CSCI 270 Homework 5

Qilin Ye

October 17, 2022

1. Chutes & Ladders

Solution. The algorithm goes as follows.

```
def hw5_1(n, ladders, chutes):

 # initializing memoization array
 dp = [float('inf') for _ in range(n+1)]
 pred = [float('inf') for _ in range(n+1)]

 dp[0] = 0

 # base case 0 to 5: if not snake, set dp[i] = 0, otherwise no change
 for i in range(6):
     if i not in chutes:
         dp[i], pred[i] = 1, 0
         if i in ladders:
             dp[ladders.get(i)], pred[ladders.get(i)] = min(dp[ladders.get(i)], dp[i]+1), i
 # inductive step:
 # if chutes: no change
 # otherwise, for dp[i]: find the min of dp and update pred accordingly
 #   if ladder, update pred of destination too
 for i in range(6, n+1):
     if i in snakes and pred[i] != float('inf'):
         # this can happen if and only if pred[i] is a ladder, so square i can still be reached via a ladder
         dp[i] = dp[pred[i]] + 1
     # a non-chute square i can be reached either via a ladder or a roll
     elif i not in snakes:
         # find the min value of the previous 6 squares, along with index
         roll_opt_val = dp[i-1]
         roll_opt_index = i-1
         for j in range(2, 7):
             if dp[i-j] < roll_opt_val:
                 roll_opt_val = dp[i-j]
                 roll_opt_index = i-j
         # if roll_opt_val + 1 < dp[i] then the optimal solution finishes with a roll
         # otherwise, check if currently stored dp[i] ends with two ladder climbs. If yes, invalid. If not,
              ladder climb is better
         if roll_opt_val + 1 < dp[i] or pred[pred[i]] in ladders:
             dp[i] = roll_opt_val + 1
             pred[i] = roll_opt_index
         # prepare for future ladder if needed
```

```
38            if i in ladders:
39                dp[ladders.get(i)], pred[ladders.get(i)] = min(dp[ladders.get(i)], dp[i]+1), i
40            # if six consecutive unreachable squares, break early
41            if dp[i] == float('inf'):
42                print("No solution exists")
43                return
44
45    backtrack = [n]
46    while backtrack[-1] != 0:
47        backtrack.append(pred[backtrack[-1]])
48    print("One optimal solution:", dp[n], "steps:", end = ' ')
49    print(', '.join(map(str, backtrack[::-1]))) # can return backtrack[::-1]
```

Proof of correctness. We claim that each `dp[i]` represents the minimum number of die rolls one needs to get to square i, with ∞ being unreachable (because of the chutes).
The base cases are simple. For $k = 0$, the claim is vacuously true. For $i = 1, ..., 6$, as long as there is no chute, we can obviously reach tile i in one roll. Furthermore, if square i is a ladder, we can then reach square `d[i]` in *at most* two steps using this ladder. (Note the `min` function: if for example square 2 has a ladder of length 3 and square 5 is not a chute, then the optimal solution for getting to 5 is in fact without the ladder.)

For the inductive step, we use strong induction and assume for $k > 6$, `dp[k-j]` outputs the correct minimum number of rolls to get to $k - j$, where $1 \leqslant j \leqslant 6$. Now, there are a few options to get to tile k. First, if k is a chute, the only valid way is if k is the end point of some ladder. In this case `dp[k]` is 1 plus `dp[pred[k]]`, as shown in line 22. If k is not a chute, there are two ways:

- From one of the squares $k - j$ as stated above. Fix any j. If square $k - j$ requires a minimum of `dp[k-j]` roles, square k requires a minimum of `dp[k-j]` $+1$ rolls using this path. Therefore, without a ladder, square k requires a minimum of

$$\min_{1 \leqslant j \leqslant 6} \texttt{dp[k-j]} + 1 \qquad (*)$$

  rolls to be reached. This, along with backtracking, is precisely what lines 29 to 31 do.

- Alternatively, if k is a ladder destination, we can also reach it via a ladder climb, *with the extra constraint that the starting square of the ladder is not the destination of another ladder*. Since we iterate through the indices in an increasing order, and every ladder has a positive length, before we visit the destination i of any ladder, we will have stored the starting square in the corresponding `pred[i]`. Thus, k is a ladder destination with a starting square that is also the destination of another ladder if and only if `pred[pred[k]]` is in `ladders`.

  If `pred[pred[k]]` is in `ladders`, then the optimal solution to square k cannot end with a ladder jump, and we simply resort the previous case.

  If not, we then need to compare the solution ending with a die roll against that with a ladder jump and take store the more optimal one in `dp[k]`. This sums up the usage of lines 34 to 36.

We now check termination conditions. Line 43 can only be reached if (i) the corresponding square is *not* a ladder destination (previous case), and (ii) all its six predecessors have infinite value. Therefore, this square is

unreachable, and we terminate the program, prompting that any later square, including n, cannot be reached. If line 43 is never executed, then the program finishes after i hits $n+1$, which is in finite time.
Finally, since the backtracking solution has precisely `dp[n]` steps and is a valid solution, it is therefore optimal.

For runtime, the code above has assumed `ladders` and `chutes` are both hash maps. Under this structure, the algorithm iterates over $\mathcal{O}(n)$ loops and in each loop performs $\mathcal{O}(1)$ amortized work. The total work is therefore $\mathcal{O}(n)$. To completely remove the concern regarding $\mathcal{O}(\log n)$ worst case operations for hash tables, we can instead store `ladders` and `chutes` as arrays, trading space complexity for a guaranteed $\mathcal{O}(1)$ lookup and therefore guaranteed $\mathcal{O}(n)$ overall runtime. □

2. Affine Cost Edit Distance

Solution. The algorithm goes as follows.

```python
def hw5_2(x, y, A, B, c):
    # string x has length n, string y has length m
    # mismatch cost = B
    # insertion cost = A + c (k-1)
    n, m = len(x), len(y)

    dp = [[0 for _ in range(m+1)] for _ in range(n+1)]
    dp_insert = [[float('inf') for _ in range(m+1)] for _ in range(n+1)]
    dp_delete = [[float('inf') for _ in range(m+1)] for _ in range(n+1)]
    dp_other = [[float('inf') for _ in range(m+1)] for _ in range(n+1)]

    # base case (0,0)
    dp_insert[0][0] = dp_delete[0][0] = dp_other[0][0] = 0

    # if c > A, it is cheaper to do multiple single-character deletions
    c = min(c, A)
    # more base cases
    for i in range(1, n+1):
        dp[i][0] = dp_delete[i][0] = A + c*(i-1)
    for j in range(1, m+1):
        dp[0][j] = dp_insert[0][j] = A + c*(j-1)

    # DP begins here
    for i in range(1, n+1):
        for j in range(1, m+1):
            dp_other[i][j] = dp[i-1][j-1] + (B if x[i-1] != y[j-1] else 0)

            dp_insert[i][j] = min(
                dp_insert[i][j-1] + c,
                dp_delete[i][j-1] + A,
                dp_other[i][j-1] + A)
            dp_delete[i][j] = min(
                dp_delete[i-1][j] + c,
                dp_insert[i-1][j] + A,
                dp_other[i-1][j] + A)

```

```
        # take min of dp_insert, dp_delete, dp_other, break ties in this order to ensure the min and the
            predecessor (used in backtracking) agree
        if dp_insert[i][j] <= dp_delete[i][j] and dp_insert[i][j] <= dp_other[i][j]:
            dp[i][j] = dp_insert[i][j]
            dp_pred[i][j] = 'insert'
        elif dp_delete[i][j] <= dp_insert[i][j] and dp_delete[i][j] <= dp_other[i][j]:
            dp[i][j] = dp_delete[i][j]
            dp_pred[i][j] = 'delete'
        elif dp_other[i][j] <= dp_insert[i][j] and dp_other[i][j] <= dp_delete[i][j]:
            dp[i][j] = dp_other[i][j]
            dp_pred[i][j] = '270 fun'

    i, j = n, m
    backtrack = []
    while i > 0 and j > 0:
        if dp_pred[i][j] == 'insert':
            backtrack.append(f'insert {y[j-1]}')
            j -= 1
        elif dp_pred[i][j] == 'delete':
            backtrack.append(f'delete {x[i-1]}')
            i -= 1
        else:
            if x[i-1] != y[j-1]: backtrack.append(f'replace {x[i-1]} with {y[j-1]}')
            else: backtrack.append('no action')
            i, j = i-1, j-1

    while i > 0:
        backtrack.append(f'delete {x[i-1]}')
        i -= 1
    while j > 0:
        backtrack.append(f'insert {y[j-1]}')
        j -= 1
    return backtrack[::-1]
```

Sample output:

- Parameters `'cb','abcd',100,15,1`: `['replace c with a', 'no action', 'insert c', 'insert d']`

Setting up recurrence relation. Given any sequence of edit operations, it either ends with (i) an insertion, (ii) a deletion, or (iii) others, namely nothing or a mismatch replacement. To this end, we partition all sequences of operations aligning $x[1,...,i]$ and $y[1,...,j]$ as described above. More concretely, we keep track of four $(n+1) \times (m+1)$ arrays:

- `dp[i][j]` records the edit distance between $x[1,...,i]$ and $y[1,...,j]$.
- `dp_insert[i][j]` records the (minimum) edit distance between $x[1,...,i]$ and $y[1,...,j]$, with the extra constraint that the last operation done is an insertion.
- `dp_delete[i][j]`, on the other hand, requires the last operation to be a deletion.
- `dp_other[i][j]` requires the last operation to be either nothing (if $x[i] = y[j]$) or a replacement (if $x[i] \neq y[j]$).

By construction, it should become clear that `dp[i][j]` is the minimum of the other three.
Now consider the update rules for the three auxiliary tables.

- If the last operation of aligning $x[1,...,i]$ and $y[1,...,j]$ is an insertion, the penultimate operation is performed on $x[i]$ and $y[j-1]$. Again, three possibilities:
  - If the penultimate operation is also an insertion, then the last operation (on i,j) incurs a cost of c, so the new cost is `dp_insert[i][j-1] + c` as shown in line 29.
  - If the penultimate operation is not an insertion, then the extra cost is A. If it is a deletion then the new cost is `dp_delete[i][j-1] + A`; otherwise the new cost is `dp_other[i][j-1] + A`.
  - Take minimum. This correpsonds to lines 28 to 31.
- Similarly, if the last operation on (i,j) is a deletion, the extra cost compared to $(i-1,j)$ will be c if the penultimate operation is also a deletion, and A otherwise.
- If the last operation on (i,j) is a match or mismatch, then this means the strings $x[1,...,i-1]$ and $y[1,...,j-1]$ are already aligned prior to this last operation. Therefore, if $x[i] = y[j]$, there is no extra cost moving from `dp[i-1][j-1]` to `dp_other[i][j]`; otherwise, an extra cost B of replacing letters is incurred.
- As mentioned before, taking the minimum of `dp_insert[i][j]`, `dp_delete[i][j]`, and `dp_other[i][j]` gives us `dp[i][j]`.

Since we have exhausted all cases, we obtain the following recurrence relation:

$$\texttt{dp[i][j]} = \min \begin{cases} \text{(..., insert)} = \texttt{dp\_insert[i][j]} = \min \begin{cases} \text{(..., insert, insert)} = \texttt{dp\_insert[i][j-1] + c} \\ \text{(..., delete, insert)} = \texttt{dp\_delete[i][j-1] + A} \\ \text{(..., other, insert)} = \texttt{dp\_other[i][j-1] + A} \end{cases} \\ \text{(..., delete)} = \texttt{dp\_delete[i][j]} = \min \begin{cases} \text{(..., delete, delete)} = \texttt{dp\_delete[i-1][j] + c} \\ \text{(..., insert, delete)} = \texttt{dp\_insert[i-1][j] + A} \\ \text{(..., other, delete)} = \texttt{dp\_other[i-1][j] + A} \end{cases} \\ \text{(..., other)} = \texttt{dp\_other[i][j]} = \texttt{dp[i-1][j-1]} + \begin{cases} \texttt{B} & \text{if } \texttt{x[i] != y[j]} \\ 0 & \text{otherwise.} \end{cases} \end{cases}$$

Proof of correctness. We claim that `dp[i][j]` outputs the correct edit distance between $x[1,...,i]$ and $y[1,...,j]$. We further claim that the other three arrays store the minimum edit distances given their corresponding constraints.
For base cases, we set the $(0,0)$ entry of all four to be 0. For $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n+1$, we initialize row 0 and column 0 entries as follows:

`dp_insert[i][0]` $= \infty$	`dp_insert[0][j]` $= A + c(j-1)$
`dp_delete[i][0]` $= A + c(i-1)$	`dp_delete[0][j]` $= \infty$
`dp_other[i][0]` $= \infty$	`dp_other[0][j]` $= \infty$
`dp[i][0]` $= A + c(i-1)$	`dp[0][j]` $= A + c(j-1)$

This is because by construction, if x is an empty string and y is not, the only way to align x with y is by insertion, and conversely if y is empty and x is not, then the only way to align is by deletion. All other approaches are infeasible, hence ∞. Note that based on our implementation, before initializing, we set $c = \min(A, c)$, so that $A + c(i-1)$ will *always* be the minimum cost for deleting / inserting i characters in a row.
Therefore, when $i = 0$ or $j = 0$, all four tables satisfy the claim.

Now we induct on $i + j$ using strong induction (assuming $i, j \neq 0$). That is, if $i + j = k$, we assume the claim holds for all (i', j') with $i' + j' \leqslant k - 1$. Note that the first three lines of the recurrence relation are precisely implemented in lines 28 to 31, and the 4^{th} to 7^{th} are implemented in lines 33 to 36. If $x[i] = y[j]$, the recurrence relation skips the last case (i.e., "0 otherwise"), and so does line 26, which performs `dp_other[i][j] = dp[i-1][j-1] + 0`. On the other hand, if $x[i] \neq y[j]$, then the recurrence relation skips the second last line ($+B$) and line 26 writes `dp_other[i][j] = dp[i-1][j-1] + B`. This shows that our algorithm agrees with the recurrence relation, thhereby outputting the optimal solutions for `dp`, `dp_insert`, `dp_delete`, and `dp_other`, concluding the induction.
The correctness of backtracking is guaranteed by the correctness of the algorithm. Finally, the main loop has nm iterations, and in each one we perform constant time work. The initialization also takes $\mathcal{O}(nm)$ work, so the total time and space complexities are $\mathcal{O}(nm)$.

3. Chocolate - Domino Tiling

```
1  def i_dont_know_how_to_count(k, n):
2      if k * n % 2 == 1: return 0
3
4      # create dp[][][], dimension k * n * 2^(n+1)
5      # instead of dealing with 0-indexed arrays, I decide to simply pad it by making dp[][][] slightly larger. Index
           problems begone!
6      # main idea: traverse through the hallway, and for each tile, keep track of all possible tilings of all
           previous tiles such that
7          # (i) no 2x1 rectangles overlap
8          # (ii) all previous tiles are covered, and
9          # (iii) it does not matter if some 2x1 rectangles cover to-be-explored tiles, but they must lie within the
               hallway, i.e., not ouside the hallway's boundary
10     # detailed explanation (e.g. choice of mask and update rules) below the algorithm
11
12     max_mask = (1 << (k+1)) - 1
13     dp = [[[0 for _ in range(1 << (k+2))] for _ in range(n+1)] for _ in range(k+1)] # slightly extra spaces
14     dp[k][0][0] = 1 # base case: vacuously one way to tile a non-existent hallway
15
16     for j in range(1, n+1):
17         # the first for loop will execute every time we move on to a new column. In particular, it will carry all
               tiling information from tile[k][j-1] to tile[0][j] and prepare it for the first tile on column j,
               namely, tile[1][j] (recall 1-indexed arrays)
18         for mask in range(max_mask+1):
19             dp[0][j][mask << 1] += dp[k][j-1][mask]
20
21         # this is how we traverse through most of the hallway, and the bulk of our DP algorithm. Main idea: decide
```

```
            whether and/or how to place the next rectangle given the current configuration, which is jointly
            decided by coordinates (i,j), and the mask.
        for i in range(1,k+1):
            for mask in range(max_mask+1):
                NE, SW = mask & (1 << (i-1)), mask & (1 << i)
                if NE and SW: continue
                if NE and not SW: dp[i][j][mask ^ (1 << (i-1))] += dp[i-1][j][mask]
                elif SW and not NE: dp[i][j][mask ^ (1 << i)] += dp[i-1][j][mask]
                else:
                    dp[i][j][mask ^ (1 << (i-1))] += dp[i-1][j][mask]
                    dp[i][j][mask ^ (1 << i)] += dp[i-1][j][mask]

    return dp[k][n][0] # done!
```