

Department of Computer Science and Engineering

## A PROJECT BASED LABORATORY REPORT ON

BRANCH AND BOUND METHOD AND

PARTICLE SWAM OPTIMIZATION

SUBMITTED BY:

|  |  |
| --- | --- |
| ID-NO | NAME |
| 2200031921 | YARRAMSETTI SIVA SANKARA VARA PRASAD |

UNDER THE ESTEEMED GUIDANCE OF

Dr. Vuda Sreenivasa Rao

Assistant Professor



KONERU LAKSHMAIAH EDUCATION FOUNDATION GREEN FIELDS, VADDESWARAM – 522502

Guntur Dist. Andra Pradesh. India.

Department of Computer Science and Engineering

Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the project-based laboratory report on

**BRANCH AND BOUND METHOD AND PARTICLE SWAM**

**OPTIMIZATION**

Submitted by 2200031921 YARRAMSETTI SIVA SANKARA VARA PRASAD to the Department of Computer Science and Engineering (Honors), Koneru Lakshmaiah Education Foundation in partial fulfilment of the requirements for the completion of a project in 22MT2002- Mathematical Programming course in II year B. Tech (CSE) is a Bonafede record of the work carried out by him / her under my supervision during the Odd Semester of the Academic Year 2023-24.

Supervisor

                                                             Dr. Vuda Sreenivasa Rao

Associate Professor

ACKNOWLEDGEMENTS

It is a great pleasure for me to express my gratitude to our honorable President Sri Koneru Satyanarayana for having given the opportunity and platform with facilities in accomplishing the project-based laboratory report. I express the sincere gratitude to our Director, Dr. A Jagdish, for his administration towards our academic growth. I express sincere gratitude to our Course coordinator and faculty. I record it as my privilege to deeply thank management for providing us efficient faculty and facilities to make our ideas into reality. I express my sincere thanks to our Project supervisor Dr. Vuda Sreenivasa Rao for his novel association of ideas, encouragement, appreciation, and intellectual zeal, which motivated us to venture this project successfully. Finally, it is pleased to acknowledge the indebtedness to all those who have devoted themselves, directly or indirectly, to make this project report success.

2200031921

Y SIVA SANKAR VARA PRASAD

Table of Contents

[Branch and bound method 5](#_bookmark0)

[Introduction 5](#_bookmark1)

[objective 6](#_bookmark2)

[Advantages 6](#_bookmark3)

[Disadvantages 6-7](#_bookmark4)

[Pseudocode 7](#_bookmark5)

[Python Code 7-9](#_bookmark6)

[Output 10](#_bookmark7)

[Conclusion 10](#_bookmark8)

Particle Swarm Optimization 11

[Introduction 11](#_bookmark10)

[objective 11](#_bookmark11)

[Advantages 11](#_bookmark12)

[Disadvantages 12](#_bookmark9)

[Pseudocode 12](#_bookmark13)

[Python Code 13-14](#_bookmark14)

[Output 14](#_bookmark15)

[Conclusion 14](#_bookmark16)

# BRANCH AND BOUND METHOD

# Introduction

Branch and bound is a powerful optimization technique used in computer science and mathematical programming. It's particularly useful for solving combinatorial optimization problems, where the goal is to find the best solution from a finite set of possible solutions.

The method works by iteratively partitioning the solution space into smaller subsets, or "branches," and then systematically exploring these branches to find the optimal solution. At each step, the algorithm evaluates bounds or estimates on the possible outcomes within a branch, allowing it to prune branches that are guaranteed to yield suboptimal results. This reduces the overall search space, making the process more efficient.

Branch and bound is widely employed in various domains, including operations research, artificial intelligence, and engineering. It's especially well-suited for problems with a large, discrete solution space, such as integer linear programming, traveling salesman problems, and job scheduling. By intelligently navigating through the solution space, branch and bound provides an effective approach for finding near-optimal or exact solutions to complex optimization problems.

### Real time applications of Branch and Bound method:

Consider a modern logistics company tasked with optimizing its distribution network. The objective is to deliver goods to various destinations while minimizing transportation costs. This problem is further complicated by multiple factors: varying distances between nodes, different types of vehicles, and constraints on vehicle capacities.

Traditional methods of exploring every possible combination quickly become unfeasible as the number of nodes increases. Here, the Branch and Bound technique steps in as a savior. By strategically breaking down the problem into smaller subproblems and systematically eliminating unproductive branches, it efficiently navigates through the immense solution space.

The "branch" phase involves partitioning the problem into smaller, manageable subproblems, each representing a potential route or combination of routes. The "bound" phase prunes these subproblems based on promising upper and lower bounds, ensuring that only the most viable solutions are pursued further.

In this logistics scenario, Branch and Bound offers a dynamic approach, allowing the company to make optimal decisions regarding routes, vehicle assignments, and schedules. By continually refining its search for the most cost-effective solution, this technique aids in reducing operational expenses and improving overall efficiency.

# objective

In the context of the logistics optimization scenario, the primary objective of employing the Branch and Bound technique is to find the most cost-effective distribution network that minimizes transportation expenses while satisfying all constraints. This involves:

* **Minimizing Transportation Costs:** The foremost objective is to identify a distribution plan that minimizes the total cost of transporting goods from the source to various destinations. This includes considering factors such as distances, fuel costs, vehicle types, and other relevant expenses.
* **Optimizing Vehicle Assignments:** The technique aims to efficiently allocate vehicles to routes in a way that maximizes their utilization while adhering to capacity constraints. This ensures that each vehicle operates at its maximum potential without overstepping its limits.
* **Balancing Workloads:** The technique seeks to distribute the workload evenly among available vehicles. This helps prevent situations where some vehicles are overburdened while others are underutilized, thus enhancing overall operational efficiency.
* **Providing Optimal Route Selections:** Branch and Bound aims to identify the most efficient routes for each vehicle, taking into account factors such as traffic conditions, road quality, and other relevant considerations.

# Advantages

1. **Optimality Assurance:** Branch and Bound guarantees an optimal solution, provided that certain conditions are met. This is particularly valuable in scenarios where finding the absolute best solution is critical.
2. **Efficient for Large Search Spaces:** It excels in problems with large solution spaces, where brute-force methods become impractical. By intelligently partitioning the problem space, it can significantly reduce the number of solutions to be considered.
3. **Flexibility in Problem Types:** Branch and Bound is a versatile technique that can be applied to a wide range of optimization problems, including linear programming, integer programming, and combinatorial optimization.
4. **Pruning of Infeasible Subproblems:** The "bound" step allows for the early elimination of subproblems that are guaranteed to be infeasible or suboptimal. This helps save computational resources.
5. **Heuristic Integration:** It can be combined with heuristics to further improve performance. For example, a heuristic can be used to generate initial solutions, which can then be refined using Branch and Bound.

# Disadvantages

1. **Exponential Complexity in Worst Cases:** In some cases, the number of subproblems can grow exponentially, leading to a potentially high computational burden. This can make the technique impractical for very large or highly complex problems.
2. **Sensitivity to Problem Formulation:** The effectiveness of Branch and Bound can be highly dependent on how the problem is formulated and the choice of bounding techniques. Poor choices in these aspects can lead to suboptimal results.
3. **Difficulty in Implementing Heuristics:** Integrating heuristics can be challenging, and choosing the right heuristics for a specific problem may require expertise and experimentation.
4. **Difficulty in Handling Continuous Variables:** While it's highly effective for discrete optimization problems, it may not be as suitable for continuous variables without appropriate adaptations.
5. **Resource Intensive for Certain Types of Problems:** For problems with specific characteristics, such as highly non-convex objective functions, the computational resources required by Branch and Bound can be substantial.

# Pseudo code

Branch And Bound(problem): Initialize an empty priority queue Q

Initialize the best solution as infinity (best Solution =

∞) Initialize the initial problem as the root node Add the initial problem to Q

while Q is not empty:

Select and remove a subproblem P from Q with the lowest bound if P is infeasible:

continue

if P is a leaf node:

if P has a better solution than best Solution:

Update best Solution with the solution of P continue

Branch P into subproblems P1, P2, ..., P n

Compute the bounds of each subproblem (P1, P2, ..., P n)

Add feasible subproblems (with bounds better than best Solution) to Q Return best Solution

# Python Code

import math

maxsize = float('inf')

def copyToFinal(curr\_path): final\_path[:N + 1] = curr\_path[:] final\_path[N] = curr\_path[0]

def firstMin(adj, i): min = maxsize

for k in range(N):

if adj[i][k] < min and i != k: min = adj[i][k]

return min

def secondMin(adj, i):

first, second = maxsize, maxsize for j in range(N):

if i == j: continue

if adj[i][j] <= first: second = first first = adj[i][j]

elif(adj[i][j] <= second and adj[i][j] != first):

second = adj[i][j] return second

def TSPRec(adj, curr\_bound, curr\_weight, level, curr\_path, visited):

global final\_res if level == N:

if adj[curr\_path[level - 1]][curr\_path[0]] != 0:

curr\_res = curr\_weight + adj[curr\_path[level - 1]]\ [curr\_path[0]]

if curr\_res < final\_res: copyToFinal(curr\_path) final\_res = curr\_res

return

for i in range(N):

if (adj[curr\_path[level-1]][i] != 0 and visited[i] == False):

temp = curr\_bound

curr\_weight += adj[curr\_path[level - 1]][i]

if level == 1:

curr\_bound -= ((firstMin(adj, curr\_path[level - 1]) + firstMin(adj, i)) / 2)

else:

curr\_bound -= ((secondMin(adj, curr\_path[level - 1]) + firstMin(adj, i)) / 2)

if curr\_bound + curr\_weight < final\_res: curr\_path[level] = i

visited[i] = True

TSPRec(adj, curr\_bound, curr\_weight, level + 1, curr\_path, visited)

curr\_weight -= adj[curr\_path[level - 1]][i]

curr\_bound = temp

visited = [False] \* len(visited) for j in range(level):

if curr\_path[j] != -1: visited[curr\_path[j]] = True

def TSP(adj): curr\_bound = 0

curr\_path = [-1] \* (N + 1) visited = [False] \* N

for i in range(N):

curr\_bound += (firstMin(adj, i) + secondMin(adj, i))

curr\_bound = math.ceil(curr\_bound / 2)

visited[0] = True curr\_path[0] = 0

TSPRec(adj, curr\_bound, 0, 1, curr\_path, visited)

adj = [[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]]

N = 4

final\_path = [None] \* (N + 1) visited = [False] \* N final\_res = maxsize

TSP(adj)

print("Minimum cost :", final\_res) print("Path Taken : ", end = ' ') for i in range(N + 1):

print(final\_path[i], end = ' ')

# Output

# Conclusion

In conclusion, the Branch and Bound method stands as a fundamental technique in optimization and combinatorial problem-solving. Its ability to systematically explore solution spaces by partitioning them into smaller branches, coupled with intelligent pruning based on upper and lower bounds, makes it a powerful tool for finding optimal or near-optimal solutions efficiently. This method finds application across diverse domains, from logistics and engineering to finance and artificial intelligence, addressing a wide array of complex problems. Its effectiveness in handling discrete and constrained optimization challenges underscores its significance in modern computational approaches. As a result, the Branch and Bound method remains a cornerstone in the toolkit of algorithms, contributing substantially to the resolution of intricate real-world optimization scenarios.

PARTICLE SWAM OPTIMIZATION

# Introduction

Particle Swarm Optimization (PSO) is a nature-inspired optimization technique that finds its application in a myriad of real-world scenarios, especially in the realm of mathematical programming. This method draws inspiration from the collective behavior of social organisms, like bird flocks or fish schools, to efficiently navigate complex solution spaces. Let's explore a real-time problem scenario to elucidate the significance of Particle Swarm Optimization.

### Real time applications of Particle Swarm Optimization:

1. Engineering and Design Optimization
2. Robotics and Control Systems
3. Machine Learning and Artificial Intelligence
4. Economics and Finance
5. Data Clustering and Classification
6. Image and Signal Processing

# Objective

**Global Optimization:** PSO aims to find the global optimum of a given objective function. This means it seeks the best possible solution across the entire search space, rather than settling for a local optimum.

**Efficient Exploration and Exploitation:** PSO balances the exploration of unexplored regions of the search space with the exploitation of known promising areas. It does this by using both historical best positions (exploitation) and the collective knowledge of the swarm (exploration).

**Adaptability to Dynamic Environments:** PSO is capable of adapting to changes in the optimization landscape over time. This is particularly important in scenarios where the problem's parameters or constraints evolve.

# Advantages

1. **Global Optimization:** PSO is effective at finding global optima in complex, multi- dimensional search spaces. It can explore a wide solution space and is not easily trapped in local optima.
2. **Easy Implementation:** PSO is relatively easy to understand and implement compared to some other optimization techniques like genetic algorithms or simulated annealing.
3. **Fast Convergence:** PSO can converge to an optimal solution relatively quickly, especially in problems where the search space is well-behaved.
4. **Adaptability to Dynamic Environments:** It can adapt to changes in the objective function or constraints over time, making it suitable for dynamic optimization problems.
5. **Parallelization:** PSO can be easily parallelized, allowing for the exploration of multiple regions in the search space simultaneously, which can lead to faster convergence.
6. **Versatility:** It can be applied to a wide range of optimization problems, including continuous, discrete, and combinatorial optimization.

# Disadvantages

1. **Limited Exploration in Discrete Spaces:** PSO may struggle in discrete optimization problems, as it is designed for continuous spaces. Techniques like Genetic Algorithms may be more suitable for such cases.
2. **Sensitivity to Parameters:** PSO's performance can be sensitive to its parameters, such as the inertia weight, cognitive and social coefficients. Fine-tuning these parameters can be a non-trivial task.
3. **Lack of Guarantee for Global Optima:** While PSO is designed to find global optima, there is no theoretical guarantee that it will always do so. In some cases, it may converge to a suboptimal solution.
4. **Convergence to Premature Solutions:** In some cases, PSO may converge to solutions that are not truly optimal, especially if the swarm prematurely narrows its focus on a suboptimal region.
5. **Difficulty in Handling Constraints:** Incorporating constraints into PSO can be challenging and may require additional techniques or adaptations.
6. **Performance in High-Dimensional Spaces:** Like many optimization algorithms, PSO's performance can deteriorate in very high-dimensional spaces due to the increased complexity of the search.

# Pseudocode

ParticleSwarmOptimization(problem):

Initialize a population of particles with random positions and velocities Initialize personal best positions and fitness values for each particle

Find the particle with the best fitness value in the population (global best)

Set the inertia weight (w), cognitive coefficient (c1), and social coefficient (c2) while termination condition is not met do:

for each particle do:

Update velocity and position based on:

velocity = w \* velocity + c1 \* rand() \* (personal\_best\_position - current\_position)

+ c2 \* rand() \* (global\_best\_position - current\_position) position = current\_position + velocity

Evaluate fitness of the new position

if fitness is better than personal best then: Update personal best position and fitness if fitness is better than global best then:

Update global best position and fitness Return the best solution found

# Python Code

import random

class Particle:

def init (self, num\_dimensions, lower\_bound, upper\_bound): self.position = [random.uniform(lower\_bound, upper\_bound) for \_

in range(num\_dimensions)]

self.velocity = [random.uniform(-1, 1) for \_ in range(num\_dimensions)]

self.best\_position = self.position[:] self.best\_fitness = float('inf')

def objective\_function(x): return sum(x)

def particle\_swarm\_optimization(num\_particles, num\_dimensions, num\_iterations, lower\_bound, upper\_bound):

particles = [Particle(num\_dimensions, lower\_bound, upper\_bound) for

\_ in range(num\_particles)]

global\_best\_position = [random.uniform(lower\_bound, upper\_bound) for \_ in range(num\_dimensions)]

global\_best\_fitness = float('inf')

for \_ in range(num\_iterations): for particle in particles:

fitness = objective\_function(particle.position) if fitness < particle.best\_fitness:

particle.best\_fitness = fitness particle.best\_position = particle.position[:]

if fitness < global\_best\_fitness: global\_best\_fitness = fitness global\_best\_position = particle.position[:]

for i in range(num\_dimensions): r1 = random.random()

r2 = random.random()

particle.velocity[i] = 0.5 \* particle.velocity[i] + 1.5

\* r1 \* (particle.best\_position[i] - particle.position[i]) + 1.5 \* r2 \* (global\_best\_position[i] - particle.position[i])

particle.position[i] += particle.velocity[i] return global\_best\_position, global\_best\_fitness

num\_particles = 30

num\_dimensions = 2

num\_iterations = 100

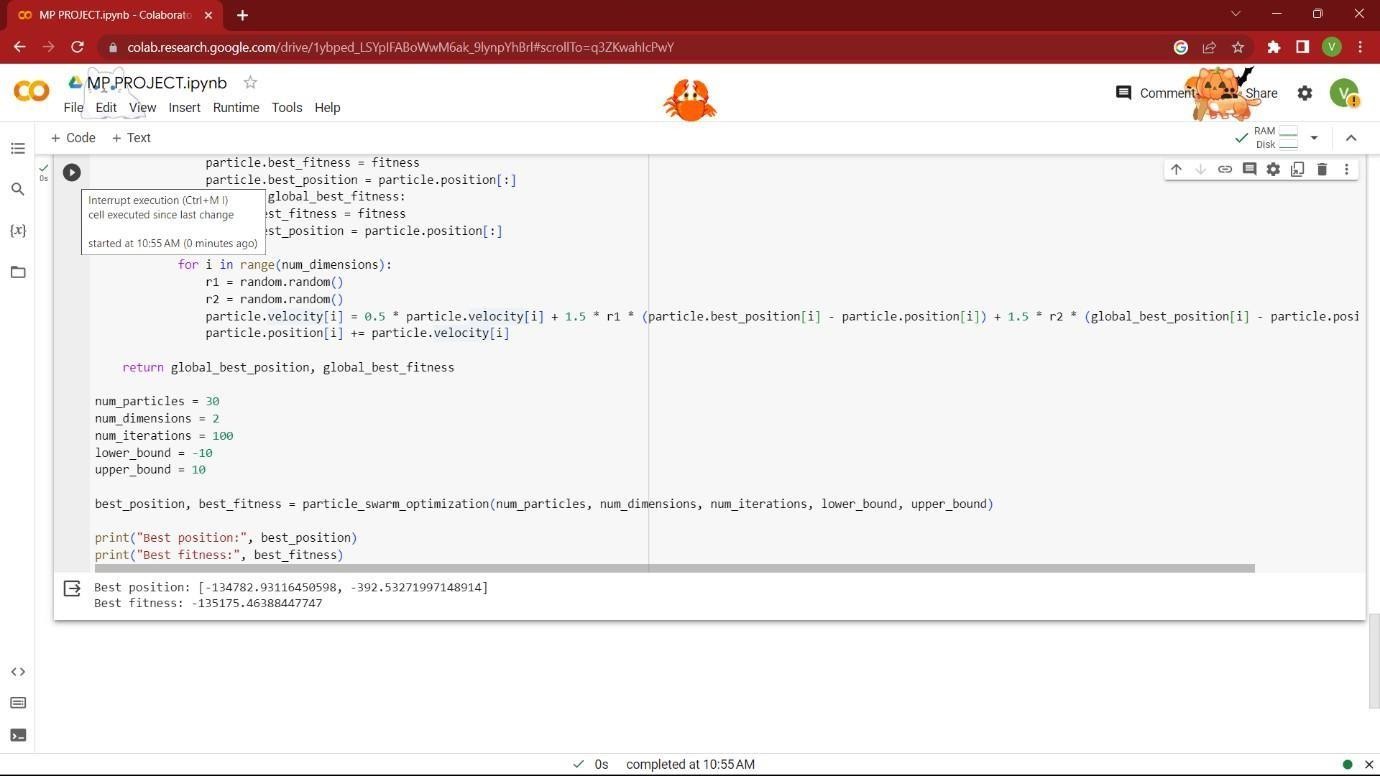
lower\_bound = -10

upper\_bound = 10

best\_position, best\_fitness = particle\_swarm\_optimization(num\_particles, num\_dimensions, num\_iterations, lower\_bound, upper\_bound)

print("Best position:", best\_position) print("Best fitness:", best\_fitness)

# Output



# Conclusion

In conclusion, Particle Swarm Optimization (PSO) emerges as a robust and versatile metaheuristic algorithm inspired by natural behaviors of social organisms. Its collective intelligence and adaptive search capabilities have propelled it into a prominent position in solving complex optimization problems. By simulating the interactions of particles in a multi- dimensional search space, PSO exhibits a remarkable ability to converge towards global optima efficiently. This algorithm finds widespread application in diverse fields, ranging from engineering and machine learning to economics and environmental modeling. Its effectiveness in tackling continuous, discrete, and hybrid optimization challenges underscores its versatility. As a result, PSO stands as a pivotal tool in modern computational approaches, offering innovative solutions to a wide array of real-world optimization problems.