1 Dynamic Programming (Part 2)

In this notebook, we'll implemement dynamic programming algorithms for some more problems: Weighted Independent Set in a Tree and TSP.

If you're using Datahub:

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

If you're running locally:

- 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 [19]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

In [1]:
```
import otter
assert (otter.__version__ >= "4.4.1"), "Please reinstall the requirements and restart your kern

grader = otter.Notebook("dp2.ipynb")
import pickle
import numpy.random as random
import random
import tqdm
import time
from autograder_utils import is_independent_set, validate_tour, handle_timeout

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

rng_seed = 42
```

1.1 Q1. Weighted Independent Set in a Tree

In class, we saw an algorithm to solve the maximum independent set on trees. For a quick refresher, you can consult https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=21.

In this problem, we'll add some twists: now, suppose that each vertex i has some weight $w[i]$, and we want to find the independent set with the maximum total weight. **Also, you must implement your algorithm with a top-down approach.**

As before, the algorithm discussed in lecture and the textbook only returns the size of the maximum independent set. However, here we want you to return the list of vertices in the acual independent set. To find the actual independent set, it may be useful to maintain an array seperate from the DP array which can help us backtrack to reconstruct the actual set (simply storing the entire set so far at each entry in the DP table is too expensive and will cause the autograder to fail).

Hint: To compute a node's children and grandchildren, feel free to re-use your code from previous homeworks.

```
In [28]: def max_independent_set(adjacency_list, weights):
             """
             Return a list containing the vertices in the maximum weighted independent set.

             args:
                 adjacency_list:ListList[[int]] = the
                 weights:List[int] = a list of vertex weights. weights[i] is the weight of vertex i.

             return:
                 List[int] Containing the labels of the vertices in the maximum weighted independent se
             """
             # BEGIN SOLUTION
             n = len(weights)
             if n == 0:
                 return []

             # DFS post number computation (modified from HW4)
             def get_post(adj_list):
                 time, post, visited = 1, {}, set()
                 unvisited = lambda x: x not in visited
                 def explore(u):
                     nonlocal time
                     visited.add(u)
                     time += 1
                     for v in filter(unvisited, adj_list[u]):
                         explore(v)
                     post[u] = time
                     time += 1
                 for i in filter(unvisited, range(len(adj_list))):
                     explore(i)
                 return post

             post = get_post(adjacency_list)

             # helper function to get the children of node u based on DFS post numbers
             def get_children(u):
                 yield from filter(lambda v: post[v] < post[u], adjacency_list[u])

             def get_grandchildren(u):
```

```
    for v in get_children(u):
        yield from get_children(v)

# initialize the DP array
dp = [None] * n

def independent_set_helper(u):
    if dp[u] is None:
        # dp[u] includes the weight of the subtree as seen in class, as well as a flag
        # indicating whether u is included in the independent set. This will be useful
        # for reconstructing the set as it will save us from recomputing this quantity
        # on the fly.
        choices = (
            (weights[u] + sum(independent_set_helper(v)[0] for v in get_grandchildren(u)),
            (sum(independent_set_helper(v)[0] for v in get_children(u)), False),
        )
        dp[u] = max(choices)
    return dp[u]

# compute the DP values
# root doesn't really matter, we choose 0 for ease
independent_set_helper(0)

# reconstruct the sequence starting from the root, checking whether each node is included,
# and adding it to the independent set if it is, and recursing on its children/grandchildr
independent_set = []
to_check = [0]

while to_check:
    u = to_check.pop()
    if dp[u][1]:
        # u is included in the independent set, so its children are not.
        # add u to the independent set and check its grandchildren
        independent_set.append(u)
        to_check.extend(get_grandchildren(u))
    else:
        # u is not included in the independent set, so check its children.
        to_check.extend(get_children(u))

return independent_set
# END SOLUTION
```

1.1.1 Debugging

A simplified 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.

Note: your solution should not take inordinate amounts of time to run. Our implementation runs on all local test cases in <1s, and your implementation should run in similar time.

Points: 3

```
In [33]: for adj_list, weights, expected_weight in tqdm.tqdm(test_data['q1.1']):
             try:
                 result = max_independent_set(adj_list, weights)
             except Exception as e:
                 raise Exception(f"Your function threw an error (see above)") from e
             assert is_independent_set(adj_list, result), f"Your output is not an independent set"
             assert sum(weights[i] for i in result) == expected_weight, f"Your output is not a maximum
```

```
100%|      | 100/100 [00:00<00:00, 4684.28it/s]
```

```
In [ ]: grader.check("independent set")
```

1.2 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 Lecture 14 or https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=20.

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

A note on implementation: 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.

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.

As with before, 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]:
```python
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
    n = len(dist_arr)
    dp = {}
    prev = {}

    def tsp_helper(S, i):
        if (S, i) in dp:
            return dp[(S, i)]

        if S == frozenset():
            return dist_arr[0][i]  # Return to the starting city

        min_cost = float('inf')
        prev_city = None
        for city in S:
            cost = dist_arr[i][city] + tsp_helper(S.difference({city}), city)
            min_cost, prev_city = min((min_cost, prev_city) , (cost, city))

        dp[(S, i)] = min_cost
        prev[(S, i)] = prev_city
        return min_cost

    best_distance = tsp_helper(frozenset(range(1, n)), 0)

    # Backtracking to reconstruct tour
    S = frozenset(range(1, n))
    city = 0  # start at the origin
    tour = [0]

    # print(prev)

    while S:
        city = prev[(S, city)]
        tour.append(city)
        S = S.difference({city})

    return tour
    # END SOLUTION
```

1.2.1 Debugging

A simplified 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 [63]: # tests on very small cases
         for dist_arr, expected_distance in tqdm.tqdm(test_data['q2.1']):
             try:
                 result = tsp_dp(dist_arr)
             except Exception as e:
                 raise Exception(f"Your function threw an error (see above)") from e
             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"
```

```
100%|      | 20/20 [00:10<00:00,  1.94it/s]
```

```
In [ ]: # tests on slightly larger cases (these might take a while to run)
        for dist_arr, expected_distance in tqdm.tqdm(test_data['q2.2']):
            try:
                result = tsp_dp(dist_arr)
            except Exception as e:
                raise Exception(f"Your function threw an error (see above)") from e
            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"
```

```
 96%|      | 48/50 [01:03<00:02,  1.33s/it]
```

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

1.3 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)
```