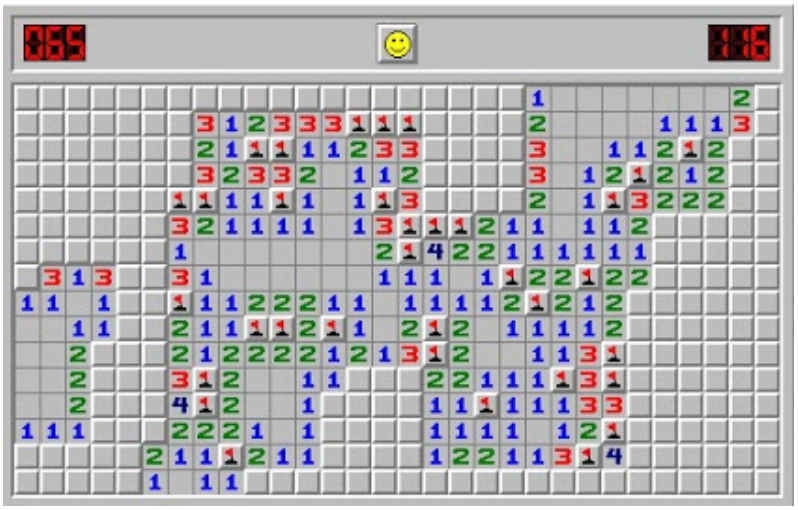
MINESWEEPER

Final project in Artificial Intelligence

Tamar Vaitzman, Galina Efremov, Modi Ennekavi, Alon Shevach, Nitai Mordechai

"Creativity is just connecting things." - Steve Jobs



Contents

Part I

Introduction

Minesweeper has its origins in the earliest mainframe games of the 1960s and 1970s. The earliest ancestor of Minesweeper was Jerimac Ratliff's Cube. The basic gameplay style became a popular segment of the puzzle video game genre during the 1980s, with such titles as Mined-Out (Quicksilva, 1983), Yomp (Virgin Interactive, 1983), and Cube.

Minesweeper is a single-player puzzle video game. The goal of Minesweeper is to uncover all the squares on a grid that do not contain mines without being "blown up" by clicking on a square with a mine underneath. The location of the mines is discovered through a logical process (but that sometimes results in ambiguity). Clicking on the game board will reveal what is hidden under neath the chosen square or squares. Some squares are blank while others contain numbers (from 1 to 8), with each number being the number of mines adjacent to the uncovered square.To help the player avoid hitting a mine, the location of a suspected mine can be marked by flagging it with the right mouse button. The total number of mines is known by the player. The game is won once all blank or numbered squares have been uncovered by the player without hitting a mine; any remaining mines not identified by flags are automatically flagged by the computer. There are three difficulty levels for Minesweeper: beginner, intermediate, and expert. Beginner has a total of ten mines and the board size is either $8 \times 8, 9 \times 9$, or 10×10. Intermediate has 40 mines and also varies in size between 13×15 and 16×16. Finally, expert has 99 mines and is always 16×30.

Part II

MineSweeper Solution

Minesweeper may seem like a simple computer game to pass the time with, but it recently became a hot topic in computational complexity theory with the proof by Richard Kaye that the minesweeper consistency problem is in a complexity class of problems known as NP− complete problems.

In our solution we will examine three important challenges a Minesweeper solving algorithm must consider to be successful: first move, finding safe moves on the board and handling guesses when the safe move is impossible.

We build two reasonable AI methods to solve a MineSweeper by a computer:

1. By using a backtracking search algorithm with heuristics.
2. By SAT with learning heuristic.

1 Backtracking Search Solution

1.1 The Game as a Constraint Satisfaction Problem

Let adapt basic terms of CSP problem to our game:

Each opened cell has constaints - a number on it, which indicates how many mines are held in the surrounding squares, which are the adjacent squares in the eight directions: north, north-east, east, south-east, south, southwest, west and north-west. Each closed cell has the values $\{0,1\}$, when 1 is for a safe cell, and 0 is for a flag(a mine). As it was said before, the total number of mines is known by the player. The unsigned variables are all the closed cells, on which we need to make a choice.

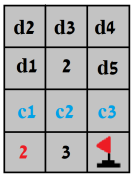
We defined "safe move" as follow: m,n is the height and width of the board respectively.

Let $F : [m] \times [n] \to \{\mathbb{T}, \mathbb{F}\}$ be a boolean function which is $\mathbb{T}$ on (i,j) iff (i,j) is flagged.

Let $N : [m] \times [n] \to [0,8]$ be a function which indicates for each (i,j) cell how many mines are around it.

Let Neighbors: $[m] \times [n] \to 2^{[m]\times[n]}$ be a function which returns a set of neighbors of a given cell (8-connected).

Then opening of $(i,j) \in [m,n]$ is considered safe iff:

- $\exists n \in neighbors\,(i,j)$ s.t n is opened and $N\,(n)$ (number of mines around it) equals to $|\{(l,k) : (l,k) \in neighbors\,(n) \wedge F\,((l,k$ (number of flags around it).
- $\exists n \in neighbors\,(i,j)$ s.t n is opened and there exists a set S s.t $S \subseteq neighbors\,(n)$,$(i,j) \notin S$ and $\#mines$ in $S = N\,(n)$- illustration 

in this example, we can see that there exists such group - beacuse of the 3 in $(3,1)$, $\#$mines in $S = \{c_1, c_2, c_3\}$ is 2, $\{d_1, .., d_5\} \not\subseteq S$ and therefore $\{d_1, ..., d_5\}$ are all safe moves.

1.2 Algorithm Stages

The backtracking search is based on recursion. The base case of the algorithm is a complete assignment, which we called "safe move". It means, that we will open a cell iff we know that there is no mine on it. We implemented the "'safe move"" function in the next way:

1. The simpliest case is when the move we make is the first one in the game - in this case we just choose a random cell and open it. It is not possible to get any information before opening the first cell, so sometimes the agent will

lose on the first move. At each step at the game, our agent holds a set of safe moves, which it finds while running the recursion.

2. The other case is to choose a cell from the set of safe moves and check if it is a complete assignment. If so - open it.

If the case is not one of the previous - not the first move and the set of safe moves is empty(or it is a complete not assignment), we start a recursion.

1.3 Heuristic functions

Backtracking is performed with the help of some heuristic functions:

1.3.1 Most Constrained Variable Heuristic

The first heuristic function chooses the next unassigned variable. At each step at the game, by openning a new cell and adding it to the knowledge, we do the follow:

We go over all the closed neighbors of this cell and save them with the number of known opened cells around them. If we saved some closed cell before, we will increase the number of its known opened neighbors by 1. If not - the number of opened neighbors will be 1. To select the next variable to assign, we will choose the cell that has the biggest number of opened neighbors.

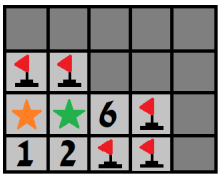
To select the value for a given variable we did not use heuristic. At this game for each cell we have three options: (1) the cell is safe to open, (2) the cell is a flag(mine) or (3) the choise is unknown, it means there is not enough information about the cell. We first check the options $(1), (2)$:

As we said before, the domain of each variable is $\{0, 1\}$, where 0 is for flag(mine) and 1 is for safe cell. We make the desicion, of which value to choose, by checking the constraints on the opened neighbors: the number of flags around the opened cell must be equal to the number that is written on this opened cell.

1.3.2 Probability Heuristic

If we do not have enough information about the cell, which are neighbors of the opened, we use the probability heuristic.

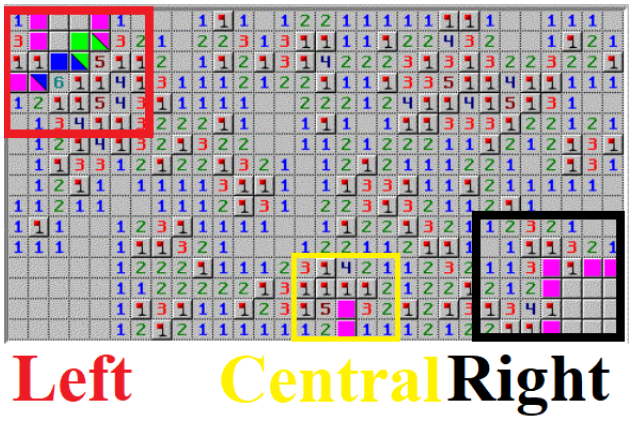
We first thought about calculating the probability for a mine to be in a close square by calculating the local probability and taking the average.We'll explain with an example: let's take a look at the picture below. By looking at the constraint of "1" open square, each of the green square and the orange square have a probability of 50% to be a mine. However, by looking at the green square, due to the constraint of the "6" square, the probability of the green square to be a mine is $\frac{2}{3}$ - which affects the probability of the orange square (to be only $\frac{1}{3}$)



Due to this dependency of probabilities, in some areas in board, we understood that we should find the connected components(disjoint sets) in a current state of board, and calculate the number of configurations to arrange the mines for each connected component. For example, if a certain square in a connected component has in 3 possible arrangement a mine and in 1 arrangement doesn't then it would have 75% probability to be a mine. But that's not it! Because the number of mines in different arrangements can be different. Thus, the complete solution as follow:

We first find the connected components as before. It means we find all the sets of closed cells that are independant of each other (by indepedent we mean that each two different sets have no common cells that assigning a value on it affects on assigments of cells in the other set).

Example: at the given board there are three disjoint sets - $Left, Right, Central$:



We take different combinatons of connected components. For each combination, for each conneced component we calculate the number of possible arrangments of mines (all the arrangments of mines we can assign with the constraints + number of arrangments of remaining mines (if there are mines left to arrange in board since we know the amount of total mines in board) in closed cells in all conneced components of the combination and divide it by the total number of arrangements in all board.

When taking different combinations we checked that we didnt't pass the number of mines in board.

If a certain square had a flag in all arrangements we knew for sure that it has a flag (or if a certain square had no flag in it in all possible arrangements, we could infer is safe for opening). The square with the maximum probability would be marked as a flag.

2 SAT solution

2.1 The Game as a SAT Problem

For this solution we first defined a "Sentence" object.

A sentence represents the information that the user gains when he opens a new cell. We defined a “Sentence” as a group of cells, and a number which represents how much of them are mines. Let $F : [m] \times [n] \to \{\mathbb{T}, \mathbb{F}\}$ be a boolean function which is $\mathbb{T}$ on (i, j) iff (i, j) is a mine, and let $S \subseteq [m], [n]$, $k \in \mathbb{N}$ then a single sentence (S, k) represents the following:

$$\varphi_k(S) = \bigvee_{\substack{|V| = k \\ V \subseteq S}} \left(\left(\bigwedge_{v_i \in V} v_i \right) \wedge \left(\bigwedge_{v_j \in S \setminus V} \overline{v_j} \right) \right)$$

there are exactly k cells that are mines in S. As the game proceeds, we gather more and more sentences in our Knowledge base. Our mission is to find such F (Assignment) that is consistent with all the sentences that we have so far.

2.2 Algorithm Stages

Open all safe moves first. Our first approach was finding safes with brute force search on the assignments space. Our technique was to iterate over the existing sentences in the knowledge base, and for each sentence finding other sentences which have the biggest common elements quantity and share at least one undiscovered cell (we chose to work with at most **eight** sentences in parallel due to computation limitations - there an information against time tradeoff). We iterate over all possible assignment but choose only those that satisfy at least one sentence, i.e for a sentence of the form $(S, 5)$ we include all the assignments that gives to exacly five elements of S the value $\mathbb{T}$ (there are $\binom{|S|}{5}$ of those). Then, we add an assignment to the collection if it satisfies each and every sentence from that group of eight.
An opening (i, j) is considered safe iff there isn't a consistent assignment F in which $F(i, j) = \mathbb{T}$. Note that if we add more sentences, intuitively the valid assignment is even more constrained, so the case in which another sentence is added and suddenly there is an assignment F in which $F(i, j) = \mathbb{F}$ that satisfies all sentences is **impossible**.
A flagging (i, j) is considered safe iff in all satisfying assignments F it holds that $F(i, j) = \mathbb{F}$.
But this first approach wasn't fast enoug
To deal with it, ee added **resolution,** which made it faster. our resolution works as follows: Infer new constraints from every opening with the new information that has been recieved - Iterate over the sentences.

The Resolution Procedure:

1. For each sentence (S_i, k_i) use **Subsumption Elimination -**
   if there exists (S_j, k_j) in sentences s.t $S_i \subset S_j$ and $k_i = k_j$, eliminate S_j, k_j from sentences and add to the constructive assignment F the following values: $\forall s \in S_j \setminus S_i$ assign $F(s) = \mathbb{F}$.
   thats because if there exists even one mine in $S_j \setminus S_i$ then $|\{s : F(s) = \mathbb{T}, s \in S_j\}| > |\{s : F(s) = \mathbb{T}, s \in S_i\}|$ in contradiction to the constraint $k_j = |\{s : F(s) = \mathbb{T}, s \in S_j\}| = |\{s : F(s) = \mathbb{T}, s \in S_i\}| = k_i$.
2. if there exists (S_j, k_j) in sentences s.t $S_i \subset S_j$ and $k_i < k_j$, we produce a new sentence of the form $(S_j \setminus S_i, k_j - k_i)$ and eliminate (S_j, k_j) , In order to reduce the number of literals in KB (and as a result, infer and check validity of the assignments faster).

2.3 Heuristic functions

When there are no safe moves to make, we pick one with a wise probablistic estimator, and we gathered few (2) such different estimators:

2.3.1 Probability Heuristic No. 1: Pattern to probability Data Base Move Heuristic:

The following is a data-structure that is pre-calculated against $1000 * \sum_{i=1}^{16} \binom{16}{i}$ 4×4 games , i.e for each combination of 1-16 bombs on a 4×4 size board, we ran our simulator 1000 times, and saved for each configuration wev'e been through, the move that has been made by our agent and didn't make it lose – afterwards, for each and every configuration we found the most common move that didn't make it lose, and calculated the probability of not losing with move by the following formula : $\frac{Number\ of\ times\ move\ made\ in\ that\ configuration\ and\ not\ lost}{number\ of\ times\ that\ this\ configuration\ appeared}$, so our DB stores $(configuration, best\ move\ from\ configuration, winning\ probability\ with\ move)$after that DB initiation, at each probebalistic choice, we iterate over all 4×4 windows in the board, collect best move and it's probability, then we choose the move with the maximum probability of not losing.

2.3.2 Probability Heuristic No. 2: Satisfying assignments heuristic:

In this heuristic, We iterate over all groups of 10-closest sentences -
The metric is as follows - $\mathrm{D}((S_1, k_1), (S_2, k_2)) = 8 - |S_1 \cap S_2|$ i.e the bigger the size of the intersection -> the closer the senteces, 8 is the maximum size of the intersection.
We then count all valid assignments of that specific group, and at the end we count the number of assignments each variable gets the values $\{0, 1\}$, and then choose $argmax_{v \in \bigcup_{i=1}^{\#sentences} S_i} \{max(\#1(v), \#0(v))\}$ when $\#1(v)$ is the number of satisfying assignment in which the variable v got 1, $\#0$ is similar.

Part III

Results and conclusions

We first define:

$CSP\,MC$ - CSP solution with most constrained heuristic function

$CSP\,PROB$ - CSP solution with probability heuristic function

$CSP\,ALL$ - CSP solution with probability and most constrained heuristic functions

$SAT\,AA$ - SAT solution with satisfying assignments heuristic function

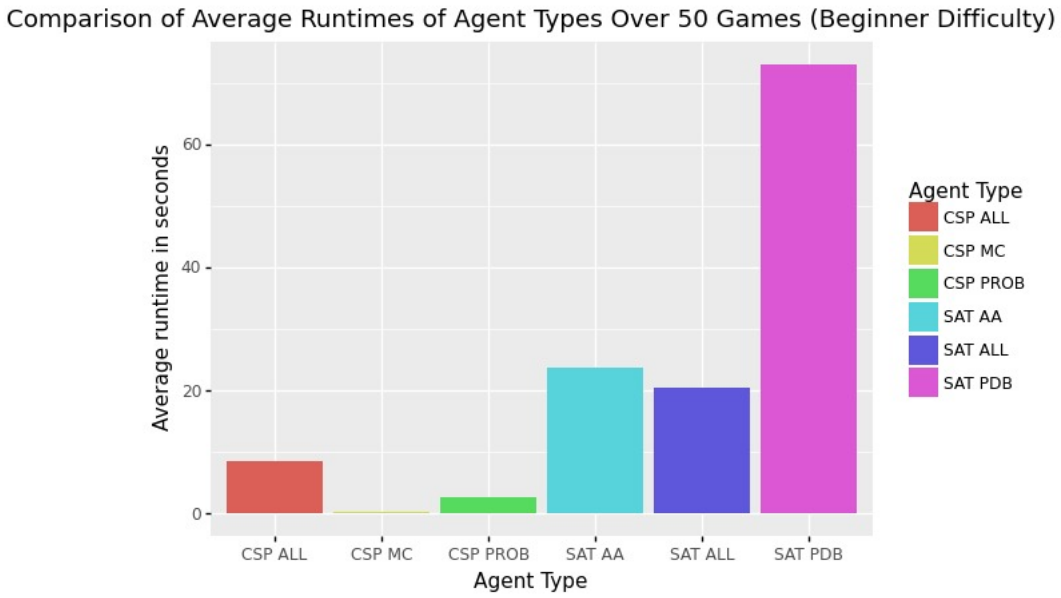
$SAT\,PDB$ - SAT solution with probability heuristic function

$SAT\,ALL$ - SAT solution with probability and satisfying assignments heuristic functions

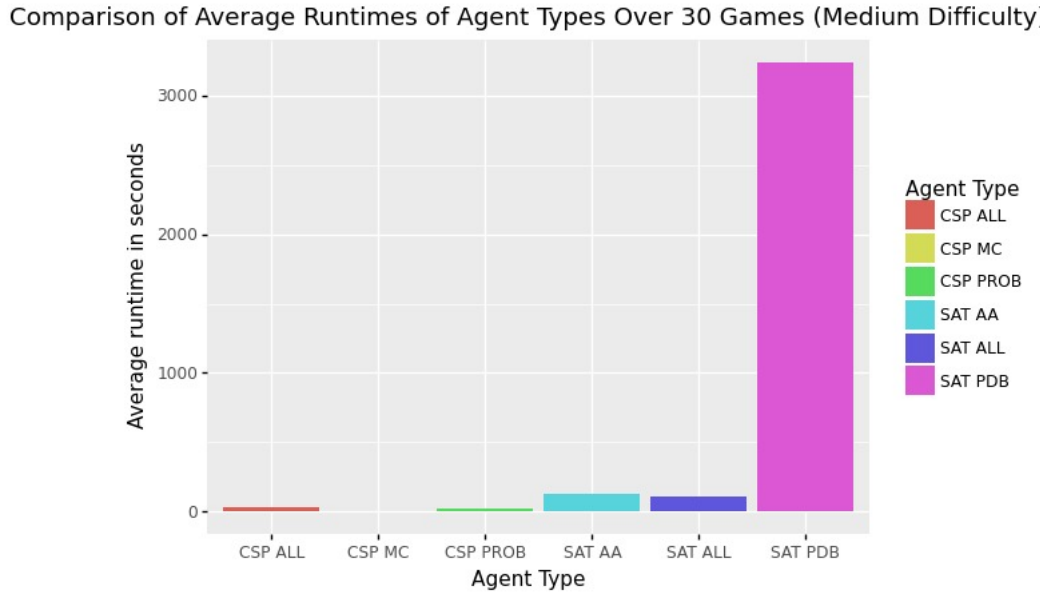
1 Comparison of runtime

We ran the game for different difficulty levels and these are the results we got:

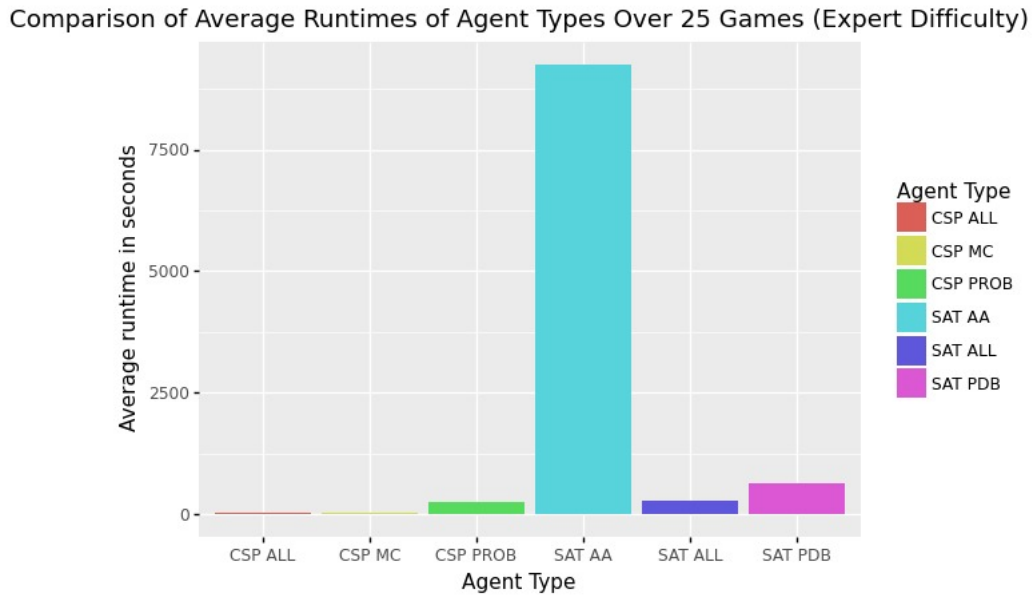
1.1 Board size 8×8 with 10 mines



1.2 Board size 16×16 with 40 mines



1.3 Board size 16×30 with 99 mines



1.4 Conclusion of runtime comparison

As we can see, the runtime of CSP with or without some of the heuristic functions is better than the runtime of SAT with or without some of the heuristic functions at each difficulty level. Let us note that when using a SAT solution with all the heuristics, then the difference between runtime of SAT and CSP decreases. In accordance with this, we can conclude, that SAT heuristic functions significantly optimize the runtime, when we use both the functions. While running CSP with all heuristics gives us minor difference compared to running CSP without some of them.

At each difficulty level, CSP with the most constrained variable heuristic gives us the fastest runtime, compared to the CSP with both heuristic functions and to the CSP with probability heuristic. And this is not surprising. The most constrained heuristic‘s goal is to find the cell which has the most constraints on it. By choosing this cell, we will reduce the depth of the backtracking, because we will have to "correct" less unsuccessful assignments.

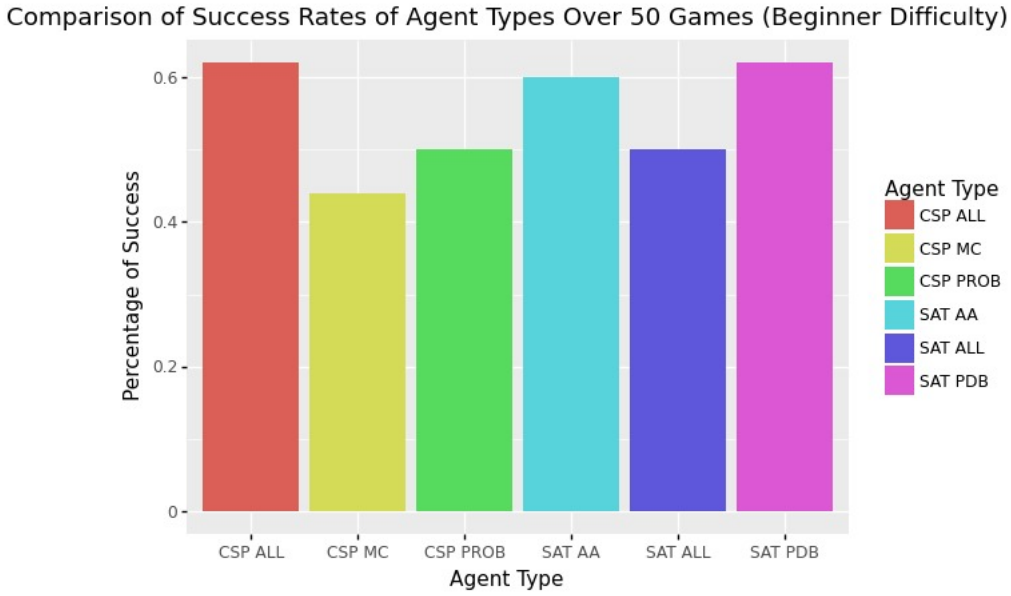
The CSP solution with the probability heuristic affects the runtime according to the difficulty level. In accordance with this, we can conclude, that the intermediate difficulty level has the smallest number of cases of unsafe moves

(because at this level the runtime of CSP solution with the probability heuristic is the smallest one).

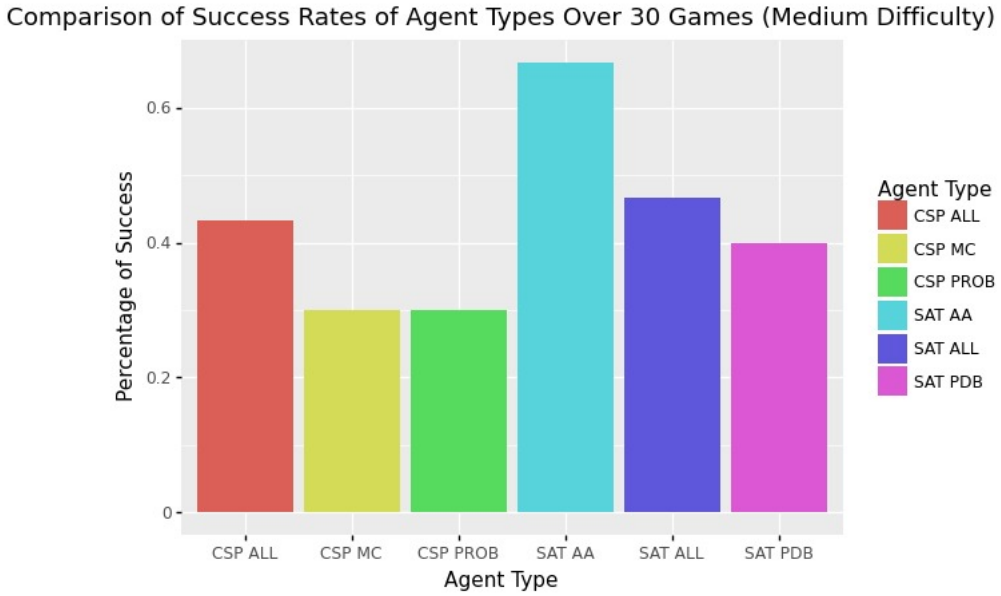
2 Comparison of success

We ran the game for different difficulty levels and these are the results we got:

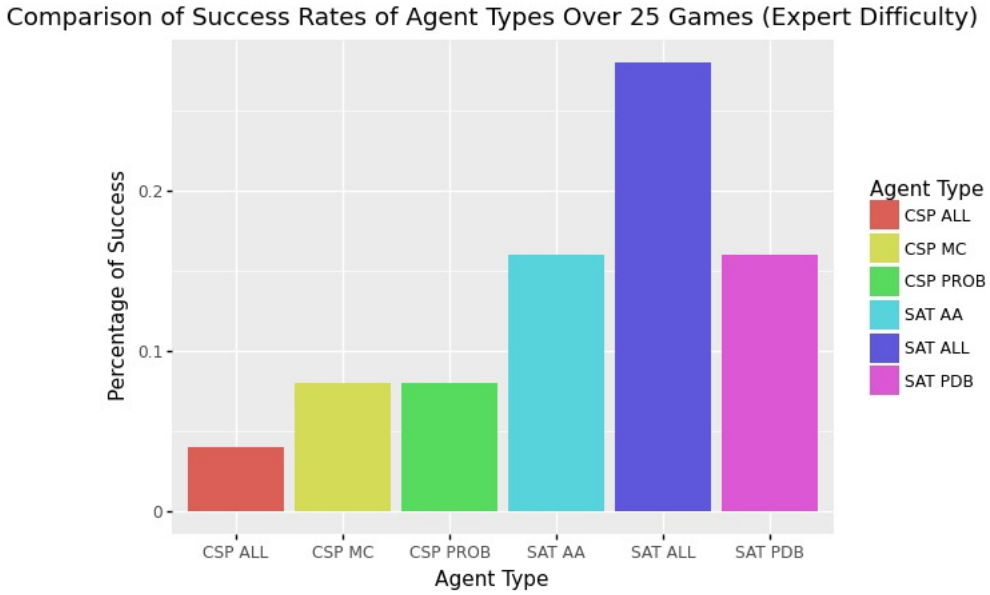
2.1 Board size 8×8 with 10 mines



2.2 Board size 16×16 with 40 mines



2.3 Board size 16×30 with 99 mines



2.4 Conclusion of success comparison

As we can see, at each difficulty level, SAT with any of the heuristics is better (or equal) at the success range, than CSP with any of the heuristics.

The SAT success depends on the heuristic it uses at each difficulty level. As we can see from graphs, at the beginner difficulty, the best heuristic to use with SAT is the probability heuristic function, at the intermediate - satisfying assignments heuristic function, and at the expert level use both. As we concluded before, the intermediate level has the smallest number of cases of unsafe moves, so we can assume, that SAT with the satisfying assignments heuristic deals less successfully in cases of unsafe moves (because it gives the best results at the intermediate level, while at the beginner and expert level the results are worse). From the same conclusion we also can assume, that SAT with both of the heuristics deals more successfully, when there are more cases of unsafe moves.

Part IV

Discussion

In this project, we presented two algorithms for the Minesweeper (CSP and SAT).

As future extension for the project, we recommend to run the DB heuristic for SAT more than 1000 iterations for each combination. This will cover more cases and give more accurate probabilities.

We also recommend to an extension for the probability part:

assemble different linear combinations of the heuristics weights and find the optimal weights by training(with neural network\q learning).

Another possible extension is constructing a neural network to be used as another probability heuristic - you can easliy train it by constructing states without safe moves and diagnose whether the move it returns is safe.

A native improvement to the extension above is taking a few neural networks with different architechture and assemble a linear combination with weights and find the best weights as mentioned above.

Part V

Running the game

Run the command **sh setup.sh** (which runs the command :pip install -r requirements.txt)

On windows - **py game.py**

On linux - **python game.py**