TEC | Tecnológico de Costa Rica

January 18, 2024

Traveling Salesman Problem

Computing Engineering

Algorithm Analysis

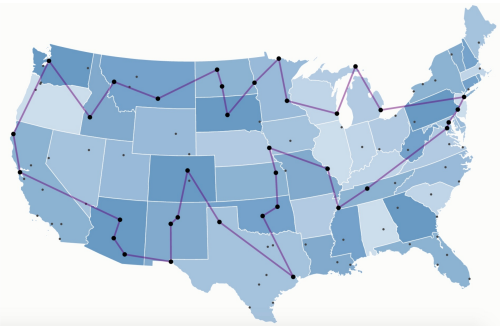
Author:
Danilo Duque Gómez

Professor:
Eddy Ramírez Jiménez

1 Executive Summary

The Traveling Salesman Problem (TSP) represents a classical optimization challenge with widespread implications across various industries. This executive summary delineates an expansive project focused on addressing the TSP through three distinctive methodologies: backtracking, dynamic programming, and genetic algorithms. The amalgamation of these approaches seeks not only to confront the TSP directly but also to yield insights into the comparative efficacy of diverse optimization strategies.

Recognizing the inherent complexity in finding the minimum Hamiltonian cycle in a graph, deterministic approaches proved impractical within reasonable timeframes, even with a relatively small number of nodes. Consequently, artificial intelligence algorithms were deployed. While these algorithms do not guarantee optimal solutions, their proximity to optimality makes the obtained solutions practical for real-world applications.



Further exploration delves into the classification of this problem as NP-Hard and why the branch and bound approach on a complete graph with 26 nodes would take approximately 127,000 years. Alternative resolutions, including dynamic programming with an algorithmic complexity of $n^2\ 2^n$, will be explored. Additionally, the genetic algorithms solution, providing a response in polynomial time, will be examined, acknowledging its inability to guarantee optimality due to the inherent randomness of its implementation.

Subsequently, the outcomes of each resolution will be scrutinized, culminating in an evaluation of the validity of each approach. A meticulous algorithmic analysis for each case will be presented, affording a comprehensive understanding of the strengths and limitations inherent in the applied methodologies.

2 Introduction

The Traveling Salesman Problem (TSP) has been a fascinating challenge in the world of optimization and route planning for quite some time. It's a bit like a classic brain teaser that has intrigued researchers and professionals since it was first thought about in the last century. At its core, the TSP poses a crucial question: What's the most efficient route for a traveling salesman to take, ensuring they visit a given set of cities exactly once before heading back to the starting point?

The TSP has a long history, going back to the 1800s, but it gained formal recognition and popularity in the 1930s thanks to Karl Menger, a mathematician and economist. Even before that, in the 19th century, the Irish mathematician William Rowan Hamilton had a similar puzzle named the "Hamiltonian game." It involved finding a path that touches each point in a graph just once, laying the groundwork for problems like the TSP.

In more straightforward terms, the essence of the Traveling Salesman Problem (TSP) lies in determining the optimal route for a salesperson. This route should encompass visiting all designated cities precisely once and concluding the journey by returning to the initial point of departure. The motivation behind this meticulous planning stems from the salesperson's sincere preference for returning home to sleep in the comfort of their own bed.

Envision this salesperson ardently crafting an efficient itinerary, strategically navigating through cities, and meticulously optimizing the allocation of time and resources throughout the entirety of their sales expedition.

In a more formal context, the Traveling Salesman Problem revolves around finding a Hamiltonian cycle that minimizes the cost of traversing edges in a weighted undirected graph. This enables the salesman to traverse all cities with the least possible waste of resources and time.

Let's assume we have a set of cities $C = \{1, 2, 3, \ldots, n\}$ where n is the number of cities.

Let d_{ij} represent the distance between city i and city j. The goal is to find a permutation π of the cities where $\pi = (\pi_1, \pi_2, ..., \pi_n)$ such that the following objective function is minimized:

$$\sum_{i=1}^{n-1} d_{\pi_i \pi_{i+1}} + d_{\pi_n \pi_1}$$

In simpler terms, the goal is to determine the optimal order for visiting cities, minimizing the sum of distances between consecutive cities and ultimately returning to the starting point.

This classification as NP, indicating the problem's resistance to efficient deterministic machine solutions within polynomial time, stems from the intricacy of identifying the minimum among numerous potential permutations within a sequence of nodes. The exponential growth in possibilities amplifies the computational formidability of the challenge, aligning the TSP with the NP-completeness paradigm and placing it within a class of problems lacking known polynomial-time solutions.

Furthermore, the evolution of computing power and algorithmic innovation has catalyzed interest in exact algorithms for TSP. Techniques such as branch and bound, integer linear programming, and dynamic programming find application in addressing smaller instances of the problem where an exhaustive search for the optimal solution is feasible.

TSP significance transcends theoretical challenges. Real-world applications span logistics, transportation planning, manufacturing, and circuit design. The efficient resolution of TSP instances holds practical implications for diverse industries, influencing decision-making processes and optimizing resource utilization.

In the following sections, key algorithmic approaches for addressing the Traveling Salesman Problem are explored. The examination illuminates their

strengths, limitations, and practical applications, aiming to deepen the understanding of the TSP landscape and contribute to the ongoing discourse regarding its resolution.

3 Theoretical Framework

During the preliminary study and execution of this project, various tools were employed to comprehend and implement specific practical effects. Some of the tools used include:

3.1 Backtracking

Backtracking is a systematic algorithmic technique employed to explore all potential solutions to a computational problem. It is particularly useful in solving problems that exhibit a combinatorial nature, where the goal is to search through a vast solution space for an optimal or feasible solution.

The backtracking process involves making a series of choices and backtracking when a chosen path leads to an infeasible solution or exhausts the search space. It operates on the principle of trial and error, systematically exploring different possibilities until a valid solution is found or all possibilities have been exhausted.

3.2 Dynamic Programming

Dynamic Programming is a systematic computational optimization technique used to solve problems by breaking them down into smaller, overlapping subproblems and efficiently solving each subproblem only once. This technique is particularly powerful when addressing problems with optimal substructure and overlapping subproblems.

3.3 Key Components

3.3.1 Principle of Optimality

This principle asserts that an optimal solution to a larger problem can be constructed from optimal solutions to its subproblems. In the context of Dynamic Programming, it is encapsulated in the Bellman equation:

$$V(i) = \max_a [R(i, a) + \gamma \cdot V(j)]$$

Where $V(i)$ represents the optimal value for state i, $R(i, a)$ is the immediate reward for taking action a in state i, γ is the discount factor, and $V(j)$ represents the optimal value of the state j to which the system transitions deterministically from the state i by taking action a.

3.3.2 Memoization

Dynamic Programming employs memoization to store and reuse solutions to previously solved subproblems. This involves maintaining a table or cache to store the results of subproblems, preventing redundant computations. In the context of the Bellman equation, memoization ensures that once the optimal value for a particular state is computed, it is stored for future reference, eliminating the need to recalculate it.

Dynamic Programming is widely applicable in various domains, including optimization problems in operations research, algorithmic challenges in computer science, and more. Its ability to efficiently solve complex problems by breaking them down into manageable subproblems and reusing solutions makes it a fundamental and powerful optimization technique.

3.4 Genetic Algorithms

Genetic Algorithms (GAs) stand at the intersection of optimization and computational intelligence, drawing inspiration from the intricate mechanisms of

biological evolution. Conceived by John Holland in the 1960s, GAs present a powerful paradigm for addressing complex optimization and search challenges across various domains.

At their essence, GAs navigate solution spaces by representing potential solutions as chromosomes, mimicking the genetic makeup found in living organisms. Each chromosome comprises genes, which, in the context of optimization, represent variables or components of a solution. The algorithm operates on a population of individuals, where each individual corresponds to a unique chromosome embodying a candidate solution.

A cornerstone in the GA framework is the fitness function, a metric that evaluates the efficacy of each individual in solving the given problem. It serves as the guiding force, assigning a quantitative measure of how well a particular solution aligns with the optimization objective. The driving principle is akin to natural selection, where individuals exhibiting higher fitness are more likely to be selected for reproduction.

The selection process establishes a mating pool, shaping the genetic diversity of the next generation. Crossover and mutation mechanisms then come into play, introducing variability and exploration. Crossover entails the exchange of genetic material between two parent chromosomes, creating offspring with a blend of characteristics. Mutation introduces randomness by applying small, random changes to the genes of selected individuals, preventing the algorithm from converging prematurely.

The iterative nature of GAs involves the generation of successive populations, each evolving towards more optimal solutions. This cyclical process of selection, crossover, mutation, and replacement continues until a termination criterion, such as a maximum number of generations or a satisfactory fitness level, is met.

3.5 Graph Theory

Graph Theory is a mathematical discipline that focuses on the investigation and analysis of structures used to model relationships and connections among entities. In this context, these structures are denoted as graphs, represented

by ordered pairs $G = (V, E)$, where V signifies the set of vertices and E denotes the set of edges. Vertices are individual elements within the graph, interconnected by edges, symbolized by e_{ij}, indicating relationships between vertices v_i and v_j. The field involves the exploration of properties, patterns, and characteristics inherent in these graphical representations. Furthermore, it encompasses the development of mathematical methods, theorems, and algorithms to comprehend and address problems and scenarios characterized by interconnected relationships within these graph structures.

3.6 Data Structures

In computer science, Data Structures provide a systematic approach to organizing and managing data, serving as the foundation for algorithmic design. These structures, categorized into primitives (integers, characters, etc.) and composites (arrays, linked lists, trees, etc.), facilitate efficient data storage and retrieval.

3.6.1 Definition

A formal definition characterizes a data structure as a systematic policy governing the management and organization of data. It serves as a framework for efficient storage, retrieval, and manipulation, optimizing computational processes within algorithms.

In computer science, the choice of a data structure profoundly impacts algorithmic efficiency, emphasizing the dynamic and essential nature of this field.

3.7 Java

Java, a versatile and object-oriented programming language, was developed by James Gosling and Mike Sheridan at Sun Microsystems in the mid-1990s. Known for its platform independence, Java allows developers to write code once and run it on any device with a Java Virtual Machine (JVM). Over the years, Java has evolved with features such as security measures, portability

through bytecode, and a robust ecosystem. Its adoption spans diverse domains, from enterprise applications to web and mobile development, making it a cornerstone in the programming landscape.

4 Solution Description

This section comprises three parts, each presenting one of the implemented solutions designed to address the Traveling Salesman Problem (TSP). These solutions showcase distinct approaches to optimizing the itinerary of the traveling salesman, offering a comprehensive view of their respective strategies and efficiencies.

4.1 Branch And Bound

Commencing with the explanation of the most basic solution, backtracking is employed. In the context of the Traveling Salesman Problem (TSP), the objective revolves around identifying the permutation π that minimizes the following objective function:

$$\sum_{i=1}^{n-1} d_{\pi_i \pi_{i+1}} + d_{\pi_n \pi_1}$$

The backtracking approach, in a nutshell, systematically explores all permutations, marking visited nodes along the way. When encountering a node without further viable connections, it discards that particular path, thus optimizing the search for the permutation with the minimum cost.

For a clearer comprehension of the algorithm's process, let's delve into the recursive function of Depth First Search (DFS).

$$\text{DFS}(G, i) = \begin{cases} \emptyset & \text{if } i \in S \\ \{i\} \cup \ \text{DFS}(G, j) \quad \forall j \in V \mid iRj & \text{otherwise} \end{cases}$$

The `DFS` function, denoted as $\texttt{DFS}(G, i)$, is a version of the Depth-First Search algorithm specifically designed for constructing a spanning tree within a graph G, starting from an initial node i. In brief:

- **If i is in the spanning tree:** If the node i is already part of the spanning tree, the function returns an empty set $\emptyset$, indicating that this node has been visited.
- **Otherwise:** In this case, the function adds the current node i to the spanning tree and recursively explores each adjacent node j (where $j \in V \mid iRj$). The results of these recursive calls are combined with the spanning tree.

Now, because in the TSP, the objective is to find the minimum Hamiltonian cycle, let's modify the general DFS algorithm to tailor it specifically for this purpose. In standard DFS, we explore the entire graph, marking visited nodes and backtracking when necessary. However, in the context of the Traveling Salesman Problem, our focus shifts to discovering the optimal order of visiting cities to minimize the overall travel cost.

$$\text{TSP}(G, i, S) = \begin{cases} \text{Cost}(i, v_0) & \text{if } S = V \\ \min\left[\text{Cost}(i, j) + \text{TSP}(G, j, S \cup \{i\})\right], \ \forall j \in V \mid j \notin S \wedge iRj \end{cases}$$

- If the set of visited nodes S equals the complete set of nodes V, the function returns the cost of returning from node i to the initial node v_0, denoted as $\text{Cost}(i, v_0)$. This serves as the base case when all nodes have been visited.

- Otherwise, the function evaluates the minimum cost among all possible transitions from the current node i to an unvisited node j ($j \notin S \wedge iRj$). For

each chosen node j, it calculates the cost of traveling from i to j ($\text{Cost}(i, j)$) and adds it to the result of recursively calling the function with the new node j added to the set of visited nodes ($S \cup \{i\}$). The final result is the minimum of these values.

In summary, the function aims to find the most efficient route visiting all nodes exactly once and returning to the starting point, minimizing the overall tour cost. The recursion explores all possible combinations of nodes, assessing the cost of each route to determine the optimal solution to the Traveling Salesman Problem (TSP).

While the development of a branching solution may seem straightforward, its true challenge lies in the daunting algorithmic complexity it brings. This approach, with its $O(n!)$ algorithmic complexity, underscores the exponential surge in computations as the factorial of the number of cities (n) grows.

This complexity places constraints on the practicality of the solution beyond 13 cities, where the factorial growth becomes overwhelmingly dominant. To provide perspective, envision a complete graph with 30 nodes, assuming a rate of 10^8 calculations per second. In such a scenario, this program would require an astronomical 8.4×10^{16} years to reach a solution. To put this in context, the estimated age of the universe is approximately 13.8×10^9 years, highlighting the impracticality of this approach on a universal timescale.

However, it's crucial to recognize that this approach reveals its practical utility when applied to a more modest input set. In scenarios with a relatively small number of cities, the computational efficiency becomes apparent, allowing for more manageable factorial growth and computation times. Moreover, it's noteworthy that this method is often considered the most straightforward and easy to implement. Despite its limitations, especially with larger instances of the Traveling Salesman Problem, this approach proves useful and efficient within certain constrained contexts.

4.2 Dynamic Programming

The resolution of the Traveling Salesman Problem (TSP) through dynamic programming involves breaking down the problem into smaller subproblems and constructing optimal solutions using information obtained from these subproblems.

Notably, the mathematical function representing the dynamic programming solution is essentially analogous to its backtracking counterpart, with the key distinction being the dynamic programming approach's capability to memoize subproblem solutions, thereby enhancing efficiency through the avoidance of redundant computations.

Formally, let's consider the TSP problem as TSP(G, i, S), where:
- G is a weighted graph representing cities and the distances between them.
- i is the current node from which we are considering the next city to visit.
- S is the set of nodes visited so far.

The optimal solution is derived by assessing smaller subproblems, characterized by the tuple (i, S). Here, i designates the current node, while S encapsulates the set of nodes visited up to that point. The construction of the overall solution for the main problem involves amalgamating optimal solutions from these subproblems.

For instance, considering the current node as i and the current visited set as S_i, we can obtain TSP(G, i, S_i) by retrieving the memoized optimal solution. This is achieved through the application of the following formula:

$$TSP(G, i, S_i) = \min[Cost(i, j) + TSP(G, j, S_i \cup \{j\})] \quad \forall j \in V \mid iRj \wedge j \notin S_i$$

In executing this process, the minimum cost for TSP(G, i, S_i) is determined. Subsequently, the solution to this subproblem is memoized, with the key being the tuple (i, S_i).

To determine the algorithmic complexity of this solution, an exploration into the potential states in a graph with n nodes is conducted. The concept of "states" refers to distinct combinations of nodes, as the algorithm navigates through various possibilities. In quantifying these states, the application of

the formula for the sum of binomial coefficients is crucial.

The sum $\sum_{k=0}^{n} \binom{n}{k}$ denotes the summation of binomial coefficients, where $\binom{n}{k}$ signifies the ways to select k nodes from a set of n. The result of this sum is 2^n, signifying 2^n potential states with n nodes. The subproblems are characterized by a tuple representing the current node and the set of visited vertices. As a consequence, the total count of possible states the algorithm might encounter is $n \cdot 2^n$. Additionally, the cost of each recursive call is n because, to determine the optimal next move, each node needs evaluation. Consequently, the effective time complexity of the Dynamic Approach is $O(n^2 \cdot 2^n)$.

Another crucial aspect to consider in this implementation is its spatial complexity. The process of saving or memoizing subproblems contributes to this aspect. As demonstrated in the earlier analysis, the program encounters a total of $n \cdot 2^n$ possible states. Consequently, the worst-case spatial complexity is $n \cdot 2^n$.

In contrast to the previous analysis of the branch and bound approach, this algorithm, when applied to a complete graph of 30 vertices and assuming a rate of 10^8 operations per second, is anticipated to converge to the optimal solution in approximately 3 hours. However, it is essential to note that this efficiency comes at a cost, requiring a substantial amount of memory—specifically, 16GB. This represents a delicate balance between time efficiency and resource consumption, a key consideration in choosing the appropriate algorithmic approach.

In conclusion, the dynamic programming approach extends the horizons further; however, this expansion comes at the cost of increased resource demands. While the dynamic programming solution enables the exploration of a bit larger problem instances, it necessitates more extensive computational resources. It is important to note that even with these advancements, there may be limits to the size of input cases that can be effectively tested due to the escalating resource requirements.

4.3 Genetic Algorithm

Finally, the utilization of genetic algorithms for solving the Traveling Salesman Problem (TSP) represents a departure from deterministic approaches. While aiming to provide a solution in significantly less time, genetic algorithms introduce an element of non-determinism. Consequently, the results obtained through this approach may not guarantee optimality but strive to be close to the optimal solution. This trade-off between computational efficiency and optimality is a defining characteristic of genetic algorithms in TSP problem-solving.

The initial and crucial step in a genetic implementation involves defining the individual or chromosome utilized for solving the Traveling Salesman Problem (TSP).

Formally defining a chromosome denoted as C involves representing it as a permutation $\pi = (\pi_0, \pi_1, ..., \pi_n)$, of the set of vertices V. Assigning a fitness value to C, denoted as $F(C)$, is achieved by evaluating the expression:

$$F'(C) = Cost(\pi_n, \pi_0) + \sum_{i=0}^{n-1} Cost(\pi_i, \pi_{i+1})$$

In this context, the function $Cost(v_i, v_j)$ signifies the distance between vertices v_i and v_j. It's crucial to note that the $Cost$ function returns ∞ when there is no direct connection or edges between v_i and v_j. This fitness value encapsulates the total cost of the given permutation, considering both the distance from the last vertex back to the initial vertex and the sum of distances between consecutive vertices in the permutation.

The Traveling Salesman Problem (TSP) is cast as a minimization problem. Consequently, as the fitness function $F'(C)$ approaches zero, the chromosome C is considered more proficient in addressing the problem. The likelihood of its reproduction and the transmission of its genetic material increases correspondingly.

To achieve this, the probability of reproduction is inversely proportional to $F'(C)$. The actual fitness function for a chromosome X in a population P is expressed as:

$$F(X) = \frac{\sum_{\forall C \in P} F'(C)}{F'(X)}$$

For the crossover operation between two chromosomes, consider two permutations of the set V, $\pi = (\pi_0, \pi_1, ..., \pi_n)$ and $\delta = (\delta_0, \delta_1, ..., \delta_n)$. Randomly select two integers from the range $[0, n]$, 2 and 3 for example. Subsequently, extract all vertices in the range $[2, 3]$ from the first parent chromosome, and compose a new permutation, γ. Then, for each $\delta_x \in \delta$ such that $\delta_x \notin \gamma$, perform a set union operation between γ and δ_x.

By executing this process, subpaths from each parent are likely to be retained in the new chromosome. Consequently, there is a probability that the new chromosome in the next generation is fitter, or at the very least, preserves certain subpaths from one of the parents.

Finally, there exists a probability of mutation for a new chromosome. This consideration is crucial to prevent the algorithm from converging solely to local minima. If each new generation exclusively inherits genes from its parents, the algorithm may become susceptible to getting stuck in a local minimum, hindering its ability to approximate the optimal value. Therefore, to introduce variability and explore a broader solution space, a mutation chance of 5% is incorporated into the algorithm.

A mutation is represented as a swap operation between two nodes in the permutation of a chromosome C. Furthermore, this function incorporates a heuristic aimed at enhancing the permutation. The heuristic evaluates pairs of consecutive nodes, i and $i+1$, and j and $j+1$, and performs the following operation if it improves the cost of both pairs:

$$\forall i, j \in [0, n], \text{if } Cost(i, i+1) > Cost(j, j+1), \text{then swap}(i+1 \rightarrow j)$$

This heuristic systematically examines the cost of connections between adjacent nodes and considers replacing the connection $i, i+1$ with i, j if the

latter results in a cost improvement for both pairs.

The algorithm operates based on the following configuration. The size of each population or generation is set to match the size of the graph. For instance, if the graph comprises 25 nodes, each population consists of 25 chromosomes.

Consider a priority queue of chromosomes, where priority is assigned to those with lower fitness values. Introduce n new chromosomes generated by randomly crossing parents. Additionally, include the existing n chromosomes of the current generation in the queue. To form the new generation, select the n best chromosomes iteratively. Repeat this process until the best chromosome in each iteration changes less frequently than the cube of the graph's size.

By following this process, the algorithm is geared towards gradually approximating the optimal value with each iteration. This is achieved by preserving the best-performing chromosomes and disregarding any aberrations that might arise from combining genes of two well-adapted parents.

In summary, the genetic algorithm for solving the Traveling Salesman Problem (TSP) introduces non-deterministic elements to achieve faster solutions, albeit with a potential trade-off in optimality. The algorithm defines chromosomes, evaluates fitness based on a permutation's cost, and employs a crossover function for genetic recombination. A mutation mechanism, with a 5% probability, adds variability to prevent convergence to local minima.

The algorithm's heuristic systematically assesses connections between nodes, swapping when it improves costs for consecutive pairs. Operating with a population size matching the graph, the algorithm selects the best chromosomes iteratively, striving to approximate the optimal solution. This approach balances computational efficiency and solution quality by preserving superior genetic material and avoiding aberrations from parent combinations.

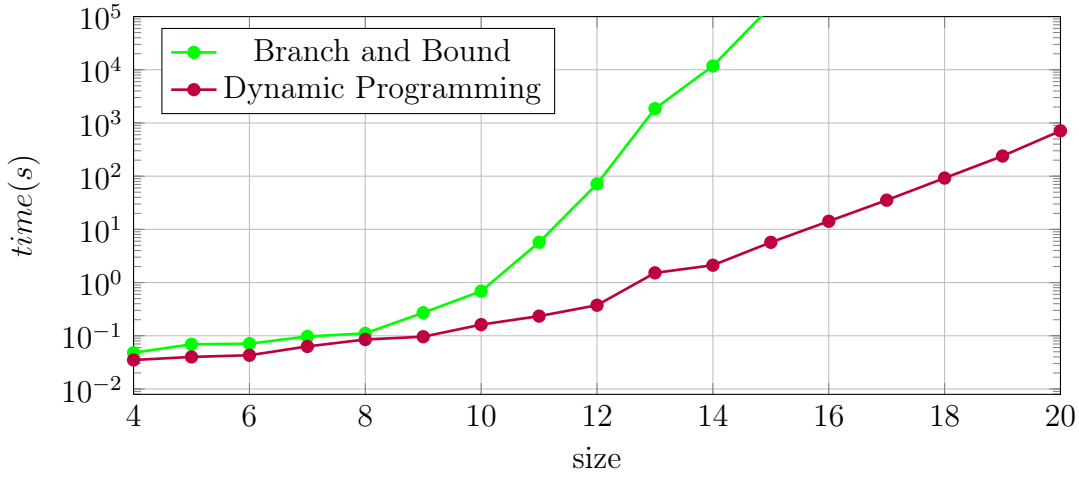
5 Test Findings

In this section, empirical results and findings obtained through testing the implemented algorithms for solving the Traveling Salesman Problem (TSP) are presented. Performance metrics, algorithmic behavior, and comparisons between different approaches are analyzed to provide valuable insights into the effectiveness and efficiency of each algorithm in addressing the TSP. These findings aid in the selection of the most suitable solution for specific scenarios.

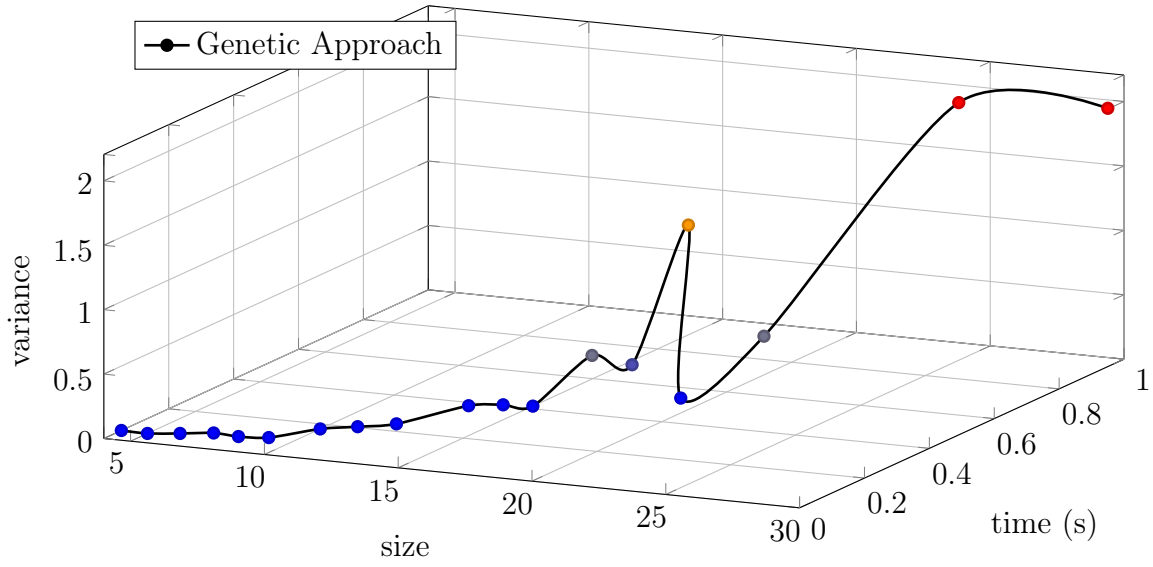
To rigorously evaluate and analyze each solution, a comprehensive testing strategy has been devised. Each implementation undergoes testing using various instances of the Traveling Salesman Problem (TSP), employing complete graphs (worst case) of varying sizes ranging from 4 to 25 nodes.

In the probabilistic implementation, the outcome will be the average of results obtained from multiple runs. Additionally, measurements like variance will be conducted for further analysis.

Initiating the evaluation of deterministic approaches, each solution undergoes testing with diverse test cases encompassing complete graphs of sizes ranging from 4 to 20. The resulting graph illustrates the input size on the x-axis and the corresponding algorithm runtime on the y-axis. The green curve signifies the branch and bound approach, whereas the purple one represents the dynamic programming (DP) approach.



In the process of generating the graphical representation for the probabilistic approach, another critical factor necessitates consideration—specifically, the variance observed between the generated results and the optimal solution to the Traveling Salesman Problem (TSP). For clarity, let the x-axis represent the size of the graph, the y-axis denote the algorithm's execution time to reach a local minimum, and the z-axis signify the average observed variance.



This graphical representation provides a visual understanding of the probabilistic approach's performance, highlighting both the computational time required to reach a local minimum and the associated variance in comparison to the optimal solution.

The analysis of the presented graphics reveals significant insights into the performance of deterministic and probabilistic solutions for the Traveling Salesman Problem (TSP). A formal mathematical analysis can be employed to elucidate the findings:

5.1 Deterministic Approaches

The runtime analysis of the deterministic approaches, represented by the branch and bound and dynamic programming algorithms, demonstrates an exponential growth in computational time as the size of the problem instance increases. This growth can be expressed mathematically as $T(n) = a \cdot b^n$, where a and b are constants. The observed steep increase in runtime aligns with the inherent complexity of the TSP, confirming theoretical expectations.

5.2 Probabilistic Approach

The probabilistic approach, implemented through the genetic algorithm, showcases noteworthy characteristics through the analysis of variance and solution proximity. For a TSP graph with less than 30 nodes, the maximum observed variance (V_n) is limited, satisfying $V_n \leq 2.1$. This limited variance, suggests that the solutions generated by the genetic algorithm exhibit consistency and proximity—a valuable trait signifying robustness and reliability. Moreover, the consistent attainment of the optimal solution for graphs with fewer than 16 nodes implies effective global minimum finding techniques.

In conclusion, the deterministic approach exhibits the expected exponential growth in computational time, while the probabilistic approach demonstrates robustness, consistency, and effective global minimum finding techniques for various problem instances.

6 Conclutions

In conclusion, the comprehensive analysis of deterministic and probabilistic solutions for the Traveling Salesman Problem (TSP) provides valuable insights into their respective performances. The study encompassed empirical evaluations, mathematical analyses, and graphical representations, shedding light on the strengths and limitations of each approach.

6.1 Deterministic Approaches

The deterministic solutions, namely the branch and bound and dynamic programming algorithms, exhibit the anticipated exponential growth in computational time as the size of the TSP graph increases. This aligns with the theoretical complexity of the TSP. Despite the inherent challenges in solving larger instances efficiently, deterministic approaches remain foundational for understanding algorithmic behavior.

6.2 Probabilistic Approach

The probabilistic approach, implemented through the genetic algorithm, emerges as a robust and consistent method for solving the TSP. The limitation variance, underscores the algorithm's ability to generate solutions that are closely clustered. This consistency is a crucial characteristic in practical applications, signifying the reliability of the genetic algorithm.

Furthermore, the observed effectiveness in consistently reaching the optimal solution for graphs with fewer than 16 nodes highlights the algorithm's proficiency in finding global minima. This is a crucial attribute, especially in real-world scenarios where optimal solutions are crucial.

6.3 Overall Insights

The project contributes not only to the understanding of algorithmic behaviors in the context of the TSP but also provides a practical basis for decision-making in algorithm selection. While deterministic approaches are computationally intensive for larger instances, the genetic algorithm proves to be a valuable alternative, particularly for larger sized TSP graphs.

In future work, further optimizations and fine-tuning of the genetic algorithm could be explored to enhance its performance on larger problem instances. Additionally, the insights gained from this study can inform the selection of appropriate algorithms for specific TSP scenarios, guiding practitioners in making informed choices based on the size and complexity of their

problem instances.

Overall, this project adds to the body of knowledge in algorithmic optimization and provides a foundation for continued exploration and refinement in the quest for effective solutions to the Traveling Salesman Problem.

7 Learnings

The experimentation highlighted the significance of artificial intelligence algorithms, particularly in real-world applications where scalability can be a critical factor. Given the absence of evidence supporting the conjecture that $P = NP$, the practicality of genetic approaches becomes evident. These algorithms demonstrate effectiveness in scenarios where the size of the problem instance may pose challenges for other methodologies. The findings underscore the relevance of exploring and leveraging genetic algorithms, emphasizing their potential as robust solutions in practical, large-scale applications.

8 References

[1] "Travelling Salesman Problem | 68 plays | Quizizz," Quizizz. [Online]. Available: `https://quizizz.com/admin/quiz/5ddf5dd1e4617d001bc08579/travelling-salesman-problem` (Accessed: Jan. 18, 2024).

[2] "Java Software | Oracle," Oracle.com, 2019. [Online]. Available: `https://www.oracle.com/java/`.

[3] M. Mitchell, "An Introduction to Genetic Algorithms." The MIT Press, 1998. [Online]. Available: `https://direct.mit.edu/books/book/4675/An-Introduction-to-Genetic-Algorithms` (Accessed: Jan. 18, 2024).

[4] "Bellman equation," Wikipedia, Apr. 25, 2021. [Online]. Available: `https://en.wikipedia.org/wiki/Bellman_equation`.