CAP 6635 Artificial Intelligence
Homework 2 | Matthew Acs

---

1. [**3 pts**] Figure 1 shows a robot navigation field, where the red square (d2) is the robot, and green square (b7) is the goal. The shad squares (such as b2, c2, etc.) are obstacles. The robot is not allowed to move in diagonal line. Nodes are coded using an alphabet letter followed by a digit (such as a0, b1, b2 etc.).
   - Use Depth First Search to find path from d2 to b7.
     - Report nodes in the fringe in the orders they are included in the fringe. [0.5 pt]
     - Report the order of the nodes being expanded. [0.5 pt]
     - Report the final path from d2 to b7. [0.5 pt]
   - Use Breadth First Search to find path from d2 to b7.
     - Report nodes in the fringe in the orders they are included in the fringe. [0.5 pt]
     - Report the order of the nodes being expanded. [0.5 pt]
     - Report the final path from d2 to b7. [0.5 pt]

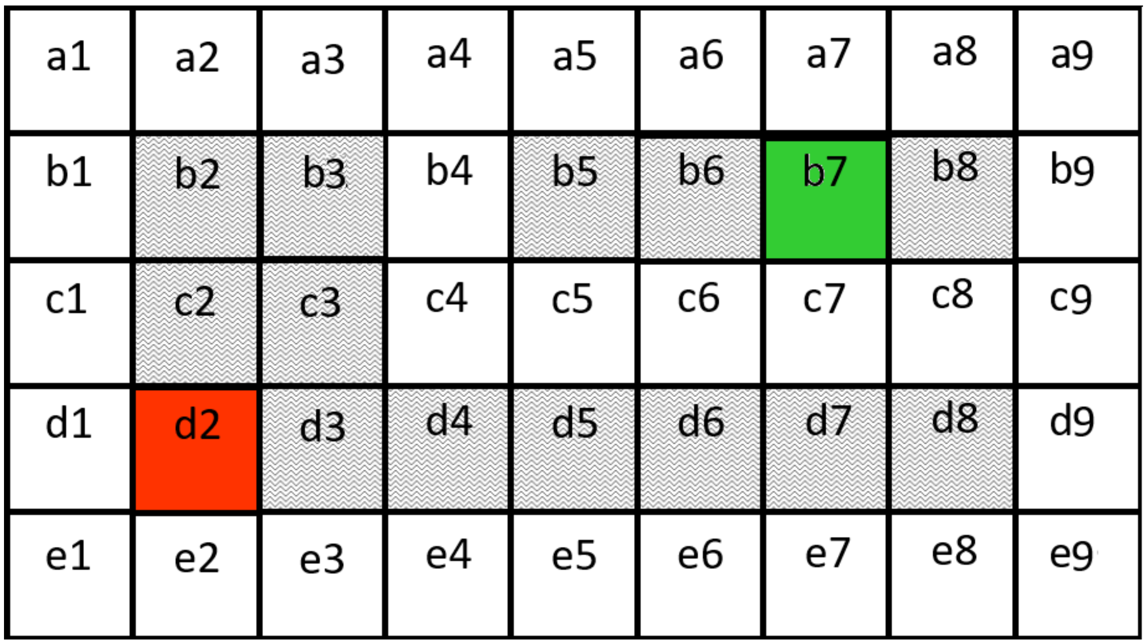

a1 a2 a3 a4 a5 a6 a7 a8 a9
b1 b2 b3 b4 b5 b6 b7 b8 b9
c1 c2 c3 c4 c5 c6 c7 c8 c9
d1 d2 d3 d4 d5 d6 d7 d8 d9
e1 e2 e3 e4 e5 e6 e7 e8 e9

Figure 1 Robot navigation field.

Depth First Search

Fringe (Last → Next):	**Node visited/expanded**
d2	d2
e2, d1	d1
e2, e1, c1	c1
e2, e1, b1	b1
e2, e1, a1	a1
e2, e1, a2	a2
e2, e1, a3	a3
e2, e1, a4	a4
e2, e1, b4, a5	a5
e2, e1, b4, a6	a6
e2, e1, b4, a7	a7
e2, e1, b4, **b7**, a8	**b7 FOUND**

Order of node expansion: d2, d1, c1, b1, a1, a2, a3, a4, a5, a6, a7
Final path: d2, d1, c1, b1, a1, a2, a3, a4, a5, a6, a7, b7

Breadth First Search

Fringe (Last → Next):	**Node visited/expanded**
d2	d2
e2, d1	d1
e1, c1, e2	e2
e3, e1$_2$, e1, c1	c1
b1, e3, e1$_2$, e1	e1
b1, e3, ~~e1$_2$~~ (e1$_2$ removed due to e1 already being explored)	e3
e4, b1	b1
a1, e4	e4
e5, a1	a1
a2, e5	e5
e6, a2	a2
a3, e6	e6
e7, a3	a3
a4, e7	e7
e8, a4	a4
b4, a5, e8	e8
e9, b4, a5	a5
a6, e9, b4	b4
c4, a6, e9	e9
d9, c4, a6	a6

a7, d9, c4	c4
c5, a7, d9	d9
c9, c5, a7	a7
b7, a8, c9, c5	**b7 FOUND**

Order of node expansion: d2, d1, e2, c1, e1, e3, b1, e4, a1, e5, a2, e6, a3, e7, a4, e8, a5, b4, e9, a6, c4, d9, a7

Final path: d2, d1, c1, b1, a1, a2, a3, a4, a5, a6, a7, b7

2. **[1.5 pts]** Figure 1 shows a robot navigation field, where the red square (d2) is the robot, and green square (b7) is the goal. The shade squares (such as b2, c2, etc.) are obstacles. The robot is not allowed to move in diagonal line. Please use best first search to find the path from d2 to b7 (Using Manhattan distance as the heuristic function)
    - Report nodes in the fringe (including their f(N) values), in the orders they are included in the fringe (Using table format showing below. Add rows if needed) [0.5]
    - Report the order of the nodes being expanded. [0.5]
    - Report the final path from d2 to b7. [0.5]

Best First Search

Fringe (Last → Next):	**Node visited/expanded**
d2 (7)	d2
e2 (8), d1 (8)	d1
e1 (9), e2 (8), c1 (7)	c1
e1 (9), e2 (8), b1 (6)	b1
e1 (9), e2 (8), a1 (7)	a1
e1 (9), e2 (8), a2 (6)	a2
e1 (9), e2 (8), a3 (5)	a3
e1 (9), e2 (8), a4 (4)	a4
e1 (9), e2 (8), b4 (3), a5 (3)	a5
e1 (9), e2 (8), b4 (3), a6 (2)	a6
e1 (9), e2 (8), b4 (3), a7 (1)	a7
e1 (9), e2 (8), b4 (3), a8 (2), **b7 (0)**	**b7 EXPANDED/FOUND**

Order of node expansion: d2, d1, c1, b1, a1, a2, a3, a4, a5, a6, a7, b7
Final Path: d2, d1, c1, b1, a1, a2, a3, a4, a5, a6, a7, b7

3. **[1.5 pts]** Figure 1 shows a robot navigation field, where the red square (d2) is the robot, and green square (b7) is the goal. The shade squares (such as b2, c2, etc.) are obstacles. The robot is not allowed to move in diagonal line. Please use A* search to find the path from d2 to b7 (Using Manhattan distance as the heuristic function)
   - Report nodes in the fringe (including their f(N) values), in the orders they are included in the fringe. (Using table format showing below. Add rows if needed) [0.5]
   - Report the order of the nodes being expanded. [0.5]
   - Report the final path from d2 to b7. [0.5]

A* Search

Fringe (Last → Next):	**Node visited/expanded**
d2 (7)	d2
e2 (9), d1 (9)	d1
e1 (11), e2 (9), c1 (9)	c1
e1 (11), e2 (9), b1 (9)	b1
e1 (11), a1 (11), e2 (9)	e2
$e1_2$ (11), e1 (11), a1 (11), e3 (9)	e3
$e1_2$ (11), e1 (11), a1 (11), e4 (9)	e4
$e1_2$ (11), e1 (11), a1 (11), e5 (9)	e5
$e1_2$ (11), e1 (11), a1 (11), e6 (9)	e6
$e1_2$ (11), e1 (11), a1 (11), e7 (9)	e7
e8 (11), $e1_2$ (11), e1 (11), a1 (11)	a1
e8 (11), $e1_2$ (11), e1 (11), a2 (11)	a2
e8 (11), $e1_2$ (11), e1 (11), a3 (11)	a3
e8 (11), $e1_2$ (11), e1 (11), a4 (11)	a4
e8 (11), $e1_2$ (11), e1 (11), b4 (11), a5 (11)	a5
e8 (11), $e1_2$ (11), e1 (11), b4 (11), a6 (11)	a6
e8 (11), $e1_2$ (11), e1 (11), b4 (11), a7 (11)	a7
a8(13), e8 (11), $e1_2$ (11), e1 (11), **b7 (11)**, b4 (11)	b4
c4 (13), a8(13), e8 (11), $e1_2$ (11), e1 (11), **b7 (11)**	**b7 EXPANDED/FOUND**

Order of node expansion: d2, d1, c1, b1, e2, e3, e4, e5, e6, e7, a1, a2, a3, a4, a5, a6, a7, b4, b7
Final Path: d2, d1, c1, b1, a1, a2, a3, a4, a5, a6, a7, b7

4. **[2.5 pts]** Figure 2 shows a search tree where A denotes the node corresponding to the initial state, and E is the goal node. In the figure, h=x denotes the heuristic function value and c=x denotes the actual cost between nodes (i.e., arch cost).
   a. Find all heuristic(s) in Figure 2 which are inconsistent [0.5 pt]
   b. Create a search tree using the heuristic values given in Figure 2 to carry out A* search from A to find goal node E (Node expanded/visited does not need to be revisited). The tree must show the complete search process with each node's f(N) values. Report the order of the nodes being expanded, and the final discovered path from A to E [0.5 pt]
   c. Is the path discovered from the above process optimal or not? Explain why [0.5 pt]
   d. Correct heuristic values in Figure 2, such that the A* using the corrected heuristics can find optimal path from A to E. Solutions must show corrected heuristic values for all nodes [0.5]
   e. Create a search tree using corrected heuristic values to carry out A* search from A to find goal node E (Node expanded/visited does not need to be revisited). The tree must show the complete search process with each node's f(N) values. Report the order of the nodes being expanded, and the final discovered path from A to E [0.5 pt]

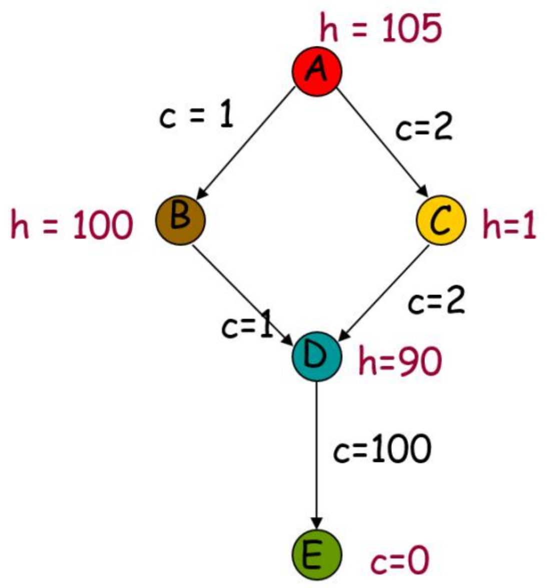
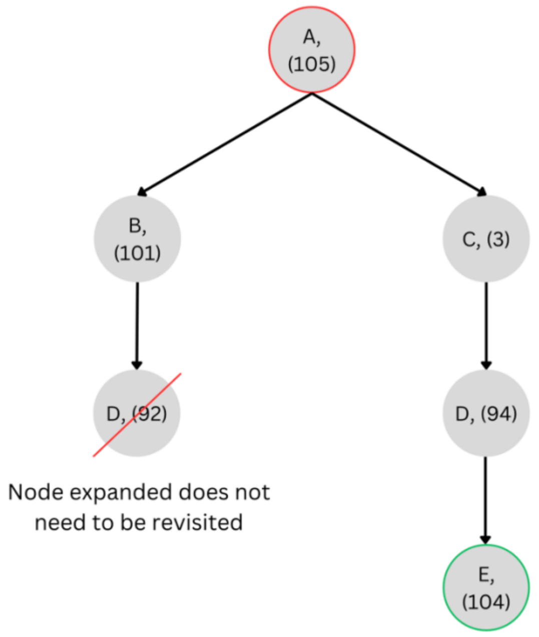


Figure 2: A search tree and husirtic function values

Heuristics Consistency

h(N) <= c(N,N') + h(N')?		Consistent?
For A → B, is 105 <= 101?	No	No
For A → C, is 105 <= 3?	No	No
For B → D, is 100 <= 91?	No	No
For C → D, is 1 <= 92?	Yes	Yes
for D → E, is 90 <= 100?	Yes	Yes

∴ Heuristics A→B, A→C, and B→D are inconsistent.



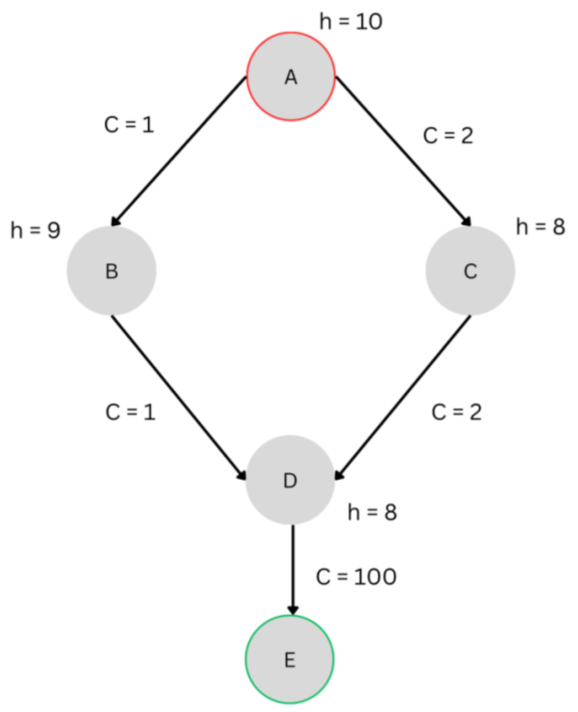
<u>A* Search Tree</u>

A* Search

Fringe (Last → Next):	**Node visited/expanded**
A (105)	A
B (101), C(3)	C
B (101), D(94)	D
E(104), B(101)	B
~~D(92),~~ **E(104)**	**E, GOAL FOUND**

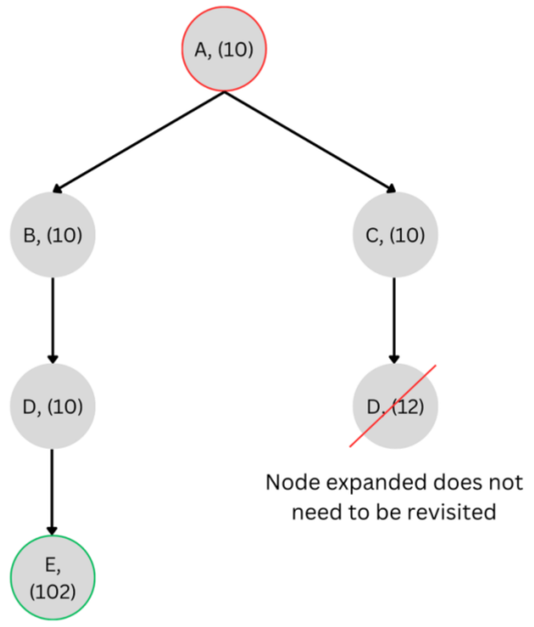
Order of node expansion: A, C, D, B, E
Final path: A, C, D, E

No, the path discovered is not optimal. The optimal path minimizes the cost from the start node to the goal node. The path A, C, D, E, has a total cost of 104, while the path A, B, D, E, has a total cost of 102. The A* search found the path with a higher cost, meaning the path discovered was not optimal due to the existence of another lower cost path. A* initially chose the path of higher cost due to the inconsistent heuristic, leading it to expand D from the higher cost path first. Due to the fact that revisitation is not allowed, when the A* search explored B, it discarded the lower cost path to D. This caused A* to find the higher cost path to E through node C. This lack of optimality is due to the inconsistent heuristic. If the heuristic is consistent, then A* preforms optimally and it returns the optimal path from the start node to the goal node. A* is optimal regardless of revisitation if the heuristic used is consistent. On the other hand, A* is not optimal if revisiting is not allowed and the heuristic is inconsistent. This occurred in the search tree, causing A* to find a sub-optimal path.



<u>**Figure 2 with corrected heuristic values**</u>

The corrected heuristic values are all consistent, leading A* search to find an optimal path.



A* Search tree for corrected heuristic values

A* Search for corrected heuristic values

Fringe (Last → Next):	**Node visited/expanded**
A (10)	A
C (10), B (10)	B
D (10), C (10)	C
D (12), D (10)	D (10)
E (102), ~~D (12)~~,	**E, GOAL FOUND**

Order of node expansion: A, B, C, D, E
Final path: A, B, D, E

5. [**2 pts**] Figure 3 shows a roadmap with four cities. The value next to each edge denotes the actual distance between two nodes. The heuristic value of each city is given in Table 1.

   f. Use greedy Best-First Search to create a search tree to find path from source node S to the goal node G. The three must show f(N) value of each node, and report final path discovered by the Best-First Search (either highlight the path on the tree, or report the path separately) [1. pt]

   g. Use A* Search to create a search tree to find path from source node S to the goal node G. The three must show f(N) value of each node, and report final path discovered by the Best-First Search (either highlight the path on the tree, or report the path separately) [1. pt]

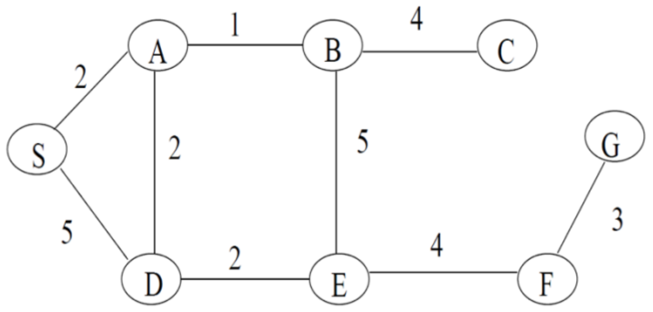
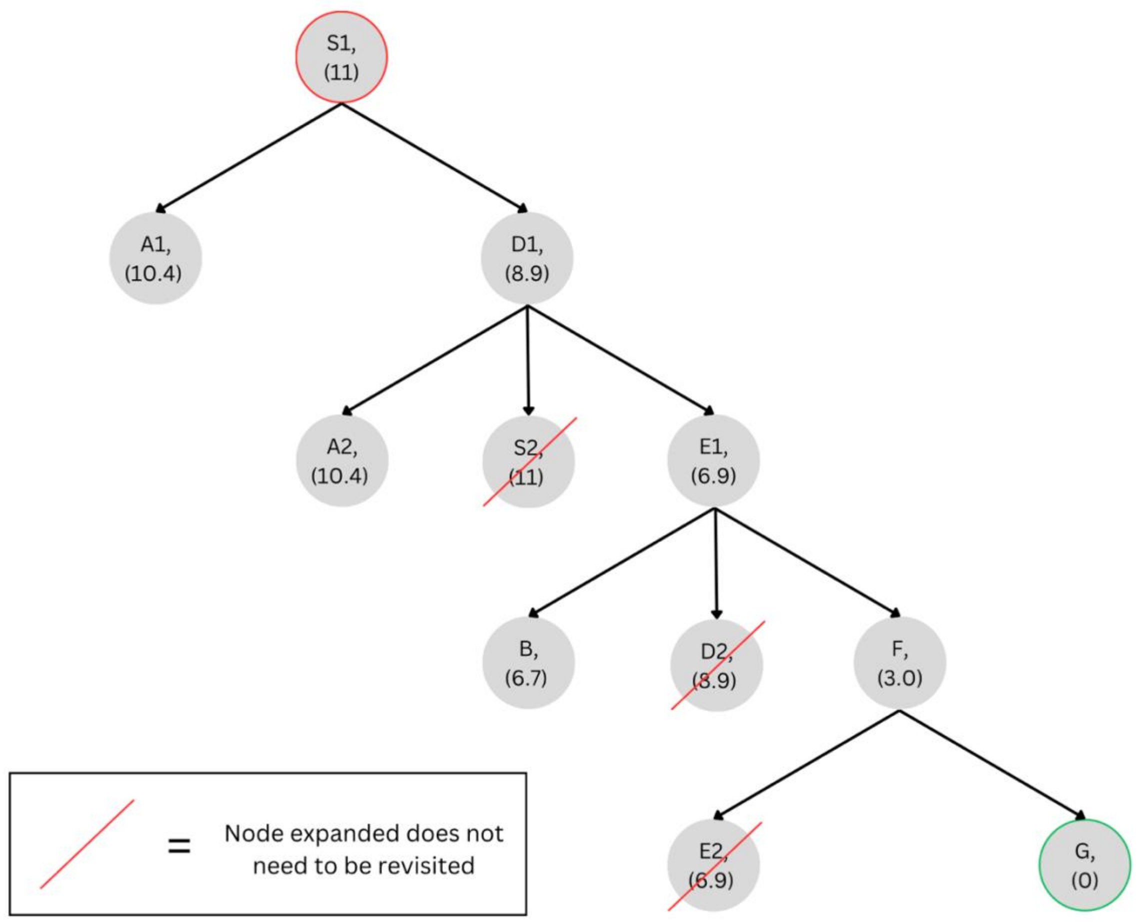


Figure 3: A field map with eight nodes. Value next to each edge denotes actual cost between two nodes. Heuristic value of each city is given in Table 1.

Table 1: Heuristic values h(N) of each node to the goal (G) node.

A	10.4
B	6.7
C	4.0
D	8.9
E	6.9
F	3.0
G	0
S	11.0

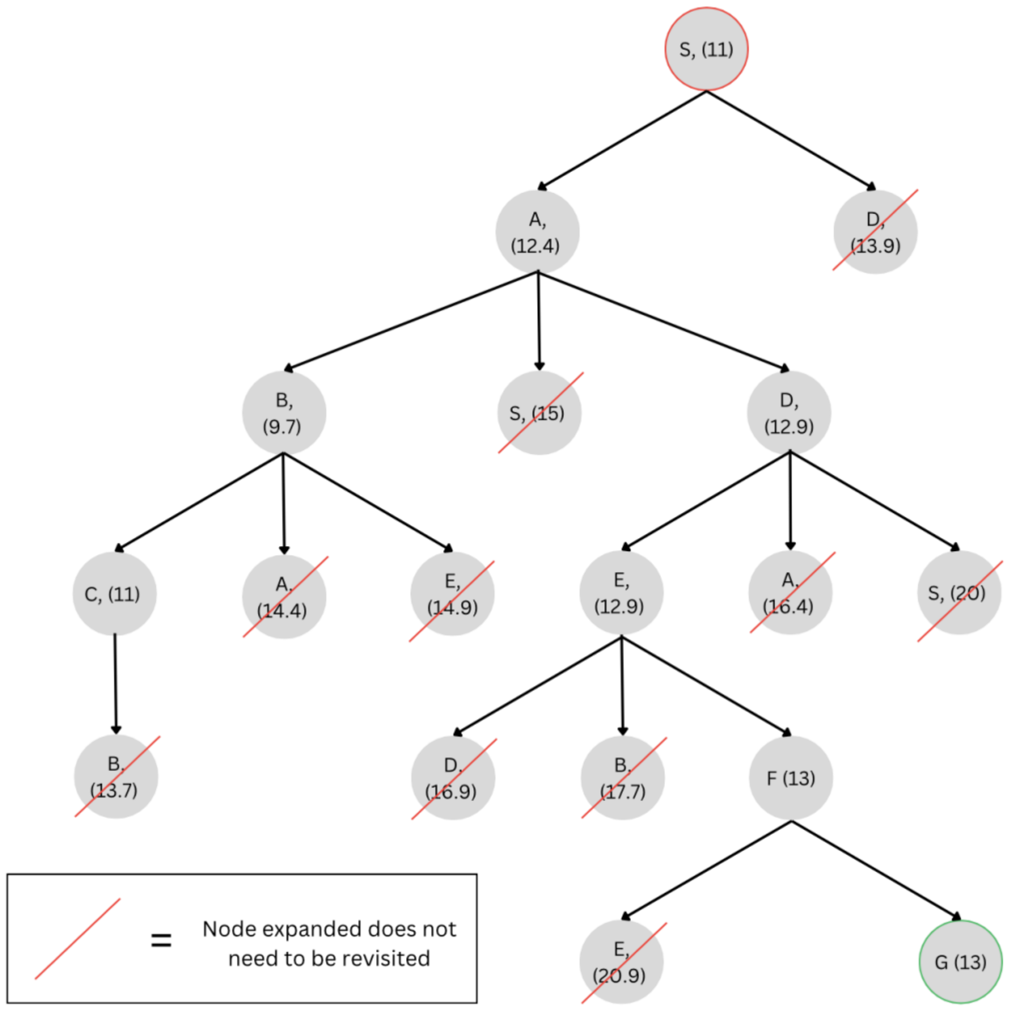


<u>**Greedy Best-First Search Tree**</u>

Greedy Best-First Search

Fringe (Last → Next):	**Node visited/expanded**
S1 (11)	S1 (11)
A1 (10.4), D1 (8.9)	D1 (8.9)
~~S2 (11)~~, A2 (10.4), A1 (10.4), E1 (6.9)	E1 (6.9)
A2 (10.4), A1 (10.4), ~~D2 (8.9)~~, B (6.7), F (3.0)	F (3.0)
A2 (10.4), A1 (10.4), ~~E2 (6.9)~~, B (6.7), **G (0)**	**G (0), GOAL FOUND**

Order of node expansion: S, D, E, F, G
Final path: S, D, E, F, G



<u>A* Search Tree</u>

A* Search

Fringe (Last → Next):	**Node visited/expanded**
S (11)	S (11)
D (13.9), A (12.4)	A (12.4)
~~S (15),~~ D (13.9), D (12.9), B (9.7)	B (9.7)
E (14.9), ~~A (14.4),~~ D (13.9), D (12.9), C (11)	C (11)
E (14.9), ~~B(13.7),~~ D (13.9), D (12.9),	D (12.9)
~~S (20), A (16.4), E (14.9), D (13.9),~~ E (12.9)	E (12.9)
~~B (17.7), D (16.9),~~ F (13)	F (13)
~~E (20.9),~~ **G (13)**	**G (13), GOAL FOUND**

Order of node expansion: S, A, B, C, D, E, F, G
Final path: S, A, D, E, F, G

6. **[1.5 pts]** Figure 4 shows an 8-Queen board layout and its genetic string code. Using same encoding format, to complete the following tasks (8-Queen's problem: place 8 queens on an 8x8 chessboard such that no two queens can attack each other (i.e. share the same row, column, or diagonal).
   h. For the following three individuals, please calculate fitness score for each of them (the fitness score is defined as number of queen pairs not attacking each other. The order of the queen does not matter. For example, queen 3 attacking queen 5 is equivalent to queen 5 attacking queen 3). [0.5 pt]

      X1=36482571

      X2=63188183

      X3=56788372
   i. Select the two individuals with the highest fitness scores to generate two children using cross-over to mix parents' genetic code. The site of the cross-over is selected as 3 (i.e., after the third queen's position). Report fitness scores for both children. [0.5 pt]
   j. Explain the strength and weakness of genetic algorithms in solving search tasks. [0.5]

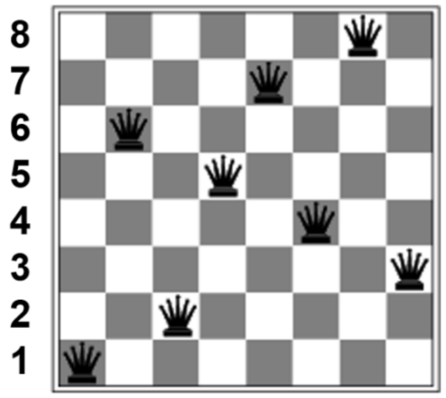
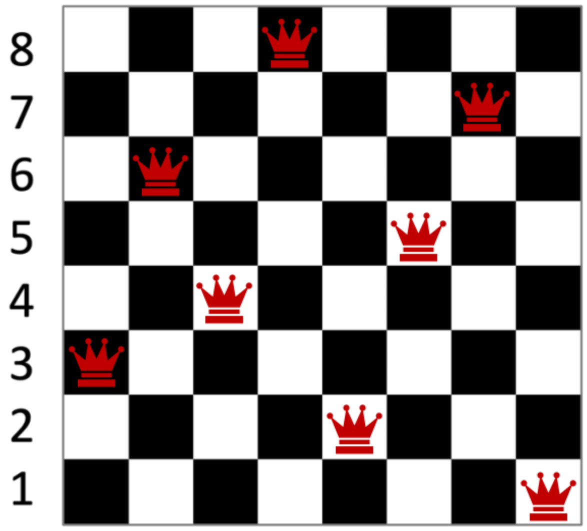


Figure 4 8-Queen problem, and the genetic string encoding: 16257483 (each digit, left to right, encodes the row number of the ith queen. For example, the first digit "1" denotes that the queen is on the 1st row).

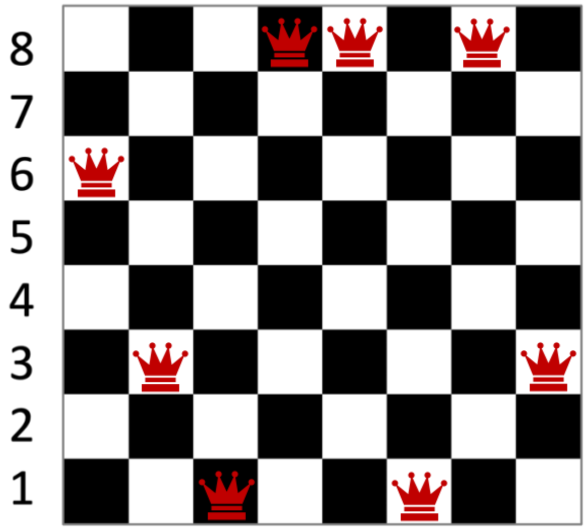
Total number of possible queen pairs = (8 * 7) / 2 = 28

- To calculate fitness score: 28 – (number of attacking pairs)



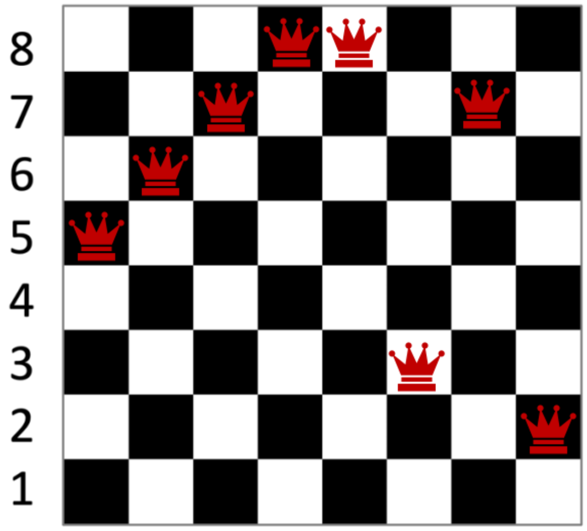
X1 = 36482571

- 2 pairs of queens attacking each other
- 28 – 2 = 26
- 26 queen pairs not attacking each other
- **Fitness Function = 26**



X2 = 63188183

- 8 pairs of queens attacking each other
- 28 – 8 = 20
- 20 queen pairs not attacking each other
- **Fitness Function = 20**

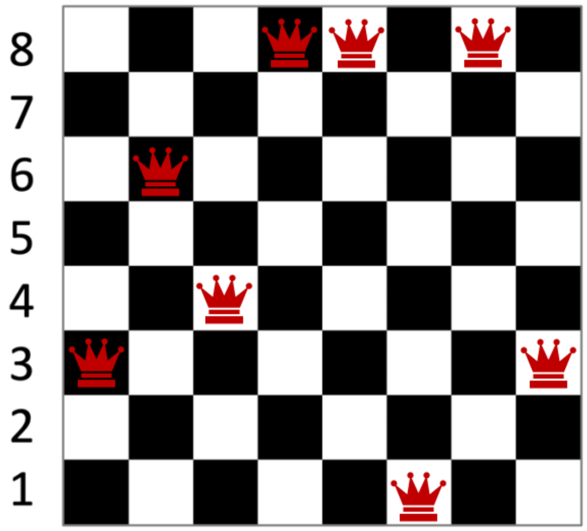


X3 = 56788372

- 9 pairs of queens attacking each other
- 28-9 = 19
- 19 queen pairs not attacking each other
- **Fitness Function = 19**

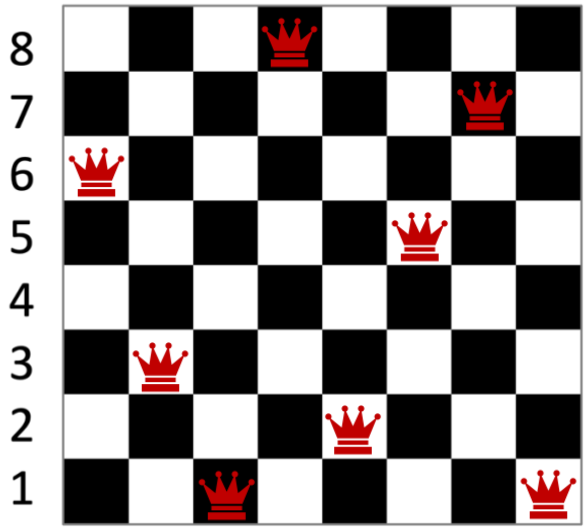
Highest Fitness Scores:
X1 = 36482571
X2 = 63188183

Cross Over:
X1 = 364 | 82571 → X1’ = 364 | 88183 = 36488183
↙ ↑
X2 = 631 | 88183 → X2’ = 631 | 82571 = 63182571



X1’ = 36488183

- 8 pairs of queens attacking each other
- 28 – 8 = 20
- 20 queen pairs not attacking each other
- **Fitness Function = 20**



X2’ = 63182571

- 2 pairs of queens attacking each other
- 28 – 2 = 26
- 26 queen pairs not attacking each other
- **Fitness Function = 26**

Genetic algorithms have several strengths and weaknesses. One of their primary strengths is their ability to avoid getting stuck in local optima. The ability to create offspring of the best solutions with cross over as well as random mutations allows diversity to be introduced into the solutions, preventing them from getting stuck in one region of the solution space. Additionally, genetic algorithms have a strong connection to human evolution, which is appealing in the context of biologically inspired artificial intelligence algorithms. However, genetic algorithms have a large number of tunable parameters which may make finding the optimal set of parameters difficult. Furthermore, this large set of tunable parameters along with the stochastic nature of genetic algorithms makes replicating results difficult. Genetic algorithms have also not been shown to be better than other non-local solutions such as hill climbing algorithms with random restarts. Finally, genetic algorithms are typically much slower than other solutions, such as hill climbing with random restarts, for general optimization problems.

7. **[2 pts]** Figure 5 shows the current layout of a tic-tac-toe game board, and its' Max ("X") turn to make the next move.
   - Please draw the remaining states of the game tree [1 pt],
   - Determine best move for Max and explain the reason [1 pt].

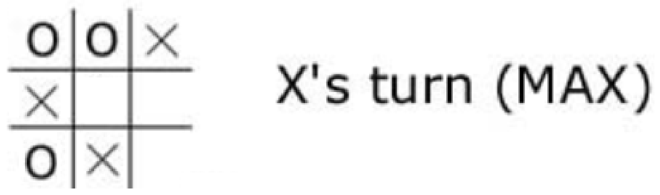
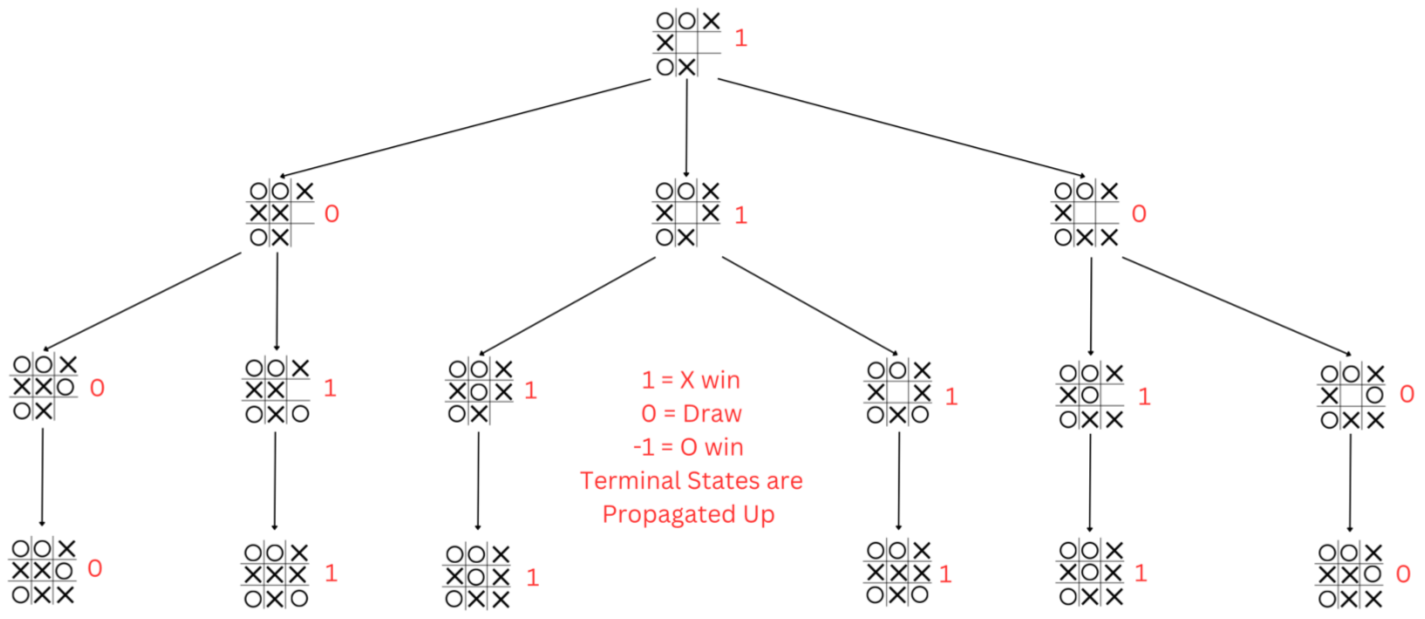


Figure 5



Remaining state of the game tree

The best move for max (x), would be to place an x on the middle row of the right column. This is because this move would result in a win for X no matter where the subsequent Os or Xs are placed. Through finding the terminal states and propagating them up the state tree according to the minimax algorithm, you can see that X will win against an optimal opponent if X plays the move previously described. Despite Os best attempt to win, any move will lead to a win for X after the initial placement of an X in the middle row of the right column. If X plays any other move, O will be able to draw the game.

8. **[2.5 pts]**. Given a two-player game tree in Figure 6,
    - Use min-max algorithm to mark Min and Max value for each Min Max node respectively [0.5]
    - Determine best path for the root node (i.e., the Min node) [0.5]
    - Use Alpha-beta pruning algorithm to find correct α value for each Max node, and correct β value for each Min node (no need to report α, β values for nodes pruned by α-β pruning). [1 pt]
    - Mark all nodes which are pruned by α-β pruning [0.5 pt]

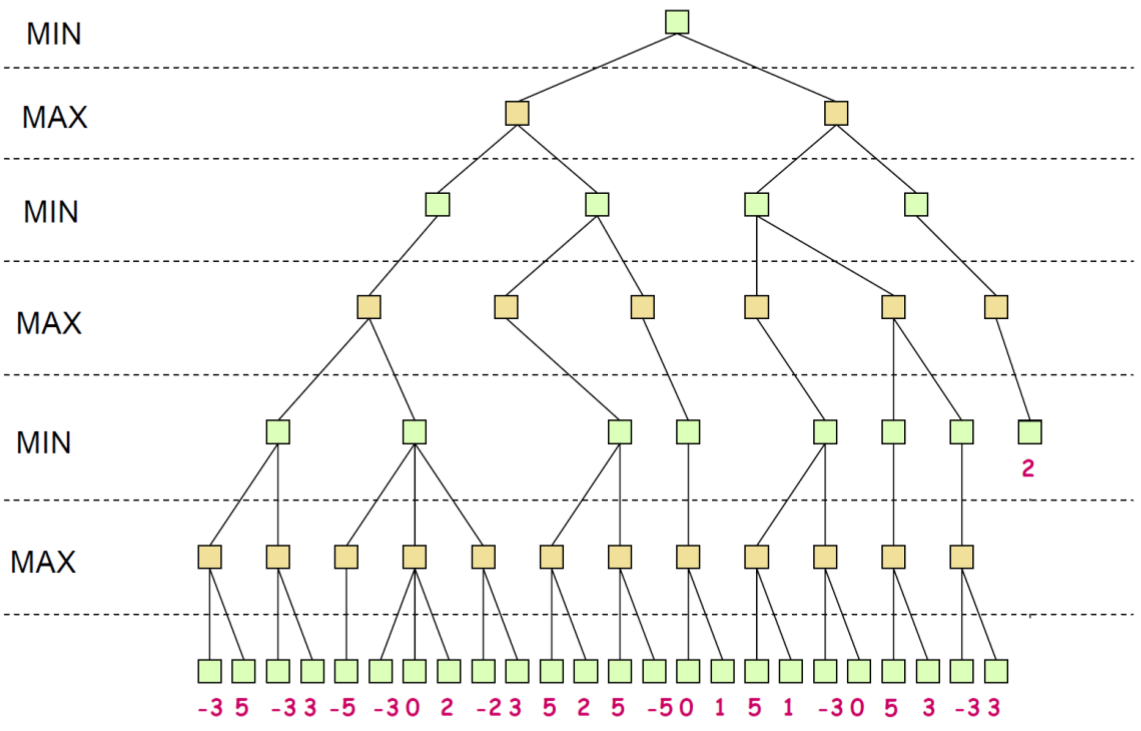


Figure 6

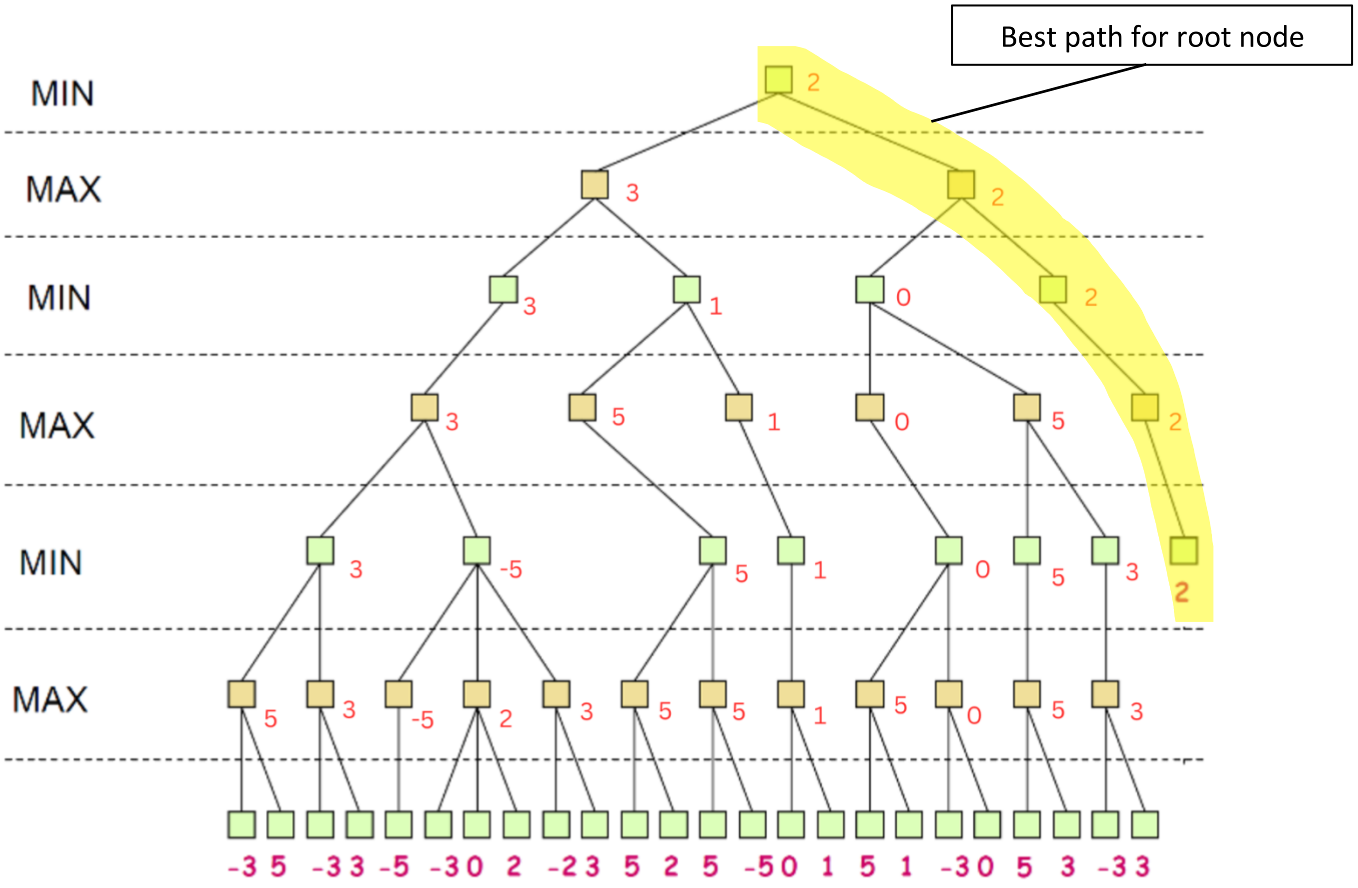
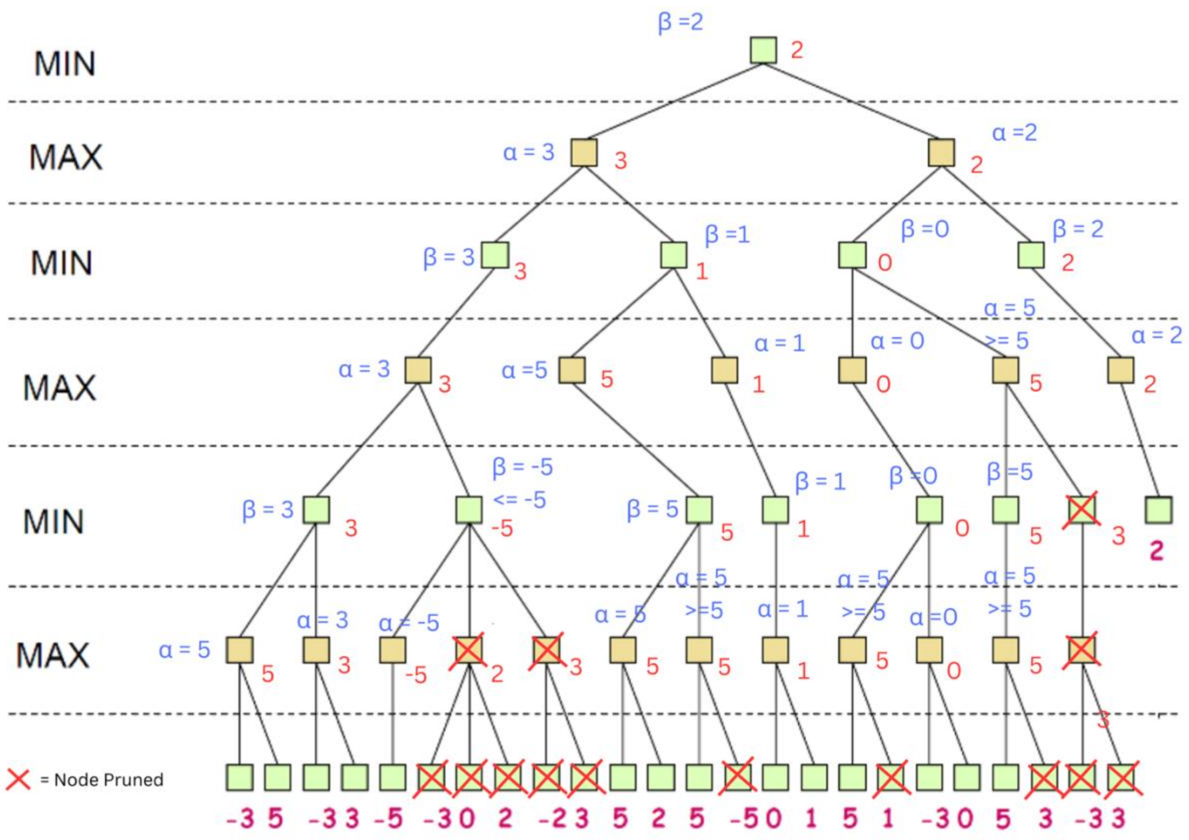


Figure 6 with marked min and max values

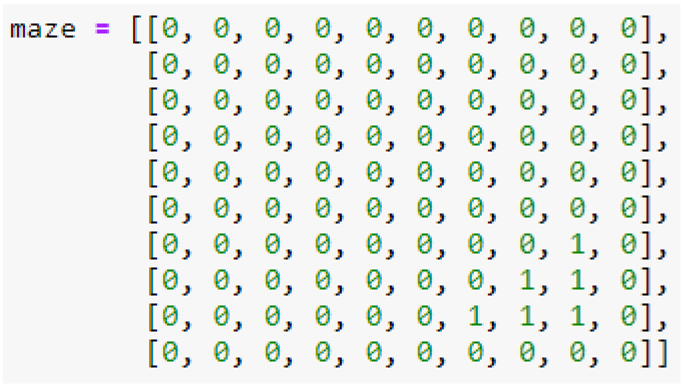
Assuming optimal play from both the min and max players, the best path for the root node is the highlighted path **(right, right, right, right)**. Any other move may allow the max player to score higher than 2, which is the best (lowest) score that min can achieve.



Alpha-beta pruning

The tree above is marked with alpha and beta values for max and min nodes respectively. The nodes marked with Xs are pruned due to alpha-beta pruning.

9. **[1.5 pts + 1 Extra credit]** The "**Informed Search to Play Maze [Notebook, html]**" posted on Canvas shows a Maze game using A-Star Search (AStar), Greedy Best First (GBF) and Uniform Cost Search (UCS) (using "AStar", "GBF", or "UCS" as parameters). Use Notebook as the skeleton code, validate and compare following settings and results.
   k. Using Figure 7 as the game field, and set initial state as [0, 0] and goal state as [9, 9]. Use A-Star Search (AStar), Greedy Best First (GBF) and Uniform Cost Search (UCS) to find path from initial state to goal state, respectively. Report path of each method [0.25 pt], and explain which method is optimal/not optimal, why? [0.25 pt].
   l. What is advantage of AStar search, comparing to greedy best first search (GBF) and Uniform Cost Search [0.5 pt]
   m. What is advantage of Greedy Best First search, comparing to A-Star search (AStar) and Uniform Cost Search [0.5 pt]



```
maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Figure 7

A*
search path length: 14.485281
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (5, 6), (5, 7), (5, 8), (6, 9), (7, 9), (8, 9), (9, 9)]

GBF
search path length: 15.313708
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (6, 7), (5, 8), (6, 9), (7, 9), (8, 9), (9, 9)]

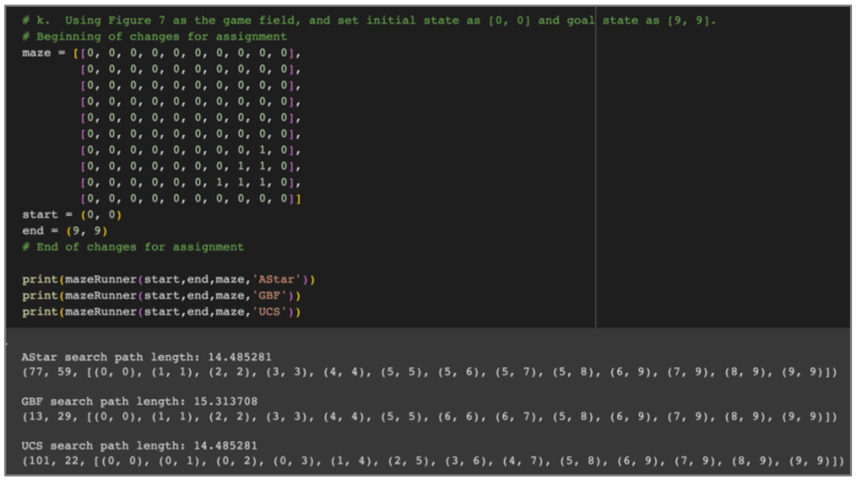
UCS
search path length: 14.485281
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 4), (2, 5), (3, 6), (4, 7), (5, 8), (6, 9), (7, 9), (8, 9), (9, 9)]

Optimality of search methods
The "informed search to play maze" code utilizes an agent that can move in the x, y, or diagonal direction and has an arch cost of the Euclidean distance between the current node and the child node for each step. Additionally, the heuristic function is defined as the Euclidean distance from the current node to the goal node. Based on this information, we can derive the fact that

the heuristic is consistent because the Euclidean distance is a consistent heuristic with diagonal steps and a Euclidean distance step cost. Based on this, A* search and UCS are both optimal while GBF is not optimal. This is because A* search is an optimal search strategy as long as the heuristic is consistent. UCS is always an optimal search strategy because it expands the lowest cost path first, thus it will find the best overall solution first. On the other hand, GBF is not optimal because it expands nodes based on the heuristic alone, which is only an optimistic estimate of the lowest cost path from the current node to the goal node. Since the heuristic is the Euclidean distance from the node to the goal node, it will keep moving to the next node with a better heuristic. In this case, the goal node is blocked by some obstacles, which make the GBF search move all the way to the obstacles, before then going around. GBF does not consider an alternative route that does not waste time moving to the obstacles because it does not consider the step cost. Thus, GBF is not optimal and does not find the optimal route.

Advantages of A* search
The main advantage of A* search is that it is faster than UCS while still being optimal, assuming the heuristic is consistent. UCS always finds the best solution, however, it creates an expansive search tree in the process. This means that it has a longer run time because it explores more options than are necessary. A* is also optimal but is much faster than UCS because it takes into account the heuristic to help it expand promising nodes. This means that it does not waste time pursuing a path that moves farther from the solution, despite a low cost. Essentially, A* expands far fewer nodes than UCS, with the same optimal solution as a result. GBF is faster than A*, however, it is not optimal. The advantage of A* over GBF is that it produces an optimal result, albeit, at the cost of more nodes expanded and a longer runtime.

Advantages of GBF
The main advantage of greedy best first search is that it is much faster than either uniform cost search or A* search. It expands far fewer nodes and only pursues the most promising nodes first, or the nodes that bring the agent closest to the solution. GBF search typically produces an acceptable result, however, GBF is not optimal, unlike both A* and UCS. In a large solution space where the time complexity may make it infeasible to find the optima, GBF may be able to find a good-enough solution in an acceptable runtime where A* and UCS may not.



```
# k.  Using Figure 7 as the game field, and set initial state as [0, 0] and goal state as [9, 9].
# Beginning of changes for assignment
maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
start = (0, 0)
end = (9, 9)
# End of changes for assignment

print(mazeRunner(start,end,maze,'AStar'))
print(mazeRunner(start,end,maze,'GBF'))
print(mazeRunner(start,end,maze,'UCS'))
```

```
AStar search path length: 14.485281
(77, 59, [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (5, 6), (5, 7), (5, 8), (6, 9), (7, 9), (8, 9), (9, 9)])

GBF search path length: 15.313708
(13, 29, [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (6, 7), (5, 8), (6, 9), (7, 9), (8, 9), (9, 9)])

UCS search path length: 14.485281
(101, 22, [(0, 0), (0, 1), (0, 2), (0, 3), (1, 4), (2, 5), (3, 6), (4, 7), (5, 8), (6, 9), (7, 9), (8, 9), (9, 9)])
```

10. [**1 pt**] The "**Local Search N Queens [Notebook, html]**" posted on Canvas shows examples of using local search, including hill-climbing Search (HC) and Simulated Annealing (SA) to solve N-Queens problem. Use Notebook as the skeleton code, validate and compare following settings and results.
    n. Set number of questions (N) as 8, 16, 32, 64, respectively. Repeat hill-climbing search (HC) and simulated annealing (SA) 10 times for each N values, calculate average runtime and success rates (1 divided by number of repetitions before success). Explain how HC and SA behave in terms of runtime and success rates. [0.25 pt]
    o. Compare costs plots (which show the costs, number of attacked pairs, with respect to the board updating) between HC and SA, explain why plot from HC is monotonically decreasing, whereas the plot from SA is going up and down [0.25]
    p. For simulated annealing method (SA), fix the number of Queen to 64, and vary the initial temperature from T=4000 to T=[400, 40, 4, 0.4], respectively. [0.25]
    q. Explain how SA behaves, and explain why SA is a better local search method, in general, than HC [0.25]

Average Success Rate

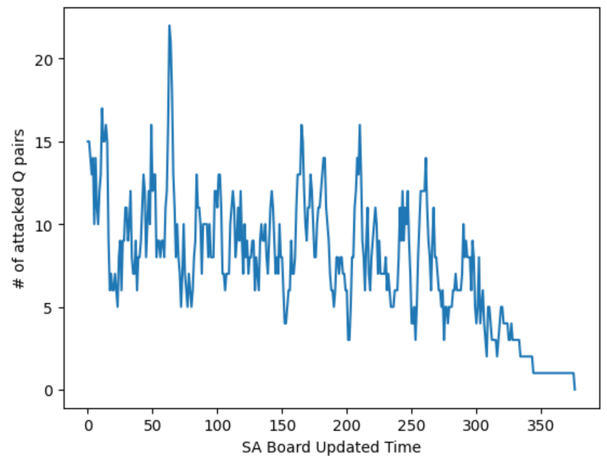
	N=8	**N=16**	**N=32**	**N=64**
Hill Climbing	0.43928571428571433	0.5693589743589744	0.24111111111111114	0.0752308693485164
Simulated Annealing	0.9	0.95	0.7	0.27675384112455054

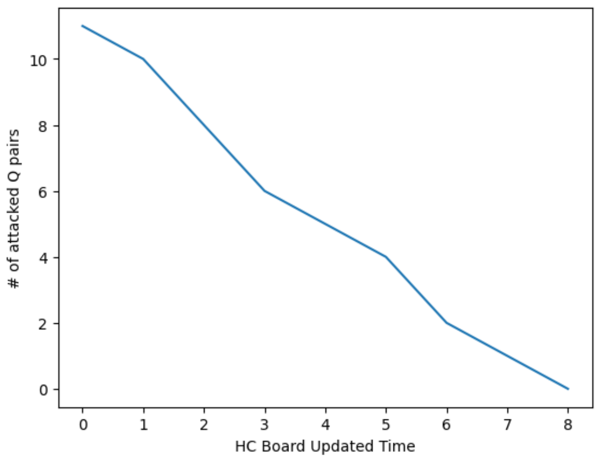
Average Runtime

	N=8	**N=16**	**N=32**	**N=64**
Hill Climbing	0.1541755437850952	0.42377538681030275	0.798902940750122	4.417231869697571
Simulated Annealing	0.04562280178070068	0.11488432884216308	0.22723042964935303	2.2052347898483275

HC and SA runtime and success rate

The simulated annealing search algorithm outperforms the hill climbing algorithm in terms of both success rate and run time. Simulated annealing is faster than hill climbing, and it has a higher success rate. Additionally, both hill climbing and simulated annealing have a downward trending success rate as the number of queens increases and an upwards trending runtime as the number of queens increases. This is consistent with general intuition, because as the number of queens increases, the solution space gets much larger, which increase the time it takes to find a valid configuration. Furthermore, this means that there will be more failed attempts as the solution space increases.





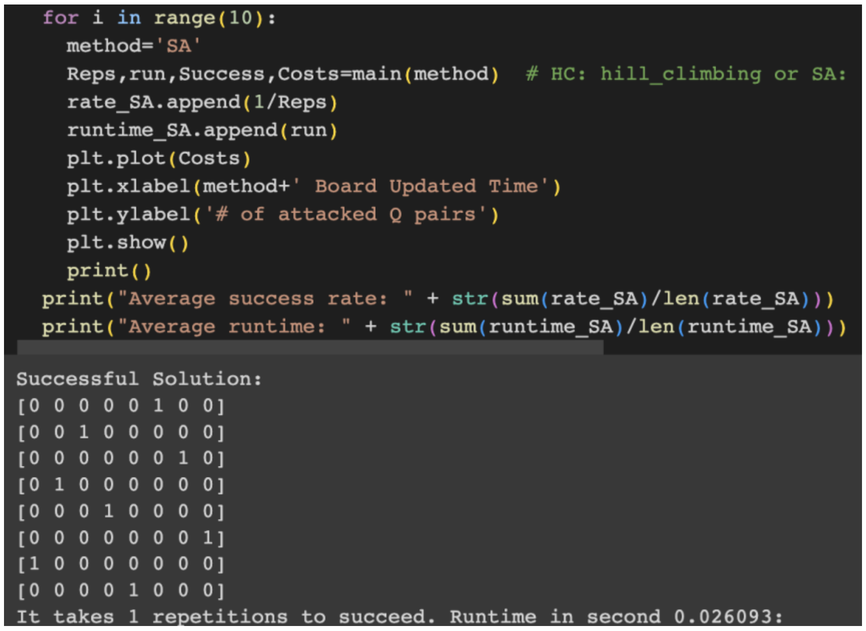
Cost plot comparisons

The cost plots demonstrate the nature of both the simulated annealing and hill climbing algorithms. The hill climbing algorithm's cost plot is a monotonically decreasing curve while the simulated annealing algorithm's cost plot contains ups and downs with a generally decreasing trend. These ups and downs decrease in frequency until the curve becomes mostly monotonically decreasing. The reason that the hill climbing cost plot is monotonically decreasing is because the hill climbing algorithm moves towards a better solution overtime through a greedy approach. Therefore, for each time unit, the solution gets better, until a solution is found. If a solution is not found and the algorithm gets stuck in a local optimum, the algorithm restarts with a random location and tries again. On the other hand, simulated annealing starts at a random point in the solution space and evolves the solution according to the heuristic, but also allows worse solutions to be considered based on a temperature probability value. Overtime, the temperature decreases and the probability that worse solutions are considered decreases, allowing the algorithm to converge on an optimum value. The up and down nature of the SA's graph shows the algorithm considering worse solutions. This prevents the algorithm from getting stuck in a local optimum. Overtime, the frequency of these ups and downs decreases due to the decrease in temperature, and as a result, a decrease in worse solutions chosen. This creates the almost-monotonically decreasing curve at the right most part of the curve. The overall curve trends downwards as it converges to a solution. It can also be seen that HC takes less time than SA, however, HC takes more attempts (restarts), making the overall runtime of HC greater than SA in general.

SA Temperature

	T=4000	**T=400**	**T=40**	**T=4**	**T=0.4**
Success Rate	0.04545	0.1	0.1429	0.0625	0.0588
Runtime	6.642561	2.879958	2.597933	2.961761	3.276815

SA behavior and performance
SA works by evolving a randomly generated solution overtime according to a heuristic. However, based on a configurable temperature value, SA also accepts worse solutions to a probability proportional to the temperature value. This temperature value decreases overtime, making the algorithm accept worse solutions to lesser degrees. This behavior allows the algorithm to better avoid getting stuck in local optima by accepting worse solutions that may bring it close to a global optimum in the overall solutions space. As the temperature value decreases overtime, the solution begins to converge on an optimum by selecting the best solution in an increasing probability. Eventually, the algorithm converges on a solution. SA is better than HC in general because it has a better chance to find the global optimum than HC. HC only ascend the general terrain of the solution space local to the randomly initiated solution that it begins with. This means that HC has a higher chance of only finding a local optimum rather than a better global optimum. SA allows for the solution to have a better chance of finding the "hill" in the solution space that would lead to the global optimum by accepting worse solutions. Eventually, worse solutions are accepted less to allow the algorithm to ascend the solution space and find an optimum. This optimum is often a better optimum than the one found by HC, and it may even find the global optima in comparatively fewer attempts than HC with random restarts. Finally, if the temperature is lowered slowly enough, SA is both complete and admissible.



```
  for i in range(10):
    method='SA'
    Reps,run,Success,Costs=main(method)  # HC: hill_climbing or SA:
    rate_SA.append(1/Reps)
    runtime_SA.append(run)
    plt.plot(Costs)
    plt.xlabel(method+' Board Updated Time')
    plt.ylabel('# of attacked Q pairs')
    plt.show()
    print()
  print("Average success rate: " + str(sum(rate_SA)/len(rate_SA)))
  print("Average runtime: " + str(sum(runtime_SA)/len(runtime_SA)))
```

```
Successful Solution:
[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 1 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
It takes 1 repetitions to succeed. Runtime in second 0.026093:
```

11. [**1 pt**] The Genetic Algorithm N Queens [Notebook, html] posted on Canvas shows example of using genetic algorithm to solve N-Queens problem. Use Notebook as the selection code, answer following questions, and implement respective tasks.
    r. For the same number of Queens (E.g., N=16), comparing speed between hill-climbing vs. genetic algorithm, explain why genetic algorithm is much slower than hill-climbing algorithm [0.25 pt]
    s. One problem of genetic algorithm is that the cross-over and mutation process may generate off-spring whose genetic code is not a permutation (i.e., a clear attack case to the N-Queens) problem. For example, genetic code "23813754" is not a permutation, because there are two queens in a row (or column) [there are two "3"], and it also misses 8. Propose a solution to generate off-spring whose genetic code is a permutation [0.25]
    t. Implement your algorithm and compare its performance vs. the original genetic algorithm to solve N=16 queens problem (repeat 5 times and report average runtime). [0.5 pt]

N = 16 is too large of a solution space to run on Google Colab or my local hardware in a reasonable amount of time. Therefore, I ran the hill climbing (average over 10 iterations), original genetic (average over 5 iterations), and permutation genetic algorithms (average over 5 iterations) using N = 8 and included the results in the table below.

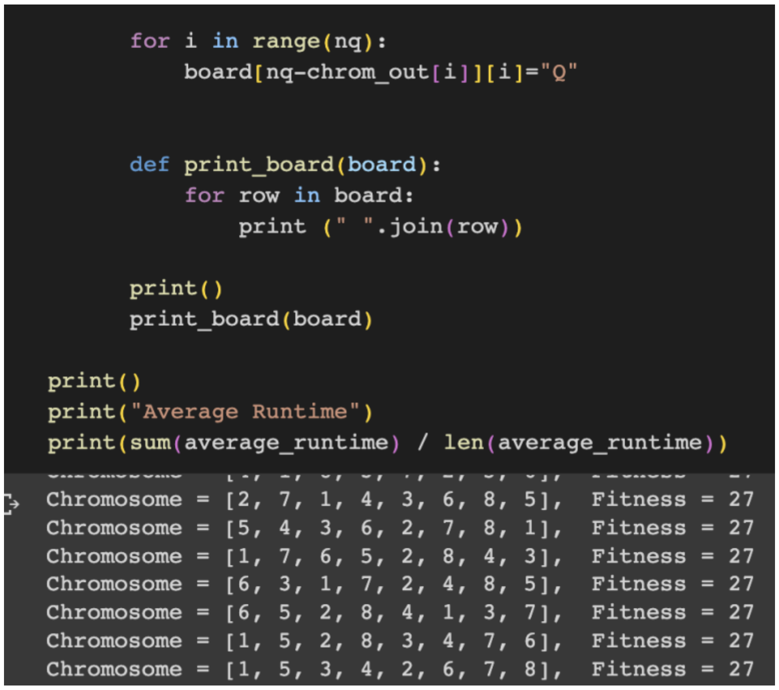
N=8

	Average Runtime
Hill Climbing	0.1541755437850952
Original Genetic Algorithm	273.0827447891235
Permutation Genetic Algorithm	0.3648380756378174

Genetic algorithm vs hill climbing speed

The genetic algorithm is much slower than the hill climbing algorithm. For instance, the above table compares the average runtimes of the solutions for N=8. The hill climbing algorithm found the global optimum in an average of 0.1541755437850952 vs an average runtime of 273.0827447891235 for the original genetic algorithm. Clearly, the genetic algorithm is far slower. This is due to the fact that the hill climbing algorithm with random restarts uses a heuristic to ascend the terrain of the solution space, finding better solutions until it converges to a clear local (or global) optimum. This process is relatively fast, and if it lands upon a local optimum, it simply restarts at a new location and tries again. Due to the limited size of the solution space, this works quite effectively. On the other hand, the genetic algorithm utilizes a fitness function to assess which solutions will have children and cross over to create a new set of solutions. However, the children are not guaranteed to be better solutions than the parents, and the algorithm's traversal in the solution space is not directly guided by a heuristic. This means that the genetic algorithm does not directly ascend the terrain of the solution space, and it relies upon informed/guided stochastic behavior to find the global optima. This stochastic process takes a long time without the direct guidance of the heuristic to ascend the terrain of the solution space.

Permutation off-spring solution
One possible solution to the problem of off-spring not being clear permutations is to do cross over and then replace duplicate values with the pool of unused values. The pool of unused values is initially populated with 1..N, and each element in the child solution is compared with this pool. If an element is found in both the solution and pool of values, it is removed from the pool. If it is only found in the solution (duplicate), its index is saved. After the entire solution string is traversed, the indices that were saved are used to replace the identified duplicates. The topmost element of the pool of unused values is used to replace to duplicate element. This is repeated until all duplicate elements are replaced (and the pool of unused values is empty). Furthermore, the mutation process swaps the value of two indices in the solution, rather than introducing a potential duplicate. The table above reports the average runtime of the permutation genetic algorithm (over five iterations) for the N=8 queens problem in comparison with the original genetic algorithm and hill climbing algorithm. The permutation genetic algorithm is far faster than the original genetic algorithm and has a runtime of about 2.5 times the runtime of the hill climbing algorithm on average. The N=8 queens problem is used rather than N=16 because local hardware and google colab can not handle the time complexity of the N=16 problem. The average runtime of the new genetic algorithm for the N=8 queens problem is 0.3648380756378174.



```
        for i in range(nq):
            board[nq-chrom_out[i]][i]="Q"


        def print_board(board):
            for row in board:
                print (" ".join(row))

        print()
        print_board(board)

    print()
    print("Average Runtime")
    print(sum(average_runtime) / len(average_runtime))
```

```
    Chromosome = [2, 7, 1, 4, 3, 6, 8, 5],  Fitness = 27
    Chromosome = [5, 4, 3, 6, 2, 7, 8, 1],  Fitness = 27
    Chromosome = [1, 7, 6, 5, 2, 8, 4, 3],  Fitness = 27
    Chromosome = [6, 3, 1, 7, 2, 4, 8, 5],  Fitness = 27
    Chromosome = [6, 5, 2, 8, 4, 1, 3, 7],  Fitness = 27
    Chromosome = [1, 5, 2, 8, 3, 4, 7, 6],  Fitness = 27
    Chromosome = [1, 5, 3, 4, 2, 6, 7, 8],  Fitness = 27
```