CS 3005: Programming in C++

The Traveling Salesman Problem (Part 2 - Good Cycle)

The Traveling Salesman Problem (TSP) is a classic NP-Complete problem. Solutions to the problem can be applied to many fields.

There are several variations of the problem. For this assignment we will use the following description. As you search for more information and solutions, be sure that they apply to this version.

Problem Definition

Given a complete graph G, with n vertices, v_1, v_2, ..., v_n, and edge weights w(u, v), such that w(u, v) == w(v, u) and the edge weights obey the triangle inequality, find the lowest weight cycle that visits all vertices.

The triangle inequality means that it is never more costly to go from vertex a to c, than it is to go from a to b then to c.

A cycle that visits all vertices must follow a path from the starting location, v_1, and visit every vertex exactly once, then return to the starting location v_1.

As mentioned above, finding an optimal solution to this problem is known to be NP-Complete. The consequence of this is that for arbitrary graphs with more than a few tens of nodes, it is not feasible to find optimal solutions.

Assignment

Since finding an optimal solution is not feasible, in this step of the assignment, you will be required to find a good cycle of a graph. We define a good cycle as one that has a cycle quality of at least 0.80.

There are many known approximation algorithms for TSP. You may implement any one algorithm you find, but it is recommended you choose one of the approximation algorithms described below. Your chosen algorithm must be able to achieve a cycle quality of at least 0.80 on all of the sample graphs. Also, the algorithm must take no more than 60 CPU seconds on any one of the sample graphs, when run on oxygen or nitrogen.

Algorithm 1 - Hill-Climbing-TSP

Hill-Climbing is a general algorithmic technique where a random solution is found and evaluated. Then, neighbor solutions (similar solutions) are found and evaluated. If any of the neighbors are better than the original, the original is replaced. This process is repeated until no better neighbors are found. It usually does not lead to an optimal solution.

Pseudo-Code

```cpp
// initialization
set the best cycle to an empty cycle
set the best quality to infinitely bad
randomly select a cycle
set the current cycle to a randomly selected cycle
set the current quality to the quality of the randomly selected cycle

// search
while current quality is better than best quality
    update best quality and best cycle from current quality and current cycle
    repeat as much as desired
        set temporary cycle as randomly selected neighbor cycle of best cycle
        set temporary quality to the quality of the temporary cycle
        if temporary quality is better than current quality
            update current quality and current cycle from temporary quality and temporary cycle
            break from "repeat as much as desired"
return best cycle
```

Be careful, repeat as much as desired, may cause your program to not find a good enough solution, or may cause your program to run too long. It’s usually a good idea to make the number of repetitions depend on the size of the problem (number of vertices in the graph).
In our problem, a neighbor cycle is generated by swapping any two vertices in the cycle. Except, never swap the first vertex. That should always be the home city, number 1.

**Algorithm 2 - Greedy-TSP**

Greedy is a general algorithmic technique where the next step in the algorithm is chosen to make the best advancement given the current state. Then, the state is updated, and the process is repeated until a solution is found. It usually does not lead to an optimal solution.

The greedy TSP algorithm, greedily adds edges to its solution, by picking the lightest edge that leads to a single cycle that visits every vertex exactly once. Such a cycle will have exactly 2 edges that touch each vertex. So, we don’t use edges that would cause more than 2 edges to touch a vertex. Also, such a cycle must include all vertices, so we will not include edges that close the cycle.

**Pseudo-Code**

```plaintext
// initialization
set solution collection of edges to be empty
sort all edges by weight in ascending order
// greedy steps
for each edge
  if either node in the edge already has 2 edges in the solution,
    skip this edge
  else if adding the edge to the solution would produce a cycle in the solution,
    skip this edge
  else
    add the edge to the solution
// close the full cycle
find the vertices with only one edge each in solution collection
add the edge between them
// build the cycle
build cycle order by putting 1 in a vector
until there is only 1 edge left in the solution collection
find an edge in the solution collection with the last vertex in the vector
add the other vertex to the end of the vector
remove the edge from solution collection
return the cycle
```

Before you implement this algorithm, you may want to spend a few minutes thinking about the data structures and algorithms you will need to answer the questions in the two `if` statements. This is where all of the real work happens.

**Requirements**

- All requirements from part 1 are still required, except the cycle found must be “good enough.”
- Program must read content from `std::cin`.
- Program must write results to `std::cout`.
- Input format is described in part 1.
- Output format is described in part 1.
- Program must run on any sample graph in no more than 60 CPU seconds.
- Graphs may have up to 1000 vertices.
- Program must obtain a cycle quality at least 0.80 on all sample graphs.
- Program executable file must be called `TSP-GOOD`.

**Timing Files**

- `zip`
- `tgz`

**Show Off Your Work**

To receive credit for this assignment, you must upload the source code (`.h` and `.cpp` files) and the `Makefile` to the Canvas submission system.

Additionally, the program must build and run. Any incorrect performance or memory errors will be counted against the assignment score. (Hint: Use `valgrind` or other memory checkers.)
against the assignment score. (Hint: Use valgrind or other memory checkers.)