3

Assignment 3: Dynamic Programming

CSDS 310: Algorithms

1

We are given n types of coin denominations with integer values $v_1, v_2, \ldots, v_n$. Given an integer t, we would like to compute the minimum number of coins to make change for t. We know that one of the coins has value 1, so we can always make change for any amount of money t.

For example, if we have coin denominations of 1, 2, and 5, then the optimal solution for $t = 9$ is $5, 2, 2$.

✓ **Answer**

Pseudocode

```
procedure coinChange(coins, amount)
        memo ← {0: (0, [])}
        return coinChangeAux(coins, amount, memo)

procedure coinChangeAux(coins, amount, memo)
        if amount in memo
                return memo.get(amount)
        ret ← min(
                [coinChangeAux(coins, amount - coin, memo), coin]
                for coin
                in coins,
                x → x[1][1]
        )
        ret[1][1] ← ret[1][1] + 1
        ret[1][2].append(ret[2])
        memo.set(amount, ret[1])
        return ret[1]
```

Subproblem Definition

I made my substructures by removing the first coin and setting the rest of the remaining amount as the substructure.

Optimal Substructure:

Say S is an optimal solution for coins c and amount a.
We claim that since S is optimal, its subproblems (S_1, S_2) must also be optimal.

Assume that S's subproblems are not optimal.

If S_1 or S_2 is non-optimal, then there must exist an optimal solution S_1^* or S_2^* such that $S_1^* < S_1$ or $S_2^* < S_2$ since optimal in this case means finding the least number of coins.

Since the minimum number of coins S is going to be the sum of its subproblems, $S = S_1 + S_2$, if the subproblems are suboptimal, $S > S_1^* + S_2$ or $S > S_1 + S_2^*$, so S is not optimal, which is a contradiction of our given.

Therefore, our assumption must be false and S's subproblems must be optimal.

Runtime:

$\theta(an)$

The auxilary loop takes $\theta(n)$ time to execute. This loop must only execute for non-memoed amounts. We may maximally run the loop a times for the amount of value we are looking for.

2

Given two strings $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$, the edit distance between X and Y is defined as the minimum number of edit operations (replacement, insertion, or deletion of a character) required to convert X to Y.

For example, the edit distance between $X = \text{esteban}$ and $Y = \text{stephen}$ is 4, comprising of 1 deletion (e), 1 insertion (h), and 2 replacements ($b \rightarrow p$ and $a \rightarrow e$). We would like to compute the edit distance between two given strings.

✓ **Answer**

Pseudocode

```
procedure stringDistance(str source, str target) → int, str[]
        m ← length(source)
        n ← length(target)

        prioQueue ← new PriorityQueue((0, 0, 0, [], source))
        visited ← new Set((0, 0))

        while prioQueue not empty
                distance, i, j, path, previous ← prioQueue.pop()

                // Reached the target
                if i == m and j == n
                        return distance, path

                // Insertion
                if j < n
                        nextPos ← (i, j + 1)
                        if nextPos not in visited
                                visited.add(nextPos)
                                newStr ←
                                        previous[0:j] +
                                        target[j] +
                                        previous[j:]
                                newOperations ← [ ... operations,
                                        `Insert ${target[j]} at
${j}\

                                         to make ${newStr}$`
                                ]
                                prioQueue.add((
                                        distance + 1,
                                        ... nextPos,
                                        newOperations,
                                        newStr
                                ))

                // Deletion
                if i < m
                        nextPos ← (i + 1, j)
                        if nextPos not in visited
                                visited.add(nextPos)
                                newStr ←
                                        previous[0:i] +
                                        previous[i+1:]
                                newOperations ← [ ... operations,
                                        `Delete ${source[i]} at
${i}\

                                         to make ${newStr}$`
                                ]
                                prioQueue.add((
                                        distance + 1,
                                        ...nextPos,
                                        newOperations,
                                        newStr
                                ))

                // Replacement
                if i < m and j < n
                        nextPos ← (i + 1, j + 1)
                        if nextPos not in visited
                                visited.add(nextPos)
                                if source[i] == target[j]
                                        prioQueue.add((
                                                distance,
                                                ...nextPos,
                                                operations,
                                                previous
                                        ))
                                else
                                        newStr ←
                                                previous[0:i] +
                                                target[j] +
                                                previous[i+1:]
                                        newOperations ←
[...operations,
                                                `Replace
${source[i]} with ${target[j]}\
                                                 at ${j} to make
${newStr}$`
                                        ]
                                        prioQueue.add((
                                                distance + 1,
                                                ...nextPos,
                                                newOperations,
                                                newStr
                                        ))
```

Subproblem Definition

We define our subproblems as the previous step to get to the current state, where it could be an insertion, removal, or replacement.

Optimal Substructure

Say S is an optimal solution for the operations required to get from $source$ to $target$. We claim that since S is optimal, its prior subproblems (S_1, S_2, S_3) must also be optimal. Getting from a prior subproblem is a single or no operation movement, and thus cannot be more optimal.

Assume that S's subproblems are not optimal.

If S_n is non-optimal, then there must exist an optimal solution S_n^* such that $S_n^* < S_n$ since optimal in this case means finding the least number of operations.

Since the minimum number of operations S is going to be the minimum of its subproblems plus one or none, $S = S_n + c$, if the subproblems are suboptimal we may construct S_n^* such that $S_n^* < S_n$. If S_n is the minimum of the subproblems, then we may construct a new $S^* = S_n^* + c$.

$$S_n^* < S_n \implies S_n^* + c < S_n + c \implies S^* < S$$

Therefore, our assumption must be false and S's subproblems must be optimal.

Runtime

$\theta(n^2)$
Where n is the length of the longer string. As we are traversing $(n+1)(m+1)$ space to find the solution, and at worse we traverse the entire space.

Example Implementation

```python
import heapq

def edit_distance_with_changes(X, Y):
    m = len(X)
    n = len(Y)

    pq = []

    heapq.heappush(pq, (0, 0, 0, [], X))

    visited = set()
    visited.add((0, 0))

    while pq:
        dist, i, j, operations, prev = heapq.heappop(pq)
        # Reached the target
        if i == m and j == n:
            return dist, operations

        # Insertion
        if j < n:
            new_state = (i, j + 1)
            if new_state not in visited:
                visited.add(new_state)
                new_str = prev[0:j] + Y[j] + prev[j:]
                new_operations = operations + [f"insert '{Y[j]}' at
{j}: {new_str}"]
                heapq.heappush(pq, (dist + 1, i, j + 1,
new_operations, new_str))

        # Deletion
        if i < m:
            new_state = (i + 1, j)
            if new_state not in visited:
                visited.add(new_state)
                new_str = prev[0:i] + prev[i+1:]
                new_operations = operations + [f"delete '{X[i]}' at
{i}: {new_str}"]
                heapq.heappush(pq, (dist + 1, i + 1, j,
new_operations, new_str))

        # Replacement
        if i < m and j < n:
            new_state = (i + 1, j + 1)
            if new_state not in visited:
                if X[i] == Y[j]:
                    heapq.heappush(pq, (dist, i + 1, j + 1,
operations, prev))
                else:
                    new_str = prev[0:i] + Y[j] + prev[i+1:]
                    new_operations = operations + [f"replace
'{X[i]}' with '{Y[j]}' at {j}: {new_str}"]
                    heapq.heappush(pq, (dist + 1, i + 1, j + 1,
new_operations, new_str))

# Example usage
X = "esteban"
Y = "stephen"
distance, changes = edit_distance_with_changes(X, Y)
print("Edit distance:", distance)
print("Changes made:")
for change in changes:
    print(change)
```

```
Edit distance: 4
Changes made:
delete 'e' at 0: steban
insert 'p' at 3: stepban
replace 'b' with 'h' at 4: stephan
replace 'a' with 'e' at 5: stephen
```

3

We are given n currencies and an exchange rate r_{ij} for any pair of currencies i and j. Namely, if we exchange 1 unit of currency i with currency j, we receive r_{ij} units of currency j . If we are given a source currency s and a target currency t, then we can go through a path of different currencies to reach t from s so as to maximize our profit. The markets can also charge an exchange fee depending on the number of exchanges we make. For example if the exchange fee is $f(k)$ for making k exchanges and we start with 1 unit of currency s, then the path of exchanges $s \to t$ will yield $r_{st} - f(1)$ units of currency t, whereas the path of exchanges $s \to i \to j \to t$ will yield $r_{si} \times r_{ij} \times r_{jt} - f(3)$ units of currency t. The problem is to find the sequence of exchanges that will maximize the amount of target currency t we can obtain for a given source currency s.

✓ **Answer**

Pseudocode

```
procedure findOptimalPath(r, f, i, j, max_iterations)
        start ← r[i]
        max_track ← Array.repeat(i, length(start)).transpose
        maxes ← start.transpose - f(1)
        for i in 0 … max_iterations - 1
                next ← r * start.transpose
                start, max_row ← next.maximum(direction=0)
                max_track ← max_track.append(max_row.transpose)
                maxes ← max.append(start.transpose)

        max_vals, max_index ← maxes.maximum(direction=1)
        final_track ← [j]
        max_track ← max_track.transpose.vflip
        final_track.unshift(max_track[i, final_track[0]])
                for i
                in 0 … max_index[j]+1
        return max_vals[j], final_track
```

Subproblem definition

We define each subproblem as the optimal path to get to all previous currencies in one less step or less.

Optimal Substructure

Say S is an optimal solution for the conversions required to get from currency i to currency j.
We claim that since S is optimal, its prior subproblems S_n must also be optimal. Where a subproblem is defined as the optimal path to get to all possible currencies in one less step. Getting from a prior subproblem to the final solution is a single or no operation movement, and thus cannot be more optimal, where the final solution is defined as one conversion from any prior subproblem, or no conversion from the subproblem corresponding to the target currency.

Assume that S's subproblems are not optimal.

If S_n is non-optimal, then there must exist an optimal solution S_n^* such that $S_n^* > S_n$ since optimal in this case means finding the most money out of all conversions.

Since the maximum conversion S is going to be the maximum of its subproblems multiplied by conversion minus the conversion costs, $S = S_n r_{n,j} + f(c)$, if the subproblems are suboptimal we may construct S_n^* such that $S_n^* > S_n$. If S_n is the maximum of the subproblems, then we may construct a new $S^* = S_n^* r_{n,j} + f(c)$.

$$S_n^* > S_n \implies S_n^* r_{n,j} + f(c) > S_n r_{n,j} + f(c) \implies S^* > S$$

Therefore, our assumption must be false and S's subproblems must be optimal.

Caveats

We cannot assume much about the problem as not much was explicitly given to us for this problem.
For one, we do not know about the structure of our cost function f at all, not even if it is strictly increasing or not. For the sake of now knowing, we make no assumptions about f in my implementation.
For two, we cannot know for sure if there are indefinite loops possible within our conversion matrix r where infinite money is allowed to be made, since we also do not know if the cost function f is positive or non-zero at all, thus we must implement a depth limit to prevent an infinite recursion from occurring.
Lastly, we cannot assume is loops are permitted at all, but their disallowance would break the issue of optimal substructure, so we assume that they are allowed.

Runtime

$O(maxIter * n^2)$ as we perform a scalar matrix multiplication every iteration we go.

Example Implementation

```
import numpy as np

def find_optimal_path(r, f, i, j, max_iter=100):
    '''
    Get from currency i to currency j with exchange cost of f(n)
where n is the number of steps.
    r[a, b] represents the exchange rate from currency a to b
    '''
    # Begin with first conversion from initial currency
    start = r[i]
    # The pathes for each target currency for each step are stored
in this array
    max_track = np.array([i] * start.shape[0]).reshape(-1, 1)
    # The max value for each currency for each step is stored in
this array
    maxes = np.array(start.reshape(-1, 1) - f(1))
    # The pathes for the max at each step is stored in this array

    for i in range(max_iter-1):
        # Calculate converting to all other currencies
        nex = r * start.reshape(-1, 1)
        # Pick the maximum conversion for each currency
        max_row = nex.argmax(axis=0)
        start = nex[max_row, np.arange(nex.shape[1])]
        # Track the path to get to the maximum conversion
        max_track = np.hstack((max_track, max_row.reshape(-1, 1)))
        maxes = np.hstack((maxes, start.reshape(-1, 1) - f(i+1)))

    maxInd = maxes.argmax(axis=1)
    maxVals = maxes[np.arange(maxes.shape[0]), maxInd]
    final_track = [j]
    max_track = np.flipud(max_track.T)
    [final_track.insert(0, max_track[i, final_track[0]]) for i in
range(maxInd[j] + 1)]
    return maxVals[j], np.array(final_track)

# Example
r = np.array([
    [1,     2,      3,      4],
    [1/2,   1,      6,      7],
    [1/3,   1/6,    1,      8],
    [1/4,    1/7,     1/8,     1]
])

f = lambda x: x+1

v, t = find_optimal_path(r, f, 0, 3)

print(v)
print(t)
```