EECS 281 – Fall 2021
Programming Project 4
Zookeeper

Due Thursday, December 9 before midnight

Overview

You are a zookeeper caring for a large number of animals. Everyday, you (the zookeeper) should provide food and water for the animals. In order to minimize the expenses of the zoo, you are planning to devise a systematic way to feed the animals. Your plan is to first make paths to each animal cage to deliver food, and then to construct canals to provide fresh water to all animal cages. Your task is to identify the most efficient routes for providing food and water to the animals.

In part A we only try to make paths to the cages to feed the animals and our goals are to minimize the expenses of building paths and to find the most efficient routes. In parts B and C, we try to find the the most cost efficient routes for constructing water canals to the cages. In order to have water flow within the canals, the route should be a cycle.

To be clear: these scenarios are separate; the program will create a plan for one or the other, but not both in the same run (though you may find that the algorithms from one mode help with another mode).

Project Goals

- Understand and implement MST algorithms. Be able to determine whether Prim's or Kruskal's is more efficient for a particular scenario.
- Understand and implement a Branch and Bound algorithm. Develop a fast and effective bounding algorithm.
- Explore various heuristic approaches to achieving a nearly-optimal solution as fast as possible.
- Use gnuplot (or other tools) for visualization

Map Input

On startup, your program reads input from standard input (`cin`) describing the locations of the cages in the zoo. The zoo is mapped on a grid with wild animals area on the bottom left quadrant (see 'Path Rules'). You will be given a list of `M` cages in the zoo with associated integer coordinates (`x`, `y`). The cages are identified by integer indices which should correspond to the order in which they are read in (i.e. the first cage coordinate you read corresponds to cage location 0, the second cage coordinate to cage location 1, etc.). For parts B and C you always start at the location of the 0-th cage.

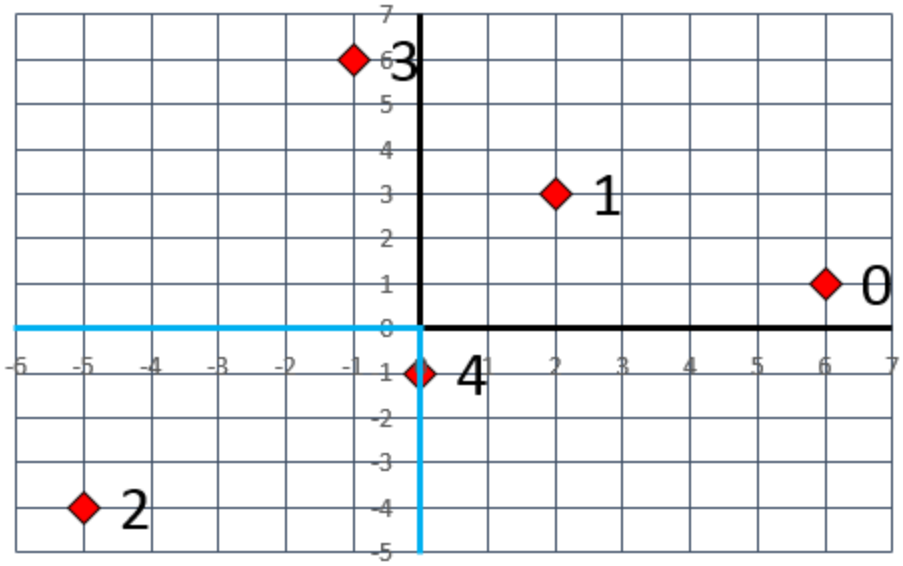
Version 11-12-21
Current version by Abraham Addisie, Holly Borla, and Rishin Doshi

Formally, the input will be formatted in this manner: The first line will contain a single number denoting the number of cages in the zoo. That will be followed by a list of integer x, y, coordinates in the form: `x y`. You may assume that the input will always be well-formed (it will always conform to this format and you do not need to error check). There may be additional blank lines at the end of the file (but nowhere else).

Sample:

```
5
6 1
2 3
-5 -4
-1 6
0 -1
```

The above sample can be visualized as in the figure below, where the numbers shown are the cages indices that starts at the 0-th cage and the blue line indicates the separating wall between the wild-animal area (for more details, see the 'Path Rules' section) and other animals' areas. The origin (0, 0) is included in the border.



There is more than one way to represent this configuration internally in your program, and this will affect your runtime. Choose your data structures wisely!

Path Rules

Distance

We represent the path[1] you take as a set of pairs of points or as a sequence of points. Your calculations should use **Euclidean distance**, and you should represent your distances as doubles.

[1] "Path," in this project, can be understood as a route between two locations.

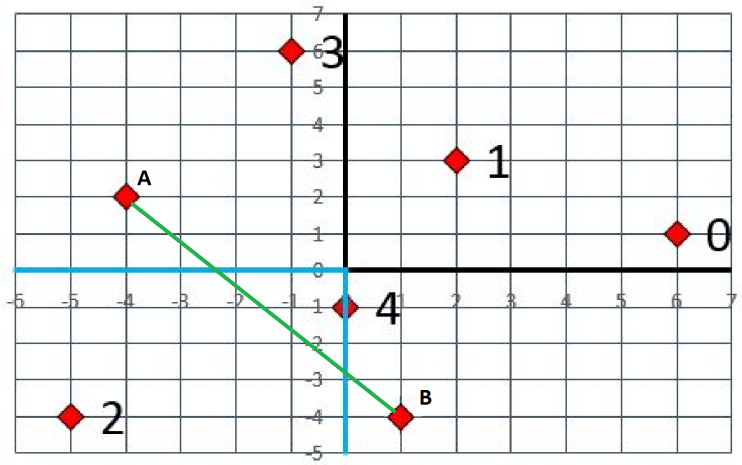
In Part A, you have to make routes in order to feed the animals. For the sake of extra caution, wild animals are in cages in the area surrounded by a wall. The only way to access this area is by passing through the doors of cages on the border.. You are allowed to move along the line segments that connect cages.

In Parts B and C, we try to find the most cost efficient routes for constructing water canals to the cages. For water to flow inside the canals, the route should be in the form of a cycle. In addition, the canal routes are underground, and can pass under the walls.

The Wild-animal area

The zoo contains a wild-animal area that is located in the **bottom left quadrant** of the map. This area is bounded by wall borders, which consist of the negative portions of the x and y axes, including the origin (0, 0). In Part A, you may only enter and exit the wild-animal area by passing through a cage located on the border. For example, you are not allowed to make a path directly from (-5, -4) to (6, 1). You must first walk from (-5, -4) to (0, -1), and then from (0, -1) to (6, 1).

For the sake of simplicity, assume two locations out of the wild-animal area that "cross" the wild-animal can be connected by a direct path. The example below shows two validly connected cages, A and B, with the mentioned situation.



In this example, there is a direct path from A to B, either through an elevated path or underground.

In parts B and C you may ignore the walls since the canals all pass underground.

Other

Important: For parts B and C, you always start at the 0-th cage location.

You are also **not** allowed to ‘wrap around the edges of the world’ (you **cannot** go above the top of the map to arrive at the bottom).

Command Line Input

Your program, `zoo`, should take the following case-sensitive command line options:

- `-m, --mode <MODE>`
  This command line option must be specified, if it is not, print a useful error message to standard error (cerr) and exit(1). `MODE` is a required argument. Set the program mode to `MODE`. `MODE` must be one of `MST`, `OPTTSP`, or `FASTTSP`. The `MODE` corresponds to the algorithm `zoo` runs (and therefore what it outputs).
- `-h, --help`
  Print a short description of this program and its arguments and exit(0).

Valid examples of how to execute the program:

`./zoo --mode MST`	(OK, but you must type the input by hand)
`./zoo -h < inputFile.txt`	(OK, -h happens before we realize there's no -m)
`./zoo -m OPTTSP < inputFile.txt`	(OK)
`./zoo -m BLAH`	(BAD mode following -m)

Remember that when we redirect input, it does not affect the command line. Redirecting input just sends the file contents to cin. You should not have to modify your code to allow this to work; the operating system will handle it.

We will not be specifically error-checking your command-line handling, however we expect that your program conforms with the default behavior of getopt_long. Incorrect command-line handling may lead to a variety of difficult-to-debug problems.

Algorithms

Your program should run **one and only one** of the following modes at runtime depending on the `mode` option for that particular program call. We divide it into 'parts' for your convenience, though *you may find that elements and algorithms of some parts can help with others*.

Part A – MST mode

Description

This mode will devise a path that connects every cage in the zoo together while minimizing the total distance of that path.

When the program is run in the `MST` mode, it should calculate and print out an MST connecting all of the cage locations. You may use any MST algorithm of your choosing to connect all the cages. *Hint*: Unless you want to implement both and compare, think about the nature of the graph (how many vertices and edges does it have?). You are free to adapt code from the lecture slides to fit this project, but you will want to carefully think about the data structures *necessary* to do each part (storing unnecessary data can slow you down). Your program must always generate one valid MST for each input.

Remember that in this part, you must respect the wild animals area at the bottom-left of the map. Your MST cannot connect a cage outside the wild animal area to a cage inside of it, without passing through a cage on the border.

Output Format (for Part A only)
For the `MST` mode, you should print the total weight of the MST you generate by itself on a line; this weight is the sum of the weights of all edges in your MST (in terms of Euclidean distance). You should then print all edges in the MST. All output should be printed to standard output (cout).

The output should be of the format:

```
weight
node node
node node
......
```

Where the nodes are the cage location numbers corresponding to the vertices of the MST and a pair of nodes on a given line of the output describes an edge in the MST from the first node to the second. To be clear, the weight should be formatted as a double (2 decimal point precision is enough - see Appendix A), and the node numbers should all be integer values when printed. For example, given the example input file above, your MST mode output might be:

```
19.02
0 1
2 4
1 3
1 4
```

You should also always print the pairs of vertices that describe an edge such that the index on the left has a smaller integer value than the index on the right. In other words:

```
1 2
```

is a possible valid edge of output, but

```
2 1
```

is not.

If it is not possible to construct an MST for all cages in the zoo (i.e. if there are some cages inside and outside of the wild-animal area with no cage on the border), your program should print the message “Cannot construct MST” to cerr and exit(1).

Parts B & C (FASTTSP & OPTTSP mode)

Description (for both Parts B and C)
In this mode, you will figure out how to devise an underground cyclic water canal that passes through all the cages. The canal will start at the 0th cage location, visit every other cage exactly once, and return to the 0th cage location. Your job will thus be to solve the TSP (Traveling Salesperson Problem) and choose canal paths to cage locations so as to minimize the total canal length. Euclidean (straight-line) distance is used here again to compute distances between cages. The wild-animal area cages do not impose restrictions in parts B and C since the canals run underground.

For FASTTSP mode, you do **not** need to produce an optimal TSP tour, but your solution should be close to optimal. Because your FASTTSP algorithm does not need to be perfectly optimal, we expect it to run much faster than finding a perfect optimal solution. Do some online searching for “TSP heuristics”. There are several types, some easier to implement, some with better path lengths, some both.

For OPTTSP mode, you must find an **optimal** solution to the TSP (the actual minimum Euclidean distance necessary). More on the differences between OPTTSP and FASTTSP modes will be discussed later in the spec.

For **both** methods:

You must start the tour from the 0-th cage location (your starting location).

The canal must pass through every cage location exactly once (there's no point in returning to a place already attached to the canal), and at the end of the tour, the canal **must return back to the 0-th location**.

The length of a tour is defined as the sum of all pairwise distances travelled - that is, the sum of the lengths of all of paths taken (using Euclidean distance).

Your program must print the indices of the locations in an order such that the length of this tour is as small as possible. More details about how to accomplish this are listed below.

Output Format (for both Parts B and C)
You should begin your output by printing the overall length of your tour on a line. On the next line, output the nodes in the order in which the canal passes. The initial node should be the starting location index and the last node should be the location number directly before returning back to the 0-th location. The nodes in your tour should be printed such that they are separated by a single space. There can be a space after the last location listed. All output should be printed to standard output (cout).

For example, if given the input file above, your program could produce:

```
31.64
0 4 2 3 1
```

or

```
31.64
0 1 3 2 4
```

Part B Only Description
In the case which we have a large number of cages in the zoo, finding an optimal method for connecting the cages may be too slow, and you may grow old and die before connecting every cage. You can use *heuristics* instead to find nearly-optimal tours. A heuristic is a problem-solving method (an algorithm) that can produce a good answer that is not necessarily the best answer. For example, you can skip a branch speculatively rather

than waiting to know for a fact that it can be skipped. There are many other simple heuristic techniques, such as starting with a random tour and trying to improve it by small changes.

You should be able to produce a solution to the Traveling Salesperson Problem (TSP) that is as close as possible to the optimal tour length **(though it does not need to be optimal)**. In the best case, your produced tour length will equal the optimal tour length.

You are allowed to use any combination of algorithms for this section that we have covered in class, including the MST algorithm you wrote for Part A.

You will need to be creative when designing your algorithms for this section. You are free to implement any other algorithm you choose, so long as it meets the time and memory constraints. **However**, you should not use any advanced algorithms or formulas (such as Simulated Annealing, Genetic Algorithms and Tabu search - they are too slow) that are significantly different from what has been covered in class. Instead, creatively combine the algorithms that you already know and come up with concise optimizations.

Part C Only Description

To find an optimal tour, you could start with the brute force method of exhaustive enumeration that evaluates every tour and picks a smallest tour. By structuring this enumeration in a clever way, you could determine that some branches of the search cannot lead to optimal solutions. For example, you could compute *lower bounds* on the length of any full tour that can be found in a given branch. If such a lower bound exceeds the cost of a full solution you have found previously, you can skip this branch as hopeless. If implemented correctly, such a **branch-and-bound** method should **always** produce an optimal solution. It will not scale as well as sorting or searching algorithms do for other problems, but it should be usable with a small number of locations. Clever optimizations (identifying hopeless branches of search early) can make your algorithm *a hundred times* faster. Drawing TSP tours on paper and solving small location configurations to optimality by hand should be very useful. **Remember that there is a tradeoff between the time it takes to run your bounding function and the number of branches that bound lets you prune.**

MAKE SURE that you use the version of `genPerms()` presented in either the "Project 4 Tutorial" or the "Backtracking BB TSP" lecture slides. The one presented earlier in the semester, in the "Stacks and Queues" lecture slides is much slower.

Given an input set of *N* locations defined by integer coordinates, your job is to produce an optimal tour using branch-and-bound algorithms. Your program should **always** produce the shortest possible tour as a solution, even if computing that solution is time-consuming. You will be given a 35-second cpu time limit to generate your solution. If your program does not produce a valid solution, it will fail the test case. Your solution will also be judged by time and space budgets as per previous projects.

Libraries and Restrictions

We highly encourage the use of the STL for this project, with the exception of several prohibited features. Do not use:

- The C++11 regular expressions library
- The STL thread/atomics libraries (which spoil runtime measurements)
- Shared or unique pointers.

- Other libraries (e.g., boost, pthreads, etc).

Testing and Debugging

Part of this project is to prepare several test files that will expose defects in buggy solutions - your own or someone else's. As this should be useful for testing and debugging your programs, we recommend that you **first** try to catch a few of our intentionally-buggy solutions with your test files, before completing your solution. The autograder will also tell you if one of your own test files exposes bugs in your solution.

Each test that you submit should consist of an input file. When we run your test files on one of intentionally-buggy project solutions, we compare the output to that of a correct project solution. If the outputs differ, the test file is said to expose that bug.

Test files should be named `test-n-MODE.txt` where 1 <= n <= 10. The autograder's buggy solutions will run your test files in the specified `MODE`. The mode must be `MST`, `FASTTSP`, or `OPTTSP`.

Your test files may have no more than 10 coordinates in any one file. You may submit up to 10 test files (though it is possible to get full credit with fewer test files). The tests the autograder runs on your solution are **NOT** limited to 10 coordinates in a file; your solution should not impose any size limits (as long as sufficient system memory is available).

Submitting to the Autograder

Do all of your work (with all needed source code files, as well as test files) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file:
- `// Project Identifier: 3E33912F8BAA7542FC4A1585D2DB6FE0312725B9`
- The `Makefile` must also have this identifier (in the first TODO block)
- You have deleted all .o files and your executable(s). Typing '`make clean`' shall accomplish this.
- Your makefile is called `Makefile`. Typing '`make -R -r`' builds your code without errors and generates an executable file called `zoo`. The command-line options -R and -r disable automatic build rules, which will not work on the autograder.
- Your `Makefile` specifies that you are compiling with the gcc optimization option `-O3`. This is extremely important for getting all of the performance points, as `-O3` can speed up code by an order of magnitude.
- Your test files are named `test-n-MODE.txt` and no other project file names begin with test. Up to 10 test files may be submitted. The "mode" portion of the filename must be `MST`, `OPTTSP` or `FASTTSP`.
- The total size of your program and test files does not exceed 2MB.
- You don't have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (i.e. the .git folder used by git source code management).
- Your code compiles and runs correctly using version 6.2.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support

anything other than GCC 6.2.0 running on Linux (students using other compilers and OS did observe incompatibilities). To compile with g++ version 6.2.0 on CAEN you **must** put the following at the top of your `Makefile`:

```
PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

Turn in all of the following files:

- All your .h and/or .cpp files for the project
- Your `Makefile`
- Your test files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission in one directory. In this directory, run

```
tar -czvf submit.tar.gz *.cpp *.h Makefile test-*.txt
```

This will prepare a suitable file in your working directory. Alternatively, the 281 `Makefile` has useful targets `fullsubmit` and `partialsubmit` that will do this for you. Use the command `make help` to find out what else it can do!

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. **When the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to four times per calendar day (more during the Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your best submission for your grade**. If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad `Makefile` or why you are segfaulting). Also be sure to check if the autograder shows that one of your own test files exposes a bug in your solution (at the bottom). Search the autograder results for the word "Hint" (without quote marks) for potentially useful information.

Grading

90 points -- Your grade will be derived from correctness and performance (runtime). This will be determined by the autograder. On this project we expect a much broader spread of runtimes than on previous projects. As with all projects, some test cases used for the final grading are likely to be different.

10 points -- Your program does not leak memory. Make sure to run your code under valgrind before each submit. This is also a good idea because it will let you know if you have undefined behavior (such as reading an uninitialized variable), which may cause your code to crash on the autograder.

10 points -- Student test file coverage (effectiveness at exposing buggy solutions).

Runtime Quality Tradeoffs

In this project there is no single correct answer (unlike previous projects). Accordingly, the grading of your problem will not be as simple as a 'diff', but will instead be a result of evaluating your output.

Particularly for Part B, we expect to see greater variation in student output. Part B asks you to solve a hard problem, and with the given time constraints, we don't actually expect your output to be optimal for most cases. The quality of your solutions may even vary from case to case. We want you to quickly produce solutions that are close to optimal. This inevitably creates tradeoffs between cycle length and runtime. In general, make sure that Part B runs in $O(n^2)$ time.

You may find it useful to implement more than one algorithm or heuristic that you use in Part B, and do some testing plus use the autograder to determine which works the best..

In addition to time and memory, your grade for Part B will be determined based on how close you are to a good solution, computed as a percentage. As with time and memory, having a cycle length that is over by a small amount is OK, but at some threshold you start losing points, and the further over you are, the more points you lose.

Hints and Advice

There's a project 4 tutorial video available. There's also a video that shows how a greedy nearest neighbor implementation looks while it's running (part B). It shows how this approach produces a suboptimal tour because it contains crossings, and how 2-OPT removes them.

It can be difficult to get this project correct without visualizing your MSTs and TSP tours. We have provided a visualization tool on the Autograder that you can use; follow this link: https://g281-2.eecs.umich.edu/p4viz/.

Running your code (on CAEN or locally) using `valgrind` can help you find and remove undefined (buggy) behavior and memory leaks from your code. This can save you from losing points in the final run when you mistakenly believe your code to be correct.

It is extremely helpful to compile your code with the following gcc options: `-Wall -Wextra -Wvla -Wconversion -pedantic`. This way the compiler can warn you about poor style and parts of your code that may result in unintended/undefined behavior.

Make sure that you are using `getopt_long()` for handling command-line options.

Appendix A

In order to ensure that your output is within the tolerable margins of error for the autograder to correctly grade your program you **must** run the following lines of code before you output anything to cout. We highly recommend that you simply put them at the top of your main function so that you don't forget about them.

```
cout << std::setprecision(2); //Always show 2 decimal places
cout << std::fixed; //Disable scientific notation for large numbers
```

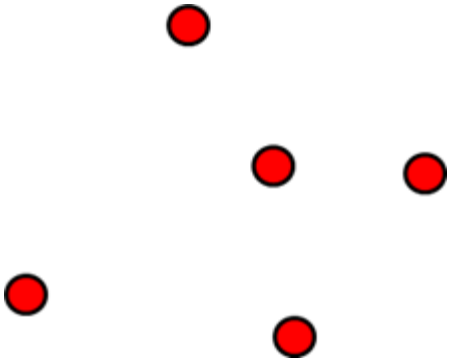
You will need to `#include <iomanip>` to use this code.

Appendix B

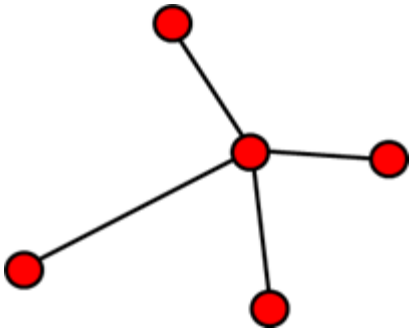
One possible heuristic (NOT the only one, NOT an easy one) is to construct a starting point for your TSP tour by following MST edges and skipping previously visited vertices. By using this technique correctly, you can find a tour that is guaranteed to be no more than twice the optimal solution's length (and use this "2x" check for debugging). You can then use this starting point to make adjustments and do better. This is not necessarily the best approach! It is one approach used to illustrate how a heuristic can be used, there are better ones out there. This example method is VERY hard to code. Do some research on other heuristics!

"Corner Cutting" algorithm illustrated

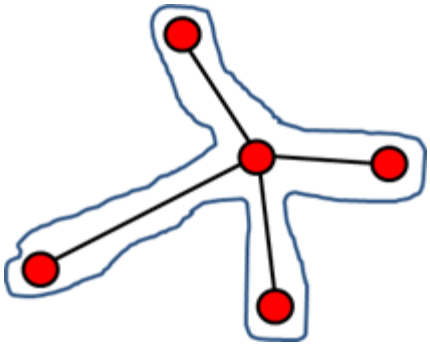
Suppose locations are distributed in an input map as shown below:



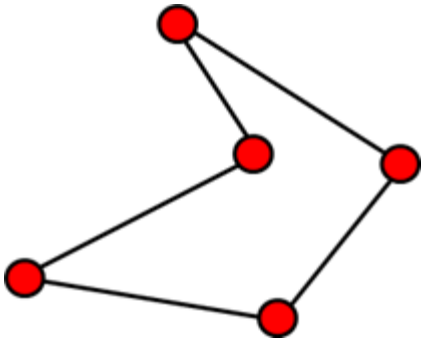
Below is an MST that would be formed from the above locations.



Below is a path taken by strictly following the edges of the MST.



However, by “cutting corners,” as described in the picture below, an effective TSP solution can be generated. This is possible because once a vertex is visited, it will not be visited again. In the above path that strictly follows the edges of the MST, the middle vertex is visited four times (visited after an outer vertex is visited). If the middle vertex is skipped after the first visit, a TSP tour shown below is achieved.



The reason we bring up this ‘twice-around-the-tree’ heuristic is to state the theorem that the resulting tours are no worse than 2x the MST cost.

The following is an explanation/proof sketch as to why the 2x bound is valid:

If you perform a DFS traversal of a tree and return to the initial node, you have visited every edge exactly twice ("going in" and "returning"), so this gives you exactly double the cost/length of the MST (the fact that the tree is minimal is not important for the logic of the proof - this works for any tree). Since the tree is spanning, you have visited all locations. The only problem with this twice-around-the-tree self-intersecting path is that it's not a tour. It can be turned into a tour by taking shortcuts (i.e., taking shortcuts is important not only to reduce the length).

When considering other heuristics:
1. The theorem allows you to upper-bound the cost of optimal TSP tours based on MST length.
2. The theorem has a constructive proof - a heuristic that always produces TSP tours that satisfy this upper bound[2].

[2] Such heuristics are also called approximation algorithms.

As a consequence, your heuristic should also satisfy the 2x upper bound; if not, you can just implement the twice-around-the-tree heuristic. However, we do not require this because there are much better heuristics – ones that are faster and produce better results.

Appendix C

Legend for test case names

Autograder test names are broken down by the mode used with each test:

- Name of test file starting with "**A**": these test files are used with the MST mode.
- Name of test file starting with "**B**": these test files are used with the FASTTSP mode.
- Name of test file starting with "**C**": these test files are used with the OPTTSP mode.

Test files that start with "**C**" are much smaller in size than those that start with "**A**" and "**B**". The number of cages for such files is smaller than 40 as compared to ten thousands of cages for the others. The reason is that TSP is NP-complete and finding an optimal solution in a reasonable amount of time is only possible for small-sized problems.

When you start submitting test files to the autograder, it will tell you (in the section called "Scoring student test files") how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!