CISC 481 Programming Assignment 1

February 13, 2022

Assignment Objectives

After completing this assignment, you'll understand the inner workings of the following search algorithms:

- iterative-deepening search
- breadth-first search
- A* search

You'll also gain some experience using a software profiler, which is a tool that lets you gather and analyze performance characteristics of your code. You'll use the statistics you gather to see in a very real sense the difference in time and space complexity of the above algorithms.

The Pancake Sorting Problem

This is a variation on the general idea of sorting. The idea is you have a stack of pancakes, each one a different size from all the others. Initially, the pancakes are stacked in some random order, and we'd like to get them stacked in order of size from smallest on top to largest on the bottom. At any point in time, you can flip a portion of the stack by placing a spatula in between two pancakes and turning over, as a block, the pancakes above the spatula. See Figure 1 for an illustration of this. You can also take a look at the Wikipedia article on pancake sorting for more information.

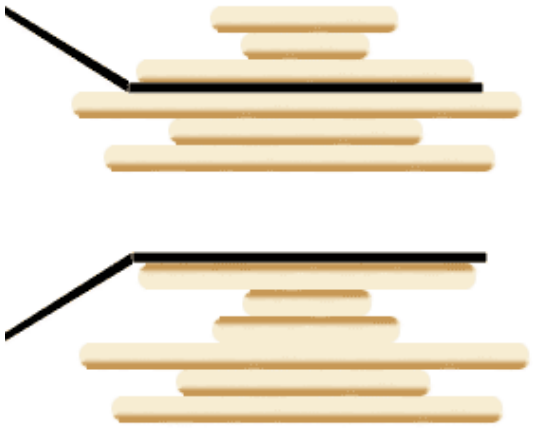

Figure 1: From Wikipedia, a graphical depiction of the `(:flip 3)` action. On the top half of the image, we see the spatula underneath the third pancake from top. On the bottom half, we see the result of the flip.

Representing the Problem

[My examples here are in Lisp, but feel free to use something similar that's easy to parse in your target language.]

While you can probably imagine many schemes to represent a stack of pancakes, there's actually an incredibly simple way of doing so: as just a simple list of numbers. These numbers don't even have to be in sequence, but it's easiest to represent them this way (*e.g.* a sorted stack of five pancakes would be represented as `(1 2 3 4 5)`. In Figures 2, 3, and 4 I give some names to a few initial unsorted stacks that I'll refer to throughout the writeup.

```
(defvar *stack-0* '(1 3 5 2 4 6))
```

Figure 2: This is a small stack, and its solution has an easily recognizable pattern to it. It's useful for testing.

```
(defvar *stack-1* '(8 2 1 7 5 4 6 3 9 10))
```

Figure 3: This stack is solvable in seven flips. This will also be useful for testing, and I'll use it to provide example output here. It may well be the only stack other than `*stack-0*` that your iterative deepening search will solve in any reasonable amount of time.

```
(defvar *stack-2* '(9 5 2 8 4 1 10 6 7 3))

(defvar *stack-3* '(2 8 10 5 7 3 4 6 1 9))

(defvar *stack-4* '(3 6 8 10 7 1 5 4 2 9))

(defvar *stack-5* '(6 9 4 8 1 3 2 7 10 5))

(defvar *stack-6* '(8 5 10 6 2 9 3 4 1 7))

(defvar *stack-7* '(8 1 10 5 3 7 4 9 2 6))
```

Figure 4: Several other stacks I randomly generated (`*stack-7*` I had to search for - it takes eleven flips to solve optimally, which is as long as a ten pancake stack can take). You'll be using these to profile and compare your breadth-first and A* searches.

Rules of Movement (Actions)

In each state, we can choose to flip the stack starting from a particular pancake. As such, an *action* can be specified as $flip(n)$[1], where n is a natural number between 1 and the number of pancakes in the stack. For example, $flip(3)$ would specify we should stick the spatula between the third and fourth pancake in the stack, and then flip the top three pancakes such that what was the third from the top pancake becomes the top, and what was the top becomes the third from the top[2] All actions have a path cost of 1.

[1] One way to represent this in Lisp is a list starting with the keyword `:flip`, followed by a number - *e.g.* `(:flip 3)`.

[2] **NB** that the pancake which was second from the top doesn't change its position.

Part 1 [10 pts]

Write a function `possible-actions` that takes a stack as input and outputs a list of all actions possible on the given stack.[3]

Part 2 [10 pts]

Write a function `result` that takes as input an action and a stack and outputs the new stack that will result after actually carrying out the input move in the input state. Be certain that you do not accidentally modify the input stack variable.[4]

Part 3 [5 pts]

Write a function `expand` that takes a stack as input, and outputs a list of all states that can be reached in one Action from the given state.[5]

Part 4 [15 pts]

Implement an *iterative deepening search* which takes an initial stack and a goal stack and produces a list of actions that form an optimal path from the initial stack to the goal. Test your search on `*stack-0*` and `*stack-1*`. You can try running it on some of the other stacks, but don't feel discouraged if it takes a very long time before returning an answer.[6]

Part 5 [15 pts]

Implement a *breadth-first search* which, like the iterative deepening search, takes an initial stack and a goal and gives an optimal sequence of actions from the initial state to the goal. Test your search on `*stack-0*` and `*stack-1*`.

Part 6 [20 pts]

For this part, you'll be implementing *A* search*.

Part 6.1

Similar to breadth-first and iterative deepening, your A* search should take as input an initial stack and a goal stack. It should additionally take a *heuristic function*[7]

You can test your code by passing as the heuristic a function that always returns 0[8], which should reduce your search to *uniform-cost search*. Test it on `*stack-0*`. You can also try it on `*stack-1*` for good measure. It may take a long time to solve any of the others.

---

[3] This is the $ACTIONS$ function we discussed in class and in the book.

[4] This is the $RESULT$ function we discussed in class; the state transition model

[5] **Hint:** This is a trivial extension of Parts 1 and 2.

[6] As I write this, I have my own implementation of iterative deepening attempting to solve `*stack-7*`. It's been at least ten, possibly twenty minutes. It's still plugging along.

[7] At this point, if you're using a language which does not have first-class functions and higher-order functions, you'll have to figure out some way to parameterize the heuristic function to your search. This could be using function pointers in a language such as C/C++, or passing in a *e.g.* `Heuristic` object that has an `call` method in Java (as a metafootnote - Java has an interface `Callable` for precisely this purpose). Common Lisp, as well as Python, JavaScript, and Haskell to name a few have first-class and higher-order functions.

[8] In Lisp, you can achieve this by calling `(constantly 0)`.

Part 6.2

Now you'll implement what we'll call the *non-adjacent pairs* heuristic. To calculate this, you'll look at each pair p_1 and p_2 of pancakes stacked one on top of the other. Add one if the pair is not in sequence (that is, if $|p_1 - p_2| \neq 1$). For example, looking back at Figure 3, for `*puzzle-1*`, the non-adjacent pairs heurstic value of `*puzzle-1*` is 6. This is because every pair except for $(2, 1)$, $(5, 4)$, and $(9, 10)$ are *not* adjacent. Test your code using this heuristic on `*stack-0*` and `*stack-1*`. You should now see that you get an answer for `*stack-1*` almost instantly. This illustrates very clearly just how much choosing a good heuristic matters for the practicality of A* search.

Part 7 [25 pts]

In this final part, you'll be benchmarking your searches to get an empirical sense of the differences in time/space complexity between them. For this you'll need a profiler. **NB** that some profilers also include memory usage statistics, which may give a good idea about the maximum amount of memory used at any point on a given run. If the profiler for your language does not include this, you'll have to instrument your code to keep track of the maximum number of nodes on the frontier at any one time over the entire run. In Figures 5 and 6 I list what I get when profiling runs of my reference implementations of A* and breadth-first search. This output was generated using SBCL's profiler. SBCL is the Common Lisp implementation that I use. Like most of Common Lisp tooling, the profiler is easily accessible using SLIME in Emacs[9]

What we're interested in here is "consed" (which is a good proxy for the total amount of memory used) and the number of calls we made to `expand`. The former gives us an idea of the space complexity, and the latter the time complexity. Note that A* performs orders of magnitude better than breadth-first on both counts! You can also see that A* is better in terms of actual time - taking a small fraction of a second to complete where breadth-first takes almost thirteen seconds.

```
  seconds  |     gc     |   consed   | calls |  sec/call  |  name
-----------------------------------------------------------
     0.002 |      0.000 | 1,730,096 |     1 |   0.002332 | CISC481-22S-PROG1::A*-SEARCH
     0.000 |      0.000 |    32,768 |    56 |   0.000003 | CISC481-22S-PROG1::EXPAND
     0.000 |      0.000 |         0 |   446 |   0.000000 | CISC481-22S-PROG1::NON-ADJACENT-PAIRS
     0.000 |      0.000 |    65,536 |   560 |   0.000000 | CISC481-22S-PROG1::RESULT
     0.000 |      0.000 |    32,752 |    56 |   0.000000 | CISC481-22S-PROG1::POSSIBLE-ACTIONS
-----------------------------------------------------------
     0.003 |      0.000 | 1,861,152 | 1,119 |            | Total

estimated total profiling overhead: 0.00 seconds
overhead estimation parameters:
  2.888e-9s/call, 1.386496e-6s total profiling, 6.45856e-7s internal profiling

These functions were not called:
 CISC481-22S-PROG1::BREADTH-FIRST-SEARCH
 CISC481-22S-PROG1::DEPTH-LIMITED-SEARCH
 CISC481-22S-PROG1::ITERATIVE-DEEPENING-SEARCH
 CISC481-22S-PROG1::QUEUE-POP CISC481-22S-PROG1::QUEUE-PUSH
```

Figure 5: Profiler output for A* on `*stack-1*` using the non-adjacent pairs heuristic.

Part 7.1

Profile each of iterative deepening search, breadth-first search, and A* search using non-adjacent pairs solving `*stack-0*`. Even on this simple puzzle solvable in only five flips, you should be

[9] If you're a VIM user, there's the analogous SLIMV. There's also a WIP plugin that might get you the functionality you need in VSCode.

```
  seconds  |     gc     |     consed     |   calls   |  sec/call  |  name
-----------------------------------------------------------------------
    12.454 |      3.478 | 10,561,050,976 |         1 |  12.453781 | CISC481-22S-PROG1::BREADTH-FIRST-SEARCH
     0.299 |      0.108 |    454,864,592 | 2,926,980 |   0.000000 | CISC481-22S-PROG1::RESULT
     0.065 |      0.000 |     33,488,896 |   945,293 |   0.000000 | CISC481-22S-PROG1::QUEUE-PUSH
     0.040 |      0.000 |    211,701,216 |   292,698 |   0.000000 | CISC481-22S-PROG1::POSSIBLE-ACTIONS
     0.025 |      0.000 |              0 |   292,698 |   0.000000 | CISC481-22S-PROG1::QUEUE-POP
     0.000 |      0.000 |     41,189,376 |   292,698 |   0.000000 | CISC481-22S-PROG1::EXPAND
-----------------------------------------------------------------------
    12.883 |      3.586 | 11,302,295,056 | 4,750,368 |            | Total

estimated total profiling overhead: 6.59 seconds
overhead estimation parameters:
  2.888e-9s/call, 1.386496e-6s total profiling, 6.45856e-7s internal profiling

These functions were not called:
 CISC481-22S-PROG1::A*-SEARCH CISC481-22S-PROG1::DEPTH-LIMITED-SEARCH
 CISC481-22S-PROG1::ITERATIVE-DEEPENING-SEARCH
 CISC481-22S-PROG1::NON-ADJACENT-PAIRS
```

Figure 6: Profiler output for breadth-first search on `*stack-1*`.

able to get a sense of the difference in performance characteristics between these three algorithms.

Part 7.2

Just to drive the point home about choosing a good heuristic, profile A* on `*stack-0*` using the constantly 0 heuristic, and compare it to the results you got when using the non-adjacent pairs heuristic.

Part 7.3

Generate profiler reports for both of A* using non-adjacent pairs and breadth-first search on `*stack-2*`, `*stack-3*`, `*stack-4*`, `*stack-5*`, `*stack-6*`, and `*stack-7*`.[10] Then calculate averages over each of the six runs for the amount of memory used and number of calls to `expand` for both A* and breadth-first search.

Solutions to `*stack-0*` and `*stack-1*`

Hopefully the listings in Figures 7, 8, and 9 will be useful for testing your code. Note that A* and breadth-first search find different solutions for `*stack-1*` - there's more than one optimal solution to this puzzle and they each find their own.

```
(:ACTION-SEQUENCE ((:FLIP 3) (:FLIP 5) (:FLIP 4) (:FLIP 3) (:FLIP 2))
 :STATE-SEQUENCE
 ((1 3 5 2 4 6) (5 3 1 2 4 6) (4 2 1 3 5 6) (3 1 2 4 5 6) (2 1 3 4 5 6)
  (1 2 3 4 5 6))
 :PATH-COST 5)
```

Figure 7: Optimal solution to `*stack-0*`

[10] For a total of 12 separate profiler reports.

```
(:ACTION-SEQUENCE
 ((:FLIP 8) (:FLIP 2) (:FLIP 4) (:FLIP 3) (:FLIP 5) (:FLIP 7) (:FLIP 2))
 :STATE-SEQUENCE
 ((8 2 1 7 5 4 6 3 9 10) (3 6 4 5 7 1 2 8 9 10) (6 3 4 5 7 1 2 8 9 10)
  (5 4 3 6 7 1 2 8 9 10) (3 4 5 6 7 1 2 8 9 10) (7 6 5 4 3 1 2 8 9 10)
  (2 1 3 4 5 6 7 8 9 10) (1 2 3 4 5 6 7 8 9 10))
 :PATH-COST 7)
```

Figure 8: Optimal solution to `*stack-1*` as found by A*.

```
(:ACTION-SEQUENCE
 ((:FLIP 8) (:FLIP 5) (:FLIP 3) (:FLIP 4) (:FLIP 2) (:FLIP 7) (:FLIP 2))
 :STATE-SEQUENCE
 ((8 2 1 7 5 4 6 3 9 10) (3 6 4 5 7 1 2 8 9 10) (7 5 4 6 3 1 2 8 9 10)
  (4 5 7 6 3 1 2 8 9 10) (6 7 5 4 3 1 2 8 9 10) (7 6 5 4 3 1 2 8 9 10)
  (2 1 3 4 5 6 7 8 9 10) (1 2 3 4 5 6 7 8 9 10))
 :PATH-COST 7)
```

Figure 9: Optimal solution to `*stack-1*` as found by breadth-first search.

Submitting

You should submit all of your code - *document it appropriately, as you'll be graded on style.* You should also submit listing of all of the raw profiler reports that you're asked to generate in Part 7, as well as the averages you calculated.