RMIT UNIVERSITY

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

Your Name and student number Here

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.

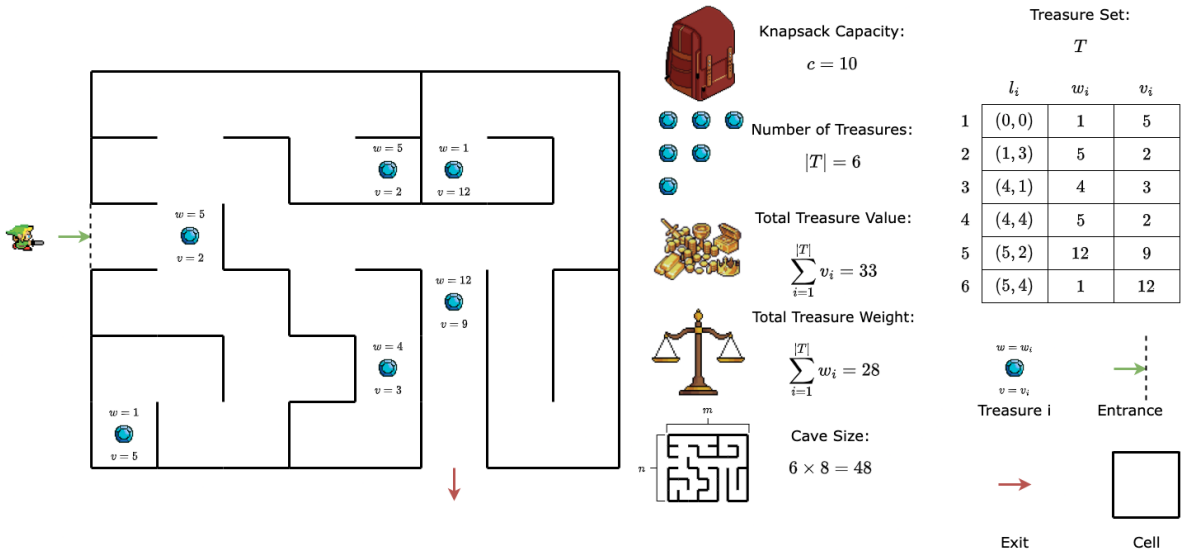
2 Overview

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": YOUR ANSWER HERE

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.**

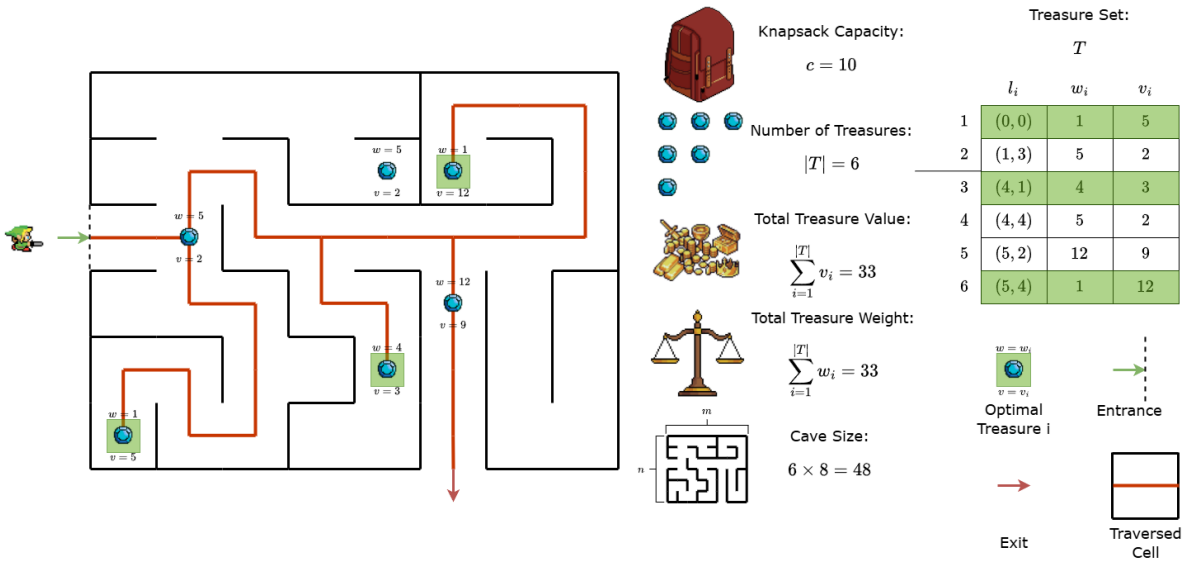


Figure 2: A solution for The Adventurer given the specific problem in Figure 1. We can see that the optimal treasures have been selected in the treasure set T, their positions highlighted on the map, and a route has been planned that takes The Adventurer through each cell containing an optimal treasure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

Task B

Algorithm 1 DynamicKnapsack(T, c, k, $filename$)

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; a positive integer k, the number of treasures; and $filename$ for saving the DP table.

Output: A list of locations L_{opt} for the optimal selection of treasures, total weight w_{opt}, and total value v_{opt} of the selected treasures.

Initialize dp as $(k+1) \times (c+1)$ table filled with $None$
$dp[0][j] \leftarrow 0$ for all $0 \leq j \leq c$ ▷ Base case: empty knapsack
function F(i, j) ▷ Memory function for DP
    if $i = 0$ **or** $j = 0$ **then**
        $dp[i][j] \leftarrow 0$
        return 0
    if $dp[i][j] \neq None$ **then**
        return $dp[i][j]$

    $t \leftarrow T[i-1]$ ▷ Get the i^{th} treasure
    $weight \leftarrow t$.weight
    $value \leftarrow t$.value

    if $weight > j$ **then**
        $dp[i][j] \leftarrow F(i-1, j)$
    else
        $dp[i][j] \leftarrow \max(F(i-1, j), F(i-1, j-weight) + value)$

    return $dp[i][j]$
end function

$v_{opt} \leftarrow F(k, c)$ ▷ Compute maximum value
$L_{opt} \leftarrow \emptyset$; $w_{opt} \leftarrow 0$; $current\_cap \leftarrow c$

for $i \leftarrow k$ **downto** 1 **do**
    $t \leftarrow T[i-1]$
    if $current\_cap \geq t$.weight **and** $dp[i][current\_cap] \neq dp[i-1][current\_cap]$ **then**
        $L_{opt} \leftarrow L_{opt} \cup \{t$.location$\}$
        $w_{opt} \leftarrow w_{opt} + t$.weight
        $current\_cap \leftarrow current\_cap - t$.weight

Reverse L_{opt} ▷ Restore original order
Save dp table to csv
return $L_{opt}, w_{opt}, v_{opt}$

Pros: Avoids redundant calculations by memoizing subproblem solutions, reducing the time complexity

Cons: Requires additional space to store the DP table, which can be memory-intensive for very large capacities

Task C

Algorithm Complexity Analysis:

Recursive Knapsack (Task A)
The recursive solution follows the recurrence relation $T(n, C) = T(n-1, C-w) + T(n-1, C) + O(1)$, where each call branches into two possibilities, including or excluding the current item. This creates a binary recursion tree, leading to exponential growth. Since overlapping subproblems are not stored, the same subproblems are recalculated multiple times, resulting in significant inefficiency for larger inputs. Therefore the time complexity is $O(2^n)$. With n being the number of items.

Dynamic Knapsack (Task B)
The dynamic programming approach optimizes the solution by storing intermediate results in a table of size $(n+1)(C+1)$. Each cell $F(i, j)$ is computed in constant time using the recurrence

$$F(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ F(i-1, j) & \text{if } w_i > j \\ \max\{F(i-1, j-w_i) + v_i,\ F(i-1, j)\} & \text{else} \end{cases}$$

This ensures that each subproblem is solved only once, reducing the time complexity from exponential to polynomial. The time complexity remains $O(n \times C)$. With n being the number of items and C being the capacity of the knapsack.

findItemsAndCalculatePath Function (Task C)
The findItemsAndCalculatePath function operates in two distinct phases:
1. Solves the knapsack problem (using either recursion or dynamic programming as above).
2. Calculates the shortest path through the maze to collect the selected treasures and exit.

The pathfinding phase works as follows:
It runs a Breadth-First Search (BFS) between every pair of points: the entrance, the selected treasures, and the exit. If there are k selected treasures, there are $(k+2)(k+1)$ BFS calls.
Each BFS runs in: $O(n \times m)$
where n and m are the maze's dimensions (row and columns respectively).

Thus, total path finding BFS time is: $O(k^2 \times n \times m)$
It then considers all possible permutations of visiting the selected treasures. There are $k!$ permutations, and for each one, it calculates the total path length and unique cells visited (both in $O(k)$).
Total permutations time:$O(k! \times k)$

Empirical Design and Analysis + Reflection:

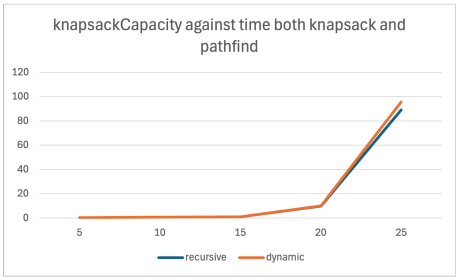
Primary Variables
1. n: The number of treasures in the maze. This directly affects the complexity of both the recursive and dynamic programming knapsack solutions: In the recursive knapsack: time complexity is exponential in n - $O(2^n)$. In the dynamic programming knapsack: time complexity is linear in n - $O(n \times C)$. Additionally, the number of treasures selected (denoted k) affects the number of path finding operations, as the number of BFS calls and permutations depends on how many treasures are picked for collection.

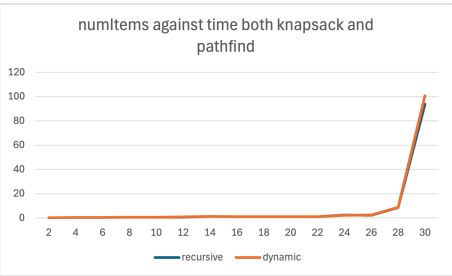
2. C: The knapsack's capacity. This only affects the dynamic programming approach, since the DP table size is proportional to $n \times C$. It also affects the possible values of k (the number of items selected) based on the available capacity and numItems.
3. $n \times m$: The size of the maze grid. The maze's total number of cells determines the cost of each BFS operation in the pathfinding phase. Each BFS runs in $O(n \times m)$, and with multiple BFS calls required (proportional to $(k+2)(k+1)$) to find shortest paths between entrance, treasures, and exit, maze size has a linear effect on total search time.

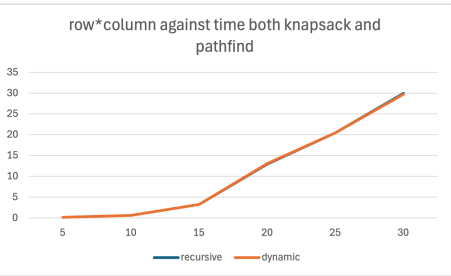
Variables Ignored and Why

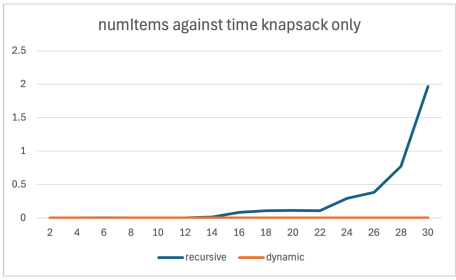
Maze layout (walls/paths): Affects BFS runtime slightly in practice, but worst-case remains $O(n \times m)$.
Number of entrances/exits: Fixed small number, constant impact on total BFS calls.
Treasure values and weights: Influence decision outcomes, but not the number of recursive calls, DP table entries, or BFS runs.

Strategy	Knapsack Time	Pathfinding Time	Total Time
Task A (Recursive)	$O(2^n)$	$O(k^2 \times nm + k! \times k)$	$O(2^n + k^2 \times nm + k! \times k)$
Task B (Dynamic Programming)	$O(nC)$	$O(k^2 \times nm + k! \times k)$	$O(nC + k^2 \times nm + k! \times k)$









The recursive knapsack algorithm shows exponential growth in runtime as the number of treasures increases for the knapsack phase, consistent with its $O(2^n)$ complexity. In contrast, the dynamic programming (DP) approach maintains near-constant runtimes for small numbers of items and grows linearly with the number of items, following its expected $O(n \times C)$ complexity for the knapsack problem.
Both strategies scale linearly with maze size, as shown in the runtime trends against maze dimensions, reflecting the $O(n \times m)$ complexity of the pathfinding algorithm.
Additionally, the runtime patterns observed in relation to the number of items and knapsack capacity confirms that increases in either capacity of numItems lead to a larger number of selected treasures, k, causing the pathfinding phase to exhibit factorial growth according to its $O(k! \times k)$ complexity.

Task D

Algorithms Design:

In Task D, the adventurer explores a maze to collect treasures, but does not know where the treasures are in advance. The maze layout, the entrance, and the exit are given, as well as the total number of treasures, their combined value, and total weight. However, the specific locations are only revealed as cells are visited.

To maximize reward (knapsack value minus the number of unique cells visited), my algorithm uses the following approach:

Algorithm 2 Greedy Shortest-Path Exploration for Hidden Treasure Collection (Task D)

Input: Maze layout (with walls, entrances, exits), Entrance coordinate, Exit coordinate, Knapsack capacity C, Number of treasures $|T|$, max treasure weight; **Output:** Path taken (P), treasures collected, total knapsack value, total unique cells visited

Initialize $P \leftarrow$ [Entrance] ▷ The path traversed
Initialize $Visited \leftarrow$ Entrance ▷ Set of unique cells explored
Initialize $PickedItems \leftarrow$ ▷ Treasures added to knapsack
Initialize $KnapsackWeight \leftarrow 0$, $KnapsackValue \leftarrow 0$
$Current \leftarrow$ Entrance
Find **shortest path** S from Entrance to Exit using BFS or A*
for each cell C_i in S **in order do**
    if C_i not in $Visited$ **then**
        add C_i to $Visited$
    add C_i to P
    if cell C_i contains treasure T_j **then**
        if $KnapsackWeight + T_j.weight \leq C$ **then**
            add T_j to $PickedItems$
            $KnapsackWeight \leftarrow KnapsackWeight + T_j.weight$
            $KnapsackValue \leftarrow KnapsackValue + T_j.value$
    if $KnapsackWeight \geq (C - MinPossibleWeight)$ **then**
        break the for-loop and immediately reroute to Exit
if $Current \neq$ Exit **then**
    Find **shortest path** Q from $Current$ to Exit
    for each cell C_k in Q (excluding the first, as it's $Current$) **do**
        if C_k not in $Visited$ **then**
            add C_k to $Visited$
        add C_k to P
        if cell C_k contains unseen treasure T_m **then**
            if $KnapsackWeight + T_m.weight \leq C$ **then**
                add T_m to $PickedItems$
                $KnapsackWeight \leftarrow KnapsackWeight + T_m.weight$
                $KnapsackValue \leftarrow KnapsackValue + T_m.value$
return: P (full path), $PickedItems$, $KnapsackValue$, $|Visited|$ (unique cells explored), Reward $= KnapsackValue - |Visited|$

The algorithm balances potential value against cell cost by trying to maximize the value in the knapsack (picking up as many treasures as possible), while using BFS to keep the cell-exploration low. To maximize reward could have made an array with treasures passed and their weight and value, then at the end when using BFS at the exit use knapsack solver to find best values but was unable to due time constraints

Assumptions:

The solution I came up with assumes that at default every cell in the maze is equally likely to have a treasure (a uniform distribution). Under this assumption, explorations such as BFS or DFS is reasonable and expected to perform well on average, as there is no prior reason to focus on one region of the maze over another.

If the probability distribution is not uniform, such as:
Linear Distance-based bias: Treasures are more likely near the farthest cells from the entrance/exit.
Clustered Zones bias: Treasures are more likely to be found in certain regions of the maze (clusters).
Then that means the BFS solution may become suboptimal. One way to adapt to the change would be to update the exploration heuristic, so that if treasures are mostly found in a certain type of location, you can prioritize visiting those regions instead of uniformly.
However, if no such information is available, the algorithm cannot take advantage of the bias, and the uniform approach is best used.

Uniform Distribution: The strategy is optimal for a uniform prior, as no region is preferred for exploration.

Clustered or Biased Distributions: The stratgy will be suboptimal but can implement it so if, during exploration, it becomes clear that treasures are likely in certain areas (e.g., deep in the maze, or clustered in blocks), the policy could be adapted to move the adventurer towards high-probability zones. For example, if all discovered treasures are clustered, the agent could detour in that direction, but would do so only if the statistical benefit outweighs the exploration cost.

Without Prior Knowledge: With no observed or provided bias, the adventurer plays it safe, minimizing path and cell penalty.

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