Ryan Keller

[**The Pancake Problem**](https://en.wikipedia.org/wiki/Pancake_sorting)**:**

Given a stack of n pancakes of n different sizes, find the minimum number of “flips” to order the pancakes from largest to smallest (from bottom to top). This Python solution uses an implementation of A\* search to find the lowest-cost “path” to order the pancakes, then saves that path as a series of “flips.” It is generalized to handle any stack of size > 3 (smaller stacks are trivial to sort, obviously).

**1. The Pancake Problem as a search problem**

A search problem has five components:

1. Initial State – the starting position. In the case of the pancakes, it is the list of numbers passed in as input data (for example on a stack size n=5, e.g. [3, 2, 4, 5, 1], the numbers 1-5 represent the smallest thru the largest pancakes, respectively) The first item in the list represents the bottom pancake in the stack, while the last item in the list represents the top pancake.
2. Possible actions available – in our case, a stack of pancakes with size n can be flipped in (n-1) locations: from the absolute bottom (flip all the pancakes; we will call this Position 1), between the bottom and second-from-the-bottom pancakes (flip the top n-1 pancakes; Position 2), between the second- and third- from the bottom (flip the top n-2, Position 3)…all the way to Position (n-1) (flipping the top 2 pancakes). We will not consider flipping the top pancake (Position >= n) as it does not alter the state of the pancake stack at all.
3. Successor Function – the flip will (intuitively) reverse the order of all the pancakes above the flipping position. So, for example, if the stack [3, 2, 4, 5, 1] (with the bottom position being the first position in the list, as keeping with the convention of the problem description) was flipped at position 2, the result would be [3, 1, 5, 4, 2].
4. Goal Test – checking if a candidate has reached the goal state (e.g. for stack size n=5, [5, 4, 3, 2, 1]).
5. Path Cost Function (see below)

**2. Backward cost function**

The backward cost function is simply the number of flips already made. So if the stack [3, 2, 4, 5, 1] was flipped at position 2, yielding [3, 1, 5, 4, 2], the cost would be 1. It doesn’t make sense to assign a higher cost to flips of more pancakes (i.e. it costs more to flip at position 1 than position 4) because any solution that needs a flip at position 2 (for instance) cannot be solved by only flipping from positions 3 and 4 (the supposed “lower cost” flips).

**3. Heuristic function (forward cost)**

Keeping in line with the logic of the backward cost, the forward cost (heuristic function) is the estimated number of flips required to order the pancakes from the current position. The best way to estimate the number of flips necessary to order a stack is to use a modified version of the gap heuristic where the cost is the number of “gaps” in the pancakes (places where a pancake is not next to its correct goal-state neighbor) plus 1 if the largest pancake is not on the bottom of the stack. With the modification of checking for the largest pancake on the bottom, the possible configurations of pancakes with 1-flip solutions all have a forward cost of 1. E.g.

[5 4 3(GAP)1 2] Cost = 1

[5 4(GAP)1 2 3] Cost = 1

[5(GAP)1 2 3 4] Cost = 1

[(5 NOT BOTTOM)1 2 3 4 5] Cost = 1

//

[(5 NOT BOTTOM)1 2 3 5(GAP)4] Cost = 2

[5(GAP)1 2 4(GAP)3] Cost = 2

**4. Search**

At each step of the A\* search, the lowest-cost stack is pulled from the priority queue (or, if this is the first step, we just pull the input). All “flips” of the current state are generated (other than those that would cause immediate backtracking to the previous state). If any of these flips generate the goal state, we have reached the end of the search. Otherwise, all the generated stacks are graded with the heuristic test. The generated stacks are then added to the priority queue, weighted by their total backwards and forwards cost. Then, the next stack is pulled from the priority queue until we reach a goal state.