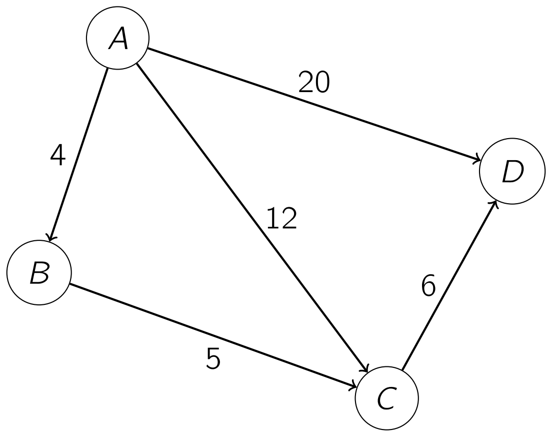
CS 161 (Stanford, Fall 2025) Section 6

1 Algorithm Practice

Given the directed graph below, run the Floyd-Warshall Algorithm, processing vertices in alphabetical order. Fill in the table below which keeps track of the shortest paths. Ordered pairs of vertices with no directed path (such as (B, A)) are omitted and their distance can be taken as ∞ for updates.



Solution

The completed table is shown below:

(A,A)	(A,B)	(A,C)	(A,D)	(B,B)	(B,C)	(B,D)	(C,C)	(C,D)	(D,D)
0	4	12	20	0	5	∞	0	6	0
0	4	12	20	0	5	∞	0	6	0
0	4	9	20	0	5	∞	0	6	0
0	4	9	15	0	5	11	0	6	0
0	4	9	15	0	5	11	0	6	0

2 Knapsack

Consider an instance of the knapsack problem with five items (where there is only one copy of each item):

Item	Value	Size
1	1	1
2	2	3
3	3	2
4	4	5
5	5	4

and knapsack capacity $C = 9$.

What are the final array entries of the Knapsack algorithm from lecture, and which items belong to the optimal solution?

Solution

With columns indexed by i and rows by c:

$c \backslash i$	0	1	2	3	4	5
9	0	1	3	6	8	10
8	0	1	3	6	8	9
7	0	1	3	6	7	9
6	0	1	3	6	6	8
5	0	1	3	5	5	6
4	0	1	3	4	4	5
3	0	1	2	4	4	4
2	0	1	1	3	3	3
1	0	1	1	1	1	1
0	0	0	0	0	0	0

Optimal items: $\{2, 3, 5\}$.

3 Encoding

Suppose we encode lowercase letters into a numeric string as follows: we encode a as 1, b as 2, . . . , and z as 26. Given a numeric string S of length n, develop an $O(n)$ algorithm to find how many letter strings this can correspond to. For example, for the numeric string 123, the algorithm should output 3 because the letter strings that map to this numeric string are abc, lc, and aw.

Solution

Intuitively, a one digit substring can be decoded to a letter if it's > 0, and a two digit substring can be decoded to a letter if it's between 10 and 26 (inclusive).
To turn this into a dynamic programming problem, we can count the number of ways to decode the substring ending at i. (ie the substring $S[0 : i + 1]$ in python notation). We start at $i = 0$ and build up.

If we know the number of ways to decode the substring ending at $k-1$ ($S[0:k]$), we can use that information to count the number of decodings for the substring ending at k ($S[0:k+1]$). If the last digit isn't zero, we can convert that to a letter directly, and check the array for the number of decodings for $S[0:k]$ to get the total number of decodings here. If the last two digits make a valid letter (between 10 and 26 inclusive), the total number of decodings using this option is equal to the total number of decodings for the substring $S[0:k-1]$. If both interpretations are valid, we add the number of decodings from the first case to the number of decodings for the second case to get the total number of decodings.
Let $f(i)$ denote the number of ways to encode the string up to and including $S[i]$ establish our recurrence as follows:

$$f(i) = \sum \begin{cases} f(i-1) & S[i] \in \{1, \ldots, 9\} \\ f(i-2) & S[i-1:i+1] \in \{10, \ldots, 26\} \end{cases}$$

To compute this in a single forward pass using dynamic programming, we build a table for each $f(i)$ and initialize base cases for $f(0)$ and $f(1)$, as shown below. We set up an array of size n and fill in each element, and return the last element. Because it takes time $O(n)$ to iterate through our array and $O(1)$ time to fill in a given array element, the total runtime of our algorithm is $O(n)$.

```python
def num_decodings(numeric_string):
    n = len(numeric_string)
    #  t[i] := how many possible decodings for s[:i]
    t = [0 for _ in range(n)]

    # base cases for 0 and 1
    t[0] = int(numeric_string[0])>0
    two_digit_num = int(numeric_string[:2])
    t[1] = 10 <= two_digit_num <= 26
    if int(numeric_string[1]) > 0:
        t[1] += t[0]

    for i in range(2, n):
        if int(numeric_string[i]) > 0:
            t[i] += t[i-1]

        two_digit_num = numeric_string[i-1:i+1]
        if 10 < two_digit_num <= 26:
            t[i] += t[i - 2]

    return t[-1]
```

4 Knight Moves

Given an 8×8 chessboard and a knight that starts at position $a1$, devise an algorithm that returns how many ways the knight can end up at position xy after k moves. Knights move ±1 squares in one direction and ±2 squares in the other direction. In other words, knights move in a pattern similar to a "L".

Note: on a chessboard, rows are labeled from 1-8 and columns are labeled from $a - h$.

Solution

We can store an array of the number of paths to each position after i moves, for each i. The base case is simple - after 0 moves, the knight has one way to end up in his starting position - not move at all!

If we know how many ways to get to each of the positions after round $i-1$, to get the number of ways they could move to some xy, we would add up the total number of ways they could have gotten to any of his last steps — 1 away in some direction and 2 away in the other. For example, there are 3 ways to get to $a2$ - either from $c1$, $c3$, or $b4$ - so we would simply add up the number of ways to get to each of these positions after $i-1$ steps to count how many ways to end up in position $a2$ after i steps.
Finally, once we compute the array for k, we can return position xy. (Notice that even though we only cared about position xy, we still computed the number of ways to get to any point on the chessboard in the previous steps - this is because these points could be on the path, despite not being the end point.)
More formally, we can define the function $f(x, y, i)$ to be the number of ways the knight can get to position xy in i moves. This gives us the following recurrence: (written out for clarity)

$$\begin{aligned} f(x,y,i) = &f(x-1,y-2,i-1) + f(x-1,y+2,i-1) + f(x+1,y-2,i-1) \\ &+ f(x+1,y+2,i-1) + f(x-2,y-1,i-1) + f(x-2,y+1,i-1) \\ &+ f(x+2,y-1,i-1) + f(x+2,y+1,i-1) \end{aligned}$$

(where the function evaluates to 0 if the position is off the board)

```
def knight moves(end_position, num_moves):
    # num_ways stores how many ways
    # to get to each reachable position
    num_ways = collections.defaultdict(int)
    num_ways[(0, 0)] = 1 #(0,0) corresponds to A1
    move_directions = [(1, 2), (1, -2), (-1, 2),
        (-1, -2), (2, 1), (2, -1), (-2, 1), (-2, -1)]

    for i in range(num_moves):
        new_num_ways = collections.defaultdict(int)
        for cur_row, cur_col in num_ways.keys():
            for move_row , move_col in move_directions:
                new_row = cur_row + move_row
                new_col = cur_col + move_col
                # check to make sure new position
                # stays within the board
                if new_row >= 0 and new_row < 8 and
                    new_col >= 0 and new_col < 8:
                     new_num_ways[(new_row, new_col)] +=
                         num_ways[(cur_row, cur_col)]
        num_ways = new_num_ways

    return num ways[ end position ]
```

Runtime and number of DP States: At each step i, the algorithm keeps track of the number of ways to reach every square on the 8×8 board. This means we have

$$64 \text{ positions} \times (k+1) \text{ layers} = 64(k+1)$$

total DP states, which is $O(k)$. Note that in practice we only keep track of the current 64 states, so we can reduce the number of states we keep down to a constant.

For each state (x, y, i), the recurrence considers up to 8 possible knight moves, each of which can be checked in $O(1)$ time. Thus, the total runtime is $O(64k \cdot 8) = O(k)$.