

15-418/618, Fall 2020
Assignment 3
Parallel Wandering Salesperson

Assigned:	Mon. Sept 28th
Due:	Fri. Oct 16th, 11:59 pm
Last day to handin:	Mon. Oct 19th, 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]
.
.
```

$d[1][N] \ d[2][N] \ d[3][N] \ \dots \ d[N-1][N]$

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.

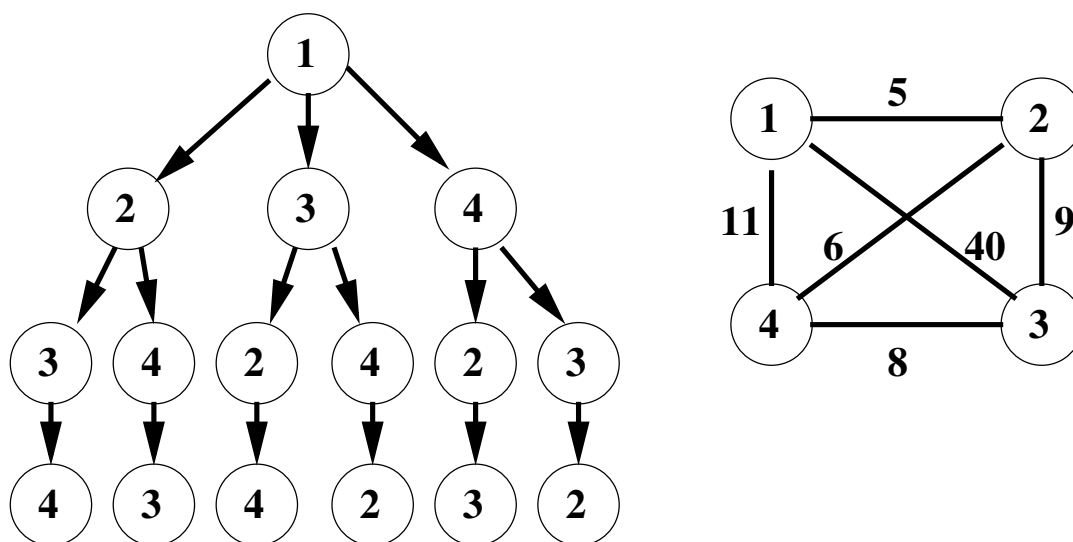


Figure 1: A WSP example

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 and then back to the root. 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 and then adding the distance back to the root. 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. You should use the input format described above.

git clone <https://github.com/cmu15418-f20/asst3-f20.git>

You’ll quickly find that there is no starter code provided. You are responsible for creating the following:

- `wsp.c` - contains your implementation for parallel BnB WSP
- `Makefile` - compiles `wsp.c` to `wsp`
- `writeup.pdf` - your report to the assignment

Your WSP program should be of the following form:

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

Where `numProcessors` is the number of openMP cores and `inputFile` is reference to a dist file containing city distances (see Section 3). Your program must output the optimal path as well as execution time, although you are allowed to output anything else you think may help you in producing your report.

Your report should include the following items:

1. A brief (roughly one or two pages) description of how your program works. Describe the general program flow and all significant data structures.
2. The solutions to `dist16`, `dist17`, and `dist18` problems. For debugging purposes, you may want to use some of the smaller input files included in the same directory. The city locations corresponding the distance files are provided in the corresponding ‘city’ files. We will be testing other cities for correctness as well.

3. Execution time and speedup for 1, 2, 4, 8, 16, 24, and 32 processors on **Latedays** (instructions on how to do so are included in the Appendix) on dist16, dist17, and dist18. For reference, our solution to dist15 takes approx. 30s with p=1 and 6s with p=8 on Latedays.
4. 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.

If the execution time for your program takes more than a few minutes, double-check your program and algorithm. Make sure your programs run on a uniprocessor before trying to run them in parallel. Also, debug your programs using smaller numbers of cities and small numbers of processors before trying larger runs. You can use `./distgen` to create more cities.

```
linux> ./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)
```

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. Try to see how fast you can get your program to run, and justify the limitations you meet with your program.

3 Hand-in

You will submit your code via Autolab and your report via Gradescope. The relevant websites are:

<https://autolab.andrew.cmu.edu/courses/15418-f20/>

<https://www.gradescope.com/courses/156025>

- **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 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.

4 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 benchmark.py (can be quoted string)
-r ROOT     Specify root name of benchmark output file
-d DIGITS   Specify number of randomly generated digits in command and
            benchmark 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 argument(s) for `benchmark.py`. Typically, `ARGS` is a quoted string. For example, specifying `-a '-I -t 6'` will cause the benchmarking to be done with instrumentation enabled and with 6 threads rather than 12.

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

./submitjob.py expects the **wsp** program to be in the same directory. Below is an example of running the command.

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

```
linux> showq
linux> 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!