Alexandria University
Faculty of Engineering
Computer & Systems Engineering Department

AI Lab 1: 8-Squares

Name	ID
Karim Fathy Abdelaziz	20011116
Mohamed Amr Abdelfattah	20011675
Hager Ashraf Mohamed	21011505
Omar Mahmoud Abdel Wahab	20011027

Algorithms:

- **DFS:**

```
Function DFS()
    Frontier = stack.insert(FirstState)
    ParentSet = HashMap.put(FirstState , FirstState)
    Explored = hashset
    While not frontier.empty()
        State = frontier.pop()
        Explored.add(state)
        If(Goal(state))
            Return success
        For neighbors in state.neighbors()
            If neighbor not in frontier and explored
                Frontier.insert(neighbor)
                ParentSet.put(neighbor , state)
    Return failed
```

- **BFS:**

```
Function BFS()
    Frontier = queue.insert(FirstState)
    ParentSet = HashMap.put(FirstState , FirstState)
    Explored = hashset
    While not frontier.empty()
        State = frontier.pop()
        Explored.add(state)
        If(Goal(state))
            Return success
        For neighbors in state.neighbors()
            If neighbor not in frontier and explored
                Frontier.insert(neighbor)
                ParentSet.put(neighbor , state)
    Return failed
```

- **A*:** (cost = g(n)+h(n) where g(n) is the total actual cost from the FirstState to state/node n , h(n) is the heuristic/estimated cost from node/state n to the goal

h(n) is calculated in 2 ways:

Euclidean : sqrt((current cell:x - goal:x)2 + (current cell.y - goal:y)2)

anhattan: abs(current cell:x - goal:x) + abs(current cell:y - goal:y)

```
Function A*()
    Frontier = PriorityQueue.insert(FirstState,cost)
    Explored = hashset
    Infrontier = hashset.add(FirstState)
    ParentSet = HashMap.put(FirstState , (FirstState,0))
    While not frontier.empty()
        state = frontier.popmin()
        Infrontier.remove(state)
        If(state in explored)
            Continue
        Explored.add(state)
        If(Goal(state))
            Return success
        For neighbors in state.neighbors()
            If neighbor not in frontier and explored
                Frontier.insert(neighbor)
                Infrontier.add(neighbor)
                ParentSet.put(neighbor,(state,g(n))
            Else if neighbor in frontier and newCost < oldCost
                Frontier.insert(neighbor,newCost)
                ParentSet.insert(neighbor,(state,new g(n))
    Return failed
```

Data Structure:

	DFS	BFS	A*
Frontier	Stack<Long>	Queue<Long>	PriorityQueue<FrontierPair>
Explored	HashSet<Long>	HashSet<Long>	HashSet<Long>
Parent	HashMap<Long, Long>	HashMap<Long, Long>	HashMap<Long,parentPair>
Nodes Depth	HashMap<Long, Integer>	HashMap<Long, Integer>	___
Path	ArrayList<Grid>	ArrayList<Grid>	ArrayList<Grid>
Infrontier	___	___	HashSet<Long>

Assumptions & Other Details:

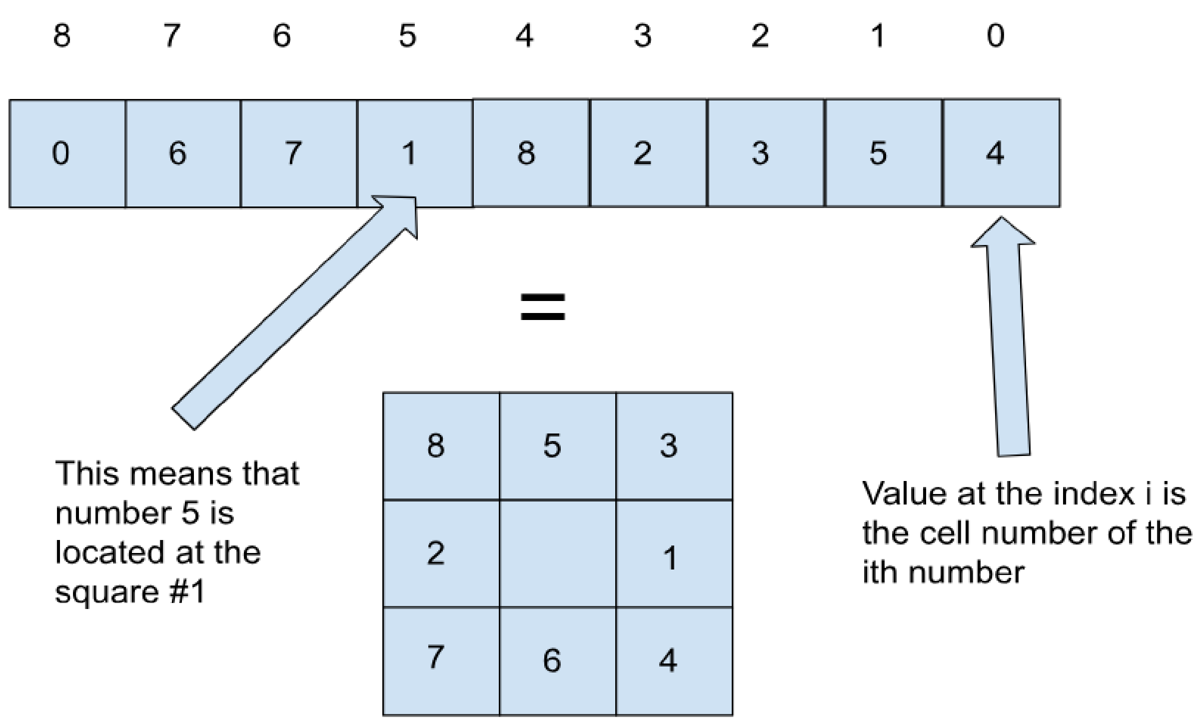
- We print the whole path in output.txt instead of console but we can print it in console state by state.
- We create a class in A* parentPair which contains the parent and the cost (g(n) only) for each state and use it with Parent.
- Displyed the path to solution in GUI only not in the console

State Representation

- Grid cells are numbered from 0 to 8, starting from the top left cell, moving row by row.

#0	#1	#2
#3	#4	#5
#6	#7	#8

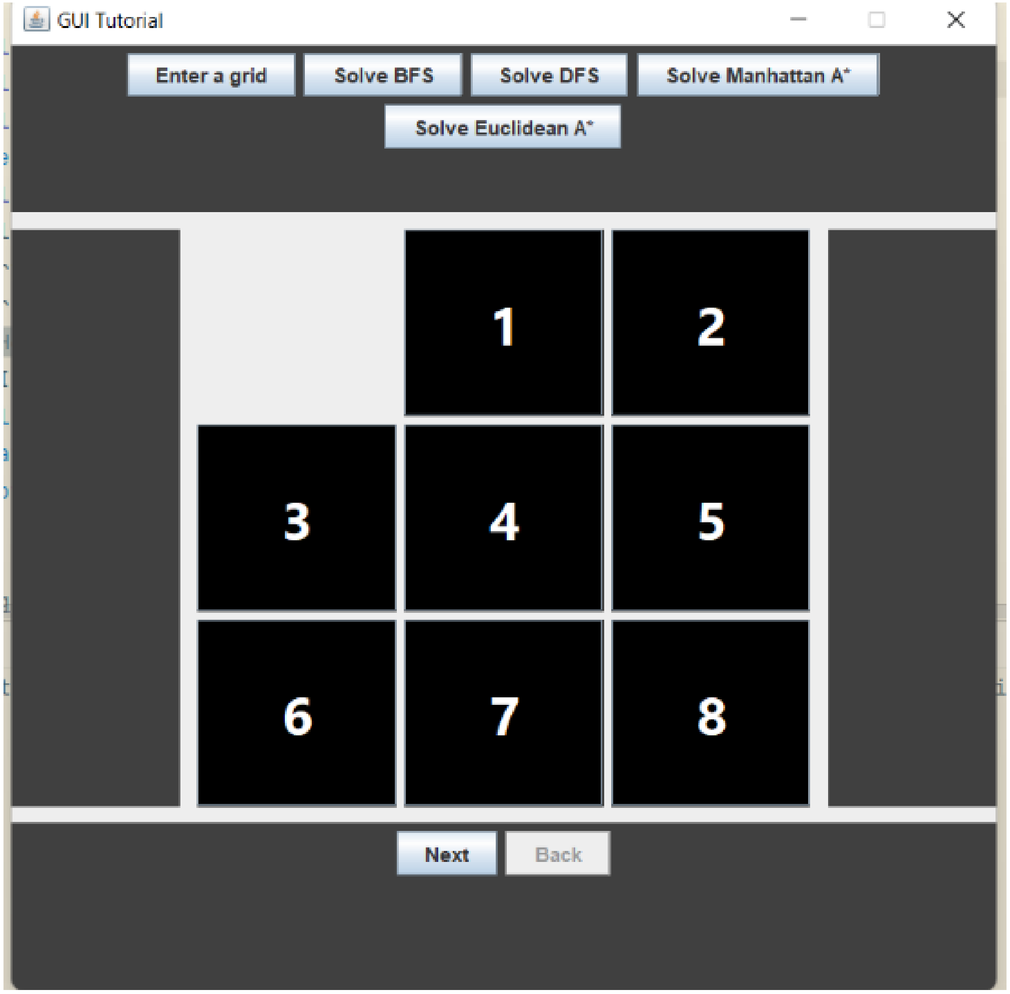
- We have 9 types of ‘numbers’ (empty & 1 to 8), each can be in one of 9 cells.
- The number of the cell is represented by 4 bits (a hex digit)
- Therefore, the total number of bits required = 4*9 = 36 (more than a 4-byte integer by 4 bits).
- So, we have chosen to store the grid as an 8-byte variable of type long.
- The figure below shows how the actual grid is mapped to our representation.



- This enables us to use bit operations to quickly access positions of the numbers and modify them.

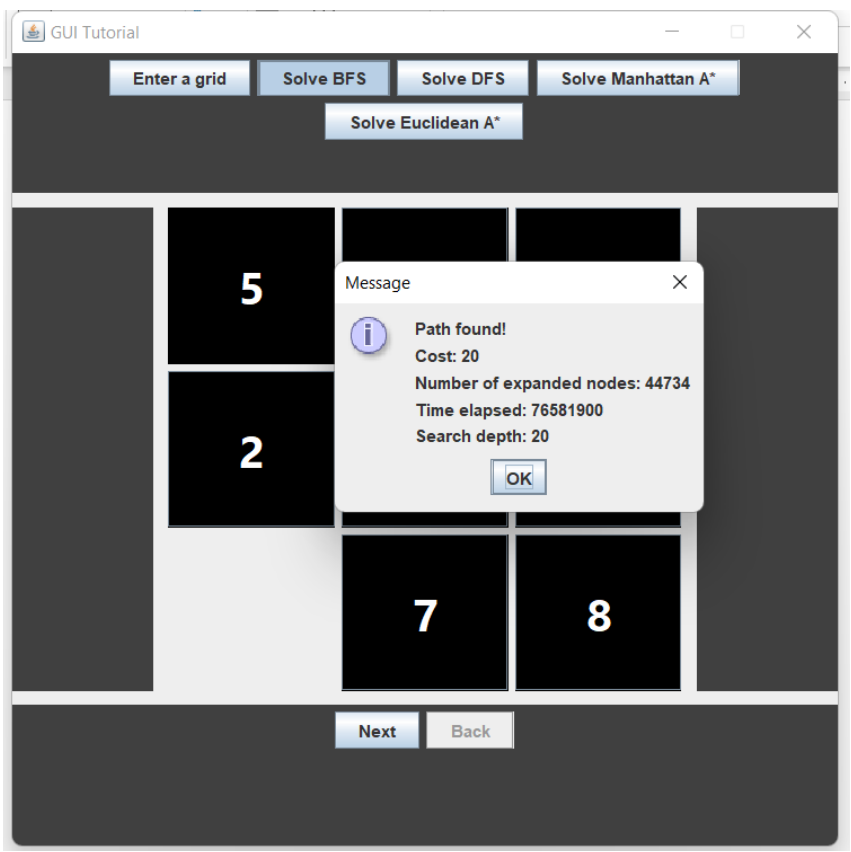
Sample Runs

Application interface

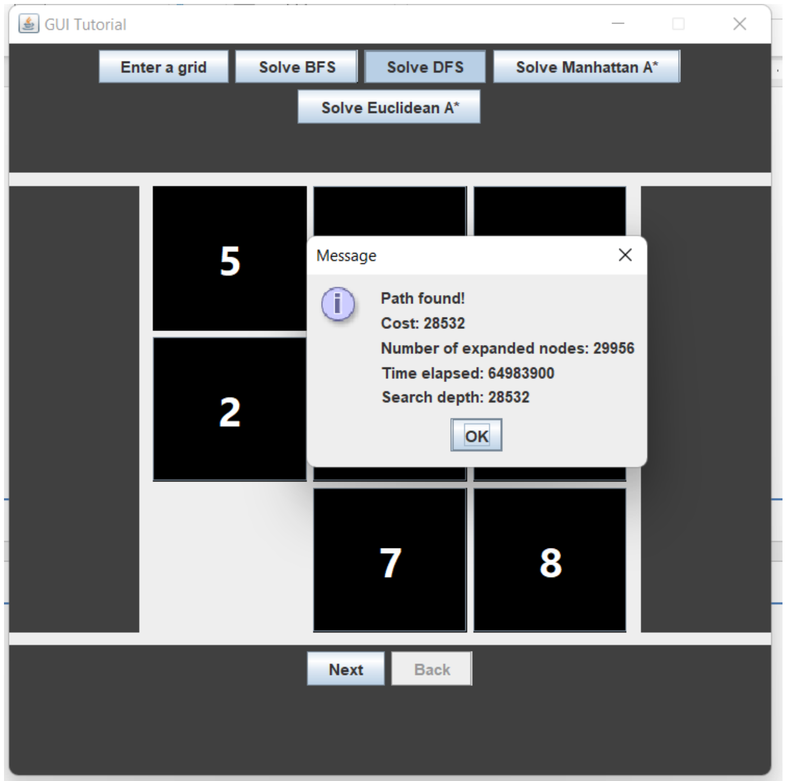


- Entering the initial state of the grid

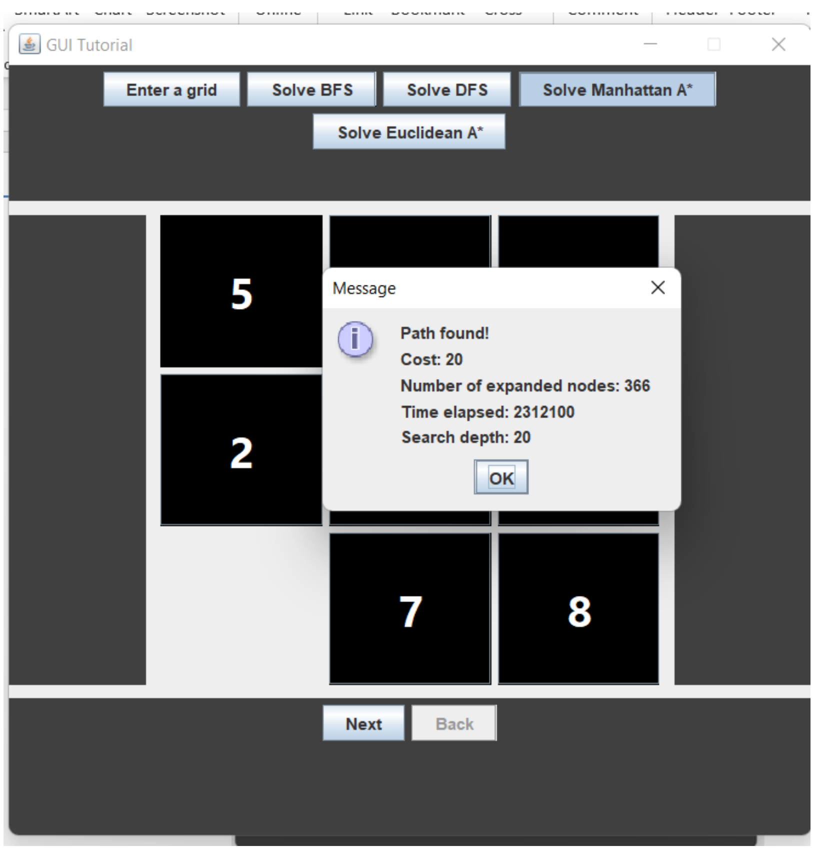
- Running BFS



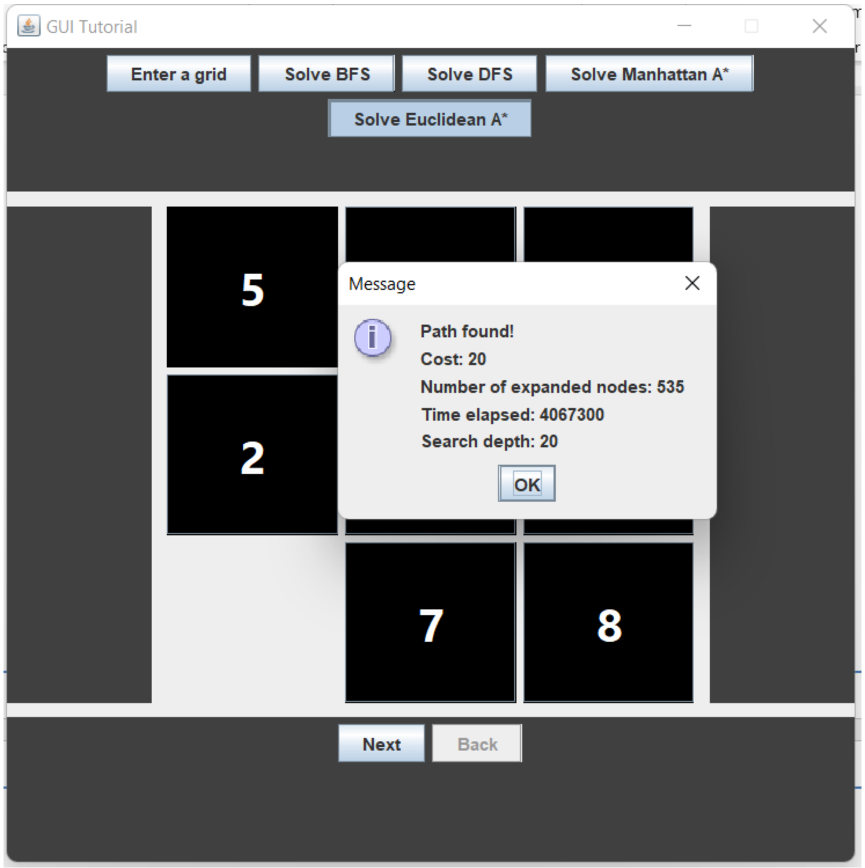
- Running DFS



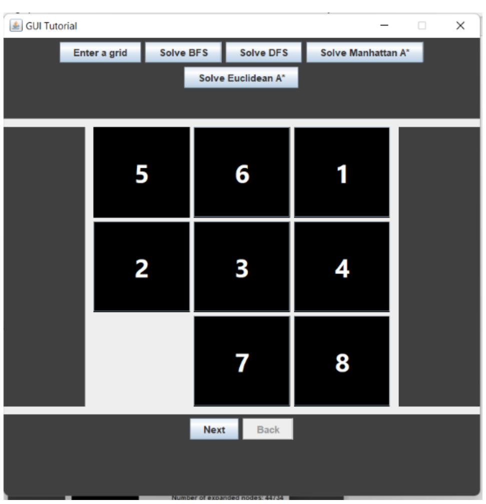
- Running A* with Manhattan distance heuristic

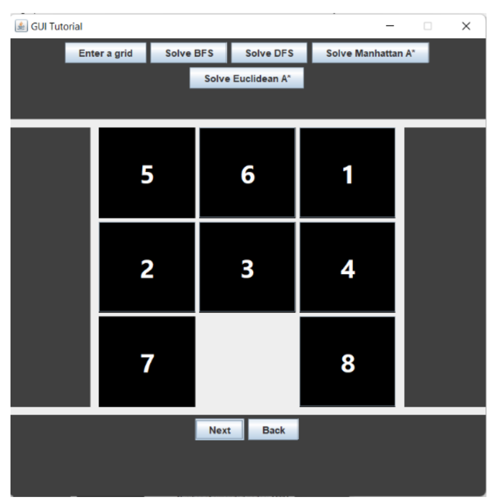


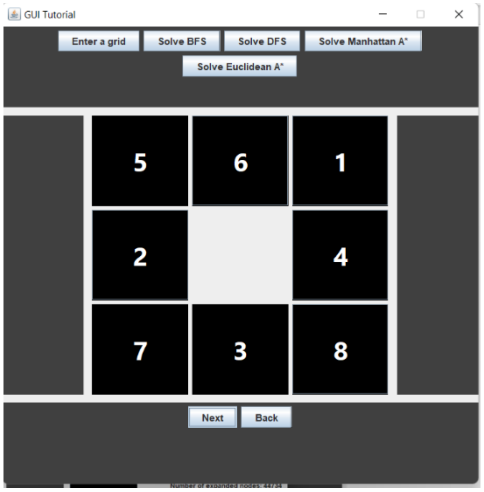
- Running A* with Euclidean distance heuristic

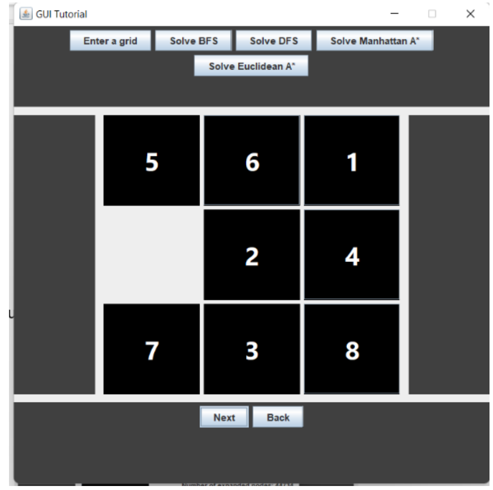


- Example path trace of A* algorithm

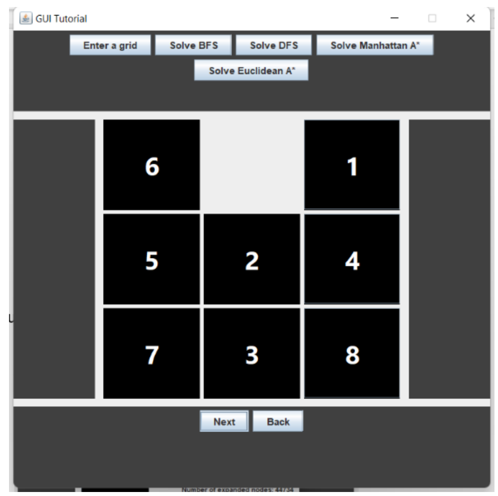




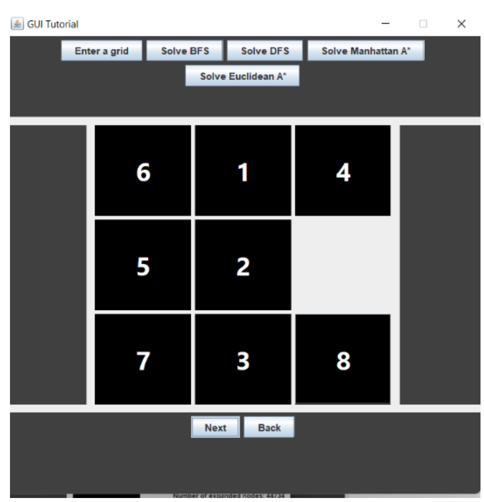

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
5 6 1
2 4
7 3 8
Next
Back

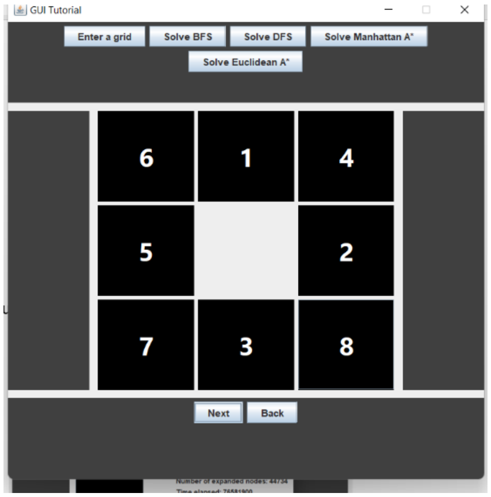

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
5 6 1
2 4
7 3 8
Next
Back

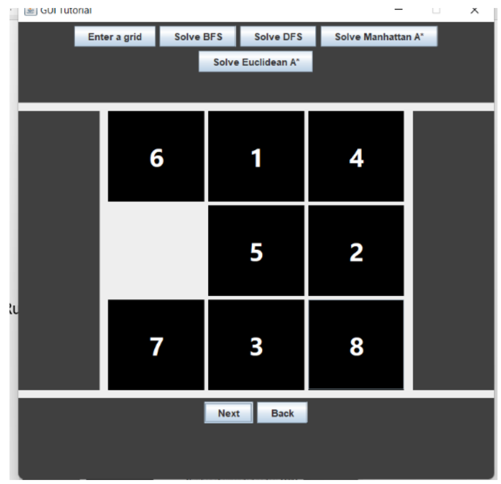

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
6 1
5 2 4
7 3 8
Next
Back

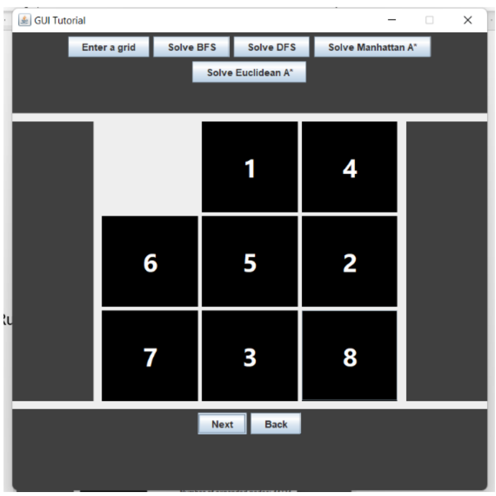

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
6 1
5 2 4
7 3 8
Next
Back

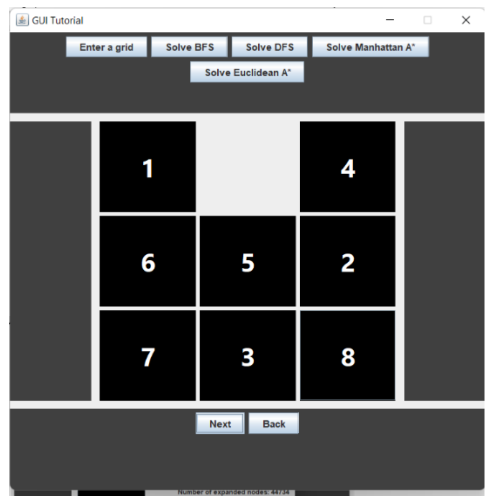

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
6 1
5 2 4
7 3 8
Next
Back

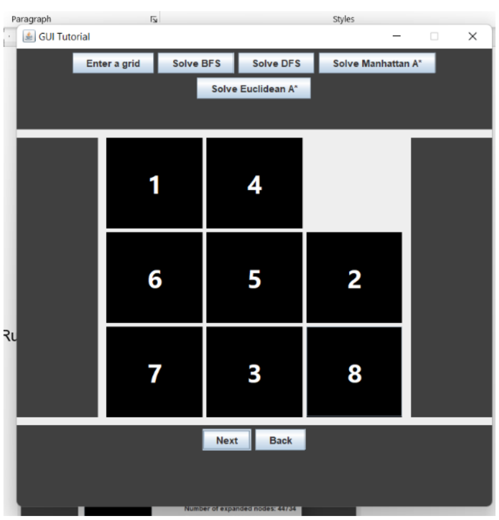

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
6 1 4
5 2
7 3 8
Next
Back

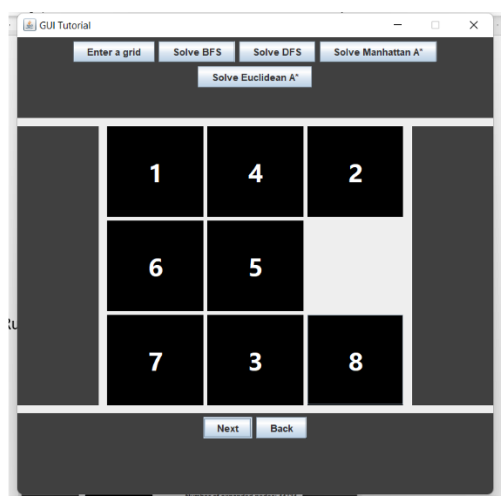

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
6 1 4
5 2
7 3 8
Next
Back

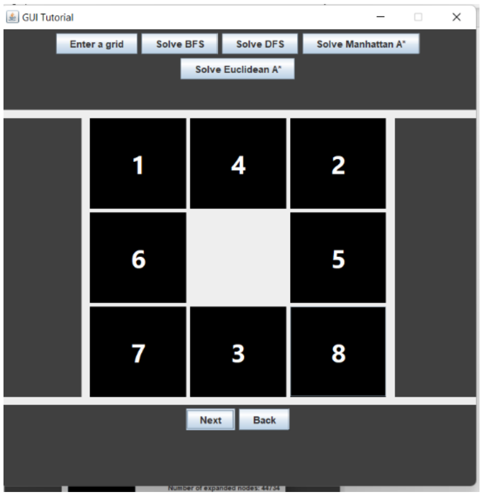

Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
6 1 4
5 2
7 3 8
Next
Back

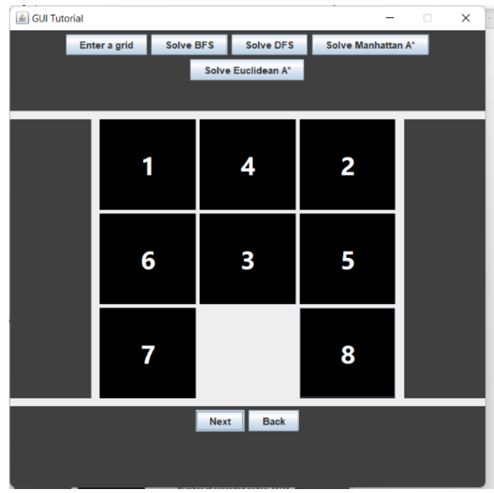

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1 4
6 5 2
7 3 8
Next
Back

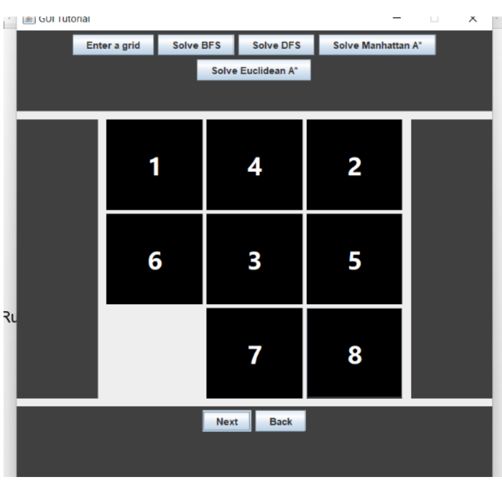

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1 4
6 5 2
7 3 8
Next
Back

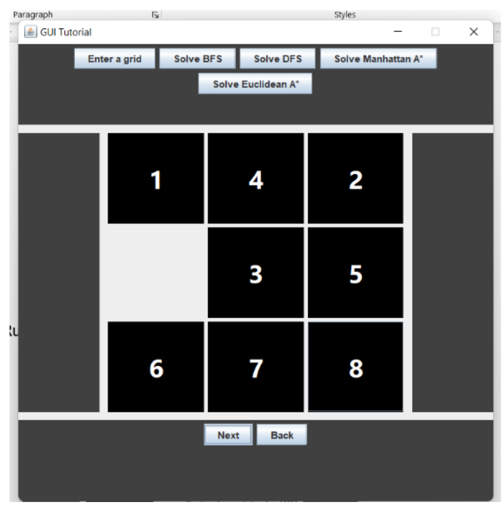

Paragraph
Styles
GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1 4
6 5 2
7 3 8
Next
Back

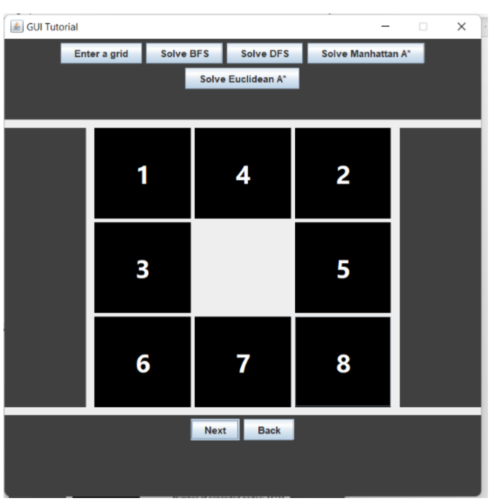

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1 4 2
6 5
7 3 8
Next
Back

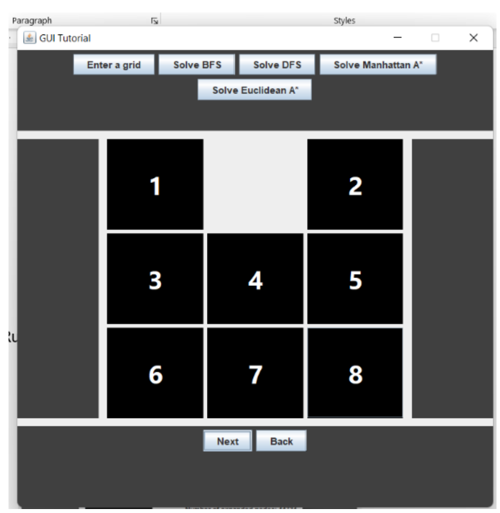

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
4
2
6
5
7
3
8
Next
Back

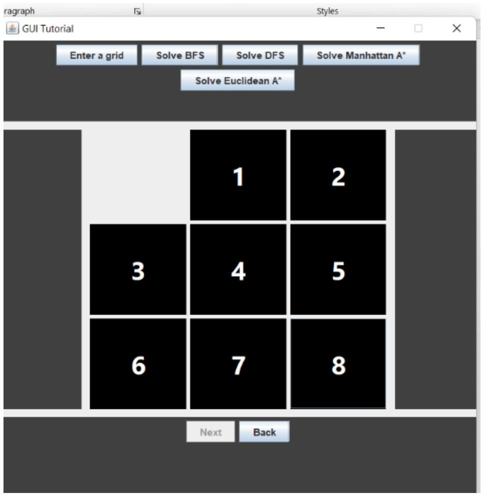

GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
4
2
6
3
5
7
8
Next
Back


GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
4
2
6
3
5
7
8
Next
Back


Paragraph
Styles
GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
4
2
3
5
6
7
8
Next
Back


GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
4
2
3
5
6
7
8
Next
Back


Paragraph
Styles
GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
2
3
4
5
6
7
8
Next
Back


GUI Tutorial
Enter a grid
Solve BFS
Solve DFS
Solve Manhattan A*
Solve Euclidean A*
1
2
3
4
5
6
7
8
Next
Back

Performance Analysis

We have tested each algorithm against the same 7 sample test cases to test their performance.

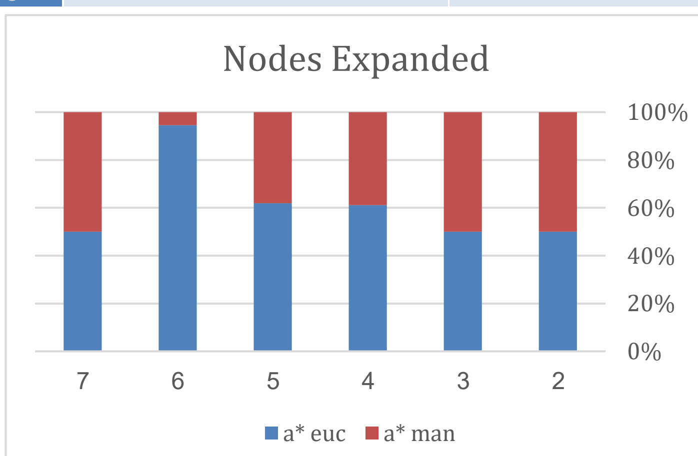
The results were as the following:

Case	Algorithm performance				
1	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
8 1 2 0 4 3 7 6 5	BFS	Infinity (no solution)	181440	692867000	31
	DFS	Infinity (no solution)	181440	644399600	66056
	A* Manhattan	Infinity (no solution)	181440	880826100	31
	A* Euclidean			910076300	31
2	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
1 2 5 3 4 0 6 7 8	BFS	3	18	339100	3
	DFS	3	181438	624903000	66125
	A* Manhattan	3	4	51300	3
	A* Euclidean	3	4	273100	3
3	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
0 1 2 3 4 5 6 7 8	BFS	0	1	70700	0
	DFS	0	1	6900	0
	A* Manhattan	0	1	20500	0
	A* Euclidean	0	1	17900	0
4	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
2 4 3 1 0 6 7 5 8	BFS	22	87746	330872900	22
	DFS	65974	95672	362662000	65974
	A* Manhattan	22	1685	7626000	22
	A* Euclidean	22	1070	6065500	22
5	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
1 5 8 0 2 3 4 6 7	BFS	19	30468	89270100	19
	DFS	3213	3300	12076700	3213
	A* Manhattan	19	456	1512900	19
	A* Euclidean	19	281	608800	19
6	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
8 7 6 5 4 3 2 1 0	BFS	28	178224	648790000	28
	DFS	64830	106784	469913400	66126
	A* Manhattan	28	5120	17917200	28
	A* Euclidean	28	297	826400	28

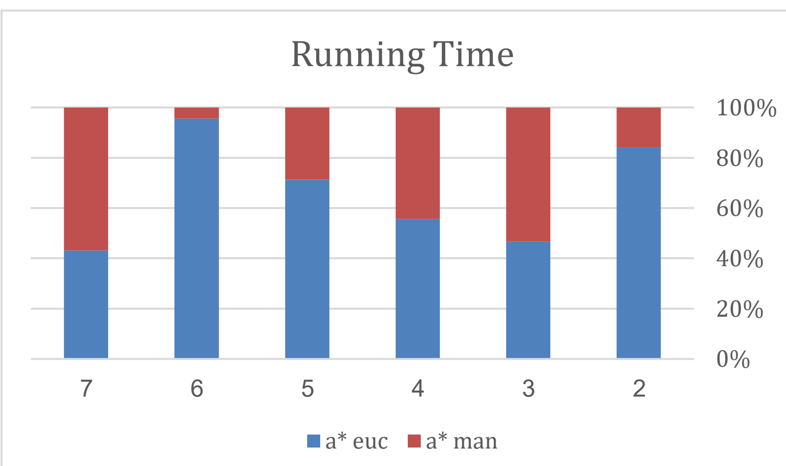
7	Algorithm	Cost	# Expanded nodes	Time (ns)	Search depth
1 2 0 3 4 5 6 7 8	BFS	2	4	8700	2
	DFS	294	181204	671679700	66488
	A* Manhattan	2	3	15000	2
	A* Euclidean	2	3	19800	2

A* Manhattan vs A* Euclidean , Cost of goal path comparison:

State	A* Euclidean	A* Manhattan
1 2 5 3 4 0 6 7 8	4	4
0 1 2 3 4 5 6 7 8	1	1
2 4 3 1 0 6 7 5 8	1685	1070
1 5 8 0 2 3 4 6 7	456	281
8 7 6 5 4 3 2 1 0	5120	297
1 2 0 3 4 5 6 7 8	3	3



Running Time	A* Euclidean	A* Manhattan
1 2 5 3 4 0 6 7 8	273100	51300
0 1 2 3 4 5 6 7 8	17900	20500
2 4 3 1 0 6 7 5 8	7626000	6065500
1 5 8 0 2 3 4 6 7	1512900	608800
8 7 6 5 4 3 2 1 0	17917200	826400
1 2 0 3 4 5 6 7 8	15000	19800



Conclusion:

Manhattan distance heuristic is better than Euclidean distance heuristic as both heuristics are admissible, Manhattan distance heuristic reaches the Goal state faster as its value is nearer to the perfect heuristic value than Euclidean distance heuristic.

Euclidean distance is not as a realistic heuristic as the Manhattan distance. Euclidean distance doesn’t consider any barriers in the puzzle and that each square can be pulled from its place and placed in its right goal place.

Manhattan distance considers the constraint of moving the squares in x-axis and y-axis values on at a time but doesn’t consider the possibility of having other squares in the path.