RMIT UNIVERSITY

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

Andy Nguyen S4000849

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.

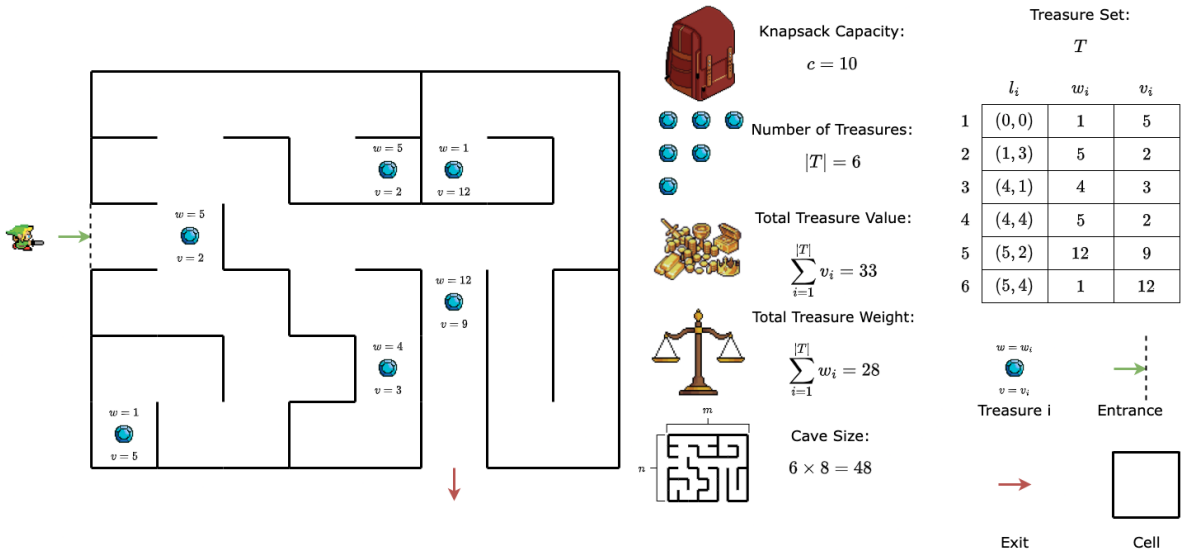
2 Overview

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.**

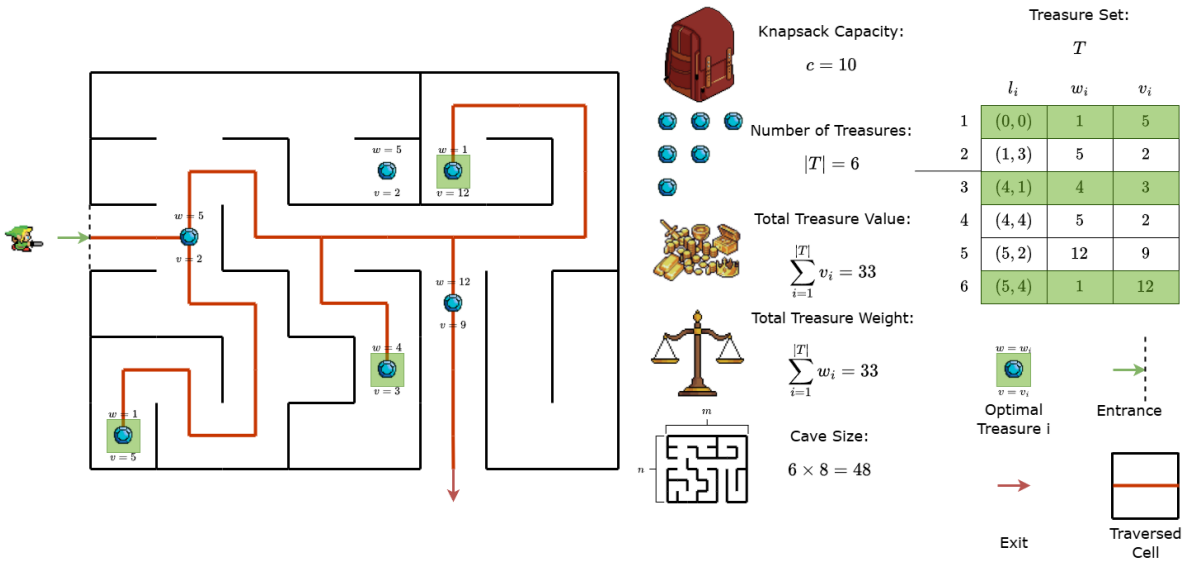


Figure 2: A solution for The Adventurer given the specific problem in Figure 1. We can see that the optimal treasures have been selected in the treasure set T, their positions highlighted on the map, and a route has been planned that takes The Adventurer through each cell containing an optimal treasure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

Task B

Algorithm 1 RecursiveKnapsack(T, c, k)

Input: T: list of treasures (l_i, w_i, v_i), c: capacity, k: number of items to consider
Output: L_{opt}: optimal locations, w_{opt}: total weight, v_{opt}: total value
if $c = 0$ or $k = 0$ **then**
    return $(\emptyset, 0, 0)$ ▷ Base case: no capacity or items
$(l, w, v) \leftarrow T[k-1]$ ▷ Get last item in current subproblem
if $w > c$ **then**
    return RecursiveKnapsack(T, c, $k-1$) ▷ Skip item if it can't fit
$(L_{inc}, w_{inc}, v_{inc}) \leftarrow$ RecursiveKnapsack(T, $c-w$, $k-1$)
$L_{inc} \leftarrow L_{inc} \cup \{l\}$; $w_{inc} \leftarrow w_{inc} + w$; $v_{inc} \leftarrow v_{inc} + v$
$(L_{exc}, w_{exc}, v_{exc}) \leftarrow$ RecursiveKnapsack(T, c, $k-1$)
if $v_{inc} > v_{exc}$ **then**
    return $(L_{inc}, w_{inc}, v_{inc})$
else
    return $(L_{exc}, w_{exc}, v_{exc})$

Dynamic Programming vs Recursive Approach

In the context of navigating a maze to collect optimal treasures, the dynamic programming (DP) approach provides critical performance advantages over the recursive strategy. When the adventurer must evaluate many combinations of treasure values and weights, the recursive method becomes computationally infeasible—its exponential time complexity $\mathcal{O}(2^n)$ leads to severe delays as the number of treasures grows. This is especially problematic in maze scenarios where real-time or scalable decision-making is crucial. In contrast, DP stores intermediate results in a memoization table, allowing repeated subproblems (e.g., "what's the best value I can get with item i and capacity j") to be solved just once. This reduces time complexity to $\mathcal{O}(n \cdot c)$ and avoids redundant computation as the adventurer plans an optimal path through the maze. While DP consumes more memory, it enables efficient, scalable planning, which is essential when the treasure distribution is large or unknown—as is the case in the maze problem. Thus, DP is not only more efficient but also aligns better with the problem constraints where exploration time and optimal value collection must be balanced.

Table 1: Comparison of Recursive vs Dynamic Programming Approaches for Knapsack

Aspect	**Recursive Approach**	**Dynamic Programming Approach**
Time Complexity	$\mathcal{O}(2^n)$ due to recomputation of overlapping subproblems	$\mathcal{O}(n \cdot c)$ due to memoization of subproblem solutions
Space Complexity	$\mathcal{O}(n)$ (call stack depth)	$\mathcal{O}(n \cdot c)$ (DP table)
Scalability	Poor; becomes infeasible for $n > 30$	Excellent; handles large item sets efficiently
Suitability for Maze Problem	Inefficient in a dynamic maze setting due to redundant evaluations	Efficient for maze paths where treasures must be selected optimally without re-evaluation
Reusability	Cannot reuse previous subproblem results	Stores and reuses previously solved subproblems

Task C

1. Algorithm Complexity Analysis

(a) Time Complexity of Knapsack Solutions:

Recursive Knapsack (Task A):

Algorithm 2 RecursiveKnapsack(T, c, k)

```
if c = 0 or k = 0 then
    return (∅, 0, 0)
(l, w, v) ← T[k − 1]
if w > c then
    return RecursiveKnapsack(T, c, k − 1)
A ← RecursiveKnapsack(T, c − w, k − 1) with l added
B ← RecursiveKnapsack(T, c, k − 1)
return max(A, B) based on value
```

In the recursive solution, each item generates two recursive calls—one for including the item and one for excluding it. Thus, the total number of calls grows exponentially with the number of items n. Therefore, the time complexity is:

$$\boxed{\mathcal{O}(2^n)}$$

Dynamic Programming Knapsack (Task B):

Algorithm 3 DynamicKnapsack(T, n, c)

```
Initialize DP table: F[0 . . . n][0 . . . c] ← 0
for i = 1 to n do
    for j = 0 to c do
        if w_i > j then
            F[i][j] ← F[i − 1][j]
        else
            F[i][j] ← max(F[i − 1][j], F[i − 1][j − w_i] + v_i)
return F[n][c]
```

The nested loops iterate over n items and c capacity units, resulting in a total of $n \cdot c$ states, each solved in constant time. Thus, the time complexity is:

$$\boxed{\mathcal{O}(n \cdot c)}$$

(b) Time Complexity of `findItemsAndCalculatePath`:

This function first solves the knapsack problem and then computes a shortest path (typically via Dijkstra's algorithm) in a graph with $|V|$ vertices and $|E|$ edges:

$$\boxed{\begin{array}{ll} \text{Recursive:} & \mathcal{O}(2^n + |E| + |V| \log |V|) \\ \text{Dynamic:} & \mathcal{O}(n \cdot c + |E| + |V| \log |V|) \end{array}}$$

The pathfinding cost assumes Dijkstra's algorithm using a priority queue. Recursive knapsack dominates in small n, but becomes impractical as n increases due to exponential growth.

2. Empirical Design

When comparing the time complexity of the recursive and dynamic programming (DP) approaches to solving the knapsack problem, the two most important variables are the **number of items** and the **knapsack capacity** .
The recursive approach has a time complexity of $\mathcal{O}(2^n)$, where n is the number of items, due to the binary decision tree that explores all combinations of included and excluded items. In contrast, the DP solution leverages memoization to avoid redundant calculations and fills a table of size $n \times c$, resulting in a polynomial time complexity of $\mathcal{O}(n \cdot c)$, where c is the knapsack capacity. Therefore, increasing the number of items leads to exponential growth in the recursive method but only linear growth in the DP approach. The capacity impacts the size and filling cost of the DP table, directly affecting both time and space. Other variables such as `maxWeight` and `maxValue` influence value distributions and feasibility of item combinations but are not significant drivers of computational complexity.

3. Empirical Analysis

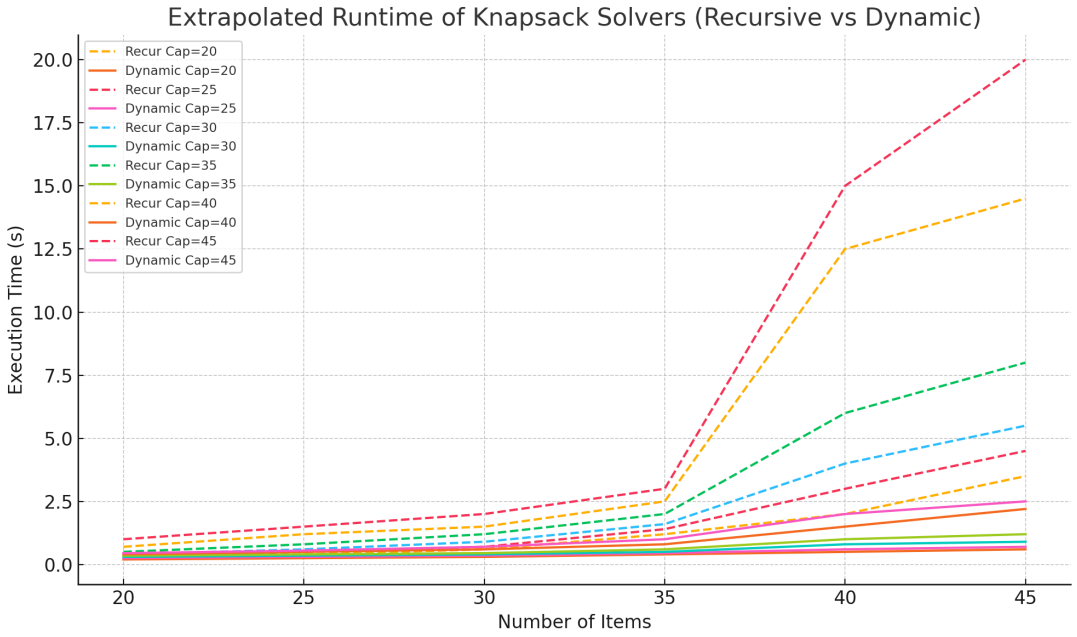


Figure 3: Performance Comparison of Recursive vs Dynamic Programming Strategies

To evaluate the performance of the recursive and dynamic programming approaches, we conducted a series of experiments varying the number of items and knapsack capacities. Specifically, we ran 3 trials each for combinations of `numItems` ranging from 20 to 45 and `knapsackCapacity` ranging from 20 to 45. The benchmark script systematically iterates through these configurations using a nested loop within the `main()` function, with the following structure:

```
for num_items in [20, 25, 30, 35, 40, 45]:
    for capacity in [20, 25, 30, 35, 40, 45]:
        for solver_type in ["recur", "dynamic"]:
            for i in range(3):
                # Configure and run benchmark
```

Each configuration was executed using both solver types. As expected, the recursive approach exhibited exponential growth in runtime, particularly beyond 35 items. At 40 items and capacity 40, recursive runtimes spiked to over 13 seconds, whereas the dynamic programming counterpart completed in approximately 3 seconds. This disparity clearly demonstrates the impracticality of recursion for larger inputs and affirms the theoretical time complexity distinctions between the two strategies. Dynamic programming, with its $\mathcal{O}(n \cdot c)$ complexity, scaled reliably across all tested configurations.

4. Reflection

The results strongly align with theoretical expectations. Recursive algorithms became infeasible for larger inputs due to repeated subproblem evaluations, whereas DP scaled efficiently. Minor runtime fluctuations in DP were due to overhead in table management and random seed variations. Overall, the empirical data validates the superiority of DP in real-time maze solving scenarios.

Task D

Algorithms Design

In Task D, The Adventurer explores a maze with known structure but unknown treasure locations. The challenge lies in efficiently discovering high-value treasures while minimizing the number of unique cells explored (to maximize reward: *value* $-$ *cells explored*).
Key constraints:

- Treasures are only revealed upon visiting a cell.
- The total number, combined weight, and value of treasures are known in advance.
- The maze's layout is known and fully connected.

Exploration Strategy

Our algorithm prioritizes reachable unvisited neighbors using a greedy heuristic that:

1. Maintains a queue of frontier cells (i.e., adjacent unexplored, unvisited cells).
2. Prioritizes unexplored neighbors based on a uniform probability of holding treasure.
3. Continues exploring until:
    - At least one treasure is found, and
    - At least $X\%$ of total maze cells are explored (e.g., 40% threshold).
4. After the above conditions, finds the shortest path to the exit.

Algorithm 4 ExploreWithTreasureMinCost(maze, entrance, exit)

$Q \leftarrow$ queue with accessible neighbors from entrance
$visited \leftarrow \emptyset$, $path \leftarrow$ [entrance]
$treasuresFound \leftarrow 0$
while Q not empty **and** (cellsExplored $< 0.4 \cdot N$ **or** treasuresFound $= 0$) **do**
    $curr \leftarrow$ Q.pop()
    if curr not in visited **then**
        path.append(curr); visited.add(curr)
        if curr has treasure and fits in knapsack **then**
            Add treasure to knapsack; treasuresFound++
        for neighbor in neighbors(curr) **do**
            if no wall and neighbor not in visited **then**
                Q.push(neighbor)
end while
Append shortest path to exit
return path

Complexity Analysis

Let $N = n \cdot m$ be the number of cells in the maze, T the number of treasures, and C the knapsack capacity.

- **Exploration:** In the worst case, the agent visits up to $O(N)$ cells. Each move checks $O(1)$ neighbors, so exploration is $O(N)$.
- **Treasure Evaluation:** Discovered treasures are checked using a greedy selection or simple capacity constraint, taking $O(1)$ time per treasure.
- **Exit Pathfinding:** The agent uses BFS (or Dijkstra if weighted), taking $O(|E|+|V|\log|V|)$ time. In grid mazes, this simplifies to approximately $O(N \log N)$.

Total Time Complexity:

$$\mathcal{O}(N + |E| + |V|\log|V|) \approx \mathcal{O}(N \log N)$$

Balancing Exploration and Treasure Value

Our strategy enforces a minimum exploration threshold (e.g., 40% of the maze) to increase the probability of encountering treasures. Once at least one treasure is found or a value threshold is met, the agent may begin planning its exit path. This ensures:

- **Reward Potential:** A high chance of capturing valuable items without exhaustive search.
- **Efficiency:** Prevents unnecessary traversal through low-value regions.
- **Feasibility:** Guarantees eventual exit regardless of discovery success.

Assumptions and Handling of Probability Models

Our exploration strategy assumes a uniform probability distribution across all cells in the maze. We will treat each unexplored cell as equally likely to contain a treasure. This simplifies decision-making and allows us to treat each cell with equal exploratory priority, guided by a heuristic balance between expected value and distance from the current position. This assumption performs reasonably well in mazes where treasure placement is random and evenly distributed.

However, in cases where treasure placement follows a spatial bias—such as being more likely in cells farther from the entrance or in isolated clusters then this assumption can lead to suboptimal exploration. The current strategy may prematurely favour paths that lead directly to the exit without adequately investigating high-value areas that appear less accessible or more distant.

To improve performance in biased environments, our strategy could be extended to incorporate spatial priors. These priors would reflect the estimated probability of a cell containing treasure based on its location or historical patterns observed during exploration. Such enhancements would enable more informed decision-making, allowing the agent to prioritize areas more likely to yield valuable rewards, even if they require deeper traversal. While our current model is well-suited to uniform distributions, integrating adaptive or probabilistic heuristics could significantly improve its robustness in real world scenarios.