COMP 120 - Problem Solving Assignment 5

Document Last Updated: January 24, 2024 @ 11:24 AM

Assignment Deadline: FIXME: UPDATE @ 10:00PM

Assignment Overview:

In this assignment you will write solve some interesting real-world problems using the power of recursive backtracking. There are two computational problems in this assignment.

- **Doctors Without Orders:** Doctors have limited time. Patients are waiting for help. Can everyone be seen?
- **Disaster Planning:** Cities need to plan for natural disasters. Emergency supplies are expensive. What's the cheapest way to prepare for an emergency?

Before you start coding, make sure you read through this whole document. If any of the instructions aren't clear, feel free to ask about it on CampusWire (use the `PSA5` category).

Pre-requisites

- Data Structures & Algorithms Textbook Chapter 4 (Recursion)
- Lab Session 10 (Recursive backtracking)

Learning Objectives

Upon successful completion of this PSA you will be able to do the following.

- Use *recursive backtracking* to solve a computational problem.
- Test Python code using the `pytest` module.
- Write Python code that uses the `assert` statement to check preconditions.

Initial Setup

Both you and your partner will need to get the starter code for your group using Git.

1. In VS Code, open the command palette and select the "Git Clone" option.
2. When prompted for the repository URL, enter the following, with X replaced by your group number (e.g. 7 or 12).

   ssh://git@code.sandiego.edu/comp120-sp24-s0Y-psa5-groupX

3. When prompted for where to save the repository, select the "comp120" folder you created earlier this semester. If you happen to get an error, make sure you click the "Git Log" button when prompted and look for the reported error message. Look at CampusWire to see if anyone else got the same error message: if not, create a new post, copying and pasting the output of the Git log.

4. Choose the "Open Repository" option in the window that pops up in the lower-right corner of the screen. In this repository you will find several Python files (*.py) and several text files in a directory named `data_files`.

Note that when working on the PSA, you should be careful to only work on one computer at a time (either you or your partner's). If you want to switch between computers, make sure you "Sync" your code.

IMPORTANT: As you complete and sync each problem, you can check the SAFE webapp for the results of some *limited* testing. Note that the messages you receive from the tests run are **purposely** vague: you should be relying on **your own** tests (as specified throughout this assignment) to get the diagnostic feedback that you need to solve your problems. Think of SAFE as a minimum backup to make sure that the very basics are working.

Computational Problem 1: Doctors Without Orders

The small country of Recursia faces a crisis: no one has told the Recursian doctors which patients they're supposed to see. They're Doctors Without Orders! As Minister of Health, it's time to help the Recursians with their medical needs.

Each doctor has a number of hours that they're capable of working in a day, and each patient has a number of hours that they need to be seen for. The question then arises: is it possible for every patient to be seen by a doctor for the appropriate number of hours, and to do so without exceeding the amount of time each doctor has available?

Your task is to write the following function inside of the `doctors.py` file.

```
def can_schedule_all(doctors: list[Doctor], patients: list[Patient],
                     schedule: dict[Doctor, set[Patient]]) -> bool:
```

This function that takes as input a list of doctors and a list of patients, then returns whether it's possible to schedule all the patients so that each one is seen by a doctor for the appropriate amount of time. Each patient must be seen by a single doctor, so, for example, a patient who needs five hours of time can't be seen by five doctors for one hour each. If it is possible to schedule everyone, the function should fill in the final `schedule` parameter by associating each doctor object (as a key) with the **set** of patients she should see (the value). See the notes below for more info about Python's `set` class, which is like lists, but with duplicate values removed.

The `doctors` parameter is list of `Doctor` objects, with two instance variables: their name (`name`) and the maximum numbers of hours they are available to see patients (`max_hours`). The `patients` parameter is a list of `Patient` objects, with two instance variables: their

name (`name`) and the number of hours they need to be seen for (`needed_hours`). Check out the docstring and doctest examples for both of these classes in the `doctors.py` and ask if you have any questions about them.

For example, suppose we have these doctors and these patients:

- Doctor Thomas: 10 Hours Free
- Doctor Taussig: 8 Hours Free
- Doctor Sacks: 8 Hours Free
- Doctor Ofri: 8 Hours Free
- Patient Lacks: 2 Hours Needed
- Patient Gage: 3 Hours Needed
- Patient Molaison: 4 Hours Needed
- Patient Writebol: 3 Hours Needed
- Patient St. Martin: 1 Hour Needed
- Patient Washkansky: 6 Hours Needed
- Patient Sandoval: 8 Hours Needed
- Patient Giese: 6 Hours Needed

In this case, everyone can be seen as follows.

- Doctor Thomas (10 hours free) sees Patients Molaison, Gage, and Writebol (10 hours total)
- Doctor Taussig (8 hours free) sees Patients Lacks and Washkansky (8 hours total)
- Doctor Sacks (8 hours free) sees Patients Giese and St. Martin (7 hours total)
- Doctor Ofri (8 hours free) sees Patient Sandoval (8 hours total)

However, minor changes to the patient requirements can completely invalidate this schedule. For example, if Patient Lacks needed to be seen for three hours rather than two, then there is no way to schedule all the patients so that they can be seen. On the other hand, if Patient Washkansky needed to be seen for seven hours instead of six, then there would indeed a way to schedule everyone. (Do you see how?)

The main challenge in solving this problem is coming up with the right recursive strategy. When generating subsets, the question we ask is “do we want to include or exclude this element?” When generating permutations, the question we ask is “which element do we want to pick next?” Now, think about what you need to do for this assignment. What question should you ask at each level of the recursion?

There are two general strategies you can use to solve this problem. One of them is to go one doctor at a time, deciding which subset of patients that doctor should see. Another is go to one patient at a time, deciding which doctor should see her. One of these strategies, in our opinion, is much easier than the other. Take a few minutes to think through which approach might be easier before starting to code anything up.

Here's what you need to do:

1. Write the docstring comment for the `can_schedule_all` function in doctors.py. No need to write doctest examples.

2. Add pytest test cases for this function to test_doctors.py. This file already has a couple of test cases filled out for you. There are several others that need to be filled out, using the comments inside them to guide you on what that test function should check.

   For these functions, you should create the list of doctors and patients manually (not from reading them in from a file). This will allow you to create simpler tests than if you were to use the given data files.

3. Implement the `can_schedule_all` function in doctors.py. You should determine whether there is a schedule in which every patient is scheduled and no doctor needs to work more hours than they have available. If so, you should fill in the `schedule` parameter with one such schedule.

4. Run your automated unit tests with the `pytest` testing framework either by running that file in VS Code or by using the following command in the terminal.

   ```
   python3 -m pytest test_doctors.py
   ```

   Once all your tests pass and you are confident that your solution works correctly, you may move onto the next step.

5. You can now run the `doctors.py` application on the data files located in the `data_files` folder. Open up one of the files with the extension ".dwo" (e.g. OneDoctor.dwo) to see what data is contained in that file. Then run the application, specifying the data file you want to run as follows.

   ```
   python3 doctors.py data_files/OneDoctor.dwo
   ```

   Ensure that the application prints out what you expect, based on the file given.

   Now run the file with as many of the ".dwo" files as you can. The grader will run **all** of these files and check that the results are correct so you would be wise to do the same.

Notes

- You may find it easier to solve this problem first by simply getting the return value right, completely ignoring `schedule`. Once you're sure that your code is always

producing the right answer, update it so that you actually fill in the schedule. Doing so shouldn't require too much code, and it's way easier to add this in at the end than it is to debug the whole thing all at once.

- You can assume the `schedule` has all of the doctors (the keys) associated with a set of empty patients (the values) when the function is first called.

  Check the "if" section at the bottom of the doctors.py to see how this is set up in the final application and use this as a template for creating the schedule in your unit tests in test_doctors.py.

- A `set` is a built-in data type in Python. You can think of a set as a list that is guaranteed to have no repeats. For this function, you should only need to use the `add` and `remove` methods on sets, as demonstrated on the example below.

```
>>> set1 = {5, 6, 12}
>>> set1
{5, 6, 12}
>>> set1.add(3)
>>> set1.add(6)
>>> set1.remove(5)
>>> set1  # look ma, no repeated 6!
{3, 6, 12}
>>> set2 = set() # This is an empty set
>>> len(set2)
0
```

  While not shown in the example, you can loop through a set by item just like you would loop through a list (e.g. `for item in set1`).

  As it applies to this problem, you will likely use the `add` method to "choose" an option, then use the `remove` method to "unchoose" that option before moving on to the next one.

- If your function returns False, the final contents of the schedule don't matter (our guess is that you will leave it the same as the starting schedule though, with all doctors assigned to no patients).

- You can assume no two doctors have the same name and no two patients have the same name.

- You might be tempted to solve this problem by repeatedly taking the patient requiring the most time and assigning them to the doctor with the most available hours, or by taking the doctor with the least time and giving them the patients requiring the fewest hours, or something like this. Solutions like these are called greedy algorithms, and while greedy algorithms do work well for some problems, this problem is not one of them. In fact, there are no known ways to solve this problem efficiently using greedy algorithms!

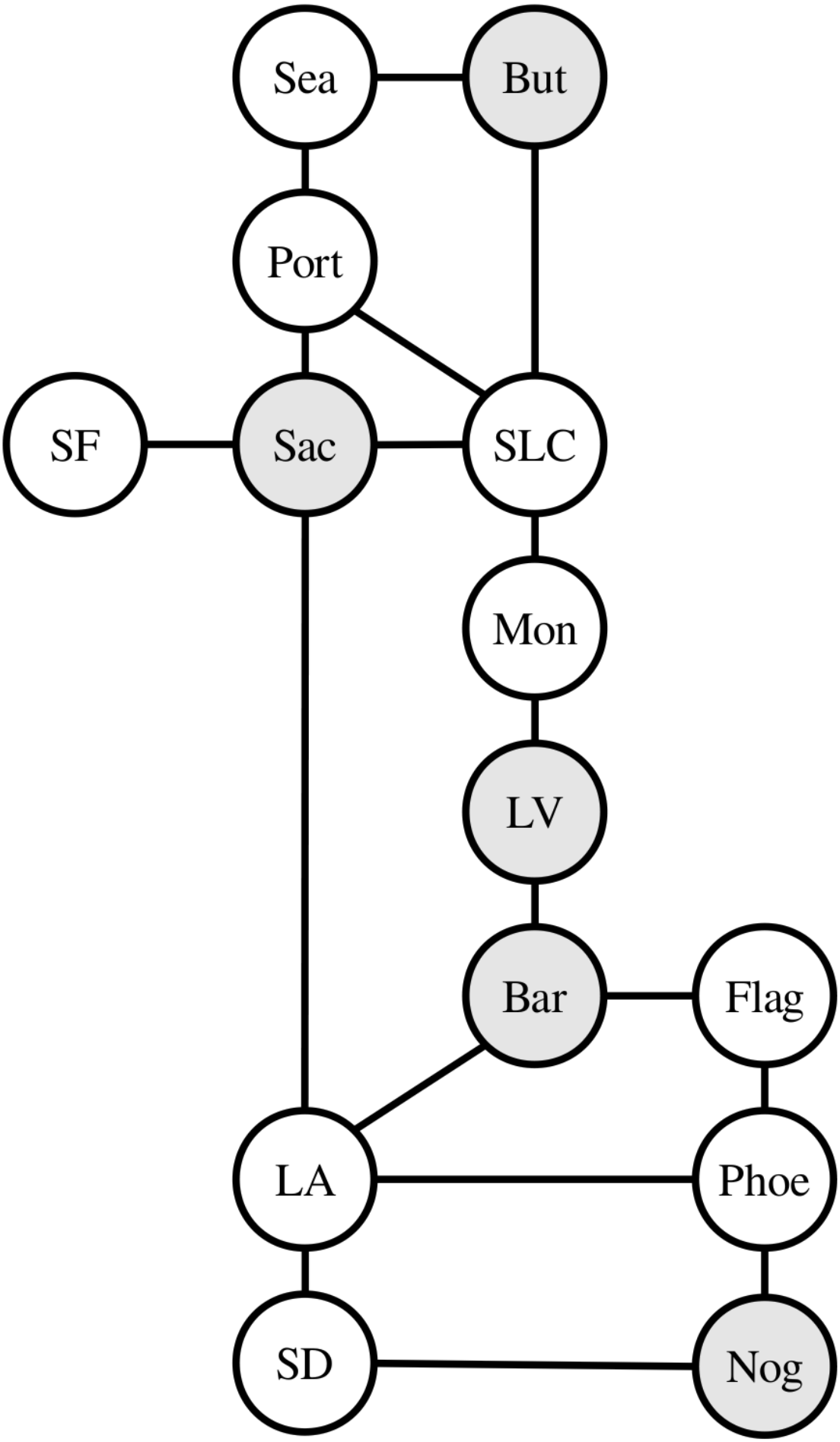
Grading

This problem will be graded as follows.

1. Correctly implemented all pytest cases in `test_doctors.py` (3 pts)
2. All pytest test cases pass. (2 pts)
3. Correctly works for all ".dwo" files in the `data_files` directory. (10 pts)

Computational Problem 2: Disaster Planning

Disasters–natural and unnatural–are inevitable, and cities need to be prepared to respond to them. The problem is that stockpiling emergency resources can be really, really expensive. As a result, it's reasonable to have only a few cities stockpile emergency supplies, with the plan that they'd send those resources from wherever they're stockpiled to where they're needed when an emergency happens. The challenge with doing this is to figure out where to put resources so that (1) we don't spend too much money stockpiling more than we need, and (2) we don't leave any cities too far away from emergency supplies.


Sea
But
Port
SF
Sac
SLC
Mon
LV
Bar
Flag
LA
Phoe
SD
Nog

Example from Western United States

Imagine that you have access to a country's major highway networks and know which cities are are right down the highway from others. To the right is a fragment of the US Interstate Highway System for the Western US. Suppose we put emergency supplies in Sacramento, Butte, Nogales, Las Vegas, and Barstow (shown in gray). In that case, if there's an emergency in any city, that city either already has emergency supplies or is immediately adjacent to a city that does. For example, any emergency in Nogales would be covered, since Nogales already has emergency supplies. San Francisco could be covered by supplies from Sacramento, Salt Lake City is covered by both Sacramento and Butte, and Barstow is covered both by itself and by Las Vegas.

Although it's possible to drive from Sacramento to San Diego, for the purposes of this problem the emergency supplies stockpiled in Sacramento wouldn't provide coverage to San Diego, since they aren't immediately adjacent.

We'll say that a country is disaster-ready if every city either already has emergency supplies or is immediately down the highway from a city that has them.

Your task is to write a function named `can_be_disaster_ready` in the `supplies.py` file.

```
def can_be_disaster_ready(road_network: dict[str, set[str]], num_cities: int,
                          supply_locations: set[str]) -> bool:
```

This function takes as input a dictionary representing the road network for a region (described below) and the number of cities you can afford to put supplies in, then returns whether it's possible to make the region disaster-ready without placing supplies in more than `num_cities` cities. If so, the function should then populate the argument `supply_locations` with all of the cities where supplies should be stored.

In this problem, the road network is represented as a dictionary where each key is a city and each value is a set of cities that are immediately down the highway from them. For example, here's a fragment of the map you'd get from the above transportation network:

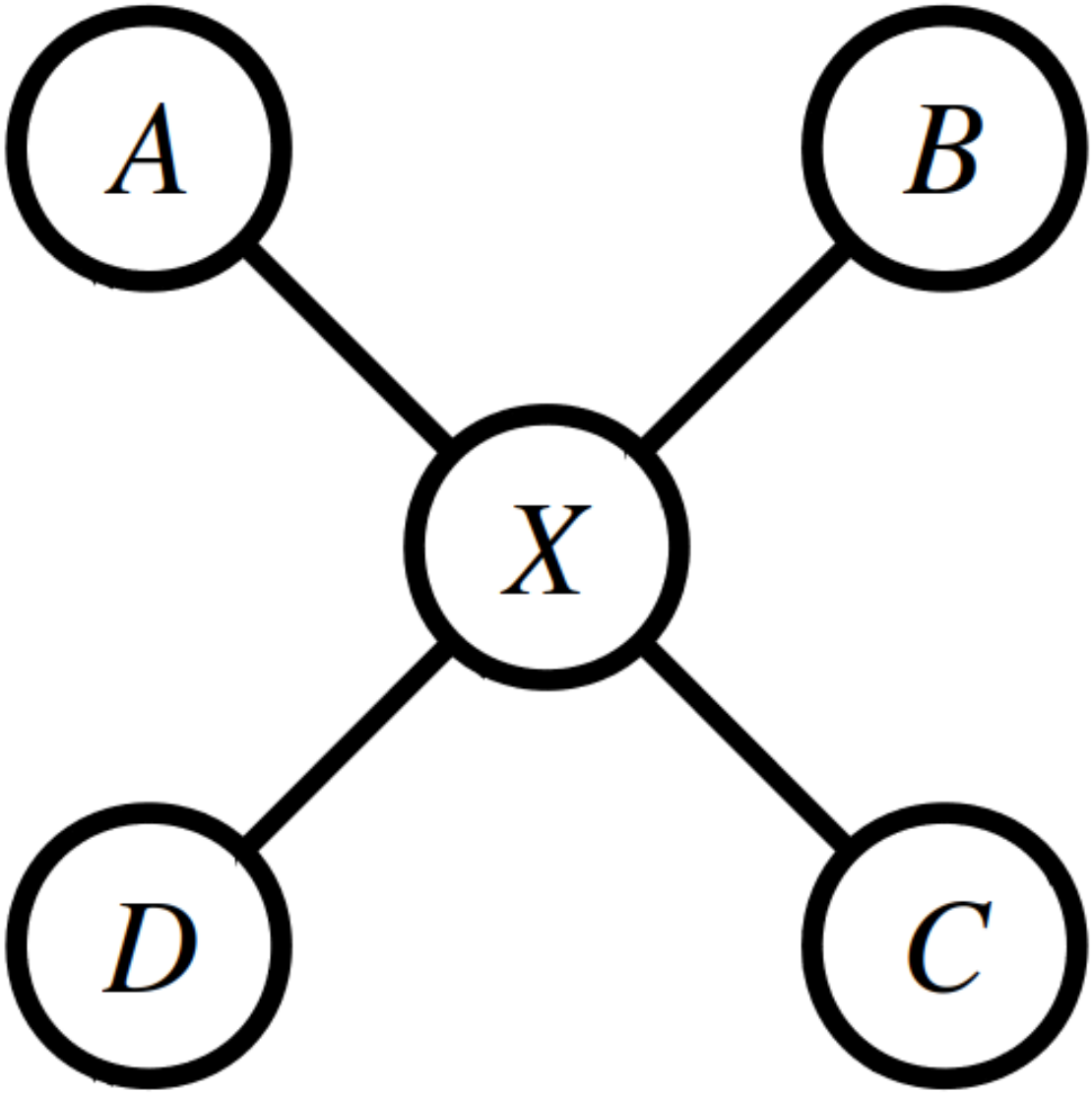
```
"Sacramento":    {"San Francisco", "Portland", "Salt Lake City", "Los
Angeles"}
"San Francisco": {"Sacramento"}
"Portland":      {"Seattle", "Sacramento", "Salt Lake City"}
```

You can assume that `supply_locations` is empty when this function is first called, and you can change it however you'd like if the function returns false.

You might be tempted to solve this problem by approaching it as a combinations problem. We need to choose some group of cities, and there's a limit to how many we can pick, so we could just list all combinations of `num_cities` cities and see if any of them provide coverage to the entire network. The problem with this approach is that as the number of cities rises, the number of possible combinations can get way out of hand. For example, in a network with 35 cities, there are 3,247,943,160 possible combinations of 15 cities to choose from. Searching over all of those options can take a very, very long time, and if you were to

approach this problem this way, you'd likely find your program grinding to a crawl on many transportation grids.

To speed things up, we'll need to be a bit more clever about how we approach this problem. There's a specific insight we'd like you to use that focuses the recursive search more intelligently and, therefore, reduces the overall search time.



Recursive Idea

Here's the idea. Suppose you pick some city that currently does not have disaster coverage. You're ultimately going to need to provide disaster coverage to that city, and there are only two possible ways to do it: you could stockpile supplies in that city itself, or you can stockpile supplies in one of its neighbors. For example, suppose city X shown to the right isn't yet covered, and we want to provide coverage to it. To do so, we'd have to put supplies in either X itself or in one of A, B, C, or D. If we don't put supplies it at least one of these cities, there's no way X will be covered.

With that in mind, **use the following strategy to solve this problem**. Pick an uncovered city, then try out each possible way of supplying that city (either by stockpiling in that city itself or by stockpiling in a neighboring city). If after committing to any of those decisions you're then able to cover all the remaining cities, fantastic! You're done. If, however, none of those decisions ultimately leads to total coverage, then there's no way to supply all the cities.

In summary, here's what you need to do:

1. Write the docstring comment for the `can_be_disaster_ready` function in supplies.py. Again, no need to write doctest examples.

2. Complete the pytest test functions in `test_supplies.py`.

   As before, a couple are already completed while the rest need to be filled in based on the comment describing their test scenario.

3. Implement the `can_be_disaster_ready` function in supplies.py using the recursive strategy outlined above. Specifically, do the following:

   - Choose a city that hasn't yet been covered.
   - For each way it could be covered–either by stockpiling supplies in that city or by stockpiling in one of its neighbors–try providing coverage that way. If you can then (recursively) cover all cities having made that choice, great! If not, that option didn't work, so you should pick another one.
   - One precondition for this function is that `num_cities` be a non-negative value. Add an `assert` statement at the top of this function to enforce this precondition.
   - Speaking of the `num_cities` parameter: be careful not subtly change its meaning. This parameter should represent the total number of cities that may contain supplies, **not** how many *more* cities (beyond those already selected) can contain supplies. With this interpretation in mind, if the number of cities is less than the number of cities found in `supplies_locations`, it is not a valid configuration so you can return False in this case.

4. Run your automated pytest tests either by running the `test_supplies.py` file in VS Code or by using the following terminal command.

   ```
   python3 -m pytest test_supplies.py
   ```

   If any of your test cases fail, use your VS Code debugger to set a breakpoint at the beginning of that test case and then step through your program to ensure the logic is correct.

Once all your tests pass and you are confident that your solution works correctly, you may move onto the next step.

5. Now you can run your app with some of the real-world examples give in the `data_files` directory. Open up one of the files with the extension ".dst" (e.g. WesternUS.dst) to see what data is contained in that file. Then run the application, specifying the data file you want to run as well as the maximum number of supply cities, as follows.

   ```
   python3 supplies.py data_files/WesternUS.dst 5
   ```

   Ensure that the application prints out what you expect, based on the file given (in this case, it is possible to be disaster-ready with only 5 supply cities). Try again, replacing the 5 with 2, a scenarios that is not possible (i.e. can't be disaster-ready with only 2 cities on that map).

   It can be very hard to understand if your answer is correct based on the text based output from the basic supplies.py application. To make your life easier, we have provided a GUI application in `supplies_vizualizer.py` that shows the map as well as the selected supply cities.

   You may run this visualizer you first need to install the proper Python GUI module (pysimplegui) as follows in your terminal:

   ```
   python3 -m pip install -U pysimplegui
   ```

   Now you can run the visualizer by opening up the `supplies_visualizer.py` file in VS Code and simply hitting the "Play" button. If you want to be cool and use the command line, here's the way to run it.

   ```
   python3 supplies_visualizer.py
   ```

   You should see a new window pop up with a visualization of the British Isles. The way this visualizer works is that it calls your function multiple times to find the minimum number of cities need to be disaster-ready and then shows the map with that number of cities selected. You can change the map using the drop-down menu at the bottom of the window and then clicking the "Solve" button.

   If the visualizer doesn't show up, it's either because your implemented function wasn't correct (or is maybe taking too long… see the notes below) or you didn't correctly install the `pysimplegui` module. Checking the Terminal should help you see what was printed to help identify which of those two was the issue.

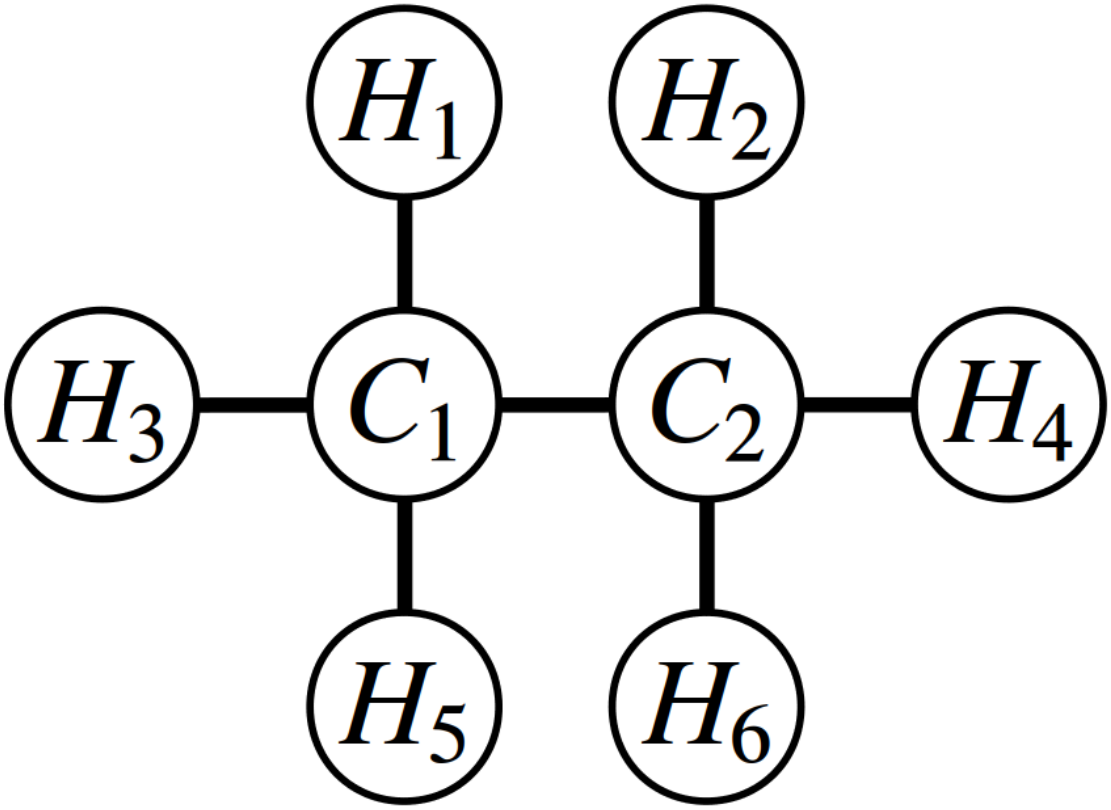
Notes

- We recommend proceeding in two steps. First, just focus on getting the return value right–that is, write a function that answers the question "Is it possible to cover everything with only this many cities having supplies?" and which ignores the

`supply_locations`. Once that's working–and no sooner– edit the code to then fill in the `supply_locations` with which cities should be chosen.

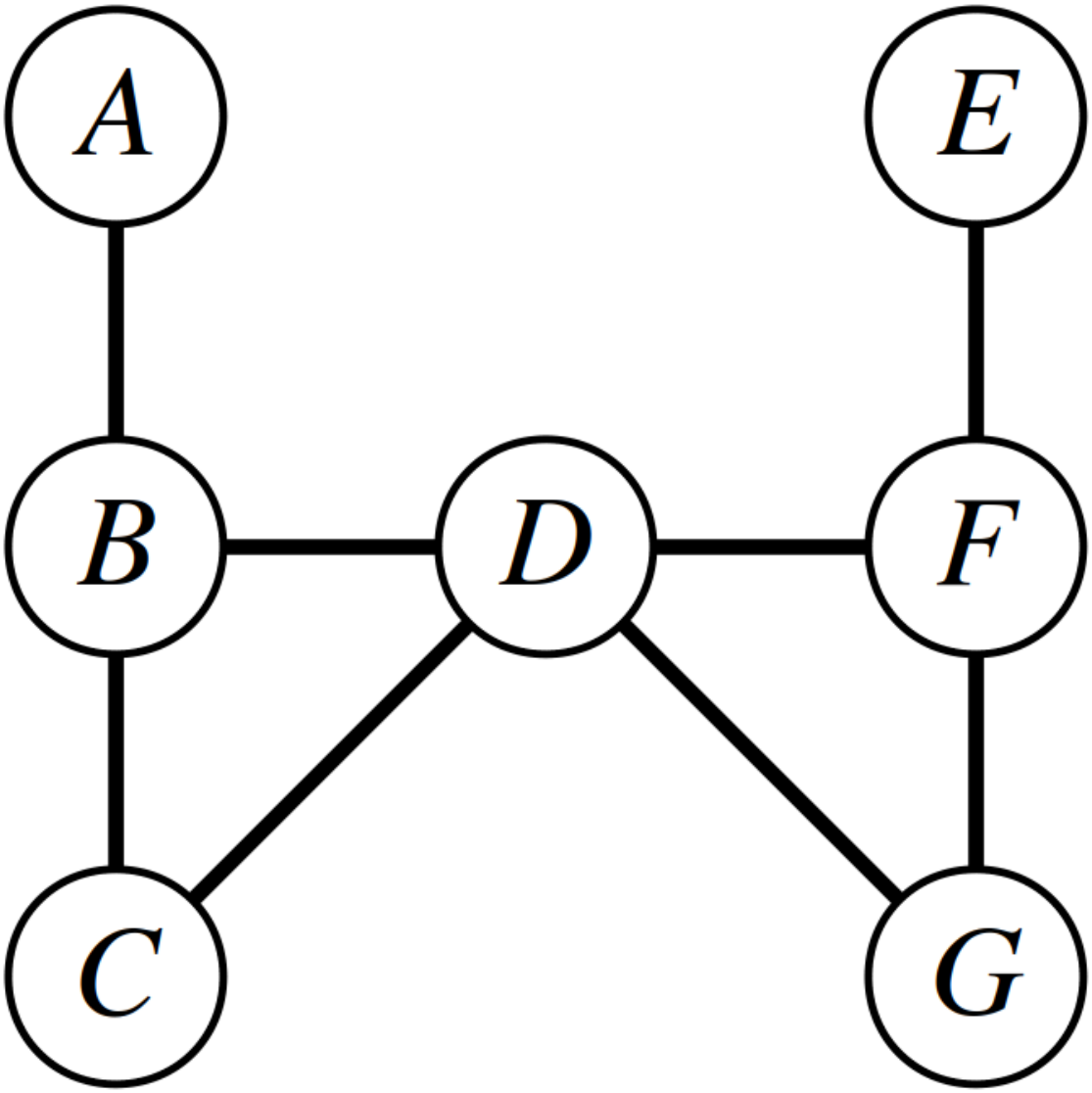
- The road network is bidirectional. If there's a road from city A to city B, then there will always be a road back from city B to city A. Both roads will be present in the parameter `road_network`. You can rely on this.
- Every city appears as a key in the dictionary. Cities can exist that aren't adjacent to any other cities in the transportation network. If that happens, the city will be represented by a key in the dictionary associated with an empty set of adjacent cities.
- The `num_cities` parameter denotes the maximum number of cities you're allowed to stockpile in. It's okay if you use fewer than `num_cities` cities to cover everything, but you can't use more.
- The `num_cities` parameter may be zero, but should not be negative. You should use an `assert` statement to enforce this precondition.
- Get out a pencil and paper when debugging this one and draw pictures that show what your code is doing as it runs. Trace through your code to see what your recursion is doing. Make sure that the execution of the code mirrors the high-level algorithm described above. Can you see your code picking an uncovered city? Can you see it trying out all ways of providing coverage to that city?
- Make sure you're correctly able to tell which cities are and are not covered at each point in time. One of the most common mistakes we've seen people make in solving this problem is to accidentally mark a city as uncovered that actually is covered, usually when backtracking.

  We strong suggest making a function named `uncovered_cities` that uses the road network and the supply locations and returns a list of cities that are uncovered.
- There are cases where the best way to cover an uncovered city is to stockpile in a city that's already covered. In the example shown to the right, which is modeled after the molecular structure of ethane, the best way to provide coverage to all cities is to pick the two central cities C_1 and C_2, even though after choosing C_1 you'll find that C_2 is already covered by C_1.



Counter-intuitive Stockpiling

- You might be tempted to solve this problem by repeatedly taking the city adjacent to the greatest number of uncovered cities and then stockpiling there, repeating until all cities are covered. Surprisingly, this approach will not always work. In the example shown to the right here, which we ve entitled “Don’t be Greedy,” the optimal solution is to stockpile in cities B and F. If, on the other hand, you begin by grabbing city D, which would provide coverage to five of the seven cities, you will need to stockpile in at least two more cities (one of A and B, and one of E and F) to provide coverage to everyone. If you follow the recursive strategy outlined above, you won’t need to worry about this, since that solution won’t always grab the city with the greatest number of neighbors first.



Don't Be Greedy

- Some of the sample files that we've included have *a lot* of cities in them. The samples whose names start with `VeryHard` are, unsurprisingly, very hard tests that may require some time for your code to solve. It's okay if your program takes a long time (say, at most two minutes) to answer queries for those transportation grids, though the other samples shouldn't take very long to complete.

Grading

This problem will be graded as follows.

1. Correctly implemented all pytest cases in `test_supplies.py` (3 pts)
2. All pytest test cases pass. (2 pts)
3. Correctly works for all ".dst" files in the `data_files` directory, using several different values for the maximum number of supply cities. (15 pts)

Commenting Standards

Code that is not well documents and tested is worthless and potentially dangerous. You therefore have an ethical responsibility to write software that has enough commenting (and tests).

For this assignment, you are expected to have an appropriate level of commenting throughout your code. This means the following.

1. A docstring comment at the beginning of **every** function/method of the **exact** format specified in Lab 01 and at the beginning **every** class you write.

2. Comments integrated into the body of your functions that clarifies the operation of the code. You should **not** have a comment for every line of code: that is unnecessary and harms the readability of the code. Instead, focus on areas that are either algorithmically challenging, have complex syntax, or need more context (e.g. explaining how this ties to the overall problem being solved).

IMPORTANT: Submissions that do not meet the minimum standards for commenting will received a 0.

Submission Instructions

To submit your code, you will simply need to ensure that all of your changes have been stages, committed, and sync'ed to the server. You are also encouraged to run your final code on both you and your partner's computers to ensure you aren't relying on a special module or program for correct functionality.

After your code is synced, review the results on the SAFE webapp. Note that these results are not comprehensive of all of the PSA requirements so having all green checkmarks does not mean you'll receive a perfect score.

WARNING: Don't assume that a "minor" change to your code (e.g. adding a comment) doesn't need to be tested. It is your responsibility to make sure that your final synced code is fully working.

Acknowledgement

The idea and write-up for this lab are heavily inspired by the work of Keith Schwartz at Stanford University. Thank you to Dr. Schwartz for sharing this assignment with the CS community!