MINESWEEPER ON GRAPHS

SHAHAR GOLAN

Abstract. Minesweeper is a popular single player game. It has been shown that the Minesweeper consistency problem is NP-complete and the Minesweeper counting problem is #P-complete. We present a polynomial algorithm for solving these problems for minesweeper graphs with bounded treewidth.

1. Introduction

Minesweeper is a popular single player game, distributed with the Microsoft Windows operating system. It consists of a grid of covered cells (see Figure 1), some of which are *mined cells*, i.e., contain mines. The cells not containing mines are *free cells*. Each free cell contains information regarding the number of its mined neighbors (where cells adjacent diagonally are also considered as neighbors). At each move, the player may uncover a cell. If the cell is mined, the game is lost; if it is free, the information contained in it is revealed, and the player may use it. The goal of the game is to uncover all free cells, leaving all mined cells covered (see Figure 2). In some versions of the game, the total number of mines is given.

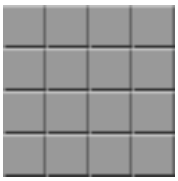

Figure 1. The initial state

Various approaches were examined for planning a strategy for maximizing the probability of winning Minesweeper. In [12], software is provided, enabling one to add his strategy (as a Java program) and test its performance on a predefined benchmark. Several basic strategies are also provided in [12]. One strategy regards the problem as a constraint satisfaction problem. A more sophisticated version of this strategy (devised independently) was examined in [15]. In [13] and [14], genetic algorithms were used in order to solve the problem. A learning

E-mail address: golansha@cs.bgu.ac.il.

(a) A lost game (b) A winning state

FIGURE 2. Ending states of Minesweeper

agent was introduced in [3]. There have been also some attempts to use neural networks [1].

The configuration given to the player before each move, consisting of a grid with some uncovered cells containing known information, will be referred to as a *Minesweeper grid* (henceforth MS grid). In such a grid there might be covered cells that can be positively inferred as free. Such cells are *safe*. A rational player will typically uncover safe cells whenever there are such, and guess only when there are none. An *assignment* $\mathcal{A}$ is a function from the Minesweeper grid's cells to the set {free, mined}. A *legal assignment* is an assignment such that each uncovered cell is assigned the value "free" and contains the correct value (i.e., exactly the number of mined neighbors it has). A covered cell is safe if and only if all legal assignments give it the value "free". Thus, an important part of any strategy is determining whether there exists a legal assignment where a given cell is mined.

The *Minesweeper consistency problem* (or simply the *Minesweeper problem*) is: Given a Minesweeper grid, does there exist a legal assignment for this grid? In the *Minesweeper counting problem* (called #Minesweeper in [11]) the required output is the number of legal assignments to the given MS grid (see Figure 3). We define the *Minesweeper constrained counting problem* as similar to the Minesweeper counting problem. In addition to the grid and labels, the input includes also the total number of mined cells. The consistency problem was proved to be NP-complete [7], and the counting problem – to be #P-complete [11].

An efficient solution for the Minesweeper consistency and counting problems can be used in order to solve other constrain satisfaction problems. For example, in [2] and [6] the problem of finding an equal moments division of a set is reduced to a constraint satisfaction problem similar to the minesweeper counting problem.

Several variations of the Minesweeper game were defined and analyzed. One way of generalizing Minesweeper is to discard the requirement that the board must be a grid, and consider Minesweeper on general graphs. An *MS graph* is the analogue of an MS grid for a

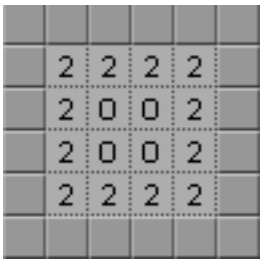

FIGURE 3. In this setting the solution for the counting problem is 1

graph. Thus, in an MS graph, each free vertex contains the information regarding the number of its mined neighbors. In [5] it was proved that Minesweeper on graphs with vertex degrees not exceeding 3 is already NP-complete. It is easy to see that the problem is polynomial if all vertices are of degree not exceeding 2. Infinite grid versions were suggested in [10] and [8].

In this paper we examine the consistency, counting and constrained counting problems for Minesweeper on trees and on graphs of bounded treewidth. We provide polynomial algorithms for these problems and prove their correctness.

In Section 2 we present the main results. Section 3 introduces notations that will be used in the rest of the paper. Section 4 contains the proofs of the results regarding Minesweeper on trees. In Section 5 we examine graphs with bounded treewidth and provide the relevant proofs.

2. The main results

Our basic result concerns the complexity of the consistency problem for trees.

Theorem 2.1. *The consistency problem can be solved for trees in linear time.*

We now strengthen the theorem in three respects.

First, we replace the consistency problem by the counting problem.

Theorem 2.2. *The counting problem can be solved for trees in* $O(n^2 \log n)$ *time.*

Next, we replace in Theorem 2.2 the counting problem by the constrained counting problem.

Theorem 2.3. *The constrained counting problem on trees can be solved in* $O(n^3 \log n)$ *time.*

The polynomiality of the constrained counting problem implies that of the counting problem, which implies in turn that of the consistency problem. However, inferring Theorem 2.2 from Theorem 2.3 (and, similarly, Theorem 2.1 from Theorem 2.2), we would get worse runtimes.

Finally, we deal with the same problem for graphs with bounded treewidth (for the full definition, see Section 5).

Theorem 2.4. *The constrained counting problem on graphs with bounded treewidth can be solved in polynomial time.*

The runtimes of our algorithm for the counting problem for graphs of treewidth k is $O(n^{k+2}\log n)$. This implies (see Proposition 5.2 below) that the constrained counting problem runs in $O(n^{k+3}\log n)$ time. Note that our results for trees in Theorem 2.2 and 2.3 give better runtimes than those implied by our Theorem 2.4.

3. Definitions and notations

In both the consistency problem and the counting problem, the input is an MS graph $G = (V, E)$, with $|V| = n$. Denote by $label(v)$ the initial information we have regarding the contents of v. Thus, $label(v)$ is a number if v is uncovered and $'?'$ otherwise. (From now on we will replace $'?'$ by ? unless a confusion may arise.) In fact, for the algorithm it will be convenient to generalize the problem, letting the possible initial information regarding the contents of each vertex be also that it is mined (denoted by $*$) or free (denoted by $\square$). Thus, $label(v)$ belongs to $\mathbb{N}$ if v is uncovered, and belongs to $\{?, *, \square\}$ otherwise. We shall let $label(v)$ assume the value -1 also (which value indicates that the graph admits no legal assignment).

Throughout this paper we use several notations regarding trees. Let T be a tree. It will be convenient to set an arbitrary vertex as the root of the tree. Denote this vertex by $\mathrm{root}(T)$. We define the corresponding parent-child relation, and denote the set of children of a vertex v by $C(v)$, and the parent of v by $p(v)$. For a label $\sigma \in \{?, *, \square\}$, put:

$$C_\sigma(v) = \{u \in C(v) \mid label(u) = \sigma\}.$$

For the consistency problem, define a function CMS from the set of all MS graphs to $\{0, 1\}$, where $CMS(G) = 1$ if there exists a legal assignment for all the covered cells in G and $CMS(G) = 0$ otherwise.

For the counting problem, we define a function $\#CMS$ from the collection of all MS trees to $\mathbb{N}$ by:

$$\#CMS(T) = |\{\mathcal{A} \ : \ \mathcal{A} \text{ is a legal assignment for } T\}|.$$

We define a similar function, $\#CC$, for the constrained counting problem. For an MS graph G and an integer x, let $\#CC(G, k)$ be the

number of legal assignments for G, in which exactly k cells are mined. For an MS tree T, denote the generating polynomial of the sequence $(\#CC(T,k))_{k=0}^{|V(T)|}$ by $P(T)$:

$$P(T) = \sum_{k=0}^{|V(T)|} \#CC(T,k)x^k.$$

For a vertex v, denote the subtree rooted at v by $T(v)$. In our algorithms we use a recursive approach. In several cases we need to consider modified subtrees. The following notations regard such modifications.

When $label(v) = ?$, we denote by $T_\square(v)$ the MS subtree rooted at v, where $label(v)$ is replaced by $\square$, and by $T_*(v)$ the analogous MS subtree for $*$. When $label(v) \in \mathbb{N} \cup \{\square\}$, we put $T_\square(v) = T(v)$ and take $T_*(v)$ as some inconsistent MS tree. When $label(v) = *$, we put $T_*(v) = T(v)$ and take $T_\square(v)$ as some inconsistent MS tree.

When $label(v)$ is a number, we denote by $T_-(v)$ the MS subtree rooted at v, with $label(v)$ reduced by 1. When $label(v)$ is not a number, we set $T_-(v) = T(v)$. This notation will be useful, for example, when $label(p(v)) = *$, and we want to inspect $T(v)$ independently.

For a vertex v and $\tau \in \{-, \square, *\}$, put $P(v) = P(T(v))$ and $P_\tau(v) = P(T_\sigma(v))$.

4. Minesweeper on trees

In this section we prove Theorems 2.1, 2.2 and 2.3. In fact, Theorem 2.3 contains its predecessors, except for the time analysis part. Thus, we first prove Theorem 2.3, and then comment on the reduction in time when dealing with the simpler cases.

Proof of Theorem 2.3:

It will be convenient to calculate $P(T)$, which gives simultaneously, for every number k, the number of legal assignments for T, in which exactly k vertices are mined. Our solution is recursive, and in the process we shall calculate for each vertex v the polynomial $P(v)$ (and as a by-product $P_-(v)$, $P_\square(v)$ and $P_*(v)$ as well; note that, in most cases, these polynomials are either $P(v)$ or 0).

Table 1 lists the polynomials $P(v)$, $P_-(v)$, $P_\square(v)$ and $P_*(v)$, in case v is a leaf.

Put $v = \text{root}(T)$, and assume that, for every child c of v, we are given $P(c)$, $P_-(c)$, $P_\square(c)$ and $P_*(c)$. Note that, since there are at most $n = |V(T)|$ mines, we have $\deg P(c) \leq \deg P(v) \leq n$. Distinguish between several cases:

label(v) \ polynomial	$P(v)$	$P_{-}(v)$	$P_{\square}(v)$	$P_{*}(v)$
$\square$	1	1	1	0
$*$	x	x	0	x
?	$1+x$	$1+x$	1	x
0	1	0	1	0
1	0	1	0	0
$2, 3, ...$	0	0	0	0

TABLE 1. $P(v)$ for a leaf v

Case i) $label(v) = \square$.
The assignments of the subtrees rooted at the vertices in $C(v)$ are independent of one another. We verify routinely that:

$$P(v) = \prod_{c \in C(v)} P(c).$$

The product of $|C(v)|$ polynomials, has degree less than n, and can be calculated in time $O(|C(v)|n \log n)$.
Case ii) $label(v) = *$.
Then:

$$P(v) = x \cdot \prod_{c \in C(v)} P_{-}(c).$$

Similarly to the previous case, we see that $P(v)$ can be calculated in $O(|C(v)|n \log n)$ time.
Case iii) $label(v) =?$.
We calculate $P_{\square}(v)$ and $P_{*}(v)$ according to Cases (i) and (ii), respectively. Now,

$$P(v) = P_{\square}(v) + P_{*}(v).$$

Case iv) $label(v) = k \in \mathbb{N}$.
Every legal assignment of $T(v)$ has exactly k vertices in $C(v)$ that are mined. Consequently:

$$P(v) = \sum_{\substack{C' \subseteq C(v) \\ |C'|=k}} \left(\prod_{c \in C'} P_{*}(c) \cdot \prod_{c \in C(v) \setminus C'} P_{\square}(c) \right)$$

The sum consists of $\binom{|C(v)|}{k}$ terms. To perform this calculation efficiently we use a generating function with two variables. For

a vertex c, define $\mathcal{P}(c) = P_{\square}(c) + P_*(c)y$. The coefficient of y^k in the product

$$\prod_{c \in C(v)} \mathcal{P}(c)$$

is exactly $P(v)$. Note that $P_{-}(v)$ is the coefficient of y^{k-1}. Replacing y by x^n we can reduce this problem to the multiplication of polynomials in one variable. Since the degree of the product is at most n^2, the product can be calculated in $O(|C(v)|n^2 \log n)$.

For a tree T we start by calculating P for the leaves of T. We continue iteratively calculating P for vertices whose children were already analyzed. It is obvious that every vertex is visited once, and is processed in polynomial time. To estimate the complexity of the algorithm, We simply charge the children of a node, v, for the calculation of $P(v)$. note that the ammortized work per child is at most $O(n^2 \log n)$.

□

Proof of Theorem 2.1: First, note that $CMS(v) = 1$ if and only if $P(v) \neq 0$. Thus, we may repeat the proof of Theorem 2.3, where each $P(v)$ is replaced by $CMS(v)$ (i.e., $CMS(T(v))$). Similarly, instead of each $P_\sigma(v)$ we need now $CMS_\sigma(v)$, defined by:

$$CMS_\sigma(v) = \begin{cases} 1, & P_\sigma(v) \neq 0, \\ 0, & P_\sigma(v) = 0, \end{cases} .$$

The calculation of $CMS(v)$, based on the values of $CMS(c)$ for $c \in C(v)$, is trivial in most cases and can be performed in $O(|C(v)|)$ time. We describe the necessary modifications only for Case (iv) in the proof of Theorem 2.3. In this case $CMS(v) = 1$ if and only if $CMS(c) = 1$ for all $c \in C(v)$, and $k \geq |C(v)| - |\{c \in C(v) \ : \ CMS_{\square}(v) = 1\}|$ and $k \leq |\{c \in C(v) \ : \ CMS_*(v) = 1\}|$. This check is done in $O(|C(v)|)$ time.

□

Proof of Theorem 2.2: First, note that $\#CMS(T) = (P(v))(1)$. The proof is analogous to that of Theorem 2.1. We describe the necessary modifications only for Case (iv) in the proof of Theorem 2.3. In this case we obtain $\#CMS(v)$ by multiplying $|C(v)|$ linear polynomials. This operation can be done in $O(|C(v)|n \log n)$ time. □

5. Minesweeper on graphs with bounded treewidth

Recall that a *tree decomposition* of a graph $G = (V, E)$ is a tree $T = (W, F)$, as follows: $W = \{W_1, \ldots, W_r\}$ is a family of (not necessarily distinct) subsets of V, such that:

(1) $\cup_{i=1}^{r} W_i = V$. That is, each graph vertex belongs to at least one tree node.
(2) For every edge (v, w) in the graph, there is a subset W_i that contains both v and w.
(3) If W_i and $W_{i'}$ both contain a vertex v, then all nodes W_l of the tree along the (unique) path between W_i and $W_{i'}$ contain v as well. In other words, the set of nodes of T containing a vertex v induces a subtree of T.

(See [9] for more details).

A tree decomposition of a graph is far from being unique; for example, each graph has a trivial tree decomposition, obtained by taking a tree with a single node, comprising all vertices of the graph.

The *width* of a tree decomposition (W, F) is $\max_{1\leq i\leq r}|W_i| - 1$. The *treewidth* of a graph G is the minimum width of all possible tree decompositions of G.

It will be convenient to set an arbitrary node of T as its root.

Now we return to the setup of Theorem 2.4. Let $T = (W, F)$ be a tree decomposition as above of an MS graph $G = (V, E)$. Denote $V(G) = \{v_1, \ldots, v_n\}$. For $1 \leq i \leq r$ denote

$$W_{i?} = W_i \cap \{v \in V \mid label(v) =?\}$$

and

$$W_{i\#} = W_i \cap \{v \in V \mid label(v) \in \mathbb{N}\}.$$

For $1 \leq i \leq r$ and an assignment $\mathcal{A} : W_{i?} \to \{\square, *\}$, denote by $P_{i,\mathrm{loc}}(\mathcal{A})$ the monomial $x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n}$, where a_j is 0 if $v_j \notin W_{i\#}$, and is the number of neighbors of v_j in $W_{i?}$, which are mined according to $\mathcal{A}$, otherwise. For such W_i and $\mathcal{A}$, let T_i be the set of nodes of the subtree of T rooted at W_i. Denote

$$T_{i?} = \bigcup_{W_{i'} \in T_i} W_{i'?}$$

and

$$T_{i\#} = \bigcup_{W_{i'} \in T_i} W_{i'\#}.$$

We define also polynomials $P_i(\mathcal{A})$, as follows. For each n-tuple $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{N}^n$, satisfying $a_j = 0$ for all j such that $v_j \notin T_{i\#}$, let $\#GCC_i(\mathcal{A}, \mathbf{a})$ be the number of extensions $\bar{\mathcal{A}}$ of $\mathcal{A}$ to an assignment from $T_{i?}$ to $\{\square, *\}$, such that:

- Each v_j in $W_{i\#}$ has (according to $\bar{\mathcal{A}}$) a_j mined neighbors in $T_{i?}$.
- Each v_j in $T_{i\#} \setminus W_{i\#}$ has in T_i exactly $label(v_j)$ mined neighbors.

It will be convenient to set $\#GCC_i(\mathcal{A}, \mathbf{a}) = 0$ for all other n-tuples $\mathbf{a}$. Put:

$$P_i(\mathcal{A}) = \sum_{\mathbf{a}\in\mathbb{N}^n} \#GCC_i(\mathcal{A}, \mathbf{a}) x_1^{a_1} x_2^{a_2} \dots x_n^{a_n}.$$

Example 5.1. *Consider the MS graph depicted in Figure 4. The tree in Figure 5 is a tree decomposition of this graph. In Table 2 we list the values of P for the decomposition nodes and assignments.*

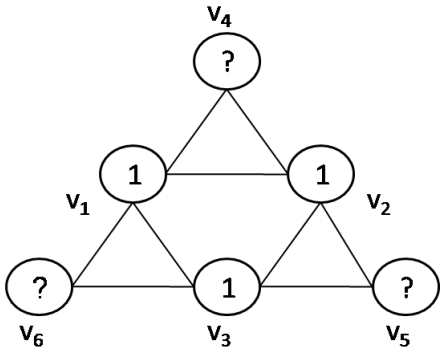

FIGURE 4. A graph G of treewidth 2

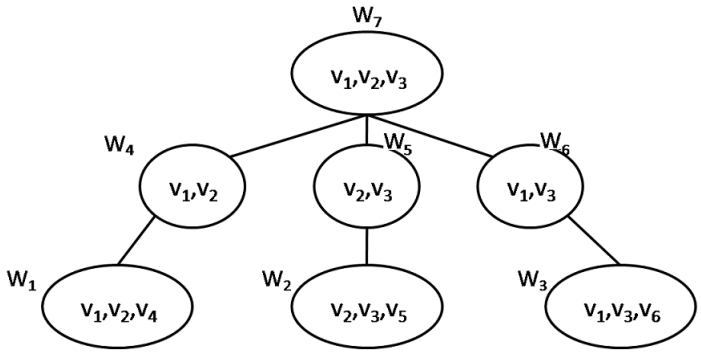

FIGURE 5. A tree decomposition of the graph in Figure 4

For the rest of the proof it will be convenient to assume, without loss of generality [9], that the tree decomposition is *nice*, namely every node $W_i \in W$ conforms to one of the following situations:

1) W_i is a leaf.
2) W_i has one child, $W_{i'}$, with $W_i = W_{i'} \setminus \{v\}$ for some $v \in V$.
3) W_i has one child, $W_{i'}$, with $W_i = W_{i'} \cup \{v\}$ for some $v \in V$.
4) W_i has two children $W_{i'}, W_{i''}$, and $W_i = W_{i'} = W_{i''}$.

In the first of these cases we have $P_i(\mathcal{A}) = P_{i,\mathrm{loc}}(\mathcal{A})$.

In the second case, assume, say, that $v = v_1$. Distinguish between two subcases:

i	$\mathcal{A}$	$P_i(\mathcal{A})$
1	$v_4 \to \square$	1
1	$v_4 \to *$	x_1x_2
2	$v_5 \to \square$	1
2	$v_5 \to *$	x_2x_3
3	$v_6 \to \square$	1
3	$v_6 \to *$	x_1x_3
4	$\emptyset$	$1 + x_1x_2$
5	$\emptyset$	$1 + x_2x_3$
6	$\emptyset$	$1 + x_1x_3$
7	$\emptyset$	$1+x_1x_2+x_2x_3+x_1x_3+x_1^2x_2x_3+$ $x_1x_2^2x_3 + x_1x_2x_3^2 + x_1^2x_2^2x_3^2$

TABLE 2. The polynomials $P_i(\mathcal{A})$ for the graph G

Subcase i) $label(v_1) =?$.

$$P_i(\mathcal{A}) = P_{i'}(\mathcal{A} \cup \{v_1 \to *\}) + P_{i'}(\mathcal{A} \cup \{v_1 \to \square\})$$

Subcase ii) $label(v_1) = s \in \mathbb{N}$.

Viewing $P_{i'}(\mathcal{A})$ as a polynomial in x_1, the coefficient of x_1^s in this polynomial is the polynomial $P_i(\mathcal{A})$ in the unknowns $x_2, \ldots, x_n$.

In the third case, denote by $\mathcal{A}'$ the assignment $\mathcal{A}$ reduced to $W_{i'?}$. We have

$$P_i(\mathcal{A}) = P_{i,\mathrm{loc}}(\mathcal{A})\frac{P_{i'}(\mathcal{A}')}{P_{i',\mathrm{loc}}(\mathcal{A}')}.$$

Finally, in the fourth case define $\mathcal{A}'$ and $\mathcal{A}''$ similarly. We have

$$P_i(\mathcal{A}) = P_{i,\mathrm{loc}}(\mathcal{A})\frac{P_{i'}(\mathcal{A}')}{P_{i',\mathrm{loc}}(\mathcal{A}')}\frac{P_{i''}(\mathcal{A}'')}{P_{i'',\mathrm{loc}}(\mathcal{A}'')}.$$

The next proposition shows how to use this method in order to solve the constrained counting problem.

Proposition 5.2. *If the counting problem on graphs with bounded treewidth can be solved in polynomial time, then the constrained problem on such graphs is also solvable in polynomial time.*

Proof. We can reduce the constrained counting problem to the counting problem as follows. Add a vertex v to the input graph G,connect v to

all vertices of G, and set $label(v)$ to be the required total number of mines. The number of solutions of the unconstrained problem for the modified graph is the same as the number of solutions of the original constrained problem. The treewidth of the modified graph is larger by at most 1 than the treewidth of G, since we can add the vertex v to every node of the tree decomposition of G and get a decomposition for the modified graph. Hence the original problem reduces to a counting problem on a graph with bounded treewidth. □

We now conclude the proof of Theorem 2.4.

Proof of Theorem 2.4 : Given a tree decomposition of width k, we calculate the polynomials $P_i(\mathcal{A})$ recursively as above. These are polynomials in at most $k+1$ variables, of degrees at most n in each variable. Hence the computation of each $P_i(\mathcal{A})$ takes polynomial time. Since there are at most $2^{k+1} = O(1)$ possible assignments, and the number of nodes in the tree decomposition is linear in the size of the input graph, the runtime of the algorithm is polynomial. The solution to the counting problem can be easily extracted from the polynomials of the root of the decomposition. By Proposition 5.2, the constrained counting problem can also be solved in polynomial time.

References

[1] J. Bellaiche, *Minesweeper with reinforcement learning neural networks* (term paper), 1998.

[2] D. Berend and S. Golan, *Littlewood polynomials with high order zeros*, Math. Comp. **75**(2006), 1541–1552.

[3] L. Castillo and S. Wrobel, *Learning minesweeper with multirelational learning*, Proceedings of the 18th International Joint Conference on Artificial Intelligence (2003), 533-538.

[4] J. Erickson, *Lower bounds for linear satisfiability problems*, Chicago J. Theoret. Comput. Sci. **8**(1999).

[5] M. Heinrich, *More properties for NP-complete Minesweeper graphs.* **http://heinrichmartin.com/mw/moreMWG.pdf.**

[6] S. Golan, *Equal moments division of a set*, Math. Comp. **77**(2008), 1695–1712.

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

[8] R. Kaye, *Infinite versions of minesweeper are Turing complete.* **http://for.mat.bham.ac.uk/R.W.Kaye/minesw/infmsw.pdf.**

[9] T. Kloks, *Treewidth. Computations and Approximations*, Lecture Notes in Computer Science **842**(1994) Springer-Verlag, Berlin.

[10] E. Mossel, *The Minesweeper game: percolation and complexity*, Combinatorics, Probability and Computing **11** n.5 (2002), 487–499.

[11] P. Nakov and Z. Wei, *MINESWEEPER, #MINESWEEPER.* **http://www.cs.berkeley.edu/∼zile/CS294-7-Nakov-Zile.pdf.**

[12] J. D. Ramsdell, **http://www.ccs.neu.edu/home/ramsdell/pgms/.**

[13] J. Rhee, *Evolving Strategies for the Minesweeper Game using Genetic Programming*, Genetic Algorithms and Genetic Programming at Stanford **6**(2000), 312-318, Stanford Bookstore, California.

[14] C. Quartetti, *Evolving a Program to Play the Game Minesweeper*, Genetic Algorithms and Genetic Programming at Stanford **6**(2000), 137-146, Stanford Bookstore, California.
[15] C. Studholme, **http://www.cs.utoronto.ca/∼cvs/minesweeper/.**

Department of Computer Science, Ben-Gurion University of the Negev, POB 653, Beer-Sheva 84105 Israel.

E-mail address: golansha@cs.bgu.ac.il