Dicretization Techniques and State Partitioning Strategies in Contemporary Dynamic Programming

Azri Arzaq Pohan - 13524139
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
azri.pohan@gmail.com , 13524139@std.stei.itb.ac.id

***Abstract*— This paper explores two advanced techniques that alleviate this issue: discretization and state partitioning. Discretization compresses large or continuous domains into compact, tractable representations, enabling efficient indexing and memory use. State partitioning, on the other hand, exploits the structure of the solution space to reduce redundant computations through mathematical properties like convexity or monotonicity. Through theoretical exposition and two practical competitive programming case studies group segmentation minimization and source inference in trees. This paper demonstrates how these strategies significantly enhance the performance and scalability of DP algorithms.**

Keywords—dynamic programming; optimization; discretization; state partitioning; coordinate compression; divide and conquer DP; tree rerooting; computational complexity

I. Introduction

In the domain of competitive programming, temporal complexity assumes a critical role, often serving as the decisive criterion between an "Accepted" solution and a "Time Limit Exceeded" outcome. The strict limitations imposed by limited computing resources, particularly in terms of memory allocation and execution time, necessitate a profound understanding of algorithm efficiency that extends beyond the concept of superiority alone and has evolved into an essential requirement for effective computing practices. Such comprehension is very critical for optimizing performance and ensuring computational viability in complex problem-solving scenarios. Within this paradigm, dynamic programming emerges as one of the most widely adopted and theoretically grounded methodologies for addressing complex optimization problems, offering a systematic framework for breaking down intricate problems into overlapping subproblems while leveraging memoization or tabulation to ensure polynomial-time solutions where naive approaches would succumb to exponential complexity.

As problem complexity escalates, the implementation of techniques such as problem discretization and state partitioning becomes crucial for developing efficient solutions. Discretization serves to transform substantial or continuous value domains into more manageable discrete representations while preserving the core characteristics of the original problem. Conversely, state partitioning is focused on probing the specific structure of the solution space, thereby reducing computational complexity.

A solid foundation in discrete mathematics is essential for the practical application of optimization strategies. Basic concepts such as set theory, functions, relations, and discrete structures play a crucial role in the process of state construction, compression, and processing in dynamic programming.

II. Theoretical Basis

A. Set Theory

A set is a collection of objects that are unordered and distinct from one another but whose values can still be defined and undefined about one another. Objects within a set can also be referred to as elements. Sets are usually denoted in curly braces, followed by their elements. One of the examples is S = {1, 2, 3, 4, 5, 6}, which represents a set with six elements, namely 1, 2, 3, 4, 5, and 6.

Within a set, there are several basic operations that can be used to form a new set.

1. Intersection

The intersection of two sets A and B, denoted as $A \cap B$, produces a set that containing elements that in both sets

Formally:

$$A \cap B = \{x \mid x \in A \; and \; x \in B\}.$$

2. Union

The union of two sets A and B, denoted as $A \cup B$, produces a set that containing all elements that are in either A, or B, or both.

Formally:

$$A \cup B = \{x \mid x \in A \; or \; x \in B\}.$$

3. Complement

The complement of a set A, denoted A^c or $\overline{A}$, is the set of all elements in the universal set U that are not in A. The universal set must be defined for all complements to be meaningful.

Formally:

$$\overline{A} = \{x \mid x \in U, x \notin A\}.$$

4. Difference

The difference between two sets A and B, denoted as $A - B$, is the set of elements that belong to A but not to B.

Formally:

$$A - B = \{x \mid x \in A \; and \; x \notin B = A \cap \overline{B}\}.$$

5. Symmetric Difference

The symmetric difference of two sets A and B, denoted $A \oplus B$, is the set of elements that are in either of the sets but not in both.
Formally:

$$A \oplus B = (A \cup B) - (A \cap B) = (A - B) \cup (B - A)$$

B. Relations and Functions Theory

1. Properties of Relations

Relations that defined on a set can have properties such as reflexive, transitive, symmetric, antisymmetric.

- Reflexive

A relation R is reflexive if every element in A is related to itself $(a, a) \in R$ for all $a \in A$. Reflexive relations have matrices whose main diagonal elements are all equal to 1, or m_{ii} for $i = 1, 2, \ldots, n$,

Fig. 1. Reflexive Matrix.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/05-Relasi-dan-Fungsi-Bagian1-(2024).pdf]

A directed graph of reflexive relation is characterized by the presence of a ring at each node.

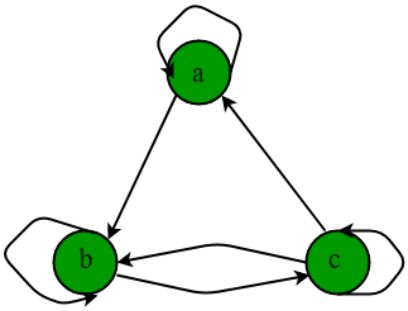

Fig. 2. Reflexive Directed Graph.
[Source: https://www.geeksforgeeks.org/relation-and-their-representations/]

- Transitive

A relation R is transitive if whenever $(a, b) \in R$ and $(b, c) \in R$ for all $a, b, c \in A$. In directed graphs, if there is a path from a to b and from b to c, there must be also be a direct edge from a to c. $x > y,\ y > z$ imply $x > z$.

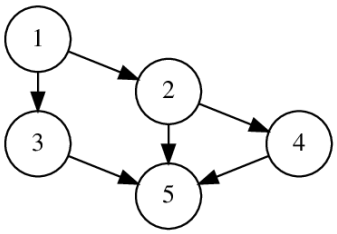

Fig. 3. Directed Graph.
[Source: https://codeforces.com/problemset/gymProblem/102411/J]

- Symmetric

A relation R is symmetric if $(a, b) \in R$ implies $(b, a) \in R$ for all $a, b \in A$. The matrix of a symmetric relation is symmetric across its diagonal.

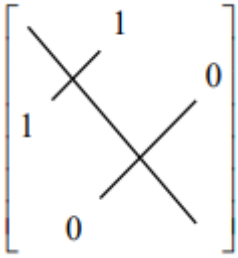

Fig. 4. Symmetric Relation Matrix.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/05-Relasi-dan-Fungsi-Bagian1-(2024).pdf]

In directed graphs, every edge must have a corresponding reverse edge.

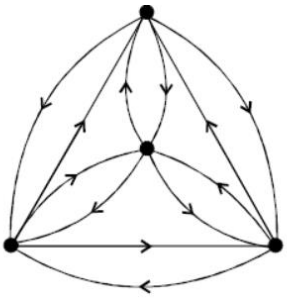

Fig. 5. Complete Symmetric Diagraph of Four Vertices.
[Source: https://skedbooks.com/books/graph-theory/types-of-digraphs/]

- Antisymmetric

A relation R is antisymmetric if $(a, b) \in R$ and $(b, a) \in R$ that imply $a = b$ for all $a, b \in A$. In matrix terms, if $m_{ij} = 1$ for $i \neq j$, then m_{ji} must be 0.

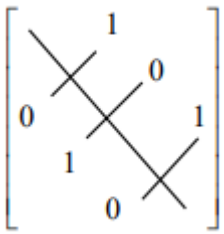

Fig. 6. Antisymmetric Matrix.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/05-Relasi-dan-Fungsi-Bagian1-(2024).pdf]

Graphically, no two distinct vertices can have edges in both directions. The divides relation is antisymmetric because a divides b and b divides a only if $a = b$.

2. Relation Composition

Let R be a binary relation from set A to set B, and S a binary relation from B to set C. The composition of relations denoted as $S \circ R$, is a relation from A to C, and formally defined as:

$$S \circ R = \{(a,c) \mid \exists b \in B \; such \; that (a,b) \in R \; and (b,c) \in S\}$$

If R_1 and R_2 relations are represented by M_{R1} and M_{R2}, then the matrix that representing the composition of two relations is

$$M_{R2 \circ R1} = M_{R1} \cdot \mathrm{M_{R2}}$$

where the operator "." is the same as in ordinary matrix multiplication, but with the multiplication sign replaced by "∧" and the addition sign replaced by "∨".

Suppose the R_1 and R_2 relations on set A is expressed by the matrix

$$R_1 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ dan } R_2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Fig. 7. R_1 and R_2 Matrices.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/06-Relasi-dan-Fungsi-Bagian2-(2024).pdf]

Then the matrix that representing $R_2 \circ R_1$ is

$$M_{R2 \circ R1} = M_{R1} \cdot M_{R2}$$

$$= \begin{bmatrix} (1 \wedge 0) \vee (0 \wedge 0) \vee (1 \wedge 1) & (1 \wedge 1) \vee (0 \wedge 0) \vee (1 \wedge 0) & (1 \wedge 0) \\ (1 \wedge 0) \vee (1 \wedge 0) \vee (0 \wedge 1) & (1 \wedge 1) \vee (1 \wedge 0) \vee (0 \wedge 0) & (1 \wedge 0) \\ (0 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 1) & (0 \wedge 1) \vee (0 \wedge 0) \vee (0 \wedge 0) & (0 \wedge 0) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Fig. 8. M_{R1} and M_{R2} Multiplication.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/06-Relasi-dan-Fungsi-Bagian2-(2024).pdf]

Furthermore, the composition of a relation with itself n times is denoted as R^n and defined recursively

$$R^n = R \circ R \circ \ldots \circ R \text{ with } R^{n+1} = R^n \circ R$$

and

$$M_{R^n} = M_R^{[n]}$$

because

$$R^{n+1} = R^n \circ R$$

then

$$M_{R^{n+1}} = M_R \cdot \mathrm{M_R^{[n]}}$$

3. Recursive Function

A function is called a recursive function if its definition refers to itself and consists of two fundamental components, such as the base case and the recurrence step.

- Base Case

The part that contains the initial value that does not refer to itself. This part also simultaneously stops the recursive definition.

- Recurrence

These sections define function arguments in their own terms. Whenever a function refers to itself, the argument of the function should be closer to the initial value (base).

Below are examples of recursive functions.

- Factorial Function

$$n! = \begin{cases} 1 & , n = 0 \\ n \times (n-1)! & , n > 0 \end{cases}$$

Fig. 9. Factorial Function.
[Source:https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/07-Relasi-dan-Fungsi-Bagian3-(2024).pdf]

- Chebysev Function

$$T(n,x) = \begin{cases} 1 & , n = 0 \\ x & , n = 1 \\ 2xT(n-1,x) - T(n-2,x) & , n > 1 \end{cases}$$

Fig. 10. Chebysev Function.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/07-Relasi-dan-Fungsi-Bagian3-(2024).pdf]

- Fibonacci Function

$$f(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ f(n-1) + f(n-2) & , n > 1 \end{cases}$$

Fig. 11. Fibonacci Function.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/07-Relasi-dan-Fungsi-Bagian3-(2024).pdf]

4. Closure of a Relation

The closure of a relation refers to the minimal extension of a given relation so that it satisfies specific property such as reflexivity, symmetry, or transitivity.

There are three main types of closures:

- Reflexive Closure

The reflexive closure of R is obtained by adding the minimal number of pairs (a,a) for every $a \in A$ not already in R.

Formally:

$$R_{reflexive} = R \cup \{(a,a) \mid a \in A\}$$

- Symmetric Closure

The symmetric closure ensures that for every $(a,b) \in R$, the pair (b,a) is also included.

Formally:

$$R_{symmetric} = R \cup R^{-1}$$

where $R^{-1} = \{(b,a) \mid (a,b) \in R\}$.

- Transitive Closure

The transitive closure is formed by iteratively adding pairs (a, c) whenever $(a, b) \in R$ and $(b.c) \in R$, until no more such pairs can be added.

Formally:

$$R^* = R \cup R^2 \cup R^3 \cup \ldots \cup R^n$$

C. *Graph Theory*

A graph G is defined as an ordered pair $G = (V, E)$, where V is a non-empty set of vertices (or nodes) and E is a set of edges (or arcs) connecting pairs of vertices. E may be empty. Graphs can be classified based on their structural properties:

- Simple Graph

  A graph with no loops or multiple edges between the same pair of vertices.

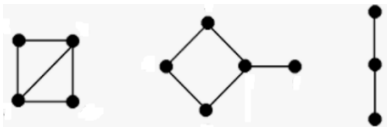

Fig. 12. Simple Graph.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf]

- Multigraph

  A graph that may have multiple edges between the same pair of vertices (no loops).

Fig. 13. Multigraph.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf]

- Pseudograph

  A graph that may contain both loops and multiple edges.

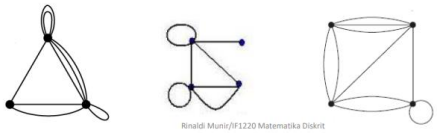

Fig. 14. Pseudograph.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf]

- Directed Graph

  A graph where edges have a direction.

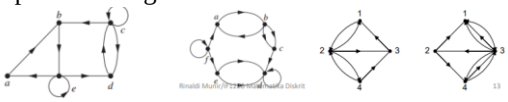

Fig. 15. Directed Graph.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf]

- Weighted Graph

  A graph where edges are assigned numerical values (weigths).

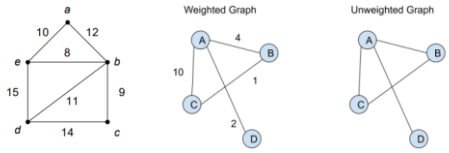

Fig. 16. Weigthed Graph.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf]

The connectivity of a graph refers to whether there exists a path between pairs of vertices in the graph. A graph is connected if there exists a path between every pair of distinct vertices, and a graph is disconnected if at least one pair of vertices lacks a connecting path.

- Strong Connectivity

  For every pair of vertices u and v, there is a directed path from u to v and from v to u.

- Weak Connectivity

  The graph is not strongly connected, but its underlying undirected graph is connected.

The structure of a graph is described through several fundamental concepts that define the relationships between its vertices and edges.

- Adjacency

  Two vertices are adjacent if they are connected directly by an edge.

- Incedency

  An edge is incident to the vertices it connects.

- Isolated Vertex

  A vertex with no incident edges.

- Degree

  The degree of a vertex is the number of edges incident to it.

- Path

  A path is a sequence of vertices where each consecutive pair is connected by an edge.

D. *Dynamic Programming*

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems, where the solution to the original problem can be viewed as a sequence of interrelated decisions. The term "dynamic" reflects the use of tables to store and build solutions incrementally, rather than implying any connection to programming languages.

1. Optimization Problems

   DP is primarily used for optimization problems, such as maximization or minimization. Unlike greedy algorithms, which make a single sequence of decisions, dynamic programming (DP) evaluates multiple decision sequences to ensure optimality.

2. Principle of Optimality

   A fundamental tenet of DP is that an optimal solution to the overall problem contains optimal solutions to its subproblems. This allows the problem to be solved stage by stage, leveraging results from previous stages without revisiting earlier decisions.

3. Approaches

- Top-Down DP (Memoization)

  In this method, the computation proceeds sequentially from

the first stage to the final stage, denoted as stage 1 to stage n. At each stage, a decision variable x_k is selected, and the optimal cost for reaching the current state is derived based on previous stages. This approach is suitable when the initial state is known and decisions are to be made progressively. It aligns with scenarios where the problem is naturally defined from beginning to end.

- Bottom-Up DP (Tabulation)
  Conversely, the backward method begins from the final stage and works in reverse towards the first stage. The decision variable sequence starts from x_n down to x_1. This approach is ideal when the final state or goal is clearly defined, and want to trace back the optimal sequence of decisions to reach it. The approach also benefits from simpler reconstruction of the optimal path.

E. *Coordinate Compression*

Coordinate compression is a technique used to convert large and sparse coordinate values or numerical data into smaller and more compact ranges of values. The goal of this technique is to reduce the size of the state space in dynamic programming, especially when the absolute values of the coordinates are not important. However, only their relative positions or orders are relevant.

Consider a problem setting that involves geometric points or intervals where coordinate values can span a vast range, such as from 1 to 10^9. Directly indexing a DP array would only lead to prohibitive memory consumption or excessively long computation times (TLE). To address this, coordinate compression is applied as a preprocessing step that remaps the original coordinate values into a smaller, contiguous range while preserving their relative order.

This transformation enables the use of efficient data structures such as segment trees, binary indexed trees, or simple arrays without requiring prohibitively large memory allocation. By mapping the original values to compressed indices ranging from 0 to $n-1$.

The steps that involved are as follows:

- Collect all numeric values that need to be compressed.
- Sort the collected values in ascending order to define their relative ranks.
- Eliminate repeated values to obtain a strictly increasing sequence of unique entries.
- Transform all values in the original dataset using the mapping to obtain compressed coordinates.

F. *Divide and Conquer*

This approach involves three main steps: divide, conquer, and combine. The divide step breaks a problem into smaller subproblems of similar type, the conquer step solves these subproblems either directly or recursively, and the combine step merges the solutions to form the final answer. The method is naturally expressed through recursive schemes and is particularly effective for problems where the input can be partitioned into smaller, manageable instances, such as arrays, matrices, exponents, or polynomials.

The algorithmic structure follows a recursive pattern. If the problem size is below a threshold n_0, it's solved directly. Otherwise, it's divided into r subproblems, each recursively solved, and their solutions are combined. The time complexity is expressed as a recurrence relation:

$$T(n) = \begin{cases} g(n) & ,n \leq \mathrm{n}_0 \\ T(n_1) + T(n_2) \ldots + T(n_r) + f(n) & ,n > n_0 \end{cases}$$

Fig. 17. Divide and Conquer Time Complexity.
[Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/07-Algoritma-Divide-and-Conquer-(2025)-Bagian1.pdf]

Where $g(n)$ is the time to solve a base case, and $f(n)$ is the time to combine solutions. A common case is dividing the problem into two equal parts, yielding $T(n) = 2T(n/2) + f(n)$.

G. *Depth-First Search (DFS)*

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. The algorithm starts at a selected node (root) and explores each adjacent node recursively, marking nodes as visited to avoid cycles. The traversal steps will continues until all reachable nodes are visited.

III. Study Case 1

A. *Problem Definition*

One of the problem discussed in this paper involves the optimal division of individuals into a fixed number of sequential groups, such that the unfamiliarity between individuals within each group is minimized.

Formally, let there be n individuals labeled $p_1, p_2, \ldots, p_n$, aligned in a queue. The goal is to allocate these individuals into exactly k non-empty groups, where each group consists of a contiguous segment of people from the front of the remaining queue. Let $q_1, q_2, \ldots, q_k$ denote the number of people in each group, then the following must hold:

$$q_i > 0 \; forall \; i, \sum_{i=1}^{\mathrm{k}} q_i = n$$

Each unordered pair of individuals (p_i, p_j) is assigned an unfamiliarity score $u_{ij} \in \mathrm{Z}_{\geq 0}$, with the following properties:

$$u_{ij} = u_{ji}, \qquad u_{ii} = 0, \qquad 0 \leq \mathrm{u_{ij}} \leq 9$$

The unfamiliarity of a group is defined as the sum of all u_{ij} for all unordered pairs i, j within that group. The objective is to find a partition into exactly k contiguous groups that minimizes the total unfamiliarity cost across all groups.

B. Solution

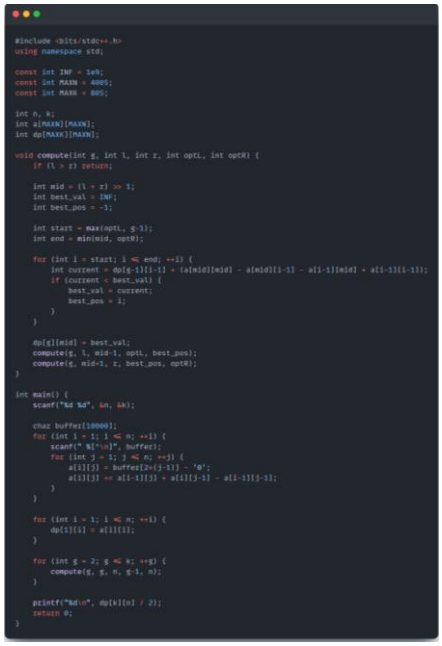

Fig. 18. Codeforces: 321E - Ciel and Gondolas Solution
[Source: Author]

In order to solve the sequential unfamiliarity minimization problem, we adopt a dynamic programming framework optimized through the divide-and-conquer paradigm. The idea is to represent the problem as a partitioning task over a queue of individuals, where each partition contributes a quadratic cost derived from pairwise unfamiliarities among its members. This can allows us to exploit structural properties of the cost function in order to improve computational efficiency.

The first part of the solution involves precomputing cumulative unfamiliarity scores using a two-dimensional prefix sum matrix. Let $a[i][j]$ denote the total unfamiliarity between the first i individuals in the queue. Using the standard inclusion-exclusion principle, this allows for constant-time queries of the total unfamiliarity cost within any rectangular submatrix of the input.

We can define a dynamic programming state $dp[g][m]$, which represents the minimal total unfamiliarity achievable by partitioning the first m individuals into g contiguous segments.

$$dp[g][m] = min_{i=g-1}^{m}\big(dp[g-1][i-1] + cost(i,m)\big)$$

where $cost(i,m)$ denotes the total unfamiliarity score for assigning individuals i through m to a group.

By applying Divide and Conquer Optimization will reduces the overall time complexitity of the dynamic programming from $O(kn^2)$ to $O(kn\,log\,n)$. Optimized recurrence can also be implemented using a recursive function $compute(g,l,r,optL,optR)$.

After computing the dynamic programming table, the final result is obtained from $dp[k][n]$, that represents the minimum possible unfamiliarity when dividing all n individuals into k groups. Since each pairwise unfamiliarity u_{ij} is counted twice, the final result is divided by two before being the output.

Problem	Lang	Verdict	Time	Memory
E - Ciel and Gondolas	C++17 (GCC 7-32)	Accepted	966 ms	75500 KB

Fig. 19. Submission result for Codeforces: 321E - Ciel and Gondolas
[Source: Author]

IV. Study Case 2

A. Problem Definition

The problem addressed in this section concerns the identification of possible source locations of a spreading anomaly (referred to as the Book of Evil) within a network of interconnected settlements. Formally, the area is modeled as a tree an undirected, connected, and acyclic graph with n nodes and $n-1$ bidirectional edges (paths).

Each edge represents a direct connection between two settlements and has a uniform traversal cost of 1. Among these n settlements, m settlements are known to be affected by the Book of Evil. The book if placed at certain settlement, can exert an influence with a fixed radius d.

The objective is to determine the number of settlements where the Book of Evil may be located, under the condition that if it were placed in such a settlement, all m affected settlements would fall within a distance of at most d from it.

B. Solution

Fig. 20. Codeforces: 337D – Book of Evil Solution
[Source: Author]

The approach leverages a bottom-up traversal to collect information from subtrees and a top-down traversal to propagate external information from parent and sibling subtrees.

Let the given tree be denoted by $T = (V, E)$, with $|V| = n$ nodes and $|E| = n - 1$ edges. A subset $A \subseteq V$, where $|A| = m$, contains all settlements reported to be affected by the Book of Evil.

The objective is to determine for how many nodes $u \in V$ it holds that the distance from u to every node in A is less than or equal to the damage threshold d. Define a function $D(u) \leq d$. Computing $D(u)$ for all nodes $u \in V$ in naïve manner would involve performing breadth-first search (BFS) or depth-first search (DFS) from each node, resulting in a time complexity of $O(nm)$ in the worst case, which is computationally infeasible for large n and m.

The first phase involves performing a bottom-up DFS traversal starting from an arbitrary root (e.g., node 1). For each node u:

- $f[u][0]$: the maximum distance from u to any affected node in its own subtree.
- $f[u][1]$: the second maximum distance, used to handle rerooting transitions correctly.

If a node u in the set A, it is better to initialize $f[u][0] = 0$ And $f[u][1] = -\infty$. Otherwise, these values are inherited from its child nodes during traversal and incremented by one.

The second phase involves a top-down traversal that propagate the maximum distance from affected nodes outside the current node's subtree. Let $up[u]$ denote the maximum distance from u to any node in A not contained in its own subtree. This value is computed recursively using the upward distance of its parent and the appropriate sibling subtree depths, ensuring that each node receives the most relevant information from the rest of the tree.

After both passes are complete, each node $u \in V$ has a complete view of its distance to the furthest affected node.

$$D(u) = max(f[u][0], up[u])$$

If $D(u) \leq d$, then u is included in the answer. This check is performed in constant time per node and since both traversals are linear in the number of nodes, the total time complexity remains $O(n)$.

Problem	Lang	Verdict	Time	Memory
D - Book of Evil	C++17 (GCC 7-32)	Accepted	248 ms	3200 KB

Fig. 21. Submission Result for Codeforces: 337D – Book of Evil
[Source: Author]

V. Conclusion

In this paper, we have explored advanced optimization techniques in dynamic programming through the lens of discretization and state partitioning. These strategies have proven indispensable in overcoming challenges posed by large state spaces and high time complexity. Discretization, through techniques such as coordinate compression, enables the transformation of large numeric domains into compact, indexable forms—making them amenable to efficient data structures. Meanwhile, state partitioning strategies, exemplified by divide and conquer DP and rerooting on trees, exploit the mathematical structure of problems to significantly reduce redundant computations.

Theoretical discussions were supported by two real-world competitive programming cases. The first showcased how divide and conquer optimization reduces DP complexity from $O(k \cdot n^2)$ to $O(k \cdot n \log n)$ while the second demonstrated how tree rerooting allows linear-time resolution of center-point queries on affected nodes.

VI. Appendix

The following is the source code used to solve several problems from Codeforces that used as study cases in this paper. https://github.com/AzriVz/makalah-matematika-diskrit

Acknowledgment

The author would like to express his deepest gratitude to Allah SWT for the continuous blessings, guidance, and strength that made the completion of this paper possible. The author is also sincerely thankful to Dr. Ir. Rinaldi Munir, M.T. for his invaluable guidance, insightful teaching, and encouragement throughout the course.

Furthermore, heartfelt appreciation is extended to the author's parents, siblings, and close friends for their unwavering moral and emotional support during his academic journey at Institut Teknologi Bandung.

References

[1] Munir, R. "*IF1220 Matematika Diskrit - Semester II Tahun 2024/2025*", *Teknik Informatika Institut Teknologi Bandung*. Available at: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025-2/matdis24-25-2.htm#SlideKuliah (Accessed: 19 June 2025).

[2] Munir, R. "*IF2211 Strategi Algoritma - Semester II Tahun 2024/2025*", *Teknik Informatika Institut Teknologi Bandung*. Available at: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm#SlideKuliah (Accessed: 20 June 2025).

[3] *Competitive Programmer's Handbook*. Available at: https://cses.fi/book/book.pdf (Accessed: 20 June 2025).

[4] *Custom comparators and coordinate compression · USACO Guide*. Available at: https://usaco.guide/silver/sorting-custom?lang=cpp (Accessed: 20 June 2025).

[5] *W3schools.com* "*Dynamic Programming*". Available at: https://www.w3schools.com/dsa/dsa_ref_dynamic_programming.php (Accessed: 20 June 2025).

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Azri Arzaq Pohan, 13524139