Algorithms and Analysis Report
COSC2123/3119
Dynamic Programming in Action: The Knapsack-Maze Challenge

Kien Hung Ly - s3977367

Assessment Type	Individual assignment. Submit online via GitHub.
Due Date	Week 11, Friday May 23, 8:00 pm. A late penalty will apply to assessments submitted after 11.59 pm.
Marks	30

1 Learning Outcomes

This assessment relates to four learning outcomes of the course which are:

- CLO 1: Compare, contrast, and apply the key algorithmic design paradigms: brute force, divide and conquer, decrease and conquer, transform and conquer, greedy, dynamic pro-gramming and iterative improvement;
- CLO 3: Define, compare, analyse, and solve general algorithmic problem types: sorting, searching, graphs and geometric;
- CLO 4: Theoretically compare and analyse the time complexities of algorithms and data structures; and
- CLO 5: Implement, empirically compare, and apply fundamental algorithms and data structures to real-world problems.

2 Overview

Across multiple tasks in this assignment, you will design and implement algorithms that navigate a maze to collect treasures. You will address both fully observable settings (where treasure locations are known) and partially observable ones (where treasure locations are unknown), which requires strategic exploration and value estimation when solving the maze. Some of the components ask you to critically assess your solutions through both theoretical analysis and controlled empirical experiments to encourage reflection on the relationship between algorithm design and real-world performance. The assignment emphasizes on strategic thinking and the ability to communicate solutions clearly and effectively.

I certify that this is all my own original work. If I took any parts from elsewhere, then they were non-essential parts of the assignment, and they are clearly attributed in my submission. I will show I agree to this honor code by typing "Yes": Yes

Task B: Maximising Knapsack using Dynamic Programming

Algorithm 1 DynamicKnapsack($items$, C, N)

```
    Input:
$items$: list of tuples (name, weight, value),
$C$: knapsack capacity,
$N$: number of items
    Output:
$L_{\text{opt}}$: list of selected item names,
$w_{\text{opt}}$: total weight of selected items,
$v_{\text{opt}}$: total value of selected items

$dp[0 \dots N][0 \dots C] \leftarrow$ None
$dp[0][0 \dots C] \leftarrow 0$                          ▷ First row (0 capacity): 0 value
$L_{\text{opt}} \leftarrow \emptyset$, $w_{\text{opt}} \leftarrow 0$, $v_{\text{opt}} \leftarrow 0$

function TOPDOWN($i, c$)                                   ▷ $i$ for index, $c$ for capacity
    if $dp[i][c] \neq$ None then
        return $dp[i][c]$
    else if $i = 0$ or $c = 0$ then
        $output \leftarrow 0$
    else if $items[i-1]$.weight $> c$ then
        $output \leftarrow$ TOPDOWN($i-1, c$)
    else
        $inc \leftarrow$ TOPDOWN($i-1, c - items[i-1]$.weight) $+ items[i-1]$.value
        $exc \leftarrow$ TOPDOWN($i-1, c$)
        $output \leftarrow \max(inc, exc)$
    end if
    $dp[i][c] \leftarrow output$
    return $output$
end function

$v_{\text{opt}} \leftarrow$ TOPDOWN($N, C$)
$i \leftarrow N$, $c \leftarrow C$

while $i > 0$ and $c > 0$ do
    if $dp[i][c] \neq dp[i-1][c]$ then
        Append $items[i-1]$.name to $L_{\text{opt}}$
        $w_{\text{opt}} \leftarrow w_{\text{opt}} + items[i-1]$.weight
        $c \leftarrow c - items[i-1]$.weight
    end if
    $i \leftarrow i - 1$
end while

return $L_{\text{opt}}, w_{\text{opt}}, v_{\text{opt}}$
```

Benefit: Dynamic programming approach in this task most of the time is faster than the recursive approach in task A since it has lower time complexity than recursive approach (O(N*C) and O(2^N)). The only scenario where a recursive approach can outperform dynamic programming is with a very low number of items.

Downside: Dynamic programming will take more memory space compared to recursive approach since it has to make a table that is used by the memory function to store data to solve the knapsack problem.

Task C: Analysis of the Complete Problem

1a.
Recursive knapsack (Task A): Time Complexity: O(2^n), where n is `num_items`

Justification: For each item, the algorithm makes two choices: either take or leave the item. This leads to a recurrence relation:

T(n, c) = T(n-1, c) + T(n-1, c-w) + O(1), where c is capacity and w is the weight of current treasure.

In the worst case, the algorithm explores all possible subsets of items. If there are n items, there are 2^n possible subsets. Each recursive call involves constant time operations (additions, subtractions, comparisons, etc…). Therefore, the time complexity is proportional to the number of nodes in the decision tree, leading to O(2^n).

Dynamic knapsack - Top Down Approach (Task B): Time Complexity: O(n*c) where n is `num_items` and c is `capacity`.

Justification: This approach uses memoization through a table size of (n+1)*(c+1). Since the **dp** table has (n+1)*(c+1) cells, the `topDown` function calculates each cell `dp[i][c]` once. Each calculation involves a few comparisons, additions, and two recursive calls (in the worst case) to `topDown` for cells that would have already been computed or will be computed once. Thus, filling the table takes O(n*c) time.

1b. findItemsAndCalculatePath

This function have 2 steps:

1. `knapsack.solveKnapsack(maze, csvFilename):` This will call the knapsack algorithms (either task A or B).
2. `solver.solveMaze(maze, entrance, exit):` This will do the pathfinding part.

For the partfinding part, let k be the number of optimal items/cells, and add 2 more cells, one for entrance and one for exit then we have k + 2 cells that we have to visit.

The code runs BFS between all pairs of these k + 2 cells. The number of pairs is (k + 2)(k + 1) so there are approximately (k+2)^2 BFS calls. Each BFS call on the maze takes O(R*C), R here means the total number of rows and C means the total number of columns in the maze. Since in the worst case, each BFS's solution will fill up every cell in the maze, which is equal to R*C. Therefore, this step is O((k + 2)^2 * R*C).

The code then goes through all permutations of the k knapsack cells to find the optimal order. There are k! permutations. And for each permutation, it calculates the length of the path by adding the length between each point of interest, and this calculation takes O(k) time. Therefore this part's complexity is O(k! * k).

Therefore, the time complexity of `solveMaze` is: O((k+2)^2 * R*C + k! * k).

Combining with the knapsack solution, the overall complexity is:

- With recursive knapsack: O(2^n + (k+2)^2 * R*C + k! * k)
- With dynamic knapsack: O(n*c + (k+2)^2 * R*C + k! * k)

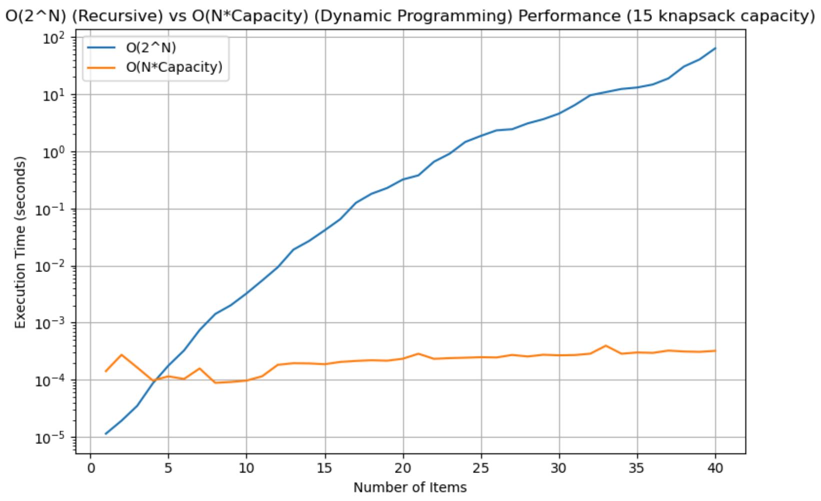
2. Empirical Design:

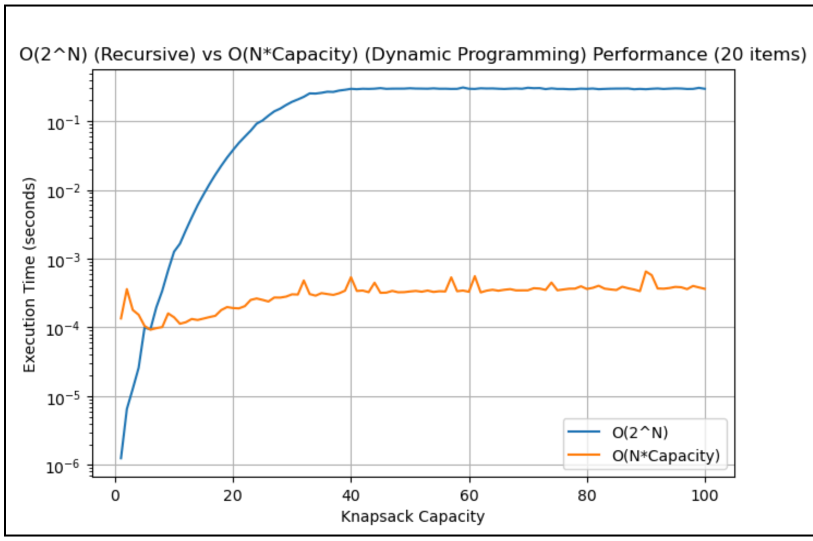
Key Variables for findItemsAndCalculatePath Complexity:

- **n (Number of items):** Affects 2^n (recursive) or n*c (DP), and potentially k.
- **c (Knapsack capacity):** Affects n*c (DP) and influences k.
- **k (Number of selected items):** Affects path-finding complexity through the k! permutations.
- **`R` and `C` (Maze size):** Affects BFS runtime.

Variables to Control/Ignore (for specific tests): Specific maze layout (beyond size), item weights/values (their impact is via k), entrance/exit locations.

3. Empirical Analysis:





“startGenTime: float = time.perf_counter()” and “endGenTime: float = time.perf_counter()” were used to time dynamic and recursive with a for loop increasing the number of Items/Capacity. I did not measure BFS but only dynamic and recursive since there are the only difference of 2 two methods in findItemsAndCalculatePath (BFS are the same).

In the left graph, the time between recursive and dynamic is measured with a fixed knapsack capacity and increasing number of items. With a very low number of items, it is clear that the recursive approach outperforms the dynamic approach since dynamic also scales with capacity. Although with 4 or more items, the recursive approach time increases rapidly (since recursive has a time complexity of O(2^N)) while dynamic is still increasing in time, but at a much slower pace.

In the right graph, the time between recursive and dynamic is measured with a fixed number of items and increasing knapsack capacity. Although the recursive approach does not scale with capacity, we can still see that the execution time is still increasing with capacity up until 40 capacity then it plateaued out past that point. The execution time for dynamic programming increases slowly with the rising number of knapsack capacity.

4. Reflection:

With an increasing number of items, we can see that recursive knapsack takes way longer compared to dynamic programming to executes on a not-low number of items, these results are as intended, as O(2^N) is the second highest time complexity class, only behind factorial (O(N!)).

With an increasing number of capacity, the unexpected happened, the execution time of recursive knapsack increases with knapsack capacity up to 40 capacity although this approach does not scale with capacity. I believe that as the capacity increases, the recursive approach has to process more combinations and 40 capacity is the break point where it can put every item in the knapsack then the amount of combinations stop rising. For dynamic programming, the time increase is as intended since it scales with knapsack capacity at the same pace as the number of items (O(N*Capacity)).

Task D: Maximizing Knapsack Value without Having the Map

1. **Algorithms Design:**

Algorithm 1 taskDSolver(*maze*, *entrance*, *exit*)

```
    Input:
maze: maze layout
entrance: the coordinates of the entrance,
exit: the coordinates of the exit
    Output: None
# Initialize knapsack
$optimalCell \leftarrow []$
$optimalValue \leftarrow 0$
$optimalWeight \leftarrow 0$

# Initialize values for for the TaskDSolver object
$solverPath \leftarrow []$
$cellExplored \leftarrow |set(solverPath)|$
$reward \leftarrow 0$
$foundTreasures \leftarrow []$

# Create points to visit on the maze
$centre \leftarrow$ Coordinates(floor($maze$.rowNum() // 2), floor($maze$.colNum() // 2))
$southWest \leftarrow$ Coordinates(0, 0)
$southEast \leftarrow$ Coordinates(0, $maze$.colNum() - 1)
$northEast \leftarrow$ Coordinates($maze$.rowNum() - 1, $maze$.colNum() - 1)
$northWest \leftarrow$ Coordinates($maze$.rowNum() - 1, 0)

# Create a list of BFS pair
$pathSegments = [$
    $(entrance, southWest)$,

    $(southWest, centre)$,
    $(centre, southWest)$,
    $(southWest, southEast)$,

    ....Repeat for other corners....

    $(southWest, exit)$
$]$

# Find a path via BFS
for $i, (start, end) \in$ enumerate($pathSegments$) do
    $segment \leftarrow$ BFS($maze, start, end$)
    if $i == 0$ then
        Extend $segment$ to $solverPath$
    else
        Extend $segment[1:]$ to $solverPath$
    end if
end for

# Check if the cell on the path contains any treasure
for $cell \in solverPath$ do
    $loc \leftarrow ((cell$.getRow(), $cell$.getCol()))
    $item \leftarrow maze$.m_items.get($loc$)          ▷ get the value of the cell
    if $item$ is not empty then                       ▷ If that cell has a treasure
        $weight, value \leftarrow item$
        Append $(loc, value, weight)$ to $foundTreasures$
    end if
end for

# Solve knapsack on found treasures
$(optimalCells, optimalWeight, optimalValue) \leftarrow$
        DYNAMICKNAPSACK($foundTreasures, knapsackCapacity, |foundTreasures|$)

# Update reward
$cellExplored \leftarrow |set(solverPath)|$           ▷ get the number of unique cells
$reward \leftarrow optimalValue - cellExplored$
```

Strategy: The Adventurer starts from the entrance and goes to one corner of the maze then from that corner, go to the center of the maze then go back to the starting corner, then they move to the next corner. Repeat the same process for 3 other corners (Order: South West - South East - North East - North West) then exit the maze. All traversing steps are done with BFS. After exiting the maze, the Adventurer will use dynamic programming knapsack (implemented on Task B) to decide which treasure on the note to pick. After that they go back and pick up the treasures.

This approach uses BFS to reduce the amount of explored cells when traversing while forcing the Adventurer to explore deep into the maze instead of letting them exit the maze immediately. Although this approach has more cell cost compared to using BFS directly from entrance to exit, it prevents scenarios where the Adventurer finishes exploring in 4 or 5 steps (with no treasure) when the entrance or exit are too close to each other. Forcing a few visiting points also helps open more chances to come across a treasure.

Time Complexity: Have 3 main steps (BFS, cell checking, Dynamic knapsack)

a. **BFS:** Time complexity of O(V + E), where V stands for vertices and E for edges, in this maze, since we can only move left-right-up-down by 1 cell, the number of edges is at most 4V. Therefore, in the worst case, the time complexity is O(V + 4V) = O(5V). And since we have 14 runs of BFS (each corner takes 3, both entrance and exit takes 2), it is O(14*5V) = O(70V) = O(V). Vertices in this case means the number of unique cells BFS have visited during the pathfinding process, in the worst case, in can be equal to the total amount of cells in the maze, which equals to Row * Column. Therefore the time complexity of this steps is **O(R*C)** where R stands for the number of rows and C stands for the number of columns of the maze.
b. **Cell Checking:** This step checks every cell in BFS's final results/paths or `m_solverPath`. In which the worst case scenario will be every cells on the maze, resulting in a **O(R*C)** time complexity.
c. **Dynamic knapsack**: As explained in task C, this approach for solving knapsack problem have a time complexity of **O(n*c)**, where n is the number of found treasures and c is the knapsack capacity.

With all three steps together, the total time complexity is:

O((R*C) + (R*C) + n*c)) = O(2(R*C) + n*c)) = **O(R*C + n*c)**

2. **Assumptions**
    a. **Uniform Distribution:** This will work well with this implementation as treasures are randomly distributed and can be discovered along the BFS path.

    b. **Linear Distance-Based Skew:** This implementation may perform better as it visits all of the corners of the maze, where most of the treasures are. In the worst case scenario, the high-value areas are not in the corners nor near the centre of the maze, resulting in this implementation underperforming. A potential change is to move the 4 corners visiting point closer to the high-value area (if we know where the zones are).

    c. **Clustered zone:** This implementation may miss the entire cluster if they are outside of the BFS path. A potential change is having one or two more visit points near or in the clustered area with checking neighbours' cells of visit points, that way more treasure will be discovered by the BFS.