# Explorations of the Minesweeper Consistency Problem

Meredith Kadlac
Humboldt State University, Arcata, CA
Advisor: Paul Cull
Oregon State University, Corvallis, OR

15 August 2003

**Abstract**

Minesweeper is a well-known computer game commonly included with Windows operating systems. Minesweeper has recently been shown to be hard in the sense that determining if there is a layout of mines and open spaces which is consistent with a grid pattern specifying mines, number of mines adjacent to a grid position, and unknowns is an NP-complete problem. We give a C++ program which uses breadth-first search to find all the layouts consistent with a particular input pattern. We give some examples of the program's input and output. We timed the program and found, as expected, that its run time is exponential in the input size for some patterns. We restricted minesweeper to one dimension to see if the problem was easier. We found that the one-dimensional minesweeper consistency problem is regular and we give a finite state machine which recognizes patterns which are consistent with some unknown layout.

## 1 Introduction

One of the most notorious open problems in mathematics is the P=NP? question. One of the more recent and engaging inquiries into this issue was made by Richard Kaye, in which he showed that the minesweeper consistency problem is NP-complete. In this paper, I present aspects of the minesweeper consistency problem into which I have investigated while attending the research experience for undergraduates at Oregon State University. My investigations into this problem have resulted in two discoveries: One, I have written a program which finds all possible solutions to a given minesweeper puzzle. Two, I have defined minesweeper consistency restricted to one dimension and shown that it is easy by exhibiting a deterministic finite state machine that solves it.

To allow me to discuss these results, I will first give some basic definitions pertaining to complexity theory and automata theory.

## 2 Computational Complexity Theory

Complexity theory seeks to analyze what resources are required during a computation to solve a given problem. Two important measures of complexity are time and space complexity. The time complexity of a problem is the number of steps required to solve it, while the space complexity of a problem is the amount of memory needed to solve it. Here I will restrict the discussion of complexity to time complexity.

**Definition 2.1** *A polynomial time algorithm is one for which the time complexity is polynomial in the size of the input. For example, algorithms with time complexity $\Theta(n)$ or $\Theta(99n^5)$ are polynomial time algorithms.*

**Definition 2.2** *An exponential time algorithm is one for which the time complexity is exponential in the size of the input. An example of an exponential time complexity is $\Theta(2^n)$.*

A polynomial time algorithm is considered to be efficient unless the input is extremely large. An exponential time algorithm can work for relatively small inputs, but as the input size increases it can become very inefficient. For instance, consider an algorithm with time complexity $\Theta(3^n)$. An input size of n=60 would require 1.3 million centuries to solve! [HU79] Problems for which only exponential time algorithms are known are often called intractable.

**Definition 2.3** *The complexity class P is the collection of problems which can be solved deterministically in polynomial time. A deterministic solution has exactly one choice at each step.*

**Definition 2.4** *The complexity class NP is the collection of problems which can be solved nondeterministically in polynomial time. A nondeterministic solution may have some steps with more than one choice.*

It is not known for sure whether P=NP although it is generally thought that they are not equal. Problems in NP that are considered the most difficult, and the most likely not to be in P, are called NP-complete. Examples of problems which have been proven to be NP-complete are satisfiability and the coloring problem. To show that a problem B is NP-complete, one must demonstrate the following:

1. $B \in NP$
2. For every other problem $A \in NP$, $A \leq B$.

Here, the less than or equal to symbol can be read ”is no harder than”, and we say that B reduces to A. To show that $A \leq B$, we show that there exists a polynomial time function f such that, for every input instance $I \in A$, the solution in problem A to instance I is the same as the solution in problem B to f(I). For a more in-depth discussion on NP-completeness, a good source is Garey and Johnson’s *Computers and Intractability* [GJ79].

## 3 Languages and Finite State Machines

I will now give some brief definitions concerning languages and finite state machines. For more details, see a text on automata theory such as Hopcroft and Ullman's *Introduction to Automata Theory, Languages, and Computation* [HU79].

**Definition 3.1** *An alphabet, denoted $\Sigma$, is a finite nonempty set of symbols.*

**Example 3.2** $\Sigma=\{0,1\}$ *is the binary alphabet.*

**Definition 3.3** *A string, or word, of length n over an alphabet $\Sigma$ is a sequence $x_1x_2...x_n$, for $n\in N$, such that $x_i \in \Sigma$ for every i, $1\leq i\leq n$.*

**Example 3.4** *0110101010111101011 is a string over the binary alphabet.*

These definitions are a sufficient basis for the definition of a finite state machine.

**Definition 3.5** *A finite state machine (FSM) is a quintuple $<\Sigma, S, s_o, \delta, F>$, where $\Sigma$ is the input alphabet, S is a finite nonempty set of states, s is the initial state, $\delta$ is the state transition function ( $\delta$:(S x $\Sigma$)$\rightarrow$S ), and $F\subseteq S$ is the set of accepting states.*

## 4 Minesweeper

Minesweeper is a popular game which is included with all Windows operating systems. I will briefly describe how the game is played.

The player is given a two-dimensional grid of blank squares, under each of which there may or may not be a bomb. Upon clicking his/her mouse on any square, s/he finds that either the square has a bomb or it provides a count of how many bombs reside in any of the eight squares adjacent to it. If there are no bombs near a particular square, then that square remains blank or in some versions of the game it will have a zero. Otherwise, the square will contain a number between one and eight which tells the player that there are anywhere from one to eight bombs in all adjacent squares. The object of the game is to determine which squares have bombs by clearing, by way of clicking on, every square that does not have a bomb. To accomplish this, the player takes the information given in the squares found not to contain bombs and uses it to deduce the locations of the bombs. For example, consider the configuration depicted below to the left.

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 2 | ? |
| 0 | 0 | 0 | 2 | ? |
| 1 | 1 | 0 | 1 | 1 |
| ? | 1 | 0 | 0 | 0 |

=>

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 2 | B |
| 0 | 0 | 0 | 2 | B |
| 1 | 1 | 0 | 1 | 1 |
| B | 1 | 0 | 0 | 0 |

It is easy to deduce that all of the squares with question marks contain bombs. Of course, the game is not always this easy. Often it is the case that the player cannot decide if a square in question has a mine without guessing. For example, the following configuration could result in any one of four different solutions, as depicted below.

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | B | 2 | 1 |
| 1 | 2 | ? | ? |
| 0 | 1 | ? | ? |

=>

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | B | 2 | 1 |
| 1 | 2 | B | 1/2 |
| 0 | 1 | 1/2 | 1/B |

or

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | B | 2 | 1 |
| 1 | 2 | 3/4 | B |
| 0 | 1 | B | 2/B |

???

If faced with such a decision, the player can click the top left ? or one of either the top right ? and the bottom left ?, and so long as s/he guesses correctly, s/he will have enough clues to continue to play. Otherwise, the game will end.

## 5 Minesweeper Consistency Problem

The minesweeper consistency problem (MCP) is posed as following: Given a minesweeper grid, is it consistent? A minesweeper puzzle is consistent if there exists at least one correspondence between the information in each square and an element of the set {–, B}, where – means no bomb and B means bomb, that solves the puzzle correctly. For example, the following grid is not consistent for various reasons.

| | | |
|---|---|---|
| 0 | ? | 1 |
| 2 | ? | 5 |
| ? | ? | ? |

For one, it is impossible that five bombs actually surround the square with the 5 as one of the squares next to it is marked with a 1, leaving only 4 possible bomb spots.

In his article, "Minesweeper is NP-Complete" in the Spring 2000 issue of The Mathematical Intelligencer, Richard Kaye proved that MCP is NP-complete by reduction from circuit satisfiability. I will not go into a detailled explanation of his work in this paper as I could not possibly do any better at describing it than he has. However, I invite the interested reader to refer to his article in the Intelligencer and his website, listed in the bibliography.

## 6 Minesweeper Consistency Exhaustive Search

I have written a program in C++ which finds, via an exhaustive search, all possible consistent configurations for a given minesweeper grid. The idea to do this came from Neville Mehta's paper [ME03], in which he discusses a similar program he had written in LISP. Since I do not know LISP, I could not use the program. As such, I decided to code it in C++. In this section, I discuss the code for the program I wrote and explain how it works. The actual code can be found in Appendix A, and I will refer to it from here on as the MS consistency code.

The minesweeper grid under consideration is inputted as a two-dimension-
al character array of size ROWS by COLS, both of which are declared as global constants. The elements $m_{ij}$ ($0 \leq i \leq$ROWS, $0 \leq j \leq$COLS) of the array are taken from the alphabet $\{0,1,2,3,4,5,6,7,8,B,?\}$. If any $m_{ij}$ is a number x between 0 and 8, then $m_{ij}$ does not contain a bomb and x corresponds to how many bombs are in all surrounding elements. The B, of course, represents a bomb and the ? represents an unknown square, which could be a bomb or a safe square.

The MS consistency program outputs all possible consistent bomb configurations that can be arranged in the given grid. In this way, it can be used to decide the consistency of a grid in the minesweeper consistency problem posed by Richard Kaye. It does so by employing a breadth-first search, similar to that used in Mehta's LISP code. The search algorithm, to start, creates a set of new grids by placing exactly one bomb (B) in each possible location where there are unknown spots (?), and a single grid by replacing all ?s with safe squares. Every new grid is then classified according to consistency and goal-status. A grid is "consistent" if, for all squares within it numbered as n, for $0 \leq n \leq 8$, it is true that both n is less than or equal to the number of surrounding Bs and that n is greater than the number of surrounding Bs plus the number of surrounding ?s. A grid is "goal-state" if, for all squares within it numbered as n, for $0 \leq n \leq 8$, n is exactly equal to the number of surrounding Bs. Note that being goal-state implies that a grid is consistent, but the converse is not true. If a new grid is goal-state, it is printed out as one of the possible consistent configurations. At the end of the first iteration, the new grids that are both consistent and still have unknown spots in them are retained, while the inconsistent grids and those for which the search has been exhausted are pruned. Then, the algorithm begins again on the grids surviving the previous expansion.

This algorithm clearly in worse case has time complexity $O(2^n)$, where n=ROWS*COLS. However, the pruning that happens with each iteration of the search makes the algorithm tractable for most inputs. I have observed that as long as the grid contains at least one numbered square in every block of about nine squares, the possibilities for consistent grids becomes sufficiently limited. Once there is a block of nine squares that has no constraints imposed on it by a nearby numbered square, for example, then of course there are $2^9$=512 possible configurations for that block, and for blocks of more than nine ?s, matters soon become intractable. Please see the code in Appendix A along with outputs for sample boards. These boards were taken from Richard Kaye's articles and website concerning the minesweeper consistency problem [K00][Internet reference 1]. Neville Mehta also has run his LISP program using the same sample boards [ME03], achieving the same consistent configurations.

I would like to mention that I attempted to code this program using dynamic arrays, where the number of rows in the arrays would be determined with each iteration based on the number of new grids in the search frontier. However, I was not able to get such a program running so in the end I employed an array with a large, constant number of rows that I could adjust if I saw the need. I am sure that there are more space efficient ways of coding this search, but at this time anything more sophisticated is beyond my knowledge.

## 7 One-dimensional Minesweeper

I will now define the minesweeper consistency problem (MCP) in one dimension rather than two. As previously discussed, the two-dimensional MCP is NP-complete. I will show that the one-dimensional MCP is easy, unlike the two-dimensional case, by exhibiting a deterministic finite state machine (FSM) which determines the consistency of any one-dimensional minesweeper puzzle in polynomial time. I will prove that this machine solves all possible inputs by arguing that inconsistencies are local to four adjacent symbols (length four substrings), categorizing all inconsistencies into types and subtypes, and showing how the machine detects inconsistencies from each inconsistency subtype. Then I will show by induction that the one-dimensional MCP FSM given determines the consistency of any minesweeper string of length n, for n∈N. Finally, I will demonstrate how the machine gives outputs a consistent bomb configuration for each consistent string.

One-dimensional minesweeper is a simplification of regular minesweeper in that the puzzle to be solved is restricted to one row of blanks and numbers of adjacent bombs. As such, in any square the possibilities are limited to 0, 1, or 2 bombs adjacent (instead of up to 8 bombs adjacent), an unknown spot, ?, or a bomb, B.

The one-dimensional minesweeper consistency problem (MCP) is posed the same way as the regular MCP, but restricted to one-dimensional puzzles. That is, given a one-dimensional minesweeper puzzle, is it consistent? I give a definition of consistency for the one-dimensional puzzle below. I will claim that the restriction of MCP to one-dimensional MCP is easy; that is, one-dimensional MCP∈P.

**Definition 7.1** *A one-dimensional minesweeper puzzle is consistent if there exists at least one correspondence between the information in each square and an element of the set {–, B}, where – means no bomb and B means bomb, that solves the puzzle correctly. Note: A minesweeper puzzle is solved correctly if each square numbered m is surrounded by exactly m bombs.*

First, I would like to describe how the FSM I have created to solve this problem works. The problem lends itself to being decided by a finite state machine since a one-dimensional minesweeper puzzle nicely translates to a string of symbols over an alphabet.

**Definition 7.2** *A* ***minesweeper string of length n*** *is a word over the alphabet* $\sum^*=\{0, 1, 2, B, ?\}$ *representing a one-dimensional minesweeper puzzle of size n. A* ***minesweeper substring of length n*** *is a block of n adjacent squares within a given minesweeper string.*

**Definition 7.3** *A* ***bomb string of length n*** *is a word over the alphabet* $\sum_B^*=\{\text{-}, B, X\}$. *A* ***consistent bomb string of length n*** *is a word over* $\sum_B^* \backslash \{X\}$. *A bomb string is* ***inconsistent*** *if it contains the symbol X.*

**Definition 7.4** *A **local inconsistency** in a minesweeper string $M$=$X_1X_2...X_n$ is some subsequence of its symbols $X_i$ in which a pattern occurs that results in the impossibility of M being globally consistent.*

**Definition 7.5** *A minesweeper string $M$=$X_1X_2...X_n$ is **locally consistent up to position i**, for $1 \leq i \leq n$, if all substrings of the string $X_1X_2...X_{i-1}$are locally consistent. M is **consistent with respect to its ends** if the substrings $eX_1X_2$… and ...$X_{n-1}X_ne$ are locally consistent.*

**Definition 7.6** *A minesweeper string is **globally consistent** if all subsequences of the sequence $eX_1X_2...X_ne$ are locally consistent.*

The finite state machine (FSM) shown in Figure 1 takes as inputs minesweeper strings of length n and outputs bomb strings of length n.

The one-dimensional minesweeper finite state machine (MCP FSM) in Figure 1 is a septuple $<\sum, \sum_B, S,$ s$_o$, $\partial$, beta, F>. $\sum$={0, 1, 2, B, ?} is the input alphabet and $\sum_B$={-, B, X} is the output alphabet.
$S$={s$_o$,0,1$_o$,1$_b$,1$_?$,2,B,?$_o$,?$_b$,?$_?$,Er} is the set of states, s$_o$ is the initial state, and F = $S \setminus$ {Er} is the set of accepting states. Delta ($\partial$) is the state transition function mapping ($S$ x $\sum$) to $S$ and beta: S $\rightarrow \sum_B$ is an output function which maps the set of states to a bomb string. Table 1 gives the mappings of $\partial$ and Table 2 gives the mappings of beta.

| | **0** | **1** | **2** | **B** | **?** | **e** |
|---|---|---|---|---|---|---|
| **s**o | 0 | 1o | Er | B | ?? | Er |
| **0** | 0 | 1o | Er | Er | ?o | 0 |
| **1**0 | Er | Er | Er | B | ?b | Er |
| **1**b | 0 | 1o | Er | Er | ?o | 1b |
| **1**? | 0 | 1o | Er | B | ?? | 1? |
| **2** | Er | Er | Er | B | ?b | Er |
| **B** | Er | 1b | 2 | B | ?? | B |
| **?**0 | 0 | 1o | Er | B | ?? | ?o |
| **?**b | Er | 1b | 2 | B | ?? | ?b |
| **?**? | 0 | 1? | 2 | B | ?? | ?? |
| **Er** | Er | Er | Er | Er | Er | Er |

**Table 1** *State transition table for the one-dimensional MCP FSM. The columns of the table are elements of $\sum$ and the rows are elements of the set of all states.*

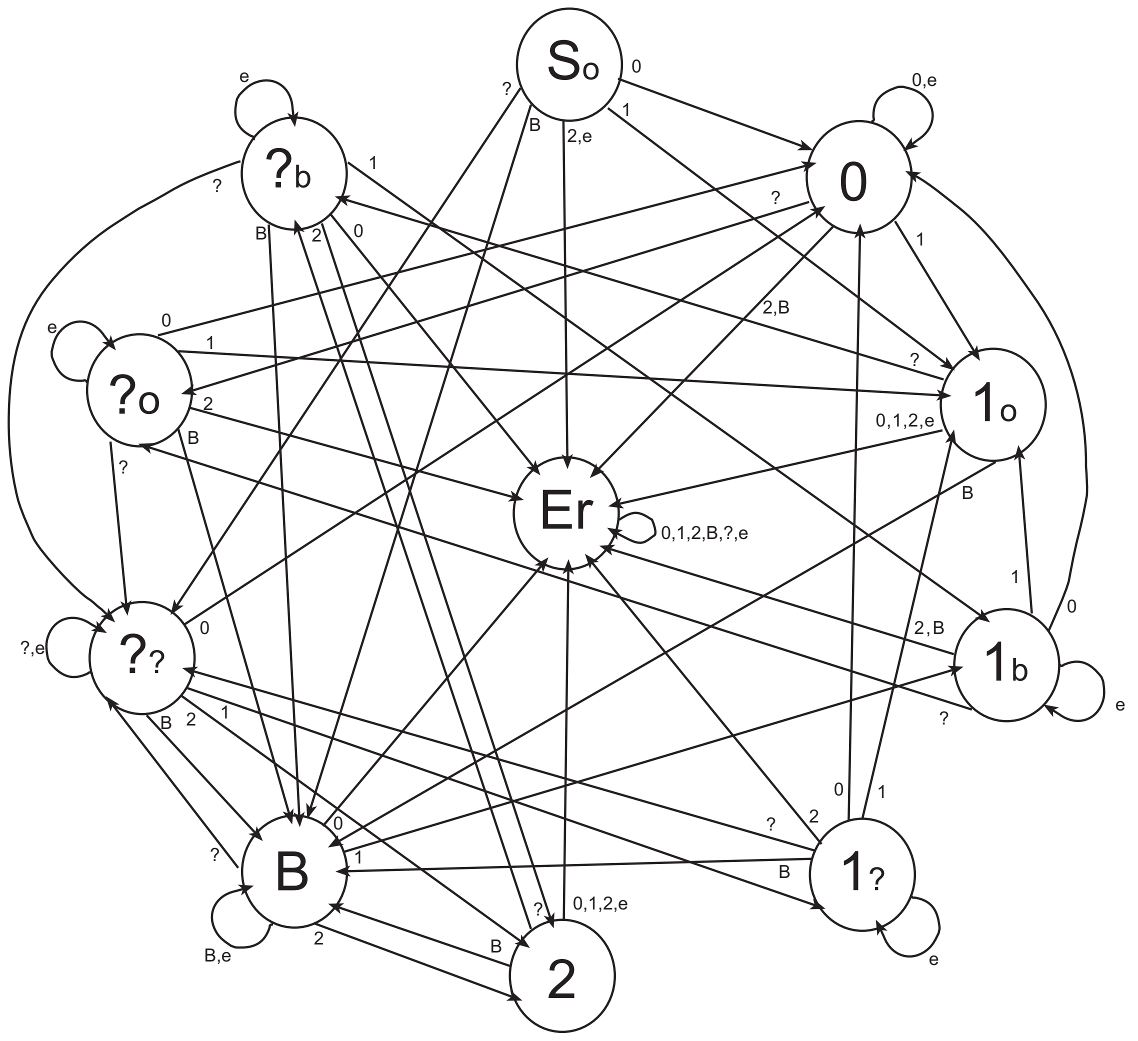

Figure 1: *Directed graph showing the action of delta, the state transition function.*

| S | $\sum_B$ |
|---|---|
| $s_o$ | X |
| 0 | 0 |
| $1_o$ | 0 |
| $1_b$ | 0 |
| $1_?$ | 0 |
| 2 | 0 |
| B | B |
| $?_o$ | 0 |
| $?_b$ | B |
| $?_?$ | B |
| Er | X |

**Table 2** *The mappings of beta: $S \rightarrow \sum_B$.*

The one-dimensional MCP FSM takes as inputs minesweeper strings of length n, arbitrarily reading them from left to right. It is set to start at the initial state, $s_o$. As $\partial$ maps from state to state, beta maps each state to the output alphabet $\sum_B$. In doing so, a bomb string of length n is created to correspond to the minesweeper string of length n being read. It passes over the string one time and stops once it detects the end of the string, the null character (e). Clearly, it has time complexity O(n). The one-dimensional MCP FSM was designed so that the only rejecting state is an error state, denoted Er. As soon as the machine detects an inconsistency, it maps to Er under delta, as I will prove. Then beta maps Er to X so that the bomb string outputted by the machine is inconsistent by definition if state Er has been mapped to under delta. After the machine has reached state Er for some input string, it continues to go to Er for every input from (S x $\sum$). I will show that any local inconsistency can be detected as a sequence of at most four symbols, or in other words all errors are local to substrings of length four.

Now I would like to state and prove the theorems which will lead to the conclusion that one-dimensional MCP is easy.

**Theorem 7.7** *The one-dimensional MCP FSM given detects all possible local inconsistencies in a minesweeper string of length n, including those peculiar to the beginning or end of the string. Furthermore, any local inconsistencies occur in substrings of length 4 or less.*

**Proof.** To show that the one-dimensional MCP FSM detects any possible local inconsistency in a minesweeper string of length n, I will argue that all possible local inconsistencies occur in substrings of at most four squares. I will arbitrarily define a classification scheme for inconsistencies with which to categorize all inconsistent length four substrings. Then, I will associate each inconsistency with its appropriate inconsistency type and subtype, and show how the machine detects each inconsistency subtype.

Let M=$X_1X_2...X_n$ be any minesweeper string. First, I will show how the one-dimensional MCP FSM detects endpoint inconsistencies. For instance, it would not be consistent to have a 2 as the first or last element of the minesweeper string. As such, if the MCP FSM is in the initial state and the next input is a 2, it goes to Er, the error state under delta (see Table 1). Also, if the MCP FSM is in state 2 and the next input is the null character e, meaning that there is a 2 at the right end of the string, then $\partial$ maps (2,e) to Er. Furthermore, it would clearly be inconsistent to have a sequence of two 1s at the beginning or end of a string. If at the initial state, the next input is a 1, then $\partial$ maps ($s_o$,1) to $1_o$. If the next input after that is a 1, then $\partial$ maps ($1_o$,1) to Er. Thus the machine detects the inconsistency of two 1s at the beginning of a string. On the other hand, if M is consistent up to the second to last symbol and the second to last symbol is a 1, then either it is in state $1_o$, $1_b$ or $1_?$ after reading this 1. If it is in $1_o$, then the input of another 1 gets mapped to Er anyway. Otherwise, if it is in $1_b$ or $1_?$ and reads a 1, $\partial$ maps to $1_o$. If it is in $1_o$ and reads an e, meaning that it has reached the right end of the string with after two 1s, then $\partial$ maps ($1_o$,e) to Er and the inconsistency is detected. The last arrangement that would be undesirable at the beginning or end of a string is a 1 followed by a 0 or a 0 followed by a 1, respectively. It is easy to verify how the machine detects these two errors using the state transition table, Table 1. This exhausts all possible endpoint inconsistencies and shows that all endpoint inconsistencies occur in a sequence of at most 3 symbols, including the null character.

Now suppose M=$X_1X_2...X_n$ is a minesweeper string found to be locally consistent up to position i and M is consistent with respect to its ends.

I argue that any local inconsistency following from $X_i$ will occur in at most four squares, including $X_i$. Note that inconsistencies arise from the nature of the puzzle and I will leave it to the reader to verify instances of them. Furthermore, in each case in the following argument, I will at times make the following claim about a minesweeper substring consistent up to some position j$\geq$i: further inconsistencies follow from $X_j$, starting over at length 1, and do not depend on any prior symbol in the string. I will also leave it to the reader's reflection to verify such claims as they are made.

I take cases on $X_i$. Where the word inconsistency is used, local inconsistency is implied.

Case 0

If $X_i$=0, then M will be locally consistent up to position i+1 only if the next symbol is a 0, 1 or ?. The sequences 02 and 0B are inconsistencies of length 2. So the consistent possibilities following from 0 are 00, 01 and 0?.

Subcase 00

This subcase is a repetition of Case 0.

Subcase 01

01 can be followed consistently by a B or a ?. The sequences 010, 011 and 012 are inconsistent. Note 010 and 011 are inconsistencies of length 3, while 012 is an inconsistency of length 2 since the inconsistency arises from the substring 12 and not the 0. So 01 can consistently be followed by a B or ? as 01B or 01?. The sequence 01BX will be consistent as long is X is consistent with respect to the B since the 01 before the B has no bearing on the consistency of M after the B. Thus

any further inconsistency following from the B will arise in Case B and the inconsistency length will be reset to 1. The sequence 01? can be followed consistently by anything but a 0. Note 01?0 is an inconsistency of length 4. Now the consistent possibilities are M=01?1, 01?2, 01?B, or 01??. At this point, it can be seen that, due to the ?, any further inconsistencies in M will follow from the fourth symbol in the sequence, and no previous symbols will effect the consistency. See Cases 1, 2, B, and ?.

Subcase 0?

The sequence 0? can be followed consistently by anything but a 2. The sequence 0?2 is an inconsistency of length 3. 0?0 cycles back to Case 0. 0?1 is a repetition of subcase 01? in reverse order if followed by a 0 or an inconsistency of length 4 if followed by a 1 as 0?11. 0?12 is an inconsistency of length 2 while 0?1B is consistent and any further inconsistencies follow in Case B. 0?? can be followed consistently by any symbol and the consistency of M following 0?B can be referred to Case B starting over at length 1.

This exhausts the possibilities in Case 0. The largest inconsistent sequences found thus far are of length 4.

Case 1

A 1 can be followed consistently by a 0, 1, B or ?. If followed by a 2, M has an inconsistency of length 2.

Subcases 10, 1B and 1?

Any further inconsistencies in 10X, 1BX or 1?X follow from the 0, B or ?, respectively. See Cases 0, B, ?.

Subcase 11

The sequence 11 can be followed consistently by the symbols B or ?. Inconsistencies in a sequence 11BX follow from the B and not the 1s. The sequence 11?0 has an inconsistency of length 4, but otherwise sequences 11?X follow from the X and not the 11?. The sequences 110, 111 and 112 are inconsistencies local to 3, 3, and 2 squares, respectively.

This exhausts the possible inconsistencies for Case 1. Still, the largest inconsistency is of length 4.

Case 2

inconsistencies of length 2 occur whenever a 2 is followed by a 0, 1, or another 2. A 2 followed by a B or ? is consistent.

Subcase 2B

Any further inconsistencies follow from the B, not the 2. See Case B.

Subcase 2?

Any symbol can consistently follow 2? except a 0. In this case, we have an inconsistency of length 3. Otherwise, following a ? in 2?X further inconsistencies then following the X depend only on X. See Cases 1, 2, B and ?.

Thus Case 2 is exhausted with the longest inconsistency found being of length 3.

Case B

A B followed by a 0 incurs an inconsistency of length 2. Any other symbol following a B is consistent. A sequence BBX cycles back to the current case. For sequences B2X and B?X, further inconsistencies are incurred following from the 2 or ?.

Subcase B1

The sequence B1 can be followed consistently by anything except a B or a 2. B1B is an inconsistency of length 3 and B12 is an inconsistency of length 2. For the sequences B10X and B11X, the initial B no longer can affect the subsequent consistency of the string. The consistency will then follow from the 0 or 1, respectively. However, the sequence B1?2 is an inconsistency of length 4.

The longest inconsistency found in Case B is of length 4.

Case ?

Any symbol Y can consistently follow a ?. The consistency of M following from ?Y depends further on Y alone.

Through each case, the longest inconsistency found was local to a sequence of no more than four symbols. This proves that inconsistencies in all minesweeper strings occur in substrings of length at most four.

Now, I define an inconsistency classification scheme (arbitrarily) according to types and subtypes of possible inconsistencies in a length four substring. Table 2 shows that there are three general types of inconsistencies: 1e, 2e and Be. Inconsistency type 1e has subtypes 010X, 011X, 111X, 01?0 and 11?0; inconsistency type 2e has subtypes 02XX, 12XX, 22XX and 0?2X; and inconsistency type Be has subtypes 0BXX, B1BX and B1?2. I will now display all possible length four substrings and show to which inconsistency type and subtype each of the inconsistent substrings belongs.

**Table 3** *Inconsistency classification scheme*

| type of inconsistency | subtypes |
|---|---|
| 1e | 010X, 011X, 111X, 01?0, 11?0 |
| 2e | 02XX, 12XX, 22XX, 0?2X |
| Be | 0BXX, B1BX, B1?2 |

The tables in Appendix A contain all 625 possibilities for minesweeper substrings of length four, with consistent substrings shown in the first column and inconsistent substrings in the second column. In the third and fourth columns, the inconsistency type and subtype is given, respectively. Note that many strings are inconsistent in more than one way, but the inconsistency type is assigned as per the first inconsistency encountered from left to right. Also, observe that there are redundancies in the substrings since many strings are identical up to flips about the space between the second and third squares. Such identical inconsistent strings are categorized under the same inconsistency subtypes, and will hence be detected in the same manner. For example, the strings 210B and 112B are different, but they are both classified as inconsistencies of subtype 12XX. This is because in either case the first inconsistency detected by the machine, going from left to right, is the 2 next to the 1. It is not matieral that the 2 and 1 are in a different order, or that they are preceded or followed by different symbols. In this light, it can be said that all inconsistent substrings of the same subtype are the same with respect to their inconsistencies.

Now I will show how the given one-dimensional MCP FSM detects each inconsistency subtype, and thus each inconsistent substring. I will take cases on each inconsistency subtype $X_1X_2X_3X_4$ and show how the state transition function, ∂, leads to an inconsistency in each case. Please refer to Table 1 to verify the mappings of ∂.

Subtype 010X

The state after $X_1$ is 0. Then $\partial(0,1)=1_o$, $\partial(1_o,0)$=Er. The inconsistency has been detected.

Subtype 011X

The state after $X_1$ is 0. Then $\partial(0,1)=1_o$, $\partial(1_o,1)$=Er. The inconsistency has been detected.

Subtype 111X

After $X_1$, the state is either $1_o$, $1_b$, or $1_?$. If $S=1_o$, $\partial(1_o,1)$=Er. If $S=1_b$, $\partial(1_b,1)=1_o$, then $\partial(1_o,1)$=Er. If $S=1_?$, then $\partial(1_?,1)$=1o and $\partial(1_o,1)$=Er. In any case, the delta function has maps the substring to the inconsistency state.

Subtype 01?0

After $X_1$, S=0. Then $\partial(0,1)=1_o$, $\partial(1_o,?)=?_b$, and $d(?_b,0)$=Er. The inconsistency has been detected.

Subtype 0?11

After $X_1$, S=0. Then $\partial(0,?)$=?o, $\partial(?_o,1)=1_o$, and $\partial(1_o,1)$=Er. The inconsistency has been detected.

Subtype 02XX

After $X_1$, S=0. Then $\partial(0,2)$=Er. The inconsistency has been detected.

Subtype 0?2X

After $X_1$, S=0. Then $\partial(0,?)=?_o$ and $\partial(?_o,2)$=Er. The inconsistency has been detected.

Subtype 12XX

After $X_1$, $S=1_o$, $1_b$ or $1_?$. For any of these three states, ∂ maps 2 to the error state, Er. As such, it detects the inconsistency.

Subtype 22XX

After $X_1$, S=2. Then $\partial(2,2)$=Er. The inconsistency has been detected.

Subtype 0BXX

After $X_1$, S=0. Then $\partial(0,B)$=Er. The inconsistency has been detected.

Subtype B1BX

After $X_1$, S=B. Then $\partial(B,1)=1_b$ and $\partial(1_b,B)$=Er. The inconsistency has been detected.

Subtype B1?2

After $X_1$, S=B. Then $\partial(B,1)=1_b$, $\partial(1_b,?)=?_o$ and $\partial(?_o,2)$=Er. The inconsistency has been detected.

This shows that the one-dimensional MCP FSM given detects all possible locally inconsistent length four substrings. ■

**Theorem 7.8** *The one-dimensional MCP FSM decides the global consistency of any minesweeper string of length n∈N.*

**Proof.** Let M be a minesweeper string of length n. Suppose n=1. Clearly, the only consistent length one minesweeper strings are M=0, B, or ?. See Table 1 to verify that the length one input sequences 1e and 2e are detected as inconsistent, while the input sequences 0e, Be, and ?e are detected as consistent. If n=2 or 3, then the MCP FSM decides the consistency of M because it was proven in Theorem 1 that all possible local inconsistencies occur in a sequence of at most four symbols. For a minesweeper string of length 2 or 3, any local inconsistency is a global inconsistency. Suppose M=$X_1X_2...X_n$ is a minesweeper string of length n for some n∈N, n≥4, and the MCP FSM has decided the global consistency of M. Now consider the string M', of length n+1, which is the concatenation of M and one additional symbol Y∈ ∑. If M was found to have been globally inconsistent by the MCP FSM, then M' will clearly also be globally inconsistent. Otherwise, if M was found to be globally consistent, the global consistency of M' depends both on the local consistency of the substring $X_{n-2}X_{n-1}X_nY$ and whether M' is consistent with respect to its right end by Theorem 1. If this substring is locally consistent and M' is consistent with respect to its right end, then M' is globally consistent. If either of these two conditions fail, then M' is globally inconsistent. Thus by the principle of induction, the MCP FSM decides the consistency of any minesweeper string of length n∈N. ■

Now I would like to give my final theorem.

**Theorem 7.9** *The output map beta in the one-dimensional MCP FSM maps consistent minesweeper strings to consistent bomb strings.*

**Proof.** Let M be a consistent minesweeper string and let B be the bomb string produced by the MCP FSM with input M. Then the MCP FSM has not at any time been in the error state, Er, under the mappings of the state transition function by the definition of delta (see Table 1). Hence beta, the output function, has not produced the character X from the alphabet $\sum_B$={-, B, X} since it only does so when S=Er. Thus the bomb string produced is restricted to symbols from the alphabet {-, B}, and must be consistent by definition.

Conversely, suppose B is a consistent bomb string produced by the input of M, where M is some minesweeper string. Then B consists only of characters from the restricted alphabet {-, B}. The states that map to {-, B} under beta are precisely the accepting states of the MCP FSM. On the other hand, the only non-accepting state, Er, is the only state that maps to {X}. Thus if B is consistent, then never has the MCP FSM been in the error state while processing M. This is sufficient to conclude that M is consistent. ■

Theorem 3 suggests that the MCP FSM outputs a consistent bomb configuration corresponding to each consistent minesweeper string. The bomb configuration outputted may be only one of a number of possible bomb configurations, and a nondeterministic finite state machine could be constructed that could determine all possible consistent bomb configurations. This would be a good starting point for future inquiry.

In conclusion, since the one-dimensional MCP FSM gives a deterministic, polynomial time method of deciding the consistency of any minesweeper string, and hence any minesweeper puzzle (providing a consistent configuration of bombs corresponding for every consistent puzzle) by Theorems 1,2, and 3, one-dimensional MCP is easy.

## 8 Conclusion

In my research this summer, I have coded a program which finds all consistent configurations to a given minesweeper grid. I have also defined the one-dimensional minesweeper consistency problem and shown that it is easy, unlike its two-dimensional counter part. I would like to end this paper with a list of topics for further inquiry into the minesweeper consistency problem and ways in which my findings could be refined.

- My one-dimensional MCP FSM lacks precision in its definition and the proof seems to be flawed.

- The efficiency of the MS consistency program has room for improvement.

- We wanted to find a finite state generator for two-dimensional minesweeper and use it to decide consistency.

- We thought it might be interesting to investigate the conditions under which the consistency of a grid could be predicted.

# 9 References and Appendices

## 9.1 References

[G85] Alan Gibbons *Algorithmic Graph Theory.* Cambridge University Press, Cambridge, 1985.

[GJ79] M. Garey and D. Johnson *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freedman, San Francisco, 1979.

[HU79] J. Hopcroft and J. Ullman *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, USA, 1979

[K00] Richard Kaye Minesweeper is NP-Complete. *The Mathematical Intelligencer*. 22:9-15, Spring 2000

[M03] Neville Mehta Minesweeper is NP-Complete. Unpublished class project paper Department of Computer Science, Oregon State University. Corvallis, Oregon. Spring 2003.

Richard Kaye's minesweeper website

http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm

Website with interesting minesweeper strategies and information

http://www.frankwester.net/winmine.html

## 9.2 Appendix A

```
/*******************************************************/
/* */
/* MS CONSISTENCY */
/* */
/*******************************************************/
#include ”stdafx.h”
#include<iostream>
using namespace std;
// Global constants to define grid size
const int ROWS=3;
const int COLS=3;
const int DIM=ROWS*COLS;
const int MAXROW=1000;
// Function declarations
void bfs( char[ROWS][COLS], int& );
//general
void grid_to_string( char[ROWS][COLS], char[DIM] );
void string_to_grid( char[DIM], char[ROWS][COLS] );
void print_frontier( char[MAXROW][DIM] );
//functions used in bfs
void load_string( char[DIM], char[MAXROW][DIM], int );
void unload_string( char[DIM], char[MAXROW][DIM], int );
int count_blanks_string( char[DIM] );
int find_next_blank( char[DIM], int );
void count_blanks_per_row( char[MAXROW][DIM], int[MAXROW] );
int count_new_grids( int[MAXROW] );
void add_bomb( char[DIM], int );
void load_consistent( char[MAXROW][2], char[MAXROW][DIM], char[MAXROW][DIM] );
//functions to calculate consistency/goal-status and output
//goal-state grids
void print_array ( char[ROWS][COLS] );
int char_to_int ( char );
char int_to_char ( int );
int surrounding_bombs ( char[ROWS][COLS], int, int );
int surrounding_blanks ( char[ROWS][COLS], int, int );
char consistency ( char[ROWS][COLS] );
char goal_state ( char[ROWS][COLS] );
void goal_state_outputter( char[ROWS][COLS], int& );
void get_info( char[MAXROW][2], char[MAXROW][DIM], int& );
//**********************************************//
// B F S //
//**********************************************//
void bfs ( char Original[ROWS][COLS], int &gs_counter )
{
//indices
int i,j,k;
char MSstring[DIM];
for (i=0; i<DIM; i++)
MSstring[i]='0';
//Load original grid onto MSstring
grid_to_string( Original, MSstring );
//New grids counts number of new grids expected in next search iteration.
int new_grids=0;
//MSarray is the array which will contain the entire search frontier.
char MSarray[MAXROW][DIM];
for (i=0; i<MAXROW; i++)
for (j=0; j<DIM; j++)
MSarray[i][j]='X';
//Load original grid, in the form of MSstring, onto MSarray
load_string( MSstring, MSarray, 0 );
//array to hold number of blanks per row in MSarray.
//Upon this information will depend the size of the new search frontier
int no_blanks[MAXROW];
//Initialize no_blanks[]
for (i=0; i<MAXROW; i++)
no_blanks[i]=-1;
//Count number of blanks per row in MSarray and store in no_blanks
count_blanks_per_row( MSarray, no_blanks );
int total_blanks=0;
total_blanks=no_blanks[0];
//Trivial case in which the Original has no ?s yet is consistent
if (total_blanks==0)
{
print_array( Original );
gs_counter++;
}
else
{
```

```
//reconfigure size of search frontier. Initially, search frontier = #of'?'s + 1.
new_grids=count_new_grids( no_blanks );
}
//variables to keep track of position in MSarray of ?'s and B to be added
int next, last;
int bomb_position=-1;
//variable to keep track of row in NewArray to be filled
int new_row;
char NewArray[MAXROW][DIM];
char consis_gs_info[MAXROW][2];
while ( new_grids > 0 && new_grids < MAXROW )
{
for (i=0; i<MAXROW; i++)
for (int j=0; j<DIM; j++)
NewArray[i][j]='X';
for (i=0; i<MAXROW; i++)
for (int j=0; j<2; j++)
consis_gs_info[i][j]='X';
//reinitialize new_row
new_row=0;
//reinitialize MSstring
for (k=0; k<DIM; k++)
MSstring[k]='0';
//Stop_row will detect whether next row in MSarray is empty at end of one iteration in loop.
char stop_row=MSarray[0][0];
//The following loop will create, by adding a bomb in the appropriate spot,
//a string for each new grid i expected, and load it into row i in
//NewArray. It will consider each row in MSarray.
for (i=0; i<MAXROW && stop_row!='X'; i++)
{
//unload row i into MSstring
unload_string( MSstring, MSarray, i );
//Initializing last.
last=-1;
//find the first ? in row i.
next=find_next_blank( MSstring, last );
for (int j=0; j<=no_blanks[i]; j++)
{
//position to place next bomb is assigned to
//bomb_position. Note that, if bomb_position=DIM (meaning that there
//are no more bombs in the row), add_bomb will just fill the remaining spots
//in the string with '-'s.
bomb_position=next;
//add bomb to MSstring and load in new array
add_bomb( MSstring, bomb_position );
load_string( MSstring, NewArray, new_row );
//increase new_row count each time
new_row++;
//recalculate next and reset MSstring
last=next;
next=find_next_blank( MSstring, last );
unload_string( MSstring, MSarray, i );
}
stop_row=MSarray[i+1][0];
}
//Now, NewArray should be completely filled.
//Check and make note of consistency and goal-status of each string in NewArray
//Outputs and counts goal-state grids. At this point, goal-state grids will be printed.
get_info( consis_gs_info, NewArray, gs_counter );
//Reinitialize MSarray and no_blanks
for (i=0; i<MAXROW; i++)
for (j=0; j<DIM; j++)
MSarray[i][j]='X';
for (i=0; i<MAXROW; i++)
no_blanks[i]=-1;
//Reload MSarray with grids in NewArray which are consistent
//and still have spaces with ?s.
load_consistent( consis_gs_info, NewArray, MSarray );
//Count number of blanks per row in MSarray and store in no_blanks
count_blanks_per_row( MSarray, no_blanks );
//Count total blanks
total_blanks=0;
int stop=no_blanks[0];
for (i=0; i<MAXROW && stop!=-1; i++)
{
total_blanks+=no_blanks[i];
stop=no_blanks[i+1];
}
if (total_blanks==0)
{
```

```
new_grids=0;
}
else
{
//recalculate size of search frontier
new_grids=count_new_grids( no_blanks );
if ( new_grids > MAXROW )
cout<<endl<<"Search frontier too large"<<endl;
}
}
}
//******************************************************************//
//General functions//
//******************************************************************//
//Loads a two-dim array of size ROWS by COLUMNS into an 1-dim array of size
// DIM=ROWS*COLUMNS, row-by-row.
void grid_to_string ( char Grid[ROWS][COLS], char String[DIM] )
{
int row, column;
for( int k=0; k<DIM; k++)
{
row=k/COLS;
column=k%COLS;
String[k]=Grid[row][column];
}
}
//Loads a one-dim array of size DIM=ROWS*COLUMNS into a 2-dim array of size
// ROWS by COLUMNS, row-by-row.
void string_to_grid ( char String[DIM], char Grid[ROWS][COLS] )
{
int row, column;
for (int k=0; k<DIM; k++)
{
row=k/COLS;
column=k%COLS;
Grid[row][column]=String[k];
}
}
void print_frontier ( char Array[MAXROW][DIM] )
{
```

```
int i,j,stop;
stop=Array[0][0];
cout<<endl;
for (i=0; i<MAXROW && stop!='X'; i++)
{
for (j=0; j<DIM; j++)
cout<<Array[i][j];
cout<<endl;
stop=Array[i+1][0];
}
}
//*****************************************************************//
//The following functions are involved in the bfs//
//*****************************************************************//
void load_string( char MSstring[DIM], char MSarray[MAXROW][DIM], int i )
{
for (int k=0; k<DIM; k++)
MSarray[i][k]=MSstring[k];
}
void unload_string( char MSstring[DIM], char MSarray[MAXROW][DIM], int i )
{
for (int k=0; k<DIM; k++)
MSstring[k]=MSarray[i][k];
}
int count_blanks_string( char MSstring[DIM] )
{
int blank_count=0;
for (int k=0; k<DIM; k++)
{
if ( MSstring[k]=='?')
blank_count++;
}
return blank_count;
}
int find_next_blank( char MSstring[DIM], int last )
{
int next, i;
//loop to find the next blank on the string, beginning with the one
//after the last.
for (i=(last+1); i<DIM && MSstring[i]!='?'; i++);
```

```
//if there are no more blanks found on the string, assign
//i the value of DIM.
if (i==(DIM-1) && MSstring[i]!='?')
i++;
next=i;
return next;
}
void count_blanks_per_row( char MSarray[MAXROW][DIM], int no_blanks[MAXROW] )
{
//variable to detect when next row is empty
char stop='0';
//search rows of MSarray for ?s and count
int i;
for (i=0; i<MAXROW && stop!='X'; i++)
{
no_blanks[i]=0;
for (int j=0; j<DIM; j++)
if (MSarray[i][j]=='?')
no_blanks[i]++;
stop=MSarray[i+1][0];
}
}
int count_new_grids( int no_blanks[MAXROW] )
{
int new_grids=0;
int stop,i;
stop=no_blanks[0];
for (i=0; i<MAXROW && stop!=-1; i++)
{
if ( no_blanks[i]>0 )
new_grids+=(no_blanks[i]+1);
stop=no_blanks[i+1];
}
return new_grids;
}
void add_bomb( char MSstring[DIM], int bomb_position )
{
if ( (bomb_position < DIM) && (0 <= bomb_position) )
MSstring[bomb_position]='B';
for (int k=0; k<bomb_position; k++)
```

```
{
if (MSstring[k]=='?')
MSstring[k]='-';
}
}
void load_consistent( char consis_gs_info[MAXROW][2], char NewArray[MAXROW][DIM],
char MSarray[MAXROW][DIM] )
{
char tempString[DIM];
int blankcount=0;
int grid_no=0;
for (int i=0; i<MAXROW; i++)
{
for (int k=0; k<DIM; k++)
tempString[k]='0';
if ( consis_gs_info[i][0]=='Y')
{
unload_string( tempString, NewArray, i );
blankcount=count_blanks_string( tempString );
if ( blankcount>0 )
{
load_string( tempString, MSarray, grid_no );
grid_no++;
}
}
}
}
//**************************************************//
// consistency/goal-status checking //
//****************************************************//
void print_array ( char Array[ROWS][COLS] )
{
for (int i=0; i<ROWS; i++)
{
for (int j=0; j<COLS; j++)
cout<<Array[i][j];
cout<<endl;
}
}
int char_to_int (char digit)
```

```
{
int integer;
switch ( digit )
{
case ’0’: integer=0;
break;
case ’1’: integer=1;
break;
case ’2’: integer=2;
break;
case ’3’: integer=3;
break;
case ’4’: integer=4;
break;
case ’5’: integer=5;
break;
case ’6’: integer=6;
break;
case ’7’: integer=7;
break;
case ’8’: integer=8;
break;
default: integer=9;
break;
}
return integer;
}
char int_to_char (int integer)
{
char character;
switch ( integer )
{
case 0: character=’0’;
break;
case 1: character=’1’;
break;
case 2: character=’2’;
break;
case 3: character=’3’;
break;
```

```
case 4: character='4';
break;
case 5: character='5';
break;
case 6: character='6';
break;
case 7: character='7';
break;
case 8: character='8';
break;
default: character='-';
break;
}
return character;
}
int surrounding_bombs ( char MSgrid[ROWS][COLS], int row, int col )
{
int bombcount=0;
if ( row==0 || col==0 || row==(ROWS-1) || col==(COLS-1) )
{
if ( row==0 && col==0 )
{
if ( MSgrid[0][1]=='B')
bombcount++;
if ( MSgrid[1][0]=='B')
bombcount++;
if ( MSgrid[1][1]=='B')
bombcount++;
}
if ( row==0 && col==(COLS-1) )
{
if ( MSgrid[0][col-1]=='B' )
bombcount++;
if ( MSgrid[1][col]=='B')
bombcount++;
if ( MSgrid[1][col-1]=='B' )
bombcount++;
}
if ( row==(ROWS-1) && col==0 )
{
```

```
if ( MSgrid[row-1][0]==’B’ )
bombcount++;
if ( MSgrid[row-1][1]==’B’ )
bombcount++;
if ( MSgrid[row][1]==’B’ )
bombcount++;
}
if ( row==(ROWS-1) && col==(COLS-1) )
{
if ( MSgrid[row-1][col-1]==’B’ )
bombcount++;
if ( MSgrid[row-1][col]==’B’ )
bombcount++;
if ( MSgrid[row][col-1]==’B’ )
bombcount++;
}
if ( (row==0) && (col!=0) && (col!=(COLS-1)) )
{
for (int i=0; i<2; i++)
for (int j=(col-1); j<=(col+1); j++)
if (MSgrid[i][j]==’B’)
bombcount++;
}
if ( (row==(ROWS-1)) && (col!=0) && (col!=(COLS-1)) )
{
for (int i=(row-1); i<=row; i++)
for (int j=(col-1); j<=(col+1); j++)
if (MSgrid[i][j]==’B’)
bombcount++;
}
if ( (col==0) && (row!=0) && (row!=(ROWS-1)) )
{
for (int i=(row-1); i<=(row+1); i++)
for (int j=0; j<2; j++)
if (MSgrid[i][j]==’B’)
bombcount++;
}
if ( (col==(COLS-1)) && (row!=0) && (row!=(ROWS-1)) )
{
for (int i=(row-1); i<=(row+1); i++)
for (int j=(col-1); j<=col; j++)
if (MSgrid[i][j]=='B')
bombcount++;
}
}
else
{
for ( int i=(row-1); i<=(row+1); i++)
{
for ( int j=(col-1); j<=(col+1); j++)
{
if ( MSgrid[i][j] == 'B' )
bombcount++;
}
}
}
return bombcount;
}
int surrounding_blanks ( char MSgrid[ROWS][COLS], int row, int col )
{
int blankcount=0;
if ( row==0 || col==0 || row==(ROWS-1) || col==(COLS-1) )
{
if ( row==0 && col==0 )
{
if ( MSgrid[0][1]!='B' || MSgrid[0][1]!='?' )
blankcount++;
if ( MSgrid[1][0]!='B' || MSgrid[1][0]!='?' )
blankcount++;
if ( MSgrid[1][1]!='B' || MSgrid[1][1]!='?' )
blankcount++;
}
if ( row==0 && col==(COLS-1) )
{
if ( MSgrid[0][col-1]!='B' || MSgrid[0][col-1]!='?' )
blankcount++;
if ( MSgrid[1][col]!='B' || MSgrid[1][col]!='?')
blankcount++;
if ( MSgrid[1][col-1]!='B' || MSgrid[1][col-1]!='?' )
blankcount++;
```

```
}
if ( row==(ROWS-1) && col==0 )
{
if ( MSgrid[row-1][0]!='B' || MSgrid[row-1][0]!='?' )
blankcount++;
if ( MSgrid[row-1][1]!='B' || MSgrid[row-1][1]!='?' )
blankcount++;
if ( MSgrid[row][1]!='B' || MSgrid[row][1]!='?' )
blankcount++;
}
if ( row==(ROWS-1) && col==(COLS-1) )
{
if ( MSgrid[row-1][col-1]!='B' || MSgrid[row-1][col-1]!='?' )
blankcount++;
if ( MSgrid[row-1][col]!='B' || MSgrid[row-1][col]!='?' )
blankcount++;
if ( MSgrid[row][col-1]!='B' || MSgrid[row][col-1]!='?' )
blankcount++;
}
if ( (row==0) && (col!=0) && (col!=(COLS-1)) )
{
for (int i=0; i<2; i++)
for (int j=(col-1); j<=(col+1); j++)
if ( MSgrid[i][j]!='B' || MSgrid[i][j]!='?' )
blankcount++;
}
if ( (row==(ROWS-1)) && (col!=0) && (col!=(COLS-1)) )
{
for (int i=(row-1); i<=row; i++)
for (int j=(col-1); j<=(col+1); j++)
if ( MSgrid[i][j]!='B' || MSgrid[i][j]!='?' )
blankcount++;
}
if ( (col==0) && (row!=0) && (row!=(ROWS-1)) )
{
for (int i=(row-1); i<=(row+1); i++)
for (int j=0; j<2; j++)
if ( MSgrid[i][j]!='B' || MSgrid[i][j]!='?' )
blankcount++;
}
```

```
if ( (col==(COLS-1)) && (row!=0) && (row!=(ROWS-1)) )
{
for (int i=(row-1); i<=(row+1); i++)
for (int j=(col-1); j<=col; j++)
if ( MSgrid[i][j]!='B' || MSgrid[i][j]!='?' )
blankcount++;
}
}
else
{
for ( int i=(row-1); i<=(row+1); i++)
{
for ( int j=(col-1); j<=(col+1); j++)
{
if ( MSgrid[i][j]!='B' || MSgrid[i][j]!='?' )
blankcount++;
}
}
}
return blankcount;
}
int surrounding_unknowns ( char MSgrid[ROWS][COLS], int row, int col )
{
int unknowns=0;
if ( row==0 || col==0 || row==(ROWS-1) || col==(COLS-1) )
{
if ( row==0 && col==0 )
{
if ( MSgrid[0][1]=='?')
unknowns++;
if ( MSgrid[1][0]=='?')
unknowns++;
if ( MSgrid[1][1]=='?')
unknowns++;
}
if ( row==0 && col==(COLS-1) )
{
if ( MSgrid[0][col-1]=='?' )
unknowns++;
if ( MSgrid[1][col]=='?')
```

```
unknowns++;
if ( MSgrid[1][col-1]==’?’ )
unknowns++;
}
if ( row==(ROWS-1) && col==0 )
{
if ( MSgrid[row-1][0]==’?’ )
unknowns++;
if ( MSgrid[row-1][1]==’?’ )
unknowns++;
if ( MSgrid[row][1]==’?’ )
unknowns++;
}
if ( row==(ROWS-1) && col==(COLS-1) )
{
if ( MSgrid[row-1][col-1]==’?’ )
unknowns++;
if ( MSgrid[row-1][col]==’?’ )
unknowns++;
if ( MSgrid[row][col-1]==’?’ )
unknowns++;
}
if ( (row==0) && (col!=0) && (col!=(COLS-1)) )
{
for (int i=0; i<2; i++)
for (int j=(col-1); j<=(col+1); j++)
if (MSgrid[i][j]==’?’)
unknowns++;
}
if ( (row==(ROWS-1)) && (col!=0) && (col!=(COLS-1)) )
{
for (int i=(row-1); i<=row; i++)
for (int j=(col-1); j<=(col+1); j++)
if (MSgrid[i][j]==’?’)
unknowns++;
}
if ( (col==0) && (row!=0) && (row!=(ROWS-1)) )
{
for (int i=(row-1); i<=(row+1); i++)
for (int j=0; j<2; j++)
```

```
if (MSgrid[i][j]=='?')
unknowns++;
}
if ( (col==(COLS-1)) && (row!=0) && (row!=(ROWS-1)) )
{
for (int i=(row-1); i<=(row+1); i++)
for (int j=(col-1); j<=col; j++)
if (MSgrid[i][j]=='?')
unknowns++;
}
}
else
{
for ( int i=(row-1); i<=(row+1); i++)
{
for ( int j=(col-1); j<=(col+1); j++)
{
if ( MSgrid[i][j] == '?' )
unknowns++;
}
}
}
return unknowns;
}
char consistency ( char MSgrid[ROWS][COLS] )
{
int bombs_expected=0;
int bombs=0;
int qs=0;
int bombs_qs=0;
char consistency='Y';
int i,j;
for (i=0; i<ROWS && consistency!='N'; i++)
for (j=0; j<COLS && consistency!='N'; j++)
{
bombs_expected=char_to_int( MSgrid[i][j] );
if ( bombs_expected!=9 )
{
bombs=surrounding_bombs( MSgrid, i, j );
qs=surrounding_unknowns( MSgrid, i, j );
```

```
bombs_qs=bombs+qs;
if ( (bombs>bombs_expected) || (bombs_qs<bombs_expected) )
consistency='N';
}
}
return consistency;
}
char goal_state ( char MSgrid[ROWS][COLS] )
{
char goal_state='Y';
int blankcount=0;
for (int i=0; i<ROWS; i++)
for (int j=0; j<COLS; j++)
if (MSgrid[i][j]=='?')
blankcount++;
if (blankcount > 0)
goal_state='N';
else
{
int bombs_expected=0;
int bombs=0;
int i, j;
for ( i=0; i<ROWS && goal_state=='Y'; i++ )
{
for ( j=0; j<COLS && goal_state=='Y'; j++ )
{
bombs_expected=char_to_int( MSgrid[i][j] );
if ( bombs_expected != 9 )
{
bombs=surrounding_bombs( MSgrid, i, j );
if ( bombs_expected != bombs )
goal_state='N';
}
}
}
}
return goal_state;
}
void gs_outputter ( char MSgrid[ROWS][COLS], int &gs_counter )
{
```

```
gs_counter++;
int i,j;
int bombcount;
for (i=0; i<ROWS; i++)
for(j=0; j<COLS; j++)
{
if ( MSgrid[i][j]=='-' )
{
bombcount=0;
bombcount=surrounding_bombs( MSgrid, i, j );
MSgrid[i][j]=int_to_char(bombcount);
}
}
cout<<gs_counter<<endl;
print_array( MSgrid );
cout<<endl;
}
void get_info( char consis_gs_info[MAXROW][2], char NewArray[MAXROW][DIM], int &gs_counter
)
{
char consistent, gs;
int i,j,k;
char tempString[DIM];
for (k=0; k<DIM; k++)
tempString[k]='0';
char tempGrid[ROWS][COLS];
for (i=0; i<ROWS; i++)
for (j=0; j<COLS; j++)
tempGrid[i][j]='0';
char stop='0';
for (i=0; i<MAXROW && stop!='X'; i++)
{
unload_string( tempString, NewArray, i );
string_to_grid( tempString, tempGrid );
consistent=consistency( tempGrid );
gs=goal_state( tempGrid );
if (gs=='Y')
gs_outputter( tempGrid, gs_counter );
consis_gs_info[i][0]=consistent;
consis_gs_info[i][1]=gs;
```

```
stop=NewArray[i+1][0];
}
}
//*****************************************************************//
// M A I N //
//*****************************************************************//
int _tmain(int argc, _TCHAR* argv[])
{
// Fill array representing MS grid
//***************ORIGINAL**GRIDS*******************//
/*
//Board 1
char Original[ROWS][COLS]= {{’?’,’?’},
{’?’,’?’}};
*/
//Board 2
char Original[ROWS][COLS]= {{’?’,’?’,’?’},
{’?’,’?’,’?’},
{’?’,’?’,’?’}};
/*
//Board 3
char Original[ROWS][COLS]= {{’?’,’?’,’?’,’?’},
{’?’,’2’,’2’,’?’},
{’?’,’2’,’2’,’?’},
{’?’,’?’,’?’,’?’}};
*/
/*
Board 4
char Original[ROWS][COLS]= {{’?’,’?’,’?’,’?’,’?’},
{’?’,’2’,’2’,’2’,’?’},
{’?’,’2’,’0’,’2’,’?’},
{’?’,’2’,’2’,’2’,’?’},
{’?’,’?’,’?’,’?’,’?’}};
*/
/*
Board 5
char Original[ROWS][COLS]= {{’?’,’?’,’?’,’?’,’?’,’?’},
{’?’,’2’,’2’,’2’,’2’,’?’},
{’?’,’2’,’0’,’0’,’2’,’?’},
{’?’,’2’,’0’,’0’,’2’,’?’},
{’?’,’2’,’2’,’2’,’2’,’?’},
{’?’,’?’,’?’,’?’,’?’,’?’}};
*/
/*
Board 6
char Original[ROWS][COLS]= {{’2’,’3’,’B’,’2’,’2’,’B’,’2’,’1’},
{’B’,’B’,’5’,’?’,’?’,’4’,’B’,’2’},
{’?’,’?’,’B’,’B’,’B’,’?’,’4’,’B’},
{’B’,’6’,’?’,’6’,’B’,’B’,’?’,’2’},
{’2’,’B’,’B’,’?’,’5’,’5’,’?’,’2’},
{’1’,’3’,’4’,’?’,’B’,’B’,’4’,’B’},
{’0’,’1’,’B’,’4’,’?’,’?’,’B’,’3’},
{’0’,’1’,’2’,’B’,’2’,’3’,’B’,’2’}};
*/
cout<<”Original Grid”<<endl;
print_array( Original );
cout<<endl;
// Call get_consistent to find all consistent configurations
int number_of_configs=0;
//ensure consistency of original grid
char consistent=consistency( Original );
if (consistent==’Y’)
bfs ( Original, number_of_configs );
cout<<number_of_configs<<” consistent configuration(s)”<<endl<<endl;
return 0;
}
/*
Board 1 Sample Run
Original Grid
??
??
1
11
1B
2
00
00
3
B2
2B
```

```
4
B1
11
5
2B
2B
6
1B
11
7
22
BB
8
11
B1
9
BB
3B
10
BB
22
11
B3
BB
12
B2
B2
13
3B
BB
14
2B
B2
15
BB
BB
16
BB
B3
16 consistent configuration(s)
```

```
*/

/*

Board 4 Sample run

Original Grid

?????

?222?

?202?

?222?

?????

1

2B2B2

B222B

22022

B222B

2B2B2

1 consistent configuration(s)

*/
```

## 9.3 Appendix B

*Set of tables listing the 625 possibilities for length four substrings, their error types and subtypes. Order is 0,1,2,B,?.*

### 9.3.1 0XXX

00XX

| 000X | | e | sub-e |
|---|---|---|---|
| 0000 | - | - | - |
| 0001 | - | - | - |
| - | 0002 | 2e | 02XX |
| - | 000B | Be | 12XX |
| 000? | - | - | - |

| 001X | | e | sub-e |
|---|---|---|---|
| - | 0010 | 1e | 010X |
| - | 0011 | 1e | 011X |
| - | 0012 | 2e | 12XX |
| 001B | - | - | - |
| 001? | - | - | - |

| 002X | | e | sub-e |
|---|---|---|---|
| - | 0020 | 2e | 02XX |
| - | 0021 | 2e | 12XX |
| - | 0022 | 2e | 22XX |
| - | 002B | 2e | 02XX |
| - | 002? | 2e | 02XX |

| 00BX | | e | sub-e |
|---|---|---|---|
| - | 00B0 | Be | 0BXX |
| - | 00B1 | Be | 0BXX |
| - | 00B2 | Be | 0BXX |
| - | 00BB | Be | 0BXX |
| - | 00B? | Be | 0BXX |

| 00?X | | e | sub-e |
|---|---|---|---|
| 00?0 | - | - | - |
| 00?1 | - | - | - |
| - | 00?2 | 2e | 0?2X |
| 00?B | - | - | - |
| 00?? | - | - | - |

01XX

| 010X | | e | sub-e |
|---|---|---|---|
| - | 0100 | 1e | 010X |
| - | 0101 | 1e | 010X |
| - | 0102 | 1e | 010X |
| - | 010B | 1e | 010X |
| - | 010? | 1e | 010X |

| 011X | | e | sub-e |
|---|---|---|---|
| - | 0110 | 1e | 011X |
| - | 0111 | 1e | 011X |
| - | 0112 | 1e | 011X |
| - | 011B | 1e | 011X |
| - | 011? | 1e | 011X |

| 012X | | e | sub-e |
|---|---|---|---|
| - | 0120 | 2e | 12XX |
| - | 0121 | 2e | 12XX |
| - | 0122 | 2e | 12XX |
| - | 012B | 2e | 12XX |
| - | 012? | 2e | 12XX |

| 01BX | | e | sub-e |
|---|---|---|---|
| - | 01B0 | Be | 0BXX |
| 01B1 | - | - | - |
| 01B2 | - | - | - |
| 01BB | - | - | - |
| 01B? | - | - | - |

| 01?X | | e | sub-e |
|---|---|---|---|
| - | 01?0 | 1e | 01?0 |
| 01?1 | - | - | - |
| 01?2 | - | - | - |
| 01?B | - | - | - |
| 01?? | - | - | - |

02XX

| 020X | | e | sub-e |
|---|---|---|---|
| - | 0200 | 2e | 02XX |
| - | 0201 | 2e | 02XX |
| - | 0202 | 2e | 02XX |
| - | 020B | 2e | 02XX |
| - | 020? | 2e | 02XX |

| 021X | | e | sub-e |
|---|---|---|---|
| - | 0210 | 2e | 02XX |
| - | 0211 | 2e | 02XX |
| - | 0212 | 2e | 02XX |
| - | 021B | 2e | 02XX |
| - | 021? | 2e | 02XX |

| 022X | | e | sub-e |
|---|---|---|---|
| - | 0220 | 2e | 02XX |
| - | 0221 | 2e | 02XX |
| - | 0222 | 2e | 02XX |
| - | 022B | 2e | 02XX |
| - | 022? | 2e | 02XX |

| 02BX | | e | sub-e |
|---|---|---|---|
| - | 02B0 | 2e | 02XX |
| - | 02B1 | 2e | 02XX |
| - | 02B2 | 2e | 02XX |
| - | 02BB | 2e | 02XX |
| - | 02B? | 2e | 02XX |

| 02?X | | e | sub-e |
|---|---|---|---|
| - | 02?0 | 2e | 02XX |
| - | 02?1 | 2e | 02XX |
| - | 02?2 | 2e | 02XX |
| - | 02?B | 2e | 02XX |
| - | 02?? | 2e | 02XX |

0BXX

| 0B0X | | e | sub-e |
|---|---|---|---|
| - | 0B00 | Be | 0BXX |
| - | 0B01 | Be | 0BXX |
| - | 0B02 | Be | 0BXX |
| - | 0B0B | Be | 0BXX |
| - | 0B0? | Be | 0BXX |

| 0B1X | | e | sub-e |
|---|---|---|---|
| - | 0B10 | Be | 0BXX |
| - | 0B11 | Be | 0BXX |
| - | 0B12 | Be | 0BXX |
| - | 0B1B | Be | 0BXX |
| - | 0B1? | Be | 0BXX |

| 0B2X | | e | sub-e |
|---|---|---|---|
| - | 0B20 | Be | 0BXX |
| - | 0B21 | Be | 0BXX |
| - | 0B22 | Be | 0BXX |
| - | 0B2B | Be | 0BXX |
| - | 0B2? | Be | 0BXX |

| 0BBX | | e | sub-e |
|---|---|---|---|
| - | 0BB0 | Be | 0BXX |
| - | 0BB1 | Be | 0BXX |
| - | 0BB2 | Be | 0BXX |
| - | 0BBB | Be | 0BXX |
| - | 0BB? | Be | 0BXX |

| 0B?X | | e | sub-e |
|---|---|---|---|
| - | 0B?0 | Be | 0BXX |
| - | 0B?1 | Be | 0BXX |
| - | 0B?2 | Be | 0BXX |
| - | 0B?B | Be | 0BXX |
| - | 0B?? | Be | 0BXX |

0?XX

| 0?0X | | e | sub-e |
|---|---|---|---|
| 0?00 | - | - | - |
| 0?01 | - | - | - |
| - | 0?02 | 2e | 02XX |
| - | 0?0B | Be | 0BXX |
| 0?0? | - | - | - |

| 0?1X | | e | sub-e |
|---|---|---|---|
| - | 0?10 | 1e | 01?0 |
| - | 0?11 | 1e | 0?11 |
| - | 0?12 | 2e | 12XX |
| 0?1B | - | - | - |
| 0?1? | - | - | - |

| 0?2X | | e | sub-e |
|---|---|---|---|
| - | 0?20 | 2e | 0?2X |
| - | 0?21 | 2e | 0?2X |
| - | 0?22 | 2e | 0?2X |
| - | 0?2B | 2e | 0?2X |
| - | 0?2? | 2e | 0?2X |

| 0?BX | | e | sub-e |
|---|---|---|---|
| - | 0?B0 | Be | 0BXX |
| 0?B1 | - | - | - |
| 0?B2 | - | - | - |
| 0?BB | - | - | - |
| 0?B? | - | - | - |

| 0??X | | e | sub-e |
|---|---|---|---|
| 0??0 | - | - | - |
| 0??1 | - | - | - |
| 0??2 | - | - | - |
| 0??B | - | - | - |
| 0??? | - | - | - |

### 9.3.2 1XXX

10XX

| 100X | | e | sub-e |
|---|---|---|---|
| 1000 | - | - | - |
| 1001 | - | - | - |
| - | 1002 | 2e | 02XX |
| - | 100B | Be | 0BXX |
| 100? | - | - | - |

| 101X | | e | sub-e |
|---|---|---|---|
| - | 1010 | 1e | 010X |
| - | 1011 | 1e | 011X |
| - | 1012 | 2e | 12XX |
| 101B | - | - | - |
| 101? | - | - | - |

| 102X | | e | sub-e |
|---|---|---|---|
| - | 1020 | 2e | 02XX |
| - | 1021 | 2e | 02XX |
| - | 1022 | 2e | 02XX |
| - | 102B | - | 02XX |
| - | 102? | - | 02XX |

| 10BX | | e | sub-e |
|---|---|---|---|
| - | 10B0 | Be | 0BXX |
| - | 10B1 | Be | 0BXX |
| - | 10B2 | Be | 0BXX |
| - | 10BB | Be | 0BXX |
| - | 10B? | Be | 0BXX |

| 10?X | | e | sub-e |
|---|---|---|---|
| 10?0 | - | - | - |
| 10?1 | - | - | - |
| - | 10?2 | 2e | 0?2x |
| 10?B | - | - | - |
| 10?? | - | - | - |

11XX

| 110X | | e | sub-e |
|---|---|---|---|
| - | 1100 | 1e | 011X |
| - | 1101 | 1e | 011X |
| - | 1102 | 1e | 011X |
| - | 110B | 1e | 011X |
| - | 110? | 1e | 011X |

| 111X | | e | sub-e |
|---|---|---|---|
| - | 1110 | 1e | 111X |
| - | 1111 | 1e | 111X |
| - | 1112 | 1e | 111X |
| - | 111B | 1e | 111X |
| - | 111? | 1e | 111X |

| 112X | | e | sub-e |
|---|---|---|---|
| - | 1120 | 2e | 12XX |
| - | 1121 | 2e | 12XX |
| - | 1122 | 2e | 12XX |
| - | 112B | 2e | 12XX |
| - | 112? | 2e | 12XX |

| 11BX | | e | sub-e |
|---|---|---|---|
| - | 11B0 | Be | 0BXX |
| 11B1 | - | - | - |
| 11B2 | - | - | - |
| 11BB | - | - | - |
| 11B? | - | - | - |

| 11?X | | e | sub-e |
|---|---|---|---|
| - | 11?0 | 1e | 0?11 |
| 11?1 | - | - | - |
| 11?2 | - | - | - |
| 11?B | - | - | - |
| 11?? | - | - | - |

12XX

| 120X | | e | sub-e |
|---|---|---|---|
| - | 1200 | 2e | 12XX |
| - | 1201 | 2e | 12XX |
| - | 1202 | 2e | 12XX |
| - | 120B | 2e | 12XX |
| - | 120? | 2e | 12XX |

| 121X | | e | sub-e |
|---|---|---|---|
| - | 1210 | 2e | 12XX |
| - | 1211 | 2e | 12XX |
| - | 1212 | 2e | 12XX |
| - | 121B | 2e | 12XX |
| - | 121? | 2e | 12XX |

| 122X | | e | sub-e |
|---|---|---|---|
| - | 1220 | 2e | 12XX |
| - | 1221 | 2e | 12XX |
| - | 1222 | 2e | 12XX |
| - | 122B | 2e | 12XX |
| - | 122? | 2e | 12XX |

| 12BX | | e | sub-e |
|---|---|---|---|
| - | 12B0 | 2e | 12XX |
| - | 12B1 | 2e | 12XX |
| - | 12B2 | 2e | 12XX |
| - | 12BB | 2e | 12XX |
| - | 12B? | 2e | 12XX |

| 12?X | | e | sub-e |
|---|---|---|---|
| - | 12?0 | 2e | 12XX |
| - | 12?1 | 2e | 12XX |
| - | 12?2 | 2e | 12XX |
| - | 12?B | 2e | 12XX |
| - | 12?? | 2e | 12XX |

1BXX

| 1B0X | | e | sub-e |
|---|---|---|---|
| - | 1B00 | Be | 0BXX |
| - | 1B01 | Be | 0BXX |
| - | 1B02 | Be | 0BXX |
| - | 1B0B | Be | 0BXX |
| - | 1B0? | Be | 0BXX |

| 1B1X | | e | sub-e |
|---|---|---|---|
| 1B10 | - | - | - |
| 1B11 | - | - | - |
| - | 1B12 | 2e | 12XX |
| - | 1B1B | Be | B1BX |
| 1B1? | - | - | - |

| 1B2X | | e | sub-e |
|---|---|---|---|
| - | 1B20 | 2e | 02XX |
| - | 1B21 | 2e | 12XX |
| - | 1B22 | 2e | 22XX |
| 1B2B | - | - | - |
| 1B2? | - | - | - |

| 1BBX | | e | sub-e |
|---|---|---|---|
| - | 1BB0 | Be | 0BXX |
| 1BB1 | - | - | - |
| 1BB2 | - | - | - |
| 1BBB | - | - | - |
| 1BB? | - | - | - |

| 1B?X | | e | sub-e |
|---|---|---|---|
| 1B?0 | - | - | - |
| 1B?1 | - | - | - |
| 1B?2 | - | - | - |
| 1B?B | - | - | - |
| 1B?? | - | - | - |

1?XX

| 1?0X | | e | sub-e |
|---|---|---|---|
| 1?00 | - | - | - |
| 1?01 | - | - | - |
| - | 1?02 | 2e | 02XX |
| - | 1?0B | Be | 0BXX |
| 1?0? | - | - | - |

| 1?1X | | e | sub-e |
|---|---|---|---|
| 1?10 | - | - | - |
| 1?11 | - | - | - |
| - | 1?12 | 2e | 12XX |
| 1?1B | - | - | - |
| 1?1? | - | - | - |

| 1?2X | | e | sub-e |
|---|---|---|---|
| - | 1?20 | 2e | 02XX |
| - | 1?21 | 2e | 12XX |
| - | 1?22 | 2e | 22XX |
| 1?2B | - | - | - |
| 1?2? | - | - | - |

| 1?BX | | e | sub-e |
|---|---|---|---|
| - | 1?B0 | Be | 0BXX |
| 1?B1 | - | - | - |
| 1?B2 | - | - | - |
| 1?BB | - | - | - |
| 1?B? | - | - | - |

| 1??X | | e | sub-e |
|---|---|---|---|
| 1??0 | - | - | - |
| 1??1 | - | - | - |
| 1??2 | - | - | - |
| 1??B | - | - | - |
| 1??? | - | - | - |

### 9.3.3 2XXX

20XX

| 200X | | e | sub-e |
|---|---|---|---|
| - | 2000 | 2e | 02XX |
| - | 2001 | 2e | 02XX |
| - | 2002 | 2e | 02XX |
| - | 200B | 2e | 02XX |
| - | 200? | 2e | 02XX |

| 201X | | e | sub-e |
|---|---|---|---|
| - | 2010 | 2e | 02XX |
| - | 2011 | 2e | 02XX |
| - | 2012 | 2e | 02XX |
| - | 201B | 2e | 02XX |
| - | 201? | 2e | 02XX |

| 202X | | e | sub-e |
|---|---|---|---|
| - | 2020 | 2e | 02XX |
| - | 2021 | 2e | 02XX |
| - | 2022 | 2e | 02XX |
| - | 202B | 2e | 02XX |
| - | 202? | 2e | 02XX |

| 20BX | | e | sub-e |
|---|---|---|---|
| - | 20B0 | 2e | 02XX |
| - | 20B1 | 2e | 02XX |
| - | 20B2 | 2e | 02XX |
| - | 20BB | 2e | 02XX |
| - | 20B? | 2e | 02XX |

| 20?X | | e | sub-e |
|---|---|---|---|
| - | 20?0 | 2e | 02XX |
| - | 20?1 | 2e | 02XX |
| - | 20?2 | 2e | 02XX |
| - | 20?B | 2e | 02XX |
| - | 20?? | 2e | 02XX |

21XX

| 210X | | e | sub-e |
|---|---|---|---|
| - | 2100 | 2e | 12XX |
| - | 2101 | 2e | 12XX |
| - | 2102 | 2e | 12XX |
| - | 210B | 2e | 12XX |
| - | 210? | 2e | 12XX |

| 211X | | e | sub-e |
|---|---|---|---|
| - | 2110 | 2e | 12XX |
| - | 2111 | 2e | 12XX |
| - | 2112 | 2e | 12XX |
| - | 211B | 2e | 12XX |
| - | 211? | 2e | 12XX |

| 212X | | e | sub-e |
|---|---|---|---|
| - | 2120 | 2e | 12XX |
| - | 2121 | 2e | 12XX |
| - | 2122 | 2e | 12XX |
| - | 212B | 2e | 12XX |
| - | 212? | 2e | 12XX |

| 21BX | | e | sub-e |
|---|---|---|---|
| - | 21B0 | 2e | 12XX |
| - | 21B1 | 2e | 12XX |
| - | 21B2 | 2e | 12XX |
| - | 21BB | 2e | 12XX |
| - | 21B? | 2e | 12XX |

| 21?X | | e | sub-e |
|---|---|---|---|
| - | 21?0 | 2e | 12XX |
| - | 21?1 | 2e | 12XX |
| - | 21?2 | 2e | 12XX |
| - | 21?B | 2e | 12XX |
| - | 21?? | 2e | 12XX |

22XX

| 220X | | e | sub-e |
|---|---|---|---|
| - | 2200 | 2e | 22XX |
| - | 2201 | 2e | 22XX |
| - | 2202 | 2e | 22XX |
| - | 220B | 2e | 22XX |
| - | 220? | 2e | 22XX |

| 221X | | e | sub-e |
|---|---|---|---|
| - | 2210 | 2e | 22XX |
| - | 2211 | 2e | 22XX |
| - | 2212 | 2e | 22XX |
| - | 221B | 2e | 22XX |
| - | 221? | 2e | 22XX |

| 222X | | e | sub-e |
|---|---|---|---|
| - | 2220 | 2e | 22XX |
| - | 2221 | 2e | 22XX |
| - | 2222 | 2e | 22XX |
| - | 222B | 2e | 22XX |
| - | 222? | 2e | 22XX |

| 22BX | | e | sub-e |
|---|---|---|---|
| - | 22B0 | 2e | 22XX |
| - | 22B1 | 2e | 22XX |
| - | 22B2 | 2e | 22XX |
| - | 22BB | 2e | 22XX |
| - | 22B? | 2e | 22XX |

| 22?X | | e | sub-e |
|---|---|---|---|
| - | 22?0 | 2e | 22XX |
| - | 22?1 | 2e | 22XX |
| - | 22?2 | 2e | 22XX |
| - | 22?B | 2e | 22XX |
| - | 22?? | 2e | 22XX |

2BXX

| 2B0X | | e | sub-e |
|---|---|---|---|
| - | 2B00 | Be | 0BXX |
| - | 2B01 | Be | 0BXX |
| - | 2B02 | Be | 0BXX |
| - | 2B0B | Be | 0BXX |
| - | 2B0? | Be | 0BXX |

| 2B1X | | e | sub-e |
|---|---|---|---|
| 2B10 | - | - | - |
| 2B11 | - | - | - |
| - | 2B12 | 2e | 12XX |
| - | 2B1B | Be | B1BX |
| 2B1? | - | - | - |

| 2B2X | | e | sub-e |
|---|---|---|---|
| - | 2B20 | 2e | 02XX |
| - | 2B21 | 2e | 12XX |
| - | 2B22 | 2e | 22XX |
| 2B2B | - | - | - |
| 2B2? | - | - | - |

| 2BBX | | e | sub-e |
|---|---|---|---|
| - | 2BB0 | Be | 0BXX |
| 2BB1 | - | - | - |
| 2BB2 | - | - | - |
| 2BBB | - | - | - |
| 2BB? | - | - | - |

| 2B?X | | e | sub-e |
|---|---|---|---|
| 2B?0 | - | - | - |
| 2B?1 | - | - | - |
| 2B?2 | - | - | - |
| 2B?B | - | - | - |
| 2B?? | - | - | - |

2?XX

| 2?0X | | e | sub-e |
|---|---|---|---|
| - | 2?00 | 2e | 0?2X |
| - | 2?01 | 2e | 0?2X |
| - | 2?02 | 2e | 0?2X |
| - | 2?0B | 2e | 0?2X |
| - | 2?0? | 2e | 0?2X |

| 2?1X | | e | sub-e |
|---|---|---|---|
| 2?10 | - | - | - |
| 2?11 | - | - | - |
| - | 2?12 | 2e | 12XX |
| - | 2?1B | Be | B1?2 |
| 2?1? | - | - | - |

| 2?2X | | e | sub-e |
|---|---|---|---|
| - | 2?20 | 2e | 02XX |
| - | 2?21 | 2e | 12XX |
| - | 2?22 | 2e | 22XX |
| 2?2B | - | - | - |
| 2?2? | - | - | - |

| 2?BX | | e | sub-e |
|---|---|---|---|
| - | 2?B0 | Be | 0BXX |
| 2?B1 | - | - | - |
| 2?B2 | - | - | - |
| 2?BB | - | - | - |
| 2?B? | - | - | - |

| 2??X | | e | sub-e |
|---|---|---|---|
| 2??0 | - | - | - |
| 2??1 | - | - | - |
| 2??2 | - | - | - |
| 2??B | - | - | - |
| 2??? | - | - | - |

### 9.3.4 BXXX

B0XX

| B00X | | e | sub-e |
|---|---|---|---|
| - | B000 | Be | 0BXX |
| - | B001 | Be | 0BXX |
| - | B002 | Be | 0BXX |
| - | B00B | Be | 0BXX |
| - | B00? | Be | 0BXX |

| B01X | | e | sub-e |
|---|---|---|---|
| - | B010 | Be | 0BXX |
| - | B011 | Be | 0BXX |
| - | B012 | Be | 0BXX |
| - | B01B | Be | 0BXX |
| - | B01? | Be | 0BXX |

| B02X | | e | sub-e |
|---|---|---|---|
| - | B020 | Be | 0BXX |
| - | B021 | Be | 0BXX |
| - | B022 | Be | 0BXX |
| - | B02B | Be | 0BXX |
| - | B02? | Be | 0BXX |

| B0BX | | e | sub-e |
|---|---|---|---|
| - | B0B0 | Be | 0BXX |
| - | B0B1 | Be | 0BXX |
| - | B0B2 | Be | 0BXX |
| - | B0BB | Be | 0BXX |
| - | B0B? | Be | 0BXX |

| B0?X | | e | sub-e |
|---|---|---|---|
| - | B0?0 | Be | 0BXX |
| - | B0?1 | Be | 0BXX |
| - | B0?2 | Be | 0BXX |
| - | B0?B | Be | 0BXX |
| - | B0?? | Be | 0BXX |

B1XX

| B10X | | e | sub-e |
|---|---|---|---|
| B100 | - | - | - |
| B101 | - | - | - |
| - | B102 | 2e | 02XX |
| - | B10B | Be | 0BXX |
| B10? | - | - | - |

| B11X | | e | sub-e |
|---|---|---|---|
| - | B110 | 1e | 011X |
| - | B111 | 1e | 111X |
| - | B112 | 2e | 12XX |
| B11B | - | - | - |
| B11? | - | - | - |

| B12X | | e | sub-e |
|---|---|---|---|
| - | B120 | 2e | 12XX |
| - | B121 | 2e | 12XX |
| - | B12 | 2e | 12XX |
| - | B12 | 2e | 12XX |
| - | B12 | 2e | 12XX |

| B1BX | | e | sub-e |
|---|---|---|---|
| - | B1B0 | Be | B1BX |
| - | B1B1 | Be | B1BX |
| - | B1B2 | Be | B1BX |
| - | B1BB | Be | B1BX |
| - | B1B? | Be | B1BX |

| B1?X | | e | sub-e |
|---|---|---|---|
| B1?0 | - | - | - |
| B1?1 | - | - | - |
| - | B1?2 | Be | B1?2 |
| B1?B | - | - | - |
| B1?? | - | - | - |

B2XX

| B20X | | e | sub-e |
|---|---|---|---|
| - | B200 | 2e | 02XX |
| - | B201 | 2e | 02XX |
| - | B202 | 2e | 02XX |
| - | B20B | 2e | 02XX |
| - | B20? | 2e | 02XX |

| B21X | | e | sub-e |
|---|---|---|---|
| - | B210 | 2e | 12XX |
| - | B211 | 2e | 12XX |
| - | B212 | 2e | 12XX |
| - | B21B | 2e | 12XX |
| - | B21? | 2e | 12XX |

| B22X | | e | sub-e |
|---|---|---|---|
| - | B22 | 2e | 22XX |
| - | B22 | 2e | 22XX |
| - | B22 | 2e | 22XX |
| - | B22 | 2e | 22XX |
| - | B22 | 2e | 22XX |

| B2BX | | e | sub-e |
|---|---|---|---|
| - | B2B0 | Be | 0BXX |
| B2B1 | - | - | - |
| B2B2 | - | - | - |
| B2BB | - | - | - |
| B2B? | - | - | - |

| B2?X | | e | sub-e |
|---|---|---|---|
| - | B2?0 | 2e | 0?2X |
| B2? | - | - | - |
| B2? | - | - | - |
| B2? | - | - | - |
| B2? | - | - | - |

BBXX

| BB0X | | e | sub-e |
|---|---|---|---|
| - | BB00 | Be | 0BXX |
| - | BB01 | Be | 0BXX |
| - | BB02 | Be | 0BXX |
| - | BB0B | Be | 0BXX |
| - | BB0? | Be | 0BXX |

| BB1X | | e | sub-e |
|---|---|---|---|
| BB10 | - | - | - |
| BB11 | - | - | - |
| - | BB12 | 2e | 12XX |
| - | BB1B | Be | B1BX |
| BB1? | - | - | - |

| BB2X | | e | sub-e |
|---|---|---|---|
| - | BB20 | 2e | 02XX |
| - | BB21 | 2e | 12XX |
| - | BB22 | 2e | 22XX |
| BB2B | - | - | - |
| BB2? | - | - | - |

| BBBX | | e | sub-e |
|---|---|---|---|
| - | BBB0 | Be | 0BXX |
| BBB1 | - | - | - |
| BBB2 | - | - | - |
| BBBB | - | - | - |
| BBB? | - | - | - |

| BB?X | | e | sub-e |
|---|---|---|---|
| BB?0 | - | - | - |
| BB?1 | - | - | - |
| BB?2 | - | - | - |
| BB?B | - | - | - |
| BB?? | - | - | - |

B?XX

| B?0X | | e | sub-e |
|---|---|---|---|
| B?00 | - | - | - |
| B?01 | - | - | - |
| - | B?02 | 2e | 02XX |
| - | B?0B | Be | 0BXX |
| B?0? | - | - | - |

| B?1X | | e | sub-e |
|---|---|---|---|
| B?10 | - | - | - |
| B?11 | - | - | - |
| - | B?12 | 2e | 12XX |
| B?1B | - | - | - |
| B?1? | - | - | - |

| B?2X | | e | sub-e |
|---|---|---|---|
| - | B?20 | 2e | 02XX |
| - | B?21 | 2e | 12XX |
| - | B?22 | 2e | 22XX |
| B?2B | - | - | - |
| B?2? | - | - | - |

| B?BX | | e | sub-e |
|---|---|---|---|
| - | B?B0 | Be | 0BXX |
| B?B1 | - | - | - |
| B?B2 | - | - | - |
| B?BB | - | - | - |
| B?B? | - | - | - |

| B??X | | e | sub-e |
|---|---|---|---|
| B??0 | - | - | - |
| B??1 | - | - | - |
| B??2 | - | - | - |
| B??B | - | - | - |
| B??? | - | - | - |

### 9.3.5 ?XXX

?0XX

| ?00X | | e | sub-e |
|---|---|---|---|
| ?00 | - | - | - |
| ?00 | - | - | - |
| - | ?002 | 2e | 02XX |
| - | ?00B | Be | 0BXX |
| ?00 | - | - | - |

| ?01X | | e | sub-e |
|---|---|---|---|
| - | ?010 | 1e | 010X |
| ?011 | - | - | - |
| - | ?012 | 2e | 12XX |
| ?01B | - | - | - |
| ?01? | - | - | - |

| ?02X | | e | sub-e |
|---|---|---|---|
| - | ?020 | 2e | 02XX |
| - | ?021 | 2e | 02XX |
| - | ?022 | 2e | 02XX |
| - | ?02B | 2e | 02XX |
| - | ?02? | 2e | 02XX |

| ?0BX | | e | sub-e |
|---|---|---|---|
| - | ?0B0 | Be | 0BXX |
| - | ?0B1 | Be | 0BXX |
| - | ?0B2 | Be | 0BXX |
| - | ?0BB | Be | 0BXX |
| - | ?0B? | Be | 0BXX |

| ?0?X | | e | sub-e |
|---|---|---|---|
| ?0?0 | - | - | - |
| ?0?1 | - | - | - |
| - | ?0?2 | 2e | 0?2X |
| ?0?B | - | - | - |
| ?0?? | - | - | - |

?1XX

| ?10X | | e | sub-e |
|---|---|---|---|
| ?100 | - | - | - |
| ?101 | - | - | - |
| - | ?102 | 2e | 02XX |
| - | ?10B | Be | 0BXX |
| ?10? | - | - | - |

| ?11X | | e | sub-e |
|---|---|---|---|
| - | ?110 | 1e | 011X |
| - | ?111 | 1e | 111X |
| - | ?112 | 2e | 12XX |
| ?11B | - | - | - |
| ?11? | - | - | - |

| ?12X | | e | sub-e |
|---|---|---|---|
| - | ?120 | 2e | 12XX |
| - | ?121 | 2e | 12XX |
| - | ?122 | 2e | 12XX |
| - | ?12B | 2e | 12XX |
| - | ?12? | 2e | 12XX |

| ?1BX | | e | sub-e |
|---|---|---|---|
| - | ?1B0 | Be | 0BXX |
| ?1B1 | - | - | - |
| ?1B2 | - | - | - |
| ?1BB | - | - | - |
| ?1B? | - | - | - |

| ?1?X | | e | sub-e |
|---|---|---|---|
| ?1?0 | - | - | - |
| ?1?1 | - | - | - |
| ?1?2 | - | - | - |
| ?1?B | - | - | - |
| ?1?? | - | - | - |

?2XX

| ?20X | | e | sub-e |
|---|---|---|---|
| - | ?200 | 2e | 02XX |
| - | ?201 | 2e | 02XX |
| - | ?202 | 2e | 02XX |
| - | ?20B | 2e | 02XX |
| - | ?20? | 2e | 02XX |

| ?21X | | e | sub-e |
|---|---|---|---|
| - | ?210 | 2e | 12XX |
| - | ?211 | 2e | 12XX |
| - | ?212 | 2e | 12XX |
| - | ?21B | 2e | 12XX |
| - | ?21? | 2e | 12XX |

| ?22X | | e | sub-e |
|---|---|---|---|
| - | ?220 | 2e | 22XX |
| - | ?221 | 2e | 22XX |
| - | ?222 | 2e | 22XX |
| - | ?22B | 2e | 22XX |
| - | ?22? | 2e | 22XX |

| ?2BX | | e | sub-e |
|---|---|---|---|
| - | ?2B0 | Be | 0BXX |
| ?2B1 | - | - | - |
| ?2B2 | - | - | - |
| ?2BB | - | - | - |
| ?2B? | - | - | - |

| ?2?X | | e | sub-e |
|---|---|---|---|
| - | ?2?0 | 2e | 0?2X |
| ?2?1 | - | - | - |
| ?2?2 | - | - | - |
| ?2?B | - | - | - |
| ?2?? | - | - | - |

?BXX

| ?B0X | | e | sub-e |
|---|---|---|---|
| - | ?B00 | Be | 0BXX |
| - | ?B01 | Be | 0BXX |
| - | ?B02 | Be | 0BXX |
| - | ?B0B | Be | 0BXX |
| - | ?B0? | Be | 0BXX |

| ?B1X | | e | sub-e |
|---|---|---|---|
| ?B10 | - | - | - |
| ?B11 | - | - | - |
| - | ?B12 | 2e | 12XX |
| - | ?B1B | Be | B1BX |
| ?B1 | - | - | - |

| ?B2X | | e | sub-e |
|---|---|---|---|
| - | ?B20 | 2e | 02XX |
| - | ?B21 | 2e | 12XX |
| - | ?B22 | 2e | 22XX |
| ?B2B | - | - | - |
| ?B2? | - | - | - |

| ?BBX | | e | sub-e |
|---|---|---|---|
| - | ?BB0 | Be | 0BXX |
| ?BB1 | - | - | - |
| ?BB2 | - | - | - |
| ?BBB | - | - | - |
| ?BB? | - | - | - |

| ?B?X | | e | sub-e |
|---|---|---|---|
| ?B?0 | - | - | - |
| ?B?1 | - | - | - |
| ?B?2 | - | - | - |
| ?B?B | - | - | - |
| ?B?? | - | - | - |

??XX

| ??0X | | e | sub-e |
|---|---|---|---|
| ??00 | - | - | - |
| ??01 | - | - | - |
| - | ??02 | 2e | 02XX |
| - | ??0B | Be | 0BXX |
| ??0? | - | - | - |

| ??1X | | e | sub-e |
|---|---|---|---|
| ??10 | - | - | - |
| ??11 | - | - | - |
| - | ??12 | 2e | 12XX |
| ??1B | - | - | - |
| ??1? | - | - | - |

| ??2X | | e | sub-e |
|---|---|---|---|
| - | ??20 | 2e | 02XX |
| - | ??21 | 2e | 12XX |
| - | ??22 | 2e | 22XX |
| ??2B | - | - | - |
| ??2? | - | - | - |

| ??BX | | e | sub-e |
|---|---|---|---|
| - | ??B0 | Be | 0BXX |
| ??B1 | - | - | - |
| ??B2 | - | - | - |
| ??BB | - | - | - |
| ??B? | - | - | - |

| ???X | | e | sub-e |
|---|---|---|---|
| ???0 | - | - | - |
| ???1 | - | - | - |
| ???2 | - | - | - |
| ???B | - | - | - |
| ???? | - | - | - |