AI prog 2

Eivind Hærum & Hong-Dang Lam

October 24, 2013

Contents

1 Working GPS

Please refer to the delivered code. We have included a textfile for each problem with some strings of inputs to easily test the program for various configurations. We are somewhat uncertain what you wanted in the "Demonstration" sections. So we have described how we solved the problems internally and if you use the provided inputs it should be quite easy to verify that our code solves the different puzzles on all difficulties with Minimum Conflict, as well as the easy case for SA (and a few runs where the medium case is also solved by SA).

2 Main classes and methods

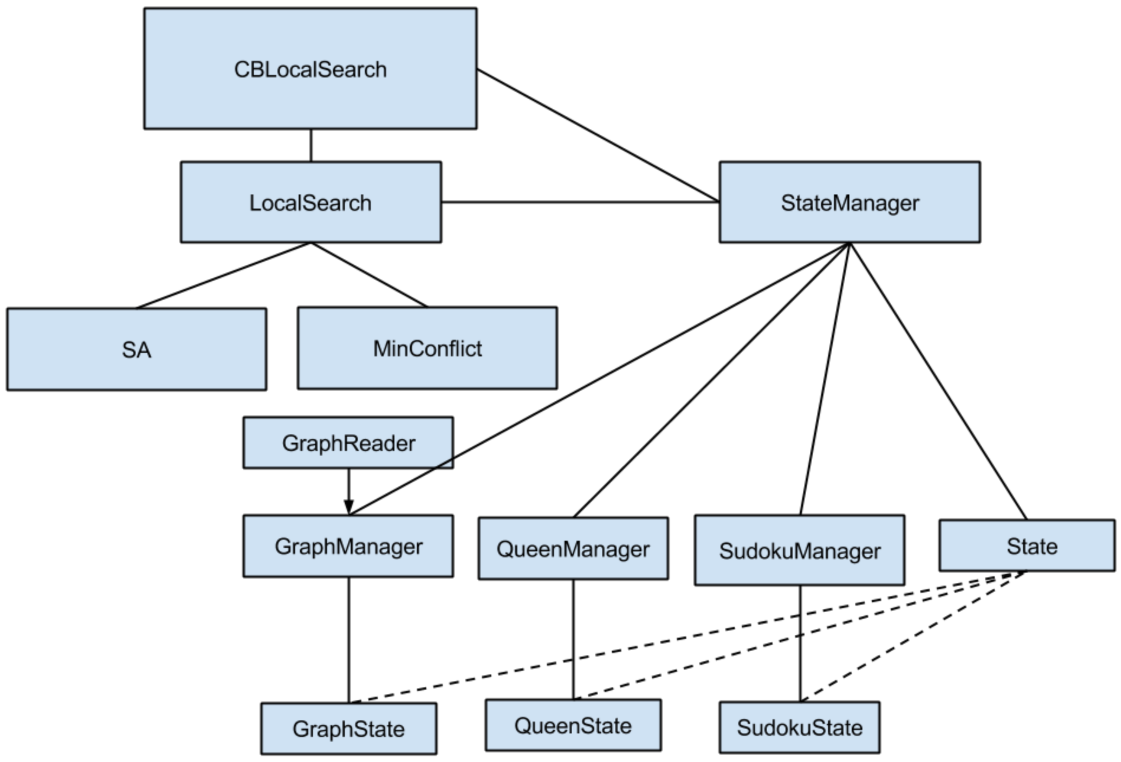

Figure 1: Diagram of the main classes

The figure shows our implementation, the first class CBLocalSearch takes care of the input from the user - number of runs, type of puzzle, type of algorithm and such. It then intiates a call to either GeneralSA or GeneralMinConflict which inherits from LocalSearch. Both these search-algorithm classes take in the manager of the selected puzzle and stores it in the superclass LocalSearch. The manager takes care of creating the initial state, this is done by calling the createInitState() method in each manager, the CurrentState is thus set to the intialState, and the

CurrentState gets updated as the algorithms do their thing. It also needs to call on updateConflicts() so we know how the currentState looks regarding to the constraints it must follow.
The CBLocalSearch initializes a new search by calling the constructor of one the subclasses of LocalSearch, either GeneralSA or GeneralMinConflict, which in turn calls on the solve() method specific to the current algorithm. From here there's two routes depending on which algorithm we have decided to use, let us take look into the SA algorithm first. The generalSA class calls the createChildren() method in the manager which creates 20 children and returns them, then a call to the manager is invoked with the findBestChild() method, which returns the state with the highest **F**-value.
The minConflict class on the other hand calls on the findBestNeighbor() method which is supposed to find a random location with a conflict and minimize the conflicts given the constraints. It then runs updateConflict() in the findBestNeighbor() method, this method updates the number of conflict and the conflict array so the next iteration has a correct indication of where it can go to work. It also requires the getGoalState() method so as to know when to terminate the search.
The code we've written tries to make the implementation as general as possible, the only thing needed to add a new puzzle is to add a manager and a new state type for the puzzle, the new state inherits the **F**-value and crashesCount from State, and needs to hold stuff specific to that problem as well as a few fields that might be used for printing out the final end state. The new manager class is required (by inheriting from StateManager) to have a constructor which calls the createInitState() which creates the initial state, as well as an implementation of updateConflicts(). For SA it needs a createChildren() method which returns an arrayList of states for the given puzzle, and a calculateF(State) which calculates the objective function for the given State. For MinConflict it needs to implement findBestNeighbor() which minimizes conflicts as well as the getGoalState() method so that minConflict knows when to terminate the search.

3 Third Puzzle

Our third puzzle is Sudoku, we have however modified the rules a bit because it makes it easier to implement and we feel that this modification fits the scheme of a scaling constraint satisfaction problem better.
The difference between this and the normal Sudoku is the fact that the normal sudoku is usually a board of 3x3 numbered grids (with numbers ranging from and including 1 to and including 9), and the board is pre-filled with some numbers already, these numbers are not allowed to be changed whatsoever.
Our modified version starts out with an empty board, follows the same basic rules as the normal Sudoku (meaning there can only be one of each number inside a square, and on each row and column) but we just fill the board in an arbitrary pattern and modify a complete "solution" untill we find one solution that doesn't violate the constraints. We do this by randomly filling each "square" with num-

bers from 1 to k*k (for instance 1-9 for k=3 - which is the normal sudoku size found in newspapers). We then only swap numbers inside each "square", thus we need only consider conflicts on the rows and columns as the numbers inside each square already meet the contraint that only one of each number may be inside each square. This modified version makes it easy to create scaling problems, as we can simply increase *k* and make bigger boards, thus making it much more complex and computationally harder.

To evaluate the board at a given time, we tally up the number of conflicts for each position with their corresponding column and row, and update a totalCount of all the conflicts.

For example if the board looks like this:

```
The board

 2  6  4     4  2  8     3  7  1
 5  9  1     6  3  5     6  9  5
 7  8  3     7  1  9     2  4  8

 6  5  1     1  6  4     7  2  4
 7  9  4     2  8  5     1  3  6
 2  8  3     7  9  3     9  8  5

 5  9  2     6  5  1     2  8  7
 8  6  7     3  2  7     5  4  6
 4  1  3     8  9  4     9  3  1
```

Then we have a conflict matrix and a totalCount that looks like this:

```
Num of conflicts in total: 100

2 1 2  1 2 0  0 0 1
3 3 1  2 0 3  1 1 3
2 2 2  2 0 0  1 1 1

1 0 2  1 1 2  0 0 1
1 2 1  0 0 1  0 1 1
1 2 3  1 2 1  2 2 1

2 2 1  1 1 0  2 1 0
0 2 1  0 1 1  0 1 2
1 1 3  0 2 2  2 2 2
```

We can for example look at the number 2 in the top leftmost square (pos(0,0)) and see that the conflict matrix states that it has 2 conflicting 2s in total, we can easily verify this by finding the 2 on the same row in the top middle square (pos(0,4)), and the 6 on the same column that is located in the middle leftmost square (pos(5,0)).
In SA this information is enough to deduce the objective function as we need only count the conflicts on the board and give an appropriate value to such a state.
However in Minimum-Conflicts we can use this information more intelligently and look at the conflict matrix to see where we can make a change (I.e. each position where there are 1 or more conflicts), and then pick one of these at random.

The way we have opted to choose a neighboring state with use of Minimum-Conflicts is that for the chosen position we temporarily swap with each of the other positions inside the same square, and then tally up the number of conflicts this new position will yield in total (I.e. count both row and column collisions of both the new position with the chosen number and also the old position with the swapped value). This is done by keeping a squareConflict matrix represented such as this:

```
Y5X4
[1, 1, 1]
[4, 4, 4]
[1, 2, 4]
```

We see here that the position Y=5, X=4 has been chosen, i.e. the algorithm will traverse the middle square and find the position to swap with that yields the least conflicts. From the matrix we can see that the set (Y3X3, Y3X4, Y3X5, Y5X3) all yield a total of 1 conflict if swapped to (whereas the original Y5X4 yields 2). The algorithm arbitrarily choose to swap with Y3X5 (so that 9 and 4 switch places) and this is then the resulting board and conflict matrix:

```
The board

 2  6  4     4  2  8     3  7  1
 5  9  1     6  3  5     6  9  5
 7  8  3     7  1  9     2  4  8

 6  5  1     1  6  9     7  2  4
 7  9  4     2  8  5     1  3  6
 2  8  3     7  4  3     9  8  5

 5  9  2     6  5  1     2  8  7
 8  6  7     3  2  7     5  4  6
 4  1  3     8  9  4     9  3  1


Num of conflicts in total: 94

2 1 2  1 2 0  0 0 1
3 3 1  2 0 3  1 1 3
2 2 2  2 0 1  1 1 1

1 0 2  1 1 1  0 0 0
1 2 1  0 0 1  0 1 1
1 2 3  1 0 1  1 2 1

2 2 1  1 1 0  2 1 0
0 2 1  0 1 1  0 1 2
1 1 3  0 1 1  2 2 2
```

The byproduct of this swap is that not only does the swapped values reduce their own conflicts but those previously crashed with as well, thus we manage to reduce the number of total conflicts from 100 down to 94.

4 Demonstration of K-queen

From the assignment text k = 8 is easy, k = 25 is medium and k=1000 is the hard-case, but k = 2500 has a runtime of approximately 1 min by using Minimum-Conflicts.

Internally the board is represented as a one dimensional matrix where each index represents the column position for the given row. For example the board matrix:

```
[5,3,0,4,7,1,6,3]
```

means that on f.ex row 0 the queen is placed on column 5. The more readable representation (the printed board) would then look like this:

```
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
```

```
[0, 0, 0, 1, 0, 0, 0, 0]
```

Because we have opted to use a one dimensional matrix internally, the creation and alteration of a state is both easier and faster than if we had to keep track a bunch of 1s and 0s in a two dimensional matrix. Now we simply need to change the number for a given index and traverse the matrix to count up conflicts. The conflict matrix is thus also a one dimensional matrix where each index tells us how many crashes the queen at a given row has. For our given board it would look like this:

```
[0,1,0,0,0,1,0,2]
```

For SA we create children where we choose one random row and move the queen on that row to a random column. The idea here is to alter the states slightly in the hope of getting closer to a goal state. The way SA determines the value of a particullar state is by counting the crashes in total and giving a value based on this.
However if we run Minimum-Conflict, and on this iteration randomly choose to alter the position for the queen at row 7 we would have to check each position for the queen on that given row. For this purpose we have yet another one dimensional matrix which counts up how many conflicts the move would produce if we opted to move to a particullar column. For this row we would have the NumberOfColumnConflict matrix appearing like this:

```
[2,1,0,2,2,3,1,2]
```

Thus the algorithm would move the queen at row 7 from column 3 to 2 and thereby reaching a goal state. (Which is shown in the next section)
If we were to encounter a node where there are no options with fewer or the same number of conflicts no matter which column it goes to, the algorithm marks this row so that it will avoid trying the same row again on the next iteration. This is a small tweak meant to avoid getting locked on trying the same row several times without moving anywhere.

5 Result table of K-queens, and a description of an easy-case

One of the solutions found for the easy-case (k=8)

```
QueenManager k=8
The board
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

```
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
```

Final statistic for 20 runs with a maximum of 10000 iterations with min-conflict (there is no need to include average and standard deviation for evaluation as it solves everything each run. Though it does happend on a few occations that the easy problem gets stuck):

Hardness:	**Easy**	**Medium**	**Hard**
Average iterations reaching goal:	72.9	64.7	695.55
Average iterations per run:	72.9	64.7	695.55
Standard deviation for iterations σ	73	38	21.5
Fewest iteration:	9	20	659
Most iterations:	276	185	747
Number of goal reached:	20/20	20/20	20/20
Total iterations:	1458	1294	13911
Total iterations where goal is reached:	1294	2099	13911
Total time spent:	87ms	120ms	128924ms

Final statistic for 20 runs with a maximum of 10000 iterations with SA:

Hardness:	**Easy**	**Medium**	**Hard**
Average iterations reaching goal:	547.35	N/A	N/A
Average iterations per run:	447.35	10000	10000
Average F value:	1.0	0.772	0.50
Standard deviation of F:	0.0	0.05	0.012
Standard deviation for iterations σ	567.41	0	0
Fewest iterations:	9	N/A	N/A
Most iterations:	2323	N/A	N/A
Number of goal states reached:	20/20	0/20	0/2
Total iterations:	10947	200000	20000
Total iterations where goal is reached:	10947	N/A	N/A
Total time spent:	301ms	14957ms	1644710ms

The Hard problem for k-queens using the SA algorithm takes around 13-14 minutes, so we couldn't really do 20 runs of this one because it takes way too much time, that's why we decided to only run 2 runs of the hard problem to have some data to document. Due to the enormous search space SA struggles to find a good way to make a describing objective value when there is so many conflicts. Thus it has a problem with climbing closer to a solution.

6 Demonstration of Graph-color

For the graph problem we decided that the number 1 case is classified as easy, 2 is medium and 3 is hard-case. These numbers correspond to the txt file given inside the project - that is 1 corresponds to *graph-color-1.txt* etc. (do note that we swapped the numbers 1 and 2 of the provided files).
The first "easy" one have a 40 nodes/countries and 67 edges/countries bordering each other, the second medium one have 40 nodes and 97 edges whilst the hardest problem have 500 nodes with 1009 edges.
We decided to classify these problems as such because it is more logical that the more edges and nodes you have, the harder the problem becomes. It's also the only input we have for this problem. This problem is also known as vertex-coloring, which is a problem where you have to color every node with a color so that none of its neighbor have the same color.
The data we got is explained here:
http://www.idi.ntnu.no/emner/it3105/materials/data/graph-color-format.txt, as you can see, these nodes are provided with coordinates. These coordinates are mostly there for the visual representation if needed, it's not needed for the actual problem, this is because there's no need for information about **where** the nodes are located as long as we know how the nodes are connected through their edges.
For this task, we've decided to use a neighbor-matrix to represent the nodes and the edges, this is represented by a 2-dimensional boolean array where a *True* indicates an edge between a node i and node j ($i \neq j$) and an int array which contains the colors of each node. For instance, if we have 3 nodes which are all connected, we would have:
Matrix:

```
[0,1,1]
[1,0,1]
[1,1,0]
```

The nodes array which contains the colors could be:

```
[1,0,0]
```

(nodes where the first node is color 1 and the rest is color 0)
This would give us a conflict array which looked like this:

```
[0,1,1]
```

Which means that node 2 and 3 each have 1 conflict each with eachother, because both of them have a neighbour with the same color as itself. For SA we calculate the objective function (F) by calculating the number of crashes in total and using this formula: $F = 1 - \frac{crashes}{numberofNodes}$
For the minconflict algorithm we take one randomly chosen node given that the chosen node has at least one conflict. This randomly chosen node then calculates

how many conflicts it will have for the four colors and stores this in an array of this nodes conflicts. For the example above, the algorithm would choose either node 1 or node 2, let's say it chose node 1 it will create a conflictForEachColor array which looked like

```
[1,1,0,0]
```

This means that if it changed its color to color 0 it would have 1 conflict, 1 for color 1 and 0 for the color 3 and 4. The algorithm will then choose either color 3 or 4 for this node randomly. It will then have solved the problem for this case, but let's say all of the color would give us 1 conflict, that is conflictForEachColor array would look like:

```
[1,1,1,1]
```

This means that no matter what color it chooses, the conflict number would still be the same. The algorithm then choose a random color other than the initial one, let's say the node had color 0, the algorithm will chose either 1,2 or 3 randomly to create jiggle so the problem doesn't get tangled. If we meet a case where we choose a node and changing the color would cause the number of conflicts to increase, it will simply mark this node so the next iteration will ignore this node and change the color of another node instead. If a node did change color, the "lastNodeChecked" variable will be set to -1 and the all nodes can be checked in the next iteration.

7 Result table of Graph-Coloring, and the description of an easy-case

One of the runs and its solution printed out, the N indicates which node we are looking at and the C tells us which color this node is. We have used numbers to represent the colors because this is easier to work with.

```
GraphManager: graph-color-1.txt

Number of conflicts in total: 0
Conflicts: N0:0  N1:0  N2:0  N3:0  N4:0  N5:0  N6:0  N7:0  N8:0  N9:0
N10:0  N11:0  N12:0  N13:0  N14:0  N15:0  N16:0  N17:0  N18:0  N19:0  N20:0
N21:0  N22:0  N23:0  N24:0  N25:0  N26:0  N27:0  N28:0  N29:0  N30:0
N31:0  N32:0  N33:0  N34:0  N35:0  N36:0  N37:0  N38:0  N39:0
N = Node C = Color
N:0 C:2| N:1 C:0
N:1 C:0| N:0 C:2  N:2 C:3  N:3 C:2  N:4 C:1
N:2 C:3| N:1 C:0  N:3 C:2  N:4 C:1
N:3 C:2| N:1 C:0  N:2 C:3  N:4 C:1
```

```
N:4 C:1| N:1 C:0  N:2 C:3  N:3 C:2  N:5 C:0
N:5 C:0| N:4 C:1  N:6 C:3
N:6 C:3| N:5 C:0  N:7 C:0  N:8 C:1
N:7 C:0| N:6 C:3  N:8 C:1  N:9 C:2  N:10 C:3
N:8 C:1| N:6 C:3  N:7 C:0  N:9 C:2  N:10 C:3
N:9 C:2| N:7 C:0  N:8 C:1  N:10 C:3  N:11 C:1
N:10 C:3| N:7 C:0  N:8 C:1  N:9 C:2  N:11 C:1  N:12 C:0
N:11 C:1| N:9 C:2  N:10 C:3  N:12 C:0
N:12 C:0| N:10 C:3  N:11 C:1  N:13 C:2
N:13 C:2| N:12 C:0  N:14 C:3
N:14 C:3| N:13 C:2  N:15 C:1  N:16 C:0
N:15 C:1| N:14 C:3  N:16 C:0  N:18 C:3
N:16 C:0| N:14 C:3  N:15 C:1  N:17 C:2  N:18 C:3  N:19 C:1
N:17 C:2| N:16 C:0  N:18 C:3  N:19 C:1
N:18 C:3| N:15 C:1  N:16 C:0  N:17 C:2  N:19 C:1  N:20 C:2
N:19 C:1| N:16 C:0  N:17 C:2  N:18 C:3  N:20 C:2  N:22 C:3
N:20 C:2| N:18 C:3  N:19 C:1  N:21 C:1  N:22 C:3
N:21 C:1| N:20 C:2  N:22 C:3
N:22 C:3| N:19 C:1  N:20 C:2  N:21 C:1  N:23 C:0
N:23 C:0| N:22 C:3  N:24 C:2  N:26 C:1
N:24 C:2| N:23 C:0  N:25 C:0  N:26 C:1
N:25 C:0| N:24 C:2  N:26 C:1  N:27 C:3
N:26 C:1| N:23 C:0  N:24 C:2  N:25 C:0  N:27 C:3
N:27 C:3| N:25 C:0  N:26 C:1  N:28 C:1  N:29 C:0  N:30 C:2
N:28 C:1| N:27 C:3  N:29 C:0  N:30 C:2  N:31 C:3
N:29 C:0| N:27 C:3  N:28 C:1  N:30 C:2  N:31 C:3
N:30 C:2| N:27 C:3  N:28 C:1  N:29 C:0  N:31 C:3
N:31 C:3| N:28 C:1  N:29 C:0  N:30 C:2  N:32 C:1
N:32 C:1| N:31 C:3  N:33 C:3  N:34 C:0
N:33 C:3| N:32 C:1  N:34 C:0  N:35 C:2
N:34 C:0| N:32 C:1  N:33 C:3  N:35 C:2
N:35 C:2| N:33 C:3  N:34 C:0  N:36 C:0
N:36 C:0| N:35 C:2  N:37 C:1
N:37 C:1| N:36 C:0  N:38 C:2  N:39 C:3
N:38 C:2| N:37 C:1  N:39 C:3
N:39 C:3| N:37 C:1  N:38 C:2
```

Final statistic for 20 runs with a maximum of 10000 iterations with min-conflict (there is no need to include average and standard deviation for evaluation as it solves everything each run):

Hardness:	**Easy**	**Medium**	**Hard**
Average iterations reaching goal:	21.45	104.95	646.4
Average iterations per run:	21.45	104.95	646.4
Standard deviation for iterations σ	10.14	42.96	67.16
Fewest iterations:	11	45	493
Most iterations:	59	191	766
Number of goal states reached:	20/20	20/20	20/20
Total iterations:	429	2099	12928
Total iterations for goal states:	429	2099	12928
Total time spent:	97ms	149ms	9336ms

Final statistic for 20 runs with a maximum of 10000 iterations with SA:

Hardness:	**Easy**	**Medium**	**Hard**
Average iterations reaching goal:	9004.55	9417.5	N/A
Average iterations per run:	9004.55	9941.75	10000
Average F value:	1.0	0.74	0.69
Standard deviation of F:	0.0	0.13	0.02
Standard deviation for iterations σ	386.74	213.98	0
Fewest iteration:	8158	9027	N/A
Most iterations:	9924	9808	N/A
Number of goal states reached:	20/20	2/20	0/20
Total iterations:	180091	198835	200000
Total iterations for goal states:	18835	2099	N/A
Total time spent:	22976ms	28580ms	2873082ms

8 Demonstration of the modified Sudoku

We have decided that k = 2 is easy case, 3 is medium, 9 is hard. We chose to use these *k*s because of the runtime and complexity in the search space. The hard problem runs in around 30 second per run with Min-Conflict, and the next k, k = 10, uses about 90 seconds with the Min-Conflict algorithm. That's why we decided that the hard problem sould be k = 9 because it already takes WAY too long per run with the SA algorithm, and we want to be able to deliver in time.

For a detailed description of how this problem was solved internally see section 3.

9 Result table of modified Sudoku, and description of an easy-case

A solution to the easy-case:

```
The board
```

```
4 2    1 3
1 3    2 4

3 1    4 2
2 4    3 1
```

Final statistic for 20 runs with a maximum of 10000/10000/30000 iterations (easy,medium,hard respectivly. Hard requires more iterations due to the size of the search space, but its still quite fast) with Minimum Conflicts (Solves everything so no need for evalution fields):

Hardness:	**Easy**	**Medium**	**Hard**
Average iterations reaching goal:	8.45	79.65	18285
Average iterations per run:	8.45	79.65	18285
Standard deviation for iterations σ	4.92	26.52	1205.73
Fewest iterations:	2	35	16503
Most iterations:	19	147	21297
Number of goal states reached:	20/20	20/20	20/20
Total iterations:	169	1593	365700
Total iterations goal states:	169	1593	365700
Total time spent:	19ms	113ms	503847ms

SA: (we've taken 20000 as the maximum iteration for the medium one, because it's still fast enough and there might be a chance of solving it)

Hardness:	**Easy**	**Medium**	**Hard**
Average iterations reaching goal:	15.8	N/A	N/A
Average iterations per run:	15.8	20000	10000
Average F value:	1.0	0.86	0.800
Standard deviation of F:	0.0	0.028	0.0013
Standard deviation for iterations σ	15.47	0	0
Fewest iterations:	1	N/A	N/A
Most iterations:	48	N/A	N/A
Number of goal reached:	20/20	0/20	0/10
Total iterations:	316	400000	100000
Total iterations for goal states:	316	N/A	N/A
Total time spent:	71ms	54755ms	3038673ms

We opted to only run the SA algorithm 10 times on the hard problem due to the very long runtime. It struggles to find a solution even on the medium problem. This algorithm just is not cut out to find a solution in such a big and diverse search space. It may find a somewhat decent solution but it has a very hard time converging onto a goal state.