**MANİSA CELAL BAYAR UNIVERSITY**

**CSE 3213**

**ARTIFICIAL INTELLIGENCE**

**FALL 2019**

**COURSE PROJECT: Pancake Problem**

*160315016 – CAN GÜR*

*170315023 - ELİF FIRTANA*

*12 December 2019*

Table of Contents

[A Problem formulation 1](#_Toc527015162)

[A.1 Initial State 2](#_Toc527015165)

[A.2 Possible Actions 2](#_Toc527015168)

[A.3 Transition Model 2](#_Toc527015166)

[A.4 Goal Test 2](#_Toc527015167)

[A.5 Path Cost 2](#_Toc527015168)

[B Heuristic functions 3](#_Toc527015163)

[C Discussion on the results 4](#_Toc527015164)

[C.1 Number of Pancake 6 4](#_Toc527015165)

[C.2 Number of Pancake 7 7](#_Toc527015168)

[D References 10](#_Toc527015164)

**A – Problem Formulation**

**A.1 Initial State:**

Initial state of pancakes can be determined with 2 ways.

We asked to user if he/she wants to determine initial order of pancakes manually or randomly ordered by computer.

orderChoice = input("Want to determine initial order of {} pancakes [y] \\ [n]\n".format(numberOfPancakes))

Creating an empty list to store pancakes.

pancakes = []

We take number of pancakes as an integer. If the user enters else then integer, ‘Type an integer’ is printed on the screen and asked again number of pancakes.

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

If the user chose to determine initial order of pancakes manually. From zeroth index to last index of the stack, size of the pancakes are added in order to list.

if orderChoice.lower() == "y":  
 for i in range(numberOfPancakes):  
 items = int(input("Pancake size: "))  
 pancakes.append(items)  
 break

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 a list which elements are 1 to number of pancakes. Then, with shuffle function from random modul, we mixed the list.

elif orderChoice.lower() == "n":  
 for i in range(1, numberOfPancakes + 1):  
 pancakes.append(i)  
 shuffle(pancakes)  
 break

**A.2 Possible Actions:**

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

In pancake problem, actions are points in the stack. It determines where stack will flipped. We created an empty list named ‘possible\_actions’. We added numbers from 2 to size of stack. Did not added 1 because if we flip first pancake, the order of the pancakes will not be changed.

**A.3 Transition Model:**

To create a new state, we split state according to action. Reversed firstPart and join with secondPart.

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

**A.4 Goal Test:**

We determined the goal in the init function. Basically the goal state is sorted initial state.

self.goal = tuple(sorted(initial))

To check if state is the goal or not, we created a ‘goal\_test’ function. In this function if state is equals to goal state, it returns true. However, if it is not equals, it returns false.

def goal\_test(self,state):

return state == self.goal

**A.5 Path Cost:**

We determined every flip’s cost is 1. In every flip, path cost increases 1.

def path\_cost(self, c, state1, action, state2):

return c+1

**B – Heuristic Functions**

We determined heuristic of the states are number of misplaced pancakes. We compare state and goal and add 1 every time that nodes are not equals.

def h(self,node):

return sum(s != g for (s, g) in zip(node.state, self.goal))

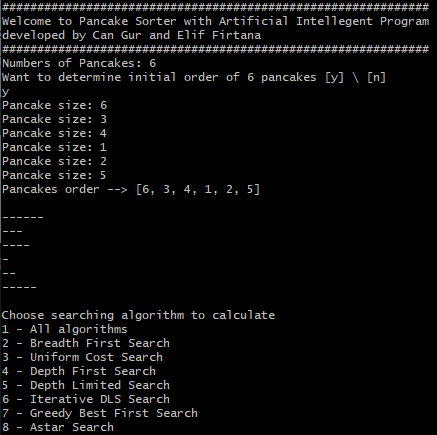
Let s = hs1,... ,sn+1i be an n-pancake state. Its heuristic value is the number of stack positions for which the pancake at that position is not of adjacent size to the pancake below:

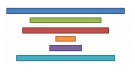
h gap(s) := |{i | i ∈ {1,... ,n}, |si − si+1| > 1}|.

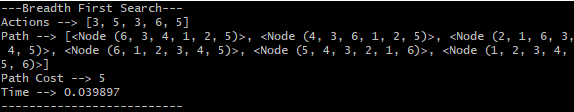
We get h gap(s) = 5 for the state in Fig. 1 because there are 5 gaps in this pancake stack, namely below positions 2, 3, 4, 5, and 6. For example, there is a gap below position 2 because the 2nd and 3rd pancake in the sequence differ in size by more than 1, and there is a gap below position 6 because the 6th pancake and the “plate” differ in size by more than 1. The only place without a gap is below position 1, since the two first pancakes in the sequence are of adjacent size. A pancake problem goal state has no gaps at all, and hence its heuristic value is 0. It is easy to see that a k-flip can reduce the number of gaps by at most 1: the only gap it can potentially “heal” is the one between positions k and k + 1. Hence, h gap is a consistent and admissible heuristic.1

**C – Discussion On the Result**

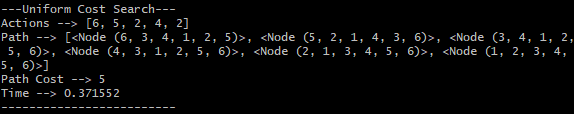
**C.1 Number of pancakes: 6**

****

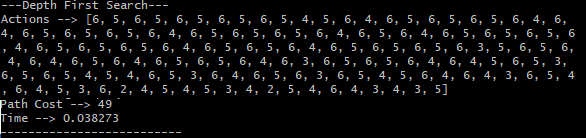
Pancakes order:   




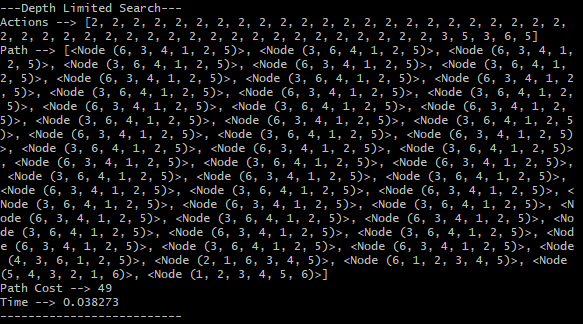
BFS is complete   
Time complexity of BFS is 0.39897 seconds  
Space complexity of BFS is **O(bd)**



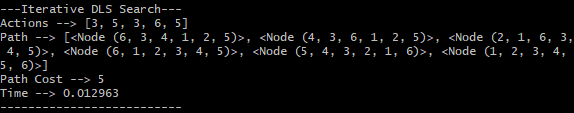
UCS is complete   
Time complexity of UCS is 0.371552 seconds  
Space complexity of UCS is **O(bc\*/e)**



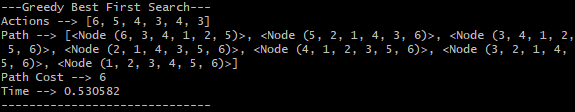
DFS is complete   
Time complexity of DFS is 0.038272 seconds  
Space complexity of DFS is **O(bm)**



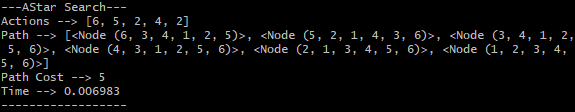
DLS is complete   
Time complexity of DLS is 0.038273 seconds  
Space complexity of DLS is **O(bl)**



IDS is complete   
Time complexity of IDS is 0.012963 seconds  
Space complexity of IDS is **O(bd)**



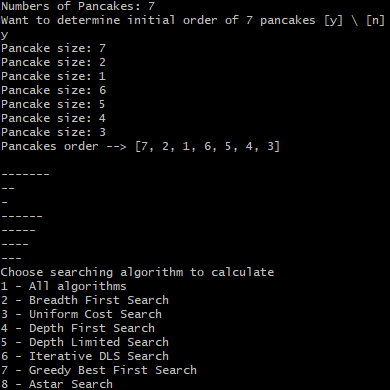
GBFS is complete  
Time complexity of GBFS is 0.530582 seconds  
Space complexity of GBFS is **O(bd)**

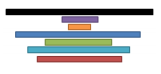


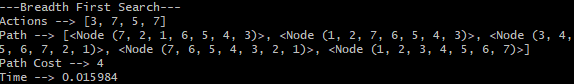
Astar is complete  
Time complexity of Astar is 0.006983 seconds  
Space complexity of Astar **O(bd)**

In 6 pancakes sorting,   
Astar algorithm caltulates at shortest time. (In 0.006983 seconds)  
Breadth First Search, Uniform Cost Search, Iterative Deeping Search, Astar Search solutions path costs are 5.

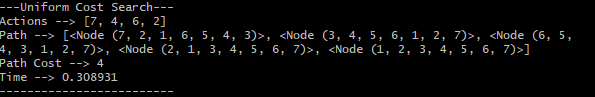
**C.2 Number of pancakes: 7**



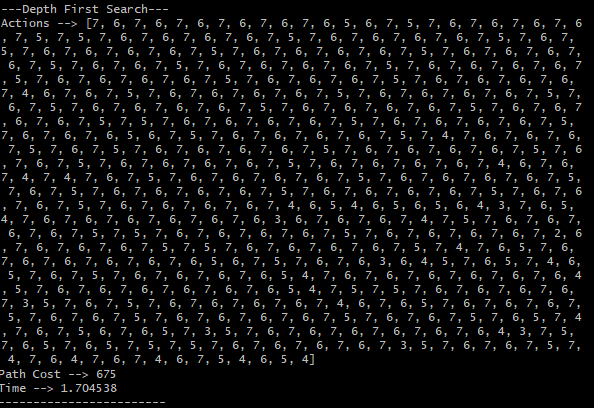
Pancakes order:   




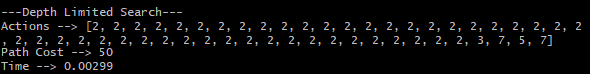
BFS is complete because b is finite.  
BFS is optimal because cost is equals 1 per step  
Time complexity of BFS is 0.015984  
Space complexity of BFS is **O(bd)**



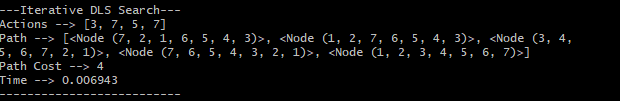
UCS is complete because step cost is positive  
UCS is optimal because nodes expanded in increasing order of 𝑝𝑎𝑡ℎ 𝑐𝑜𝑠𝑡  
Time complexity of UCS is 0.308931  
Space complexity of UCS is **O(bc\*/e)**



DFS is complete because b is finite  
DFS is not optimal because it finds the “leftmost” solution, regardless of depth or cost  
Time complexity of DFS is 1.704538  
Space complexity of DFS is **O(bm)**



DLS is complete  
DLS is  
Time complexity of DLS is 0.00299  
Space complexity of DLS is



IDS is complete because b is finite  
IDS is optimal because step cost equals 1  
Time complexity of IDLS is 0.006943  
Space complexity of IDLS is **O(bd)**

altılı pancake

GBFS is complete  
GBFS is  
Time complexity of GBFS is  
Space complexity of GBFS is **O(bd)**

altılı pancake

Astar is complete  
Astar is optimal  
Time complexity of Astar is  
Space complexity of Astar **O(bd)**

**References**1 Helmert M: Landmark Heuristics for the Pancake Problem. 2010 <http://www2.informatik.uni-freiburg.de/~ki/papers/helmert-socs2010.pdf>