Assignment 02 - Informed Search

Peter Nadel

June 19, 2025

1 Instructions

To run the Pancake Problem script, follow these steps:

- Make sure your working directory is the "informed-search" directory with the `pwd` command.
- Once you are in that directory, simply run the file with `python pancake_problem.py`. This will solve the pancake problem on a random ordering of 1 through 10 using the gap heuristic.
- You can also specify a list that you want the script to solve by writing the list after the script command. For example, `python pancake_problem.py 3 2 5 1 4` would solve the pancake problem for the list [3, 2, 5, 1, 4].
- There are also a number of flags which alter behavior, as will be expanded upon below. `-u` or `--ucs` will solve the pancake problem with the Uniform Cost Search algorithm.

  `-t` or `--top` will solve the pancake problem with A*, using the top heuristic.

  `-p` or `--topprime` will solve the pancake problem with A*, using the top' heuristic.

  And `-l` or `--ltopprime` will solve the pancake problem with A*, using the L-top' heuristic.
- At the conclusion of the script, you should see reporting on what steps the algorithm took to solve the problem, how long it took, and how many nodes were visited.

2 Problem definitions

The pancake problem can be defined as a search problem. Though very different from the examples we've seen in class, this problem, when represented as a search tree, can be solved using the same informed search methods that are used for finding the destination in a maze or the cheapest cost path over or around geographic obstacles. Indeed, we can define each possible flip of the pancake stack as a node in a tree. A given node's parent is the stack orientation from which it was flipped and its children are all of the possible flips that can be done.

When viewed this way, the pancake problem can seem overwhelming. The possible solution space would increase dramatically if we were to check every possibility. Thankfully, using algorithms like Uniform Cost Search (UCS) and A* will allow us to efficiently traverse this tree and find the solution. To do so, I first defined the backward cost of flipping the pancake stack, by incrementing the backward cost of the parent by one for each new generation. As a result, the more flips the algorithm chooses to make, the more costly each flip will be. Second, I defined the heuristic for the A* algorithm. For this base case, I implemented the provided gap heuristic from Helmert [1].

Importantly, UCS can be used to solve the pancake problem and a thorough description of this can be found in my extra experiment. Suffice it to say that we can view UCS as a variant of A* but with a total cost equal to the backward cost or as A* with a heuristic that always returns zero.

3 Implementation

To solve the pancake problem, I began by implementing the `PancakeStack` class. This class represents a single node in the search tree described above. It takes in a state, a parent, an order of when it was

added to the tree, a backward cost and a heuristic. A `PancakeStack` also has an `obsolete` flag which may be set to `True` or `False`.

So that it could be used in the priority queue needed to implement A*, I added a `__lt__` method, which returns `True` if the current stack has a lower cost than the node it is compared to. Additionally, this method uses the order that the node was added to the priority queue as a way to break ties in total cost, with nodes added earlier taking precedence over those added later. I now needed a way to calculate the total cost, so I implemented a total_cost method, which adds together the backward cost and the result of the heuristic calculation. Last, I implemented a method that would flip the stack at a given index position and return a new `PancakeStack` object with the flipped state, the correct parent, and an incremented order added and backward cost.

Next, I took on implementing the `PriorityQueue` class. This object is a light wrapper around Python's `heapq` module, but I added several components that are specific to this problem. For example, when initializing the the `PriorityQueue`, in addition to a list to store the contents of the queue, a dictionary is also created. This dictionary stores state tuples and links them to the lowest cost node for that state. This step was critical in my implementation. As I will explain later, the A* algorithm requires that nodes whose state is the same but have a lower cost be popped first. To both facilitate this process and speed up finding nodes in the priority queue, this dictionary played an important role in this implementation.

With these two abstractions, I was able to implement the A* algorithm with the `AStar` class. First, I took in the initial state and the user-selected heuristic and created the root node, pushing it into the priority queue. Importantly, I also initialized a `visited` set, which keeps track of all of the states the algorithm has already seen. To search, I first noted all of the base cases, those times when the algorithm is supposed to stop:

- If the priority queue is empty, the algorithm should stop. This can only happen when the initial state is degenerate from the problem specification.
- If the current node is the `None` singleton, the algorithm should stop. This can only happen if the priority queue is only made up of obsolete nodes, again a situation which may only happen in a degenerate case.
- If the heuristic of the current stack is zero (or `None` in the case of the UCS heuristic), the algorithm has found the solution.

In all other cases, the A* algorithm will generate a new set of successor nodes and expand the frontier. This expansion is achieved by iterating through all of the index positions in the stack for which a flip would change the state, from 2 to the length of the stack plus 1. If the successor is in the `visited` set then it is passed over. Instead, if it is not in the `visited` set or the priority queue then it is pushed into the queue. Sometimes though, this state is in the priority queue already, in which case the algorithm cannot simply ignore it. Instead, it must first check if the existing state already in the priority queue has a higher cost than this new state. If it does, then it replaces the higher cost node with the lower cost one and marks the higher cost one as obsolete. Finally, to report the solution, we can take the final solution node and trace back to its parent until we return to the root node.

4 Extra experiment

4.1 Methods

For my extra experiment, I wanted to look at heuristic functions other than the gap heuristic. I chose three additional heuristics, described below:

- The Top Heuristic, which counts the number of pancakes that are not in their correct final position [2].
- The Top' (Top-Prime) Heuristic, which counts the pairs of adjacent pancakes that are not in the proper order [2].
- The L-Top' (L-Top-Prime) Heuristic, which counts the `L` length runs of adjacent pancakes that are not in the proper order, where `L` is any integer between 2 and the length of the stack plus 1 [2].

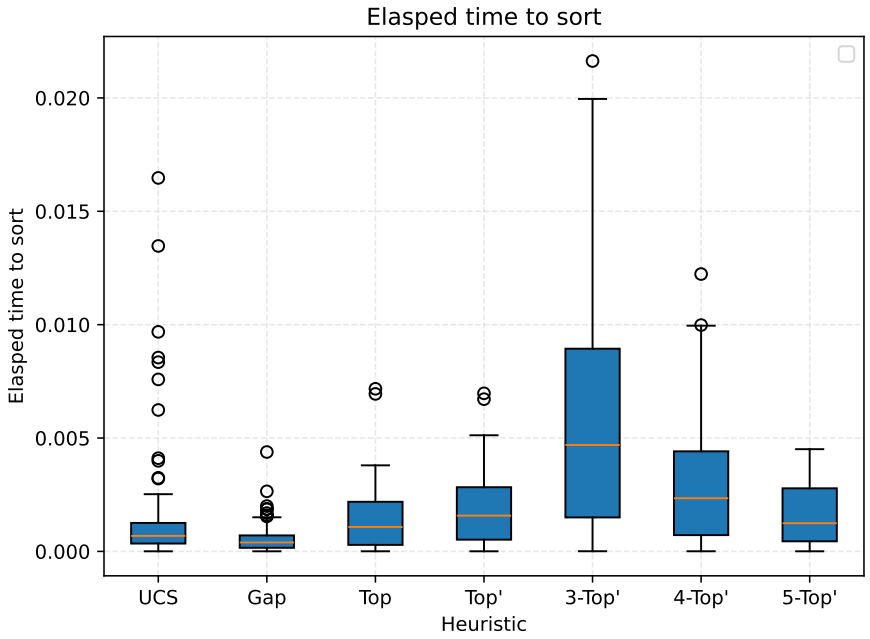

Figure 1: Distribution of elapsed time

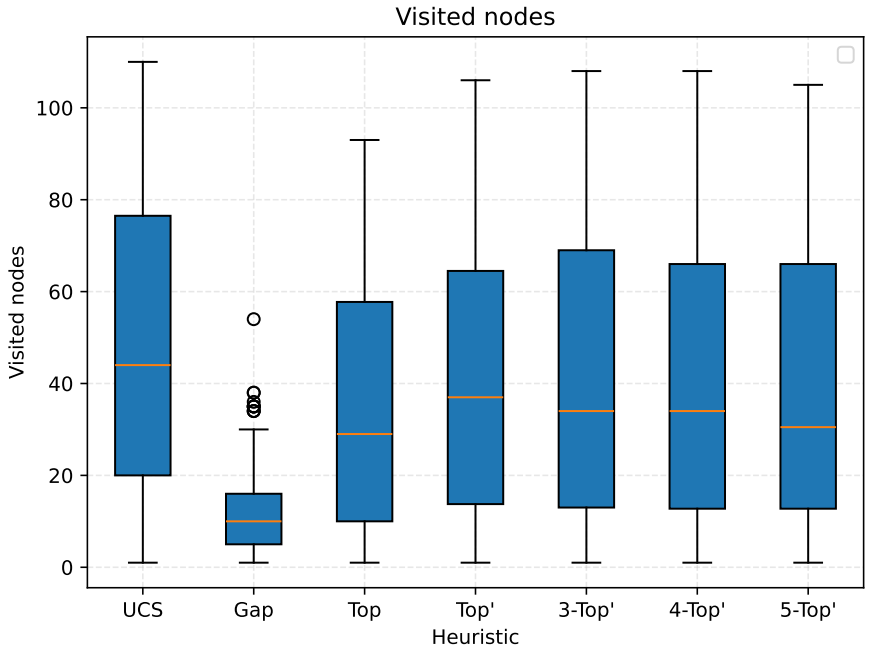

Figure 2: Distribution of visited nodes

Rather than in the gap heuristic, all three of these heuristics are based on the index positions of the pancakes in the state. This feature makes them more informed than the gap heuristic, that is, their heuristic cost is closer to the theoretical true cost of getting to the goal. To test these different heuristics, I generated 100 random stacks of 5 pancakes, as can be seen in the `benchmark.py` file. I then ran the A* algorithm with each of these heuristics. I also compared these results to the results with the UCS algorithm. I used two metrics to evaluate these heuristics: time elapsed to find a solution and the number of nodes visited by the algorithm. See the plots below.

4.2 Results and Discussion

As can be noted by the visuals above, the gap heuristic outperforms the other heuristics in terms of elapsed time and nodes visited, despite having a theoretically worse internal representation of the cost to the goal. Not only do the other heuristics report higher average elapsed times and visited nodes, but they also show a higher variance from this mean, suggesting that the gap heuristic is also more consistent than these other methods.

Interestingly, UCS maintains similar or better performance to the other heuristics which may tell us that these other heuristics are harming the search process as much as helping it. This is a surprising result and may be a result of the small size of the pancake stacks. Because I am only using stacks of length 5, the search space is small enough that UCS is not penalized for not having a heuristics, while the other non-gap heuristics face a penalty for searching the tree more rigorously. This result indicates that, for these small sequences, the forward cost, the heuristic, matters much less than the backward cost. This should stand to reason as the non-gap-heuristics are based solely off of index position. When these index positions are small, in the case of small sequences, the heuristic cost may be overwhelmed by the backward cost, allowing UCS to reach the performance levels of A* with these heuristics. The question for future research, then, is what qualifies as a small sequence, that is, at what point does the search space become too unwieldy for UCS and the forward cost begins to outweigh the backward cost.

References

[1] Malte Helmert. "Landmark Heuristics for the Pancake Problem." In: *Proceedings of the Third Annual Symposium on Combinatorial Search (SOCS-10)*. Jan. 2010.

[2] Danniel Yang. "Improved Heuristics for Solving the Pancake Problem". In: *Review of Undergraduate Computer Science* (2016).