MINESWEEPER, #MINESWEEPER

Preslav Nakov, Zile Wei
{nakov,zile}@eecs.berkeley.edu

May 14, 2003

"Hence the easiest way to ensure you always win:
100-square board, 99 mines."
– Chris Mattern, Slashdot

Abstract

We address the problem of finding an optimal policy for an agent playing the game of minesweeper. The problem is solved *exactly*, within the reinforcement learning framework, using a modified value iteration. Although it is a *Partially Observable* Markov Decision Problem (MDP), we show that, when using an appropriate state space, it can be defined and solved as a fully observable MDP.

As has been shown by Kaye, checking whether a minesweeper configuration is non-contradictory is an NP-complete problem. We go a step further, as we define a corresponding counting problem, named #MINESWEEPER, and prove it is #P-complete. We show that computing both the state transition probability and the state value function requires finding the number of solutions of a system of *Pseudo-Boolean* equations (or *0-1 Programming*), or alternatively, counting the models of a CNF formula describing a minesweeper configuration. We provide a broad review of the existing approaches to exact and approximate model counting, and show none of them matches our requirements exactly.

Finally, we describe several ideas that cut the state space dramatically and allow us to solve the problem for boards as large as 4×4. We present the results for the optimal policy and compare it to a sophisticated probabilistic greedy one, which relies on the same model counting idea. We evaluate the impact of the *free-first-move* and provide a list of the best opening moves for boards of different sizes and with different number of mines, according to the optimal policy.

1 About minesweeper

Minesweeper is a simple game, whose popularity is due to a great extent to the fact that it is regularly included in Microsoft Windows since 1991 (as implemented by Robert Donner and Curt Johnson). Why is it interesting? Because it hides in itself one of the most important problems of the theoretical computer science: the question whether P = NP. *P vs. NP* is one of the seven *Millennium Prize Problems*, for each of which the *Clay Mathematics Institute* of Cambridge, Massachusetts offers the prize of $1,000,000 [2]. Its connection to minesweeper has been revealed by Kaye, who proved minesweeper is NP-complete [36], and is further discussed by Stewart in *Scientific American* [48].

Some interesting resources about Minesweeper on the Web include:

♢ *Richard Kaye's minesweeper pages.* Focused on the theory behind the game [35].

♢ *The Authoritative Minesweeper.* Tips, downloads, scores, cheats, and info for human players. A *world records* list is also supported [12].

♢ *Programmer's minesweeper.* Java interface that allows a programmer to implement a strategy for playing minesweeper and then observe how it performs. Contains several ready strategies [6].

♢ *Minesweeper strategies.* Tips and tricks for human players [4].

♢ *Minesweeper strategies & tactics.* Strategies and tactics for the human player [3].

2 SAT, k-SAT

We start with some definitions.

Literals are Boolean variables (with values from {0,1}) and their negations. A *Boolean formula* (or *propositional formula*) is an expression written using AND, OR, NOT, parentheses and variables.

A Boolean formula is in *Disjunctive Normal Form* (DNF) if it is a finite disjunction (logical OR) of a set of *DNF clauses.* A *DNF clause* is a conjunction (logical AND) of a finite set of literals.

A Boolean formula is in *Conjunctive Normal Form* (CNF) if it is a finite conjunction (logical AND) of a set of *CNF clauses.* A *CNF clause* is a disjunction (logical OR) of a finite set of literals.

Unless otherwise stated, below we will work with Boolean formulas (or just *formulas*) in CNF. We will use n to refer to the number of variables, and m − to the number of CNF clauses (or just *clauses*).

The Boolean formulas are in the heart of the first known NP-complete problem: SAT (Boolean SATisfiability) [24]. Given a Boolean formula, it asks whether there exists an assignment that would make it true. The problem remains NP-complete even when is restricted to CNF formulas with up to k variables per clause (a.k.a. k-CNF), provided that $k \geq 3$. Some particular cases can be solved in polynomial time: e.g. 2SAT, Horn clauses etc. It is interesting to note that DNF-SAT is also solved in polynomial time (with respect to the number of clauses, *not* the variables!): it is enough to check whether there exists a clause without contradictions: it should not contain x and $\neg x$ at the same time. SAT is an important problem for logic, circuit design, artificial intelligence, probability theory etc. A major source of information about SAT is the *SAT Live!* site [8]. Ready implementations of different solvers, tools, evaluation, benchmarks etc. can be obtained from *SATLIB − The Satisfiability*

Library [9].

3 Related Work

The most systematic study of minesweeper so far is due to Kaye. He tried to find a strategy such that no risk is taken when there is a square that can be safely uncovered and this led him to the following decision problem definition [36]:

MINESWEEPER: Given a particular minesweeper configuration determine whether it is consistent, which is whether it can have arisen from some pattern of mines.

In fact, he named the problem *minesweeper consistency problem*, but we prefer to use MINESWEEPER. Kaye studied the following strategy: To determine whether a background square is free we can mark it as a mine. If the resulting configuration is inconsistent (MINESWEEPER solution is NO), the square is safe to uncover.

It is easy to see that MINESWEEPER is a special case of SAT, and more precisely, 8-SAT, since no clause can contain more than 8 literals. Is it NP-complete however? This question is reasonable since the MINESWEEPER is a proper subset of 8-SAT: only formulas of some *special* forms are possible. Namely, all clauses can have either only positive or only negative literals. In addition, for each positive-only clause there exists exactly one negative-only one, which contains the same set of variables but all they are negated.

Kaye showed how different minesweeper configurations could be used to build logic circuits and thus, to simulate digital computers. For the purpose, he presented configurations that can be used as *wire*, *splitter* and *crossover*, as well as a set of *logic gates*: NOT, AND, OR and XOR. The presence of AND and NOT gates (or alternatively OR and NOT gates) proves that MINESWEEPER is NP-complete [36, 1]. This proof follows to a great extent the way Berlekamp and al. [19] proved that the John Conway's game of life [29] is Turing-complete. In addition, Kaye observed that the game of life is played on an infinite board, so he defined an infinite variant of minesweeper and then proved it is Turing-complete as well [37].

There was also some work on the analysis and implementation of real game playing strategies. Adamatzky [13] describes a simple cellular automaton, capable to populate a $n \times n$ board in time $\Omega(n)$, with each automaton cell having 27 states and 25 neighbors, including itself. Rhee [46], Quartetti [45] used genetic algorithms. There have been also some attempts to use neural networks (see the *term projects* [18], [33]).

Castillo and Wrobel address the game of minesweeper from a machine learning prospective. Using *Mio*, a general-purpose system for *Inductive Logic Programming* (*ILP*), they learned a strategy, which achieved a "better performance than that obtained on average by non-expert human players" [23]. In fact the rules learned by the system were for a board of a fixed small size: 8×8. The system learned four Prolog-style rules, exploiting very local information only. Three of the rules were virtually the same as the most trivial ones as implemented in the *equation strategy* of PGMS [6]. While the average human beginner wins 35% of the time, they achieved 52%. This rate rose to 60% when a very simple local probability, defined separately over each equation is employed to help guessing in case no rule is applicable, thus achieving the win rate of PGMS.

In fact, all these attempts, although interesting, are inadequate since solving MINESWEEPER requires *global* logical inference and model counting.

4 Pseudo-Boolean problems

Should we necessarily think of minesweeper in terms of Boolean formula satisfiability? The equations that arise there are of the form:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 = 5$$

All the coefficients of the c unknowns on the left-hand-side are 1, and the number k on the right-hand-side is an integer, $1 \leq k \leq 7$ (we do not allow 0 and 8, which are trivial), and in addition, $k < c$.

Converting this equation to CNF form requires putting positive-only clauses (e.g. $x_1 \vee x_3 \vee x_4 \vee x_6 \vee x_7$) and C_8^5 negative-only counterparts (e.g. $\bar{x_1} \vee \bar{x_3} \vee \bar{x_4} \vee \bar{x_6} \vee \bar{x_7}$), which is 112 in total. Do we really need to do this? Cant't we just look at MINESWEEPER as a *constraint satisfaction* problem of a special kind and try to solve it more efficiently?

Let us explore the options. First, note that SAT (suppose a CNF formula is given) can be written as an Integer Programming problem the following way:

$$\begin{aligned} &\text{minimize} && w \\ &\text{subject to: } w + \sum_{y_i \in c_j^+} x_i + \sum_{y_i \in c_j^-} (1 - x_j) \geq 1, j = 1, \cdots, m \\ &&& x_i \in 0, 1, i = 1, \cdots, n \\ &&& w > 0 \end{aligned} \tag{1}$$

where $x_i (1 \leq i \leq n)$ are the unknowns, $c_j (1 \leq j \leq m)$ are the CNF clauses (c_j^- and c_j^+ are the sets of the negative and the positive literals of the clause c_j), and w is a new variable, added to ensure there will be a feasible solution. The formula is unsatisfiable, if $w > 0$.

In the case of minesweeper, we can allow not only 1, but also other natural numbers on the right-hand-side of the equation. In addition, we will need to put two inequalities for each equation, but we will have no combinatorial explosion of clauses:

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \leq 5 \\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \geq 5 \end{aligned} \tag{2}$$

In fact, the reduction, although instructive, is not necessarily the best one. Note, that in SAT we do not have a function to optimize. In addition, in

MINESWEEPER we always have only equalities (for the local constraints), which means we need to solve a system of linear equations. In general this can be done in $\Theta(n^3)$ using Gaussian elimination. In the discrete case this is a problem of solving a system of Diophantine equations, which is an NP-complete problem. Note however, that our problem is more restricted than that: we want the values to be not just integers, but 0 or 1.

So, looking back at the Integer Programming problem with inequalities, we can see it is a special instance, namely a *pure 0-1 programming*, where the variable values are restricted to 0 and 1. It is natural to allow the inequalities to accept natural numbers other than 1 on the right-hand-side. If in addition we allow the coefficients to take integer values other than 0 and 1, we obtain the *Pseudo-Boolean* problem.

The problem can be solved by means of pure Boolean methods (e.g. a Pseudo-Boolean version of *relsat* [26], randomization [44], or DPLL techniques [14]) or using Integer Programming techniques (e.g. branch-and-bound and branch-and-cut [26], [15]]). In the latter case it would be more natural to name the problem *0-1 programming.* A good source of information about solving pseudo-Boolean problems is *PBLIB – the Pseudo-Boolean Library* [5], where can be found several useful links to sites, papers, people, implementations, tools etc.

5 Model counting

5.1 #P, #P-complete, #SAT, #k-SAT

It has been realized that there are problems, primarily in AI, that require not satisfiability checking but rather counting the number of models $\mu(F)$ of a CNF formula F. A broad overview and several examples can be found in [47]. Some of these include: computing the number of solutions of a constraint satisfaction problem, inference in Bayesian networks, finding the degree of belief of a propositional statement s with respect to a knowledge base B (computed as $\mu(B + \{s\})/\mu(B)$), estimating the utility of an approximate theory A with respect to the original one O (i.e. $\mu(O) - \mu(T)$), etc.

For the purpose of studying counting problems Valiant [49] introduced some new complexity classes: #P, #P-complete etc. Formally, a problem is in #P if there is a nondeterministic, polynomial-time Turing machine that, for each instance of the problem, has a number of accepting computations, exactly equal to the number of its distinct solutions. It is easy to see that a #P problem is necessarily at least as hard as the corresponding NP problem: once we know the number of solutions, we can just compare this number to zero and thus solve the NP problem. So, any #P problem, corresponding to an NP-complete problem, is NP-hard. In addition to #P, Valiant defined the class #P-complete. A problem is in #P-complete, if and only if it is in #P, and every other problem in #P can be reduced to it in polynomial time.

The problem of finding the number of models of a Boolean formula is known in general as #SAT. Just like SAT, #SAT has versions #2SAT, #3SAT, #k-SAT etc., where the maximum number of literals in a clause is limited by k. #SAT has been shown to be #P-complete in general [49] and thus, it is necessarily exponential in the worst case (unless the complexity hierarchy collapses). Note, that while 2SAT is solvable in polynomial time, #2SAT is #P-complete (and so is k-SAT, $k \geq 2$).

5.2 #MINESWEEPER

While Kaye [36] addressed the problem of finding sure squares only, this is not enough to play good, since most of the real game configurations will have no sure cases. So, we face a problem of reasoning under uncertainty. A reasonable heuristic would be to probe the square with the lowest probability to contain a mine, thus acting greedily. This probability can be estimated as the ratio of the number of different mine distributions in which a fixed background square contains a mine to the number of all different mine distributions consistent with the current configuration. So, we need to define the #MINESWEEPER problem:

#MINESWEEPER: Given a particular minesweeper configuration find the number of the possible mine distributions over the covered squares.

Theorem 5.1. #MINESWEEPER is #P-complete.

Proof. Specialization to #CIRCUIT-SAT.

#CIRCUIT-SAT is the basic #P-complete problem and is defined as follows: given a circuit, find the number of inputs that make it output TRUE. Using the Kaye's result that a minesweeper configuration can represent logical gates, wires, splitters and crossovers, it follows that any program that tries to solve #MINESWEEPER should be able to solve any logical circuit, i.e. solve #CIRCUIT-SAT.

□

The theorem above means we cannot hope to solve #MINESWEEPER in polynomial time unless all problems in #P-complete can be solved in polynomial time, i.e. unless P=NP. But we can try to approximate it.

5.2.1 Inclusion-exclusion principle

Before we present some of the most interesting approaches so far, it is interesting to say that the problem of model counting is a special case of a more general problem (again #P-complete): finding the probability of the union of n events from some probability space. More formally, let $A_1, A_2, ..., A_m$ be events from s probability space and we want to find $P(\bigcup A_i)$. This is given by the inclusion-exclusion formula:

$$P(\bigcup_{i=1}^{m} A_i) = \sum_{i=1}^{m} P(A_i) - \sum_{1 \leq i \leq j \leq m} P(A_i \cap A_j) + \sum_{1 \leq i \leq j \leq k \leq m} P(A_i \cap A_j \cap A_k) - \cdots + (-1)^{m-1} P(A_1 \cap A_2 \cdots \cap A_m)$$

This formula contains $2^m - 1$ terms, and thus the direct evaluation would take exponential time. In addition, it cannot be calculated exactly in the absence of any of these terms. So, here comes the natural question: how well it can be approximated? It is interesting to note, that this is an old question, first studied by Boole [21]. In fact, he was interested in approximating the size of the union of a family of sets, which is essentially an equivalent problem; we just have to divide by 2^m, since:

$$|\bigcup_{i=1}^{m} A_i| = \sum_{i=1}^{m} |A_i| - \sum_{1\leq i\leq j\leq m} |A_i \cap A_j| + \sum_{1\leq i\leq j\leq k\leq m} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{m-1} |A_1 \cap A_2 \cdots \cap A_m|$$

How do we go from finding the probability of the union of probabilistic events to #SAT? Let's assume a probability space $\{0,1\}^n$ under uniform distribution and let A_i be the event that the i-th clause ($1 \leq i \leq m$) in a DNF formula is satisfied. Then $|A_i|$ is the number of models of the i-th clause and $| \cup_{i=1}^{m} A_i|$ is the number of models of all clauses in the formula F, i.e. the number of models $\mu(F)$ of F.

But we are interested in CNF, not in DNF. While these forms are equivalent, it is not straightforward to see how the inclusion-exclusion principle can be applied to CNF. The solution comes easily though, if we revise the definition of A_i: now it will be the event that the i-th clause in the CNF formula is *falsified*. So, $|A_i|$ will be the number of *falsifiers* of the i-th clause, and $| \cup_{i=1}^{m} A_i|$ – the number of *falsifiers* of all CNF clauses in F, i.e. $(2^m - \mu(F))$.

Note, that the uniform distribution over $\{0,1\}^n$ for DNF (or equivalently for CNF) is in fact a special case. As we will see below, #SAT is much easier problem than finding the probability of the union of probabilistic events.

5.3 Approximation algorithms

5.3.1 Luby & Velickovic algorithm

Luby and Velickovic [41] try to approximate the proportion of truth assignments of the n variables that satisfy the DNF formula F, i.e. P(F) $= \mu(F)/2^n$. They propose a *deterministic* algorithm, which for any ϵ and δ ($\epsilon > 0, \delta > 0$) calculates an estimate Q, such that $(1-\delta)Pr(F) - \epsilon \leq Q \leq P(F) + \epsilon$. The running time of this algorithm is polynomial in nm, multiplied by the term:

$$\Big(\frac{\log(nm)}{\delta\epsilon}\Big)^{\frac{\log^2(m/\epsilon)}{\delta}}$$

If the maximum clause length is t, then for any $\beta(\beta > 0)$ the deterministic algorithm finds an approximation satisfying: $(1-\beta)P(F) \leq Q \leq (1+\beta)P(F)$. This time the algorithm is polynomial in nm, multiplied by:

$$\Big(\frac{\max(t, \log n)}{\beta}\Big)^{\frac{t^2}{\beta}}$$

5.3.2 Inclusion-exclusion algorithms

Linial and Nisan addressed the more general problem of finding the size of the union of a family of sets [38]. They showed that it could be approximated effectively by using only *part of* the intersections. In particular, if the sizes of all intersections of at most k distinct sets are known, and $k \geq \Omega(\sqrt{m})$, then the union size can be approximated within $e^{-\Omega(k/\sqrt{m})}$.

The bound above has been shown to be non-tight, and has been improved by Kahn&al. [34] to $e^{-\tilde{\Omega}(k/\sqrt{m})}$. This resulted in a polynomial time approximation algorithm, which is optimal in a sense that this bound is tight and in general, it is impossible to achieve a better approximation (regardless of the complexity). Note that this is only an approximation, and there can be an error even when $k = m - 1$.

5.4 Exact algorithms

5.4.1 Exponential algorithms

The obvious and naive implementation of exact model counting enumerates all the models and thus runs in time $O(m2^n)$. Can we do better? If we restrict the maximum number of literals contained in a single clause to r, then we can decrease the basis of the exponention to α_r, the unique positive root of the polynomial $y^r - y^{r-1} - ... - 1$. This gives an algorithm running in time $O(m\alpha_r^n)$. Note, that this is generally better than the naive approach since $\alpha_2 \approx 1.62$, $\alpha_3 \approx 1.84$, $\alpha_4 \approx 1.93$,..., but in the limit ($r \to \infty$) we still have $\alpha_r \to 2^n$. This algorithm, as described, has been proposed by Dubois [27]. A similar approach (with similar complexity) has been presented by Zhang [50]. Despite the improvement though, the complexity remains exponential (and there is no improvement in the limit).

5.4.2 Lozinskii algorithm

Iwama [30] used model counting together with formula falsification to solve SAT. He achieved a polynomial average running time $O(m^{c+1})$, where c is a constant such that $\ln c \geq \ln m - p^2 n$, where p is the probability that a particular literal appears in a particular clause in the target formula.

Lozinskii [40] introduced a similar algorithm but for #SAT. Under reasonable assumptions, it calculates the exact number of models in time $O(m^c n)$, where $c \in O(\log m)$.

5.4.3 CDP algorithm

Birnbaum and Lozinskii [20] proposed an algorithm, named *CDP* (*Counting by Davis-Putnam*), for counting the number of models of a Boolean CNF formula exactly, based on a modification of the classic Davis-Putnam procedure [25]. There are two essential changes: first, the pure literal rule is omitted (otherwise some solutions will be missed), and second, in case the formula is satisfied by setting values to only part of the variables, the remaining unset i variables can take all possible values and thus, the value 2^i is returned.

The *average* running time of CDP is $O(m^d n)$, where $d = \lceil -1/\log_2(1-p) \rceil$ and p is the probability that a particular literal appears in a particular clause in the target formula.

5.4.4 DDP algorithm

The *DDP* (*Decomposing Davis-Putnam*) algorithm has been proposed by Bayardo and Pehoushek [16]. Just like CDP [20], it is an extension of the Davis-Putnam (DP) procedure [25]. The idea is to identify groups of variables that are independent of each other, which is essentially a kind of divide and conquer approach: find the connected components of the graph, solve the problem for each of them independently and then combine the results. This is a fairly old idea, already exploited for SAT as well as for more general constraint satisfaction problems by Freuder and Quinn [28] and many other researchers thereafter. The idea of DDP is to apply it recursively, as opposed to just once at the beginning. During the recursive exploration of the graph DP assigns values to some of the variables and thus breaks it into separate connected components, which are in turn identified and exploited, which dramatically cuts the calculations.

The algorithm is implemented as an extension of version 2.00 of the *relsat* algorithm [17] and the source code is available on the Internet [7]. The *relsat* algorithm is an extension of the Davis-Putnam-Logman-Loveland (DPLL) method [39], and it dynamically learns *nogoods* (the only difference with DPLL), which can dramatically speed it up: in case the variables form a chain, SAT can be solved in quadratic time. For #SAT though one would need also to record the *goods* (this was not implemented and tested). In the case of DDP this would lead to quadratic complexity for solving #SAT even in case the variables form a tree. Anyway, in some hard cases (e.g. when $m/n \approx 4$) the complexity will be necessarily exponential. Performance evaluation tests show DDP is orders of magnitude faster than CDP both on random instances, as well as on benchmarks like the ones in SATLIB [9].

5.4.5 Inclusion-exclusion algorithms specialized to #SAT

Looking carefully at the general problem of finding the size of the union of a family of sets (see 5.2.1 above), Kahn&al. [34] found that in the special case of #SAT it is enough to require to know the number of models for every subset of up to $1 + \log n$ clauses. Moreover, this has been proved to fix the number of satisfied assignments for the whole formula, and thus allows the calculation of the exact number of models, not just approximate it. Note also, that this is dependent on the number of the *variables only*, and not on the number of the clauses!

The latter proof of Kahn&al. was only *existentional*, and they did not come up with an algorithm. It has been found by Melkman and Shimony [42].

The result has been extended thereafter and specialized to #k-SAT, $k \geq 2$, as Iwama and Matsuura showed that n can be changed to k and that only the clauses of some $fixed$ size matter (thus, we should not require to know the ones of smaller size). So, for #k-SAT knowing the number of models for all clauses of size *exactly* $2 + \lfloor \log k \rfloor$ has been proved to be both necessary and sufficient [30]. Unfortunately, the latter proof is only existentional and there is no practical algorithm developed yet.

5.5 What are we using?

Initially, we used the Boolean Constraint Solver library module [10], provided with SICStus Prolog [11]. It is an instance of the general Constraint Logic Programming scheme introduced in [32]. The internal representation of the Boolean functions is based on Boolean Decision Diagrams as described in [22] and it allows using linear equalities as constraints, which saved us from combinatorial explosions of the number of clauses needed. While we found it fairly fast for solving MINESWEEPER, it was not suitable for #MINESWEEPER: the only way to find the number of satisfying assignments was to enumerate them all, which was unacceptable. We then performed an extensive study of the literature on model counting and adopted another approach.

Currently, we use the implementation of DDP (see 5.4.4) as provided in version 2.00 of the *relsat* algorithm [7]. We apply the model counting for reasoning with incomplete information: when we cannot logically conclude that neither there is nor there is not a mine in a particular square x, we solve the #MINESWEEPER problem twice: once with a mine put in x and once for the original configuration, and then take their ratio.

We could also use an approximate inclusion-exclusion algorithm, as specialized to #SAT by Melkman and Shimony [42], or even the one of Iwama and Matsuura [31], if a practical implementation is found in the future. In addition, we think in the case of #MINESWEEPER it might be possible to improve the result of Iwama and Matsuura further because of the special CNF structure: all clauses are paired positive-negative, and for each one with only positive literals there is another one that contains the same variables but all negated. Looking from the Pseudo-Boolean perspective, the problem is still a proper subset of it: we do not need inequalities, and we do not have coefficients, other than 0 and 1.

Ideally, solving #MINESWEEPER would be done by means of model counting for Pseudo-Boolean formulas: e.g. a combination between the Boolean Constraint Solver library module of SICStus Prolog and the *relsat* model counter.

6 Minesweeper

Minesweeper is a simple game whose goal is to clear all squares on a board that don't contain mines. The basic elements of the game are: playing board, remaining mines counter, and a timer. The rules are the following:

♢ The player can uncover an unknown square by clicking on it. If it contains a mine, the game is lost.

♢ If the square is free, a number between 0 and 8 appears on it, indicating how many mines are there in its immediate neighbor squares.

♢ A square can be marked as a mine by right clicking on it.

♢ The game is won when all non-mine squares are uncovered.

Minesweeper is an imperfect information computer game. The player probes on a $n \times n$ map, and has to make decisions based on the current observations only, without knowing which squares contain mines and which are free. The only additional information provided is the total number of remaining mines. If the probed square contains a mine, the player loses the game, otherwise the number of mines in the adjacent squares (up to 8) appears in the probed square. In most computer implementations (and in our interpretation in particular) the player never dies on the first probe, but, as we will see below, this doesn't change the complexity of the problem substantially.

We assign a $\{0/1\}$ variable to each unknown square in the map: the value is 1, if it contains a mine, and 0 − otherwise. All variables must obey the constraints imposed by the numbers in the uncovered squares, as well as the global constraint on the remaining number of mines, namely:

$$\begin{aligned} \text{for each known square } i \ : \sum_{X_j \in N(i)} X_j &= O(i) \\ \sum_{X_j} X_j &= M \end{aligned} \tag{3}$$

Where $N(i)$ is the set of the neighbors (up to 8) of the known square i, $O(i)$ is the value of the uncovered square i and M is total number of mines in the map. An example follows:

X_1	X_2	1
X_3	X_4	2
1	2	m

The following equations correspond to the configuration above:

$$\begin{aligned} X_2 + X_4 &= 1 \\ X_3 + X_4 &= 1 \\ X_1 + X_2 + X_3 + X_4 &= M \end{aligned} \tag{4}$$

When $M = 1$ or $M = 3$, there is a unique solution, but in case $M = 2$, there are two: $X_2 = X_3 = 1$ and $X_1 = X_4 = 1$.

Let $\mathcal{X}$ be the set of all solutions of (3). Then we can easily compute for each square the probability that it contains a mine, as a simple ratio:

$$P(X_i = 1) = \frac{|\{X : X \in \mathcal{X}, X(i) = 1\}|}{|\mathcal{X}|}$$

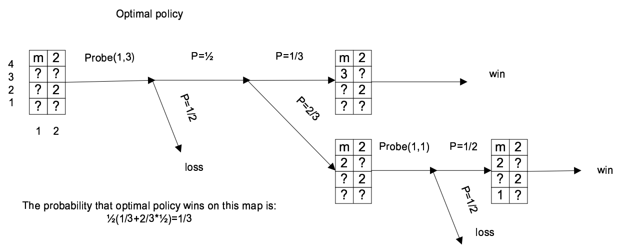

Figure 1: Optimal policy on 2×4 map with 2 mines

So, when $M = 2$ we have $P(X_1 = 1) = 1/2$. In order for a square to be a mine or free for sure, its value in all solutions should be 1 or 0, respectively.

Once we have computed the probability of containing a mine for each square, we can come up with a fairly simple greedy policy that always tries the safest square. We will show, this is not always optimal. Consider the 2×4 map with 2 mines, shown on figure (1). There are 5 unknown squares, 2 probed free ones and one sure mine. (The mine was not probed, of course, but was discovered as a result of logical inference.) $P(X_{1,1} = 1) = P(X_{1,2} = 1) = P(X_{2,1} = 1) = 1/3$, but $P(X_{1,3} = 1) = P(X_{2,3}) = 1/2$. Although (1,1), (1,2) and (2,1) are safer than (1,3) and (2,3), the *optimal* policy probes (1,3).

Let's try to analyze the probability that the optimal policy will win on this map. At the first step, probing at (1,3) will succeed with probability 1/2. There are two possible outcomes after a successful probe at (1,3). If the square (1,3) contains 3 (this will happen with probability 1/3), then the other squares containing a mine will be (1,2) and (2,3), and so the game will be won. If (1,3) contains 2 (which has a probability 2/3), the square (1,2) is surely not a mine and (2,3) is surely a mine. Hence, the agent has to guess the position of the last mine by choosing between (1,1) and (2,1) only, and thus will succeed with probability 1/2. So the total probability that the optimal policy will win at this configuration is $1/2 \times (1/3 + 1/2 \times 2/3) = 1/3$.

If the agent follows the greedy policy though, then he has to choose between (1,2), (1,1) and (2,1), which look equally good. Suppose the agent probes (1,2). Then he will succeed with probability 2/3. But in the outcome he has to make two guesses, between (1,3) and (2,3), and between (1,1) and (2,1). Hence it can succeed with probability 1/4. If the agent probes (1,1) or (2,1), and tries (1,3), which is from the optimal policy, the possibility that it can succeed is at most 1/2. So the expected probability that the greedy policy wins at this configuration is $1/3(2/3 \times 1/4 + 2/3 \times 1/2 + 2/3 \times 1/2) = 2/9$, which is less than for the optimal policy.

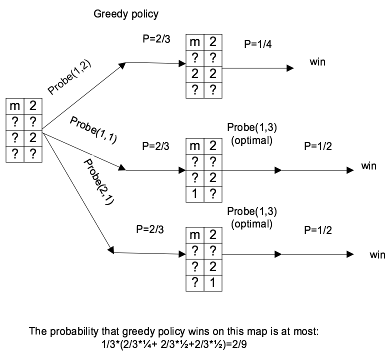

Figure 2: Greedy policy on 2×4 map with 2 mines

7 Minesweeper in Reinforcement Learning

The minesweeper problem can be fully interpreted in the reinforcement learning framework. We can easily define the five-tuple for MDP in minesweeper.

State space. The state space will consist of all valid configurations. Any square could be either known or unknown. When it's known, it will show the number of mines in its (up to 8) neighbors. So the maximum number of states for a square is 10: the numbers $0, 1, ..., 8$, and *unknown*. If we take the boundary conditions into consideration, given a $W \times H$ minefield, where W is the width and H is the height, then the number of all possible configurations is at most $5^4 \times 7^{2W+2H-4} \times 10^{(W-2)(H-2)}$. However, most of the configurations are invalid: they either contradict to themselves on the known square or contradict to the global constraint (the total number of mines), which means MINESWEEPER is unsatisfiable. Two examples follow:

2	2	2
?	?	?

1	1	1
?	?	?

Both configurations are invalid for a 3×2 map with 2 mines. In fact, the valid maps is quite limited. For example, on a 4×4 map with 4 mines the total number of configurations is $5^4 \times 7^8 \times 10^4 = 3.6 \times 10^{13}$, while the valid ones are only 3.2×10^6. Thus, just one out of 10 million configurations is valid! The larger the size of the board, the smaller the portion of valid configurations will be. But this number will still grow exponentially with respect to the map size. Here is the state space in the minesweeper problem

$$S = \{s : \text{linear equations defined in (3) from state s are satisfiable.}\}$$

Action set. The legal actions are of the form "probe the unknown square at (i, j)", $1 \leq i \leq W$, $1 \leq j \leq H$. We don't really need an equivalent of the "mark the unknown square (i,j) as a mine" action from the real game. We are interested in marking sure mines only, in which case this would not provide new constraints. Moreover, the squares surely containing a mine can be found directly from the current configuration, although this is the NP-complete problem MINESWEEPER, as defined above. This is not the case with an unknown square is being probed, when either new constraints are introduced or the game is "lost". The actions set is defined over all unknown squares. When an action is executed succesfully, the configuration changes and we have a new *valid* state. In case the number of unknown squares is equal to the number of mines we face a terminal state. When an action probes an unknown square with mine, the game is lost. The set of action is

$$A = \{a : \text{probe an unknown square in state } s\}$$

State transition. The transition probability from one state to another depends only on the square being probed and the on current configuration. Put another way, the previous actions and states don't affect the current transition probability distribution. We can compute the transition probability in a way similar to computing the probability that a square contains a mine (as a simple ratio of two #MINESWEEPER problems). Meanwhile, we don't allow the automatic opening of the neighbors of the probed square, although this is normally performed in the real game, in case the square happens to contain 0. It is trivial to see that this is not a limitation (these squares are all free for sure and we will infer it and thus probe them), but allows us to focus on step-by-step moves and thus, simplifies the analysis of the problem.

We can solve the equations in (3) derived from the current configuration and, by enumerating the possibilities and count the corresponding models, we can obtain the transition probability. A key observation is that the difference between the current state s and the successive state s' is only in the value of the square i, where the action $a(i)$ has been executed.

$$T(s, a(i), s' | s'(i) = k) = \frac{|\{X : X \in \mathcal{X}, X_i = 0 \text{ and } \sum_{X_j \in N(i)} X_j = k\}|}{|\mathcal{X}|}$$

The probability that the action a on square i probes a mine is given by $\left(1 - \sum_{j=0}^{8} T(s, a(i), s' | s'(i) = j)\right)$.

Reward. The reward is

$$R(s, a, s') = \begin{cases} 1 \text{ if action} a \text{ is successful.} \\ 0 \text{ otherwise.} \end{cases}$$

Actually, this reward function is shaping reward that can lead to fast convergence rate in learning state value. The real reward should be one for a sequence of actions that can probe all mines. But the convergence rate is too slow to be practical in solving problem. The shaping reward is given to any action that makes progress toward the goal. Ng [43] et al. have proved that the reward function R can be changed to $R + F$, in which the optimal policy is preserved, provided F satisfies

$$F(s, a, s') = \gamma\Psi(s') - \Psi(s)$$

Particularly, in the minesweeper, we can define $\Psi(s)$ as the number of unknown squares in state s. Hence shaping function F is the difference of numbers of unknown squares in two states. If s' is the next state of s, the biggest difference of unknown square is 1. Therefore, the optimal policy from shaping reward function is the same as the one from original reward.

8 Finding the state value

Once we have defined the five-tuple for the minesweeper problem, we compute the state value function by means of *value iteration.* Taking into account the huge number of states in a 4×4 map, an iterative loop over them, as required by value iteration, is almost infeasible.

Let us define M_i to be the observation of a map that contains i unknown squares. Let $M_i|_{X_j} = k$ be the state that the number in the X_j square is k and all others squares are the same as in the state M_i. Let $V(M_i)$ be the state value of the observation M_i, and let $\#P(M_i)$ be the number of the feasible mine configurations for the state M_i.

We notice that the MDP in the minesweeper problem is *acyclic* and *finite-horizon.* The difference between two successive states is in one unknown square only. Instead of directly applying value iteration, we compute the value function progressively, including two steps, branching and reverse counting.

At the very game beginning we have to branch a map M_n, where all squares are unknown. We use the DDP counting method, (the v. 2.0 of *relsat*, described above), to find the number of valid configurations in M_n. All unknown squares are iteratively set to all values $0, 1, ..., 8$, which provides an additional constraint over the map M_n. So, the states, following M_n, are:

$$F(M_n) = \{M_n|_{X_i=k} : X_i \text{ is an unknown square}, k \in [0, 8], \#P(M_n|_{X_i=k}) > 0\}$$

The transition probability is defined as:

$$T(M_n, M_n|_{X_i=k}) = \frac{\#P(M_n|_{X_i=k})}{\#P(M_n)}$$

where $\#P(M_n)$ is the number of solutions of the configuration M_n.

We can iteratively collect all the following valid states after the initial one, together with the corresponding transition probability. The states in minesweeper are grouped by the number of unknown squares. We begin with all states with M unknown squares. They are all combinations of mine positions in the map. Then we deal with all states with $M + 1$ unknown squares. All state value functions from level $(i + 1)$ can be computed using level i and the transition probability.

$$V(M_{i+1}) = \max_{X_j \text{ is unknown}} \sum_{k=0}^{8} T(M_{i+1}, M_{i+1}|_{X_j=k})(1 + V(M_{i+1}|_{X_j=k})) \qquad (5)$$

where $M_{i+1}|_{X_j=k}$ is a state from level i.

From equ (5), we obtain following result:

Theorem 8.1. Let X_j and X_k be unknown squares in state M. Let X_j be a sure-not-mine and X_k has none-zero probability to be a mine. Then the expected reward the agent will receive if he first tries X_j and then X_k will be larger than if he does it in the reverse order.

The theorem implies that the optimal policy will probe all sure-not-mines squares before taking the risk to try other squares.

Proof. A square is safe iff the sum of the transition probabilities at that square is always 1. (The sum of the transition probabilities of an unsafe square is less than 1.) The reward for X_j and X_k, taken in this order, is:

$$\begin{aligned} V(M) &= \sum_{X_j} T(M, M|_{X_j})\Big(1 + \sum_{X_k} T(M|_{X_j}, M|_{X_j X_k})(1 + V(M|_{X_j X_k}))\Big) \\ &= 1 + \sum_{X_j, X_k} T(M, M|_{X_j}) T(M|_{X_j}, M|_{X_j X_k})(1 + V(M|_{X_j X_k})) \end{aligned} \tag{6}$$

And the reward for the reverse order, probe X_k then X_j, is:

$$\begin{aligned} V'(M) &= \sum_{X_k} T(M, M|_{X_k})\Big(1 + \sum_{X_j} T(M|_{X_k}, M|_{X_k X_j})(1 + V(M|_{X_k X_j}))\Big) \\ &< 1 + \sum_{X_k, X_j} T(M, M|_{X_k}) T(M|_{X_k}, M|_{X_k X_j})(1 + V(M|_{X_k X_j})) \end{aligned} \tag{7}$$

The joint transition probability in equ (6) and (7) is actually the same, since both they are ratios of the numbers of solutions in state M and state $M|_{X_j, X_k}$. Therefore $V(M) > V'(M)$. The optimal policy from the maximal state value will guarantee that it probes the sure-not-mine square first. □

9 Cutting the state space

The most important problem for solving minesweeper in the reinforcement learning framework is the huge state space. Looking at the number of configurations, the situation seems hopeless and one hardly hopes the problem can be solved on a board bigger than 2×3. Happily, there are several ways we can dramatically cut the search space and still come up with an *exact* solution. A brief list of some useful techniques follows:

♢ work on the problem *graph*, not the *tree*;
♢ drop the inconsistent configurations;
♢ exploit the symmetries;
♢ exploit the sure moves;
♢ don't cares and mine removal;
♢ back-off to less mines;
♢ back-off to lower dimensionality.

9.1 Work on the graph, not the tree

We work on the problem *graph* (it is a DAG - directed acyclic graph) as opposed to the problem *tree*. Thus, we can reach the same state by following different paths from the root. So, in order to prevent redundant calculations, we store all the configurations calculated so far in a hash table. This is a standard dynamic programming approach, and follows directly our variant of the Bellman equations.

9.2 Drop the inconsistent configurations

This is the most obvious optimization. As we investigate what could be the outcomes of some probe, we can immediately check whether it leads to a consistent configuration (by solving MINESWEEPER): e.g. it can be impossible to have the number 7 inside the probed square because of the constraints. Obviously, such inconsistent configurations can be cut immediately. This is an enormous saving. In addition, we do not need to store them in the hash table.

9.3 Exploit the symmetries

Another obvious way to cut the state space is: exploit the symmetries. Each configuration has up to 8 distinct variants that could be obtained through symmetries. So, the idea is the following, once we have a configuration and we want to calculate its state value, we first look for it in the hash table. If not found, we try the other 7 symmetries. Only in case none of these has been calculated, we perform the calculations and the store the result.

9.4 Exploit the sure moves

Consider the configuration (we use " ? " to denote an unknown square):

?	?	?	?	?
2	?	2	?	?
?	m	?	?	?

Now we probe the lower left corner square and we enumerate all the possible outcomes: $\{0, 1, ..., 8, m\}$. Consider we want to find the value function of the new state where the probed square contains 2:

?	?	?	?	?
2	?	2	?	?
2	m	?	?	?

Do we really want to evaluate *this* configuration? Not actually, since there are obvious inferences. Consider there are 2 or 3 mines remaining to be found. Then we can infer there is one sure mine. Let us mark it with "+":

?	?	?	?	?
2	+	2	?	?
2	m	?	?	?

Then we can make all other possible inferences (we mark the free squares with "$-$"):

-	-	-	-	?
2	+	2	-	?
2	m	-	-	?

Now we can try to recover the values that are to be put instead of "$+$" and "$-$":

1	1	1	-	?
2	m	2	-	?
2	m	2	-	?

As we can see, this is straightforward for the "$+$"s: they just become mines. But the "$-$"s are a little bit tricky since some of them will not be resolved for sure: we know there is no mine there but this is not always enough to infer the correct value inside.

So, what we can do now? The simplest thing is to pretend that all unresolved "$-$" are just unknowns "? ". Note, that what we are really interested in is the *value* of the configuration. Then, we can just evaluate the following configuration and add 5 to obtain the value of the original one, without the inferences:

1	1	1	?	?
2	m	2	?	?
2	m	2	?	?

This is an obvious saving. But we can do more. Consider the last configuration has not been evaluated before and thus its value is not stored in the hash table. Well, we can investigate the configuration and find that there are 3 sure moves! They are all of the type "$-$" and cannot be resolved (we cannot decide in a deterministic way which number $\{0, 1, ..., 8\}$ any of them should contain, although we can possibly limit these options). But still we can exploit the fact there are sure "$-$". One way to do so is to introduce the special kind of squares of value "$-$" and reason with them. This might speed up the process but will make it more complicated.

There is a better idea. As mentioned above, the best move when there is a sure "$-$" is to probe it. First, this guarantees us one more move, and second, gives us more information, since we will discover the correct value there: $0, 1, ..., 8$. In the real game, some other free adjacent squares may open, but we do not allow this. Still, we can infer they should be sure non-mines (but we have to probe them to reveal their value).

Another important observation: The best three consecutive moves in the last configuration are to probe the three sure "–" we discovered. Note also, that the order doesn't matter! Well, then we can probe any of them: say the one with the smallest x and, if several, the one with the smallest y. No other moves need to be investigated! This is a large tree cut. And we will do the same on the next turn: first look for sure cases and their consequences, fill in all the mines and all the sure numbers, then probe a sure "–", whose numerical value cannot be inferred (if any). Only when none of these is possible, we will investigate the "? ". Note, that this does not compromise the calculation of the correct value of the state value function.

9.5 Don't cares and mine removal

Consider again the last configuration. We have 2 regions there: known (uncovered squares) and unknown (covered squares). We can further split the known region into *boundary-known* (*BK*), i.e. adjacent to at least one unknown square, and *non-boundary-known* (*NBK*). A critical observation here is that what is there inside NBK is irrelevant – we "don't care" (well, except for just the *number of mines* inside this region). So, we can group the configurations with the same set of NBK squares, regardless of what they contain inside the NBK region: we are interested in the *future*, not the *past*, and they obviously will have the same state value function. So, we introduce "*" for all NBKs:

$\star$	$\star$	1	?	?
$\star$	$\star$	2	?	?
$\star$	$\star$	2	?	?

There are two problems with this however: First, we have to update the numbers in the BK cells, so that they are independent on what is inside NBK. This is easy and requires just looking at the adjacent mines:

0	0	1	?	?
0	0	2	?	?
0	0	2	?	?

The second problem is a problem of performance – how do we store the configurations with "don't cares" in the hash table? And how, given a particular instance without "don't cares", we find in the hash table the patterns that it matches?

While this could be done effectively, there is a much simpler solution. Each time a mine is discovered, we can perform a further checking: if everything around it is known, we can remove the mine and pretend there have never been one in this square (but we should decrease the number of remaining mines). Since everything is already discovered in all adjacent squares, they can contain only numbers $1 - 8$ (0 is impossible) or m. So, we decrease by 1 all ones that contain a number. Then we count those that contain a mine and put this number instead of m in the target square. Going back to our example, if we remove the upper mine first, we obtain:

0	0	0	?	?
1	1	1	?	?
1	m	1	?	?

Then, we cope with the second one, and we have:

0	0	0	?	?
0	0	0	?	?
0	0	0	?	?

As a result, there are no "don't cares" at all – so the configuration can be stored in the hash table and looked for effectively. The idea is: if we keep doing so, the explored zone will be filled with 0s... Note, that it will not be that nice always. Let us for example, change $(1, 3)$ from 1 to 2, i.e.:

1	1	**2**	?	?
2	m	2	?	?
2	m	2	?	?

Let us now repeat the procedure, removing the upper mine first:

0	0	1	?	?
1	1	1	?	?
1	m	1	?	?

And then - the second one:

0	0	**1**	?	?
0	0	**0**	?	?
0	0	**0**	?	?

So, we have a 1 in the BK region (shown in bold). Of course, the BK region can be much more complex, but anyway – each time we discover everything around a sure mine, we can update the values of the squares around it.

9.6 Back-off to less mines

Let the board size be fixed to $H \times W$ and we want to solve the problem for M mines. An obvious strategy is to solve it for 1 mine, then for 2 mines, ..., and finally – for M mines. We keep the solutions in a hash table. Consider we are solving the problem for k $(1 < k \leq M)$ mines. Then, after the first sure mine has been discovered, we are done: we already have the solutions for all configurations with $k - 1$ mines in the hash table, so we just take the value from there. Which is, we always will have to solve the problem only until the first mine is found.

9.7 Back-off to lower dimensionality

Another direction to go is to solve all configurations of *smaller dimensions* first and store their values in the hash table. Then we can see that:

0	0	0	?	?
0	0	0	?	?
0	0	0	?	?

reduces to solving:

?	?
?	?
?	?

Similarly, the configuration:

0	0	**1**	?	?
0	0	**0**	?	?
0	0	**0**	?	?

actually requires solving:

1	?	?
0	?	?
0	?	?

Although this can cut a lot and could be an important element of a problem decomposition, looking for the subcases is computationally expensive. And it is partially accounted for by the mine removal.

10 Experiments

We implement the progressive state value algorithm on a 4×4 map. The number of mines ranged from 1 to 14, and we iteratively calculated all the state values using equ (5). Both the greedy and the optimal policies have been applied in order to locate the mines on 1000 uniformly randomized mine fields.

The state value column in table (1) shows that the state value, which is the expected number of moves starting from a completely unknown board, decreases steeply after the total number of mines in a 4×4 map becomes larger than 4. When the mine number grows further, the expected number of moves shrinks to 0. Correspondingly, the performance of the optimal policy in figure (3) decreases when the number of mines increases.

Unlike the greedy policy, which randomly selects unknown squares at the beginning, the open move of the optimal policy on a initial totally uncovered map is fixed. On a 4×4 map, when the number of mines is small (less than 4), it always tries the corner squares in order to maximize the number of expected

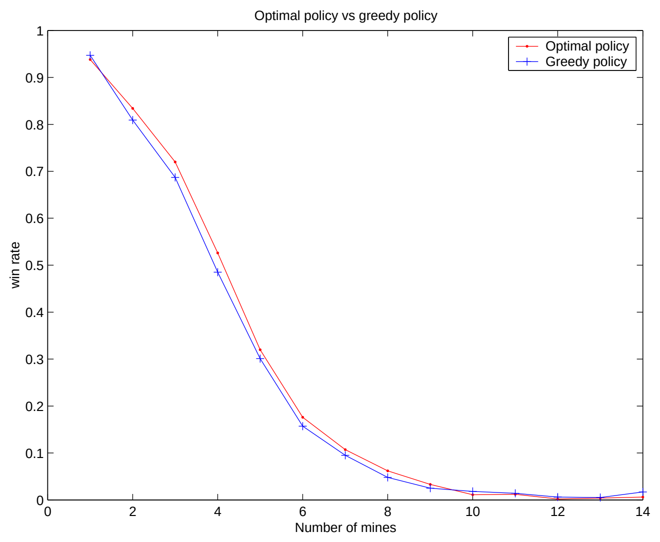

Figure 3: Optimal vs. greedy policy on a 4×4 map, open-move-not-free

moves. In case the mines count is between 4 and 11, the optimal policy only tries the border squares. On 3×3 maps the center square is never chosen as an open move.

We compare the optimal and the greedy policies on two kinds of mine fields: open-move-free (OMF) and open-move-not-free (OMNF). In the OMF case, figure (4) the optimal policy runs uniformly better than the greedy policy. Both policies perform well on maps with small number of mines and drop into a fairly poor level on maps with large number of mines. Since the 4×4 map is relatively small, after the successful open move, the optimal policy has a better chance to locate all other mines and thus, to win.

However, as figure (5) shows, in the OMNF case the optimal policy has only a very slight improvement over the greedy one. On a map, almost full of mines (i.e. more than 9 mines for a 4×4 map), the optimal policy is even slightly worse than the greedy one (hopefully, the difference is really very small, and could just be due to a bad chance).

As mentioned above, we exploit the symmetries, since the symmetric states have the same state value and can be collapsed. Table (1) shows the number of states without symmetry is only about one seven times more than for the original state space size (mainly because on a small board different symmetries often produce the same configuration).

# of mines	State.value .	# of states	# of states (with symmetry)	Open move
1	14.0625	463,479	58,490	all squares
2	11.925	1,482,751	186,819	four corners
3	9.546429	2,879,304	361,587	four corners
4	7.154945	3,837,425	482,056	eight border squares
5	4.992674	3,758,616	471,886	eight border squares
6	3.340909	2,824,116	355,025	eight border squares
7	2.277185	1,673,834	210,602	eight border squares
8	1.557265	795,610	100,453	eight border squares
9	1.075087	305,172	38,678	eight border squares
10	0.753621	93,916	12,036	eight border squares
11	0.528159	22,755	2,965	eight border squares
12	0.365934	4,083	558	four corners
13	0.241071	479	76	four corners
14	0.141667	33	7	all squares except four corners

Table 1: State value, # of states and best open move on a 4×4 map

# of mines	State.value .	# of states	# of states (with symmetry)	Open move
1	7.111111	1,898	289	all squares except center
2	4.888888	2,974	443	four border squares
3	2.904762	2,720	405	four border squares
4	1.698413	1,596	245	four border squares
5	0.952381	626	106	four border squares
6	0.535714	144	30	four border squares
7	0.277777	18	6	all square except center

Table 2: State value, # of states and best open move on a 3×3 map

# of mines	State.value .	# of states	# of states (with symmetry)	Open move
1	1.5	11	4	all squares
2	0.666666	5	2	all squares

Table 3: State value, # of states and best open move on a 2×2 map

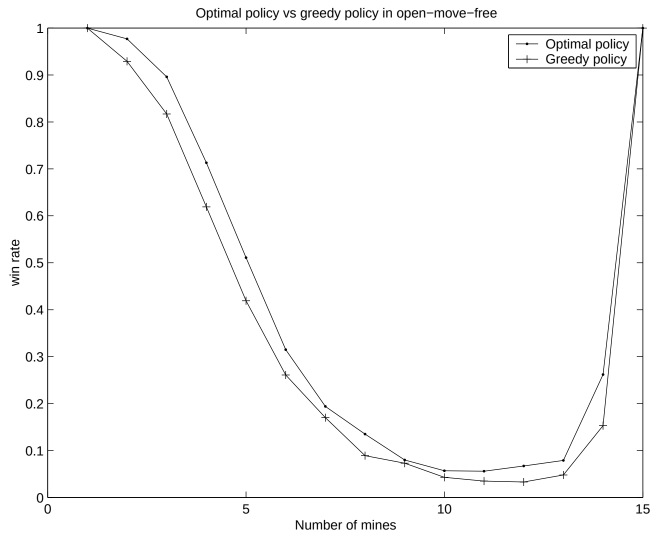

Figure 4: Optimal vs. greedy policy on a 4×4 map with open-move-free

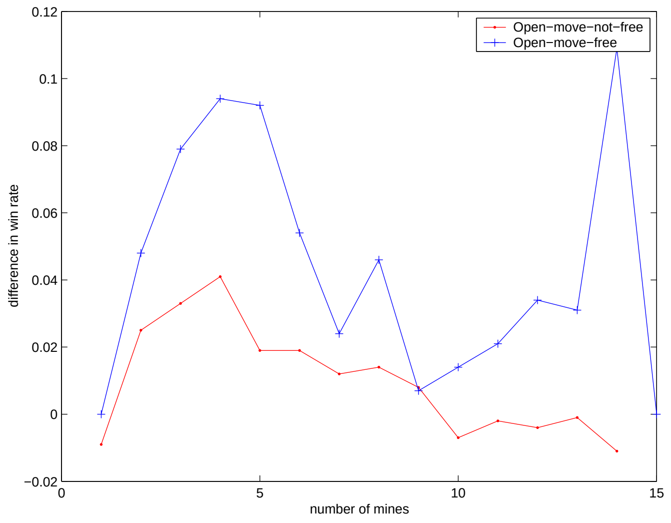

Figure 5: open-move-free vs. open-move-not-free

11 Function approximation

We can see that both the greedy policy and the reinforcement learning, as compared on a 4×4 map, drop in performance in a similar way. Generally, the performance of the optimal policy is better than the greedy one. However, even when we already eliminate all possible states into the lowest level, the total number of valid states are out of control in 5×5 map. This means we need a state-value function approximation.

There are several different ways we can define a function approximation. The classic approach would be to use a method dependent on local patterns (e.g. squares of size 3×3, 4×4 or 5×5), such as neural network or genetic algorithm. We can also try to apply some machine learning approach to learn interesting patterns or even rules. But we are not very happy with any of these solutions since they all are necessarily local and cannot take into account all the logical dependencies present on a particular board configuration.

The strong results of the greedy policy make it a good candidate for function approximation. Unfortunately, we cannot apply it directly, since it does not give us an estimate of the expected number of moves we can perform on a particular configuration. But it could still be possible to do so. The idea is the following: we can find (exactly of approximately), the probability of containing a mine for each uncovered square. Then we can estimate how far we can go, if we never learn anything more, e.g. if the lowest probabilities of having a mine are $0.1, 0.2, 0.5, ...$, then we can survive one move with probability $(1-0.1) = 0.9$, two moves - $(1-0.1)\times(1-0.2) = 0.72$, three moves $(1-0.1)\times(1-0.2)\times(1-0.5) = 0.36$ etc. So, we have to calculate the corresponding expectation. We can also take into account how often sure cases are likely to appear (and how it is related to the number of uncovered neighbors, mines count, board size etc.) In fact, we already have the *exact* solutions for boards as large as 4×4, so we can try to *learn* the value function approximation from them.

So, we will possibly be able to scale $-$ our observations show that the greedy policy, when using *relsat* is almost always capable to solve the expert size board in a reasonable amount of time. Thus, our function approximation will be able to do so as well. But we will need a much more effective compression of the state space in order to be able to apply reinforcement learning on big boards.

12 Future work

There are several interesting things to be done, including:

◇ try the function approximation, described above;

◇ apply a polynomial approximation to model counting;

◇ automatically learn the state value from trials, instead of computing the transition probability from SAT;

◇ automatically discover deterministic patterns;

◇ apply state partitioning;

◇ apply neural net approximation to state value;

♢ try other reinforcement learning algorithms: e.g. (modified) policy iteration;

♢ apply the last space cut, described above and try to come up with a divide&conquer algorithm to cope with bigger boards;

♢ think of more powerful state space cuts;

♢ implement a greedy policy with multiple lookahead: e.g. 2 or 3 moves, etc.

13 Acknowledgements

We are very grateful to Prof. Stuart Russell for pointing us to useful materials on minesweeper (especially in the early stages of the project development), as well as for the valuable discussions. We would like to thank also Ariel Schwartz, for sharing with us the term paper and the code from his old project on a similar subject, as well as for the informal discussions. We are also grateful to Nikola Marchev for providing us with the code and some useful comments about his own implementation of minesweeper.

References

[1] Kaye, R. Some Minesweeper Configurations. August 13, 2000. (http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf).

[2] Millennium Prize Problems, Clay Mathematics Institute of Cambridge, Massachusetts: http://www.claymath.org//Millennium_Prize_Problems/.

[3] Minesweeper strategies & tactics: http://www-cs-students.stanford.edu/~jl/Essays/minesweeper.html.

[4] Minesweeper strategies: http://www.frankwester.net/winmine.html.

[5] PBLIB: The Pseudo-Boolean Library. http://www.cirl.uoregon.edu/PBLIB/.

[6] Programmer's minesweeper: http://www.ccs.neu.edu/home/ramsdell/pgms/index.html.

[7] RELSAT,v2.00: http://www.almaden.ibm.com/cs/people/bayardo/resources.html.

[8] SAT Live! http://www.satlive.org.

[9] SATLIB - The Satisfiability Library: http://www.satlib.org.

[10] SICStus Prolog: Boolean Constraint Solver http://www.cl.cam.ac.uk/ailanguages/sicstus3.7/sicstus_31.html.

[11] SICStus Prolog: http://www.sics.se/sicstus/.

[12] The Authoritative Minesweeper: http://www.metanoodle.com/minesweeper/.

[13] A. Adamatzky. How cellular automaton plays Minesweeper Applied Mathematics and Computation. 85(2-3):127–127, 1997.

[14] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A Backtrack Search Pseudo-Boolean Solver. *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[15] P. Barth. Linear 0-1 Inequalities and Extended Clauses. Technical Report MPI-I-94-216, Saarbrücken, Germany, 1994.

[16] R. Bayard and J. Pehoushek. Counting Models using Connected Components. *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI)*, 2000.

[17] R.J. Bayardo and R.C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

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

[19] E.R. Berlekamp, J.H. Conway, and R.K. Guy. *Winning Ways for Your Mathematical Plays.* Academic Press, 1982.

[20] E. Birnbaum and E.L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.

[21] G. Boole. *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities.* Dover 1st printing, 1854.

[22] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp*, pages 35(8):677–691, 1986.

[23] L. Castillo and S. Wrobel. Learning Minesweeper with Multirelational Learning. *Proceedings of IJCAI 2003 (International Joint Conference on Artificial Intelligence). In press.*, 2003.

[24] S.A. Cook. The complexity of theorem-proving procedures. pages 151–158. ACM Symp. on Theory of Computing, ACM, New York, 1971.

[25] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, pages 7(3):201–215, Jult 1960.

[26] H.E. Dixon and M.L. Ginsberg. Combining satisfiability techniques from AI and OR.

[27] O. Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81:49–64, 1991.

[28] E.C. Freuder and M.J. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *In Proceedings of IJCAI-85*, pages 1076–1078, 1985.

[29] M. Gardner. The Fantastic Combinations of John Conway's New Solitaire Game 'Life.', October 1970.

[30] K. Iwama. CNF satisfiability test by counting and polynomial average time. In *SIAM J. Comput*, volume 18, pages 385–391, 1989.

[31] K. Iwama and A. Matsuura. Inclusion-Exclusion for k-CNF Formulas.

[32] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *In J. Lassez (ed.), Logic Programming - Proceedings of the 4th International Conference*, number 1, pages 196–218, 1987.

[33] P. Jordan. Evolving Minesweeper Strategies Using Genetic Algorithms (CS 472 term paper).

[34] J. Kahn, N. Linial, and A. Samorodnitsky. Inclusion-Exclusion: Exact and Approximate. *Combinatorica*, 16(4):465–477, 1996.

[35] R. Kaye. Richard Kaye's minesweeper pages. http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm.

[36] R. Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer*, 22:9–15, 2000.

[37] R. Kaye. Infinite versions of minesweeper are Turing-complete., August 2002.

[38] N. Linial and N. Nisan. Approximate Inclusion-Exclusion. In *ACM Symposium on Theory of Computing*, pages 260–270, 1990.

[39] D. Loveland. Automated Theorem Proving: A Logical Basis. *Fundamental Studies in Computer Science*, 6, 1978.

[40] E. Lozinskii. Computing propositional models. *Inform. Process. Lett.*, pages 41:327–332, 1992.

[41] Luby, M. and Veličkovic, B. On deterministic approximation of DNF. pages 430–438, 1991.

[42] A. Melkman and S. Shimony. A note on approximate inclusion-exclusion. *Discrete Appl. Math.*, pages 73:23–26, 1997.

[43] A.Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th International Conf. on Machine Learning*, pages 278–287. Morgan Kaufmann, San Francisco, CA, 1999.

[44] S. Prestwich. Randomised Backtracking for Linear Pseudo-Boolean Constraint Problems. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 7–20, 2002.

[45] C Quartetti. Evolving a Program to Play the Game Minesweeper. *Genetic Algorithms and Genetic Programming at Stanford*, pages 137–146, 2000.

[46] J. Rhee. Evolving Strategies for the Minesweeper Game using Genetic Programming. *Genetic Algorithms and Genetic Programming at Stanford*, (6):312–318, 2000.

[47] D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[48] I. Stewart. Million Dollar Minesweeper. *Scientific American*, pages 94–95, October 2000.

[49] L. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, pages 8: 189–201, 1979.

[50] W. Zhang. Number of Models and Satisfiability of Sets of Clauses. *TCS*, pages 155(1): 277–288, 1996.