Computer & Systems Engineering Department

CSE 351: Intro To AI

Assignment 1: 8-Puzzle

Contributors:

	Name	ID
1	Adel Mahmoud Mohamed Abd El Rahman	20010769
2	Mohamed Hassan Sadek Abd El Hamid	20011539
3	Mahmoud Attia Mohamed Abdelaziz Zian	20011810
4	Mahmoud Ali Ahmed Ali Ghallab	20011811

---

1. Design decisions & Assumptions:

- **Programming language used:** Python.
- **State representation:** state is represented as a string of 9 characters with '0' for the empty cell, for example "125340678" is the representation of this state:

1	2	5
3	4	
6	7	8

- **Get the neighbors of a given state:** we first detect the '0' position in the string of the state and the number of the neighbors depend on this position so we typically swap the possible elements around the '0' with it as in this state above we can make 3 possible swaps of the zero with either 5, 4, or 8. And then we return an array containing all the possible neighbors of this state which will be at most 4 states if the zero is at the center or at least two neighbors when the zero is at any corner and the arrangement of the neighbors that we return is as follows {up, down, left, right}.
- **In case of the unsolvable state:** we get the number of inversions in the string representing the state and an inversion occurs if state[i] > state[j] for each i < j And if this number of inversions was even then this state is solvable and we can run the search algorithms on it but if it's not then we can't like the following state is non-solvable so we can't run the search algorithms on it

2. Algorithms & Data Structures:

(1) BFS:

a. Data Structure:

i. **Frontier:** A Queue (FIFO) and it's implemented in python built in collections as a **deque()** but we only use it as a one-ended queue.

ii. **Explored:** A hash set to keep track of the explored nodes and it's implemented in python as **set().**

iii. **Parent**: A HashMap to keep track of the parent of each node and it's implemented in python as **dict()** and its structure here in this algorithm is mapping the node-state-to a pair (the parent of this node, and the level that's this node is at).

iv. **Search_depth**: it's a variable to keep track of the maximum depth we got till we reach the goal.

b. Algorithm:

```
First initialize the queue with the start state, the parent map with
parent[start] = start, and an empty explored set.
while(frontier is not empty):
   curr = frontier.pop()
   explored.add(curr)
   if curr == goal: SUCCESS
   for neighbour in get_neighbor(curr):
      //to check that the neighbour is not in the frontier nor in the explored

   if neighbour not in parent:
         frontier.append(neighbour)
         parent[neighbour] = curr
```

(2) DFS:

a. Data Structures:

i. **Frontier**: A stack (LIFO) and it's implemented in python built in collections as a **deque()** but we only use it as a one-ended stack.

ii. **Explored:** A hash set to keep track of the explored nodes and it's implemented in python as **set().**

iii. **Parent:** hash map to keep track of the parent of each node and it's implemented in python as **dict()** and its structure here in this algorithm is mapping the node-state- to a pair (the parent of this node, and the level that's this node is at).

b. Algorithm:

```
First initialize the stack with the start state, the parent map with
parent[start] = start, and an empty explored set.
while (frontier is not empty):
   curr = frontier.pop()
   explored.add(curr)
   if curr == goal: SUCCESS
   for neighbour in get_neighbor(curr):
   //to check that the neighbour is not in the frontier nor in the explored
      if neighbour not in parent:
         frontier.append(neighbour)
         parent[neighbour] = curr
```

(3) A*:

a. *Data Structures*:

i. **Frontier (Priority Queue):** which is based on min heap data structure (Python Built in library called "*heapq*") and holds the pair (cost of the state and the state) where the key is the cost, and the value is the state.

ii. **Explored**: Set (HashSet): Explored hash set to keep track of the explored nodes and it's implemented in python as **set().**

iii. **Parent:** Map (HashMap): Parent hash map to keep track of the parent of each node and it's implemented in python as **dict()** and its structure here in this algorithm is mapping the node-state- to a Tuple of 3 (the parent of this node, cost from the starting node to this one and the level that's this node is at).

b. Algorithm:

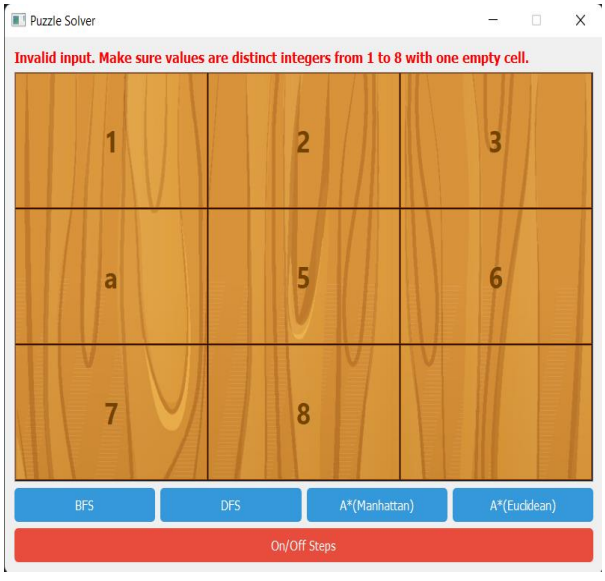
```
First initialize the priority queue with the pair: (heuristic(starting state), start
state), and the parent map with parent[start] = (start, 0), and an empty explored
set.
while (frontier is not empty):
   (cost, curr) = frontier.pop()
   If current in explored: continue
   actual cost = cost - heuristic(current)
   explored.add(curr)
   if curr == goal: SUCCESS
   for neighbour in get_neighbor(curr):
      g = actual cost + 1 // g is the new cost
      total cost = g + heuristic(neighbor)

   //to check that the neighbour is not in the frontier nor in the explored
   if neighbour not in parent:
      frontier.append((total cost, neighbour))
      parent[neighbour] = (curr, g)
   else if neighbor is not in explored and g < neighbor.cost)
      parent[neighbor] = (curr, g)
      frontier.add((total cost, neighbor))
```

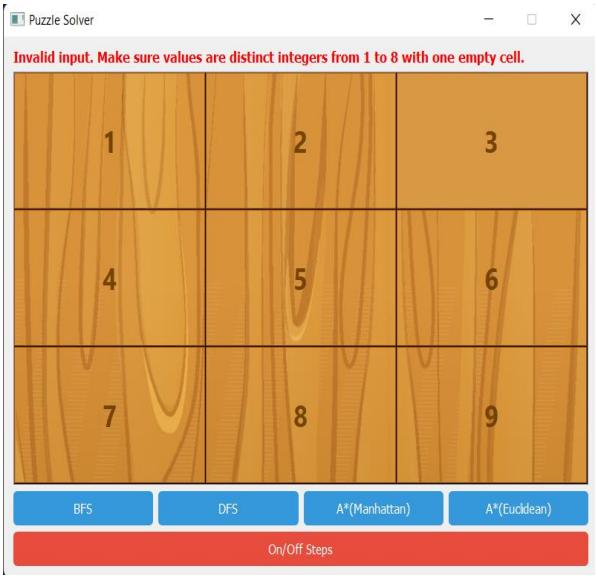
---

3. Extra work:

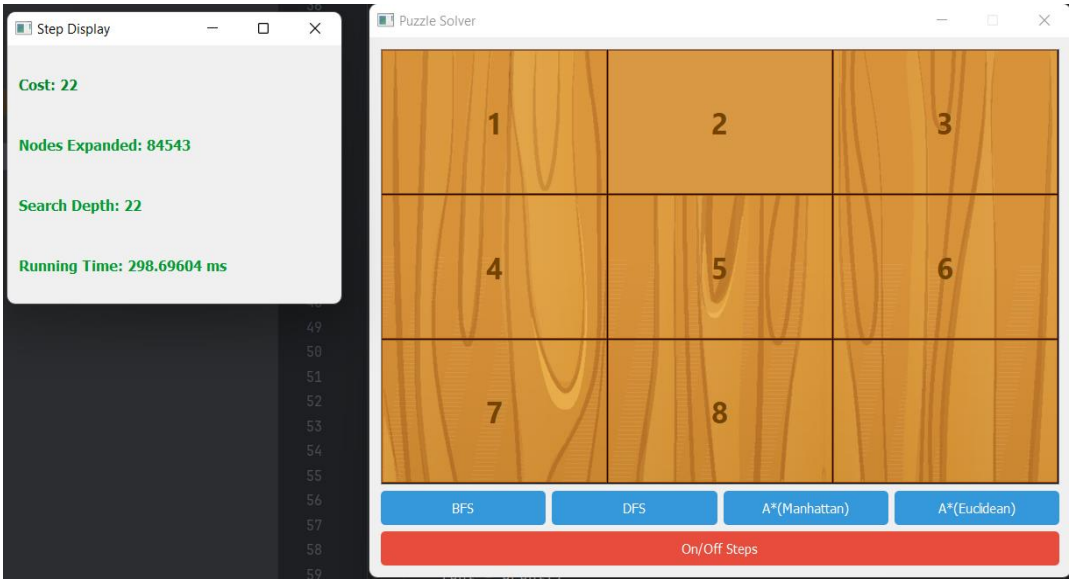
- We have added a friendly graphical user interface so the user can enter the state in an easier interactive way and choose one of the 4 algorithms to run and whether to show steps or not.
- Also, the steps are displayed in a traceable way that enables the user to see previous and next steps from start to goal.

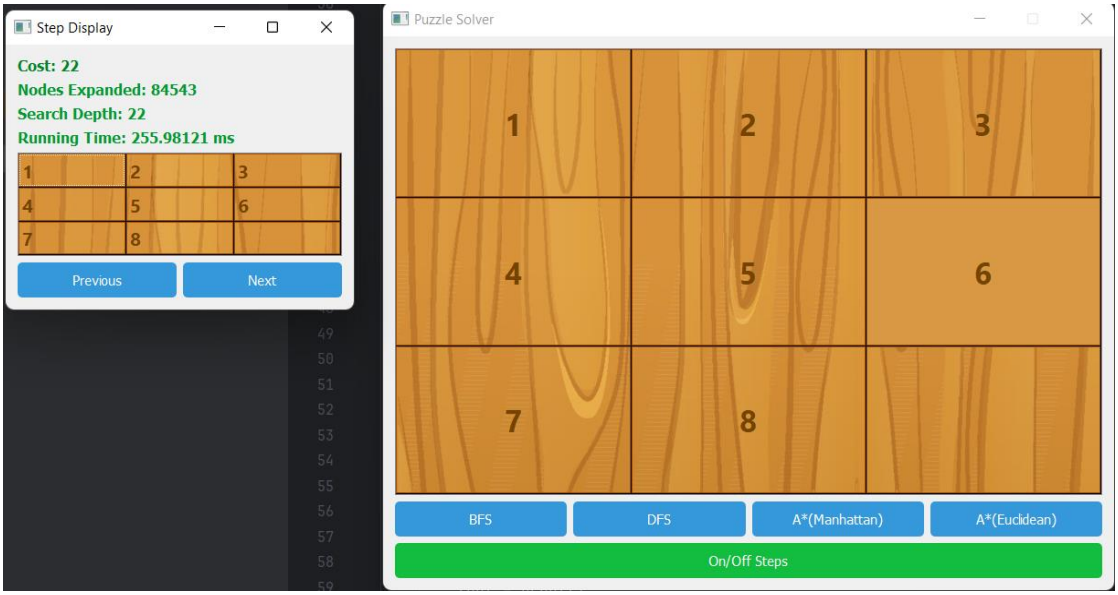
➢ Validate input:





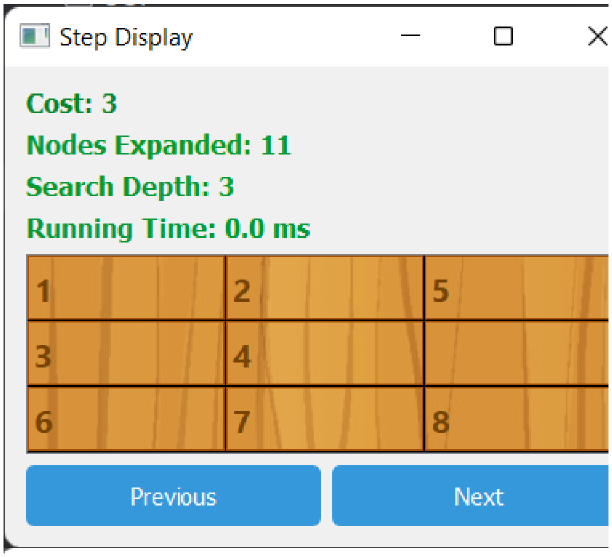
- On/off steps:

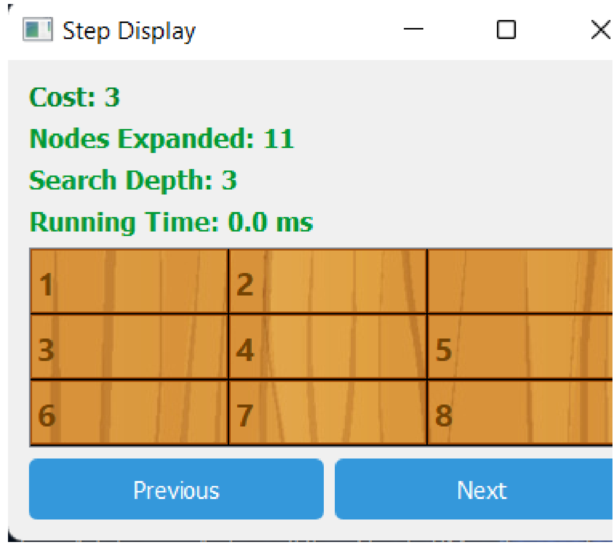

Step Display
Cost: 22
Nodes Expanded: 84543
Search Depth: 22
Running Time: 298.69604 ms
Puzzle Solver
1
2
3
4
5
6
7
8
BFS
DFS
A*(Manhattan)
A*(Euclidean)
On/Off Steps

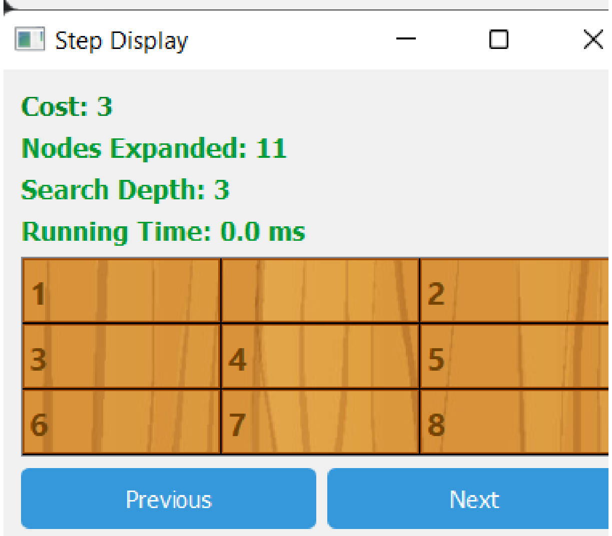

Step Display
Cost: 22
Nodes Expanded: 84543
Search Depth: 22
Running Time: 255.98121 ms
1
2
3
4
5
6
7
8
Previous
Next
Puzzle Solver
1
2
3
4
5
6
7
8
BFS
DFS
A*(Manhattan)
A*(Euclidean)
On/Off Steps

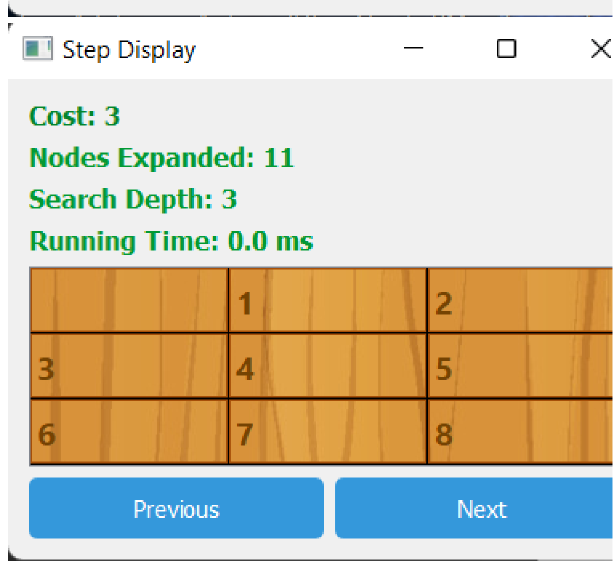
4. Sample Runs:

- Sample run 1 (with steps):
  - BFS

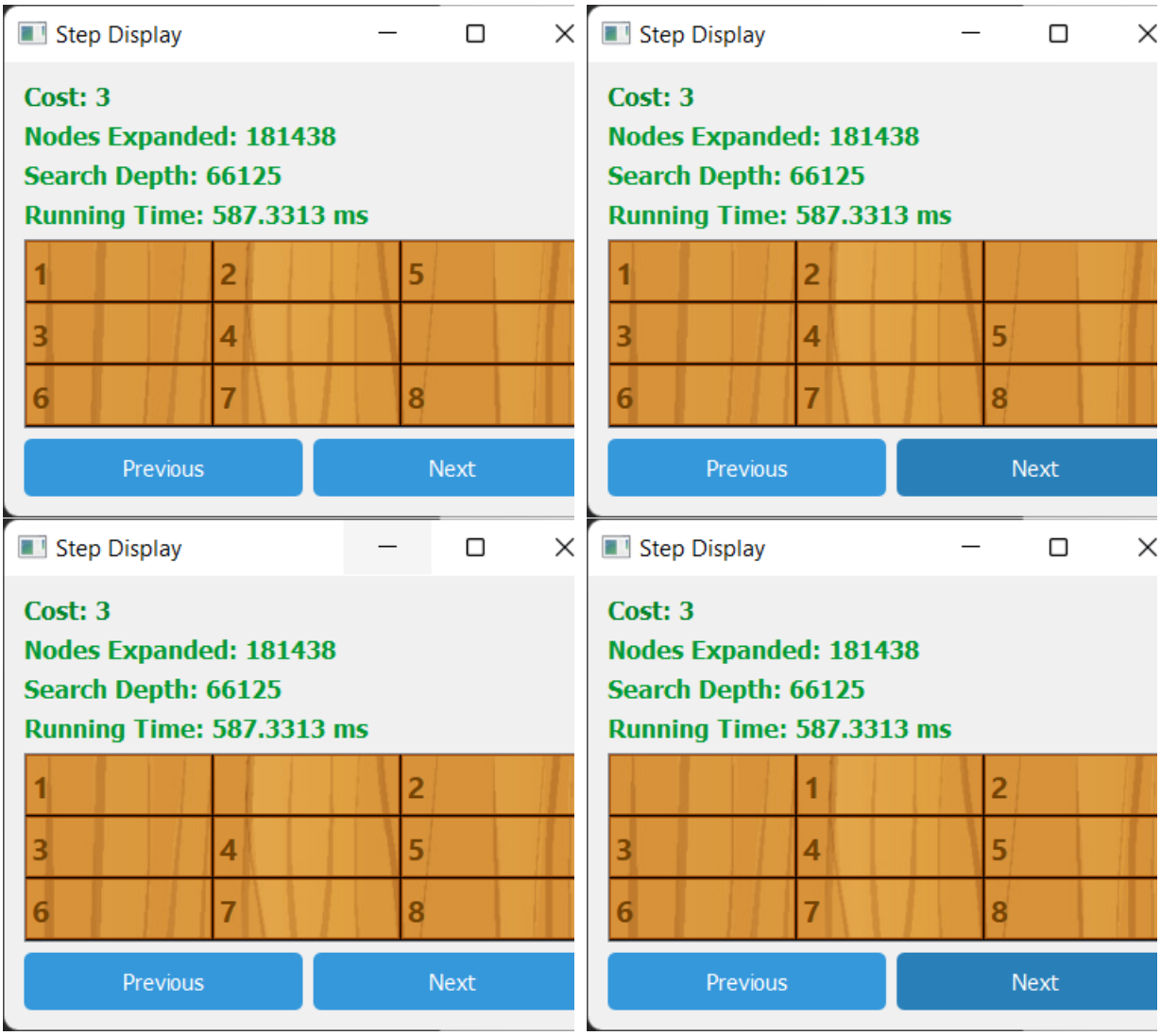




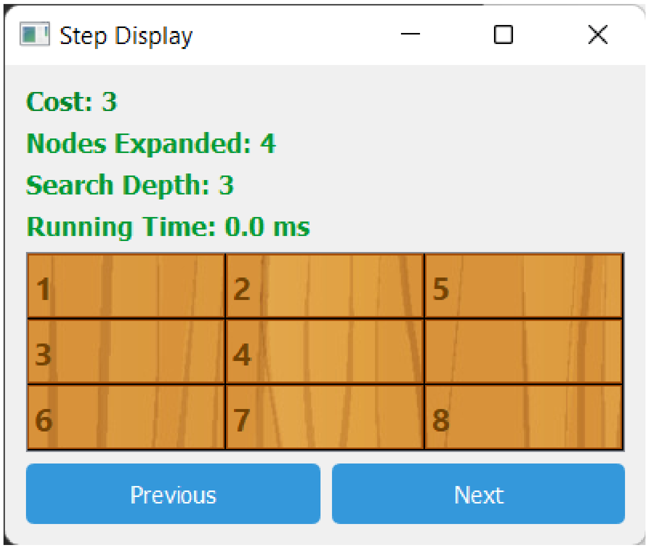


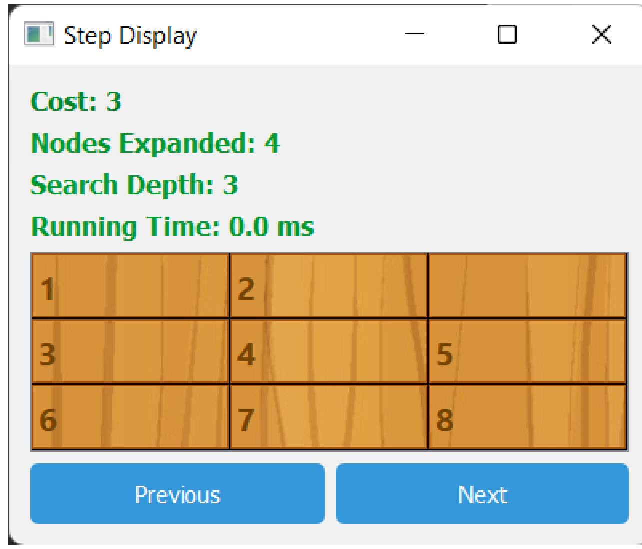


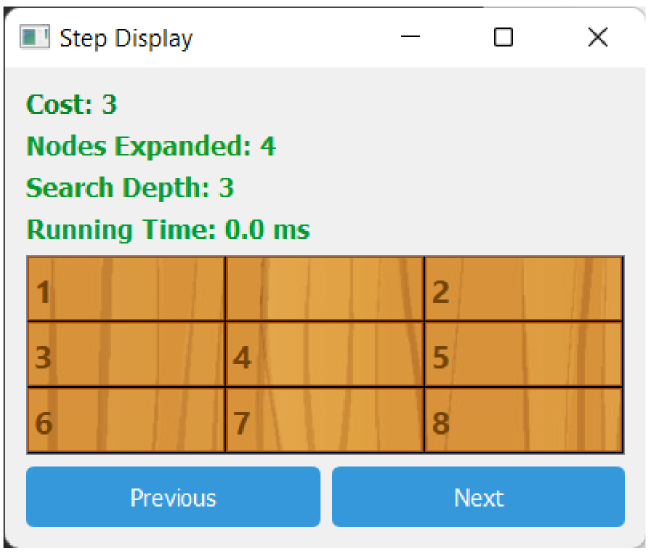
- DFS

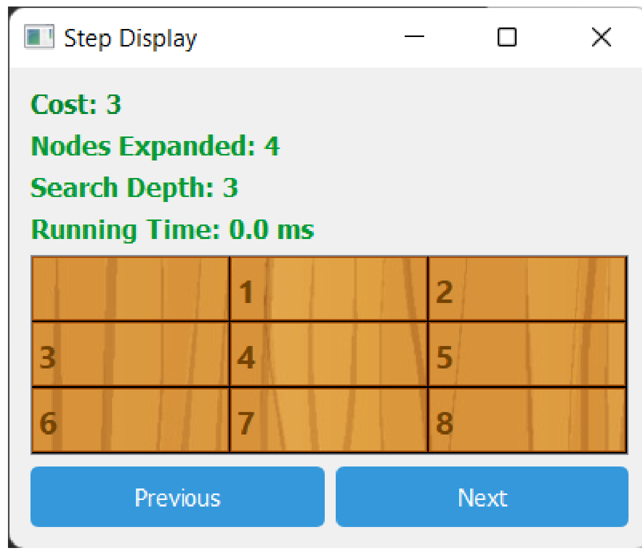

Step Display
Cost: 3
Nodes Expanded: 181438
Search Depth: 66125
Running Time: 587.3313 ms
1 2 5
3 4
6 7 8
Previous
Next
Step Display
Cost: 3
Nodes Expanded: 181438
Search Depth: 66125
Running Time: 587.3313 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 3
Nodes Expanded: 181438
Search Depth: 66125
Running Time: 587.3313 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 3
Nodes Expanded: 181438
Search Depth: 66125
Running Time: 587.3313 ms
1 2
3 4 5
6 7 8
Previous
Next

- A*(Manhattan)

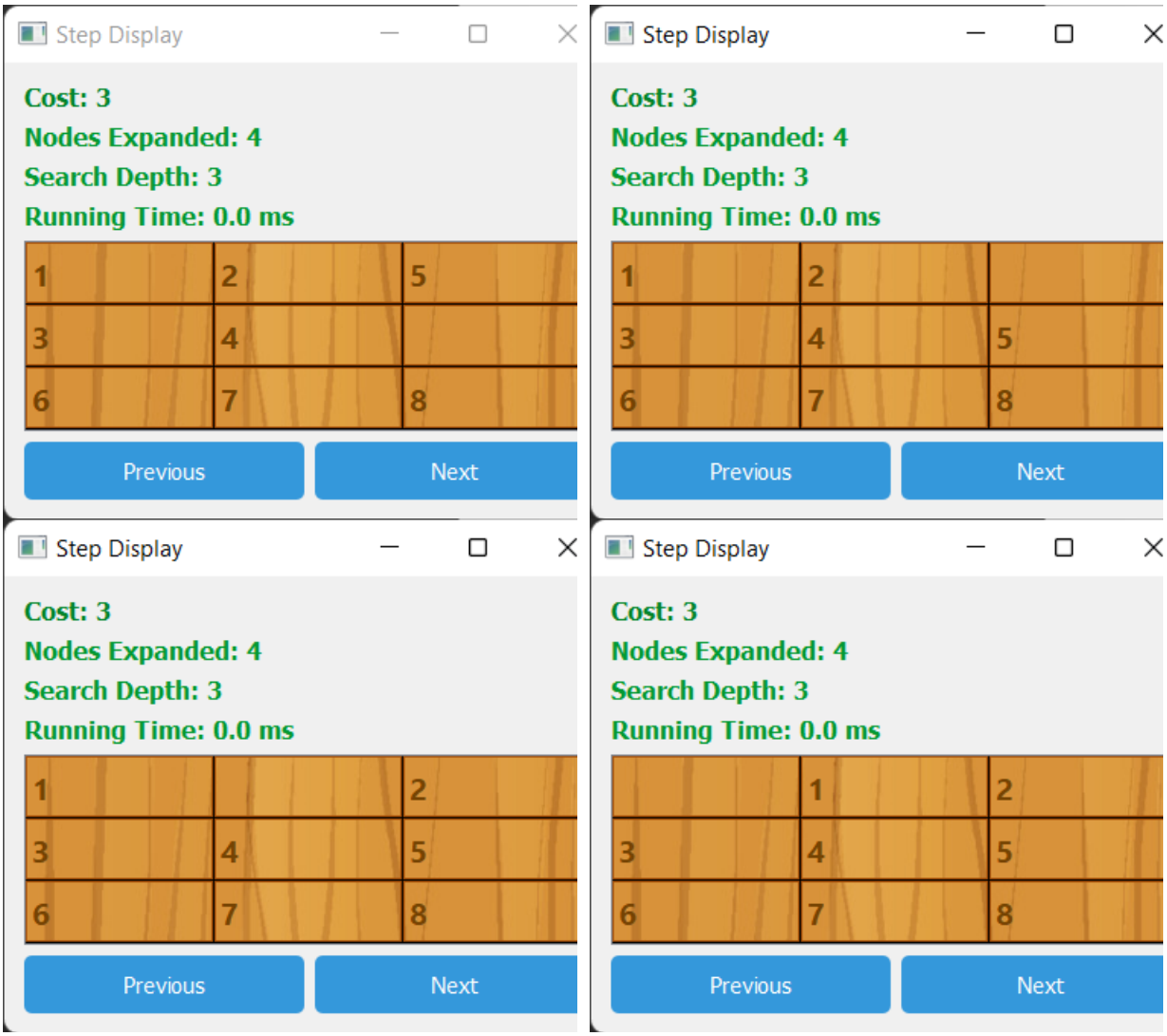

Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1
2
5
3
4
6
7
8
Previous
Next


Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1
2
3
4
5
6
7
8
Previous
Next

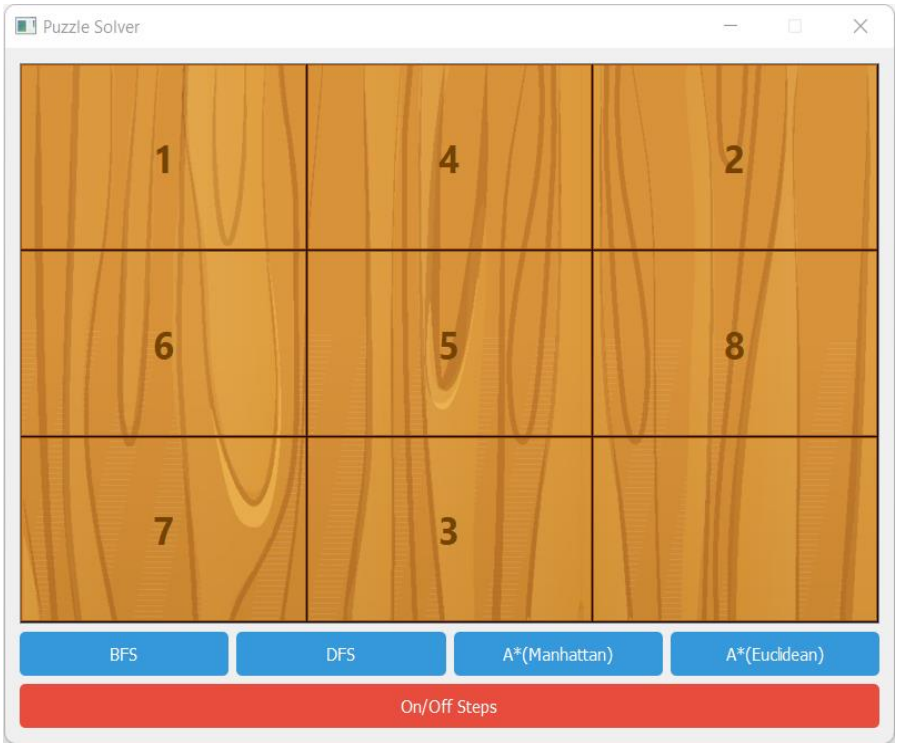

Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1
2
3
4
5
6
7
8
Previous
Next


Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1
2
3
4
5
6
7
8
Previous
Next

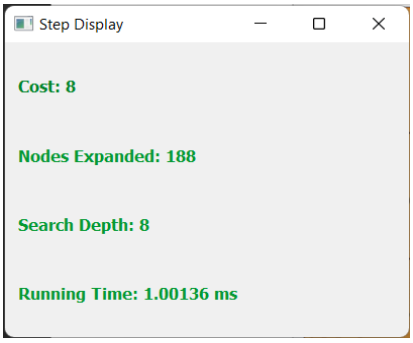
- A*(Euclidean)


Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1 2 5
3 4
6 7 8
Previous
Next
Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 3
Nodes Expanded: 4
Search Depth: 3
Running Time: 0.0 ms
1 2
3 4 5
6 7 8
Previous
Next

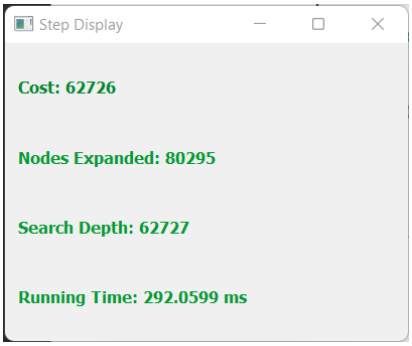
➢ Sample run 2.



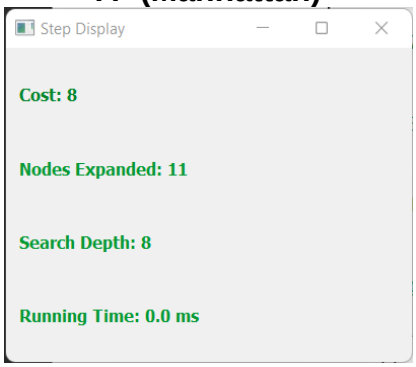
BFS



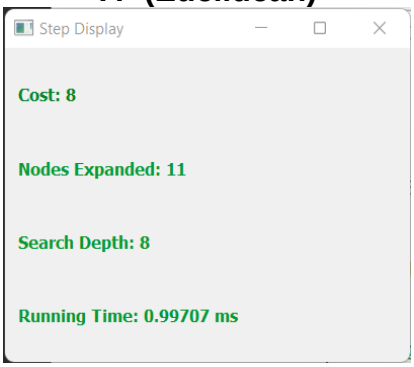
DFS



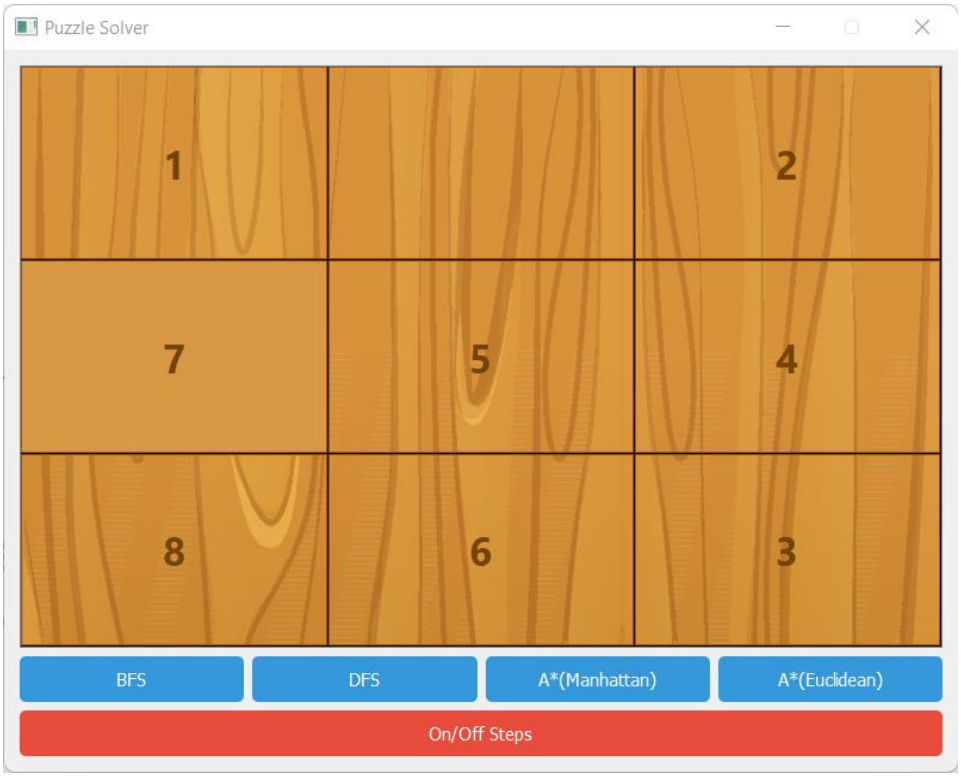
A* (Manhattan)



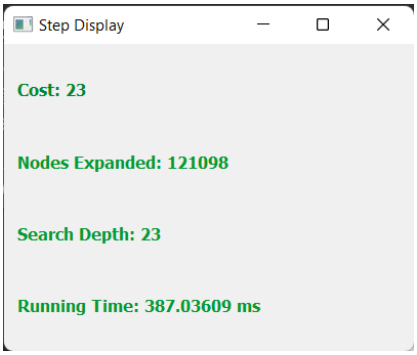
A* (Euclidean)



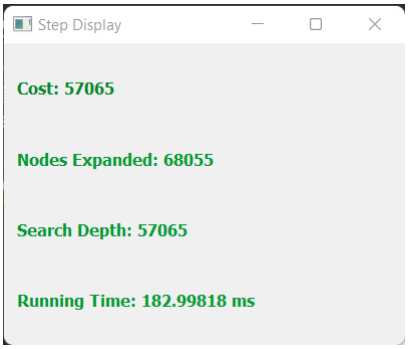
➢ Sample run 3:



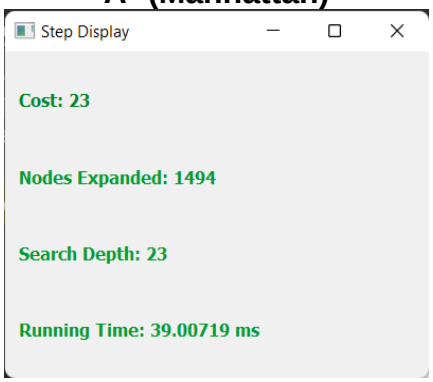
BFS



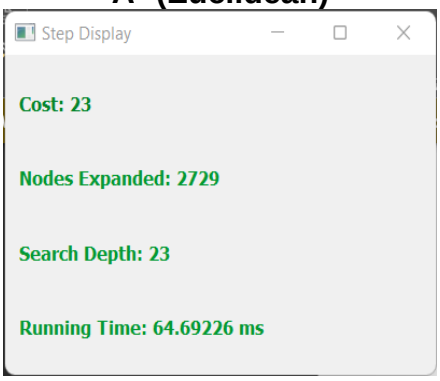
DFS



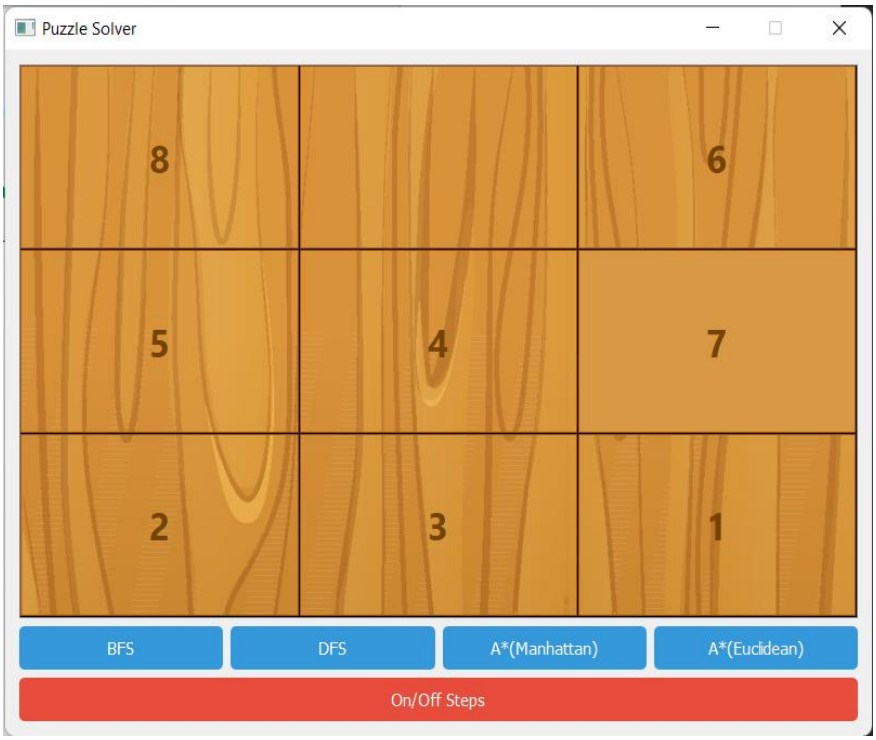
A* (Manhattan)



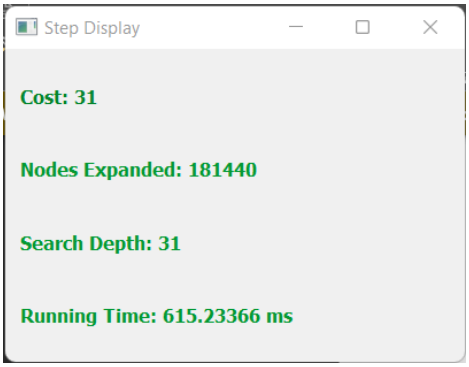
A* (Euclidean)



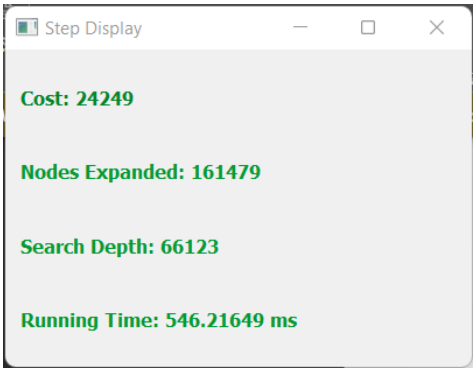
➢ Sample run 4:



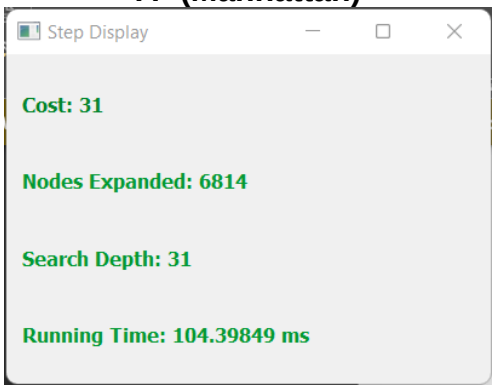
BFS



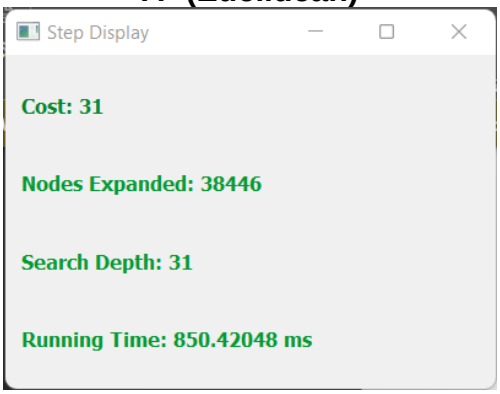
DFS



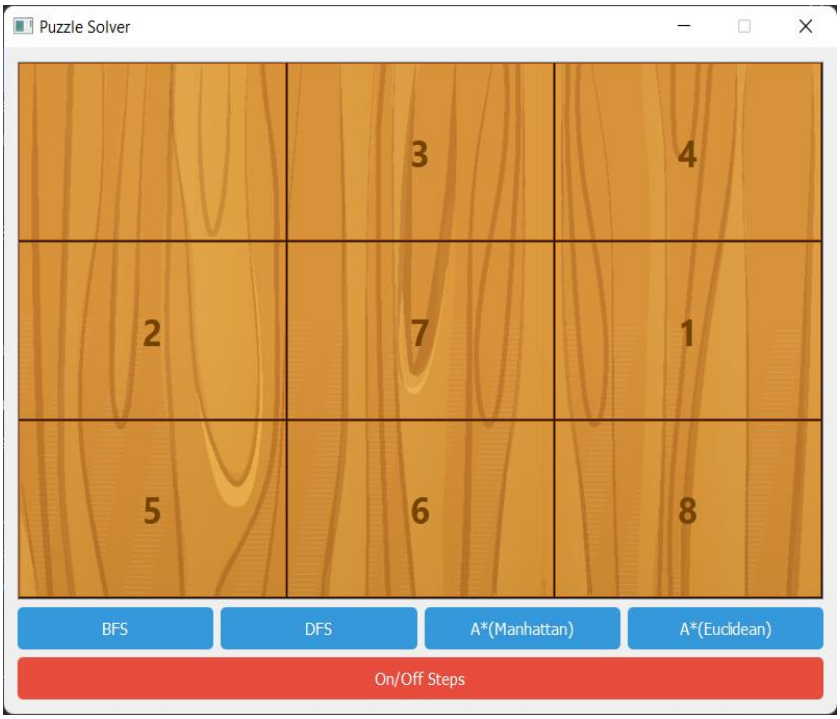
A* (Manhattan)



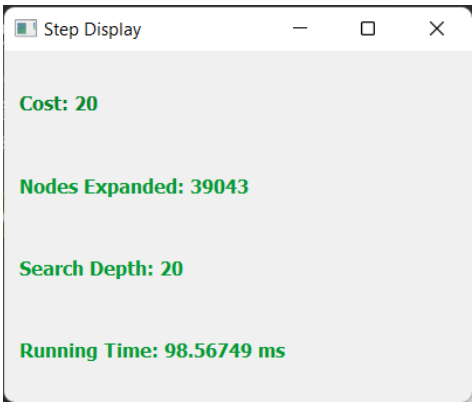
A* (Euclidean)



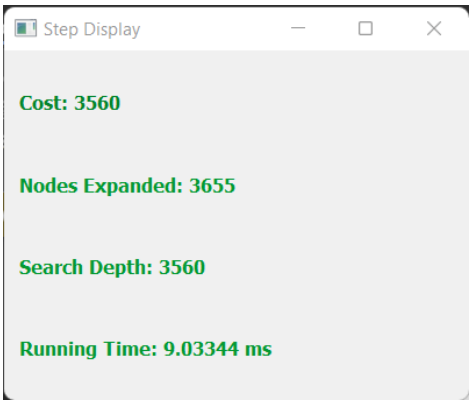
➢ Sample run 5:



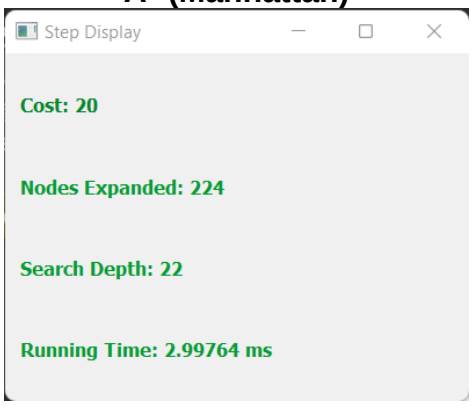
BFS



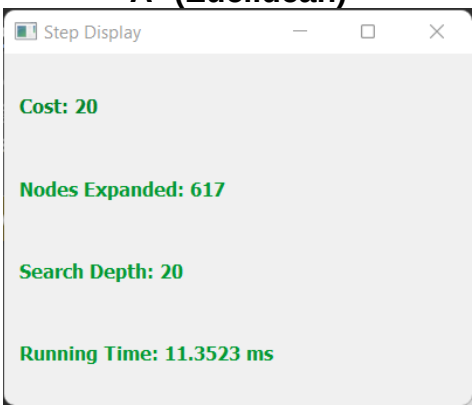
DFS



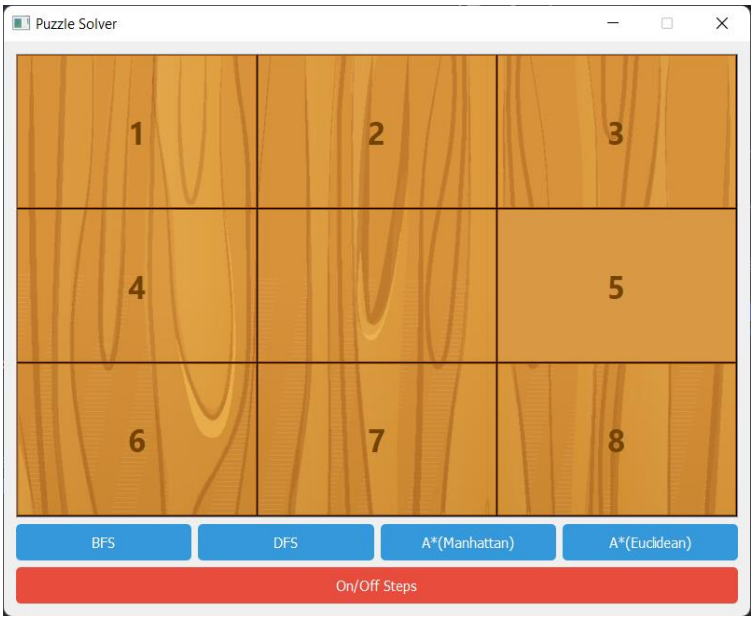
A* (Manhattan)



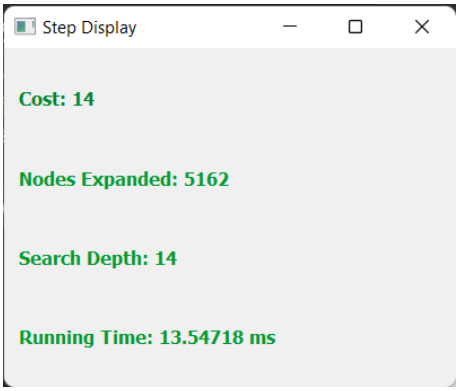
A* (Euclidean)



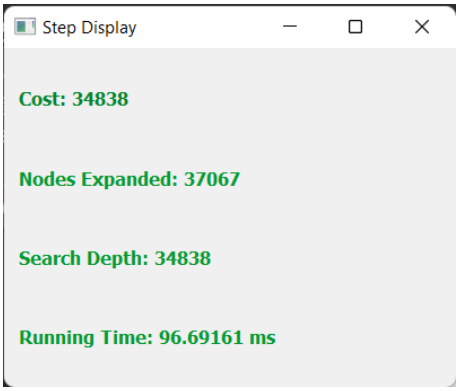
➢ Sample run 6:



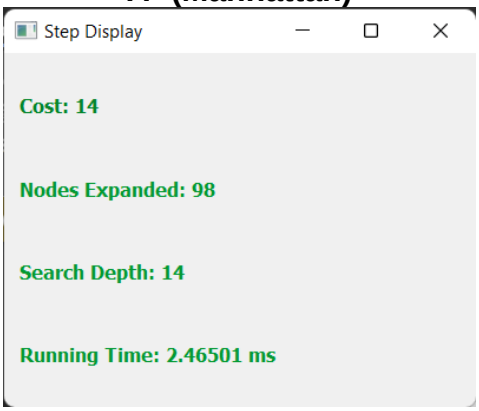
BFS



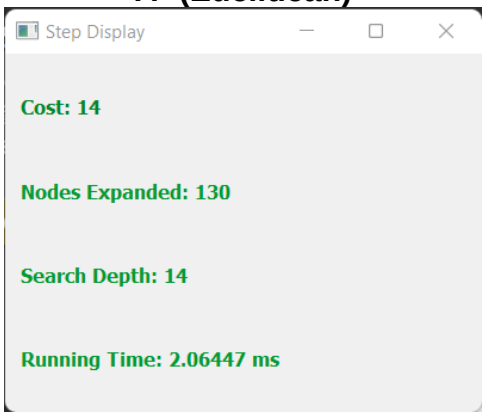
DFS



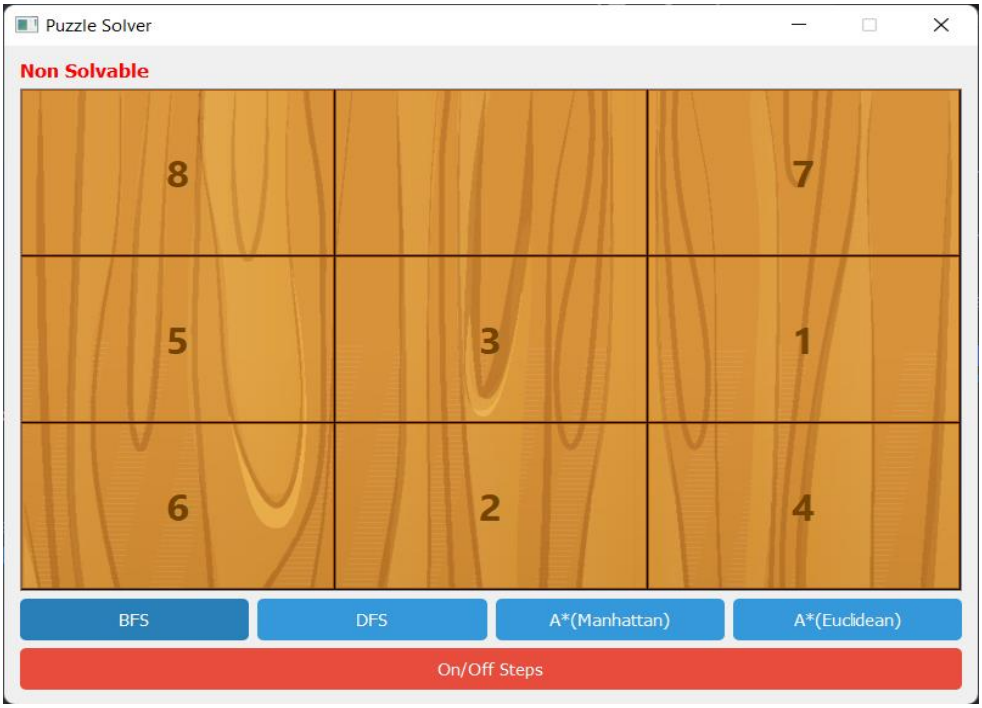
A* (Manhattan)



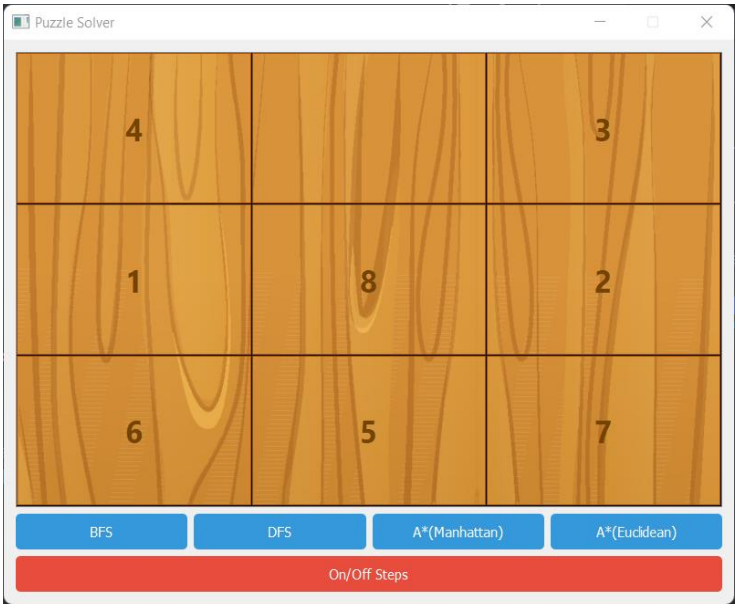
A* (Euclidean)



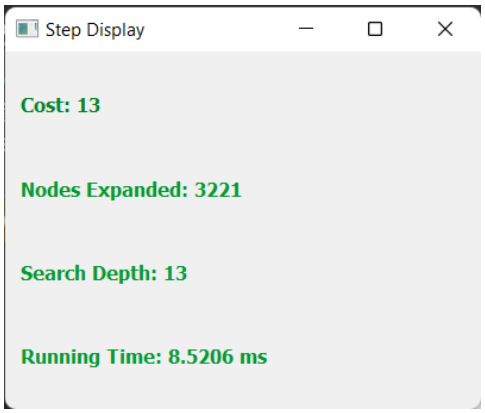
➢ Sample run 7:



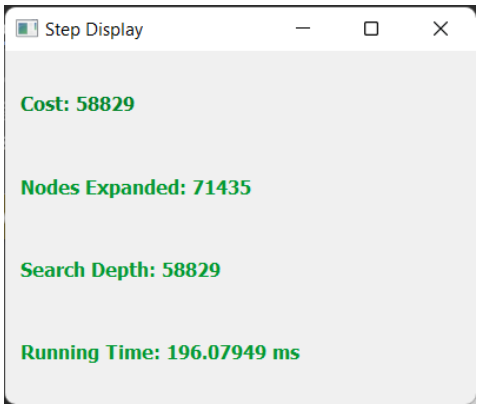
➢ Sample run 8:



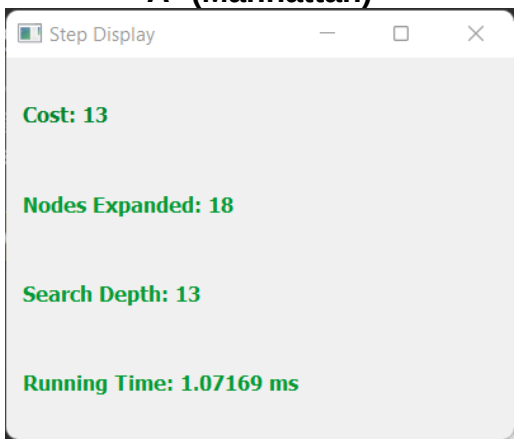
BFS



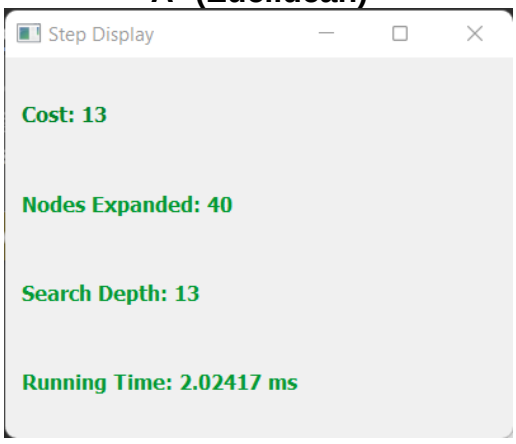
DFS



A* (Manhattan)



A* (Euclidean)



5. Comparisons:

Test1	BFS	DFS	A* (Manhattan)	A* (Euclidean)
Cost	8	62726	8	8
Nodes Expanded	188	80295	11	11
Search Depth	8	62727	8	8
Running Time (ms)	1.001	292.059	0	0.997
notes	➢ BFS and A* get the optimal path while the DFS not. ➢ DFS has the largest search depth and that's because it goes further to the left-most first so if the goal was shallow then we'll have large depth and all for nothing. ➢ DFS expanded the largest number of nodes and took a longer time. ➢ A* algorithm expands fewer nodes as it uses the heuristic function to estimate the distance to the goal.			

Test2	BFS	DFS	A* (Manhattan)	A* (Euclidean)
Cost	23	57065	23	23
Nodes Expanded	121098	68055	1494	2729
Search Depth	23	57065	23	23
Running Time (ms)	387.036	182.998	39.007	64.692
notes	➢ BFS and A* get the optimal path while the DFS not. ➢ BFS and $A^*_{\text{Manhattan and Euclidean}}$ get the same cost because the edge cost here equals to 1. ➢ BFS has the largest search depth because the goal here is to be found in a deep level and it traverses each node in the level, so more nodes in each level than the previous level. ➢ DFS takes less nodes than BFS to traverse because it traverses all nodes in a path and the goal might be in the first left paths. ➢ BFS takes more time and space than DFS. ➢ A* Manhattan gets the optimal path with the smallest running time as its heuristic equals the actual heuristic which guarantees the optimality for A* search.			

Test3	BFS	DFS	A* (Manhattan)	A* (Euclidean)
Cost	20	3560	20	20
Nodes Expanded	39043	3655	224	617
Search Depth	20	3560	22	20
Running Time (ms)	98.567	9.03344	2.998	11.352
notes	➢ Same notes as test 2			

Test4	BFS	DFS	A* (Manhattan)	A* (Euclidean)
Cost	31	24249	31	31
Nodes Expanded	181440	161479	6814	38446
Search Depth	31	66123	31	31
Running Time (ms)	615.23	546.2164	104.39	850.42
notes	➢ DFS has the largest search depth. ➢ BFS expands more nodes and takes more time than DFS. ➢ A* using the Manhattan heuristic gets a better estimation for the goal so expanding fewer nodes takes less time than The Euclidean heuristic, but both are better than BFS and DFS in terms of nodes expanded and running time.			

- When looking at the test cases above we can observe that in all of them, the A* using the Manhattan distance heuristic is more optimal than when using the Euclidean in terms of **the running time, the nodes expanded but both, of course, have the same length of the path** to the goal and the reason for this is as follows:

    - Manhattan distance is guaranteed to be a more **admissible** heuristic for the 8-puzzle problem. An admissible heuristic never overestimates the true cost to reach the goal-, and that's because the h(state) of the Euclidean distance is smaller than the one of the Manhattan so the h(state) of the Manhattan distance heuristic is closer to the h*(state) and this is the actual distance from the state to the goal which is practically hard to find.

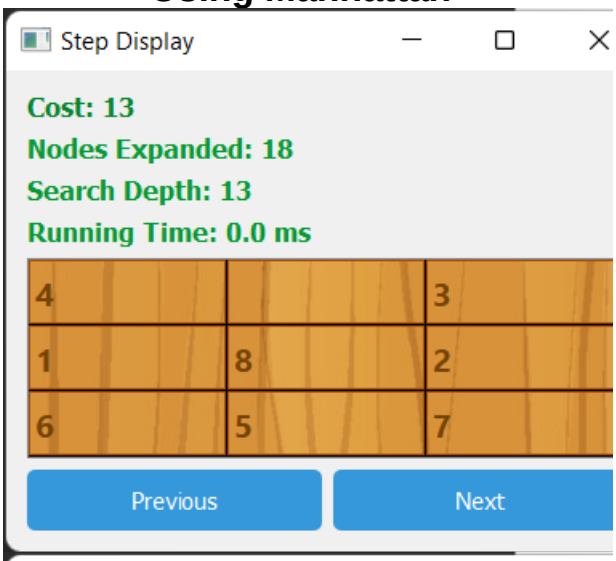
    - Additionally, the 8-puzzle operates on a grid with discrete movements (up, down, left, right) rather than continuous movements. Manhattan distance directly aligns with these constraints because it only counts the number of moves (horizontal and vertical) required to move each tile to its goal position. Euclidean distance, on the other hand, considers the diagonal movements, which are not allowed in the 8-puzzle. Therefore, Euclidean distance may extremely underestimate the cost to reach the goal So Manhattan distance is a more admissible heuristic than Euclidean but both of them are admissible so both get the optimal path (with minimum cost) but Manhattan expand fewer nodes to reach that goal.

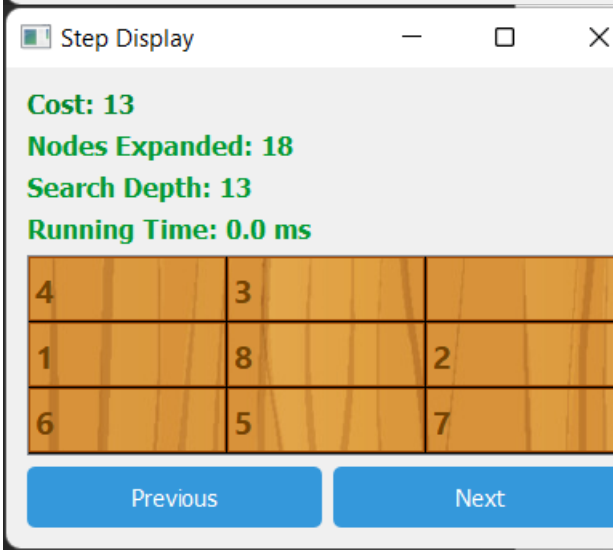
    - And in the following example we can see all of what we are saying above:

        - Solving using Manhattan expands fewer nodes (18 nodes) than Euclidean (4o nodes)

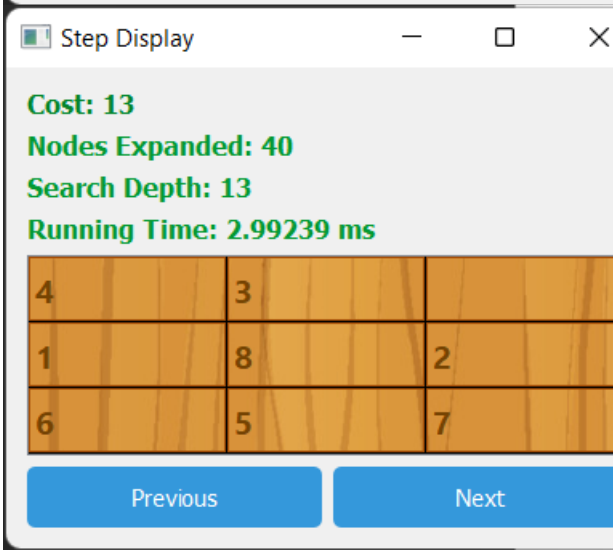
- Manhattan takes less time to reach the goal (0 ms) than Euclidean (2.99 ms)
- Both reach the optimal goal with minimum cost (13) but each one has a different route to reach that goal according to the heuristic function used and the selected nodes to expand at each step.

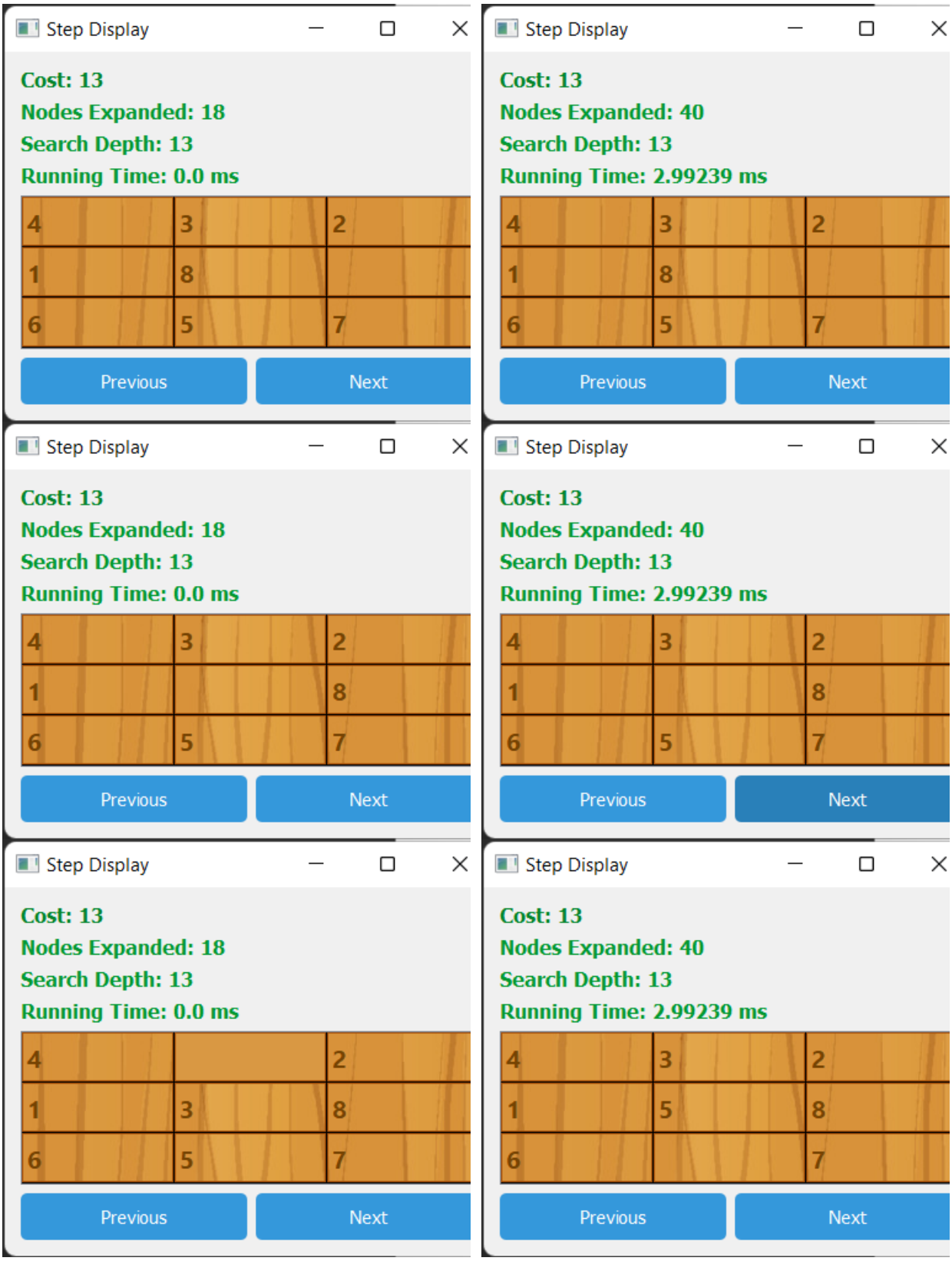
Using Manhattan

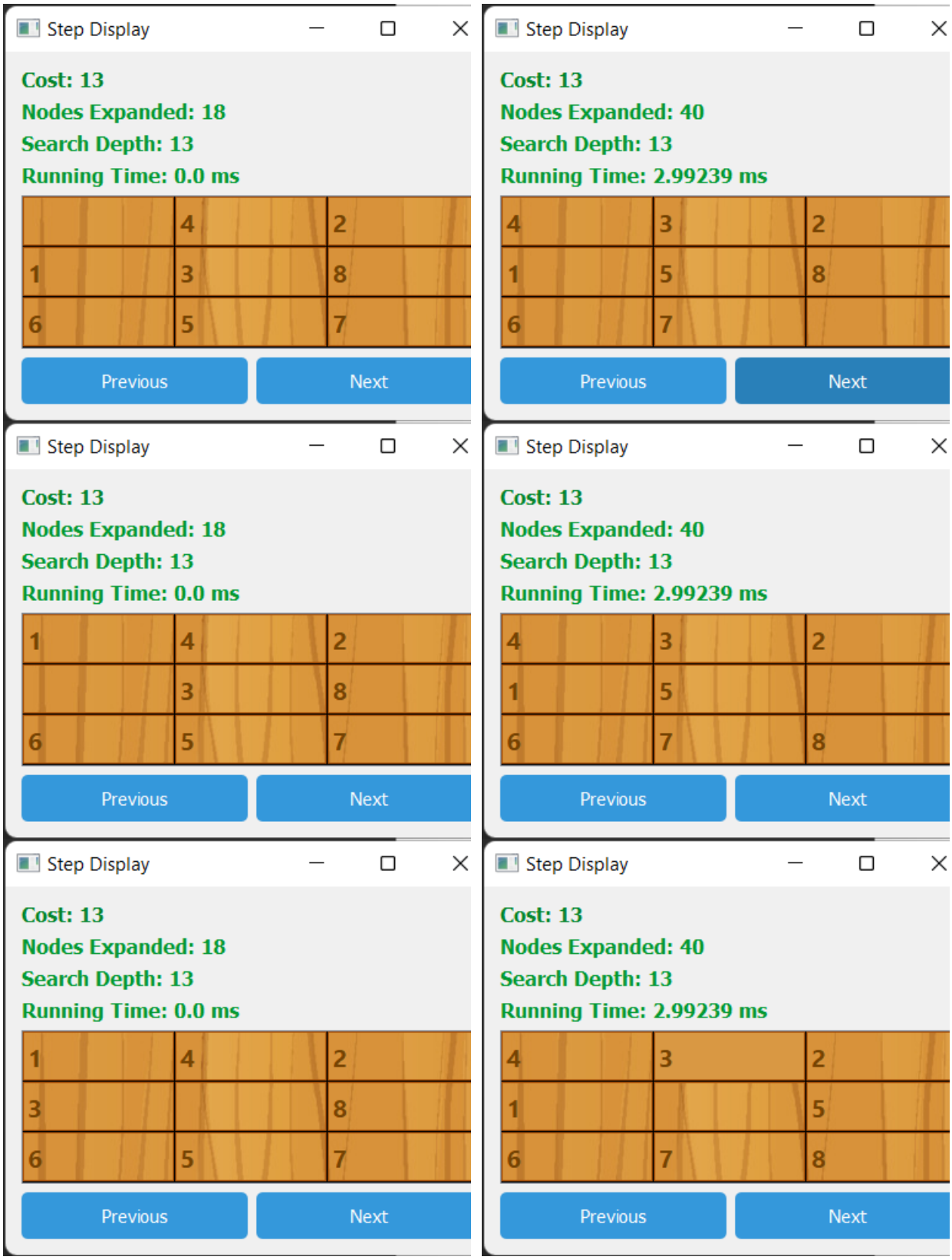


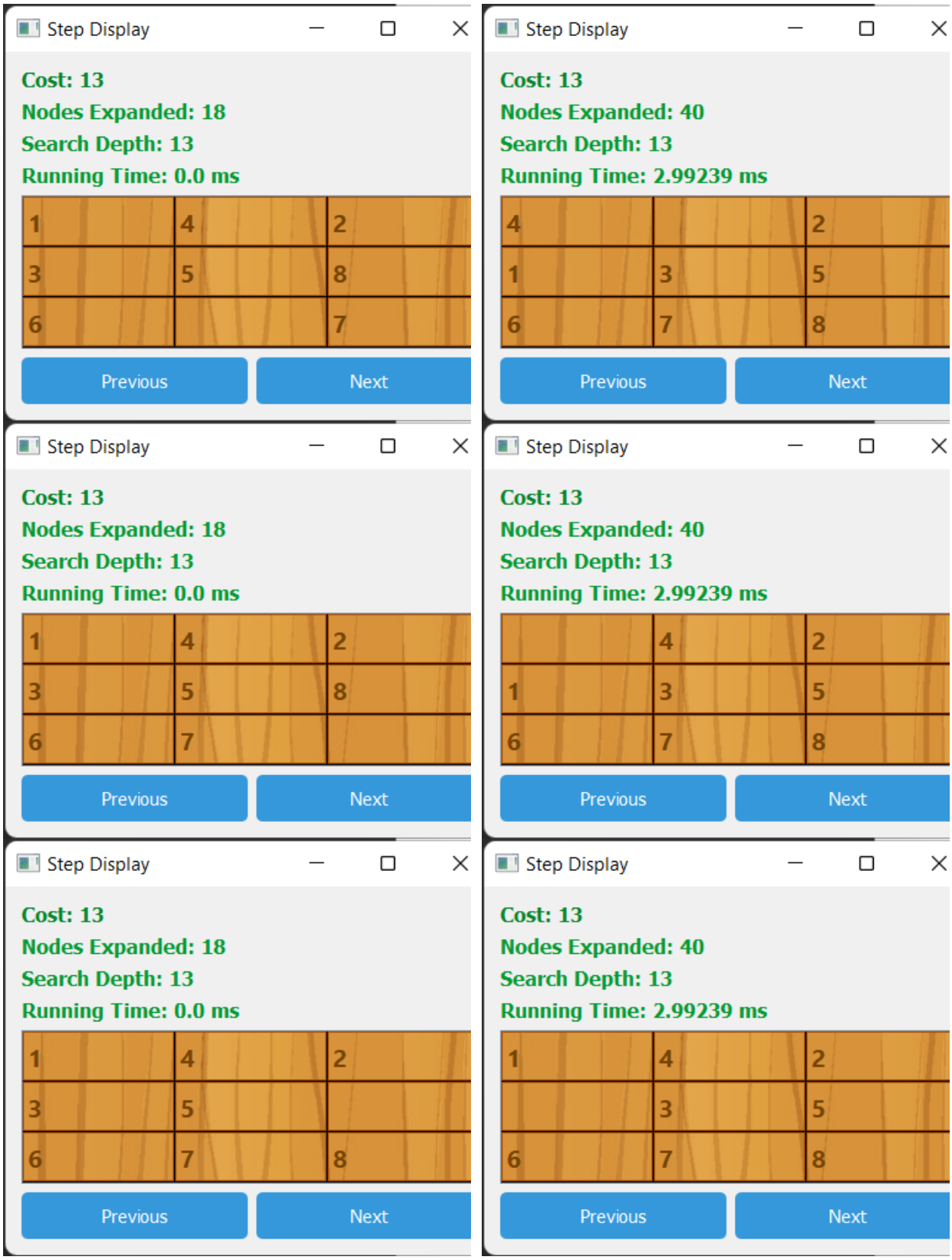


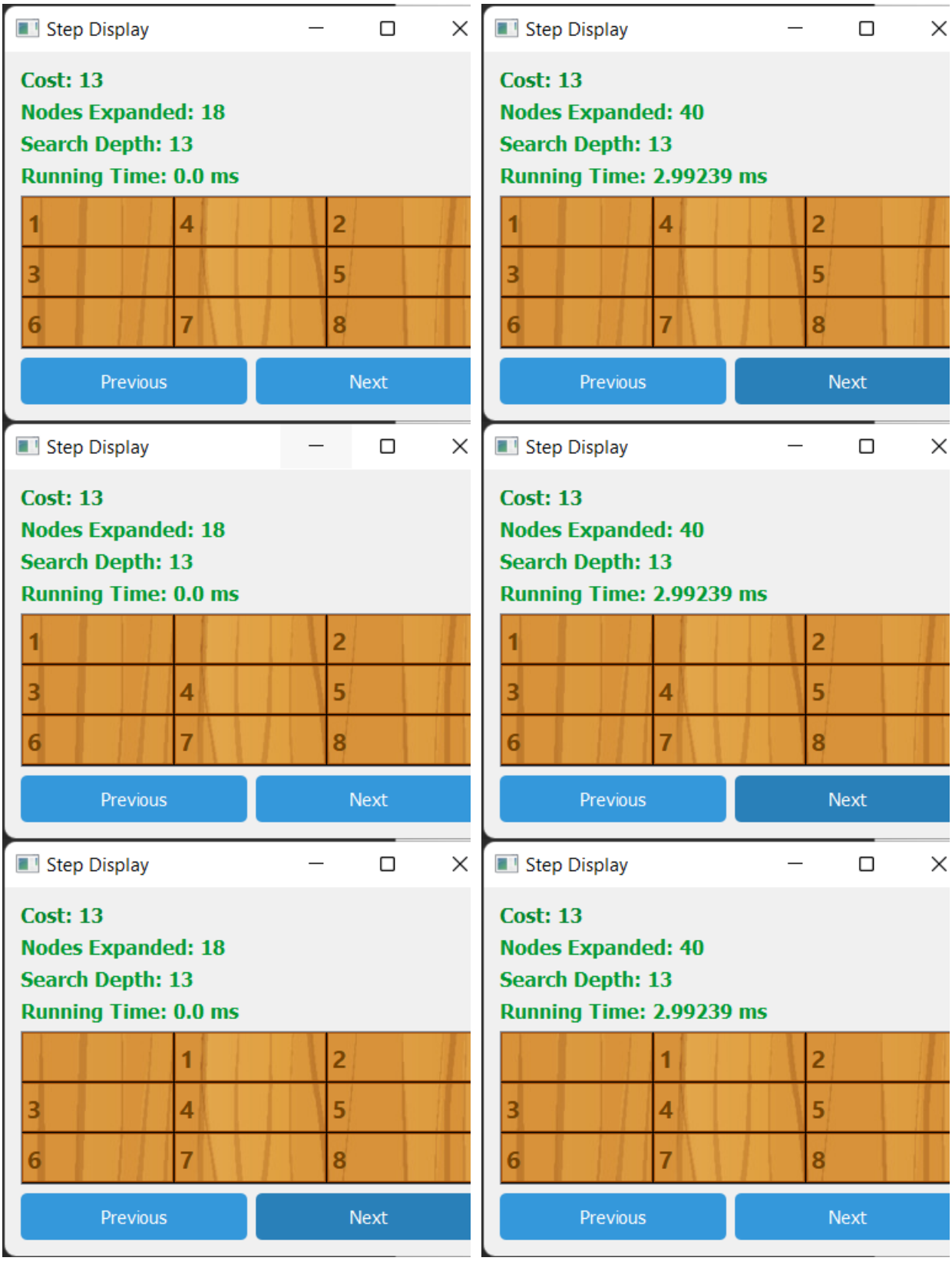
Using Euclidean




Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
4 3 2
1 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 3 2
1 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
4 3 2
1 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 3 2
1 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
4 2
1 3 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 3 2
1 5 8
6 7
Previous
Next


Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
4 2
1 3 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 3 2
1 5 8
6 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 4 2
3 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 3 2
1 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 4 2
3 8
6 5 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 3 2
1 5
6 7 8
Previous
Next


Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 4 2
3 5 8
6 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 2
1 3 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 4 2
3 5 8
6 7
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
4 2
1 3 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 4 2
3 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
1 4 2
3 5
6 7 8
Previous
Next


Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 4 2
3 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
1 4 2
3 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 18
Search Depth: 13
Running Time: 0.0 ms
1 2
3 4 5
6 7 8
Previous
Next
Step Display
Cost: 13
Nodes Expanded: 40
Search Depth: 13
Running Time: 2.99239 ms
1 2
3 4 5
6 7 8
Previous
Next