15-418/618, Spring 2021
Assignment 3
Parallel Wandering Salesperson

Assigned:	Mon. March 1st
Due:	Wed. March 17th, 11:59 pm
Last day to handin:	Sat. March 20th, 11:59 pm

Overview

In this assignment you will be solving the Wandering Salesman Problem (WSP) using the Branch-and-Bound technique. WSP is representative of combinatorial optimization problems. The Branch-and-Bound technique is an exhaustive evaluation technique that tries to make use of knowledge of the underlying problem to reduce the amount of computation.

You are permitted to work in groups of 2 people to solve the problems for this assignment (hand in one assignment per group.) You will be submitting to both Autolab and Gradescope.

Before you begin, please take the time to review the course policy on academic integrity at:

http://www.cs.cmu.edu/418/academicintegrity.html

The Wandering Salesman Problem

The object of WSP is to find the shortest route for a traveling salesman so that the salesman visits every one of a set of cities exactly once. (Note: the salesman doesn't return home. This is the difference between the wandering salesman problem and the traveling salesman problem.) This is a hard combinatorial optimization problem since for N cities there are at most N! possible routes (we assume that the cities are fully connected).

The input to the problem is given in the form of a matrix. An element of the matrix, `d[i][j]` gives the distance between city i and city j. The input to your program should be a file organized as follows:

```
N
d[1][2]
d[1][3] d[2][3]
d[1][4] d[2][4] d[3][4]
.
.
```

.
`d[1][N] d[2][N] d[3][N] ... d[N-1][N]`

where `N` is the number of cities, and `d[i][j]` is an integer giving the distance between cities i and j. The output from the program should be an ordered list of cities (numbers between 1 and N). Clearly, there are 2 equivalent permutations - either one is acceptable.

1 Branch-and-Bound Solutions to Combinatorial Optimization Problems

First, consider a simple exhaustive evaluation as a way of solving the WSP. One way you might think about evaluating every possible route is to construct a tree that describes all of the possible routes from the first city, as shown in Figure 1 for a four city example.

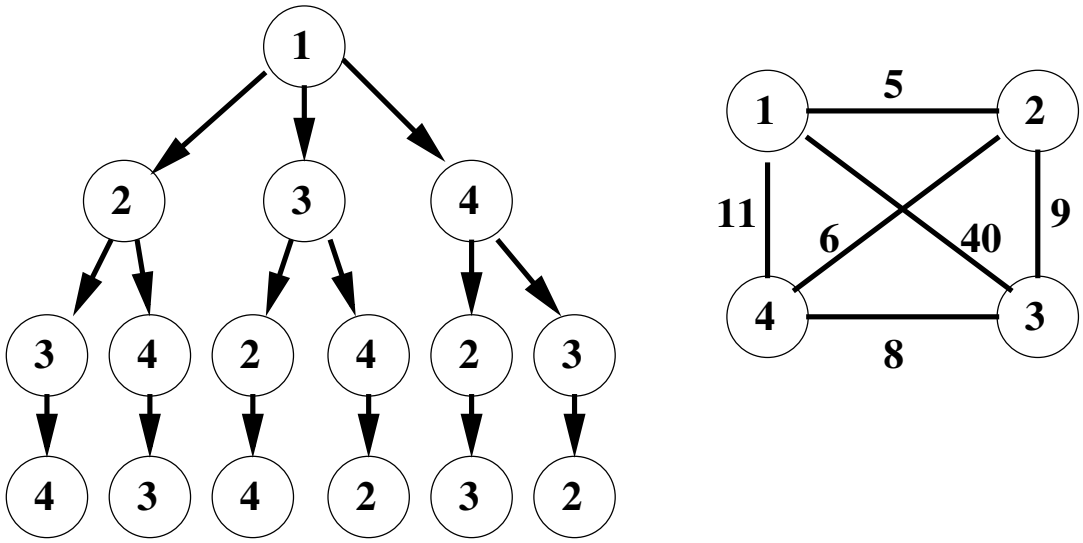

Figure 1: *A WSP example*

For this WSP, `d[1][2]` = 5, `d[1][3]` = 40, `d[1][4]` = 11, `d[2][3]` = 9, `d[2][4]` = 6, `d[3][4]` = 8. All of the possible routes can be found by traveling from the root to a different leaf node three times, once for each unique path. For example, by taking the middle branch from the root node and the right branch after that, the route $1 \rightarrow 3 \rightarrow 4 \rightarrow 2$ is produced, and has a distance of 40+8+6 = 54. There are six unique root to leaf paths in this tree. Each possible route is represented twice, once in each direction.

A simple exhaustive evaluation of WSP for this example would then be to follow the six root to leaf paths and determine the total distance for each path by adding up the distances indicated by each edge in the tree. The route with the smallest distance is then chosen.

A better way to traverse each tree is to do it recursively. Here the summation of the earlier parts of each route is not repeated every time that route portion is reused in several routes. The Branch-and-Bound approach uses this type of problem formulation, but with some added intelligence. It uses more knowledge of the problem to prune the tree as much as possible so that less evaluation is necessary. The basic approach works as follows:

1. Evaluate one route of the tree in its entirety, (say $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$) and determine the distance of

that path. Call this distance the current "bound" of the problem. The bound for this path in the above tree is $5 + 9 + 8 = 22$.

2. Next, suppose that a second path is partially evaluated, say path $1 \rightarrow 3$, and the partial distance, 40, is already greater than the *bound*. If that is the case, then there is no need to complete the traversal of any part of the tree from there on, because all of those possible routes (in this case there are two) must have a distance greater than the bound. In this way the tree is *pruned* and therefore does not have to be entirely traversed.

3. Whenever any route is discovered that has a better distance than the current bound, then the bound is updated to this new value.

The Branch-and-Bound approach always remembers the best path it has found so far, and uses that to prevent search down parts of the tree that couldn't possibly produce better routes. You can see that for larger trees, this could result in the removal of many possible evaluations.

2 The Assignment

The assignment is to write a parallel Branch-and-Bound program for the WSP, using OpenMP. The objective is to obtain the best speedup possible. Start by cloning the below code. We recommend creating a private repo for you and your partner to make collaboration easier.

git clone https://github.com/cmu15418-s21/asst3-s21

The starter code is contained in `wsp.c`. Run `make` to compile your program. You can run your program using the following command.

```
./wsp -p [numProcessors] [inputFile]
```

Where numProcessors is the number of openMP cores and inputFile is reference to a dist file containing city distances. An example of running the starter code is as such:

```
$ ./wsp -p 1 city/dist10

============ Time ============
Time: 0.013 ms (0.000 s)
========== Solution ==========
Cost: 0
Path: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9
```

The program computes the execution time of `wsp-start()` responsible for all the logic relevant to computing an optimal path. Your first task is to implement the WSP algorithm sequentially. You may include global variables, but **all logic and allocations must run within wsp-start()** in order for its time to be correctly recorded.

We want you to focus on algorithm design and analysis rather than trying to hit a target benchmark. Good speedups are meaningless unless you have the proper justification to accompany it.

As such, we do not want you aiming to hit a target speedup, as your analysis to the problem will be worth more than your numeric results. Try to see how fast you can get your program to run, and justify the limitations you meet with your program.

We've provided you a script `verify.py` that you can use to check your program against some of the smaller sequential `wsp-ref` solutions. If you'd like to verify any of the larger distance files individually, you may do so by running the `wsp-ref` program. Please note that `wsp-ref` is sequential and does not take the number of cores as an argument.

```
$ ./wsp-ref city/dist10

============ Time ============
Time: 181.825 ms (0.182 s)
========== Solution ==========
Cost: 189
Path: 6 -> 1 -> 2 -> 7 -> 4 -> 5 -> 8 -> 3 -> 0 -> 9
```

Keep in mind that multiple paths of the optimal solution may exist. For this reason, we are only checking that your distance cost matches that of the reference program.

We've also provided you with a program to help create additional distance files. You can make and run `./distgen` in the `city/` directory to create more cities. Please do not overwrite the distance files we've provided you. An example of running `./distgen` is provided below.

```
$ ./distgen
10           // number of cities
15418        // random seed
customCity   // output file name
City  0: ( 54, 58)
City  1: ( 57,  3)
City  2: ( 80,  3)
City  3: ( 74, 40)
City  4: ( 72, 51)
City  5: ( 22, 62)
City  6: ( 80, 30)
City  7: ( 97, 73)
City  8: ( 34, 43)
City  9: ( 45, 10)
```

3 Write-Up

(10 pts) A brief (roughly one or two pages) description of how your program works. Describe the general program flow and all significant data structures. You should first explain your sequential solution, and then explain any modifications you made to your code when creating your parallel implementation.

(5 pts) The solutions (both the cost and path) to dist16, dist17, and dist18 problems. You should verify your results with the sequential reference program we provide. We will be running your programs to verify the correctness on these cases.

(10 pts) Execution time and speedup for 1, 2, 4, 8, 16, 24, and 32 cores on `Latedays` (instructions on how to do so are included below) on dist16, dist17, and dist18. For reference, our solution to dist15 takes approx. 30s with p=1 and 6s with p=8 on Latedays.

(25 pts) Discuss the results you expected and explain the reasons for any non-ideal behavior you observe. In particular, if you don't get perfect speedup, explain why. Is it possible to get better than perfect speedup? Give measurements to back up your explanations.

4 Hand-in

You will submit your code via Autolab and your report via Gradescope.

- **If you are working with a partner, form a group on Autolab.** Do this before submitting your assignment. One submission per group is sufficient.
- Make sure all of your code is compilable and runnable. We should be able to simply run make, then execute your programs on AFS/Latedays without manual intervention.
- Run the command **tar -czvf handin.tar wsp.c Makefile** to create a handin.tar for you to upload to Autolab.
- Upload your report as file `report.pdf` to Gradescope, one submission per team, and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the *add group members* button on the top right of your submission.

5 Running on the Latedays Cluster

The Latedays cluster contains 18 machines (1 head node plus 17 worker nodes). Each machine features:

- Two, six-core Xeon e5-2620 v3 processors (2.4 GHz, 15MB L3 cache, hyper-threading, AVX2 instruction support).
- 16 GB RAM (60 GB/sec of BW)

You can login to the Latedays head node latedays.andrew.cmu.edu via your Andrew login. You can edit and compile your code on the head node, and then run jobs on the cluster's worker nodes using a batch queue. You have a home directory on Latedays that is not your Andrew home directory. (You have a 2GB quota.) However, your Andrew home directory is mounted as /AFS.

Do not attempt to submit jobs from your AFS directory, since that directory is not mounted when your job runs on the worker nodes of the cluster. (It is only mounted and accessible on the head node.) Instead, copy your source code over to a subdirectory you have set up in your Latedays directory and recompile it.

The program **submitjob.py** is used to generate and submit command files to the job queue. It is invoked as follows:

```
linux> ./submitjob.py -h
Usage: ./submitjob.py [-h] [-J] [-s NAME] [-a ARGS] [-r ROOT] [-d DIGITS]
-h          Print this message
```

```
-J.         Don't submit job (just generate command file)
-s NAME     Specify command file name (output file)
-a ARGS     Arguments for wsp (can be quoted string)
-r ROOT     Specify root name of output file
-d DIGITS   Specify number of randomly generated digits in command and output file names
```

Here's a brief description of the options:

-J Generate the command file, but do not submit it.

-s NAME - Specify the name of the command file. The default will name is of the form latedays-DDDD.sh, where DDDD is a sequence of random digits.

-a ARGS - Provide arguments(s) for ./wsp. Typically, ARGS is a quoted string. For example, specifying -a '-p 8 city/dist10' will cause the wsp program to be done with 8 cores on dist10.

-r ROOT - Specify the prefix of the summary output file name. The output file is generally of the form ROOT-DDDD.out, where DDDD is a sequence of random digits.

-d DIGITS - Specify the number of random digits to include in the command and output file names.

Running submitjob.py on Latedays expects the wsp program and the city folder to be in the same directory. Below is an example of running the command.

```
$ ls
wsp  submitjob.py  city/
$ ./submitjob.py -s test -a "-p 1 input/dist15"
Generated script test-3724.sh
368497.latedays.andrew.cmu.edu
$ ls
wsp  submitjob.py  city/  test-3724.sh  test-3724.sh.e368497  test-3724.sh.o368497
```

test-3724.sh contains the executable script and test-3724.sh.o368497 contains the output for this specific instance. You'll be using Latedays to time your program.

The inclusion of random digits in the file names provides a way to avoid naming collisions. For example, you can invoke submitjob.py n times with the same arguments, submitting n jobs, each with distinct file names. After a successful submission, the program will echo the ID of your job. For example, if your job was given the number 337 by the job queue system, the job would have the ID 337.latedays.andrew.cmu.edu. After you submit a job, you can check the status of the queue via one of the following commands:

```
$ showq
$ qstat
```

When your job is complete, log files from standard output and standard error will be placed in your working directory. If the command file was latedays-1234.sh, then the generated files will be latedays-1234.sh.o337 and latedays-1234.sh.e337 (substituting your job number for 337, of course). In addition, a summary of the results will be written to the output file.

Looking at the command file, you will see that the maximum wall clock time for the job is limited to 30 minutes. This means that the job scheduler will cut your program off after 30 minutes without generating any output.

Good luck!