver cannot lose on the first click. Losing on the initial move does not reflect the efficiency of a solver since the first probing is always a guess. Therefore, the possibility of revealing a mine on the opener is removed from the tests.

The expected number of guesses for each algorithm is also noteworthy. For each game, the total number of guesses will be aggregated allowing the mean to be calculated across all sets. Ideally, the best solver will have a high win percentage and a low guess average. However, mean guesses alone is not indicative of the performance of a solver. For example, one solver may simply probe squares left to right, top to bottom. Such a solver would have a guess average of zero, yet that same solver would likely have low win rate.

Chapter 5

Single Point Strategies

The simplest category of Minesweeper algorithms is the single point (SP) strategy. As the name suggests, SP models consider at any given instance the constraints one particular cell applies onto its immediate neighbors. The exact implementation of this approach may vary, as we will see in the following sections, but this general concept of single square computations is the defining feature of a single point solver.

The SP design captures the two types of outcomes that can occur when looking at a single square: locating neighboring free cells or marking adjacent mines. The first type of deduction occurs when the effective label of a probed square equals zero (i.e. the label equals the number of marked neighbors). In this instance, the player can conclude that all adjacent unmarked cells are free cells. The second form of deduction identifies mines. When the effective label equals the number of unmarked neighbors, all unmarked neighbors can be marked as mines because the missing mines cannot be in any other squares. Due to the prevalence of these two inference techniques in various algorithms, I will name them **all free neighbors (AFN)** and **all mine/mark neighbors (AMN)** moves, respectively.

5.1 Cellular Automaton

A cellular automaton defines a set of cell states Q and a transition function f which takes a state at time t to the state at time $t + 1$. In Andrew Adamatzky's model [3], each state is represented as a tuple. More specifically, the set of states is defined to be the following Cartesian product:

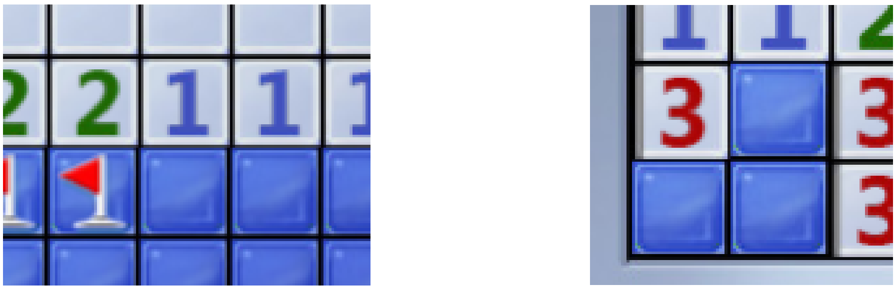

FIGURE 5.1: (Left) An instance of all free neighbors. The two covered squares below the center **1** must be mine-free since the mine has already been marked. (Right) An instance of all mine neighbors. The effective label is three and there are only three covered cells. Therefore, all three of the covered cells are mines.

$$Q = \{\#, \bullet, \circ\} \times \{0, 1, \ldots, 8\},$$

where for $x \in Q$, $x^t = \#$ means x contains a mine at time t, $x^t = \bullet$ means x is a covered square at time t, and $x^t = \circ$ means x is an uncovered cell at time t. The label of cell x is the second value in the tuple. For simplicity, I will adopt Adamtzky's notation and define $x^t \in \{\#, \bullet, \circ\}$ and $v(x) = \mathsf{Label}(x) \in \{0, 1, \ldots, 8\}$. Therefore a cell state is the tuple $(x^t, v(x))$. To reflect the rules of Minesweeper, the agent can only know $v(x)$ at time t if $x^t = \circ$.

The transition function f is what makes the cellular automaton approach a single point approach. f encapsulates both AFN and AMN. Here is the transition function defined by Adamatzky:

$$x^{t+1} = f(x^t) \begin{cases} \circ & (x^t = \bullet) \wedge \Big(\exists y \in \mathsf{Neighbors}(x) : y^t = \circ \wedge (v(y) = \mathsf{NumMarkedNeighbors}(y^t))\Big) \\ \# & (x^t = \bullet) \wedge \Big(\exists y \in \mathsf{Neighbors}(x) : y^t = \circ \wedge |\mathsf{UncoveredNeighbors}(y^t)| = 1 \\ & \qquad \wedge v(y) - \mathsf{NumMarkedNeighbors}(y^t) = 1\Big) \\ \bullet & \text{otherwise} \end{cases}$$

Adamatzky's AMN implementation (i.e. the second case in the transition function) only accounts for the case where the effective label of the observed square is one. However, it is easy to expand the AMN case by changing the 1 to an integer n.

The cellular automaton operates by imagining a configuration as a state in time. Performing an action advances time to a new state. The solver proceeds by advancing

each cell state using the transition function. The solver ends when it reaches a stationary configuration (i.e. when the configuration at time t equals the configuration at time $t + 1$). As previously mentioned, the termination of the CA does not imply the game was successfully completed. One of the main limitations of the CA approach is that it does not account for nondeterminism. As explained in section 4.2, handling guesses is crucial for a Minesweeper solver. In fact, Adamatzky's model can not even start a game; it has to be given a partially uncovered configuration [3].

Despite its tendency to become stuck, the CA approach is useful in identifying the strengths and weaknesses of the single point model. SP algorithms tend to be simple to understand and implement. However, SP solvers alone cannot complete most games; stochastic decision making is necessary.

5.2 Naive Single Point

Kasper Pederson and the author of PGMS describe alternative SP algorithms that make use of randomness [4]. This section will first focus on the general SP model these two follow as both approaches have a similar foundation. Then, the specific differences of the two models will be discussed.

The general approach of the strategy is to maintain a set S which is a collection of safe cells the algorithm will probe. If S is empty, the algorithm selects a covered square at random to insert into S. As a result, the first move will be a randomly chosen square from the entire board.

For each position x in S, the solver will determine if x falls into AFN or AMN. This is done by looking at the effective label of x or $\mathsf{elabel}(x)$. If $\mathsf{elabel}(x)$ is zero or is equal to the number of unknown neighbor cells, then x is an instance of AFN or AMN, respectively. The solver will then proceed to probe or mark all adjacent unknown squares depending on which case was found.

In the case where $\mathsf{elabel}(x)$ is neither of the two above options, the solver typically abandons x. However, discarding squares introduces an ordering problem. More specifically, the sequence in which cells are considered becomes crucial. There is no guarantee that a removed square will not yield a favorable deduction later on when the configuration has more available information. In other words, a square may not be a case of

AMN or AFM when considered the first time; however, if considered again with more board information, then it may.

PGMS resolves the ordering problem by making a simple observation. The SP solver must reconsider square x only when the states of its neighbors change. The adjacent cells could change anytime a square in the second order neighborhood around x is removed from S. Therefore, after performing the AMN or AFM deductions, the solver inserts all the second order neighbors of x into S.

Algorithm 5.1 Naive Single Point

```
S ← {}
while game is not over do
  if S is empty then
    x ← SELECT-RANDOM-SQUARE()
    S ← {x}
  end if
  for x ∈ S do
    probe(x)
    if x = mine then
      return failure
    end if
    U_x ← UNMARKED-NEIGHBORS(x)
    if isAFN(x) = True then
      for y ∈ U_x do
        S ← S ∪ {y}
      end for
    else if isAMN(x) = True then
      for y ∈ U_x do
        mark(y)
      end for
    else
      Ignore x
    end if
  end for
end while
```

Kasper's solution for the ordering problem is slightly different. When a square x is removed from S, instead of categorizing x into AMN and AFM, the solver reasons about the neighbors of x. If one of x's neighbors y is found to have a AMN or AFM deduction, then the neighbors of y are added to S which essentially adds the second order neighbors of x like PGMS.

PGMS and Kasper Pederson both define the set S slightly differently. In Pederson's implementation, S is a priority queue of viable moves, where a move is a tuple containing an uncovered position and an associated action to perform at that cell (i.e. mark or

probe). Squares that identify mines are at the front of the queue. The queue ordering reflects the idea that knowing something with a low probability is more valuable than knowing something with a high probability; a square is much more likely to be free than contain a mine. In contrast, PGMS implements S as an unordered set containing probed cells.

5.3 Double Set Single Point

The double set single point (DSSP or double set SP) algorithm maintains two sets S and Q. As before, S contains safe squares the solver will probe. The additional set Q is how this implementation resolves the ordering problem mentioned in the previous section. Rather than abandoning probed squares which do not fall into AFN or AMN, double set SP classifies these points as questionable and inserts them into Q. The idea is that the solver will return to these points once it has enough information to make use of them.

Q can also be thought of as the frontier of the current board state; Q contains all the cells which form the boundary between covered squares and uncovered squares. In other words, an element $q \in Q$ is a position that still has unknown neighbors. The cell q is an unresolved square, and therefore the solver must consider q in the future.

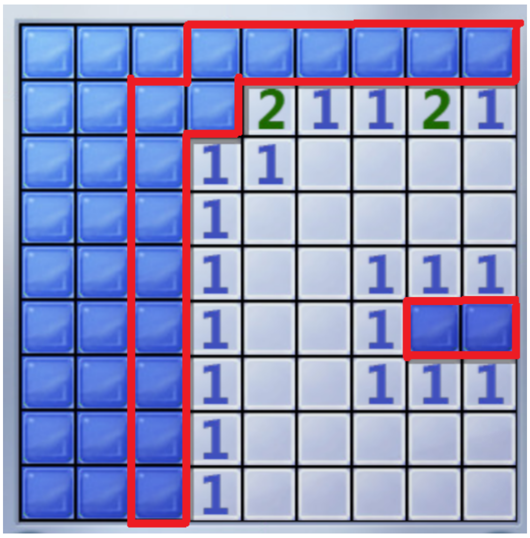

FIGURE 5.2: The red outlined covered squares are the elements in Q. These squares form the boundary between the uncovered squares and the covered ones.

Algorithm 5.2 Double Set Single Point

```
opener = First-Move()
S ← {opener}
Q ← {}
while game is not over do
  if S is empty then
    x ← Select-Random-Square()
    S ← {x}
  end if
  while S is not empty do
    x ← S.remove()
    probe(x)
    if x = mine then
      return failure
    end if
    if isAFN(x) = True then
      S ← S ∪ Unmarked-Neighbors(x)
    else
      Q ← Q ∪ {x}
    end if
  end while
  for q ∈ Q do
    if isAMN(q) = True then
      for y ∈ Unmarked-Neighbors(q) do
        mark(y)
      end for
      Q.remove(q)
    end if
  end for
  for q ∈ Q do
    if isAFN(q) = True then
      S ← S ∪ Unmarked-Neighbors(q)
      Q.remove(q)
    end if
  end for
end while
```

Double set single point alternates between probing and marking stages. The first phase is a probing step. The solver iterates and probes through each square x in S. Like previous implementations, if S is empty the algorithm randomly selects a covered square to insert into S. If x is an occurrence of AFN, the solver inserts the covered neighbors of x into S. Squares that are not examples of AFN are removed from S and inserted into Q. Once this phase is complete, S will be empty and Q will have free cells that were not AFN instances.

The second stage, a marking phase, searches for AMN cases in Q. If cell $q \in Q$ is

found to be AMN, then the neighbors are marked and q is removed from Q. At this point in the procedure, the neighbors of squares in Q may have different states. Therefore, the last stage in DSSP is another probing step. Unlike the first probing phase, the solver iterates through Q rather than S searching for instances of AFN. When an instance is found, the covered neighbors are added to S. At the end of the three phases, Q will have questionable squares on the frontier that cannot be resolved with AFN and AMN alone. The solver returns back to stage one to repeat the process.

The approach of this algorithm was modeled after how a player proceeds with solving a game. On the first click, if a zero is revealed, the board reveals all neighboring zeros until a mine frontier is reached. The first probing phase reflects this initial propagation of zeros. Next, the player looks for deductions, marking mines and then uses those flags to probe more squares. Similarly, the first stage is followed by a marking and probing phase.

5.3.1 Performance Analysis: DSSP versus Naive SP

PGMS's single point implementation was utilized to represent the naive single point method for all testing below since it was freely available on the internet. In fact, Kasper Pederson's implementation has extremely similar win percentages and average guesses to PGMS's SP [4]. In addition, DSSP's opener will be a randomly selected square for comparison with naive SP. Results for DSSP with a random corner as a first move are also shown.

Table 5.1 and Table 5.2 show the performance of double set SP and naive SP, respectively, on the three standard difficulty levels. Naive single point has a win percentage of 64.95%, 30.79%, and 0.51% for beginner, intermediate, and expert, respectively. In contrast, DSSP manages to win 68.19% of beginner games, 35.92% of intermediate matches, and 0.89% of expert boards

In general, the two algorithms have comparable win percentages and mean guesses per game. Nevertheless, double set SP consistently results in a slightly higher win percent. This discrepancy results from the differences in how the two approaches handle the ordering problem. Naive SP fails to perform AMN or AFN on all the uncovered squares. As a result it is unable to reveal additional board information resulting in more random moves as seen in naive SP's slightly higher average guesses per game.

Naive SP	Beginner	Intermediate	Expert
Avg. Wins (%)	64.95	30.79	0.51
Wins Std. Dev	4.72	4.47	0.71
Avg. Guesses	3.06	4.40	5.42
Guess Std. Dev	2.09	2.85	3.66

TABLE 5.1: Results for running naive SP on beginner, intermediate, and expert.

DSSP	Beginner	Intermediate	Expert
Avg. Wins (%)	68.19	35.92	0.89
Wins Std. Dev	4.56	4.90	0.94
Avg. Guesses	2.97	4.24	5.43
Guess Std. Dev	1.98	2.71	3.63

TABLE 5.2: Results for running DSSP on beginner, intermediate, and expert with a random square as an opener.

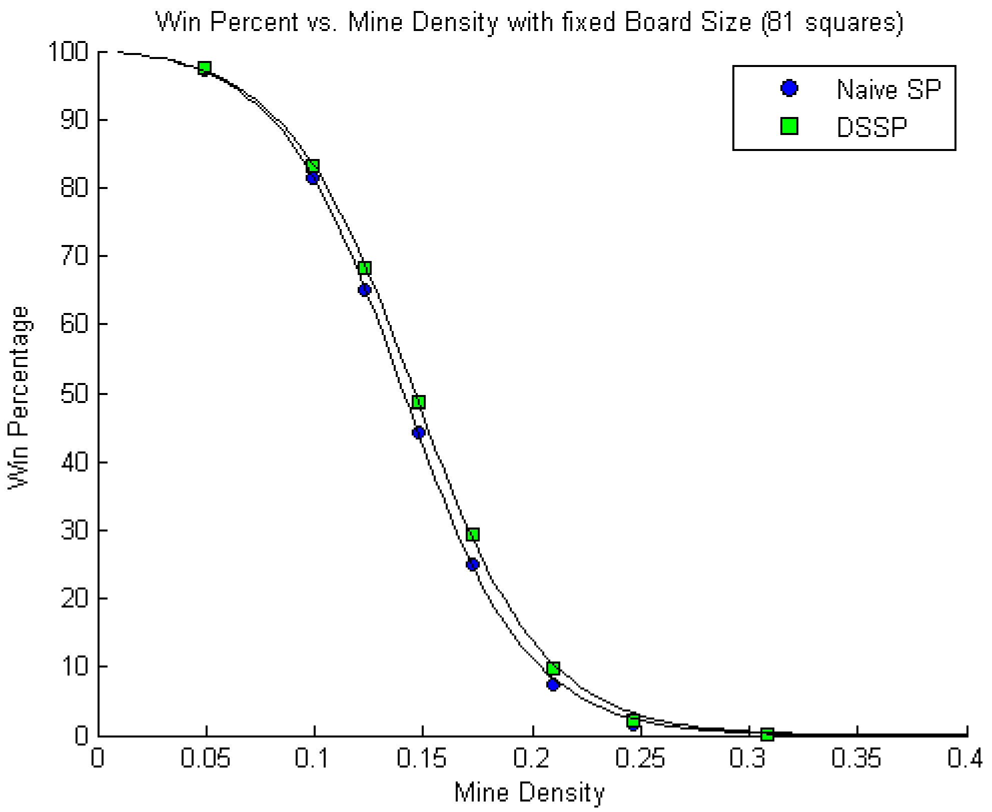

FIGURE 5.3: Win percent as a function of mine density for naive SP and double set SP. The board size is a constant 81 squares (9×9 board). Each set of data is fitted to a logistic function.

Overall, both solvers perform rather poorly. For expert, neither method is able to win even 1% of the total games. Moreover, on beginner, the two approaches are unable to win more than 70% of the games. One possible source for the low performance is the rudimentary random square selection each algorithm utilizes. As explained in section 4.2, random selection does not take advantage of the different probabilities each square has of being zero. Thus, for boards with higher mine densities which require more guessing, naive SP and DSSP are inefficient. Figure 5.3 illustrates how there is a sharp drop in win percent for mine densities between 0.1 and 0.2, relatively low densities. Furthermore, figure 5.4 suggest that board size also causes the decrease in win ratio for both single point methods.

As expected the number of guesses made on average increased with game difficulty. As stated before, harder game types tend to require more instances of nondeterministic decision making. Naive SP and DSSP both perform around 3 guesses on average per beginner game and approximately 5.4 guesses on average for expert boards.

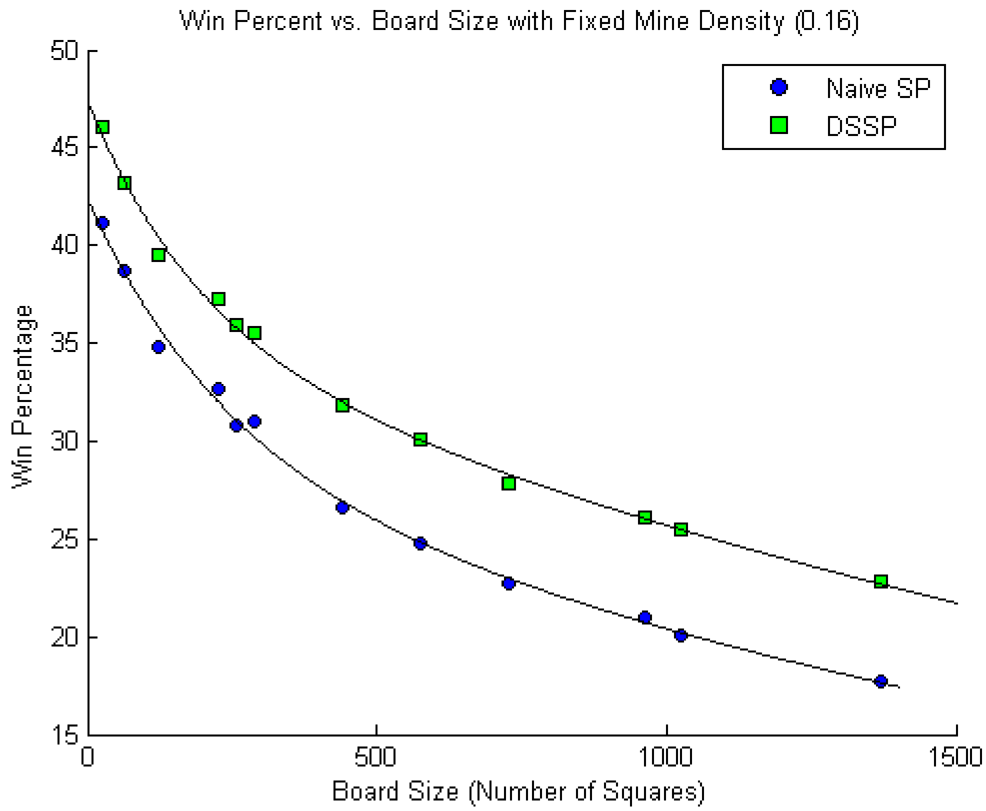

FIGURE 5.4: Win percent as a function of board size for naive SP and double set SP. The mine density is a constant 0.16. Each set of data is fitted to an exponential function.

DSSP - corner	Beginner	Intermediate	Expert
Avg. Wins (%)	71.07	37.91	0.92
Wins Std. Dev	4.39	4.93	0.95
Avg. Guesses	2.70	4.09	5.41
Guess Std. Dev	1.88	2.67	3.63

TABLE 5.3: Results for running DSSP on beginner, intermediate, and expert with a random corner as an opener.

Finally, Table 5.3 contains the data for double set SP when the first move is a random corner. The win percents for beginner, intermediate, and expert are 71%, 38%, and 0.92%, respectively, which is approximately 3% better than DSSP with a random opener.

5.3.2 Double Set Single Point with Improved Guessing

Double set single point struggles to solve any expert difficulty configurations due to its naive approach to guessing. In "Minesweeper as a Constraint Satisfaction Problem", Chris Studholme suggests improvements to the random move selection process that are independent of the CSP model [6]. The first modification leverages what section 4.1 already proved. That is to say, when the solver must stochastically select a square, DSSP will randomly select a corner square first since it has a higher probability of being a zero. If no covered corner squares are available, then DSSP will move to edge squares and finally internal squares. Again, this hierarchy of cell selection reflects the chances of revealing more board information as a function of the square location.

If an interior cell is indeed chosen, DSSP will choose a square that lies on the border of the frontier set Q. The idea here is that cells bordering the uncovered state space will more likely lead to further deterministic deductions as opposed to isolated internal cells. In particular, an isolated cell is useful to a solver only if it contains a zero. On the other hand, a border square may be useful so long as it does not contain a mine.

5.3.2.1 Performance Analysis

From Table 5.4 it is clear that the modifications to double set single point's guessing significantly boosted the performance of the solver. The effects are most notable on the easy and intermediate difficulties in which the win percent is 78.85% and 44.42%, respectively, an approximate 10% increase to the previous version of DSSP. On expert,

	Beginner	Intermediate	Expert
Avg. Wins (%)	78.85	44.42	1.32
Wins Std. Dev	3.97	4.92	1.12
Avg. Guesses	2.54	3.93	5.36
Guess Std. Dev	1.63	2.37	3.42

TABLE 5.4: Performance of double set single point solver with improved guessing.

however, there was only a 0.4% increase suggesting that the poor performance of single point methods is not solely due to inadequate probabilistic decisions.

On average, the improved DSSP guesses 2.54 times on beginner, 3.93 times on intermediate, and 5.36 times on expert, a 0.1-0.4 reduction from its previous implementation. This result implies that the educated random choices have a tendency to yield further deterministic actions reducing the need for additional guessing.

Nevertheless, the poor performance on higher difficulties and mine densities suggests that single point algorithms have a fundamental issue with their design outside of simply probabilistic choices, as mentioned earlier.

5.3.3 DSSP with Various Openers

This section will explore whether the theoretical explanation in section 4.1 is valid. In particular, the following tests will compare the performance of DSSP when the first move is an edge square and an internal square. Note that the experiments in this section use improved DSSP (double set single point with the probabilistic modifications from the previous section).

5.3.3.1 Performance Analysis

The results shown in tables 5.5 and 5.6 coincide with the findings in section 4.1. Namely, opening with an edge square results in worse performance than starting with a corner square. Starting with an internal cell does even worse than the other two.

On higher difficulties, this decrease in win ratio is less noticeable than in boards with lower mine densities. This stands to reason as SP solvers already perform poorly on the harder boards. In other words, although the first click is significant in that it impacts the execution of the algorithm, it does not completely dictate the outcome. Since single

	Beginner	Intermediate	Expert
Avg. Wins (%)	76.86	43.81	1.27
Wins Std. Dev	4.00	4.83	1.06
Avg. Guesses	2.64	2.99	4.36
Guess Std. Dev	1.68	2.39	3.45

Table 5.5: Performance of double set single point solver with random edge square as first move.

	Beginner	Intermediate	Expert
Avg. Wins (%)	75.28	42.07	1.28
Wins Std. Dev	4.43	4.83	1.16
Avg. Guesses	2.83	3.13	4.41
Guess Std. Dev	1.68	2.41	3.47

Table 5.6: Performance of double set single point solver with random interior square as opener.

point methods have a high probability of losing challenging Minesweeper games, opening with a better action does not offset the low win ratio.

5.4 Single Point Summary

Single point algorithms excel in their simplicity and ease of implementation. One does not need a deep understanding of Minesweeper and its mechanics to create a moderate SP solver. However, this straightforward design comes at the expense of the algorithm's overall performance. Single point methods do not generalize to harder Minesweeper boards.

The main flaw in the single point approach is that it is too myopic. The deterministic deductions of SP models focus on a small board region missing useful information that could lead to further deductions. In other words, considering a single square at a time is insufficient. Often times, the constraints of several nearby cells together can lead to the discovery of various actions. Figure 5.5 provides a concrete example of a configuration from which single point algorithms cannot discover any safe moves.

Even with improved stochastic decision making, single point solvers struggle with higher mine densities. This suggests that the narrow view of SP models is a more significant obstacle for their performance. In fact, the probabilistic elements added to single point algorithms are uninformed as they do not leverage what the player currently

FIGURE 5.5: In this configuration, the covered square directly below the center two is mine-free. This is due to the constraints from the surrounding ones in addition to the middle two. A single point solver would be unable make this conclusion since it only considers the constraints from an individual cell.

knows about the configuration. Therefore, other approaches need to reconcile the limited scope of single point techniques.

Chapter 6

Constraint Satisfaction Problem

A constraint satisfaction problem, or CSP, is a mathematical representation of a problem consisting of states whose solutions must satisfy a set of constraints. CSP models use a factored representation which allows the model to reduce unsatisfactory sections of the search space. Constraint satisfaction problems are common in artificial intelligence and can represent a wide array of problems.

There are three components to a constraint satisfaction problem: a set of variables, a set of domains for each variable, and a set of constraints. Constraints are a list of variables and a relation that defines or limits the available values the variables can take on. The objective is to find a value assignment for every variable such that none of the constraints are violated. Such an assignment is said to be complete and consistent [9].

6.1 Minesweeper as CSP

Minesweeper fits naturally into the CSP framework. Each square will be represented by a variable with a binary domain of 0 and 1. A value of 0 indicates the square is mine-free while a 1 denotes a cell containing a mine. Every time a square x is probed, x constrains the number of mines present in its neighbors. More specifically, if a probed square has label n, the constraint is that the sum of the neighboring variables must equal n. Therefore, a constraint will be a linear equation of variables that sum up to an integer n where $n \in \{0, \ldots, 8\}$. The integer n will be denoted as the label of a constraint.

With the model in place, one may note several insightful observations. First of all, instances of all-free-neighbors (AFN) and all-mine-neighbors (AMN) are easy to detect.

When the label of a constraint equals 0, then all the variables in that constraint must be mine-free; a label of 0 corresponds to AFN. On the other hand, if the label is equal to the number of variables in the constraint, then the constraint is an occurrence of AMN. Constraints that follow these two cases are degenerative or trivial constraints. Equations that involve a single variable are also forms of trivial constraints. These equations automatically define the value the variable must take on, and thus can be eliminated from the model immediately.

Another important observation involves reducing subsets of constraints. Consider the configuration in Figure 6.1. The middle square to the left of B implies that A or B or C is a mine. In the CSP model this would translate to the equation: A+B+C=1. The 2 right below the flag creates the constraint A+B=1 implying that either square A or B is a mine. Note that the second constraint is a subset of the first. Therefore, the first constraint can be simplified into C = 0 while the second constraint can be left the same: A+B=1. This reduction technique will be utilized heavily in the algorithms in the following sections.

FIGURE 6.1: There are three constraints in this configuration: A+B = 1, A+B+C = 1, and B+C =1. The two variable constraints are both subsets of the three variable constraint. Performing a constraint reduction yields C=0 or A=0 replacing A+B+C=1. Recall that squares with a label of zero are grey in this Minesweeper implementation.

6.2 Coupled Subsets CSP (CSCSP)

The CSP algorithm formulated by Chris Studholme [6] dynamically maintains a set of constraints C. Each time the player probes a square, the solver generates the corresponding constraint. Any nontrivial constraints are inserted into C while degenerative constraints are thrown away after completing the corresponding AFN or AMN actions. Furthermore, the solver identifies any constraint subsets and carries out the reduction outlined in the previous section.

Once C consists solely of reduced, nontrivial constraints, the algorithm performs backtracking search to aggregate all possible solutions. The backtracking algorithm selects variables to assign values based on the minimum-remaining value heuristic (i.e. variables with the smallest value domain). Any ties are broken with the highest-degree heuristic (i.e. variables involved in the largest number of constraints). In addition, the solver utilizes chronological backtracking if an inconsistent assignment is found; the most recently assigned variable is reassigned and the search continues. Lastly, the solver does forward checking after each assignment. Note that the order in which the algorithm selects values is not important since the backtracking search is computing all possible solutions.

To reduce the search space size, C is partitioned into coupled subsets. Two constraints are coupled if they share a common variable. As a result, backtracking search can solve each subset separately as an independent subproblem.

After the algorithm has found all viable solutions, it looks for squares which are a mine or mine-free in all solutions. The solver can mark or probe these points, respectively. Some solutions may reveal constraints which require a "crap-shoot" guess, as mentioned in section 4.2. Crap-shoot guesses are simply instances where a guess is required, typically between two unknown squares. When the solver finds any crap-shoots, it handles them immediately so that it can proceed to a new game as quickly as possible.

The coupled subsets CSP solver cleverly handles guesses. If no safe move is discovered from the constraint solutions, the solver calculates a probability estimate for each square being mine-free based on the solutions found through backtracking search. Mathematically, the probability estimate is the number of solutions with square x assigned 0, divided by the total number of solutions found. The cell with the highest probability of being mine-free is considered the best guess square.

Next, the algorithm compares the best guess cell with unconstrained squares (covered squares not involved in any constraint). Given the number of mines a solution requires and the total number of mines still in play, the solver calculates the probability of an unconstrained square being mine-free. If this probability is lower than the best guess probability, then the solver probes the best guess square. Otherwise, the solver randomly probes an unconstrained corner, edge, or internal square in that order. For an internal square, the algorithm selects the square with the largest overlap of variables with the current constraints. Doing so will reveal information relevant to the current constraint set.

6.3 Connected Components CSP

The connected components constraint satisfaction problem solver (C3SP) provides an alternative approach to the coupled subsets search space reduction that CSCSP does. In particular, the new solver divides the search space into subsets of variables that share constraints. Ultimately, this generates an equivalent partitioning of the constraint space. However, framing the partition as a connectivity problem allows for the solver to utilize existing algorithms to find connected components.

Consider a graph whose nodes are the variables in the CSP model. Nodes x and y share an edge (x, y) if there exists a constraint containing both variables. A connected component of this graph is a set of variables where assignment could potentially influence the assigned values of the other variables in that component.

The C3SP algorithm performs backtracking search on each connected component treating them as separate constraint satisfaction problems with independent solutions. Similar to the coupled subsets of the previous solver, breaking the search space into connected components reduces the complexity and the number of computations backtracking search needs to perform.

Lastly, C3SP simplifies the guessing heuristics which coupled subsets CSP implements. In particular, the connected components solver performs the same stochastic square selection described in section 5.3.2. However, the frontier set Q is replaced by the covered variables involved in the constraint set.

	Beginner	Intermediate	Expert
Avg. Wins (%)	90.85	74.18	25.96
Wins Std. Dev	2.79	4.47	4.30
Avg. Guesses	1.70	2.35	4.14
Guess Std. Dev	0.93	1.49	7.69

Table 6.1: Connected components CSP run on beginner, intermediate, and expert.

	Beginner	Intermediate	Expert
Avg. Wins (%)	91.25	75.94	32.90
Wins Std. Dev	2.77	4.25	4.75
Avg. Guesses	1.70	2.39	4.73
Guess Std. Dev	0.95	1.49	2.88

Table 6.2: The results of running coupled subsets CSP on the three standard difficulty levels.

6.3.1 Performance Analysis: Coupled Subsets CSP versus CSP with Connected Components

Table 6.1 and 6.2 show the performance of the coupled subsets solver and the connected components algorithm on beginner, intermediate and expert. On beginner and intermediate, CSCSP has a win ratio of 91.25% of 75.94%, while C3SP has slightly lower values of 90.85% and 74.14%. Nevertheless, the difference in win percentages is less than 2%. Furthermore, both solvers guess 1.7 and 2.4 times on average for beginner and intermediate games, respectively. The choice of using connected components as opposed to coupled subsets clearly does not impact the overall performance of the algorithm. However, in expert, the differences in how the algorithms handle guessing are noticeable. Coupled subsets CSP won 32.90% of the expert trials, while C3SP won 25.96%, roughly 7% less games.

Chapter 7

Conclusion

This paper verified several algorithmic design decisions for Minesweeper solvers. It provided a new mathematical explanation based on conditional probabilities for why corner squares are the preferred opener. The derived formula gave the same results as Kaye Pederson's combinatorial equation in "The Complexity of Minesweeper and strategies for game playing." In addition, empirical results further demonstrated how randomly selecting a corner is the best opener.

The importance of handling guesses was discussed several times when investigating various algorithmic approaches. Certain Minesweeper games will always have instances of guessing. Therefore, it is crucial that solvers incorporate efficient methods of performing nondeterministic moves. Furthermore, guessing heuristics that leverage information known about the current configuration performed better than uninformed approaches.

The double set single point algorithm was shown to perform better than the naive single point solvers. Modifications to DSSP were also made, but ultimately its performance was limited by the narrow single point design. Nonetheless, the constraint satisfaction problem model for Minesweeper was quite successful. The coupled subsets CSP approach had the highest win ratio of all algorithms. The connected components CSP model performed slightly worse, especially at higher difficulties, a result of it using simpler probabilistic heuristics.

For future research, I would like to explore different search methods for the CSP model like a SAT solver or perhaps a backtracking algorithm with conflict-directed backjumping. Alternatively, it would be worthwhile to consider algorithms which are designed around uncertainty such as Markov decision processes.

Bibliography

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

[2] Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be NP-complete but is hard nonetheless. *The Mathematical Intelligencer*, 33(4):5–17, 2011.

[3] Andrew Adamatzky. How cellular automaton plays minesweeper. *Applied Mathematics and Computation*, 85(2–3):127–137, 1997.

[4] Kasper Pedersen. The complexity of minesweeper and strategies for game playing. *Project report, univ. Warwick*, 2004.

[5] Lourdes Peña Castillo and Stefan Wrobel. Learning minesweeper with multirelational learning. In *IJCAI*, pages 533–538, August 2003.

[6] Chris Studholme. Minesweeper as a constraint satisfaction problem. *Unpublished project report*, 2000.

[7] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computer and games*, pages 72–83. Springer, 2007.

[8] Michele Sebag and Olivier Teytaud. Combining myopic optimization and tree search: Application to minesweeper. In *LION6, Learning and Intelligent Optimization*, volume 7219, pages 222–236. Sringer Verlag, 2012.

[9] Stuart J. Russell and Peter Norvig. Artificial intelligence: a modern approach. *Prentice Hall series in artificial intelligence*, 2010.