MANİSA CELAL BAYAR UNIVERSITY

CSE 3213

ARTIFICIAL INTELLIGENCE

FALL 2020

HOMEWORK: Search Algorithms

PROJECT SUBJECT: Pancake Problem

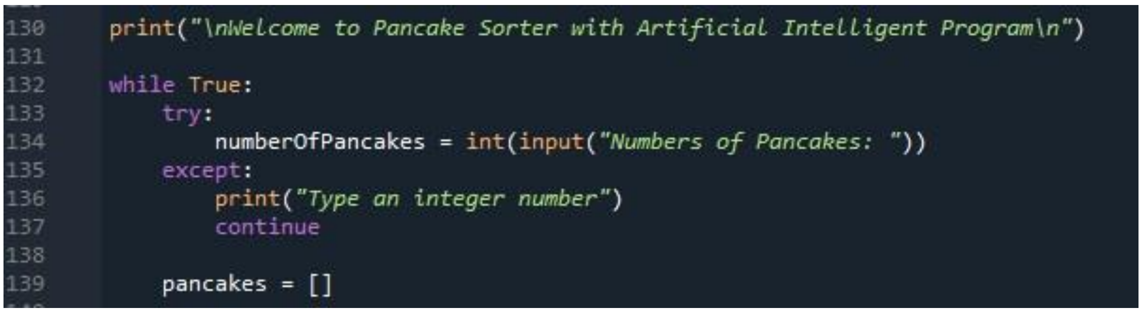
170315022 – EKİN ŞUATAMAN

13 December 2020

A – Problem Formulation

1) Initial State

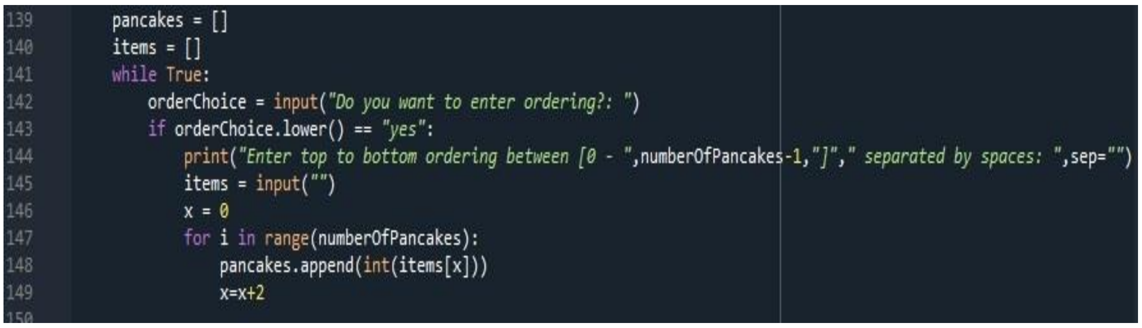
We used the try-except structure in figure 1.0 to prevent the user from entering a value other than integer. If the user enters a value other than integer, we make it possible to enter an integer number by printing 'Type an integer number' on the screen. Then we created an empty list to keep the pancake values.



```
print("\nWelcome to Pancake Sorter with Artificial Intelligent Program\n")

while True:
    try:
        numberOfPancakes = int(input("Numbers of Pancakes: "))
    except:
        print("Type an integer number")
        continue

    pancakes = []
```

Figure 1.0

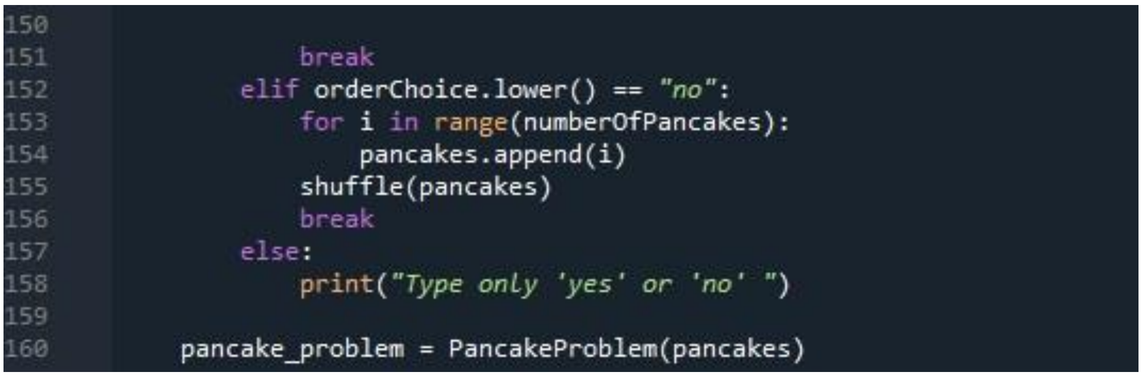
If we want to manually enter the size of the pancakes: The input string list is received as (items []) and must be in the form of 'number' 'space' 'number' 'space' 'number' 'space' (e.g. 1 0 2). Then the string values in the indexes corresponding to the numbers are converted to integer type and loaded into the pancake [] list.



```
    pancakes = []
    items = []
    while True:
        orderChoice = input("Do you want to enter ordering?: ")
        if orderChoice.lower() == "yes":
            print("Enter top to bottom ordering between [0 - ",numberOfPancakes-1,"]"," separated by spaces: ",sep="")
            items = input("")
            x = 0
            for i in range(numberOfPancakes):
                pancakes.append(int(items[x]))
                x=x+2
```

Figure 1.1

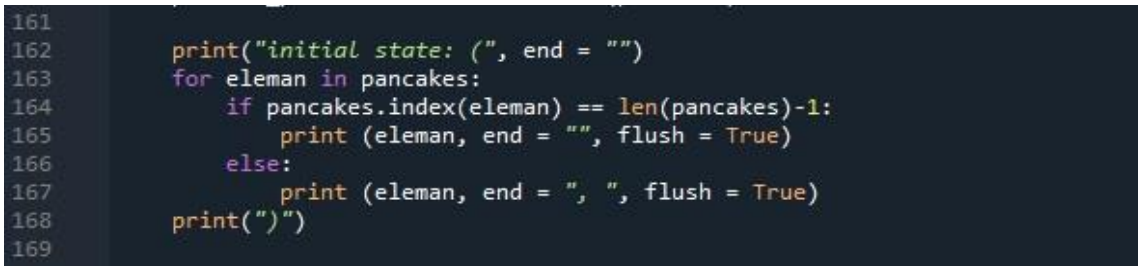
If the user chose to determine initial order of pancakes randomly ordered by computer. From 1 to number of pancakes, size of the pancakes are added in order to list. We had which elements are 1 to number of pancakes. Then, with shuffle function from random modul, we mixed the list.



```python
            break
        elif orderChoice.lower() == "no":
            for i in range(numberOfPancakes):
                pancakes.append(i)
            shuffle(pancakes)
            break
        else:
            print("Type only 'yes' or 'no' ")

    pancake_problem = PancakeProblem(pancakes)
```

Figure 1.2

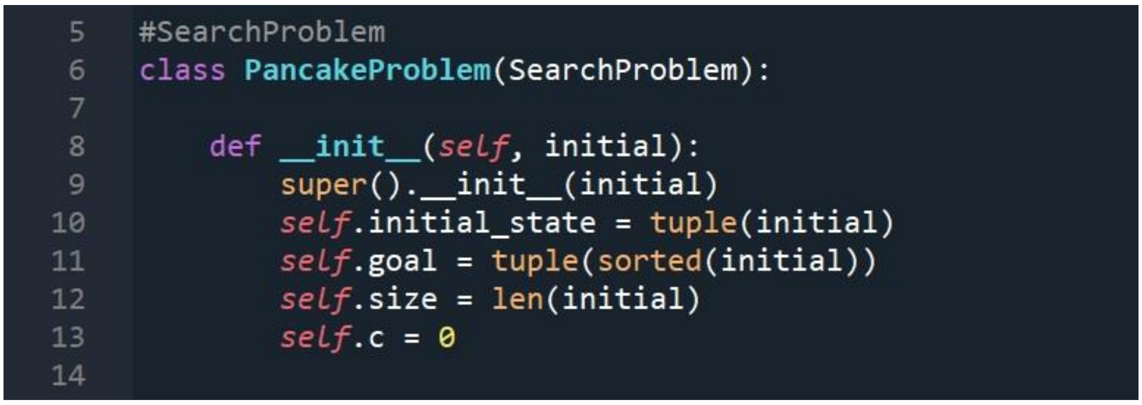
We typed fluesh = true because we want the change to be applied immediately. If we used 'end', there would be problems. It says False as 'Default', we changed it to True. Just like the change we made to the 'sep' variable in "print".



```python

    print("initial state: (", end = "")
    for eleman in pancakes:
        if pancakes.index(eleman) == len(pancakes)-1:
            print (eleman, end = "", flush = True)
        else:
            print (eleman, end = ", ", flush = True)
    print(")")

```

Figure 1.3

While creating problem, we have defined the initial state, goal, size and c variable. We used variable C in cost funksion. We have always converted the lists here into tuples. Because, This is what the "simpleai.search" library wants. Otherwise (especially lists) the program says “unhashable type: 'list'”.

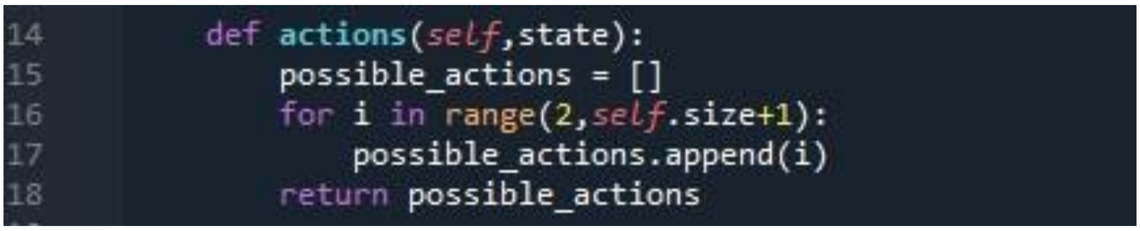


```python
#SearchProblem
class PancakeProblem(SearchProblem):

    def __init__(self, initial):
        super().__init__(initial)
        self.initial_state = tuple(initial)
        self.goal = tuple(sorted(initial))
        self.size = len(initial)
        self.c = 0

```

Figure 1.4

2) Possible Actions

We have created an empty list called 'possible_actions' to determine where to turn our pancakes. Then we added from the number 2 to the number of pancakes.The reason we started from the number 2 of the list is that turning it 1 time will not change the order of the pancakes.

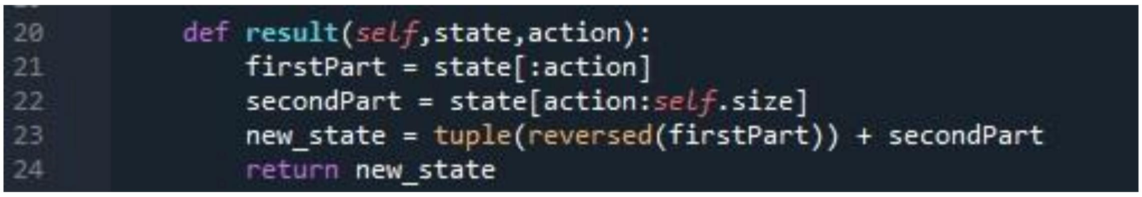


```
    def actions(self,state):
        possible_actions = []
        for i in range(2,self.size+1):
            possible_actions.append(i)
        return possible_actions
```

Figure 1.5

3) Transition Model

We copied the first part of state to firstPart and kept the rest of state at secondPart. After that we reversed the first part and combined it with the second part to create a new state.

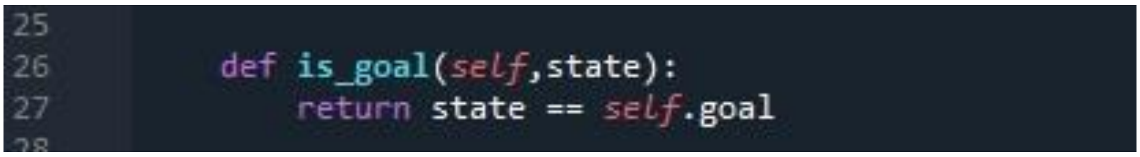


```
    def result(self,state,action):
        firstPart = state[:action]
        secondPart = state[action:self.size]
        new_state = tuple(reversed(firstPart)) + secondPart
        return new_state
```

Figure 1.6

4) Goal Test

We created a "goal_test" function, in this function we check whether the situation is a goal or not. In this function, if the state equals the target state, it returns true. However, if it's not equal, it returns false.

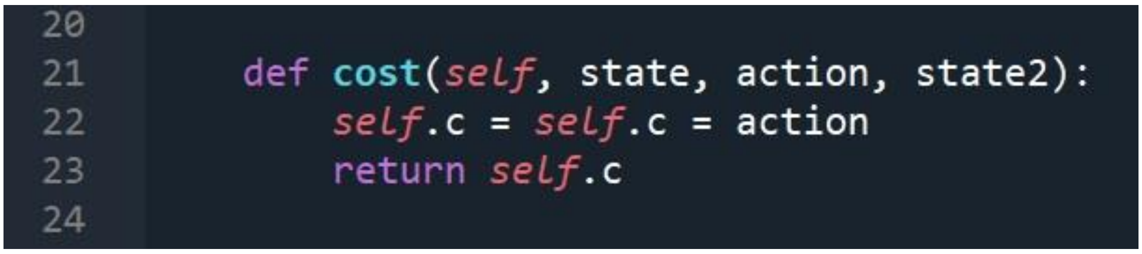


```
    def is_goal(self,state):
        return state == self.goal
```

Figure 1.7

5) Path Cost

We defined an object while calculating the path cost (The object is self.c on def __init__(self,initial)). Then, we collected the actions taken.

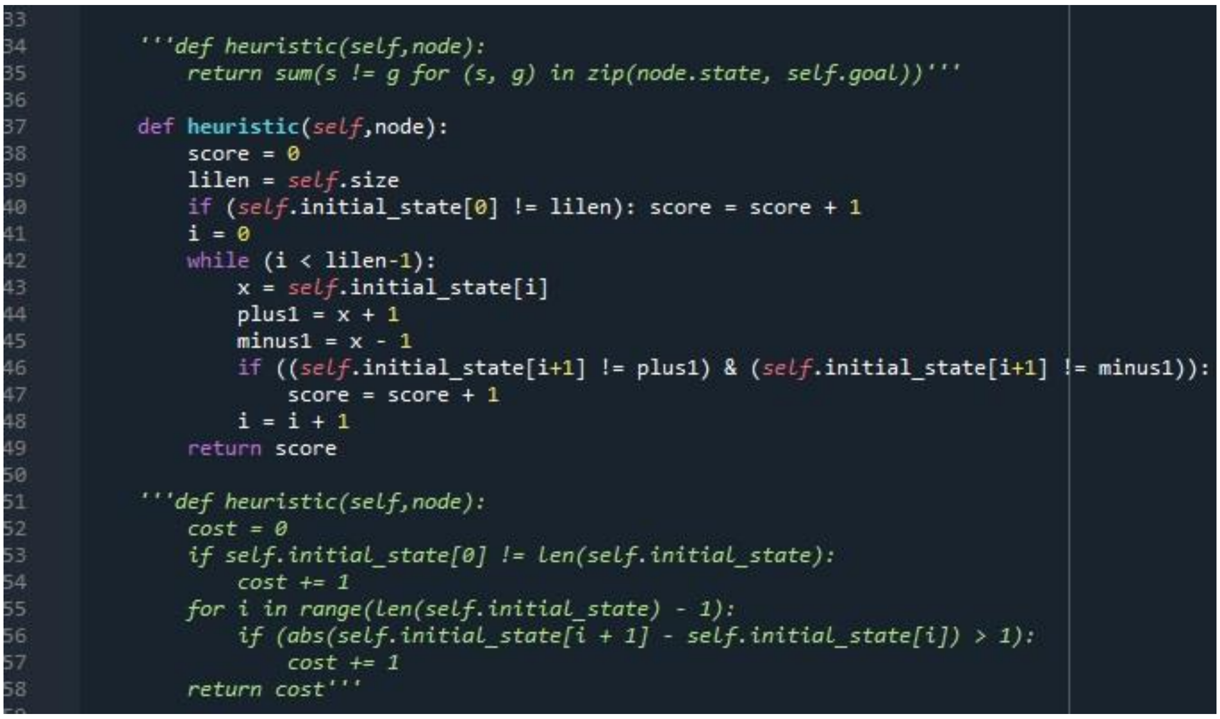


```python
    def cost(self, state, action, state2):
        self.c = self.c = action
        return self.c
```

Figure 1.8

B – Heuristic Functions

Heuristic is a function that calculates the estimated number of turns required to get the pancakes to the desired position. The heuristic function we have chosen among 3 heuristic functions works in the most useful way. We calculate the cost value according to the number of gaps in the pancake. The number of gaps is 1 if a pancake is not on the correct goal and the large pancake is not at the bottom of the stack.

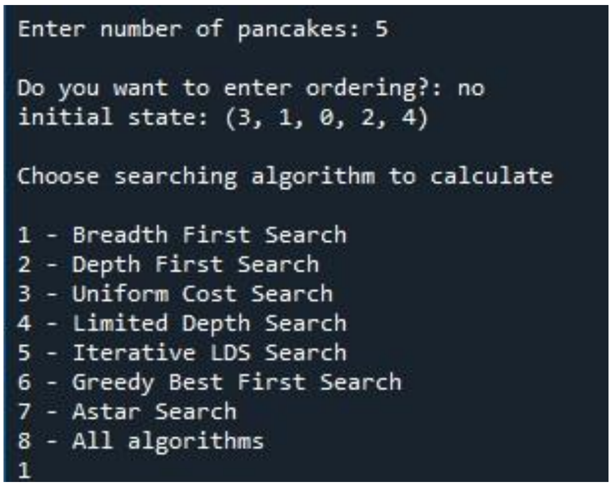


```python
    '''def heuristic(self,node):
        return sum(s != g for (s, g) in zip(node.state, self.goal))'''

    def heuristic(self,node):
        score = 0
        lilen = self.size
        if (self.initial_state[0] != lilen): score = score + 1
        i = 0
        while (i < lilen-1):
            x = self.initial_state[i]
            plus1 = x + 1
            minus1 = x - 1
            if ((self.initial_state[i+1] != plus1) & (self.initial_state[i+1] != minus1)):
                score = score + 1
            i = i + 1
        return score

    '''def heuristic(self,node):
        cost = 0
        if self.initial_state[0] != len(self.initial_state):
            cost += 1
        for i in range(len(self.initial_state) - 1):
            if (abs(self.initial_state[i + 1] - self.initial_state[i]) > 1):
                cost += 1
        return cost'''
```

Figure 1.9

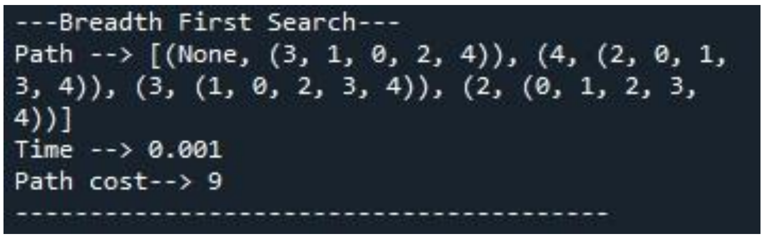
C – Discussion on the Result

1) Number of Pancakes = 5



```
Enter number of pancakes: 5

Do you want to enter ordering?: no
initial state: (3, 1, 0, 2, 4)

Choose searching algorithm to calculate

1 - Breadth First Search
2 - Depth First Search
3 - Uniform Cost Search
4 - Limited Depth Search
5 - Iterative LDS Search
6 - Greedy Best First Search
7 - Astar Search
8 - All algorithms
1
```

Figure 2.0



```
---Breadth First Search---
Path --> [(None, (3, 1, 0, 2, 4)), (4, (2, 0, 1,
3, 4)), (3, (1, 0, 2, 3, 4)), (2, (0, 1, 2, 3,
4))]
Time --> 0.001
Path cost--> 9
----------------------------------------
```

Figure 2.1

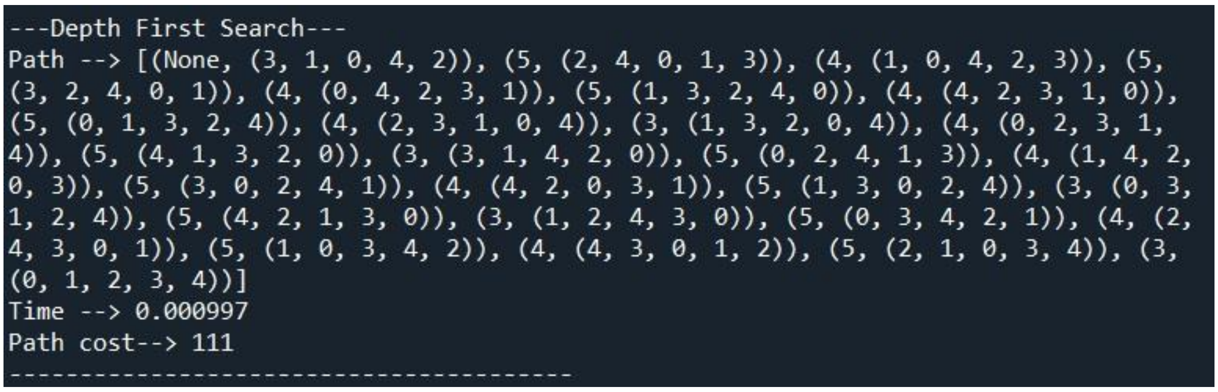
BFS is complete and time complexity of BFS is 0.001 seconds.

Completeness: Yes because in pancake problem b is finite.

Optimality: No because cost isn't 1 per step.

Time complexity: BFS Algorithm's time complexity is **O(b^d)** for that reason time complexity is bad because it is exponential.

Space complexity of BFS is **O(b^d)** for that reason space complexity is bad because it is exponential.



```
---Depth First Search---
Path --> [(None, (3, 1, 0, 4, 2)), (5, (2, 4, 0, 1, 3)), (4, (1, 0, 4, 2, 3)), (5,
(3, 2, 4, 0, 1)), (4, (0, 4, 2, 3, 1)), (5, (1, 3, 2, 4, 0)), (4, (4, 2, 3, 1, 0)),
(5, (0, 1, 3, 2, 4)), (4, (2, 3, 1, 0, 4)), (3, (1, 3, 2, 0, 4)), (4, (0, 2, 3, 1,
4)), (5, (4, 1, 3, 2, 0)), (3, (3, 1, 4, 2, 0)), (5, (0, 2, 4, 1, 3)), (4, (1, 4, 2,
0, 3)), (5, (3, 0, 2, 4, 1)), (4, (4, 2, 0, 3, 1)), (5, (1, 3, 0, 2, 4)), (3, (0, 3,
1, 2, 4)), (5, (4, 2, 1, 3, 0)), (3, (1, 2, 4, 3, 0)), (5, (0, 3, 4, 2, 1)), (4, (2,
4, 3, 0, 1)), (5, (1, 0, 3, 4, 2)), (4, (4, 3, 0, 1, 2)), (5, (2, 1, 0, 3, 4)), (3,
(0, 1, 2, 3, 4))]
Time --> 0.000997
Path cost--> 111
----------------------------------------
```

Figure 2.2

DFS is complete and time complexity of DFS is 0.00097.

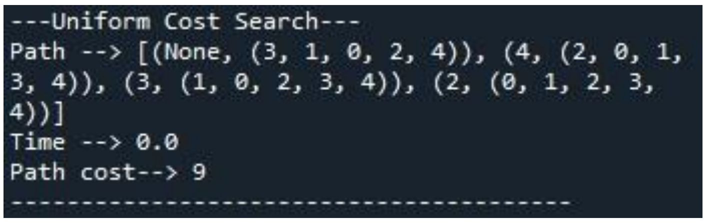
Completeness: For this pancake problem there is a finite space so the result is reached but if depth space were going indefinitely the result would not be found.

Optimality: It is not optimal as it cannot find the lowest cost while there is more than one result.

Time complexity: m is a finite expression fot that reason time complexity is **O(b^m)**.

Space complexity: For tree search version space complexity of DFS is **O(bm)** also in DFS space complexity is linear.

→ In this place, we used the tree structure. Because in the other case, we usually couldn't get a solution.



```
---Uniform Cost Search---
Path --> [(None, (3, 1, 0, 2, 4)), (4, (2, 0, 1,
3, 4)), (3, (1, 0, 2, 3, 4)), (2, (0, 1, 2, 3,
4))]
Time --> 0.0
Path cost--> 9
----------------------------------------
```

Figure 2.3

UCS is complete and time complexity of UCS is 0.0 seconds.

We realized that UCS search algorithm is the most efficient search algorithm among all search algorithms.

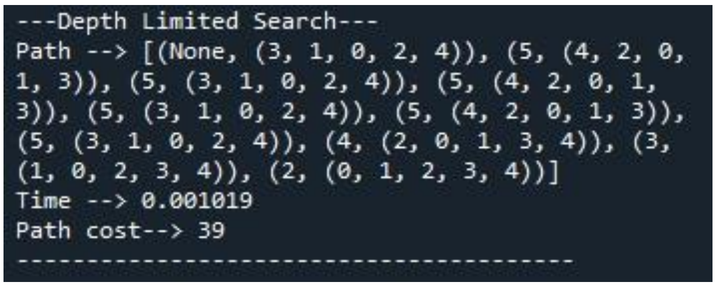
Completeness: Yes because step cost is positive.

Optimality: Yes. Nodes expanded in increasing order of path cost.

Time complexity: Time complexity of UCS is **$O(b^{c^*/e})$**. (exponential effective depth)

Space complexity of UCS is **$O(b^{c^*/e})$**.

Time complexity and space complexity is not something we want to be exponential.



```
---Depth Limited Search---
Path --> [(None, (3, 1, 0, 2, 4)), (5, (4, 2, 0,
1, 3)), (5, (3, 1, 0, 2, 4)), (5, (4, 2, 0, 1,
3)), (5, (3, 1, 0, 2, 4)), (5, (4, 2, 0, 1, 3)),
(5, (3, 1, 0, 2, 4)), (4, (2, 0, 1, 3, 4)), (3,
(1, 0, 2, 3, 4)), (2, (0, 1, 2, 3, 4))]
Time --> 0.001019
Path cost--> 39
----------------------------------------
```

Figure 2.4

DLS is complete and time complexity of DLS is 0.001019 seconds.

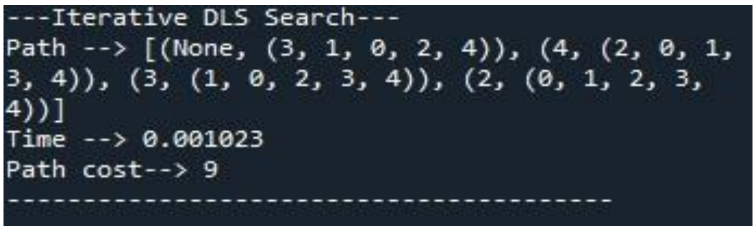
Completeness: No. l < d for that reason incomplete.

Optimality: No. Because l > d for that reason nonoptimal.

Time complexity : Time complexity is **$O(b^L)$** and much better than DFS because l is smaller than m.

Space complexity: Space complexity of DLS is **O(bl)** because there is b children were kept at each level but there were l levels in total.

→We gave “depth_limit” as 10 and it solved the problems we usually run. If we don't give enough limits, the Python says -'NoneType' object has no attribute 'path'-.



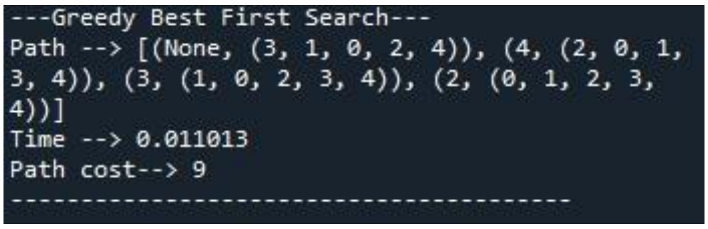
```
---Iterative DLS Search---
Path --> [(None, (3, 1, 0, 2, 4)), (4, (2, 0, 1,
3, 4)), (3, (1, 0, 2, 3, 4)), (2, (0, 1, 2, 3,
4))]
Time --> 0.001023
Path cost--> 9
----------------------------------------
```

Figure 2.5

ILDS is complete and time complexity of IDS is 0.001023 seconds.

It has become a more efficient search algorithm by taking BFS to be complete and optimal and space complexity to be linear from DFS.

Space complexity of IDS is **O(bd)**.



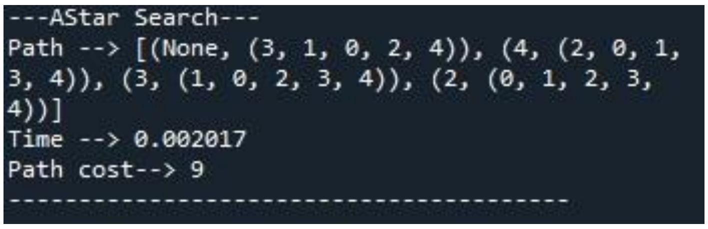
```
---Greedy Best First Search---
Path --> [(None, (3, 1, 0, 2, 4)), (4, (2, 0, 1,
3, 4)), (3, (1, 0, 2, 3, 4)), (2, (0, 1, 2, 3,
4))]
Time --> 0.011013
Path cost--> 9
----------------------------------------
```

Figure 2.6

GBFS is complete and time complexity of GBFS is 0.011013 seconds.

GBFS is a informed search algorithm. For that reason GBFS can find solutions more efficiently than can an uninformed algorithm.

Space complexity of GBFS is **O(b^d)**.



```
---AStar Search---
Path --> [(None, (3, 1, 0, 2, 4)), (4, (2, 0, 1,
3, 4)), (3, (1, 0, 2, 3, 4)), (2, (0, 1, 2, 3,
4))]
Time --> 0.002017
Path cost--> 9
----------------------------------------
```

Figure 2.7

A* is complete and time complexity of A* is 0.002017 seconds.

Astar is a informed search algorithm. For that reason Astar can find solutions more efficiently than can an uninformed algorithm.

Space complexity of Astar **O(b^d)**.

Notes

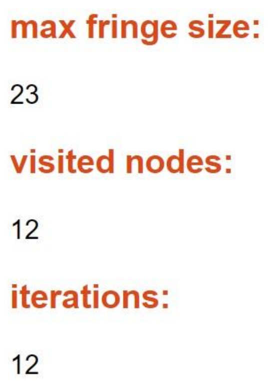
→When Python runs the search algorithms, the amount of memory it uses can find 96%. When this happens, it can strain the computer. Based on this, having a good computer is a must to solve big problems. It creates difficulties not only during decoding but also when dumping data.

→The computers we have become unstable in such difficult situations. Our intention here is, that the python console responded too late.

→ According to us, the most difficult part in doing this homework was the problem definition.

→We used the Spyder 4.1.4 version. (Python 3.8.3 64-bit | Qt 5.9.7 | PyQt 5.9.2 | Windows 10)

Speed Comparisons with the Viewer



max fringe size:

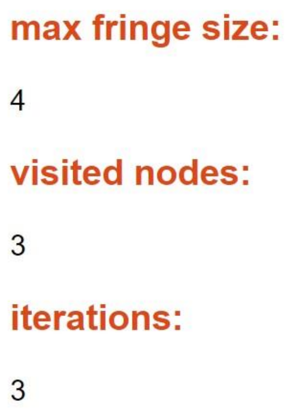
23

visited nodes:

12

iterations:

12



max fringe size:

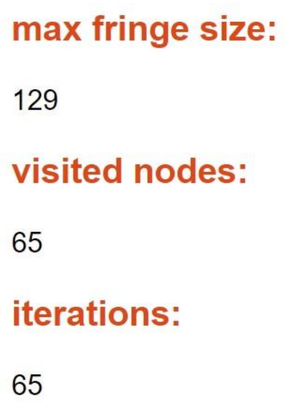
4

visited nodes:

3

iterations:

3



max fringe size:

129

visited nodes:

65

iterations:

65

Figure 2.8: Breadth || Figure 2.9: Depth use tree || Figure 3.0: Depth ∞

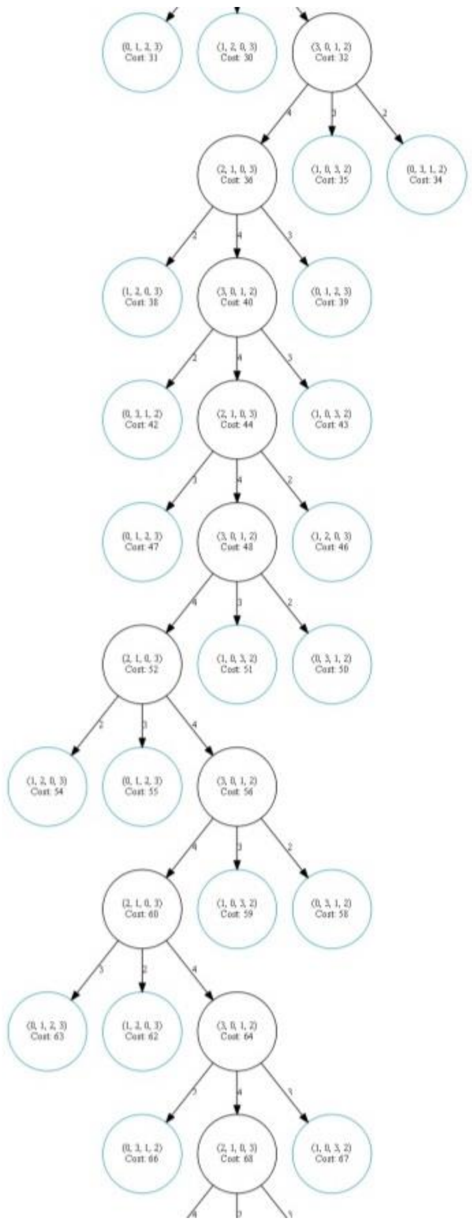


Figure 3.1: Depth Search in infinite loop/continuous loop “∞”

max fringe size:

29

visited nodes:

15

iterations:

15

max fringe size:

21

visited nodes:

12

iterations:

12

max fringe size:

5

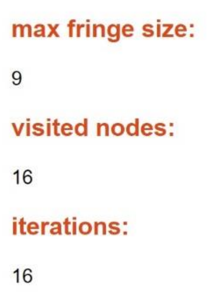
visited nodes:

9

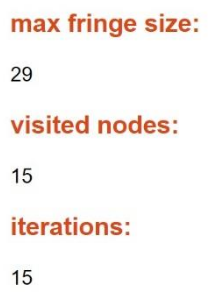
iterations:

9

Figure 3.2: Uniform Cost || Figure 3.3: Limited Depth Search || Figure 3.4: Iterative LDS



```
max fringe size:
9
visited nodes:
16
iterations:
16
```



```
max fringe size:
29
visited nodes:
15
iterations:
15
```

Figure 3.5: Greedy || Figure 3.6: A*

We entered 4 for initialize size. The initialize state: (3 0 1 2). After that we ran all of them one by one. And finally, at least part visit was done at Iterative limited depth search. Also, we used a tree for the Depth limit to finish. In this case there is a little less visiting process.

*The Iterative limited depth search gives the best performance in this example.

**Enters infinite loop at depth limit if the tree is not used.

References

https://pypi.org/project/simpleai/

https://simpleai.readthedocs.io/en/latest/

- ➔ **Stuart Russell and Peter Norvig;**
  Artificial Intelligence: A Modern Approach, Pearson, 2017