Homework 1

Advanced AI

UAA CSCE A601

Patrick Pragman, UAID 30812903

Question 1:

Assuming that the chef has three pancakes, what are the goal state, action(s), and transition model?

Answer:

Goal State

The goal state is straight forward. If our state is a list of three pancakes:

```
s = (p1, p2, p3)
```

the goal state is to have those pancakes in order such that

```
p1 <= p2 <= p3
```

in our case specifically, where initial state `s_initial` is `s_initial = (2, 3, 1)`, we want to find

```
s_goal = (1, 2, 3)
```

Action(s) and Transition Model

The action that you can perform is to insert the spatula somewhere, and flip the stack of pancakes. There are really only a few possible changes you can make.

1. You can do nothing at all (which flips the 'zeroth' pancake) - this does nothing.
2. You can flip at the first pancake - this also does nothing.
3. You can flip at the second pancake which reverses `p1` and `p2`.
4. Finally, you can flip at the third pancake, which reverses the whole stack.

In psuedocode, if the state at any point looks like `(p1, p2, p3)` the results of flips kind look kind of like this:

- `Flip at 0 -> (p1, p2, p3)`
- `Flip at 1 -> (p1, p2, p3)`
- `Flip at 2 -> (p2, p1, p3)`
- `Flip at 3 -> (p3, p2, p1)`

So in general, for a state `s = (p1, p2, p3)`

`Actions((p1, p2, p3)) = {Flip0, Flip1, Flip2, Flip3}`

Since `Flip0 and Flip1` do not change our situation, and we don't think operating the spatula is a uniquely terrible burden, we're going to say that the only actions that we can perform then are two flip at 2 or flip at 3.

Thus the transition model is the description of actions available to us, in general:

`Result( (p1, p2, p3), Flip2) = (p2, p1, p3)`

and

`Result( (p1, p2, p3), Flip3) = (p3, p2, p1)`

For an arbitrarily sized pancake stack (state) `s = (p1, p2, p3, ... , pn)`.

The actions for a given state `s` are:

`Actions(s) = {Flip2, Flip3, Flip4, ..., Flipn}`

and the transition model follows the following pattern:

`Result( (p1, p2, p3, ... , pn), FlipM ) = (pM, p(M-1), p(M-2), ..., p1, p(M+1), ..., pn )`

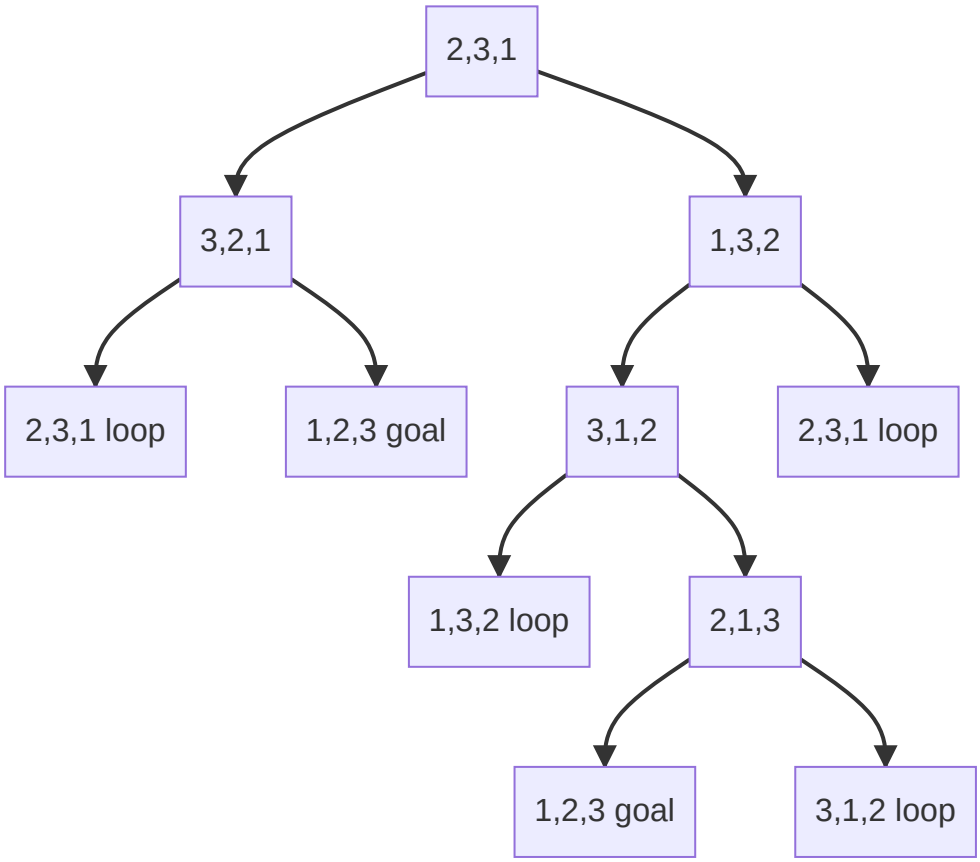
Question 2:

Assuming the following initial state, draw a search tree for the problem. You can show the nodes as (2, 3, 1).

Answer:

Ok, let's draw a tree!

General Tree for s = (2, 3, 1).

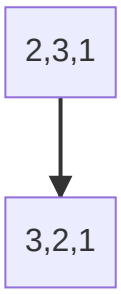


Question 3:

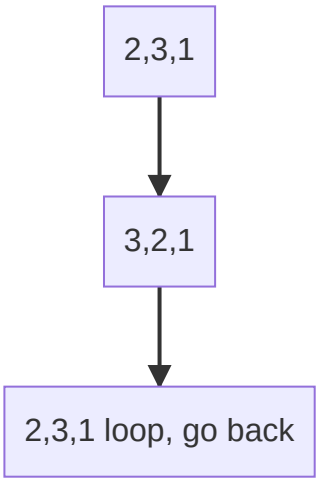
Find the path to the goal states using breadth-first search and depth-first search methods.

Depth First

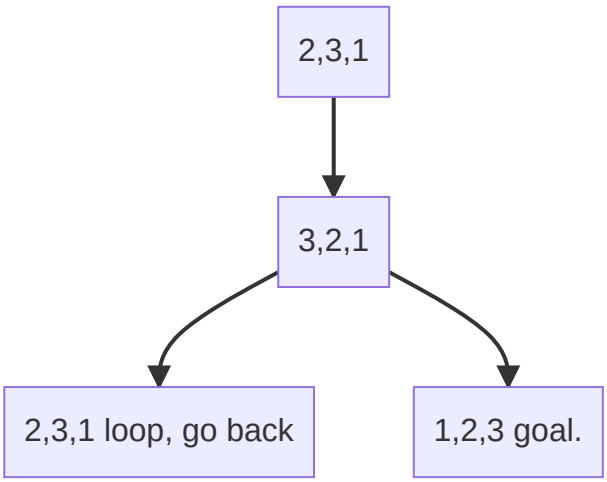
Step one: flip at 2



Step Two: flip at 2



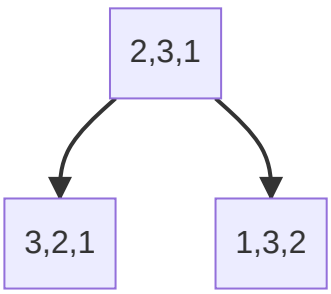
Step Three: alright try the other branch, now flip at 3



Success!

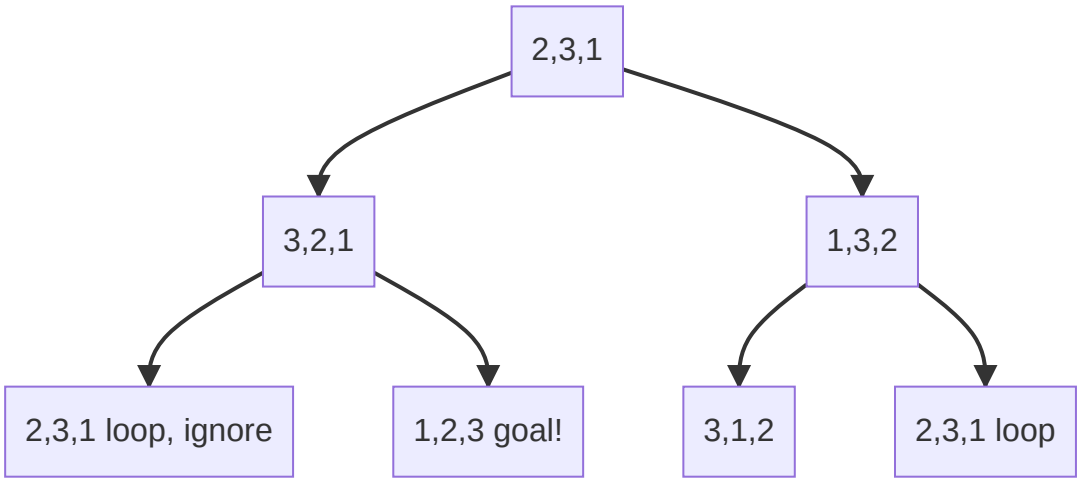
Breadth First

Step one: Expand the initial node



our goal isn't in the expanded nodes, so keep expanding

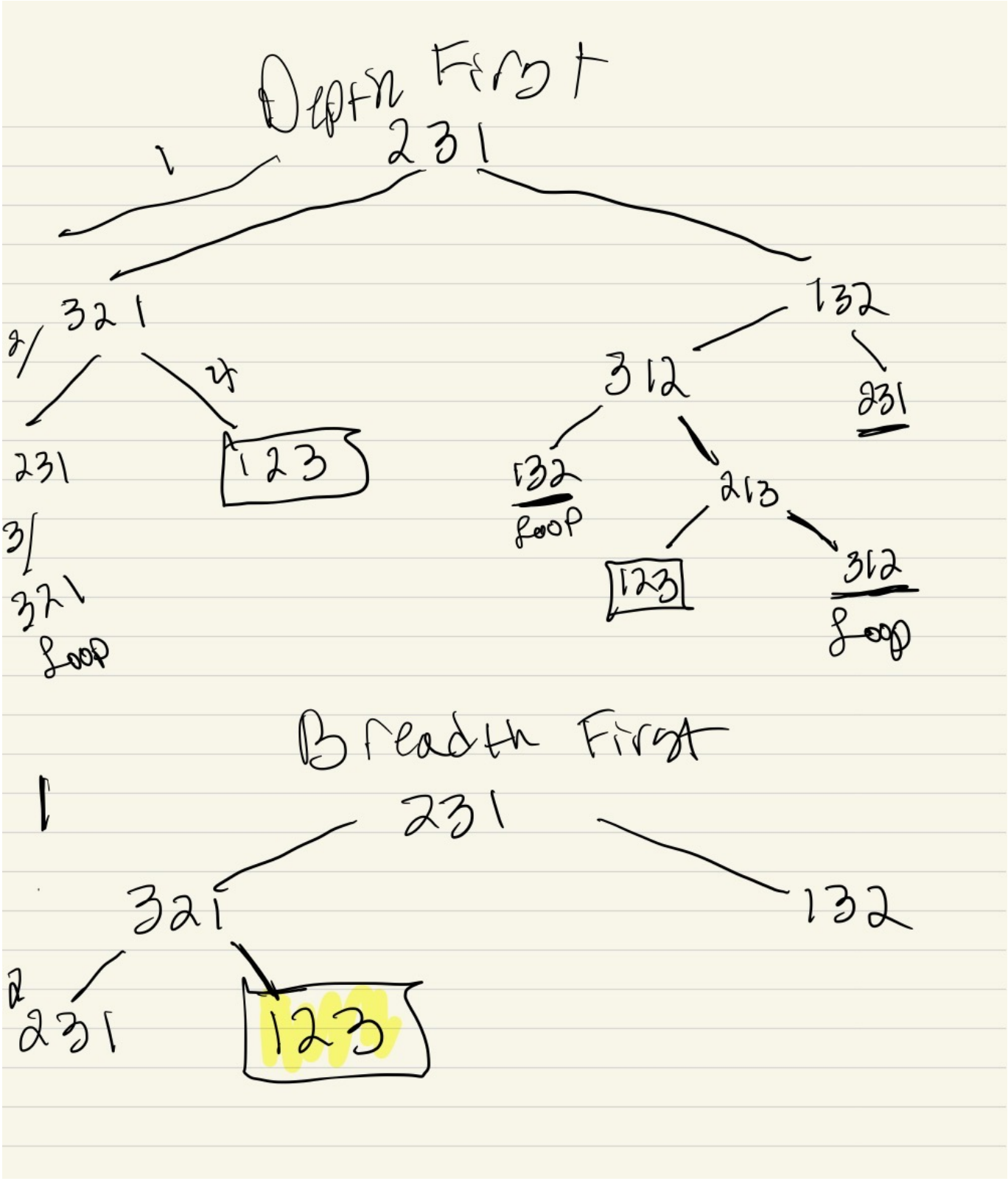
Step Two: Expand (3,2,1) and (1,3,2)



our goal node is in the second expansion.

Success!

My poorly hand drawn copy showing my train of thought when I sketched this out originally.



Question 4:

Answer:

Q4-1

```
frontier = PriorityQueue([node], key=f) ### Q4-1 It is a data structure defined above!
```

This is kind of cool, a priority queue is a stack where the "highest priority" element gets popped off first.

Q4-2

```
if s not in reached or child.path_cost < reached[s].path_cost: ### Q4-2 Look at the 4th bullet point in the slide 38 of uninformed search (Best-first search)
```

In this we're looking to compare the path cost, since it's best first search we only replace a child if the path cost of the child you're looking at is less than the previously reached node.

Q4-3

```
frontier = FIFOQueue([node])### Q4-3 It is a data structure defined above!
```

That's cool - I've never actually used the deque module before. Looks like we use a FIFOQueue here because the frontier we just opened is expanded first.

Q4-4

```
frontier = LIFOQueue([Node(problem.initial)])### Q4-4 It is a data structure defined above!
```

It makes sense for us to use a "Last in First Out Queue" in the Depth Limited Search, because we need to keep track of the depth of the search... if we go to deep, the LIFOQueue gets to long and we cut off the search.

Question 5:

Answer:

Q5-1

A* is equivalent to best-first-search where the evaluation function is:

```
f=lambda n: g(n) + h(n)
```

We see this in the notebook in Cell 3, Line 23:

```
return best_first_search(problem, f=lambda n: g(n) + h(n))
```

Q5-2

Breadth-first-search is equivalent to best-first-search where the evaluation function is `len`.

We see this in the notebook in Cell 3 line 39:

```
return best_first_search(problem, f=len)
```

the `f=len` sends the length unction to the best first search function as the key for the Priority Queue.

But, `len` isn't defined like it would be for a list. It's *not* the length of the priority queue `frontier`. The `len` is being used on `Node` objects.

In the very first cell (Cell 1, Line 38), in the `Node` class, the `len` function is overloaded with the double-underscore method `__len___`, now `len` returns:

```
return 0 if self.parent is None else (1 + len(self.parent))
```

Or, the function returns zero if it doesn't have a parent, or that parent's length function plus one.

In essence then, this returns the amount of steps from the initial node to the current node. That is, "how many steps" it would take to get to this same places. Which is particularly useful in a graph traversal.

Q5-3

Depth-first-search is equivalent to best-first search where the evaluation function is `-len(n)`. This is specified on Line 44 of cell 3.

A closer examination though, reveals the same thing as before, the `len` is the distance from the starting node. Since we send `-len` it looks like it swaps the order such that the deepest node is expanded first.

Q6

Look into the cell [7] class `PancakeProblem`. Where is the heuristic function defined? What does it mean?

Answer

It's defined on line 16 of that cell:

```
return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

in particular, in what's after the `return` statement.

This function takes the state `s` which is a tuple of integers (representing pancake size), then looks at the order of each `s[i]` and checks to see if `s[i - 1]` is greater than `s[i]`. That is at each `s[i]` it checks to see if it is in the right order. If it's in the wrong order, it adds 1 to a list. Finally it sums that list.

In short, the heuristic function returns the amount of "out of order" entries in the current state.

Q7

C5 (cell [8]), defines an an initial state for three pancakes. Run the program for the breadth-first-search, uniform-cost-search, depth-limited-search, and A*.

What do you get?

Answer

For all of them, I get the same thing:

```
[(2, 3, 1), (3, 2, 1), (1, 2, 3)]
```

Q8

C3 (cell [8]), defines an an initial state for seven pancakes.
Run depth-limited-search with depth of 4,5,6,7,8, and 9.
Then run the iterative-deepening-search.
What did you get? What do we understand?

Answer

So, I ran the following code in the last couple of cells:

```
print('Depth Limited Search Results:')
for max_depth in range(4, 10):
    print('Running DLS with limit =',max_depth)
    print(path_states(depth_limited_search(c3, limit=max_depth)))

print('Iterative Deepening Search Results:')
print(path_states(depth_limited_search(c3)))
```

which yielded the following results:

```
Depth Limited Search Results:
Running DLS with limit = 4
[]
Running DLS with limit = 5
[]
Running DLS with limit = 6
[(1, 7, 2, 6, 3, 5, 4), (7, 1, 2, 6, 3, 5, 4), (4, 5, 3, 6, 2, 1, 7), (5, 4, 3,
6, 2, 1, 7), (3, 4, 5, 6, 2, 1, 7), (6, 5, 4, 3, 2, 1, 7), (1, 2, 3, 4, 5, 6,
7)]
Running DLS with limit = 7
[(1, 7, 2, 6, 3, 5, 4), (4, 5, 3, 6, 2, 7, 1), (5, 4, 3, 6, 2, 7, 1), (3, 4, 5,
6, 2, 7, 1), (6, 5, 4, 3, 2, 7, 1), (2, 3, 4, 5, 6, 7, 1), (7, 6, 5, 4, 3, 2,
1), (1, 2, 3, 4, 5, 6, 7)]
Running DLS with limit = 8
[(1, 7, 2, 6, 3, 5, 4), (4, 5, 3, 6, 2, 7, 1), (7, 2, 6, 3, 5, 4, 1), (1, 4, 5,
3, 6, 2, 7), (6, 3, 5, 4, 1, 2, 7), (2, 1, 4, 5, 3, 6, 7), (5, 4, 1, 2, 3, 6,
7), (3, 2, 1, 4, 5, 6, 7), (1, 2, 3, 4, 5, 6, 7)]
Running DLS with limit = 9
[(1, 7, 2, 6, 3, 5, 4), (4, 5, 3, 6, 2, 7, 1), (7, 2, 6, 3, 5, 4, 1), (1, 4, 5,
3, 6, 2, 7), (2, 6, 3, 5, 4, 1, 7), (4, 5, 3, 6, 2, 1, 7), (5, 4, 3, 6, 2, 1,
7), (3, 4, 5, 6, 2, 1, 7), (6, 5, 4, 3, 2, 1, 7), (1, 2, 3, 4, 5, 6, 7)]
Iterative Deepening Search Results:
[(1, 7, 2, 6, 3, 5, 4), (4, 5, 3, 6, 2, 7, 1), (7, 2, 6, 3, 5, 4, 1), (1, 4, 5,
3, 6, 2, 7), (2, 6, 3, 5, 4, 1, 7), (4, 5, 3, 6, 2, 1, 7), (6, 3, 5, 4, 2, 1,
7), (3, 6, 5, 4, 2, 1, 7), (4, 5, 6, 3, 2, 1, 7), (6, 5, 4, 3, 2, 1, 7), (1, 2,
3, 4, 5, 6, 7)]
```

I think this tells that if we know the depth of that the target node is on, then a Depth Limited search can be better than arbitrarily digging down to some "big" depth limit. You'll notice when the depth is greater than the first level that the goal is on, unnecessary steps get completed.

This tells me that a "breadth-first-search" is probably the better way to go when tackling tree-like structures. Especially when your target node is at an unknown depth, or the tree is extremely broad.

Similarly, iterative deepening search *will* yield a result eventually, but it may be much deeper than optimal if the tree is laid out in the wrong way. A bigger limit doesn't result in faster search times and smaller answers.