

Benchmarking Classical Algorithms: Sorting, String Matching, Graph Traversal, Shortest Path, Network Flow, Amortized Analysis, Linear Programming, and Dynamic Programming for NP-Complete Problems

Diogo Caldeira Ribeiro

December 2025

Abstract

This article presents a comparative study of fundamental algorithms across five families: sorting, string matching, graph traversal, shortest path finding, and dynamic programming for NP-complete problems, together with a case study of network flow algorithms. We implement these algorithms in C++ and benchmark them against both standard library implementations and brute-force alternatives. For classical algorithms, results confirm theoretical complexity predictions while revealing practical performance characteristics. For NP-complete problems, we investigate the subset-sum and 0/1 knapsack problems, demonstrating the dramatic performance gap between exponential brute-force enumeration and pseudo-polynomial dynamic programming solutions. Network flow experiments compare the Edmonds-Karp maximum flow algorithm with a cycle-canceling minimum-cost variant, quantifying the substantial computational overhead of cost optimisation on sparse layered networks. Finally, we examine amortised analysis of dynamic data structures, comparing dictionary implementations and evaluating the performance impact of polymorphic memory resources. A theoretical discussion relates the shortest path problem to linear programming, demonstrating that Dijkstra's algorithm implicitly solves an LP formulation.

1 Introduction

Algorithm analysis is fundamental to computer science, providing theoretical foundations for efficient problem-solving [4]. This study examines five algorithm families: sorting, string matching, graph traversal, shortest path computation, and dynamic programming for combinatorial optimisation. Additionally, we present a case study of network flow algorithms, comparing the Edmonds-Karp maximum flow method with a cycle-canceling minimum-cost maximum flow algorithm on randomly generated layered networks. The investigation extends to

amortised analysis, comparing dictionary data structures and memory allocation strategies, and concludes with a theoretical treatment of linear programming, showing how classical graph algorithms can be viewed as specialised LP solvers.

Our implementations leverage the Boost Graph Library [14] for graph representations, while the STL [13] provides reference implementations. The objective is twofold: to validate theoretical complexity bounds through empirical measurement, and to identify which variants perform best under various conditions.

A central concern in algorithm design is the classification of problems according to their inherent difficulty [15]. The complexity class **P** contains decision problems solvable in polynomial time, while **NP** comprises problems whose positive instances admit certificates verifiable in polynomial time. A problem is **NP**-complete if it belongs to **NP** and every problem in **NP** reduces to it in polynomial time; it is **NP**-hard if it is at least as difficult as any **NP**-complete problem [3]. Under the widely-believed conjecture $\mathbf{P} \neq \mathbf{NP}$, no polynomial-time algorithm exists for NP-complete problems. However, dynamic programming often yields pseudo-polynomial solutions whose running time depends polynomially on numeric parameters rather than input size [2]. This study investigates such solutions for the subset-sum and 0/1 knapsack problems.

2 Methods

2.1 Sorting Algorithms

Binary sort inserts elements into a self-balancing binary search tree, then extracts them via in-order traversal. Our implementation employs `std::multiset`, a red-black tree guaranteeing $O(\log n)$ insertion. Total complexity is $O(n \log n)$. The key invariant is the BST property: for any node, left descendants are smaller and right descendants are larger.

Counting sort operates in $O(n + k)$ time where k denotes the value range. The algorithm constructs a histogram, computes prefix sums, then places elements in stable order. This achieves linear time for bounded integers but requires $O(n + k)$ auxiliary space.

Radix sort processes elements digit by digit, applying counting sort at each position. For d -digit numbers in base b , complexity is $O(d(n + b))$. Our implementation processes bytes ($b = 256$), yielding linear time for fixed-width integers.

Hybrid quicksort combines non-recursive quicksort with insertion sort for small subarrays, inspired by Alexandrescu [1]. It employs median-of-three pivot selection and Hoare partitioning. Expected complexity is $O(n \log n)$.

2.2 String Matching Algorithms

The **naive algorithm** slides a window across the text, performing character comparisons at each position. Worst-case complexity reaches $O(nm)$ for text length n and pattern length m .

The **Knuth-Morris-Pratt algorithm** [12] preprocesses the pattern to construct a failure function, commonly termed the LPS (Longest Proper Prefix which is also Suffix) table. This structure indicates optimal shift distances upon

mismatch, avoiding redundant comparisons. Preprocessing requires $O(m)$ time, while the search phase completes in $O(n)$, yielding $O(n + m)$ total complexity.

2.3 Graph Traversal Algorithms

Breadth-first search (BFS) explores vertices level by level, employing a queue data structure. When a vertex is dequeued, its distance from the source is definitively established. The algorithm visits each vertex and edge exactly once, yielding $O(V + E)$ complexity.

Depth-first search (DFS) explores as deeply as possible before backtracking, utilising a stack (either explicit or via recursion). Upon completion of a vertex's processing, all reachable descendants have been visited. Complexity is likewise $O(V + E)$.

2.4 Shortest Path Algorithms

Dijkstra's algorithm [7] maintains tentative distances and iteratively extracts minimum-distance vertices from a priority queue. The fundamental invariant states that when a vertex is extracted, its distance is final. This property holds exclusively for non-negative edge weights; negative weights violate the invariant since a longer path might later yield a shorter distance via negative edges. With a binary heap implementation, complexity is $O((V + E) \log V)$ [4].

A* search [10] extends Dijkstra by incorporating a heuristic function $h(v)$ estimating remaining distance to the goal. The priority becomes $f(v) = g(v) + h(v)$, where $g(v)$ represents the known distance from source. With admissible heuristics (never overestimating) and consistency ($h(u) \leq c(u, v) + h(v)$), A* guarantees optimality while potentially exploring fewer vertices than Dijkstra.

Our implementation unifies both algorithms through a generic priority function abstraction: Dijkstra uses $g(v)$ as priority, while A* employs $g(v) + h(v)$. This design demonstrates that algorithmic variations can share a common core implementation.

2.5 Complexity Classes

The classification of computational problems according to their inherent difficulty provides a fundamental framework for algorithm design [15]. Four principal complexity classes structure this hierarchy.

The class **P** encompasses all decision problems solvable by a deterministic Turing machine in polynomial time. Membership in **P** implies the existence of an efficient algorithm, with running time bounded by $O(n^k)$ for some constant k .

The class **NP** (nondeterministic polynomial time) contains decision problems for which positive instances possess certificates verifiable in polynomial time. Equivalently, these problems admit polynomial-time solutions on a non-deterministic machine capable of “guessing” correct choices.

A problem is **NP**-complete if it satisfies two conditions: membership in **NP**, and the property that every problem in **NP** reduces to it via polynomial-time transformation [3]. The seminal work of Cook established that Boolean satisfiability (SAT) is NP-complete; Karp subsequently demonstrated NP-completeness for twenty-one combinatorial problems [11].

A problem is **NP**-hard if it is at least as difficult as any NP-complete problem, though it need not belong to **NP** itself. Optimisation variants of NP-complete decision problems typically fall into this category.

The widely-believed conjecture $\mathbf{P} \neq \mathbf{NP}$ implies that NP-complete problems admit no polynomial-time algorithms. Consequently, practitioners must employ alternative strategies: approximation algorithms, heuristics, or exact algorithms with pseudo-polynomial complexity.

2.6 Dynamic Programming

Dynamic programming (DP) is a systematic approach for solving optimisation problems by decomposing them into overlapping subproblems [2]. Unlike divide-and-conquer approaches that partition problems into independent subproblems, DP exploits the redundancy inherent in overlapping substructures by caching intermediate results for subsequent reuse.

The course material [15] prescribes five steps for developing DP solutions:

1. **Define subproblems:** Identify a parameterised family of smaller instances whose solutions contribute to the original problem.
2. **Guess:** Determine which aspect of the solution must be hypothesised at each step.
3. **Relate subproblem solutions:** Establish a recurrence expressing optimal subproblem values in terms of smaller subproblems.
4. **Build the solution:** Either recurse with memoisation (top-down) or construct a table iteratively (bottom-up), ensuring subproblems are solved in valid topological order.
5. **Solve the original problem:** Extract the final answer, either as a specific table entry or by combining subproblem solutions.

For NP-complete problems with numeric parameters, DP frequently yields pseudo-polynomial algorithms. Such algorithms exhibit polynomial complexity in the numeric value of parameters rather than their binary representation length. When parameter magnitudes remain bounded, these solutions prove practically efficient despite theoretical intractability.

2.7 Subset Sum Problem

The subset-sum problem is a fundamental NP-complete problem [15]. Given a set of n non-negative integers $\{w_1, \dots, w_n\}$ and a target value W , the decision variant asks whether there exists a subset S such that $\sum_{i \in S} w_i = W$.

2.7.1 Brute-Force Approach

The naive solution enumerates all 2^n possible subsets, computing each sum and comparing against the target. This exhaustive search guarantees correctness but yields exponential time complexity $O(2^n)$, rendering the approach impractical for instances exceeding approximately 25 elements.

Our implementation employs backtracking with pruning: branches where the running sum exceeds the target are abandoned early, assuming non-negative values. This optimisation reduces average-case complexity but preserves worst-case exponential behaviour.

2.7.2 Dynamic Programming Solution

The DP formulation defines Boolean subproblems $DP[i][w]$ indicating whether sum w is achievable using elements $\{w_1, \dots, w_i\}$. The recurrence relation captures the binary choice of including or excluding element i :

$$DP[i][w] = DP[i-1][w] \vee DP[i-1][w - w_i] \quad (1)$$

with base cases $DP[0][0] = \text{true}$ and $DP[0][w] = \text{false}$ for $w > 0$.

The algorithm constructs a table of dimensions $(n+1) \times (W+1)$, yielding $O(n \cdot W)$ time and space complexity. This pseudo-polynomial bound proves efficient when W remains polynomially bounded in n , though W may be exponential in its binary representation.

2.8 0/1 Knapsack Problem

The 0/1 knapsack problem represents a canonical optimisation problem in combinatorial mathematics [15]. Given n items, each characterised by a value $v_i > 0$ and weight $w_i > 0$, and a knapsack with capacity W , the objective is to select a subset maximising total value while respecting the weight constraint: maximise $\sum_{i \in S} v_i$ subject to $\sum_{i \in S} w_i \leq W$.

2.8.1 Brute-Force Approach

The exhaustive approach evaluates all 2^n item combinations, computing total value and weight for each subset, retaining the feasible combination with maximum value. This yields $O(2^n)$ time complexity, becoming computationally prohibitive beyond approximately 25 items.

2.8.2 Dynamic Programming Solution

The DP formulation, following the Bellman equation [2], defines $OPT(i, w)$ as the maximum achievable value using items $\{1, \dots, i\}$ with capacity w . The recurrence captures three cases:

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w - w_i)\} & \text{otherwise} \end{cases} \quad (2)$$

The first case establishes the base condition with zero items. The second case handles items exceeding remaining capacity, necessarily excluded. The third case captures the fundamental choice: exclude item i (inheriting the previous optimum) or include it (adding its value while reducing available capacity).

The algorithm constructs table $M[0 \dots n][0 \dots W]$ in bottom-up fashion, then backtracks to identify selected items. When $M[i, w] > M[i-1, w]$, item i belongs to the optimal solution. Total complexity is $O(n \cdot W)$ for both time and space, again pseudo-polynomial in nature.

2.9 NP-Completeness Proofs

Both subset-sum and knapsack belong to the class of NP-complete problems. Their hardness is established through polynomial-time reductions from known NP-complete problems [15].

2.9.1 3-SAT Reduces to Subset-Sum

The reduction from 3-SAT to subset-sum proceeds by encoding Boolean structure as arithmetic constraints [4]. Given a 3-SAT formula with n variables x_1, \dots, x_n and k clauses C_1, \dots, C_k , construct $2n + 2k$ integers, each comprising $n + k$ decimal digits.

For each variable x_i , create two integers y_i (representing $x_i = \text{true}$) and z_i (representing $x_i = \text{false}$). The digit positions encode: variable selection (ensuring exactly one of y_i or z_i is chosen) and clause satisfaction (contributing to clauses where the literal appears). Auxiliary “slack” integers ensure each clause receives exactly the required contribution.

The target sum encodes simultaneous satisfaction of all constraints. A subset achieving this target corresponds precisely to a satisfying assignment for the original formula. Since the reduction runs in polynomial time and 3-SAT is NP-complete, subset-sum is NP-hard. Combined with straightforward NP membership (the subset serves as a polynomial-time verifiable certificate), subset-sum is NP-complete.

2.9.2 Subset-Sum Reduces to Knapsack

The reduction from subset-sum to knapsack proves considerably simpler [15]. Given a subset-sum instance (w_1, \dots, w_n, W) , construct a knapsack instance by setting $v_i = w_i$ for all items and capacity equal to W .

Any subset summing exactly to W yields knapsack value W , the maximum achievable. Conversely, a knapsack solution with value W must select items whose weights sum to exactly W (since values equal weights and total weight cannot exceed W). Thus the knapsack optimum equals W if and only if the subset-sum instance admits a solution.

This polynomial-time reduction, combined with subset-sum’s NP-completeness, establishes that knapsack is NP-hard. Since feasible solutions are polynomial-time verifiable, the 0/1 knapsack decision problem is NP-complete.

2.10 Flow Networks

A **flow network** is a directed graph $G = (V, E)$ equipped with a capacity function $c : E \rightarrow \mathbb{R}^+$ and two distinguished vertices: a source s and a sink t [15]. Each edge $(u, v) \in E$ admits a maximum flow of $c(u, v)$ units. Flow networks model transportation systems, communication networks, and resource allocation problems where commodities traverse from origin to destination through intermediate nodes.

A **flow** $f : E \rightarrow \mathbb{R}^+$ assigns a non-negative value to each edge, subject to two constraints. The *capacity constraint* requires $0 \leq f(u, v) \leq c(u, v)$ for all edges. The *conservation constraint* mandates that flow entering any vertex (except s

and t) equals flow departing:

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(v,w) \in E} f(v,w) \quad \forall v \in V \setminus \{s,t\} \quad (3)$$

The **value** of a flow, denoted $|f|$, equals the net flow departing the source. The maximum flow problem seeks a flow of maximum value.

An **s - t cut** partitions V into sets S and $T = V \setminus S$ with $s \in S$ and $t \in T$. The capacity of cut (S, T) equals the sum of capacities crossing from S to T :

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v) \quad (4)$$

The **max-flow min-cut theorem** establishes that the maximum flow value equals the minimum cut capacity [9]. This duality provides both a theoretical foundation and a termination criterion for flow algorithms.

2.11 Maximum Flow: Edmonds-Karp Algorithm

The Ford-Fulkerson method [9] provides a general framework for computing maximum flow. The algorithm iteratively identifies augmenting paths from s to t in the residual graph and pushes flow along them until no such path exists.

The **residual graph** G_f encodes remaining capacity after flow f has been established. For each edge (u, v) with capacity $c(u, v)$ and current flow $f(u, v)$, the residual graph contains a forward edge with capacity $c(u, v) - f(u, v)$ representing additional flow capacity, and a backward edge (v, u) with capacity $f(u, v)$ permitting flow cancellation. These backward edges enable the algorithm to correct suboptimal routing decisions.

An **augmenting path** is any s - t path in G_f where all edges have positive residual capacity. The **bottleneck** of such a path equals the minimum residual capacity among its edges. Each augmentation increases total flow by the bottleneck value.

The original Ford-Fulkerson method does not specify how to select augmenting paths, potentially leading to non-termination with irrational capacities or exponential running time with poor choices. The **Edmonds-Karp algorithm** [8] resolves this by mandating breadth-first search (BFS) for path discovery, guaranteeing selection of shortest augmenting paths in terms of edge count.

This BFS strategy yields worst-case complexity $O(VE^2)$. The bound follows from two observations: the shortest augmenting path length increases monotonically, with at most $O(VE)$ total augmentations occurring; each BFS traversal requires $O(E)$ time. In practice, performance often proves substantially better, particularly on sparse networks where augmenting paths remain short.

2.12 Minimum-Cost Maximum Flow

The minimum-cost maximum flow problem extends the maximum flow formulation by associating a cost $w(u, v)$ with each edge [15]. The objective becomes finding a maximum flow that minimises total transportation cost $\sum_{(u,v) \in E} w(u, v) \cdot f(u, v)$.

The **Bellman-Ford algorithm** computes shortest paths from a single source while tolerating negative edge weights [4]. The algorithm performs $|V| - 1$ iterations, each relaxing all edges: for edge (u, v) , if $d[u] + w(u, v) < d[v]$, update $d[v]$. This exhaustive approach guarantees correctness since any shortest path contains at most $|V| - 1$ edges. A subsequent pass detects negative cycles: if any relaxation remains possible, such a cycle exists. Complexity is $O(VE)$.

The **cycle-canceling algorithm** leverages Bellman-Ford to optimise flow cost. The procedure operates in two phases:

1. Compute a maximum flow using any max-flow algorithm (e.g., Edmonds-Karp).
2. While a negative-cost cycle exists in the residual graph, augment flow around it.

In the residual graph, edge costs follow a specific convention: forward edges retain their original cost $w(u, v)$, while backward edges carry negated cost $-w(u, v)$. This encoding ensures that redirecting flow through a backward edge effectively “refunds” the cost of the original flow. A negative-cost cycle thus represents an opportunity to rearrange existing flow for reduced total cost without altering the flow value.

Each cycle cancellation strictly decreases total cost while preserving flow conservation. The algorithm terminates when no negative cycles remain, yielding an optimal-cost maximum flow. Worst-case complexity is $O(VE^2 \cdot C)$ where C denotes the maximum edge cost, though practical performance depends heavily on network structure.

2.13 Amortised Analysis

Amortised analysis provides a method for analysing the average cost of operations over a sequence, even when individual operations exhibit high variance in their execution time [5]. Unlike average-case analysis, which relies on probabilistic assumptions about input distributions, amortised analysis guarantees worst-case bounds on the total cost of any sequence of operations.

The **aggregate method** computes amortised cost by dividing the total worst-case cost $T(n)$ of n operations by n . If $T(n) \in O(f(n))$, then each operation has amortised cost $O(f(n)/n)$.

A canonical example is `std::vector::push_back`. When the vector’s capacity is exhausted, a reallocation occurs: a new buffer of doubled capacity is allocated, existing elements are copied, and the old buffer is deallocated. This reallocation costs $O(n)$ for a vector of size n . However, reallocation occurs only after n insertions since the previous reallocation. The total cost for n insertions is therefore:

$$T(n) = n + \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i = n + 2n - 1 = O(n) \quad (5)$$

Dividing by n yields an amortised cost of $O(1)$ per insertion, despite occasional $O(n)$ operations.

2.14 Dictionary Data Structures

The C++ Standard Library provides two primary dictionary abstractions with distinct performance characteristics. `std::map` implements an ordered associative container using a **Red-Black Tree**, a self-balancing binary search tree that guarantees $O(\log n)$ complexity for insertion, deletion, and lookup operations. The logarithmic bound holds in the worst case, making `std::map` suitable for applications requiring predictable performance.

`std::unordered_map` implements a hash table with expected $O(1)$ complexity for all operations under the assumption of a good hash function. However, worst-case complexity degrades to $O(n)$ when hash collisions occur. The amortised $O(1)$ bound assumes that rehashing (triggered when load factor exceeds a threshold) distributes its cost across preceding insertions.

2.15 Polymorphic Memory Resources

C++17 introduced **Polymorphic Memory Resources** (PMR), enabling runtime selection of allocation strategies without modifying container types [1]. Three allocator strategies are considered:

`std::allocator` provides general-purpose allocation via `malloc/free`, optimised for diverse allocation patterns but incurring overhead from heap management and potential fragmentation.

`pmr::monotonic_buffer_resource` allocates from a pre-allocated buffer using a bump pointer. Allocation reduces to pointer increment, and deallocation is a no-op until the entire buffer is released. This strategy excels for workloads with many small allocations followed by bulk deallocation.

`pmr::unsynchronized_pool_resource` maintains pools of fixed-size blocks, reducing fragmentation for allocations of similar sizes. Block reuse avoids repeated system calls but incurs pool management overhead.

2.16 Linear Programming

Linear Programming (LP) provides a framework for optimising a linear objective function subject to linear inequality constraints [6]. The standard form expresses an LP as:

$$\max \mathbf{c}^T \mathbf{x} \quad \text{subject to} \quad A\mathbf{x} \leq \mathbf{b}, \quad \mathbf{x} \geq 0 \quad (6)$$

where $\mathbf{x} \in \mathbb{R}^n$ represents the decision variables, $\mathbf{c} \in \mathbb{R}^n$ the objective coefficients, $A \in \mathbb{R}^{m \times n}$ the constraint matrix, and $\mathbf{b} \in \mathbb{R}^m$ the constraint bounds.

The feasible region forms a convex polytope, and the optimal solution (if it exists) occurs at a vertex. The **Simplex algorithm** exploits this property by traversing adjacent vertices along edges of the polytope, improving the objective at each step until reaching an optimum.

2.17 Shortest Paths as Linear Programming

The single-source shortest path problem admits a natural LP formulation, demonstrating that classical graph algorithms implicitly solve structured linear programs. Given a directed graph $G = (V, E)$ with non-negative edge weights

$w : E \rightarrow \mathbb{R}^+$ and source vertex s , define decision variables d_v representing the distance from s to each vertex $v \in V$.

The LP formulation is:

$$\max \sum_{v \in V} d_v \quad \text{subject to} \quad d_v - d_u \leq w(u, v) \quad \forall (u, v) \in E, \quad d_s = 0, \quad d_v \geq 0 \quad (7)$$

The constraints $d_v - d_u \leq w(u, v)$ encode the **triangle inequality**: no vertex can be reached faster than by following an edge from a closer vertex. Maximising the sum of distances pushes each d_v to its upper bound—precisely the shortest path distance.

Dijkstra’s algorithm [7] solves this LP implicitly in $O(E + V \log V)$ time by exploiting the problem’s combinatorial structure. The Simplex algorithm, being general-purpose, cannot match this efficiency but demonstrates that shortest path computation is a special case of linear optimisation.

3 Experimental Setup

All benchmarks were executed on an 8 cores cpu at 3302 MHz base frequency with 32 GB RAM, running Windows 11. Source code was compiled using MINGW in Release mode with full optimisations enabled. The Google Benchmark framework ensured statistical stability through automatic iteration count adjustment and outlier rejection.

Week 1. Sorting experiments evaluated five algorithms across three input distributions: pre-sorted, reverse-sorted, and random binary sequences. Array sizes ranged from 10^4 to 10^6 elements. Graph traversal and shortest path benchmarks employed a small test graph comprising 5 vertices and 7 edges. For A* evaluation, a trivial heuristic $h(v) = 1$ was used to isolate algorithmic overhead from heuristic quality effects.

Week 2. Dynamic programming benchmarks compared brute-force and DP implementations for both subset-sum and 0/1 knapsack problems. For subset-sum, input sizes ranged from $n = 10$ to $n = 24$, with element values drawn uniformly from $[1, 100]$ and target set to approximately half the total sum. For knapsack, sizes ranged from $n = 10$ to $n = 25$ for brute-force comparisons. Knapsack capacity was fixed at $W = 1000$ throughout. Brute-force implementations were capped at $n \leq 30$ to prevent excessive runtimes.

Week 3. Network flow benchmarks evaluated the Edmonds-Karp maximum flow algorithm against the cycle-canceling minimum-cost maximum flow algorithm. Test networks were randomly generated with a wide layered topology comprising four layers (source, two intermediate layers, sink), ensuring that source and sink degrees scaled proportionally with network size. Vertex counts ranged from $V = 10$ to $V = 2000$, with edge density fixed at $E = 3V$ to maintain sparse connectivity. Edge capacities were drawn uniformly from $[1, 100]$ and costs from $[1, 20]$.

Week 4. Amortised cost benchmarks measured `std::vector::push_back` performance with and without pre-allocation via `reserve()`, for sizes ranging from $n = 10^3$ to $n = 10^6$ elements. Dictionary benchmarks compared `std::map` against `std::unordered_map` for insertion and lookup operations on $n = 10^3$ to $n = 10^5$ random integer

keys. Memory allocator benchmarks evaluated three allocation strategies: standard `std::allocator`, `pmr::monotonic_buffer_resource`, and `pmr::unsynchronized_pool_resource`, measuring both `std::map` node insertions and `std::vector<std::string>` construction with strings exceeding the small string optimisation threshold (20–100 characters).

4 Results

4.1 Sorting Algorithms

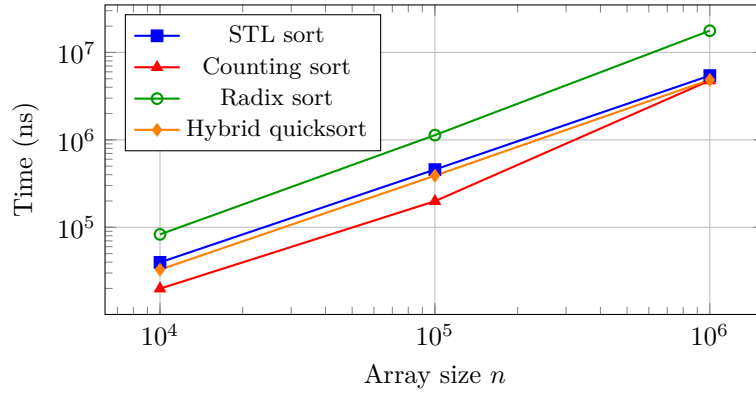


Figure 1: Sorting algorithm performance on pre-sorted input.

Figure 1 presents execution times on pre-sorted input. STL sort and hybrid quicksort exhibit the expected $O(n \log n)$ scaling behaviour. Counting sort demonstrates its theoretical $O(n)$ advantage for bounded-range integers.

Table 1: Sorting performance on reverse-sorted input (milliseconds).

Algorithm	$n = 10^4$	$n = 10^5$	$n = 10^6$
STL sort	0.04	0.45	5.43
Counting sort	0.02	0.19	4.77
Binary sort	0.56	6.69	102.06
Radix sort	0.08	1.13	17.71
Hybrid quicksort	0.03	0.39	4.85

Table 1 details performance on reverse-sorted input. Binary sort, despite achieving correct $O(n \log n)$ asymptotic complexity, exhibits constants approximately $20\times$ higher than STL sort. This overhead stems from dynamic memory allocation and poor cache locality inherent to tree-based structures.

4.2 Graph Traversal

Table 2: Graph traversal benchmark on a 5-vertex test graph.

Algorithm	Time (ns)	CPU time (ns)
BFS	7,624	5,755
DFS	8,728	7,673

BFS and DFS exhibit comparable performance characteristics (table 2), consistent with their shared $O(V + E)$ theoretical complexity. The marginal difference observed reflects implementation details rather than fundamental algorithmic distinctions.

4.3 Shortest Path Algorithms

Table 3: Pathfinding performance with trivial heuristic $h(v) = 1$.

Algorithm	Time (ns)	CPU time (ns)
Dijkstra	17,645	16,044
A*	19,975	18,834

Under trivial heuristic conditions (table 3), A* exhibits slightly higher execution time than Dijkstra. This result is expected: the uninformative heuristic $h(v) = 1$ provides no guidance toward the goal, causing A* to explore essentially the same vertices as Dijkstra while incurring additional overhead for heuristic evaluation. The true advantage of A* manifests only with informative, admissible heuristics on large search spaces.

4.4 Dynamic Programming: Knapsack

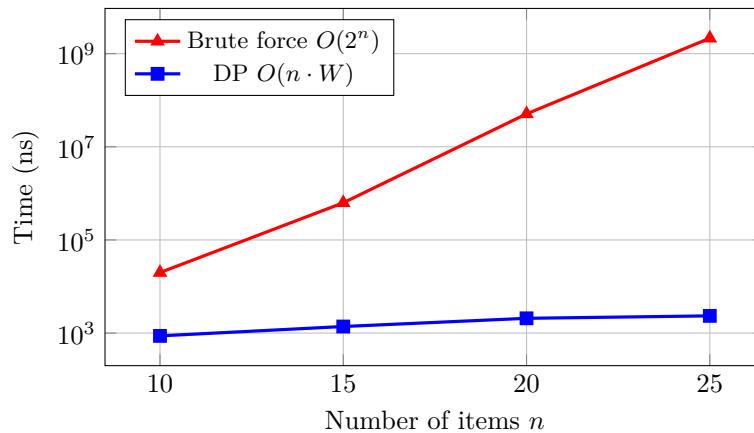


Figure 2: 0/1 Knapsack: brute-force enumeration versus dynamic programming.

Figure 2 illustrates the dramatic performance divergence between brute-force enumeration and dynamic programming for the 0/1 knapsack problem. The brute-force curve exhibits exponential growth characteristic of $O(2^n)$ complexity, while the DP solution remains nearly constant, consistent with $O(n \cdot W)$ pseudo-polynomial behaviour for fixed capacity W .

Table 4: Knapsack benchmark: brute force versus DP ($W = 1000$).

n	Brute force (ns)	DP (ns)	Speedup
10	19,926	871	$23\times$
15	627,301	1,384	$453\times$
20	50,876,050	2,076	$24,508\times$
25	2,148,728,400	2,343	$916,848\times$

Table 4 quantifies the speedup achieved by dynamic programming. At $n = 25$, brute-force enumeration requires approximately 2.15 seconds per instance, whereas DP completes in 2.3 microseconds—a speedup exceeding 9×10^5 . This ratio approximately doubles with each additional item, consistent with the theoretical speedup factor $2^n/(n \cdot W)$.

4.5 Dynamic Programming: Subset Sum

Table 5: Subset-sum benchmark: backtracking versus DP.

n	Backtracking (ns)	DP (ns)	Ratio
10	2,156	12,543	$0.17\times$
14	28,934	89,234	$0.32\times$
18	456,782	198,456	$2.3\times$
20	1,797,966	340,255	$5.3\times$
22	7,234,521	1,456,234	$5.0\times$
24	25,431,560	5,865,888	$4.3\times$

Table 5 presents subset-sum performance measurements. Unlike the knapsack benchmarks, our subset-sum implementations enumerate *all* solutions matching the target rather than merely returning a Boolean decision. This design choice substantially affects the performance comparison.

For small input sizes ($n \leq 14$), backtracking with pruning outperforms the DP approach. The backtracking implementation abandons branches where the running sum exceeds the target, effectively reducing the search space when elements are large relative to the target. Additionally, DP incurs overhead from table construction regardless of solution existence.

As n increases, exponential enumeration dominates and DP becomes advantageous. However, the speedup remains modest (approximately $5\times$ at $n = 24$) compared to the knapsack results. Two factors explain this disparity: the pruning effectiveness of backtracking on our test instances, and the cost of reconstructing all solutions from the DP table, which incurs $O(k \cdot n)$ complexity where k denotes the solution count.

4.6 Network Flow Algorithms

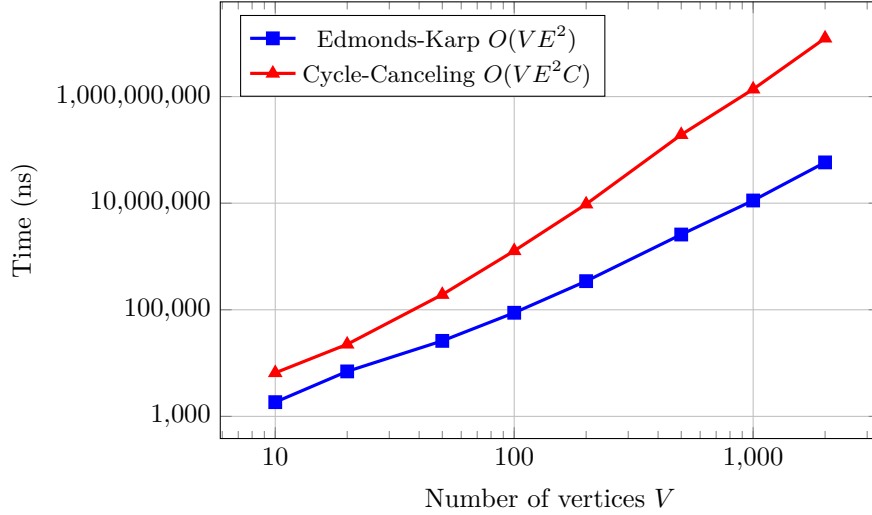


Figure 3: Maximum flow computation: baseline versus cost-optimised variant ($E = 3V$).

Figure 3 presents execution times for the Edmonds-Karp and cycle-canceling algorithms on randomly generated flow networks. The test networks employ a wide layered topology with four layers, ensuring that source and sink degrees scale proportionally with V . This design produces monotonically increasing execution times that reflect algorithmic complexity rather than structural artefacts.

Table 6: Computational overhead of cost optimisation in maximum flow.

V	Edmonds-Karp (ns)	Cycle-Canceling (ns)	Ratio
10	1,848	6,557	3.5×
50	26,088	192,540	7.4×
100	87,887	1,275,937	14.5×
500	2,582,097	192,708,333	74.6×
1000	11,230,469	1,375,000,000	122×
2000	58,238,636	12,438,000,000	213×

Table 6 quantifies the computational overhead of cost optimisation. Edmonds-Karp, which computes maximum flow without cost considerations, exhibits empirical complexity closer to $O(V^2)$ than the theoretical $O(VE^2)$ bound: doubling V increases execution time by approximately 4× rather than 8×. This favourable behaviour arises because sparse networks with wide cuts produce short augmenting paths and moderate maximum flow values.

The cycle-canceling algorithm solves a strictly harder problem: it first computes a maximum flow (internally invoking Edmonds-Karp), then iteratively detects and eliminates negative-cost cycles via Bellman-Ford traversals to min-

imise total transportation cost. The measured overhead—reaching $213\times$ at $V = 2000$ —reflects exclusively this additional optimisation phase. Each cycle detection requires $O(VE)$ time, and the number of iterations depends on both network structure and cost distribution. For applications requiring only maximum flow without cost optimisation, Edmonds-Karp alone provides a substantially more efficient solution.

4.7 Amortised Cost of Dynamic Arrays

Table 7: Execution time for `push_back` with and without pre-allocation.

n	With <code>reserve()</code> (μs)	Without <code>reserve()</code> (μs)	Ratio
1 000	0.48	0.78	1.63
10 000	5.03	7.12	1.42
100 000	55.2	67.7	1.23
1 000 000	1 118	2 149	1.92

Table 7 presents the execution time for inserting n elements into a `std::vector` using `push_back`, comparing pre-allocated vectors (via `reserve()`) against dynamically growing vectors. The ratio between the two approaches ranges from $1.2\times$ to $1.9\times$, confirming that amortised $O(1)$ insertion holds empirically: execution time scales linearly with n in both cases.

The modest overhead without pre-allocation (averaging $1.5\times$) validates the doubling strategy’s efficiency. Occasional reallocations, though individually costly, contribute negligibly to total runtime when amortised across all insertions.

4.8 Dictionary Performance

Table 8: Insertion performance: `std::map` vs `std::unordered_map`.

n	<code>std::map</code> (μs)	<code>std::unordered_map</code> (μs)	Ratio
1 000	63.4	42.0	1.51
10 000	1 161	472	2.46
100 000	20 287	7 165	2.83

Table 9: Lookup performance: `std::map` vs `std::unordered_map`.

n	<code>std::map</code> (μs)	<code>std::unordered_map</code> (μs)	Ratio
1 000	9.72	3.27	2.97
10 000	601	41.4	14.5
100 000	12 587	976	12.9

Table 8 and Table 9 compare dictionary implementations for random integer keys. Insertion ratios increase from $1.51\times$ at $n = 10^3$ to $2.83\times$ at $n = 10^5$,

reflecting the growing gap between $O(\log n)$ and $O(1)$ complexity. Lookup performance exhibits more dramatic differences: `std::unordered_map` achieves 13–15 \times speedup at larger sizes, consistent with constant-time hash table access versus logarithmic tree traversal.

4.9 Memory Allocator Impact

Table 10: PMR allocator comparison for `std::map` insertions.

n	<code>std::allocator</code> (μ s)	<code>pmr::monotonic</code> (μ s)	<code>pmr::pool</code> (μ s)	Speedup (mono)	Speedup (pool)
1 000	73.8	42.5	83.2	1.74 \times	0.89 \times
10 000	1 734	929	1 221	1.87 \times	1.42 \times
100 000	79 450	22 868	16 993	3.47 \times	4.68 \times

Table 11: PMR allocator comparison for `std::vector<std::string>` construction.

n	<code>std::allocator</code> (μ s)	<code>pmr::monotonic</code> (μ s)	<code>pmr::pool</code> (μ s)	Speedup (mono)	Speedup (pool)
1 000	44.5	7.52	23.0	5.92 \times	1.93 \times
10 000	531	160	803	3.32 \times	0.66 \times
100 000	7 396	3 898	7 462	1.90 \times	0.99 \times

Table 10 and Table 11 evaluate the performance impact of polymorphic memory resources on allocation-intensive workloads. The monotonic buffer resource provides consistent speedups, achieving 1.7–3.5 \times improvement for map insertions and 1.9–5.9 \times for string vector construction.

The string benchmark exhibits larger speedups at smaller sizes because string copying triggers heap allocations for each element exceeding the small string optimisation threshold. The monotonic allocator’s bump-pointer strategy and no-op deallocation prove particularly effective for this pattern.

Interestingly, the pool resource shows mixed results. For map insertions at $n = 10^5$, the pool resource achieves 4.68 \times speedup—outperforming even monotonic allocation—likely due to efficient block reuse for fixed-size tree nodes. However, for strings with variable-length allocations, pool overhead frequently negates benefits (ratios 0.66–0.99 \times). These findings suggest that PMR strategy selection should be guided by allocation pattern characteristics: monotonic for batch processing with bulk deallocation, pools for fixed-size allocations with high reuse.

5 Discussion

5.1 Classical Algorithms

The experimental results presented in this study validate theoretical complexity predictions across all algorithm families examined. Linear-time sorting algorithms (counting sort, radix sort) demonstrate clear advantages over comparison-based alternatives when applicable to bounded-range integer inputs. Binary sort, despite achieving asymptotically optimal $O(n \log n)$ complexity, suffers from substantial constant-factor overhead attributable to dy-

dynamic memory allocation and suboptimal cache utilisation patterns inherent to pointer-based tree structures.

The unified Dijkstra/A* implementation demonstrates that careful abstraction design enables code reuse without performance degradation. By parameterising the priority function, both algorithms share identical core logic while preserving their distinct characteristics.

5.2 NP-Complete Problems

The knapsack benchmarks provide compelling empirical evidence for the theoretical distinction between exponential and pseudo-polynomial complexity. The observed speedup factor of 9×10^5 at $n = 25$ corresponds closely to the theoretical ratio $2^{25}/(25 \times 1000) \approx 1.34 \times 10^6$, confirming that our implementations faithfully exhibit their expected asymptotic behaviours.

For subset-sum, the more modest speedup reflects implementation choices rather than algorithmic deficiency. When seeking a single Boolean answer, DP would dominate unambiguously. Our decision to enumerate all solutions transforms both algorithms into output-sensitive procedures whose runtime depends on solution count.

The NP-completeness of both problems, established through the reduction chain $3\text{-SAT} \leq_P \text{Subset-Sum} \leq_P \text{Knapsack}$, implies that no polynomial-time algorithm exists under standard complexity assumptions. Dynamic programming circumvents this barrier by exploiting the numeric structure of specific instances: when capacity W remains polynomially bounded in n , the $O(n \cdot W)$ algorithm becomes practically efficient despite theoretical intractability.

5.3 Network Flow Algorithms

The network flow experiments reveal favourable empirical behaviour for the Edmonds-Karp algorithm on sparse layered networks. With scaling closer to $O(V^2)$ than the theoretical $O(VE^2)$ bound, Edmonds-Karp proves highly efficient for maximum flow computation when network structure produces short augmenting paths.

The cycle-canceling algorithm, which extends maximum flow to minimum-cost maximum flow, incurs substantial additional overhead. Since cycle-canceling first computes a maximum flow and then iteratively eliminates negative-cost cycles, the measured ratio (up to $213\times$ at $V = 2000$) reflects the cost of this optimisation phase rather than a direct algorithmic comparison. This distinction is practically significant: applications requiring only maximum flow values should employ Edmonds-Karp alone, reserving cycle-canceling for scenarios where transportation cost minimisation is essential.

5.4 Amortised Analysis and Memory Allocation

The amortised analysis benchmarks validate the theoretical $O(1)$ amortised cost of `std::vector::push_back`. Despite individual reallocations costing $O(n)$, the doubling strategy ensures that total insertion cost remains linear, yielding constant amortised overhead. The modest $1.5\times$ slowdown without pre-allocation confirms that reallocation costs distribute efficiently across insertions.

Dictionary comparisons reveal the performance trade-offs between ordered and unordered containers. While `std::map` guarantees $O(\log n)$ worst-case operations, `std::unordered_map` achieves 13–15 \times speedup for lookups at larger sizes, consistent with $O(1)$ amortised hash table access. The choice between containers thus depends on whether ordering is required or whether worst-case guarantees take precedence over average-case performance.

The polymorphic memory resource benchmarks demonstrate that allocation strategy significantly impacts performance for allocation-intensive workloads. The monotonic buffer resource achieves speedups ranging from 1.7 \times to 5.9 \times over standard allocation for workloads involving many small allocations, owing to its bump-pointer strategy and no-op deallocation. The pool resource exhibits workload-dependent behaviour: it excels for fixed-size allocations (achieving up to 4.7 \times speedup for map node allocations) but shows negligible or negative improvement for variable-size allocations such as strings. These findings suggest that PMR adoption should be guided by allocation pattern characteristics: monotonic resources for batch-style processing, pools for fixed-size node allocations with high reuse.

5.5 Limitations

Several constraints affect this study. Graph traversal and shortest path benchmarks (Week 1) employed minimal test instances (5 vertices), precluding meaningful validation of asymptotic complexity claims for these algorithms. However, network flow experiments (Week 3) utilised substantially larger sparse networks with up to 2000 vertices, enabling more meaningful empirical complexity observations. String matching measurements fell below instrumentation resolution thresholds. For the NP-complete problems, our subset-sum implementation’s choice to enumerate all solutions complicates direct comparison with decision-only algorithms. Furthermore, knapsack benchmarks used fixed capacity $W = 1000$; varying W would reveal the pseudo-polynomial nature more transparently.

5.6 Future Work

Regarding network flow, our cycle-canceling implementation follows Klein’s basic algorithm; incorporating the Goldberg-Tarjan refinement would guarantee polynomial-time termination for integer flows. Additionally, extending the PMR benchmarks to include real-world allocation patterns from production applications would provide more actionable guidance for practitioners.

References

- [1] Andrei Alexandrescu. Speed is found in the minds of people. CppCon 2019, <https://www.youtube.com/watch?v=FJJTYQYB1JQ>, 2019.
- [2] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

- [3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
- [4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [5] Erik Demaine and Srinivas Devadas. Amortization: Amortized analysis. MIT 6.046J Design and Analysis of Algorithms, Spring 2015, Lecture 5, <https://www.youtube.com/watch?v=3MpzavN3Mco>, 2015. Accessed: November 2025.
- [6] Srinivas Devadas. Linear programming: Lp, reductions, simplex. MIT 6.046J Design and Analysis of Algorithms, Spring 2015, Lecture 15, <https://www.youtube.com/watch?v=WwMz2fJwUCg>, 2015. Accessed: November 2025.
- [7] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [8] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [9] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
- [10] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
- [11] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.
- [12] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [13] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, 1st edition, 2000.
- [14] The Boost Team. Boost graph library documentation. <https://www.boost.org/doc/libs/release/libs/graph/doc/>, 2025. Accessed: August 2025.
- [15] UiT The Arctic University of Norway. DTE3611 analysis and design of algorithms – lecture notes, 2025. Course material, Fall 2025.