Minimum Camera Placement for Forest Monitoring (General Graph) – DP Algorithm

Mustafa Bozyel
Student ID: 32417
mustafa.bozyel@sabanciuniv.edu

1. Recursive Formulation (Task 1)

Let $G = (V, E)$ be an undirected graph (not necessarily a tree). Each shared region $\{u, v\}$ corresponds to an edge (u, v). Placing a camera at a cdp u monitors all shared regions incident to u. Thus, monitoring the whole forest means selecting a set $S \subseteq V$ such that for every $(u, v) \in E$, $u \in S$ or $v \in S$. This is exactly the **Minimum Vertex Cover** problem.

DP subproblem (state): Let $R \subseteq V$ be the set of *remaining (not yet selected)* vertices. Define $dp(R)$ as the minimum number of additional cameras needed to cover all edges whose both endpoints are in R (i.e., edges of the induced subgraph $G[R]$).

Recursive formulation: If $G[R]$ has no edges, then $dp(R) = 0$. Otherwise pick any uncovered edge (u, v) in $G[R]$. Any vertex cover must include u or v, so:

$$dp(R) = 1 + \min\big(dp(R \setminus \{u\}),\ dp(R \setminus \{v\})\big).$$

The answer is $dp(V)$.

Why tree/forest DP does not apply to general graphs

Tree-DP transitions rely on a parent/child decomposition. In a general graph with cycles (e.g., triangle 0–1–2–0), a DFS that only avoids the immediate parent can revisit an already-seen node through a different edge, causing either infinite recursion or double-counting. More importantly, the subproblem boundary "subtree" is not well-defined on cycles, so local DP states cannot summarize global constraints. This is consistent with theory: Minimum Vertex Cover is NP-hard on general graphs, so we should expect an exponential-time exact algorithm unless additional structure (like trees) is assumed.

Justification for DP

The recursion revisits the same subsets R through different branching orders. Memoizing $dp(R)$ turns the exponential recursion into a DP with a table over subsets. This guarantees correctness (exploring both necessary choices for each uncovered edge) and avoids recomputation.

2. Pseudocode (Task 1)

We represent R as a bitmask over vertices $0..n-1$ and memoize results. For reconstruction, we store which vertex was chosen at each state.

Algorithm 1 Exact DP for Minimum Vertex Cover

```
function SOLVE(R)
    if G[R] has no edge then
        return 0
    end if
    Pick any edge (u, v) inside R
    a ← 1 + SOLVE(R \ {u})
    b ← 1 + SOLVE(R \ {v})
    return min(a, b)                                  ▷ Memoize by R
end function
return SOLVE(V)
```

Reconstructing the chosen cdps

To output the actual camera locations (not just the minimum count), we store a decision for each state R: whether the optimal branch removed u or v. Starting from $R = V$, we replay decisions until $G[R]$ has no uncovered edges; all removed vertices form the selected cdp set.

3. Asymptotic Time Complexity (Task 2)

There are up to 2^n subsets R. Each DP state finds an uncovered edge (can be done in $O(n)$ with adjacency bitmasks) and makes two transitions. Thus worst-case time is $O(2^n \cdot n)$ (or $O(2^n \cdot \text{poly}(n+m))$). Space is $O(2^n)$ for memoization and reconstruction. This exponential complexity matches NP-hardness for general graphs.

Practical feasibility

For n
$approx$20–30, the exact DP can run in seconds to minutes depending on density and structure; for larger n it can take hours in the worst case. This aligns with the assignment remark asking for benchmark sizes ranging from "seconds" to "hours" (for an exact NP-hard solver, runtime grows very quickly with n).

4. Example (Task 3)

Sample graph with 5 cdps where degrees are 2–3 (each cdp can monitor 2–3 regions):

$$E = \{(0,1),(1,2),(2,0),(2,3),(3,4),(4,2)\}.$$

This is two triangles sharing vertex 2 (vertex 2 has degree 4; vertices 0,1,3,4 have degree 2).

Step-by-step DP branching:

- Start with $R = V = \{0,1,2,3,4\}$. Pick uncovered edge $(0,1)$.
- Branch A (choose 0): solve $R \setminus \{0\} = \{1,2,3,4\}$. Now edge $(1,2)$ is uncovered, so choose 1 or 2.
- Branch B (choose 1): symmetric to A.
- The optimal solution chooses vertex 2 plus one vertex from each triangle, e.g., $\{2,0,3\}$ (size 3) covers all edges.

The DP explores both endpoints for each uncovered edge and memoizes repeated subsets.

5. Functional Testing (Task 5)

We designed 7 test instances for white-box and black-box testing, targeting: (i) base cases (no edges), (ii) correctness on cycles (triangle/cycle-5), (iii) dense graphs (K5), and (iv) disconnected graphs. Outputs also verify the chosen set is a valid vertex cover.

Table 1: Functional Testing Results

Instance	**Expected**	**Actual**	**Status**	**Properties Tested**
Single node (no edges)	0	0	✓	Base case: empty edge set
Two nodes (one edge)	1	1	✓	Minimal edge must be covered
Triangle cycle	2	2	✓	Cycle handling, correctness
Star (n=5)	1	1	✓	High-degree vertex case
Cycle-5	3	3	✓	Odd cycle minimum cover
Disconnected (edge+triangle)	3	3	✓	Component independence
Complete graph K5	4	4	✓	Dense graph worst-case flavor

6. Computational Performance (Task 6)

Benchmark design: Because the exact DP is exponential in n, we benchmarked modest sizes. We generated 200 instances across 20 input sizes ($n = 10..29$), with 10 instances per size. Graph families: paths, cycles, random $G(n, p)$ graphs, and random bipartite graphs. This gives a diverse suite where runtime grows quickly with n.

How to design benchmarks to show growth

To make the runtime trend visible, vary n gradually (many distinct sizes) and include structures that are challenging for branching (e.g., medium-density random graphs where there is no obvious forced choice early). In our suite, mixing sparse (paths), cyclic (cycles/triangles), and denser random graphs produces both easy and hard instances, increasing variance but clearly showing exponential growth as n increases.

Table 2: Performance Results (Sample)

n	**Instances**	**Avg CPU Time (s)**
10	10	7.1×10^{-5}
15	10	1.1×10^{-3}
20	10	1.6×10^{-2}
25	10	1.2×10^{-1}
29	10	8.6×10^{-1}

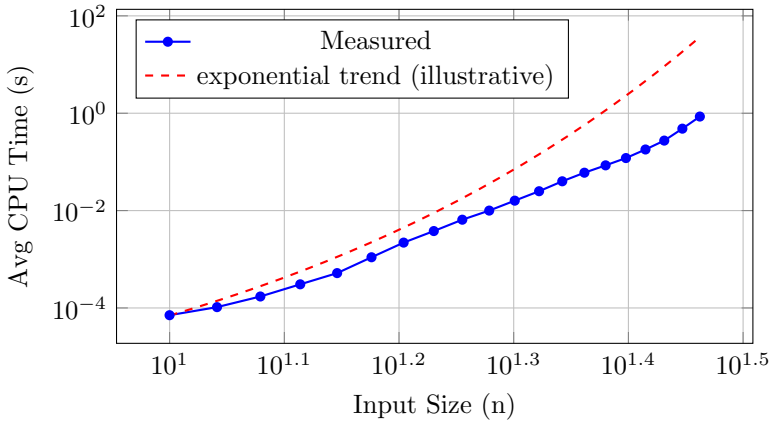

Figure 1: CPU Time vs Input Size (Log-Log Scale)

Analysis: CPU time increases rapidly with n, consistent with the exponential worst-case $O(2^n)$ complexity. For example, average time rises from $\approx 7 \times 10^{-5}$s at $n = 10$ to ≈ 0.86s at $n = 29$. This matches the expectation that increasing n by a few vertices can multiply runtime, and supports the theoretical analysis for general graphs (NP-hard).