McGill University

Homework 1

Nabil Chowdhury

ID: 260622155

COMP 424

January 31, 2018

1 Question 1: Six Puzzle

1.1 PART A: Solution paths for various search algorithms

The paths were all found programmatically using Java. Please see Question1.java for implementation details.
Note: As specified by the assignment, neighboring states are explored by lowest-valued piece moved. For example, if the valid moves are moving 2 and 4, then moving the 2 is explored first.

1.1.1 Breadth-First Search

[[1, 4, 2][5, 3, 0]] —> [[1, 4, 2][5, 0, 3]] —> [[1, 0, 2][5, 4, 3]] —> [[0, 1, 2][5, 4, 3]]

1.1.2 Uniform Cost Search

[[1, 4, 2][5, 3, 0]] —> [[1, 4, 2][5, 0, 3]] —> [[1, 0, 2][5, 4, 3]] —> [[0, 1, 2][5, 4, 3]]

1.1.3 Depth-First Search

[[1, 4, 2][5, 3, 0]] —> [[1, 4, 0][5, 3, 2]] —> [[1, 0, 4][5, 3, 2]] —> [[0, 1, 4][5, 3, 2]] —> [[5, 1, 4][0, 3, 2]] —> [[5, 1, 4][3, 0, 2]] —> [[5, 0, 4][3, 1, 2]] —> [[5, 4, 0][3, 1, 2]] —> [[5, 4, 2][3, 1, 0]] —> [[5, 4, 2][3, 0, 1]] —> [[5, 4, 2][0, 3, 1]] —> [[0, 4, 2][5, 3, 1]] —> [[4, 0, 2][5, 3, 1]] —> [[4, 2, 0][5, 3, 1]] —> [[4, 2, 1][5, 3, 0]] —> [[4, 2, 1][5, 0, 3]] —> [[4, 0, 1][5, 2, 3]] —> [[4, 1, 0][5, 2, 3]] —>

[[4, 1, 3][5, 2, 0]] —> [[4, 1, 3][5, 0, 2]] —> [[4, 0, 3][5, 1, 2]] —> [[4, 3, 0][5, 1, 2]] —> [[4, 3, 2][5, 1, 0]] —> [[4, 3, 2][5, 0, 1]] —> [[4, 3, 2][0, 5, 1]] —> [[0, 3, 2][4, 5, 1]] —> [[3, 0, 2][4, 5, 1]] —> [[3, 2, 0][4, 5, 1]] —> [[3, 2, 1][4, 5, 0]] —> [[3, 2, 1][4, 0, 5]] —> [[3, 0, 1][4, 2, 5]] —> [[3, 1, 0][4, 2, 5]] —> [[3, 1, 5][4, 2, 0]] —> [[3, 1, 5][4, 0, 2]] —> [[3, 0, 5][4, 1, 2]] —> [[0, 3, 5][4, 1, 2]] —> [[4, 3, 5][0, 1, 2]] —> [[4, 3, 5][1, 0, 2]] —> [[4, 3, 5][1, 2, 0]] —> [[4, 3, 0][1, 2, 5]] —> [[4, 0, 3][1, 2, 5]] —> [[4, 2, 3][1, 0, 5]] —> [[4, 2, 3][0, 1, 5]] —> [[0, 2, 3][4, 1, 5]] —> [[2, 0, 3][4, 1, 5]] —> [[2, 1, 3][4, 0, 5]] —> [[2, 1, 3][0, 4, 5]] —> [[0, 1, 3][2, 4, 5]] —> [[1, 0, 3][2, 4, 5]] —> [[1, 3, 0][2, 4, 5]] —> [[1, 3, 5][2, 4, 0]] —> [[1, 3, 5][2, 0, 4]] —> [[1, 3, 5][0, 2, 4]] —> [[0, 3, 5][1, 2, 4]] —> [[3, 0, 5][1, 2, 4]] —> [[3, 2, 5][1, 0, 4]] —> [[3, 2, 5][0, 1, 4]] —> [[0, 2, 5][3, 1, 4]] —> [[2, 0, 5][3, 1, 4]] —> [[2, 1, 5][3, 0, 4]] —> [[2, 1, 5][0, 3, 4]] —> [[0, 1, 5][2, 3, 4]] —> [[1, 0, 5][2, 3, 4]] —> [[1, 5, 0][2, 3, 4]] —> [[1, 5, 4][2, 3, 0]] —> [[1, 5, 4][2, 0, 3]] —> [[1, 5, 4][0, 2, 3]] —> [[0, 5, 4][1, 2, 3]] —> [[5, 0, 4][1, 2, 3]] —> [[5, 2, 4][1, 0, 3]] —> [[5, 2, 4][0, 1, 3]] —> [[0, 2, 4][5, 1, 3]] —> [[2, 0, 4][5, 1, 3]] —> [[2, 1, 4][5, 0, 3]] —> [[2, 1, 4][5, 3, 0]] —> [[2, 1, 0][5, 3, 4]] —> [[2, 0, 1][5, 3, 4]] —> [[0, 2, 1][5, 3, 4]] —> [[5, 2, 1][0, 3, 4]] —> [[5, 2, 1][3, 0, 4]] —> [[5, 0, 1][3, 2, 4]] —> [[0, 5, 1][3, 2, 4]] —> [[3, 5, 1][0, 2, 4]] —> [[3, 5, 1][2, 0, 4]] —> [[3, 5, 1][2, 4, 0]] —> [[3, 5, 0][2, 4, 1]] —> [[3, 0, 5][2, 4, 1]] —> [[0, 3, 5][2, 4, 1]] —> [[2, 3, 5][0, 4, 1]] —> [[2, 3, 5][4, 0, 1]] —> [[2, 0, 5][4, 3, 1]] —> [[0, 2, 5][4, 3, 1]] —> [[4, 2, 5][0, 3, 1]] —> [[4, 2, 5][3, 0, 1]] —> [[4, 2, 5][3, 1, 0]] —> [[4, 2, 0][3, 1, 5]] —> [[4, 0, 2][3, 1, 5]] —> [[4, 1, 2][3, 0, 5]] —> [[4, 1, 2][0, 3, 5]] —> [[0, 1, 2][4, 3, 5]] —> [[1, 0, 2][4, 3, 5]] —> [[1, 2, 0][4, 3, 5]] —> [[1, 2, 5][4, 3, 0]] —> [[1, 2, 5][4, 0, 3]] —> [[1, 0, 5][4, 2, 3]] —> [[0, 1, 5][4, 2, 3]] —> [[4, 1, 5][0, 2, 3]] —> [[4, 1, 5][2, 0, 3]] —> [[4, 0, 5][2, 1, 3]] —> [[0, 4, 5][2, 1, 3]] —> [[2, 4, 5][0, 1, 3]] —> [[2, 4, 5][1, 0, 3]] —> [[2, 4, 5][1, 3, 0]] —> [[2, 4, 0][1, 3, 5]] —> [[2, 0, 4][1, 3, 5]] —> [[0, 2, 4][1, 3, 5]] —> [[1, 2, 4][0, 3, 5]] —> [[1, 2, 4][3, 0, 5]] —> [[1, 0, 4][3, 2, 5]] —> [[0, 1, 4][3, 2, 5]] —> [[3, 1, 4][0, 2, 5]] —> [[3, 1, 4][2, 0, 5]] —> [[3, 0, 4][2, 1, 5]] —> [[0, 3, 4][2, 1, 5]] —> [[2, 3, 4][0, 1, 5]] —> [[2, 3, 4][1, 0, 5]] —> [[2, 3, 4][1, 5, 0]] —> [[2, 3, 0][1, 5, 4]] —> [[2, 0, 3][1, 5, 4]] —> [[0, 2, 3][1, 5, 4]] —> [[1, 2, 3][0, 5, 4]] —> [[1, 2, 3][5, 0, 4]] —> [[1, 0, 3][5, 2, 4]] —> [[0, 1, 3][5, 2, 4]] —> [[5, 1, 3][0, 2, 4]] —> [[5, 1, 3][2, 0, 4]] —> [[5, 0, 3][2, 1, 4]] —> [[5, 3, 0][2, 1, 4]] —> [[5, 3, 4][2, 1, 0]] —> [[5, 3, 4][2, 0, 1]] —> [[5, 3, 4][0, 2, 1]] —> [[0, 3, 4][5, 2, 1]] —> [[3, 0, 4][5, 2, 1]] —> [[3, 2, 4][5, 0, 1]] —> [[3, 2, 4][5, 1, 0]] —> [[3, 2, 0][5, 1, 4]] —> [[3, 0, 2][5, 1, 4]] —> [[3, 1, 2][5, 0, 4]] —> [[3, 1, 2][5, 4, 0]] —> [[3, 1, 0][5, 4, 2]] —> [[3, 0, 1][5, 4, 2]] —> [[0, 3, 1][5, 4, 2]] —> [[5, 3, 1][0, 4, 2]] —> [[5, 3, 1][4, 0, 2]] —> [[5, 3, 1][4, 2, 0]] —> [[5, 3, 0][4, 2, 1]] —> [[5, 0, 3][4, 2, 1]] —> [[5, 2, 3][4, 0, 1]] —> [[5, 2, 3][4, 1, 0]] —> [[5, 2, 0][4, 1, 3]] —> [[5, 0, 2][4, 1, 3]] —> [[5, 1, 2][4, 0, 3]] —> [[5, 1, 2][0, 4, 3]] —> [[0, 1, 2][5, 4, 3]]

1.1.4 Iterative Deepening Search

Iterative Deepening Search does not find solutions for depths 0 to 3. This makes sense since the shortest solution path is at depth 4.

[[1, 4, 2][5, 3, 0]] —> [[1, 4, 2][5, 0, 3]] —> [[1, 0, 2][5, 4, 3]] —> [[0, 1, 2][5, 4, 3]]

1.2 Part B: Is Manhattan distance still admissible for the new cost scheme?

From class, we know that the Manhattan distance is admissible under the unit cost scheme where moving to a neighboring state costs 1. Under this new cost scheme, moving to a neighboring state costs the value of the piece moved. The lowest piece on the board is 1, and so the Manhattan distance heuristic is still admissible since the new cost scheme ≥ the unit cost scheme, since it is lower bounded by the unit cost scheme. Mathematically, we have the following:

h(n) = Manhattan distance heuristic
u(n) = unit cost scheme
u'(n) = new cost scheme
S = The state space

$$h(n) <= u(n) <= u'(n)\ \forall n \in S$$

1.3 Part C: Design an admissible heuristic that dominates the heuristic from Part B under the same cost scheme

We say that a heuristic $h_2(n)$ dominates $h_1(n)$ if $h_2(n) \geq h_1(n)\ \forall n$ and both heuristics are admissible. The Manhattan distance of a piece from its current state to goal state can be multiplied by the value of the piece to get a heuristic that dominates the original heuristic.

Original Manhattan distance heuristic:
Let P be the highest valued piece on the board (i.e. the board has piece from 1 to P).
Let g_{xi} be the x coordinate of piece i in the goal board state.
Let g_{yi} be the y coordinate of piece i in the goal board state.
Let c_{xi} be the x coordinate of piece i in the current board state.
Let c_{yi} be the y coordinate of piece i in the current board state.

$$\sum_{i=1}^{P} |g_{xi} - c_{xi}| + |g_{yi} - c_{yi}| = h$$

New dominant heuristic:

$$\sum_{i=1}^{P} i \times (|g_{xi} - c_{xi}| + |g_{yi} - c_{yi}|) = h'$$

This new heuristic dominates the old heuristic for 2 reasons:
i) $h' \geq h$ since we are multiplying by $i \geq 1$
ii) h' is still admissible under the new cost scheme since we know in the unit cost case, the Manhattan distance is a lower bound on the actual cost. Similarly, the new cost scheme poses a greater cost than h' since we cannot actually move a piece directly to its goal state since there may be other pieces in the way. Thus, more moves are required than what Manhattan distance assumes.

1.4 Part D: New cost scheme in which moving pieces to the left or right costs 2, whereas moving pieces up or down costs 0.5

Under this new cost scheme, it is easy to find an example showing that the Manhattan distance is not admissible:

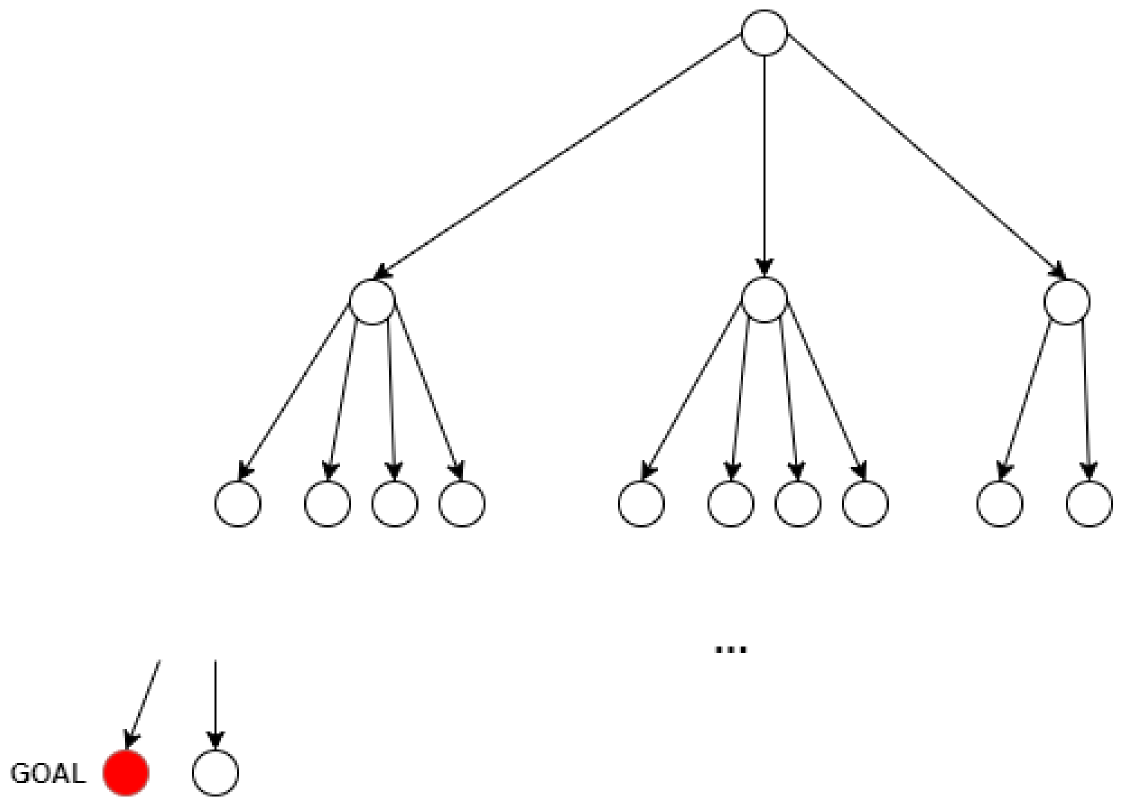
[[5, 1, 2] [0, 4, 3]] – 0.5 –> [[0, 1, 2] [5, 4, 3]]

In the above example, the goal state can be reached by moving the 5 down into the empty cell. The actual cost is 0.5 whereas the Manhattan distance cost is just 1. Therefore, there exists a state where $h(n) \geq cost(n)$, and so *h(n)* is not admissible.

2 Question 2: Search Algorithms

2.1 Part A: Describe a state space in which iterative deepening search performs much worse than depth-first search

Consider a state space where the goal state is found by DFS without the need for branching. In other words, the goal state is at a large depth *d*, but DFS can find it in linear time (see diagram below). In such a case, iterative deepening will perform much worse than DFS since it will not get to depth *d* until it fully exhausts the previous *d - 1* levels, whereas DFS, with the right choice of neighbors, can get there in linear time. Here is an example of such a search tree:



2.2 Part B: T/F: Breadth-first search is a special case of uniform-cost search/

True. BFS is Uniform-cost search with unit cost (transition cost is 1). This can be seen by noticing that BFS explores the search tree layer by layer (it explores all nodes at depth d before exploring depth $d + 1$). Uniform cost search uses a priority queue where the cost so far is the priority. With a transition cost of 1, all neighbors are pushed to the back of the priority queue since the cost of all neighbors is the cost of the current node plus 1, and this emulates standard BFS.

2.3 Part C: T/F: Depth-first search is a special case of best-first tree search

True. Best-first tree search becomes the standard Depth-first search if the heuristic $h(n) = -depth(n)$. The child n' of a node n with depth $depth(n)$, has depth $depth(n) + 1$. Evaluating $h(n')$ gives us $-(depth(n) + 1)$. Since $h(n') < h(n)$, n' is now at the front of the priority queue, and will be explored next.

2.4 Part D: T/F: Uniform cost search is a special case of A* search

True. A* search is a greedy search with respect to the function $f(n) = g(n) + h(n)$, where g is the cost of the path so far, and h is the admissible heuristic. Since A* expands nodes with the lowest f first, we can set $h(n) = 0 \; \forall \; n$. Now, f is simply the cost of the path so far, and this is exactly uniform cost search.

3 QUESTION 3: OPTIMIZATION

3.1 PART A: HILL CLIMBING

Step size	**X_0**	**Iterations**	**X***	**Y***
0.01	0	175	1.7400000000000013	0.3960158890993247
0.01	1	75	1.7400000000000007	0.39601588909932467
0.01	2	27	1.7399999999999998	0.3960158890993246
0.01	3	127	1.740000000000021	0.3960158890993246
0.01	4	5	3.960000000000001	0.3341097537786864
0.01	5	33	5.319999999999993	0.31049661129613665
0.01	6	40	6.389999999999992	0.2961072260386306
0.01	7	32	7.309999999999993	0.28572281037034464
0.01	8	13	8.119999999999997	0.2777774823660866
0.01	9	15	8.860000000000003	0.27132890618876104
0.01	10	1	10	-0.06891263333078856
0.02	0	88	1.740000000000001	0.39601588909932467
0.02	1	38	1.7400000000000007	0.39601588909932467
0.02	2	14	1.7399999999999998	0.3960158890993246
0.02	3	64	1.7399999999999989	0.3960158890993247
0.02	4	3	3.96	0.3341097537786864
0.02	5	17	5.319999999999993	0.31049661129613665
0.02	6	20	6.379999999999992	0.29554413206295477
0.02	7	16	7.299999999999994	0.28536751773684105
0.02	8	7	8.119999999999997	0.2777774823660866
0.02	9	8	8.860000000000003	0.27132890618876104
0.02	10	1	10	-0.06891263333078856
0.03	0	59	1.7400000000000013	0.3960158890993247
0.03	1	26	1.7500000000000007	0.39595546426910133
0.03	2	10	1.7299999999999998	0.3959593690679719
0.03	3	43	1.7400000000000062	0.3960158890993246
0.03	4	2	3.97	0.33381970937891065
0.03	5	12	5.330000000000003	0.30967579571438547
0.03	6	14	6.390000000000003	0.29610722603863054
0.03	7	11	7.3000000000000025	0.285367517736842
0.03	8	5	8.119999999999997	0.2777774823660866
0.03	9	6	8.850000000000003	0.26986547867397453
0.03	10	1	10	-0.06891263333078856
0.04	0	44	1.7200000000000009	0.395787755789096
0.04	1	19	1.7200000000000006	0.395787755789096
0.04	2	8	1.7199999999999998	0.39578775578909586
0.04	3	33	1.7199999999999989	0.3957877557890959
0.04	4	2	3.96	0.3341097537786864
0.04	5	9	5.32	0.3104966112961364
0.04	6	11	6.4	0.2954619405059111
0.04	7	9	7.32	0.28455196632004937
0.04	8	4	8.119999999999997	0.2777774823660866
0.04	9	4	8.880000000000003	0.267870590383232
0.04	10	1	10	-0.06891263333078856
0.05	0	36	1.7500000000000009	0.39595546426910133
0.05	1	16	1.7500000000000007	0.39595546426910133
0.05	2	6	1.7499999999999998	0.3959554642691013
0.05	3	26	1.7500000000000033	0.3959554642691012
0.05	4	2	3.95	0.3338766568306922
0.05	5	7	5.299999999999999	0.30950706404208217
0.05	6	9	6.399999999999999	0.2954619405059114

Step size	**X$_0$**	**Iterations**	**X***	**Y***
0.05	7	7	7.299999999999999	0.2853675177368416
0.05	8	3	8.100000000000001	0.2734366204794694
0.05	9	4	8.849999999999998	0.26986547867397315
0.05	10	1	10	-0.06891263333078856
0.06	0	30	1.740000000000001	0.39601588909932467
0.06	1	13	1.7200000000000006	0.395787755789096
0.06	2	5	1.7599999999999998	0.395776269411191
0.06	3	22	1.7399999999999989	0.3960158890993247
0.06	4	2	3.94	0.3331241131595326
0.06	5	6	5.299999999999998	0.3095070640420821
0.06	6	7	6.359999999999998	0.2908255095935611
0.06	7	6	7.299999999999998	0.2853675177368415
0.06	8	3	8.120000000000001	0.27777748236608674
0.06	9	3	8.879999999999999	0.26787059038323324
0.06	10	1	10	-0.06891263333078856
0.07	0	26	1.750000000000001	0.39595546426910133
0.07	1	12	1.7700000000000007	0.39547650773131193
0.07	2	5	1.7199999999999998	0.39578775578909586
0.07	3	19	1.740000000000002	0.39601588909932467
0.07	4	2	3.93	0.331856611048492
0.07	5	6	5.350000000000001	0.3053941220963408
0.07	6	7	6.420000000000002	0.2905430641005972
0.07	7	5	7.280000000000001	0.28011470568142693
0.07	8	3	8.14	0.27480991554259754
0.07	9	3	8.86	0.2713289061887609
0.07	10	1	10	-0.06891263333078856
0.08	0	23	1.7600000000000005	0.395776269411191
0.08	1	10	1.7200000000000006	0.395787755789096
0.08	2	4	1.7599999999999998	0.395776269411191
0.08	3	17	1.7199999999999989	0.3957877557890959
0.08	4	2	3.92	0.3300794122070053
0.08	5	5	5.32	0.3104966112961364
0.08	6	6	6.4	0.2954619405059111
0.08	7	5	7.32	0.28455196632004937
0.08	8	3	8.16	0.264568277560923
0.08	9	3	8.84	0.2662912365420045
0.08	10	1	10	-0.06891263333078856
0.09	0	20	1.7100000000000004	0.3955029254359734
0.09	1	9	1.7200000000000006	0.395787755789096
0.09	2	4	1.7299999999999998	0.3959593690679719
0.09	3	15	1.7400000000000013	0.3960158890993247
0.09	4	1	4	0.32978608220779393
0.09	5	5	5.359999999999999	0.30193730078371744
0.09	6	5	6.359999999999999	0.2908255095935618
0.09	7	4	7.27	0.27525524691533304
0.09	8	2	8.09	0.2685674597141708
0.09	9	3	8.82	0.25296140148905594
0.09	10	1	10	-0.06891263333078856

Step size	X_0	Iterations	X*	Y*
0.1	0	18	1.7000000000000004	0.3951067768881972
0.1	1	8	1.7000000000000006	0.3951067768881972
0.1	2	4	1.6999999999999997	0.3951067768881971
0.1	3	14	1.6999999999999988	0.3951067768881971
0.1	4	1	4	0.32978608220779393
0.1	5	4	5.299999999999999	0.30950706404208217
0.1	6	5	6.399999999999999	0.2954619405059114
0.1	7	4	7.299999999999999	0.2853675177368416
0.1	8	2	8.1	0.27343662047946876
0.1	9	2	8.9	0.2559781881712985
0.1	10	1	10	-0.06891263333078856

The table above shows all possible combinations of step sizes and starting points for hill climbing. The maximum value achieved was X* = 1.7400000000000013 and Y* = 0.3960158890993247, using a step of 0.01. It started at $X_0 = 0$ and took 175 iterations to converge. The starting point had a clear impact on whether or not the algorithm would converge to this global maximum. Starting points of 0, 1, 2, or 3, regardless of the step size, yielded the global maximum whereas any other starting point caused the algorithm to get stuck at a local maximum. We can also notice, for higher step sizes, the accuracy deviates slightly. For example, looking at the results of step size 0.1 with starting points 0 to 3, the reported X* and Y* are slightly off from the actual maximum. This is because a large step size causes the algorithm to overshoot its choice of neighbor.

3.2 Part B: Simulated Annealing

Based on the results from Part A, step sizes of 0.01 and 0.02 were chosen for simulated annealing. This is because these step sizes produced near-optimal results (finding the global maximum) for starting points from 0 to 3. From the chart, we can see that for step sizes 0.03 onwards, the performance degrades and we no longer get roughly 1.74 for starting points from 0 to 3.

Step = 0.01, α = 0.99, T_0 = 20

X_0	Iterations	X*	Y*
0	1674	1.7400000000000013	0.3960158890993247
1	1674	1.7400000000000007	0.39601588909932467
2	1674	1.7399999999999995	0.3960158890993246
3	1674	1.740000000000021	0.3960158890993246
4	1674	3.960000000000001	0.3341097537786864
5	1674	5.319999999999993	0.31049661129613665
6	1674	6.389999999999992	0.2961072260386306
7	1674	7.309999999999993	0.28572281037034464
8	1674	8.119999999999997	0.2777774823660866
9	1674	8.860000000000003	0.27132890618876104
10	1674	9.54000000000001	0.2657197296699736

The α is the ratio by which T decreases each iteration i.e. T = T * α. We can immediately see that simulated annealing finds the global maximum (just like Hill Climbing). Now, let's use α = 0.75.

Step = 0.01, α = 0.75, T_0 = 20

X_0	Iterations	X*	Y*
0	60	0.11999999999999998	0.003524812254538553
1	60	1.1800000000000002	0.27025688518490515
2	60	1.7999999999999998	0.39383632075494984
3	60	2.860000000000003	-0.29240837026144856
4	60	3.960000000000001	0.3341097537786864
5	60	5.169999999999996	0.2239706578239484
6	60	6.249999999999995	0.18766928685830425
7	60	7.199999999999996	0.20325235331422675
8	60	8.119999999999997	0.2777774823660866
9	60	8.860000000000003	0.27132890618876104
10	60	10	-0.06891263333078856

The maximum found here is Y* = 0.3938 at X* = 1.7999. This means that decreasing the temperature at a faster rate means sometimes not achieving the best result.

The purpose of simulated annealing is to escape local maximas, and here is an example of that:

Step = 0.02, α = 0.999, T_0 = 10000

X_0	Iterations	X*	Y*
0	23016	1.740000000000001	0.39601588909932467
1	23016	1.7400000000000007	0.39601588909932467
2	23016	1.7399999999999998	0.3960158890993246
3	23016	1.7399999999999989	0.3960158890993247
4	23016	3.9599999999999995	0.3341097537786864
5	23016	1.7400000000000193	0.39601588909932467
6	23016	6.379999999999992	0.29554413206295477
7	23016	5.320000000000036	0.31049661129613504
8	23016	6.3800000000000345	0.2955441320629598
9	23016	8.120000000000019	0.2777774823660874
10	23016	8.12000000000004	0.2777774823660881

We can see that for X_0 = 5, it escapes its local maximum (unlike Hill climbing which gets stuck), and finds the global maximum. To achieve this however, a much larger T value and a larger decay rate α was required, so that T decays much slower per iteration.

4 Constraint Satisfaction

4.1 Part A: Formulate this problem as a Constraint Satisfaction problem. Describe the set of variables, and their domain. List the set of constraints

Let us put one rook per column. Then the values become the row of each rook. We have:

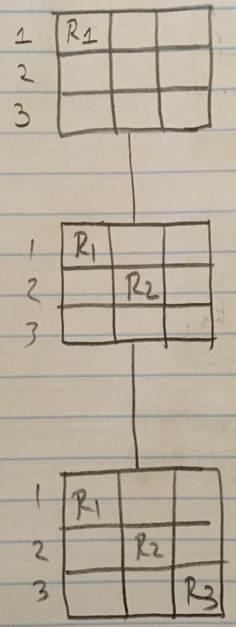
Variables: $\{R_1, R_2, ..., R_k\}$
Domain: $\{1, 2, ..., N\}$
Constraints: $\{R_i \neq R_j \ \forall i, j \leq k\}$

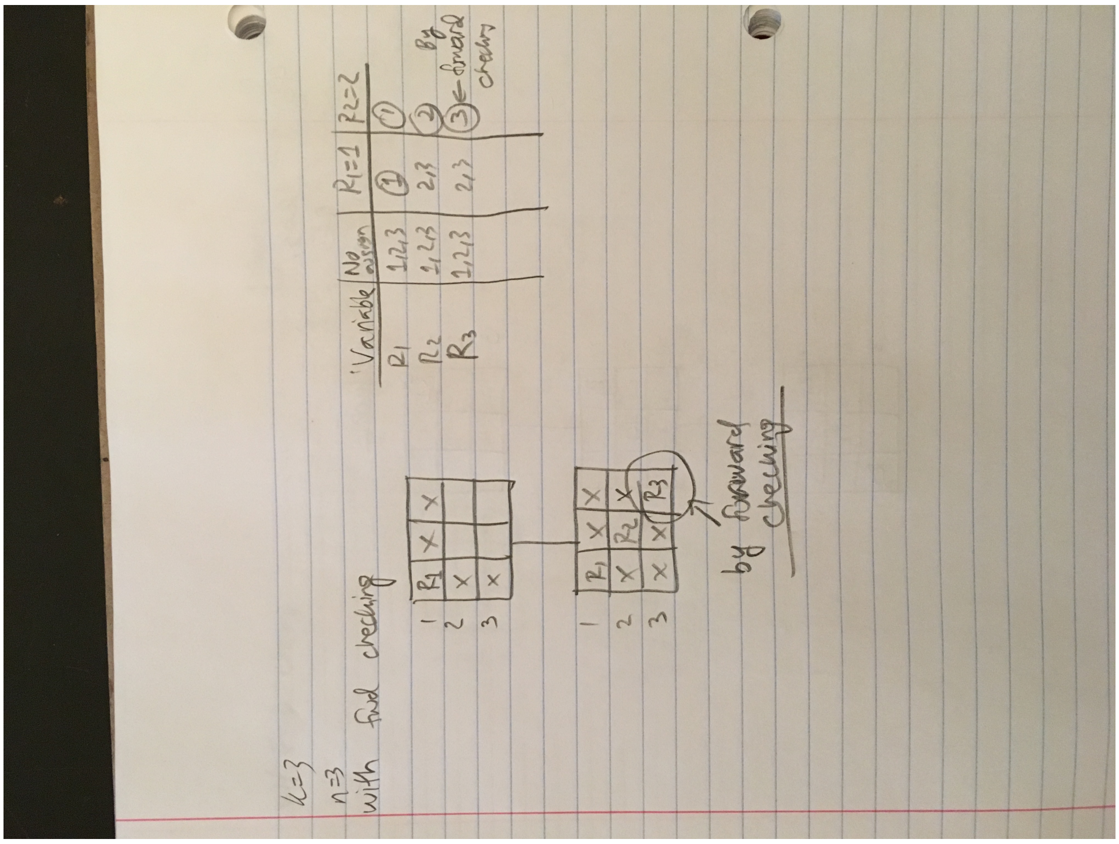
Basically, rooks cannot be on the same row as each other or else they would attack each other. This is similar to the N-Queens example from class, without the diagonal constraint since rooks do not attack diagonally. We also notice the assignment poses that $k \leq n^2$. Although it is impossible to have a solution to $k > n$, this does not affect the CSP since our domain is only 1 through n. By the Pigeonhole principle, we must have 2 or more rooks assigned to the same value if $k > n$, but the constraints prevent that. This means that there is no solution, and this is fine since for CSPs, we want to find any solution or find that there is no solution.

4.2 Part B: Show the search tree generated with applying backtracking search, without forward checking for the problem with k = 3 and n = 3. Show clearly the domain for each variable at every point in the search.

Without forward checking, we get the following:

4b) $k=3$,
$n=3$
no forward checking





k=3
n=3
with fwd checking

Variable	No assign	R1=1	R2=2
R1	1,2,3	①	①
R2	1,2,3	2,3	②
R3	1,2,3	2,3	③ ← By forward checking

With forward checking we can immediately assign R3 without requiring an additional call.