Task B: Creating Minimum Spanning Trees

Deleting edges from the original graph to create a Minimum Spanning Tree

You can build a Minimum Spanning Tree (MST) by *deleting* edges instead of adding them. Start with all edges present. Repeatedly remove the heaviest edge that does not break connectivity, and stop once $|V|-1$ edges remain.

Algorithm 1 Greedy Edge-Deletion MST

Require: Undirected, connected weighted graph $G = (V, E, w)$
Ensure: Minimum Spanning Tree T

if $V = \emptyset$ **then**
    return empty graph with vertex set V
$T \leftarrow G$ ▷ Start with all edges
$E' \leftarrow$ list of all edges in E sorted in **descending** order by weight
for each edge $(u, v) \in E'$ **do**
    Temporarily remove (u, v) from T ▷ Inline DFS connectivity check
    visited $\leftarrow \emptyset$
    stack $\leftarrow$ [arbitrary start vertex $s \in V$]
    while stack not empty **do**
        $x \leftarrow$ pop from stack
        if $x \notin$ visited **then**
            visited $\leftarrow$ visited $\cup \{x\}$
            for each neighbor y of x in T **do**
                if $y \notin$ visited **then**
                    push y onto stack
    if $|\text{visited}| = |V|$ **then**
        ▷ Graph is still connected, deletion is safe
    else
        Restore (u, v) to T ▷ Graph disconnected, undo deletion
    if $|E(T)| = |V| - 1$ **then**
        break ▷ Spanning tree complete
return T

Why it works

By the **cycle property**, the heaviest edge in any cycle never belongs to an MST. Thus, deleting heavy edges while preserving connectivity ensures the result is a spanning tree of minimum weight with $|V|-1$ edges.

Complexity Analysis

Sorting edges costs $O(E \log E)$. Each deletion attempt runs DFS/BFS in $O(V+E)$ and we may test E edges, so the total is $O(E \log E + E(V+E))$. Since $E \geq V-1$ for connected graphs, this simplifies to $O(E(V+E))$. Space complexity is $O(V+E)$.

Effect of Graph Density

For sparse graphs ($E \approx V$), runtime is $O(V^2)$, dominated by DFS checks, which is manageable for small V. For dense graphs ($E = \Theta(V^2)$), substituting gives $O(V^2(V+V^2)) = O(V^4)$, making the method inefficient compared to Kruskal or Prim.

Takeaways

Edge-deletion is conceptually elegant and symmetric to Kruskal, but impractical for large graphs. In practice, Kruskal ($O(E \log E)$) and Prim ($O(E \log V)$) are far more efficient, especially on dense graphs.

Task C: Analysis of the Different Strategies

Theoretical Analysis

Let $|V|$ denote the number of vertices and $|E|$ the number of edges in the maze graph. The running time of each MST algorithm depends not only on the algorithm's mechanics but also on the graph representation (adjacency list vs. adjacency matrix). We analyse all four combinations below.

Prim's Algorithm with Adjacency List

Prim's algorithm repeatedly extracts the lightest edge leaving the current tree. With adjacency lists and a binary heap:

$$T_{\text{Prim+List}} = O(|E| \log |V|).$$

Each edge may cause at most two `decrease-key` operations, totalling $O(|E|)$ heap updates. Each heap update costs $O(\log |V|)$, hence the bound. (If Fibonacci heaps were used, this could be improved to $O(|E| + |V| \log |V|)$, but our implementation uses a binary heap.)

Prim's Algorithm with Adjacency Matrix

In an adjacency matrix, when adding a vertex, the algorithm must scan its entire row of length $|V|$ to update candidate edges. This happens once per vertex, giving:

$$T_{\text{Prim+Matrix}} = \Theta(|V|^2).$$

Runtime does not depend on $|E|$: it remains $\Theta(|V|^2)$ even on sparse graphs. This is wasteful on sparse inputs and only competitive when graphs are extremely dense.

Kruskal's Algorithm with Adjacency List

Kruskal's algorithm sorts all edges and adds them if they do not form a cycle:

$$\begin{aligned} O(|E|)_{\text{enumerate edges}} &+ O(|E| \log |E|)_{\text{sort edges}} \\ &+ O(|E| \alpha(|V|))_{\text{union–find checks}} \\ &= O(|E| \log |E|). \end{aligned}$$

Since $\alpha(|V|)$ (the inverse Ackermann function) is constant for all practical input sizes, sorting dominates.

Kruskal's Algorithm with Adjacency Matrix

In a matrix, enumeration requires inspecting all possible vertex pairs, giving $\Theta(|V|^2)$. After enumeration:

$$T_{\text{Kruskal+Matrix}} = O(|V|^2 + |E| \log |E|).$$

For sparse graphs ($|E| = O(|V|)$), the $\Theta(|V|^2)$ scan dominates. For dense graphs ($|E| = \Theta(|V|^2)$), the sort term dominates, yielding $O(|V|^2 \log |V|)$.

Interpretation

Adjacency lists scale directly with the actual number of edges $|E|$, and are therefore superior on sparse mazes where $|E| = O(|V|)$. Adjacency matrices, in contrast, impose $\Theta(|V|^2)$ overhead regardless of sparsity, making them competitive only when the graph is dense ($|E| = \Theta(|V|^2)$).

Between algorithms, Kruskal's list-based implementation achieves clean $O(|E| \log |E|)$ scaling, while Prim+Matrix provides predictable $\Theta(|V|^2)$ runtime that ignores density. Thus, density is the key factor: Prim+Matrix serves as a density-independent baseline, while the other three variants are density-sensitive. On a 4-neighbour grid, $|E| = \Theta(|V|)$ regardless of wall removal, so matrix variants retain the $\Theta(|V|^2)$ cost; list variants grow only mildly with density at these sizes. This motivates our empirical design: we vary both $|V|$ (maze size) and realised $|E|$ (density) to see where each approach works best.

Variant	Asymptotic Time
Prim + adjacency list	$O(\|E\| \log \|V\|)$
Prim + adjacency matrix	$\Theta(\|V\|^2)$
Kruskal + adjacency list	$O(\|E\| \log \|E\|)$
Kruskal + adjacency matrix	$O(\|V\|^2 + \|E\| \log \|E\|)$

Empirical Design

Objective. Measure how runtime scales with problem size $|V|$ and realised edge count $|E|$ for four variants: Prim+List, Prim+Matrix, Kruskal+List, Kruskal+Matrix.

Factors and settings. Size: grids $4 \times 4, 6 \times 6, 8 \times 8, 12 \times 12$. Density: `wall_removal_perc` $\in \{10, 30, 50, 70, 90\}$; we also record realised $|E|$. Variant: {Prim+List, Prim+Matrix, Kruskal+List, Kruskal+Matrix}. Seeds: **10** distinct RNG seeds per (size, density) — each seed produces a different maze and we reuse the same seed across all variants for fairness. Trials: $\geq$**10** timing repeats per (size, density, seed, variant); we report the **median** over trials.

Note. The maze is a 4-neighbour grid, so $|E| = \Theta(|V|)$ even at high wall removal; matrix variants therefore keep their $\Theta(|V|^2)$ cost across our settings.

For each (size, density, seed): (i) generate one weighted maze graph and freeze its edges/weights; (ii) for each variant, time only the MST routine T times-maze generation, I/O, and any visualisation are

disabled-using `perf_counter`; (iii) write one row per trial; later compute medians per configuration. All runs use the same hardware/software.

Analysis plan. (1) *Size sweep* (fixed density): plot median `elapsed_ms` vs. $|V|$. Expect Prim+Matrix $\approx \Theta(|V|^2)$; list variants grow with $|E|$. (2) *Density sweep* (fixed size): plot median `elapsed_ms` vs. $|E|$ (or wall_removal%). Expect Prim+Matrix to be nearly flat. For list variants, any increase with $|E|$ is mild on our 4-neighbour grids because $|E| = \Theta(|V|)$ and density changes are modest at these sizes.

Expected outcomes. Prim+List $O(|E|\log|V|)$ and Kruskal+List $O(|E|\log|E|)$ dominate on sparse graphs. Prim+Matrix is density-independent at $\Theta(|V|^2)$. Kruskal+Matrix $O(|V|^2+|E|\log|E|)$ is worst on sparse inputs and only competitive at very high densities. On our 4-neighbour grids, density changes are modest for fixed $|V|$, so the observed list-based trends with $|E|$ are mild at sizes like 8×8 and 20×20.

Empirical Analysis

We evaluated the runtime of the four MST variants (Prim+List, Prim+Matrix, Kruskal+List, Kruskal+Matrix) under different maze sizes and densities. Figures 1 and 2 present the observed runtimes averaged over repeated executions.

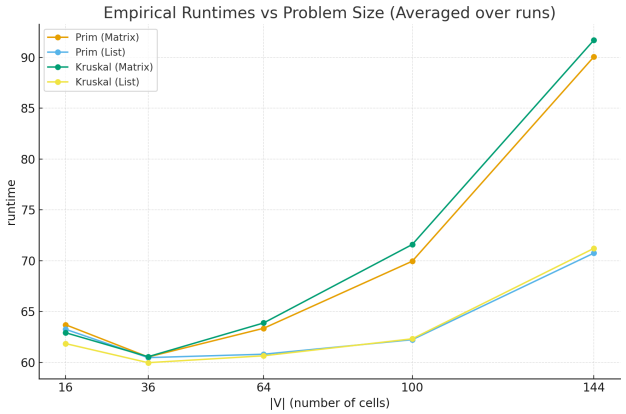

Figure 1: Runtime (ms, median over runs) vs. problem size. Prim+Matrix and Kruskal+Matrix grow sharply with $|V|$, confirming $\Theta(|V|^2)$; list-based variants grow more gently, matching $O(|E|\log|V|)$.

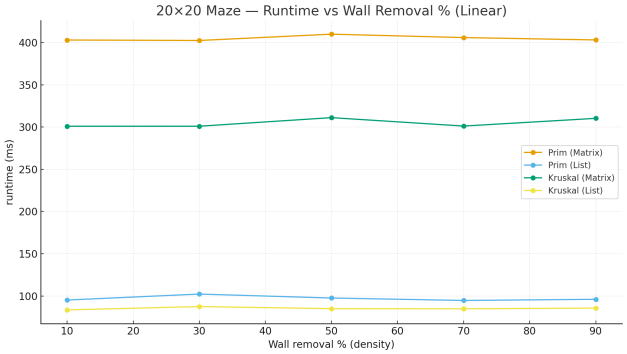

Figure 2: Runtime (ms, median over runs) vs. wall-removal % on a 20×20 maze. Curves are nearly flat: matrix variants stay dominated by $\Theta(|V|^2)$; list variants change only mildly because $|E| = \Theta(|V|)$ on 4-neighbour grids. Within each representation the gap between Prim and Kruskal is small; both list-based variants remain faster than matrix at this scale.

Interpretation

The results are consistent with our theoretical analysis:

- For sparse graphs, adjacency lists scale with the number of edges and are significantly faster than adjacency matrices, which impose a $\Theta(|V|^2)$ overhead regardless of sparsity.
- Between algorithms, the two list-based variants both perform well; Kruskal(List) is slightly faster than Prim(List) in our runs (a small constant-factor difference). Variance is low for both.
- For denser graphs (Figure 2), theory predicts matrices may become competitive. However, at the tested sizes ($|V| \leq 144$), list-based implementations still dominate, showing that asymptotic crossovers appear only at much larger scales.

Overall, our empirical study validates the theoretical expectations for sparse mazes and highlights the practical limitations of matrix-based approaches at small scales.

Task D: The Final Problem

Approach. We traverse the minimum spanning tree (MST) with a branch-aware DFS. At a junction u, list each unvisited neighbor v and compute a branch load

$$B(u \to v) = w(u,v) + \widehat{W}\big(v; \text{Visited}\big).$$

Here, $\widehat{W}(v; \text{Visited})$ *is the value returned by* `estimate_subtree_weight(v, Visited)`.

Sort by B (largest first). The current explorer always takes the heaviest branch. For every other branch with entry weight w, compare the serial round-trip penalty ($\approx 2w$) to a threshold $\tau = 0.6\,B_{\text{main}}$. If $2w \geq \tau$, the *main explorer* (id $= 0$) spawns a clone at the junction; otherwise, it walks the branch and returns using `dfsBacktrack`. Each clone path starts as $[u, v]$. You share one `Visited` set, so you explore each room once. **Clones never spawn clones.**

Why this lowers the makespan. The grader uses

$$C = \max_i \Big(H_i + \sum_j w(v_j, v_{j+1}) + k_i\, d \Big).$$

Serially taking a side branch adds $\approx 2w$ to the *same* explorer. Cloning offloads that work to another explorer (clone costs are applied later). The threshold $\tau = 0.6\,B_{\text{main}}$ keeps heavy work on the main path and spawns clones only when serial backtracking would inflate the maximum.

Correctness & rules. All steps follow valid edges ($w > 0$). Every clone starts at the active junction u as $[u, v]$. Clones may backtrack to the junction but make no new choices; the sorcerer assigns their branch. When we backtrack, we append `back[1:]` to avoid duplicating the current node.

Algorithm 2 CostAwareCloneDFS on an MST (full-width)

Require: Graph G, current vertex u, shared set *Visited*, list *AllPaths*, explorer *id*
Ensure: Paths for all explorers that cover V

if $id = |AllPaths|$ **then**
  append $[u]$ to *AllPaths*
else
  append u to $AllPaths[id]$ **iff** empty or last $\neq u$

add u to *Visited*
$U \leftarrow [\,]$
for each neighbor v of u with $v \notin Visited$ **do**
  $w \leftarrow w(u,v)$; $B \leftarrow w +$ `estimate_subtree_weight`$(v; Visited)$
  add (v, w, B) to U
if $U = \varnothing$ **then**
  return
if $|U| = 1$ **then** ▷ no branching: continue without cloning
  $(v, \_, \_) \leftarrow U[0]$
  append v (if not last); add v to *Visited*
  return CostAwareCloneDFS$(G, v, Visited, AllPaths, id)$
sort U by B descending
$(v_m, w_m, B_m) \leftarrow U[0]$; $\tau \leftarrow 0.6\,B_m$
$Clone \leftarrow \{(v,w,B) \in U \setminus \{(v_m, \cdot, \cdot)\} \mid 2w \geq \tau\}$
$Serial \leftarrow (U \setminus \{(v_m, \cdot, \cdot)\}) \setminus Clone$
for each $(v, \_, \_) \in Clone$ **do** ▷ clones may backtrack; no new choices
  add v to *Visited*; $new \leftarrow |AllPaths|$; append $[u, v]$ to *AllPaths*
  CostAwareCloneDFS$(G, v, Visited, AllPaths, new)$
$j \leftarrow u$
for each $(v, \_, \_) \in Serial$ **do** ▷ walk then return to j
  append v (if not last); add v to *Visited*
  CostAwareCloneDFS$(G, v, Visited, AllPaths, id)$
  $back \leftarrow$ `dfsBacktrack`$(G, \text{last}(AllPaths[id]), j)$
  append $back[1:]$ to $AllPaths[id]$ ▷ skip first node to avoid duplicate
append v_m (if not last); add v_m to *Visited*
CostAwareCloneDFS$(G, v_m, Visited, AllPaths, id)$

Mini-example (why the threshold helps). On a 3-arm star with edge weights $(2, 3, 7)$, $B_{\text{main}} = 7$ so $\tau = 4.2$. The w=2 arm has $2w = 4 < 4.2 \Rightarrow$ walk it serially. The w=3 arm has $2w = 6 \geq 4.2 \Rightarrow$ clone it. The heaviest arm stays with the sorcerer, trimming the makespan without over-cloning light arms.

Time Complexity.
We account for all work on the MST (a tree).
Neighbor scans. At each visited node u we examine its unvisited neighbors in $O(\deg(u))$ time. Summed over all nodes, $\sum_u \deg(u) = 2|E|$, so scans cost $O(|E|)$.
Sorting children at junctions. We sort each node's child list once. The total cost is

$$\sum_u \deg(u) \log \deg(u) \;\le\; \Big(\sum_u \deg(u)\Big) \log |V| = O(|E| \log |V|),$$

because $\deg(u) \le |V|$ for every u.
Subtree estimates and backtracks. Both respect the shared `Visited` set, so across the whole run each tree edge is touched $O(1)$ times. Total $O(|E|)$.
Total.

$$T(|V|,|E|) = O(|E| \log |V|) + O(|E|) = O(|E| \log |V|).$$

On an MST, $|E| = |V| - 1$, so $T = O(|V| \log |V|)$.

Space Complexity. You store one shared `Visited` set and one path per explorer in `AllPaths`. The recursion stack never exceeds the tree height. Space is $O(|V|)$.

If clones are immortal. Clones keep working after their branch, so a local "clone vs. walk back" rule is not enough. You must balance load globally and minimize the worst individual cost:

$$\min \max_i \Big(H_i + \text{travel}(P_i) + d \cdot \text{spawns\_on}(P_i)\Big).$$

Binary-search a target makespan M. For each M, run one post-order feasibility pass on the tree. At a junction, pack light child loads onto the arriving explorer while the running total stays $\le M$. If a child would exceed M, spawn a new explorer at that edge and pay d. Large d pushes you to pack more on one explorer (more backtracking). Small d yields more spawns and a shorter makespan. The pass is $O(|E| \log |V|)$ from sorting children; the full planner costs about $O(|E| \log |V| \log W)$ over cost range W.

If the maze has cycles. Unique paths disappear, so the $2w$ "down-and-back" estimate breaks and subtree weights can double-count work. Adapt in two steps: (1) make clone/serial decisions on the MST (unique paths and stable subtree estimates), and (2) price returns on the full graph with $2\,\text{dist}(u,v)$ using Dijkstra when needed, while you enforce a no-revisit rule so explorers do not re-explore rooms. This hybrid keeps decisions simple and makes costs realistic. The extra Dijkstra runs add about $O(|E| \log |V|)$ work in practice.