Lab5

Lab5 Submission link

Main TAs for this lab

Lab5 part I: Yuntao, Xuan

Lab5 part II (Regex) : Rishabh, Suofei, Divya

Lab5

EE 599 Lab5
Spring 2019 Nazarian **120+10 Points**

Assigned: Wednesday Feb. 6, 2019

~~**Due: Wednesday Feb. 13 at 11:59pm**~~

~~**Late submissions will be accepted for two days after the deadline with a maximum penalty of 15% per day. For each day, submissions between 12am and 1am: 2% penalty; between 1 and 2am: 4%; 2-3am: 8%; and after 3am: 15%. No submissions accepted after Friday Feb. 15 at 4am.**~~

After getting feedback from many of you, it seems at least some are falling behind, and some also have other course deadlines. I understand a considerable part of class are very good with coding and finish up the labs fairly quickly. If you are among those, please use the extra time to work more on your final project of 599.

Extended Deadline: Sunday Feb. 17 at 11:59pm. Submissions by original deadline (Wed. Feb. 13 at 11:59pm) will receive 15% of lab5 grade as extra credit add to the lab5 grade. No penalty for submissions between original and extended times. No late submissions after Sunday at 11:59pm.

Check the general Lab Guidelines.

What You Will Practice

In Part I of this lab, you are going to get familiar with an optimization algorithm – simulated annealing.

In Part II, you will practice unix/regex/shell/python scripting skill.

Part I - Simulated Annealing

Last Update: 2/6/19 @ 7:32pm

Part I - Simulated Annealing for Traveling Salesman Problem (80pts)

1. Background

Finding an optimal solution for certain optimization problems can be an incredibly difficult task, often practically impossible, as we would need to search through an enormous number of possible candidate solutions to find the optimal one. Even with modern computing power, that could take too long. In this case because we can't realistically expect to find the optimal one within a sensible length of time, we must settle for something that's close enough. An example optimization problem with many possible solutions is the traveling salesman problem (TSP). In order to find a solution to a problem such as the TSP we need to use a methodology (typically referred to as a heuristic) that's able to find in a reasonable amount of time, a solution which is good enough.

2. Problem Description

The TSP description as follows: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route for a traveling salesman to visit each city and return to the origin city?" The optional solution to the TSP is therefore the most efficient route (or tour) through a set of known cities (i.e., given locations). Each location should be passed through only once, and the route should loop so that it ends at the starting location. We focus on the symmetric TSP, in which the distance between any two cities is the same in both directions.
Theoretically, one could use what is referred to as "brute force search" and consider each possible tour through the given set of cities to determine which tour is the shortest. This results in runtime complexity of O(n!).

This is easy for very small sets, e.g., smaller than 5 cities, however, TSP is a NP-hard problem and no polynomial time solution is known for TSP. This means as the size of the problem increases, its runtime complexity grows exponentially. Consider the small set of points {A, B, C, D, E}. The number of possible tours beginning and ending with point and traversing every other point exactly once is equal to the number of permutations of the set {B, C, D, E}, which is 4! = 24. However for a symmetric TSP, that would reduce to half, i.e., 12. We can generalize the number of possible tours to consider for any set of size to be $\frac{(n-1)!}{2}$

. If we want to find the shortest route for 20 locations, we would have to evaluate 121645100408832000 different routes! Even with modern computing power this is terribly impractical, and for even bigger problems, it's close to impossible. Obviously, if one wants to solve the TSP for any large , alternative (heuristic) methods to brute-force search must be utilized.

3. Simulated Annealing Introduction

The technique to apply to TSP should provide solutions to large sets, that are considered sufficiently good, given the fairly short amount of time it would take to find that solution.

Many heuristics have been developed that can provide a close approximation to the true solution of occurrences of the TSP in relatively short amounts of time. Heuristics therefore are utilized to quickly obtain a near-optimal solution that is considered close enough to the global optimum solution to be useful. In our lab, one of these heuristic methods will be used, namely, simulated annealing.

The simulated annealing optimization method was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In simulated annealing we keep a temperature variable to simulate this heating process. We initially set it high and then allow it to slowly 'cool' as the method runs. While this temperature variable is high, the method accepts with higher probabilities, to accept solutions that are worse than our current solution. This gives the method the ability to jump out of any local optimum, it may get trapped, in early steps of execution. As the temperature is reduced, the probability of accepting worse solutions decreases, therefore letting the method gradually focus in on an area of the search space in which hopefully, a solution close to optimum can be found. This gradual 'cooling' process is what makes the simulated annealing optimization method remarkably effective at finding satisfactory solution for scenarios with large inputs sizes with numerous local optimum cases.

Comparison with Hill Climbing

The Hill Climbing (HC) optimization technique, is a type of a local search which would accept neighboring solutions that are better than the current solution. When HC can't find any better neighbors, it stops. Although HC can be surprisingly effective at finding a good solution when the search time is limited, e.g., in real-time applications, one of the major drawbacks to this technique is that it is very likely to lead to a local optimum.

Simulated annealing works slightly differently and will occasionally accept worse solutions. This is one of the key concepts of simulated annealing that makes it effective. With simulated annealing, a random tour is chosen, and its permutation is slightly modified by switching the order of as few as two points to obtain a new tour. If this new tour's distance is shorter than the original tour, it becomes the new tour of interest. Else, if this new tour has a greater distance than the original tour, there exists some probability that this new tour is adopted anyways. At each step along the way, this probability decreases to 0 until a final solution is settled upon.

Simulated annealing slowly and increasingly restricts the freedom of the solution search until it only approves moves toward superior tours. The probability of accepting an inferior tour is defined by an acceptance probability function. This function is dependent on both the change in tour distance, as well as a time-dependent decreasing parameter appropriately referred to as the Temperature. The way that the Temperature changes is referred to as the Cooling Schedule. There is no single way to define the probability acceptance function or the Cooling Schedule in such a way that will work well for any problem. In this lab, we simply utilize a commonly used acceptance probability function (known as Kirkpatrick's model).

4. The Algorithm Definitions:

State (s): a particular tour through the set of given cities or points

Neighbor State (s′): state obtained by randomly switching the order of two cities

Cost Function (C): determines the total cost (distance) of a state

Relative Change in Cost (δ

): the relative change in cost between s and s′

Cooling Constant (β

): the rate at which the Temperature is lowered each time a new solution is found

Acceptance Probability Function (P): determines the probability of moving to a more costly state

n: number of cities (or locations or points)

T_0

: initial Temperature

T_K

: the Temperature at the k^th instance of accepting a new solution state, meaning $T_{k+1} = \beta$ T_k, where is the Cooling constant, between 0 and 1.

$$P(\delta, T_K) = \begin{cases} e^{(\frac{-\delta}{T_K})} & , \delta > 0 \quad for\ T_K > 0 \\ 1 & , \delta \leq 0 \end{cases}$$

Note that for δ

>0, for any given T, P is greater for smaller values of δ. In other words, a state s′ that slightly costs more than s is more likely to be accepted over a state s′ that costs significantly higher than s. Also, as T decreases, P also decreases. For $\delta > 0$, when $T \rightarrow 0$, $P \rightarrow 0$

Pseudocode:

```
1) Choose a random state s and define $T_0$ and $\beta$

2) Create a new state s′ by randomly swapping two cities in s
3) Compute
   $\delta = \frac{C(s') - C(s)}{C(s)}$

    a. If $\delta \le 0$, then s= s'

    b. If $\delta > 0$, then assign s= s′ with probability $P(\delta, T_k)$ (Such as if P > random (0, 1), assign)

       Compute $T_{k+1} = \beta T_k$ and increment k.
4) Repeat steps 2 and 3 keeping track of the best solution until stopping conditions are met
```

Stopping Conditions: The stopping conditions are quite important in simulated annealing. If the process is stopped too early, the approximation won't be as close to the global optimum as desired, and if not stopped soon enough, the wasted time and calculations offer little to no benefit. In this lab, we will specify a minimum temperature. If the value is reached, the search will stop. We will also periodically check to ensure that the cost of the most optimal state found so far is changing. If it doesn't change within a particular period of iterations, the search will be stopped to limit unproductive work.

5. Tasks:

1. (30pts) Implement the Simulated Annealing method for TSP in C++. Set parameters as follows: T_0= 100, β = 0.95. Set the stopping conditions as: minimum temperature = $1e^{-6}$, and a solution change check after every 50,000 iterations (If the most optimal state does not change after 50,000 iterations, stop the search).
2. (20pts) Implement Hill Climbing optimization (greedy algorithm) for TSP in C++.
3. (20pts) Explore the effects of changing the Cooling Schedule. Specify the initial temperature $T_0$and the rule for decrementing the temperature.

   A. Specify T_0: Kirkpatrick [1] suggested that a suitable initial temperature T_0 is one that results in an average increase acceptance probability P of about 0.8. The value of T_0 will clearly depend on and, hence, be problem-specific. It can be estimated by conducting an initial search in which all increases are accepted and calculating the average objective increase observed δ. T_0 is then given by:

$$T_0 = \frac{-\delta}{\ln P}$$

, where P = 0.8. (1)

Conduct a search in all accepted increases (when $\delta > 0$ and accepted the worse solution), and then calculate the average value of δ. Calculate the optimized T_0 using equation (1).

B. Specify the cooling constant β: in task 1, we use exponential cooling scheme, the most common temperature decrement rule, was first proposed Kirkpatrick et al., i.e., $T_{k+1} = \beta T_k$, with $\beta = 0.95$. β is constant close to, but smaller than 1. Cooling constant affects the run time and the final optimal solution.

Using the T_0 value obtained in Task 2A, change the value for cooling constant β, where $0.9 < \beta < 1$. Compare the optimal results of the distance and the total number of iterations of program for different β. Understand the effect of β on run time and final optimal solution. For each β, run program at least 5 times. Report the maximum final optimal distance obtained by your program, minimum final optimal distance, and the average final optimal distance of the 5 results, and the corresponding total number of iterations in each time in a table. Report the best cooling constant β in terms of average optimal distance from your table. Show your results in the Readme.pdf file.

4. (10pts) Explore other acceptance probability functions (Extra credit)

Use a different acceptance probability function and implement the simulated annealing algorithm again. You can find other acceptance probability functions proposed in research papers of simulated annealing. Compare the effectiveness of your acceptance probability function with the Kirkpatrick's model in terms of optimal results and number of iterations.

Input File Format

You will be given a few text files named "input.txt" one at a time. This file contains two pieces of information: (i) *total number*of cities (ii) *the x,y coordinates* of each city, separated by space. The format of input files is defined: The first line is the total number of the cities. Starting from second line, each line represents a city, and it consists the list number and the X Y coordinates of each city. Here is an example of "input.txt". There are 10 cities in total. Line 2 to Line 11 list the city 1 to city 10.

In "input.txt" file:

```
10
1 5298 3001
2 814 6223
3 4808 5675
4 3716 8648
5 9945 8482
6 3983 1263
7 3464 5035
8 9160 8453
9 6312 3323
10 115 5141
```

C++ Data Structure

The following classes, data and function members are required:

Class City

This class represents the information for a city, including the following data attributes:

CityName (a C++ string is recommended): for simplicity assume the name of each city is "CityNoX", where X is the city number. E.g., city 2 has a name attributed that is initialized to "CityNo2". The initialization should occur using a constructor method.

X & Y: stores the coordinates of the cities.

Class SimulatedAnnealing

City *cities;

Class Greedy

City *cities;

We need constructors and destructors for both Greedy and SimulatedAnnealing classes. We will specify shortly, what exactly the constructors and destructors should do beyond the default.

See the attachment EE599_Lab5_1234567890_task1.cpp

Output File

After obtaining the final route (tour) solution, you should write your results to two text files with name "**Lab5_partI_task1_output.txt**" and " **Lab5_partI_task2_output.txt** " . You should output the final solution's distance in the first line. Then output the route in the following 2 – 11 lines. You only need to write the city number in each line. You should write your result to files instead of printing in terminal. For example, the desired **task1_output** for above example will be as follows:

```
Final Distance of SA Method: 30107
2
10
6
1
9
5
8
4
3
7
2
```

1. Notes

- X>=1, Y>=1. The value of X, Y, will be at most 10000.
- Your code should be written in only **ONE** file named "**EE599_Lab5_ _task#.cpp** " where you shouldimplement all the classes and the main driver function for each task. Please replace with your own 10 digits USC-ID, the name of the file would be something like: EE599_Lab5_1234567890_task1.cpp
- An "input.txt" and correct "output_golden.txt" files for task 1 are included in the starter repo for your reference. You can use them to test your code, but there will be other inputs for grading your code.

Summary

- You are allowed to use stdlib, math, time library in this lab. You can use standard libraries, such as iostream, fstream, string, etc. For other unsure libraries, please ask in discussion forum first before you use it.
- You have to follow the exact formats and syntax for generating the required results. Even adding an extra white line in the beginning of your output file or an extra space, etc., will result in losing the whole score of this lab.
- You should also write a README file. You can write a .txt file or .md file. A README file typically contains information of author's name and email, a brief program summary, references and instructions. Besides, include any information that you think the course staff, especially the grader should know while grading your assignment: references, any non-working part, any concerns, etc.
    1. Any non-working part should be clearly stated.
    2. The citations should be done carefully and clearly, e.g.: "to write my code, lines 27 to 65, I used the Diijkstra's shortest path algorithm c++ code from the following website: www.SampleWebsite.com/ ..."

3. The Readme file content of labs cannot be hand-written and should only be typed.

Submission

Submit your code files and a README file using GitHub. Please replace with your own 10 digits USC-ID, the name of the file would be something like: EE599_Lab5_1234567890_task1.cpp

Part II - Regex

Part II – Regular Expression (40pts)

A regular expression is a sequence of characters that implement a search pattern. Regular expressions are very commonly used in software management and analysis. You can think of regular expressions as wildcards. An example wildcard notation is *.txt where * is a wildcard that represents any string of characters. In this part of Lab5, you will practice some basics unix commands using regular expressions that are very helpful to your future software analysis and management tasks.

Problem 1. Develop a unix command for a binary string that represents an unsigned value that is divisible by 3.

Example:

Should be accepted:

```
11
110
1001
1100
10101
```

Should be rejected:

```
1
100
111
1000100
10010010
```

Note: print result in output1.txt

Problem 2. You are in the market to buy a red pick-up truck, and you wish to develop an automated web searching program (a spider) to search daily through various online newsgroups and classified ad websites to find text containing the word red and the phrase pick-up truck close to each other, followed by a price. Specifically, you should write unix commands that match the words red and the phrase (pickup/pick-up/pick up) truck separated by at most two other words in between. The pick-up truck phrase could appear before or after the word red. After the words red and the phrase pick-up truck, the text should also contain a price.

Example:

Should be accepted:

```
red pickup truck $1,234.56
red pick-up truck $5000
blah blah red toyota 1993 pick-up truck blah blah $5000 blah
pickup truck red $5000
desperate: red 1993 toyota pickup truck for sale. $2,000 o.b.o.
```

Should be rejected:

```
red truck $5000
$5000 red pickup truck
blue pickup truck $5000
red car $5000
red toyota 1993 pick-up truck
1993 toyota automatic pick-up truck $5000
fred's pick-up truck sold for $50
```

Note: print result in output2.txt

Problem 3.Match Dates

Write a unix command to match a date in the format "**dd{separator}MMM{separator}yyyy**". Every valid date has the following characteristics:

- Always starts with **two digits**, followed by a **separator**
- After that, it has **one uppercase** and **two lowercase** letters (e.g. `Jan`, `Mar`).
- After that, it has a **separator**and **exactly 4 digits**(for the year).
- The separator could be either of three things: a period ("**.**"), a hyphen ("**-**")or a forward slash ("**/**")
- The separator needs to be **the same**for the whole date (e.g. 13**.**03**.**2016 is valid, 13**.**03**/**2016 is **NOT**).

Example:

Should be accepted:

```
13/Jul/1928
10-Nov-1934
25.Dec.1937
```

Should be rejected:

```
01/Jan-1951
23/sept/1973
1/Feb/2016
```

Note: print result in output3.txt

Problem 4. Imagine this is your final semester and you will leave the campus for an internship. So you want to choose an EE course with DEN session. You have 3 units remained and you can only choose a course that is500-level or above (ie. Course code greater than EE500)

USC classes list can be found via https://classes.usc.edu/term-20191/classes/ee/The html page source for this page was saved in the file "ee_classes.html".

Write a Unix command query to find all the 500-level or abovecourses that have a DEN session, and at least 3 units. Ignore the course that has a range of units (ie. Only 3 or 4 units are possible). Sort the course by ascending units. For the course with same units, sort them by course code (ascending). Print the result in output4.txt, in the following format:

```
EE 512: Stochastic Processes (3.0 units)
...
```

Hint:

HTML block for each course start from `<div class="course-info expandable"`

Submission

Put all your command in one shell scipt ***Lab5_partII_<YourID>.sh***. Submit it with a README file using GitHub.

Part I - input and output files

Attached Files: Lab5_partI_input.txt (131 B)
Lab5_partI_task1_output.txt (70 B)
Lab5_partI_task2_output.txt (74 B)
EE599_lab5_1234567890_task1.cpp (906 B)

Part II - input and output files

Attached Files: Lab5_partII_Q1_input.txt (674 B)
Lab5_partII_Q1_golden_output.txt (227 B)
Lab5_partII_Q2_input.txt (619 B)
Lab5_partII_Q3_input.txt (190 B)
ee_classes.html (371.805 KB)
Lab5_partII_Q4_golden_output.txt (1.14 KB)
Lab5_partII_script.txt (758 B)

Lab5_partII_Q2_golden_output.txt (266 B)
Lab5_partII_Q3_golden_output.txt (96 B)

Common Problems

1 The output of TSP should around 80000 if you implemented it correctly with the parameters we provided to you. For huge output, please check you function SimulatedAnnealing() especially where you accept worse values.

2 No random process in Greedy.

3 Before you use use rand(), make sure you add srand() to initialize it.