RMIT UNIVERSITY

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

Nathan Yong and 4090456

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 programming 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.

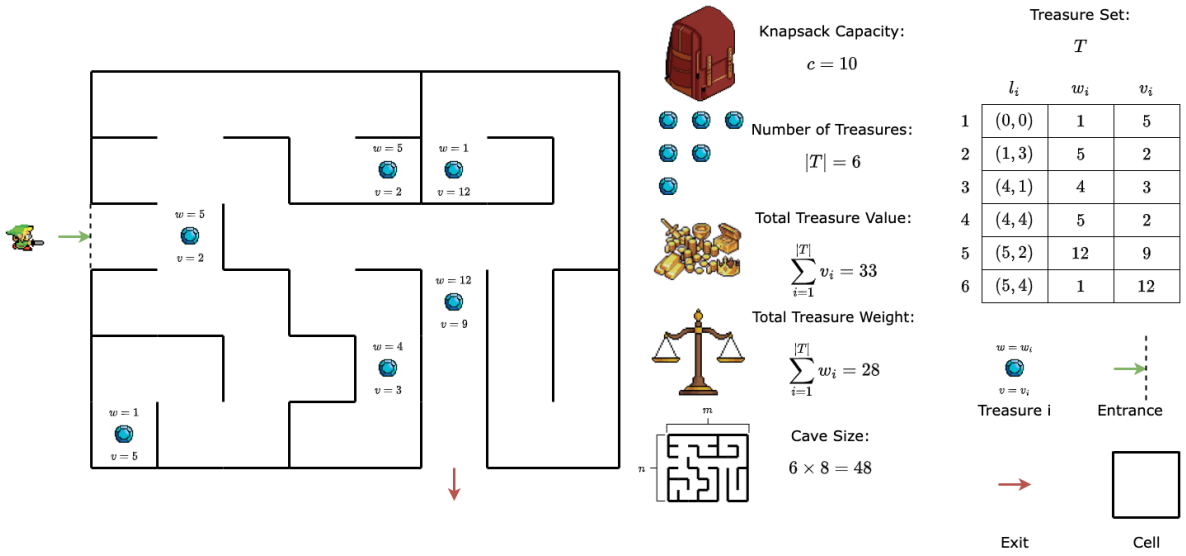
2 Overview

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*

Motivation

You are to assist an adventurer searching for treasure. They have heard tales about a maze filled with valuable treasures. The Adventurer's goal is to enter the maze, find and collect as many treasures as they can, and then leave through a different exit. Once they enter, the entrance will close behind them, so they must find another way out!



	l_i	w_i	v_i
1	$(0,0)$	1	5
2	$(1,3)$	5	2
3	$(4,1)$	4	3
4	$(4,4)$	5	2
5	$(5,2)$	12	9
6	$(5,4)$	1	12

Figure 1: A breakdown of all the information available to The Adventurer. The layout of the maze (made up of $n \times m$ cells), as well as entrance/exit locations and treasure locations (including the value and weight of each treasure). The book tells The Adventurer the information in the centre column - number of treasures, total value and total weight of all treasures.

As shown in Figure 1, in their possession The Adventurer has:

- an enchanted bag with carrying capacity c (the maximum weight it can hold);
- a mystical map, which tells them:
  - the layout of the maze (which is fully connected but may have cycles and is always rectangular of size $n \times m$ with square cells) including the entrances, exits, and walls;
  - the location, l_i, weight w_i and value v_i of each treasure i; The location of a treasure is in the form of (col, row), where $0 \leq col \leq m-1$ and $0 \leq row \leq n-1$. As can be seen in the Treasure Set T in Figure 1, the row numbers start from bottom to top and the column numbers start from left to right. The treasures are also sorted using a bottom-to-top, left-to-right sweep.
- a magical book, which contains:
  - the number of treasures in the maze $|T|$;
  - the total value of all the treasures in the maze $v = \sum_{i=1}^{|T|} v_i$; and
  - the total weight of all the treasures in the maze $w = \sum_{i=1}^{|T|} w_i$.

Objective

Your objective is to assist The Adventurer in gathering as many valuable treasures as their knapsack can carry. Mathematically, you wish to:

$$\text{maximise} \sum_{i=1}^{|T|} v_i s_i \qquad \text{subject to} \sum_{i=1}^{|T|} w_i s_i \leq c \text{ and } s_i \in \{0, 1\}$$
$$\text{minimise } |P| \qquad \text{subject to } l_i \in P \text{ if } s_i = 1$$

where:

- $s_i = 1$ means the i^{th} treasure is picked up while $s_i = 0$ means the i^{th} treasure is left behind;
- $P = \langle (i, j), ... \rangle$ is the ordered multiset of cells The Adventurer plans to visit (where the first element is the entrance, the last element is the exit, and each element is adjacent to the previous element).

In other words, we wish to find the shortest path through the maze that maximises the value of our knapsack, subject to the condition that the total weight of the knapsack is less than or equal to its maximum carrying capacity.

Completing Tasks A, B, C, and D will take you through different methods for completing this objective. **Please read each Task carefully, and implement only what is asked in each Task.**

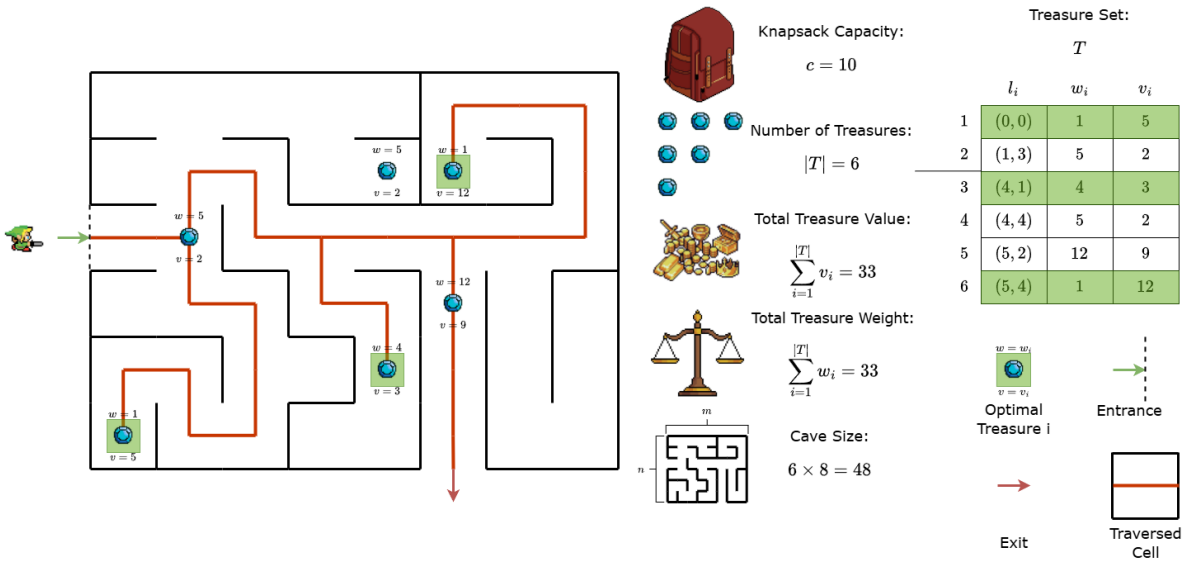


Figure 2: A solution for The Adventurer given the specific problem in Figure 1. We can see that the optimal treasures have been selected in the treasure set T, their positions highlighted on the map, and a route has been planned that takes The Adventurer through each cell containing an optimal treasure to collect before going through the exit.

Task B

Algorithm 1 DynamicKnapsack(T, c, k)

Input: T, a map of treasures where treasure IDs are mapped to location, weight, value tuples, i.e., $t_i : (l_i, w_i, v_i)$; a positive integer c representing the knapsack capacity; and a positive integer k, the number of treasures.

Output: A list of locations $selected\_items$ for the optimal selection of treasures, total weight $selected\_weight$, and maximum value max_value of the selected treasures.

```
w_L ← ∅; v_L ← ∅
function MFKNAPSACK(k, c)
    if k = 0 or c = 0 then
        return dp[c][k]
    for i in range(k) do
        w ← w + T[i][1]
    for i in range(k) do
        v ← v + T[i][2]
    if dp[k][c] is none then
        if c < w[k − 1] then
            x ← MFKNAPSACK(k − 1, c)
        else
            x ← max(MFKNAPSACK(k − 1, c), v[k − 1] + MFKNAPSACK(k − 1, c − w[k − 1]))
        dp[k][c] ← x
    return dp[k][c]
end function
max_value ← MFKnapsack(k, c)
capcity ← c

for i in range(k, 0, −1) do
    if dp[i][capacity] != dp[i − 1][capacity] then
        loc, wt, val ← items[i − 1]
        selected_items ← selected_items ∪ {loc}
        selected_weight ← selected_weight + wt
        capacity ← capacity − wt

return selected_items, selected_weight, max_value
```

The potential benefits of DynamicKnapsack are that it only calculates the necessary items to find in order to gather the maximum value of treasure, therefore, it is more efficient. Whereas, recursiveKnapsack goes through all the possible combinations. A downside to DynamicKnapsack is that it requires memory to store previous solutions to sub-problems. Indicating it can be high memory usage.

Task C

1. Algorithm Complexity Analysis

(a) In the pseudocode of Task A, lines 13 and 14 both contain a recurrence relation. Thus, we call the recurrence relation twice through each loop. The first call includes the item, and therefore, the weight of the item is subtracted from the knapsack capacity, and the second call excludes the item. Both calls subtract 1 from the number of items as it goes through each item. The recursion continues until the base case is true, when the capacity of the knapsack is zero ($c = 0$) or when the number of items is zero ($k = 0$). The algorithm calls itself twice each iteration, which gives the time complexity of the knapsack to be $O(2 \cdot C(k-1) + 2)$ or $O(2^k)$.

In the pseudocode of Task B, line 12 is a 2D dynamic programming table (number of items multiplied by the knapsack capacity) that stores the values of the items that need to be calculated in order to find the maximum value of treasure. With the memory table, this prevents the algorithm from recalling a value that has already been solved. Therefore, the dynamic programming table (`dp[c][k]`) substantially decreases the number of times the recursive algorithm is called. Since the number of times the recursive algorithm is called is dependent on the dynamic programming table, the time complexity is $O(c \cdot k)$.

(b) The function `findItemsAndCalculatePath` has multiply functions `solveKnapsack` Breadth-First-Search `bfs` and `solveMaze`. SolveKnapsack uses the method selected, which could be recursive or dynamic. Next, `bfs` is called in a nested for loop in `solveKnapsack`, resulting to $O(n^2)$. Following this, in `solveKnapsack`, permutations are used to find the shortest route using the locations of optimal cells.This has a factorial time complexity producing $O(n!)$. If the recursive method is selected then adding in the time complexity for recur of $O(2^k)$, resulting in $O(2^k + n!)$. If the dynamic method is selected then adding in the time complexity for dynamic of $O(c \cdot k)$, resulting in $O(c \cdot k + n!)$. Since, $O(n!)$ is the highest complexity, the overall time complexity of `findItemsAndCalculatePath` is $O(n!)$

2. Empirical Design:

The variables I believe are important are the number of items and capacity of knapsack in regards to the time complexity of `findItemsAndCalculatePath`. The number of items is significant, as `findItemsAndCalculatePath` time cost is heavily dependent on the number of cells with treasure to visit. Since `solveKnapsack` function uses the locations of the treasures to find the shortest path, the number of items and the time complexity have a positive relationship therefore, a decrease in the number of items leads to a decrease in time complexity, as the permutation computation won't be as great. Also, in general if there are less number of items then there will be less combinations of what items to pick up. For the capacity of knapsack if the maximum capacity is lower, then knapsack solver will find less optimal locations for treasures, as the adventurer won't be able to carry many items, and therefore a decrease in run time of `findItemsAndCalculatePath`. The other variables i chose to ignore such as the size of the maze do play an important part however, the greatest time complexity is $O(n!)$, which takes into account for the optimal cells to visit. All the other variables don't affect this as much as the number of items and knapsack capacity.

3. Empirical Analysis:

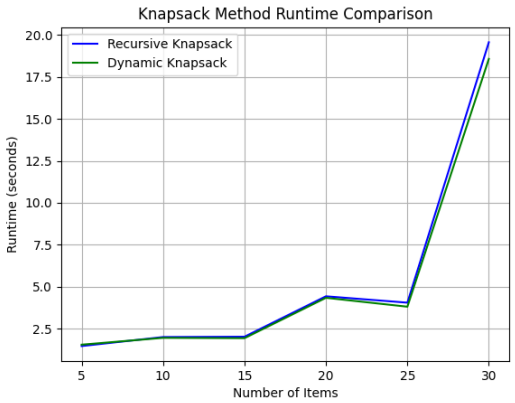


Figure 1: The change in runtime with increasing number of items

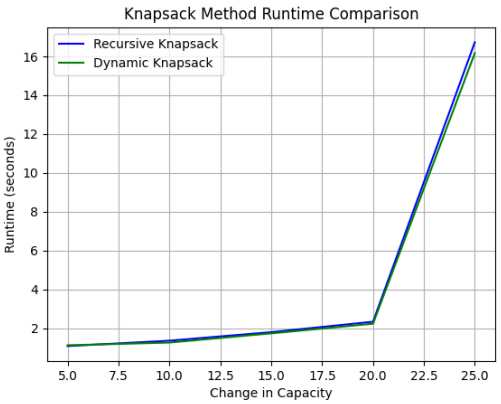


Figure 2: The change in runtime with increasing the capacity of knapsack

Figure 1 illustrates how an increase in the number of items substantially increases the runtime. However, even though the dynamic knapsack has a more efficient time complexity of $O(c \cdot k)$ compared to the recursive knapsack time complexity of $O(2^k)$ both methods have almost identical results and this is because of `findItemsAndCalculatePath` time complexity of $O(n!)$ which overrides the difference. Similarly, in Figure 2, an increase in the knapsack capacity also leads to a drastic increase in runtime.

4. Reflection:

My empirical results do align with my theoretical complexity predictions. In my theoretical complexity predictions I stated the time complexities for the recursive knapsack being $O(2^k)$ and the dynamic knapsack being $O(c \cdot k)$. However, since `findItemsAndCalculatePath` has a time complexity of $O(n!)$ it disregards the lower time complexity values. Thus, causing almost identical results in both Figure 1 and Figure 2.

Task D

Algorithms Design:

The exploration and treasure-selection strategy for my solution is to use the number of items multiplied by the total value, then divided by the size of the maze $(|T| \cdot |v|/n \cdot m)$. This gives an estimate of how likely each cell is to have a treasure and the average value of each cell, let this be called the $initalThreshold$. The adventurer keeps exploring until the current estimate ($density$) is less than the $initalThreshold$. When it finds an item in the current cell it will take note of its location, weight, value and add them to $items\_data$. Addtionally, the solution will subtract the total number of items by 1 $(|T| - 1)$ and subtract the total value by the value of the new item discovered $(|v| - v)$. As the explorer travels cell by cell, the size of the maze decreases as there will be less unique cells to explore $(n \cdot m - 1))$ and therefore, a higher estimate for a treasure item in the next cell. However, if an item is found, the treasure probability will decrease $((|T| - 1)/n \cdot m)$. If the current estimate ($density$) is less than the initalThreshold then stop exploring and head for the exit. While heading for the exit, if any new items are

discovered, add them to $items\_data$. This solution minimises exploration while still trying to find treasure. If it finds that the current likelihood of each unvisited cell to have treasure with a decent average value is less than the initialThreshold then it is not worth exploring more. Therefore, this approach balances between expected treasure value and cell cost.

Input: Maze object $maze$, entrance coordinate $entrance$, exit coordinate $exit$
Output: Path from $entrance$ to $exit$, collecting treasures (marked green)

$optimalCells_K \leftarrow \emptyset$; $optimalValue_K \leftarrow 0$; $optimalWeight \leftarrow 0$
$items\_in\_maze \leftarrow maze.m\_itemParams[0]$
$maze\_value \leftarrow sumofvaluesinmaze.m\_items$
$mazeSize \leftarrow$ total number of cells in maze
$currItems \leftarrow items\_in\_maze$
$currValue \leftarrow maze\_value$
$initialThreshold \leftarrow \frac{currItems \cdot currValue}{mazeSize}$
$itemData \leftarrow \emptyset$, $numItems \leftarrow 0$
$currentCell \leftarrow entrance$
$solverPath \leftarrow [currentCell]$
$visited \leftarrow \{currentCell\}$
$cellsExplored \leftarrow 1$
$mazeSize \leftarrow mazeSize - 1$
while $currentCell \neq exit$ **do**
    $cellTuple \leftarrow (currentCell.\texttt{row}, currentCell.\texttt{col})$
    if $cellTuple$ in $maze.m\_items$ **then**
        $(weight, value) \leftarrow maze.m\_items[cellTuple]$
        $density \leftarrow \frac{currItems \cdot currValue}{mazeSize}$
        if $cellTuple \notin optimalCells_K$ **then**
            $currItems \leftarrow currItems - 1$
            $currValue \leftarrow currValue - value$
            Append $(cellTuple, weight, value)$ to $itemData$
            $numItems \leftarrow numItems + 1$
        if $density < initialThreshold$ **then**
            $pathToExit \leftarrow$ BFS$(maze, currentCell, exit)$
            for all $step \in pathToExit[1:]$ **do**
                Append $step$ to $solverPath$
                $stepTuple \leftarrow (step.\texttt{row}, step.\texttt{col})$
                if $stepTuple \in maze.m\_items$ and $stepTuple$ not in $optimalCells_K$ **then**
                    $(w, v) \leftarrow maze.m\_items[stepTuple]$
                    $currItems \leftarrow currItems - 1$, $currValue \leftarrow currValue - v$
                    Append $(stepTuple, w, v)$ to $itemData$
                    $numItems \leftarrow numItems + 1$
                if $step$ not in $visited$ **then**
                    Add $step$ to $visited$, increment $cellsExplored$
            break
    $neighbors \leftarrow maze.neighbours(currentCell)$
    $validNeighbors \leftarrow$ [n for n in neighbours if n not in visited and no wall]
    if $validNeighbors$ not $\emptyset$ **then**
        $nextCell \leftarrow$ random choice from $validNeighbors$
    else
        $foundNewPath \leftarrow$ False

```
        for i = len(solverPath) − 2 to 0 do
            backtrackCell ← solverPath[i]
            if not maze.hasWall(currentCell, backtrackCell) then
                currentCell ← backtrackCell
                Append currentCell to solverPath
                if currentCell not in visited then
                    Add currentCell to visited, increment cellsExplored
                for all neighbor in maze.neighbours(currentCell) do
                    if neighbor not in visited and not maze.hasWall(currentCell, neighbor)
then
                        currentCell ← neighbor
                        Append currentCell to solverPath
                        Add currentCell to visited, increment cellsExplored
                        foundNewPath ← True
                        break
            if foundNewPath then
                break
        if not foundNewPath then
            break
        continue
    currentCell ← nextCell
    Append currentCell to solverPath
    if currentCell not in visited then
        Add currentCell to visited, increment cellsExplored
    mazeSize ← mazeSize − 1
end while
(optimalCells_K, optimalWeight, optimalValue_K) ← dynamicKnapsack(itemData, capacity, nu
Update entranceUsed, exitUsed, and reward
```

Assumptions:

My solution is more prone for uniform distribution as it calculates it initialThreshold assuming each cell has the same likelihood of having treasure. My algorithm randomly selects valid neighbouring cells as long as the current estimate ($density$) is greater than or equal to the $initalThreshold$. For Linear Distance-Based Skew my solution solely depends on which random neighbour it selects, if it is a path towards the yellow cells then my solution will do quite well as it navigates that area until the current estimate ($density$) is less than the $initalThreshold$ and as it heads towards the exit the items found in the path will be added to $items\_data$. For the Clustered Zones again my solution solely depends on which random neighbouring cell is selected if the path enters a clustered zone it will add all items to $items\_data$, then head for the exit, as there will be less treasure to find and therefore minimizing exploration and cell cost.