# XCS221 Assignment 2 — Route Planning

**Due Sunday, March 24 at 11:59pm PT.**

### Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs221-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

### Submission Instructions

**Written Submission:** Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Also note that your answers should be in order and clearly and correctly labeled to receive credit. Be sure to submit your final answers as a PDF and **tag all pages correctly when submitting to Gradescope**.

**Coding Submission:** Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

### Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

### Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.
- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

### Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

### Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

### Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```



### Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

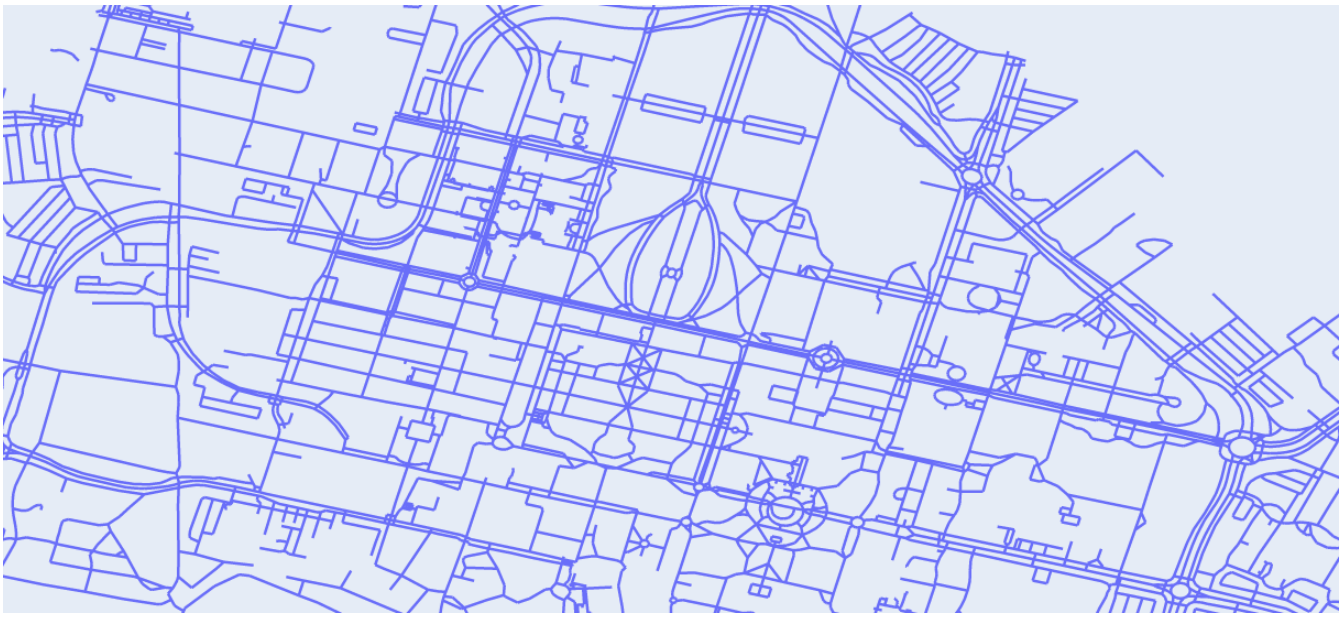
Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

**Introduction**



In route planning, the objective is to find the best way to get from point A to point B (think Google Maps). In this homework, we will build on top of the classic shortest path problem to allow for more powerful queries. For example, not only will you be able to explicitly ask for the shortest path from the Gates Building to Coupa Cafe at Green Library, but you can ask for the shortest path from Gates back to your dorm, stopping by the package center, gym, and the dining hall (in any order!) along the way.

We will assume that we have a *map* of a city (e.g., Stanford) consisting of a set of *locations*. Each location has:

1. a unique label (e.g., 6608996258),
2. a (latitude, longitude) pair specifying where the location is (e.g., 37.4299866, -122.175519), and
3. a set of *tags* which describes the type of location (e.g., amenity=food).

There are a set of *connections* between pairs of locations; each connection has a *distance* (in meters), and can be traversed in both directions (if the distance from A to B is 100 meters, then the distance from B to A is also 100 meters).

There are two city maps that you'll be working with: a grid map (`createGridMap`) and a map of Stanford (`createStanfordMap`), which is derived from Open Street Maps. We have also included instructions on how to create your own maps in `README.md`.

1. **Grid City**

Consider an infinite city consisting of locations $(x, y)$ where $x, y$ are integers. From each location $(x, y)$, one can go east, west, north, or south. You start at $(0, 0)$ and want to go to $(m, n)$, where $m, n \geq 0$. We can define the following search problem to capture this:

- $s_{\text{start}} = (0, 0)$
- $\text{Actions}(s) = \{(+1, 0), (-1, 0), (0, +1), (0, -1)\}$
- $\text{Succ}(s, a) = s + a$
- $\text{Cost}((x, y), a) = 1 + \max(x, 0)$ (it is more expensive as $x$ increases)
- $\text{IsEnd}(s) = \mathbf{1}[s = (m, n)]$

(a) **[2 points (Written)]**

What is the minimum cost of reaching location $(m, n)$ (note that $m, n \geq 0$) starting from location $(0, 0)$ in the above city? Describe one possible path achieving the minimum cost. Is it unique (i.e., are there multiple paths that achieve the minimum cost)?

**What we expect:** Provide an expression for the minimum cost. In addition, using 1 - 2 sentences describe one possible minimum cost path and state whether it is unique.

(b) **[3 points (Written)]** How will Uniform Cost Search (UCS) behave on this problem? Mark the following as true or false:

i. UCS will never terminate because the number of states is infinite.

ii. UCS will return the minimum cost path and explore only locations between $(0, 0)$ and $(m, n)$; that is, $(x, y)$ such that $0 \leq x \leq m$ and $0 \leq y \leq n$.

iii. UCS will return the minimum cost path and explore only locations whose past costs are less than the minimum cost from $(0, 0)$ to $(m, n)$.

**What we expect:** T/F for each subpart.

(c) **[3 points (Written)]** Now consider running UCS on an arbitrary graph. Mark the following as true or false:

i. If you add a connection between two locations, the minimum distance cannot go up.

ii. If you make the cost of an action from some state small enough (possibly negative), that action will show up in the minimum cost path.

iii. If you increase the cost of each action by 1, the minimum cost path does not change (even though its cost does).

**What we expect:** T/F for each subpart.

## 2. Finding Shortest Paths

We first start out with the problem of finding the shortest path from a *start location* (e.g., the Gates Building) to some end location. In Google Maps, you can only specify a specific end location (e.g., Coupa Cafe at Green Library).

In this problem, we want to give the user the flexibility of specifying *multiple* possible end locations by specifying a set of "tags" (e.g., so you can say that you want to go to *any* place with food versus a specific location like Tressider).

(a) **[8 points (Coding)]**

Implement `ShortestPathProblem` so that given a `startLocation` and `endTag`, the minimum cost path corresponds to the shortest path from `startLocation` to *any* location that has the `endTag`.

In particular, you need to implement `startState()`, `isEnd(state)`, `successorsAndCosts(state)`.

Recall the separation between search problem (modeling) and search algorithm (inference). You should focus on modeling (defining the `ShortestPathProblem`); the default search algorithm, `UniformCostSearch` (UCS), is implemented for you in `util.py`.

**What we expect:** An implementation of the `ShortestPathProblem` class.

(b) **[3 points (Written)]**

Run `python mapUtil.py > readableStanfordMap.txt` to write a (long-ish) file of the possible locations on the Stanford map along with their tags. Each tag is a `[key]=[value]`. Here are some examples of keys:

- `landmark`: Hand-defined landmarks (from `src/data/stanford-landmarks.json`)
- `amenity`: Various amenity types (e.g., "park", "food")
- `parking`: Assorted parking options (e.g., "underground")

Choose a starting location and end tag (perhaps that's relevant to your daily life) and implement `getStanfordShortestPathProblem()` to create a search problem. Then, run `python grader.py 2b-0-helper` to generate `path.json`. Once generated, run `python visualization.py` to visualize it (opens in browser). Try different start locations and end tags. Pick two settings corresponding to the following:

- A start location and end tag that produced new insight into traveling around campus. Describe whether the system was useful.
- A start location and end tag where the minimum cost path found isn't desirable. Is this due to incorrect modeling assumptions? Explain.

You should feel free to add new landmarks and if you are not at Stanford, follow the instructions in the `README.md` to use your own map and landmarks.

**What we expect:** A screenshot of the visualization of your two solutions as well as i) one or more sentences describing something interesting you've learned about traveling (around campus, or elsewhere) and ii) something incorrect about either the map or modeling assumptions (such a landmark being out of place, etc.).

### 3. **Finding Shortest Paths with Unordered Waypoints**

Let's introduce an even more powerful feature: unordered waypoints! In Google Maps, you can specify an ordered sequence of waypoints that a path must go through – for example, going from point A to point X to point Y to point B, where [X, Y] are "waypoints" (such as a gas station or a friend's house).

However, we want to consider the case where the waypoints are *unordered*: $X, Y$, so that both $A \to X \to Y \to B$ and $A \to Y \to X \to B$ are allowed. Moreover, $X$, $Y$, and $B$ are each specified by a tag like in Problem 2 (e.g., `amenity=food`)

This is a neat feature if you think about your day-to-day life; you might be on your way home after a long day, but need to stop by the package center, Tressider to grab a bite of food, and the bookstore to buy some notebooks. Having the ability to get a short, quick path that hits all these stops might be really convenient (rather than searching over the various waypoint orderings yourself).

(a) **[11 points (Coding)]**

Implement `WaypointsShortestPathProblem` so that given a `startLocation`, set of `waypointTags`, and an `endTag`, the minimum cost path corresponds to the shortest path from `startLocation` to a location with the `endTag`, such that all of `waypointTags` are covered by some location in the path.

Note that a single location can be used to satisfy multiple tags.

Like in Problem 2, you need to implement `startState()`, `isEnd(state)`, and `successorsAndCosts(state)`.

There are many ways to implement this search problem, so you should think carefully about how to design your `State`. We want to optimize for a compact state space so that search is as efficient as possible.

**What we expect:** An implementation of the `WaypointsShortestPathProblem` class. To get full credit, your implementation must have the right asymptotic dependence on the number of locations and waypoints. Note that your code will timeout if you do this incorrectly.

(b) **[2 points (Written)]**

If there are $n$ locations and $k$ waypoint tags, what is the maximum number of states that UCS could visit?

**What we expect:** A mathematical expression that depends on $n$ and $k$, with a brief explanation justifying it.

(c) **[3 points (Written)]**

Choose a starting location, set of waypoint tags, and an end tag (perhaps that captures an interesting route planning problem relevant to you), and implement `getStanfordWaypointsShortestPathProblem()` to create a search problem. Then, similar to Problem 1b, run `python grader.py 3c-0-helper` to generate `path.json`. Once generated, run `python visualization.py` to visualize it (opens in browser).

**What we expect:** A screenshot of the visualized path with a list of the waypoint tags you selected. In addition, provide one or more sentences describing unexpected or interesting features of the selected path. Does it match your expectations? What does this path fail to capture that might be important?

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

---

# 1.a

# 1.b

# 1.c

## 2.b

## 3.b

## 3.c

# 4.d