AI Project 1: Brute Force TSP

Robert Max Williams

Introduction

Traveling Salesperson Problem (TSP) is the problem of finding the minimum path that visits n points and returns to the start points. The brute force approach to TSP is very simple: generate all permutations of n points, and pick the permutation with the minimum round-trip distance. This approach is guaranteed to find the minimal length path(s) but is extremely expensive as n becomes large. Specifically, it runs in $O(n!)$.

Development

I chose between C, Python, and Common Lisp for the implementation language. Development is fastest in Python, but the code runs around 100x slower than C so it was not a good candidate for doing barely-feasible NP hard problems: the 100x speed difference could mean the difference between a few minutes and several hours. C requires much more bit mangling, and its verbosity leads to bloated, difficult to develop code, especially in a prototyping environment. So, I picked Common Lisp because it is nearly as expressive as Python (and sometimes more so) while being nearly as fast as C.

The first step is reading the `.tsp` files. The files consist of a header, with some metadata, followed by a line containing only "`NODE_COORD_SECTION`", followed by the points in the format "`point# x y`". I wrote regular expressions to extract the data from the header and code coord section. I ended up never using any of the data from the header, but it might be useful in future projects.

Next, I create a point-to-point distance function, `distance`. I kept the format from the input file, storing points as lists like this: `'(1 12.326521 54.12932)`. The two points are destructured ("unpacked" in Python terminology) and compared with this simple one-liner:

```
(sqrt (+ (expt (- x1 x2) 2) (expt (- y1 y2) 2)))
```

which is equivalent to this equation:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Then I created the function `evaluate` which takes in an array of points and calculates the round trip distance between all of them. It iterates through the array of points, while keeping track of the last point, so that at each iteration the distance between two adjacent points can be calculated. The previous-point variable is initialized to the last point in the sequence, otherwise the distance from the last point to the start would not be included.These are all added up to get the total distance.

Finally, all permutations of the points have to be generated and the best one saved or printed out, by `find-best-path`. The algorithm for generating permutations is quite simple, here in pseudocode:

```
def permutation(sequence, n=0):
      if n == sequence.length - 1:
            print(sequence)
            return
      for i from n to sequence.length:
            sequence.swap(i, n)
            permutation(sequence, n+1)
            sequence.swap(i, n)
```

Originally, I was keeping all of the permutations in memory, but this created an $n!$ memory cost, which used up all of my RAM for $n = 10$. In Python, the solution would be to `yield` from the permutation generating function, which acts as a coroutine. Common Lisp lacks this convenient method for creating coroutines (or any built in concurrency) so I had to refactor my code to avoid storing all of the permutations. As the algorithms in the course get more complex, I need to familiarize myself with better means of lazy iteration in Common Lisp to keep my code modular. Instead, I worked the cost minimization into the permutation generating loop, returning `'(cost points)` pairs at the bottom of the recursion and minimizing in each loop before returning up to the next level. Pseudocode for my solution is shown here:

```
def permutation(sequence, n=0):
      if n == sequence.length - 1:
            return [evaluate(sequence), sequence]
      min_cost = INFINITY
      best_path = None
      for i from n to sequence.length:
            sequence.swap(i, n)
            cost, path = permutation(sequence, n+1)
            if cost < min_cost:
                  min_cost = cost
                  best_path = path.copy()
            sequence.swap(i, n)
      return [min_cost, best_path]
```

If a lazy iterator for the permutations was available, this part of the program would be much simpler, only a single loop and no recursion. In both cases, memory usage is greatly reduced and even with $n = 12$ memory usage is very small.

Results

name	time	distance	path
Random4.tsp	318 MICROS	215	[1, 3, 2, 4]
Random5.tsp	232 MICROS	139	[1, 2, 5, 3, 4]
Random6.tsp	459 MICROS	119	[1, 2, 3, 4, 5, 6]
Random7.tsp	2 MILLIS	64	[1, 2, 7, 3, 6, 5, 4]
Random8.tsp	21 MILLIS	311	[1, 6, 8, 4, 5, 2, 3, 7]
Random9.tsp	154 MILLIS	131	[1, 7, 6, 3, 5, 2, 9, 4, 8]
Random10.tsp	1.3610001 SECS	107	[1, 2, 7, 6, 8, 5, 9, 10, 4, 3]
Random11.tsp	14.670001 SECS	253	[1, 2, 4, 3, 5, 7, 9, 8, 11, 10, 6]
Random12.tsp	197.58601 SECS	66	[1, 8, 2, 3, 12, 4, 9, 5, 10, 6, 7, 11]

Table 1: Path, distance, and time for all runs.

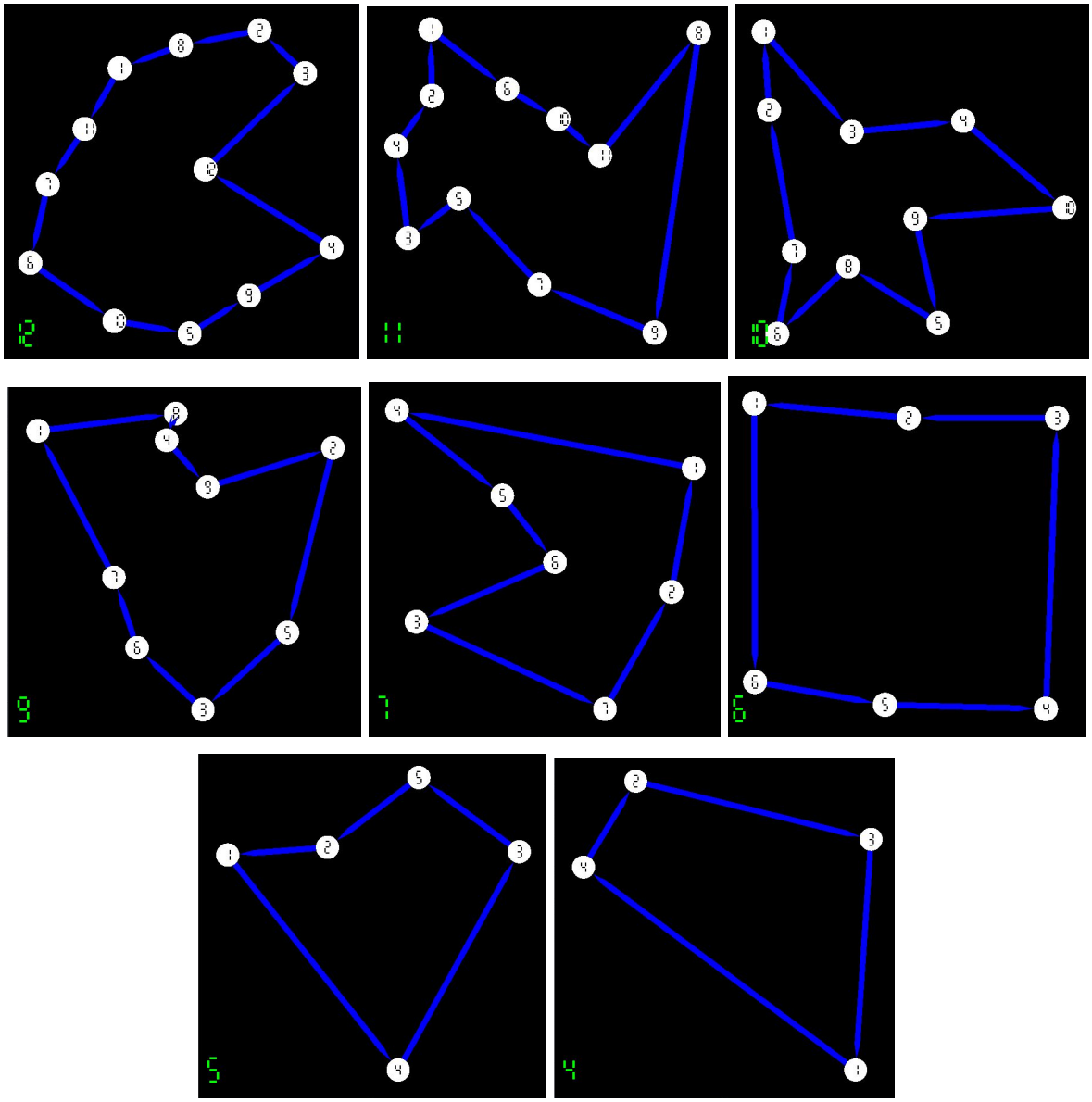

Figure 1: Plots of each solution. This is part of an SDL2/OpenGL user interface that will be extended in future projects. *Side note: is n=12 supposed to be pacman, or is that the form of a certain kind of problem?*