COMP2611

Artificial Intelligence

Assignment 1: Search Algorithms

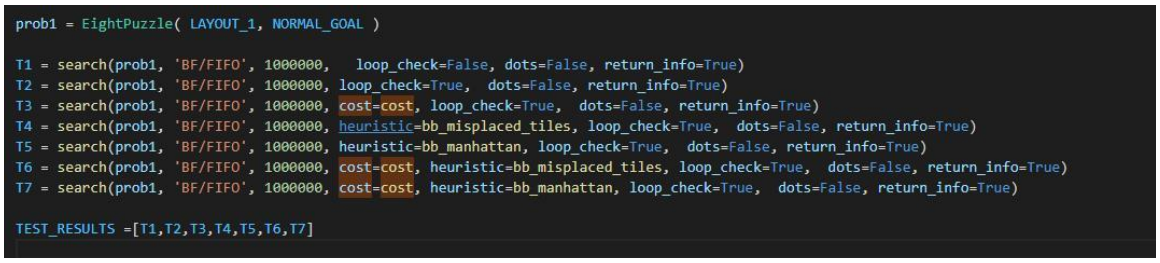
Student A: Leshan Wang 201388927

Student B: Xinyu Li 201388943

A. 8-Puzzle Search Investigation

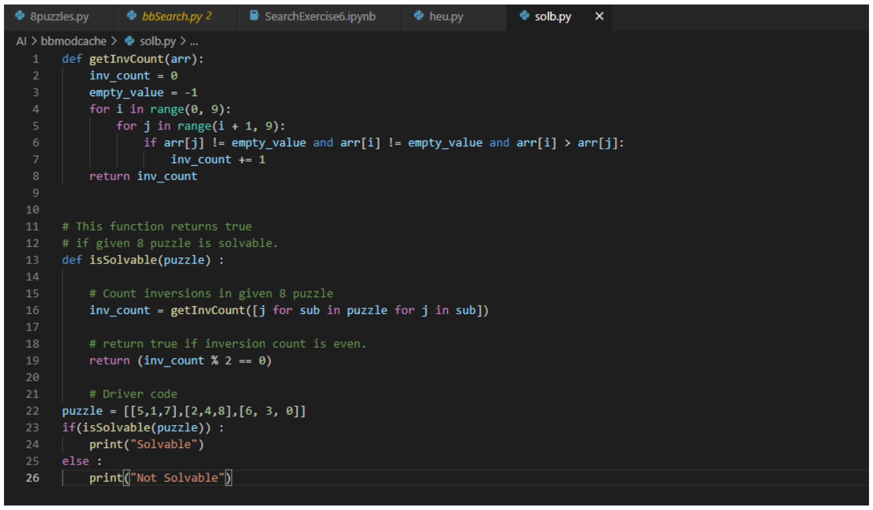
1. Search Algorithm Test Sequence

We have done a complete investigation of the codes and tested multiple scenarios to find out the possible key options/features, which will dramatically change the performance and result of the search sequence. It had been confirmed that Loop_check, Cost and Heuristic functions are the key elements, therefore we designed a sequence to test and show the difference of distinct configurations of search algorithm.



```
prob1 = EightPuzzle( LAYOUT_1, NORMAL_GOAL )

T1 = search(prob1, 'BF/FIFO', 1000000,   loop_check=False, dots=False, return_info=True)
T2 = search(prob1, 'BF/FIFO', 1000000, loop_check=True,  dots=False, return_info=True)
T3 = search(prob1, 'BF/FIFO', 1000000, cost=cost, loop_check=True,  dots=False, return_info=True)
T4 = search(prob1, 'BF/FIFO', 1000000, heuristic=bb_misplaced_tiles, loop_check=True,  dots=False, return_info=True)
T5 = search(prob1, 'BF/FIFO', 1000000, heuristic=bb_manhattan, loop_check=True,  dots=False, return_info=True)
T6 = search(prob1, 'BF/FIFO', 1000000, cost=cost, heuristic=bb_misplaced_tiles, loop_check=True,  dots=False, return_info=True)
T7 = search(prob1, 'BF/FIFO', 1000000, cost=cost, heuristic=bb_manhattan, loop_check=True,  dots=False, return_info=True)

TEST_RESULTS =[T1,T2,T3,T4,T5,T6,T7]
```

Shown by the screenshot, we made a set of tests using the same layout [5,1,7,2,4,8,6,3,0] and goal [1,2,3,4,5,6,7,8,0].



```python
def getInvCount(arr):
    inv_count = 0
    empty_value = -1
    for i in range(0, 9):
        for j in range(i + 1, 9):
            if arr[j] != empty_value and arr[i] != empty_value and arr[i] > arr[j]:
                inv_count += 1
    return inv_count


# This function returns true
# if given 8 puzzle is solvable.
def isSolvable(puzzle) :

    # Count inversions in given 8 puzzle
    inv_count = getInvCount([j for sub in puzzle for j in sub])

    # return true if inversion count is even.
    return (inv_count % 2 == 0)

    # Driver code
puzzle = [[5,1,7],[2,4,8],[6, 3, 0]]
if(isSolvable(puzzle)) :
    print("Solvable")
else :
    print("Not Solvable")
```

A pair of tiles form an inversion if the values on tiles are in reverse order of their appearance in goal state, as the parity of inversions remains same after a set of moves, in the goal state, there are 0 inversions. So, we can reach goal state only from a state which has even inversion count. Therefore, using the codes above, we checked whether the puzzle can be solved or not.

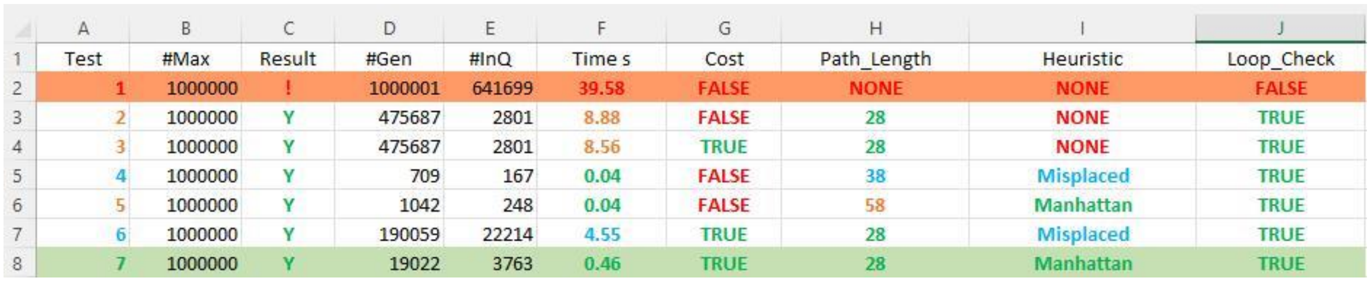
The max nodes are all set to be 1000000, because we don't want the limitation of nodes to be the reason for search failure. T1 is the naked test, which only implement the BF/FIFO algorithm with no loop check, nor cost or heuristic function. T2 is a comparison to T1 with loop check on, this is designed to show the difference loop check could made. Similarly, T3 has cost function implemented comparing to T2.

T4 and T5 are set to rival the performance of two heuristic functions, Misplaced and Manhattan. T6 and T7 implemented both cost function and heuristic function, which made them the A* algorithm. T6 and T7 will be compared, also they can be compared to T4 and T5 to see the difference between A* and basic heuristic algorithms. The test sequence showed interesting results which will be discussed in detail in following sections.

2. Results

In order to make the results highly readable and understandable, we posted the results into Excel and implemented different color schemes to highlight key points.

Here is the brief explanation of the colors and their meanings. Red indicates the worst result or not implemented, orange indicates not good, blue indicates good, green indicates best or implemented.



	A	B	C	D	E	F	G	H	I	J
1	Test	#Max	Result	#Gen	#InQ	Time s	Cost	Path_Length	Heuristic	Loop_Check
2	1	1000000	!	1000001	641699	39.58	FALSE	NONE	NONE	FALSE
3	2	1000000	Y	475687	2801	8.88	FALSE	28	NONE	TRUE
4	3	1000000	Y	475687	2801	8.56	TRUE	28	NONE	TRUE
5	4	1000000	Y	709	167	0.04	FALSE	38	Misplaced	TRUE
6	5	1000000	Y	1042	248	0.04	FALSE	58	Manhattan	TRUE
7	6	1000000	Y	190059	22214	4.55	TRUE	28	Misplaced	TRUE
8	7	1000000	Y	19022	3763	0.46	TRUE	28	Manhattan	TRUE

The table contains information of (from left to right): test number, max nodes, result (fail represented by **!**, found represented by **Y**), nodes generated, nodes left in queue, time consumed(important), cost function, path length(important), heuristic function(important), loop check(important).

Tests with more red or orange entries will be considered as not being well performed, while tests with more green or blue entries will be seen as better solutions. The best and worse result are emphasized with green and red background colors respectively.

3. Observations

After examining the results, we discovered several key aspects which had significant impact on the search algorithm.

The loop check function is definitely a game changer. Without loop check, even by given the max nodes of 1000000, the algorithm still cannot find the path towards the goal. Therefore test 1 is the worst result we've got. On the other hand, with loop check enabled, the remain 6 tests each was able to find an answer.

For this test sequence, we decided to determine the efficiency of the algorithm by checking its time and path length, if the time is below 1 second then it is considered efficient, if it has the shortest path length at the same time, then it will be considered as the best approach. Thus, test 7 which consumes 0.46 seconds and has the 28-path length is the best result of this test sequence.

Comparing test 2 with test 3, the cost function doesn't make a big difference on its own, this is because in the 8 puzzles, every move costs the same; therefore, the effect of cost function is the same as running a breadth-first search.

Comparing test 4 with test 5, it shows that by consuming the same amount of time, the Misplaced tile heuristic function has better result than Manhattan distance heuristic function. Generally, Manhattan distance heuristic function should perform better than Misplaced tile heuristic function, however, in certain conditions, well-constructed Misplaced tile heuristic function can outperform the other one.

Comparing test 6 with test 4 and test 7 with test 5, it indicates that with both cost function and heuristic function, which is the A* algorithm, it gets better result as it gives priority to states with the lowest sum of both cost and heuristic.

Finally, as it was mentioned, test 7 using both cost and Manhattan distance heuristic function has the best result. It proves that Manhattan distance heuristic function is very efficient and suitable for the 8 puzzles.

4. Heuristics

A. Misplaced Tile Heuristic Function

```
        def misplaced(state):

  mp = 0
  for n in range (1,9):
      x,y = number_position_in_layout(n,state[1])
      if (x,y) != NORMAL_GOAL_POSITIONS[n]:
          mp= mp + 1
  return mp
```

This function calculates the total number of misplaced tiles by comparing each tiles position to the goal state.

B. Manhattan Distance Heuristic Function

```
def manhattan(state):
  md = 0
  for n in range(1,9):
      x,y = number_position_in_layout(n,state[1])
      gx, gy = NORMAL_GOAL_POSITIONS[n]
      md += ( abs(x-gx) + abs(y-gy))
  return md
```

This function calculates the sum of the distance between each tile and their goal position.

B. Robot Worker Scenario

1. My/Our Robot Scenario

There is a robot that can help pick up items. The robot can grab any item as long as their weight is less than the robot's maximum grab weight of 15. All of the item information and their location are shown below.

Workshop:

Rusty key 0

Store room:

Bucket 2

Suitcase 4

Tool cupboard:

Sledge hammer 5

Anvil 12

Saw 2

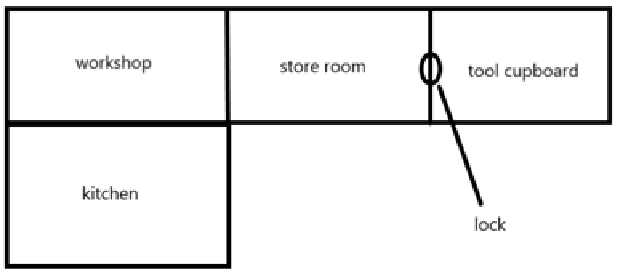
Screwdriver 1

Kitchen:

Apple 1

Chocolate 0

Knife 1



Robot can pick up and put down items and move to other rooms with the items that they picked up. And when the robot goes from the store room to the tool cupboard, it needs to pay attention to the state of the door lock at this time. If the door is locked, it needs to carry the rusty key at the same time; if the door is not locked, it can enter directly. The goal of the robot is to put the sledge hammer, screwdriver and anvil away in the store room.

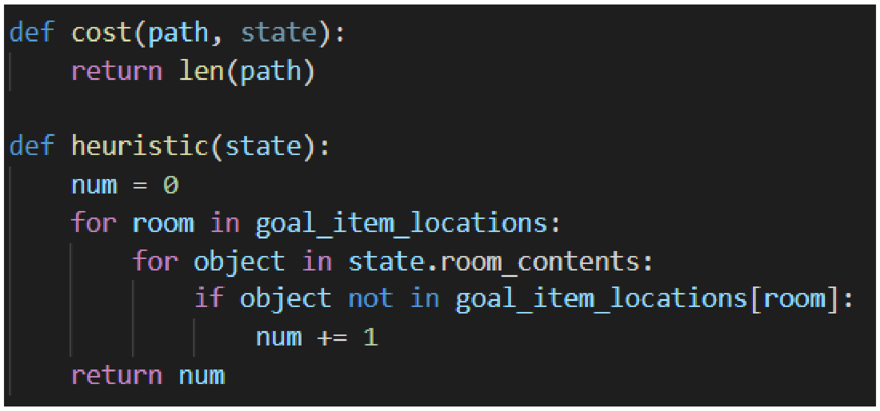
Test:

```
Path length = 10

Goal state is:

Robot location: store room

Robot carrying: []

Room contents: {'workshop': {'rusty key'}, 'store room': {'sledge hammer', 'anvil', 'bucket', 'screwdriver', 'suitcase'}, 'tool cupboard': {'saw'},

'kitchen': {'apple', 'knife', 'chocolate'}}

The action path to the solution is:

   ('move to', 'tool cupboard')

   ('pick up', 'sledge hammer')

   ('pick up', 'screwdriver')

   ('move to', 'store room')

   ('put down', 'sledge hammer')

   ('put down', 'screwdriver')

   ('move to', 'tool cupboard')

   ('pick up', 'anvil')

   ('move to', 'store room')

   ('put down', 'anvil')

SEARCH SPACE STATS:
```

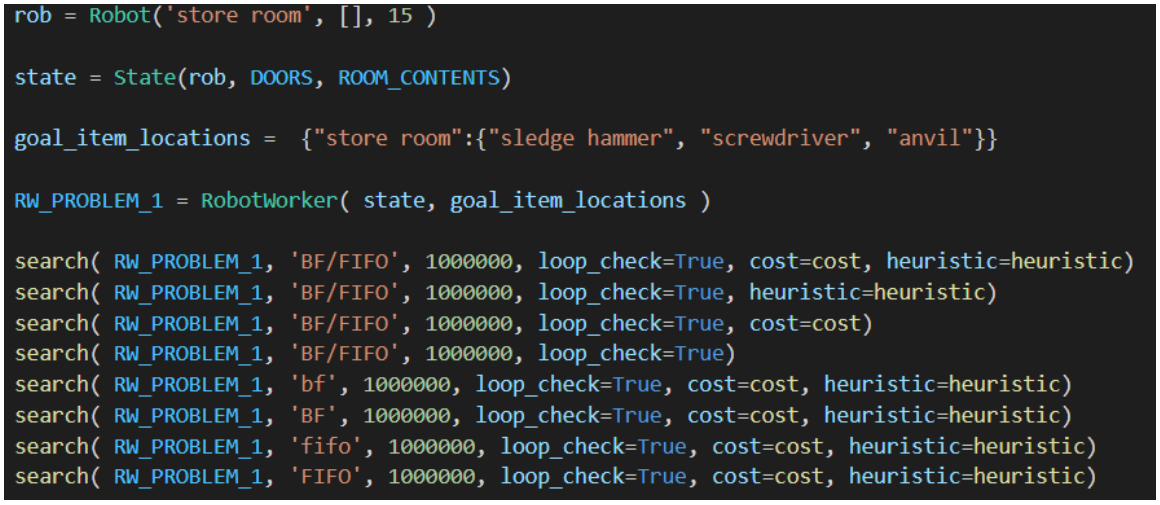
```
Total nodes generated          =   382430  (includes start)

Nodes discarded by loop_check  =   211711  (170719 distinct states added to queue)

Nodes tested (by goal_test)    =    63636  (63635 expanded + 1 goal)

Nodes left in queue            =   107083


Time taken = 36.7507 seconds
```

2. Heuristic(s)

I use the A-star search algorithm as a heuristic. Take the path the robot has moved as the cost, namely c(n), and take the cost to the destination state as the heuristic, namely h(n). This gives the smallest heuristic f(n) = c(n) + h(n). The cost, assuming that each operation consumes one unit of effort, returns the length of the path array. For an "admissible" heuristic, the value should always be equal to or less than the actual cost.

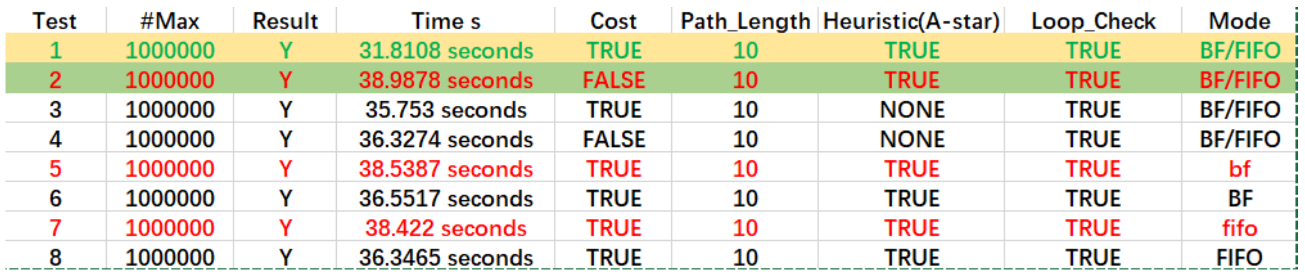


```python
def cost(path, state):
    return len(path)

def heuristic(state):
    num = 0
    for room in goal_item_locations:
        for object in state.room_contents:
            if object not in goal_item_locations[room]:
                num += 1
    return num
```



```python
rob = Robot('store room', [], 15 )

state = State(rob, DOORS, ROOM_CONTENTS)

goal_item_locations =  {"store room":{"sledge hammer", "screwdriver", "anvil"}}

RW_PROBLEM_1 = RobotWorker( state, goal_item_locations )

search( RW_PROBLEM_1, 'BF/FIFO', 1000000, loop_check=True, cost=cost, heuristic=heuristic)
search( RW_PROBLEM_1, 'BF/FIFO', 1000000, loop_check=True, heuristic=heuristic)
search( RW_PROBLEM_1, 'BF/FIFO', 1000000, loop_check=True, cost=cost)
search( RW_PROBLEM_1, 'BF/FIFO', 1000000, loop_check=True)
search( RW_PROBLEM_1, 'bf', 1000000, loop_check=True, cost=cost, heuristic=heuristic)
search( RW_PROBLEM_1, 'BF', 1000000, loop_check=True, cost=cost, heuristic=heuristic)
search( RW_PROBLEM_1, 'fifo', 1000000, loop_check=True, cost=cost, heuristic=heuristic)
search( RW_PROBLEM_1, 'FIFO', 1000000, loop_check=True, cost=cost, heuristic=heuristic)
```

3. Results



Test	#Max	Result	Time s	Cost	Path_Length	Heuristic(A-star)	Loop_Check	Mode
1	1000000	Y	31.8108 seconds	TRUE	10	TRUE	TRUE	BF/FIFO
2	1000000	Y	38.9878 seconds	FALSE	10	TRUE	TRUE	BF/FIFO
3	1000000	Y	35.753 seconds	TRUE	10	NONE	TRUE	BF/FIFO
4	1000000	Y	36.3274 seconds	FALSE	10	NONE	TRUE	BF/FIFO
5	1000000	Y	38.5387 seconds	TRUE	10	TRUE	TRUE	bf
6	1000000	Y	36.5517 seconds	TRUE	10	TRUE	TRUE	BF
7	1000000	Y	38.422 seconds	TRUE	10	TRUE	TRUE	fifo
8	1000000	Y	36.3465 seconds	TRUE	10	TRUE	TRUE	FIFO

After adding search criteria to the search method of the question, its time is reduced to a certain extent. I used A-star as a heuristic method to search, and obtained test data by controlling the difference in cost, Heuristic and search mode (mode). Among them, when the final path length is the shortest case 10, the time used by the A* algorithm is the shortest, that is, the A* algorithm shortens the search time and ensures the accuracy of the results.

Specifically, by comparing the first four sets of tests, we found that when the search method is BF/FIFO, when the A-star algorithm is used as a heuristic, the correct result will be obtained in the shortest time.

Test 1 and test 2 show that the use of the Cost function will lead to a relatively large difference in search time. In the case of not using the Cost function, the search time will be significantly increased, resulting in a wider range of searches.

Test 1 and test 3 show that whether to use the Heuristic function will also cause a difference in the search time. If the Heuristic function is not used, the search time will be longer. The same situation will appear in the comparison of test 1 and test 4.

It can be seen from the comparison of test 1 and test 5, 6, 7, and 8 that the search time is the shortest only when the search mode is BF/FIFO. In other cases, whether bf, BF, fifo, FIFO search way, it will take more time.

4.Key Findings

1. A general observation is that no matter which search method is used, in the face of relatively simple problems, the robot can always find the shortest path, but the search time varies depending on the search method.

2. When faced with the same problem, using A-star as a heuristic search method can often get the correct result in the shortest time. At the same time, when encountering complex problems, the time is also changed instead of correct rate.

3. In the case of using breadth search, bf and fifo take more time, while BF and FIFO tend to take less time under the same conditions.

4. In general, I found that using the A-star method as a heuristic method will search for the correct answer in the shortest time, whether it is a breadth search or a deep search.