1 Dynamic Programming (Part 2)

In this notebook, we'll implemement dynamic programming algorithms for the Traveling Salesperson Problem.

1.0.1 If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

1.0.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.8` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

Every time you want to work

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

```
In [11]: # Install dependencies
         !pip install -r requirements.txt --quiet
```

```
In [1]: import otter
        assert (otter.__version__ >= "5.4.1"), "Please reinstall the requirements and restart your kern

        grader = otter.Notebook("hw08-coding.ipynb")
        import pickle
        import tqdm
        import time
        from itertools import combinations
        from autograder_utils import validate_tour

        with open('public_data.pkl', 'rb') as f:
            test_data = pickle.load(f)

        rng_seed = 42
```

1.1 Traveling Salesperson DP

Now, we'll implement the dynamic programming algorithm for the traveling salesperson problem (TSP). A brute force solution will be hopelessly slow even for moderate-sized test cases, but we can use dynamic programming to get a solution in slightly more reasonable (but still exponential) time. For a refresher on the TSP algorithm, see https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=20.

As with problems in previous homeworks, we want you to return the actual tour, not just the cost of the tour. We can once again apply the same procedure of backtracking through our subproblem array to reconstruct this tour.

Subproblem Representation: Since our subproblem definition takes in a set as one of its parameters, we can't just use a 2D array to store our subproblems. Instead, we recommend storing subproblems in a dictionary, where the keys are tuples of the form `(S, i)`, where `i` represents the last city visited before returning home and `S` is the set of cities visited so far.

To ensure that the keys are hashable, we recommend using Python's built-in `frozenset` class for `S`. `frozenset` is built-in to Python so you can use it without any additional imports, and works just like a normal set, except that it is immutable and hashable. You can read more about `frozenset` here: https://docs.python.org/3/library/stdtypes.html#frozenset.

or:

An alternative approach would be to use a 2D array as usual, but use a bitmask to represent the set of visited cities. In this approach, `S` would be represented as an n-bit unsigned integer, and the i-th bit of `S` would be set to 1 if and only if the i-th city is part of the set of visited cities. Then, since `S` is an integer, we can use it to index into our 2D array.

Implementation Details: There are many possible solutions to this problem.

One can use a bottom-up approach similar to the pseudocode from DPV, in which case the provided helper function below may be useful.

or:

One can use a top-down approach, in which case a recursive helper function may be useful.

Regardless of the approach, **storing the entire tour at each step is too memory-intensive and will cause the autograder to fail.** Instead, consider maintaining a separate dictionary or array which stores a smaller amount of information but can still help you reconstruct the tour.

```
In [2]: def get_subsets(n, size=None):
            '''Get all subsets of size `size` of the set {0, 1, ..., n-1}.
            If size is None, return all subsets of all sizes.'''

            if size is None:
                for size in range(n + 1):
                    yield from get_subsets(n, size)
            else:
                yield from map(frozenset, combinations(range(n), size))
```

since the provided function is a generator function, you can't directly treat its output as a list. Instead, you should iterate through it using a loop as shown below:

```
In [3]: # print the power set of {0, 1, 2, 3}
        for subset in get_subsets(4, size=None):
            print(subset)
```

```
frozenset()
frozenset({0})
frozenset({1})
frozenset({2})
frozenset({3})
frozenset({0, 1})
frozenset({0, 2})
frozenset({0, 3})
frozenset({1, 2})
frozenset({1, 3})
frozenset({2, 3})
frozenset({0, 1, 2})
frozenset({0, 1, 3})
frozenset({0, 2, 3})
frozenset({1, 2, 3})
frozenset({0, 1, 2, 3})
```

```
In [4]: def tsp_dp(dist_arr):
            """Compute the exact solution to the TSP using dynamic programming and returns the optimal 

            Args:
                dist_arr (ndarray[int]]): An n x n matrix of distances between cities. dist_arr[i][j] i

            Returns:
                List[int]: A list of city indices representing the optimal path.
            """
            # BEGIN SOLUTION
            # Main Algorithm
            C = {}
            prev = {}
            C[(frozenset([0])), 0] = 0
            prev[(frozenset([0])), 0] = None
            n = len(dist_arr)
```

```
    for s in range(2, n+1):
        for S in get_subsets(n, s):
            if 0 in S:
                for j in S:
                    if j == 0:
                        C[(S, j)] = float('inf')
                    else:
                        mini = float('inf')
                        S_prime = S.difference(frozenset([j]))
                        ideal_i = 0
                        for i in S_prime:
                            if C[(S_prime, i)] + dist_arr[i][j] < mini:
                                mini = C[(S_prime, i)] + dist_arr[i][j]
                                ideal_i = i
                        C[(S, j)] = mini
                        prev[(S, j)] = (S_prime, ideal_i)

    # Finding ideal j (last city)
    S_full = frozenset(range(n))
    min_dist, ideal_j = min((C[(S_full, j)] + dist_arr[j][0], j) for j in range(1, n))

    # Backtracking
    result = []
    backtrack = (S_full, ideal_j)
    while prev[backtrack] != None:
        result.append(backtrack[1])
        backtrack = prev[backtrack]
    result.append(0)
    return result
    # END SOLUTION
```

1.1.1 Debugging

A simplified, non-comprehensive verion of the otter tests are pasted here for your convenience. Feel free to add whatever print statements or assertions you'd like when debugging.

Points: 6

```
In [5]: # tests on very small cases
        for dist_arr, expected_distance in tqdm.tqdm(test_data['debugging_tests']):
            result = tsp_dp(dist_arr)

            assert set(result) == set(range(len(dist_arr))), f"Your output does not visit all cities"
            student_length = validate_tour(result, dist_arr)
            assert student_length >= 0, f"Your output is not a valid tour"
            assert student_length == expected_distance, f"Your output is not a minimum distance tour"

        print("All tests passed!")
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[5], line 3
      1 # tests on very small cases
      2 for dist_arr, expected_distance in tqdm.tqdm(test_data['debugging_tests']):
----> 3     result = tsp_dp(dist_arr)
      5     assert set(result) == set(range(len(dist_arr))), f"Your output does not visit all cities"
      6     student_length = validate_tour(result, dist_arr)

Cell In[4], line 12, in tsp_dp(dist_arr)
      2 """Compute the exact solution to the TSP using dynamic programming and returns the optimal pat
      3 
      4 Args:
   (…)
      8     List[int]: A list of city indices representing the optimal path.
      9 """
     10 # BEGIN SOLUTION
     11 # Main Algorithm
---> 12 raise NotImplementedError()
     13 C = {}
     14 prev = {}

NotImplementedError:
```

```
In [ ]: grader.check("TSP")
```

1.2 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

```
In [ ]: grader.export(pdf=False, force_save=True)
```