COMP 472 - Mini Project 1

Instructor: Sabine Bergler

2023-03-14

Contents

I certify that this submission is my original work and meets the Faculty's Expectations of Originality

1 8-Puzzle

1.1 Introduction

In assignment #1 we described the implementation of searches to solve a 8-puzzle.

We will first detail the representation and adaptation made to it compared to the first assignment.

Then we will describe the heuristics and searches created and their modifications.

Finally we will present the experiments we ran and the results from the different types of searches.

1.2 Representation

In the assignment we proposed to represent the 8-puzzle state as shown in figure 1

```
((A B C) (D E F) (G H 9)) (Row Col))
```

Listing 1: Initial proposition for representation

We first produced a prototype using oriented object constructs. This permitted laying out the concepts and algorithms. Then this representation reverted back to a list representation. This permitted the use of unit tests and easier debugging for most errors.

The final representation was also adapted to the different values required to produce the algorithms, as shown in 2

```
puzzle_repr = [
    tiles,
    goal,
    h_val,
    h_func,
    e_pos,
    parent,
    cost
]
```

Listing 2: Chosen representation

Each element of the list is as follows:

- tiles and goal: is a list of size 9 holding the tiles values
- h_{val}: stores the value produced by the heuristic after a move
- h_{func}: holds the heuristic function. This lets us define the heuristic on creation of the board and pass it to children
- e_{pos}: holds the position of the empty tile
- parent: is the index in the `closed` list of the parent that generated this state
- cost: numeric value holding the actual cost of the state

We also de-coupled the tile values from their actual position on the board. For this, we used a mapping storing the real position of the tile on the board based on their value, as seen in Listing 3

It should be noted that this mapping is used when the goal solution is of the form `(1 2 3 8 B 4 7 6 5)`. For the goal with the Blank at the end, the mapping would not be necessary or in our case can be replaced by `(1 2 3 4 5 6 7 8 9)`, as we did for the Challenge at the end.

```
mapping = [0,1,2,5,8,7,6,3,4]
```

Listing 3: Mapping between tile value and position

1.3 Search and Heuristics

The searches need to have a successor generation tool. We first created the functions to produce those as required. It uses a list containing the value to apply to move a tile and a function to ensure that the move is possible. From this list, the successors function can generate only possible moves of the empty tile, for example if it is at the top of the board, the state cannot have a successor that moves the tile up.

After testing the successors, we implement a generic search. It follows BFS and can receive two arguments:

- `popfunc`: this lets us decide if the search follows the DFS (stack) or BFS (queue) selection from the opened list
- `sortfunc`: this lets us decide the order of the successors in the opened list.

  The `opened` and `closed` list use the `deque` implementation as this lets us decide the pop and sorting actions.

  The BFS and DFS use this generic search and pass a `lambda` that uses respectively `popleft()` or `pop()`.

  The next search implemented is Best First Search. This reuses the BFS implementation and passes a custom `sortfunc` in the form of a lambda calling `subset.insort()`. This ensures that the opened list stays sorted, based on the heuristic selected. The heuristic is set directly on the initial state at creation.

  Finally, A* implementation reuses BFS, this time we set a function as the heuristic that calls the desired heuristic and adds the cost to it.

1.3.1 Heuristics

We implemented the following heuristics:

- Hamming: this calculates the number of missplaced tiles
- Manhattan: this calculates the minimum distance a tile has to move to reach its destination
- Inversion-Permutation: computes the number of tiles missplaced on the left of each tile
- More informed Manhattan: as defined in the first assignment, this adds the linear conflicts to the Manhattan heuristic.

  The heuristics are wrapped in a function that ensures the validity of the board - if the number of inversions is odd then the board is not solvable - and returns 0 if the board is in the goal state.

1.4 Experiment and results

The design was first tested to verify that it could find the solution.

We then produced different board settings to gradually test the performance of the different designs. The results shown in Table 1 shows the results. The values are an average of 10 runs, except for BFS which took too long to test many times. The second column shows the different initial states tested

The problem with a longer solution path was generated using a function that shuffles the numbers until it finds a goal state.

First, we observe the difference between the two basic searches, DFS and BFS. We see that for the complex input DFS was faster and visited less nodes to find a solution. We also see that BFS always found the minimum amount of moves, which is optimal.

Then we see that adding a heuristic, to create the Best First Search algorithm, dramatically improves the performance of the program. For the large input, using the linear conflict detection provides the best results in terms of speed and number of nodes visited, but doesn't provide the optimal solution. As expected, Manhattan visits less nodes than Hamming. Surprisingly, Hamming provides a slightly more optimal solution.

Finally, compared to the others, we see that A* provides speed and an optimal solution, as well as a reduced number of visited nodes, with the linear conflict implementation showing better performance.

For comparing Best First and A*, the number of visited nodes and duration do not vary by much on inputs that require a low number of moves to find the solution. The difference is more visible with problems that have a longer path to solution.

Table 1: Results of running 8-puzzle search for different inputs

Search	Input	Nodes visited	Duration	Solution Length
bfs	(2,4,7,1,5,9,6,3,8)	17913	104331.54	18
.	(1,2,3,8,9,4,7,6,5)	1	0.03	1
.	(1,9,3,8,2,4,7,6,5)	4	0.32	2
.	(1,2,3,7,8,4,9,6,5)	5	0.22	3
.	(2,8,3,1,6,4,7,9,5)	34	1.61	6
dfs	(2,4,7,1,5,9,6,3,8)	8544	38433.77	7660
.	(1,2,3,8,9,4,7,6,5)	1	0.03	1
.	(1,9,3,8,2,4,7,6,5)	2	0.14	2
.	(1,2,3,7,8,4,9,6,5)	233	34.68	215
.	(2,8,3,1,6,4,7,9,5)	18349	185273.65	16400
b1s manhattan	(2,4,7,1,5,9,6,3,8)	432	93.95	42
.	(1,2,3,8,9,4,7,6,5)	1	0.02	1
.	(1,9,3,8,2,4,7,6,5)	2	0.08	2
.	(1,2,3,7,8,4,9,6,5)	3	0.29	3
.	(2,8,3,1,6,4,7,9,5)	6	0.33	6
b1s hamming	(2,4,7,1,5,9,6,3,8)	957	402.70	32
.	(1,2,3,8,9,4,7,6,5)	1	0.01	1
.	(1,9,3,8,2,4,7,6,5)	2	0.10	2
.	(1,2,3,7,8,4,9,6,5)	3	0.25	3
.	(2,8,3,1,6,4,7,9,5)	6	0.27	6
Best 1st Search	(2,4,7,1,5,9,6,3,8)	209	32.28	48
Manhattan	(1,2,3,8,9,4,7,6,5)	1	0.01	1
with Linear	(1,9,3,8,2,4,7,6,5)	2	0.19	2
Conflict	(1,2,3,7,8,4,9,6,5)	3	0.15	3
.	(2,8,3,1,6,4,7,9,5)	6	0.45	6
Best First Search	(2,4,7,1,5,9,6,3,8)	276	42.64	32
with Permutation	(1,2,3,8,9,4,7,6,5)	1	0.02	1
Inversion	(1,9,3,8,2,4,7,6,5)	2	0.13	2
.	(1,2,3,7,8,4,9,6,5)	3	0.13	3
.	(2,8,3,1,6,4,7,9,5)	6	0.38	6

Table 2: Results of running 8-puzzle A* for different inputs

Search	Input	Nodes visited	Duration	Solution Length
A* manhattan	(2,4,7,1,5,9,6,3,8)	63	4.59	18
.	(1,2,3,8,9,4,7,6,5)	1	0.02	1
.	(1,9,3,8,2,4,7,6,5)	2	0.14	2
.	(1,2,3,7,8,4,9,6,5)	3	0.28	3
.	(2,8,3,1,6,4,7,9,5)	6	0.53	6
A* hamming	(2,4,7,1,5,9,6,3,8)	932	370.58	18
.	(1,2,3,8,9,4,7,6,5)	1	0.02	1
.	(1,9,3,8,2,4,7,6,5)	2	0.16	2
.	(1,2,3,7,8,4,9,6,5)	3	0.16	3
.	(2,8,3,1,6,4,7,9,5)	7	0.67	6
A* with	(2,4,7,1,5,9,6,3,8)	64	8.46	18
Manhattan	(1,2,3,8,9,4,7,6,5)	1	0.01	1
and Linear	(1,9,3,8,2,4,7,6,5)	2	0.22	2
Conflict	(1,2,3,7,8,4,9,6,5)	3	0.38	3
.	(2,8,3,1,6,4,7,9,5)	9	0.77	6
A* with	(2,4,7,1,5,9,6,3,8)	210	34.25	18
permutation	(1,2,3,8,9,4,7,6,5)	1	0.02	1
inversion	(1,9,3,8,2,4,7,6,5)	2	0.20	2
.	(1,2,3,7,8,4,9,6,5)	3	0.13	3
.	(2,8,3,1,6,4,7,9,5)	6	0.38	6

1.4.1 Challenge

We ran the challenge provided by Professor Bergler with our different searches and produced the results shown in Table 3.

We see that A* shows slightly better performance in memory but not in time. We had to remove the DFS run as it never completed

Table 3: 8-puzzle challenge run

Search	Nodes visited	Duration	Solution Length
bfs	311	59.16	9
dfs			
b1s manhattan	11	1.02	9
b1s hamming	132	15.03	19
b1s $\text{manhat}_{\text{collision}}$	21	2.19	9
b1s $\text{permutation}_{\text{inversion}}$	10	0.97	9
A* manhattan	11	0.85	9
A* hamming	21	1.57	9
A* $\text{manhat}_{\text{collision}}$	13	3.27	9
A* $\text{permutation}_{\text{inversion}}$	10	1.27	9

2 Numberlink

We first go over the representation of the puzzle, and its evolution from the first assignment. Then we discuss the heuristics implemented to solve it.

We then show an analyze the different experiments that we ran with the implementations.

Finally, we run the implementation on two challenges submitted by Professor Bergler and analyze the results.

2.1 Representation

Similar to the 8-Puzzle, we kept the same representation as discussed in the first assignment.

During implementation of the algorithms and helper functions we added attributes to the representation

The representation is a list with each index used as follows:

- 0: This holds the list representing the grid
- 1: This hold a list of lists, representing the links for each number in the grid. The lists hold None between the two number to link, unless the link completly connects the number, in which case `None` is removed.
- 2: This holds the dimensions of the grid. Those dimensions are used for various computations in the program, for example to verify valid moves.
- 3: This holds the value of the heuristic for the given state
- 4: This holds the heuristic functions. Having it defined on the state lets us reuse it for the successors and also lets us change the heuristic used in our tests.
- 5: The index of the parent that generated this state in the closed list
- 6: The actual cost of the current state

2.2 Heuristics

2.2.1 Admissible

As proposed in the first assignment, we used the following heuristic:

- compute the distance between each end of the links to their target

This generated the results shown in Tables 5, 6 and 7. We tested the solution with different input sizes and averaged ten runs.

2.2.2 Inadmissible

The implementation of the inadmissible heuristic proposed in the first assignment was not wrong enough to see a difference with the admissible heuristic. We changed it to the following:

- for each end of unconnected link, sum up the number of empty contiguous cells

2.3 Experiment and Results

As described earlier, we generated different input states, along the two given by Professor Bergler as a challenge, shown in Figure 4.

Table 4: The different inputs tested

't1'	't2'	't3'	't4'
3 · · 2	3 · · 2	1 2 · · 2	1 · · · 2
· 2 1 ·	· · · ·	· · 3 · 4	· 2 4 · 4
· · 3 1	· 2 1 ·	· · 4 · 5	· 3 · · 5
4 · · 4	· · 3 1	· · · 3 ·	· · · 3 ·
	4 · · 4	· · · 1 5	· · · 1 5

't5'	't6'	't7'
1 · · · ·	1 2 3 4	1 2 · ·
· 2 · 3 2	2 · · ·	· 1 3 ·
· · · · 4	3 · · ·	2 3 · ·
· · 4 · 5	4 · · ·	· · · ·
· · · 3 ·		
· · · 1 5		

Table 5 show the results of the experiment using the designed heuristics and a cost of one for each step. After seeing the difference in time and number of visited nodes between Best First and A*, we tested different heuristics and cost calculation.

Table 5: Cost = 1

Input	Algorithm	Nodes Visited	Time to solution	Solution length
t1	b1s admissible	9	1.57	9
t2	b1s admissible	33	6.81	13
t3	b1s admissible	17	9.72	16
t4	b1s admissible	44	17.51	16
t5	b1s admissible	51	20.26	21
t6	b1s admissible	10	1.13	10
t7	b1s admissible	18	1.59	11
t1	a* admissible	75	18.08	9
t2	a* admissible	996	461.08	13
t3	a* admissible	1374	842.89	16
t6	a* admissible	61	11.65	10
t7	a* admissible	63	7.82	11

Table 6 shows the results using a different cost calculation of 1 divided by the number of empty cells. The logic being that we count the cost as a proportional to completion. We now observe that it performs the same as Best First.

Table 6: Cost = 1/# empty cells

Input	Algorithm	Nodes Visited	Time to solution	Solution length
t1	b1s admissible	9	1.93	9
t2	b1s admissible	33	7.99	13
t3	b1s admissible	17	6.56	16
t4	b1s admissible	44	15.26	16
t5	b1s admissible	51	20.28	21
t6	b1s admissible	10	0.94	10
t7	b1s admissible	18	1.94	11
t1	a* admissible	9	1.65	9
t2	a* admissible	33	10.80	13
t3	a* admissible	17	13.38	16
t6	a* admissible	10	1.08	10
t7	a* admissible	18	1.69	11
t1	b1s inadmissible	9	1.42	9
t2	b1s inadmissible	13	2.22	13
t3	b1s inadmissible	554	222.73	16
t4	b1s inadmissible	35	8.06	16
t5	b1s inadmissible	53	12.37	21
t6	b1s inadmissible	14	1.64	10
t7	b1s inadmissible	11	1.68	11
t1	a* inadmissible	9	1.37	9
t2	a* inadmissible	13	2.20	13
t3	a* inadmissible	556	228.83	16
t6	a* inadmissible	14	1.55	10
t7	a* inadmissible	16	2.70	11

Finally, we experimented with a minimal cost. This idea was so that it would stay in proportion to the heuristic and permit consistency. We observe in Table 7 that it does not change from the previous cost computation.

Table 7: Results of search using Cost = 1/100

Input	Algorithm	Nodes Visited	Time to solution	Solution length
t1	bfs	353	60.76	9
t2	bfs	4389	2835.64	13
t6	bfs	191	24.01	10
t7	bfs	110	10.29	11
t1	b1s admissible	9	1.57	9
t2	b1s admissible	33	6.41	13
t3	b1s admissible	17	6.49	16
t4	b1s admissible	44	14.55	16
t5	b1s admissible	51	18.80	21
t6	b1s admissible	10	1.01	10
t7	b1s admissible	18	1.67	11
t1	a* admissible	9	1.52	9
t2	a* admissible	33	8.93	13
t3	a* admissible	17	6.78	16
t6	a* admissible	10	0.91	10
t7	a* admissible	18	1.61	11
t1	b1s inadmissible	9	1.30	9
t2	b1s inadmissible	13	2.78	13
t3	b1s inadmissible	554	209.55	16
t4	b1s inadmissible	35	5.11	16
t5	b1s inadmissible	53	15.24	21
t6	b1s inadmissible	14	1.45	10
t7	b1s inadmissible	11	0.79	11
t1	a* inadmissible	9	1.29	9
t2	a* inadmissible	13	2.77	13
t3	a* inadmissible	556	225.06	16
t6	a* inadmissible	14	1.50	10
t7	a* inadmissible	11	0.72	11

2.4 Analysis and Conclusion

From the experiment, we see that algorithms don't perform better for every type of input. We also see that Best First search seems to perform better than A* which is unexpected. We suspect that there are a few issues that would need to be addressed to perform better.

First, by using a cost of 1 for each step while the heuristics measures the distance left to complete links, the cost at the end is much higher than the sum of H+G of the other nodes, and therefore the algorithm will jump to other branches that are farther from the solution, back and forth until H+G brings it back to the solution.

Second, we see that when the cost is minimal, it performs similarly to Best First. Which is logical since it almost cancel it. This suggests that a better cost function could help produce more consistent results for A*

Lastly, contrarily to the 8-Puzzle problem, the numberlink puzzle has a maximum length each link can have and a fixed amount of tiles to fill. The sequence of the moves is not important since only the final state is necessary to have the complete solution of the puzzle, the rest is just intermediate step. Given these observations, it seems appropriate to not use a search that uses a cost since it does not give more information about the problem.

2.5 Challenge

Two challenges were provided by Professor Bergler

2.5.1 Challenge #1

Challenge #1 is shown as follows. To match with our representation we use `[1,2,3,4,2,0,0,0,3,0,0,0,4,0,0,`

1	2	3	4
2	·	·	·
3	·	·	·
4	·	·	·

We note that 1 cannot be connected to any other number. We consider it connected to itself or as a space that cannot be used. To do so, the list of links creates a closed link containing a start and end point with the same value.

Table 8: Challenge 1 solution

b1s admissible	a* admissible	b1s inadmissible	a* inadmissible
Solution found	Solution found	Solution found	Solution found
[1, 2, 3, 4]	[1, 2, 3, 4]	[1, 2, 3, 4]	[1, 2, 3, 4]
[2, 2, 3, 4]	[2, 2, 3, 4]	[2, 2, 3, 4]	[2, 2, 3, 4]
[3, 3, 3, 4]	[3, 3, 3, 4]	[3, 3, 3, 4]	[3, 3, 3, 4]
[4, 4, 4, 4]	[4, 4, 4, 4]	[4, 4, 4, 4]	[4, 4, 4, 4]
closed: 10	closed: 61	closed: 14	closed: 79
solution path length: 10	solution path length: 10	solution path length: 10	solution path length: 10
Time to solve: 1.26	Time to solve: 13.46ms	Time to solve: 4.41ms	Time to solve: 17.32ms

As shown in the previous section, A* performs worse than Best first, in time and memory. We also observe that as expected the inadmissible heuristic does not perform as well.

2.5.2 Challenge #2

Challenge #2 is shown as follows. To match with our representation we use `[1,2,0,0,0,1,3,0,2,3,0,0,0,0,0,`

1	2	·	·
·	1	3	·
2	3	·	·
·	·	·	·

Running it with Best First search and A*, using the admissible and the inadmissible heuristics produces the following results.

Table 9: Challenge 2 solution

b1s admissible	a* admissible	b1s inadmissible	a* inadmissible
Solution found	Solution found	Solution found	Solution found
[1, 2, 2, 2]	[1, 2, 2, 2]	[1, 2, 2, 2]	[1, 2, 2, 2]
[1, 1, 3, 2]	[1, 1, 3, 2]	[1, 1, 3, 2]	[1, 1, 3, 2]
[2, 3, 3, 2]	[2, 3, 3, 2]	[2, 3, 3, 2]	[2, 3, 3, 2]
[2, 2, 2, 2]	[2, 2, 2, 2]	[2, 2, 2, 2]	[2, 2, 2, 2]
closed: 18	closed: 63	closed: 11	closed: 61
solution path length: 11	solution path length: 11	solution path length: 11	solution path length: 11
Time to solve: 2.75	Time to solve: 12.93	Time to solve: 1.36	Time to solve: 11.59

This shows similar results, except that A* with the inadmissible heuristic gives better performance in time and memory than A* with admissible heuristic. This shows as suggested in the previous section that either the heuristic or the cost function are not properly matched, or that Numberlink is not an appropriate problem for A*.