**Assignment No. 1**

Name: Gaurang P. Chavan

Class: TY-AI

Roll no:331004

Batch:A1

Title:Compare different algorithms and evaluate their performance/cost.E.g. depth- first search (DFS) to heuristic algorithms such as Monte Carlo tree search (MCTS.

# Study Assignment 1: Comparison of Depth-First Search (DFS) and Monte Carlo Tree Search (MCTS)

## Aim

The aim of this study is to compare two fundamentally different search algorithms — Depth-First Search (DFS), a classical uninformed search technique, and Monte Carlo Tree Search (MCTS), a heuristic and probabilistic approach — by evaluating their performance in terms of:

* Effectiveness: How quickly a solution is found from a given solvable problem.
* Efficiency: The number of nodes traversed before reaching the solution.
* Practical suitability: The type of problems where each algorithm is more advantageous.

By understanding these differences, we can choose the appropriate algorithm for solving different classes of problems, especially those involving complex or large search spaces.

## Theory

### Depth-First Search (DFS)

DFS is one of the fundamental graph traversal algorithms. It explores nodes starting from a root (or start node) by moving deep into the graph, following one path until it either reaches a solution or a dead end. Upon reaching a dead end, DFS backtracks and explores alternative paths.

Key characteristics:

* Blind Search: DFS does not utilize any information about the goal’s location or the problem’s structure.
* Stack-based Exploration: DFS uses an explicit or implicit stack (via recursion) to remember nodes to revisit.
* Completeness: DFS is complete in finite search spaces without cycles.
* Non-optimal: It does not guarantee the shortest or best path to a solution.

DFS works well when the solution is expected to be located deep in the search tree or when memory is constrained.

### Monte Carlo Tree Search (MCTS)

MCTS is a more recent, heuristic-driven algorithm designed for problems where the search space is enormous or not fully understood, such as strategic games. It builds a search tree incrementally and uses statistical sampling (random simulations) to evaluate potential moves or states.

Core principles:

* Exploration vs. Exploitation: MCTS balances trying new moves (exploration) and leveraging moves known to perform well (exploitation) using policies like Upper Confidence Bound for Trees (UCT).
* Four Phases: Selection, Expansion, Simulation (Rollout), and Backpropagation iteratively refine the tree.
* Probabilistic Evaluation: Unlike DFS, MCTS does not guarantee immediate optimality but converges toward the best solution over time.
* Scalability: Efficiently handles large and complex search spaces by focusing computational effort on promising areas.

MCTS is widely applied in game AI and planning domains where exhaustive search is infeasible.

## Example & Algorithm

### DFS Example and Pseudocode

Example Scenario:  
Finding a path in a maze from a start position to the exit.

Pseudocode:

def DFS(node, goal, visited):

if node == goal:

return True

visited.add(node)

for neighbor in node.neighbors:

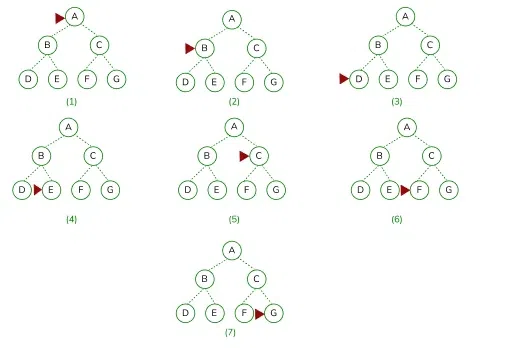
if neighbor not in visited:

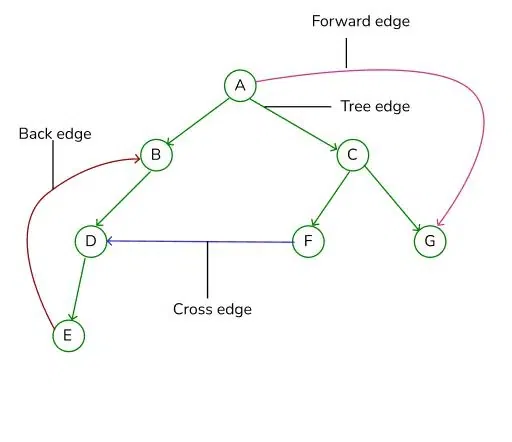
if DFS(neighbor, goal, visited):

return True

return False

* DFS explores neighbors depth-wise.
* If the goal is found, it returns immediately.
* Otherwise, it backtracks and tries other paths.





### MCTS Example and Algorithm Overview

Example Scenario:  
Selecting the best next move in the game of Go.

Algorithm Steps:

1. Selection: Starting at the root, recursively select child nodes based on a policy like UCT:

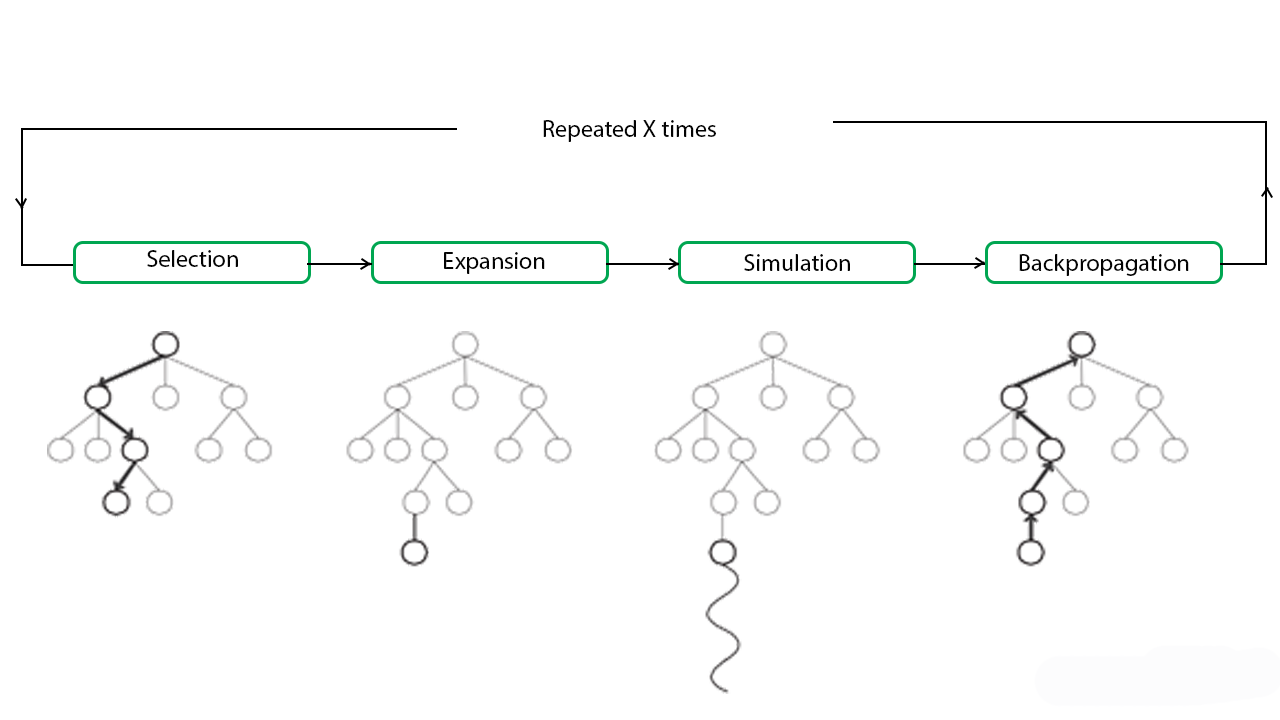


where; Si = value of a node i xi = empirical mean of a node i

C = a constant  t = total number of simulations

1. Expansion: Add a new child node to the tree once an unexplored move is reached.
2. Simulation: Run a random playout (rollout) from the new node to the end of the game or some horizon.
3. Backpropagation: Update statistics (visits, wins) of all nodes on the path back to the root based on simulation results.

Repeat these steps for a set number of iterations or time budget. The move leading to the child node with the highest win ratio or visit count is selected.

Code:

# main function for the Monte Carlo Tree Search

def monte\_carlo\_tree\_search(root):

while resources\_left(time, computational power):

leaf = traverse(root)

simulation\_result = rollout(leaf)

backpropagate(leaf, simulation\_result)

return best\_child(root)

# function for node traversal

def traverse(node):

while fully\_expanded(node):

node = best\_uct(node)

# in case no children are present / node is terminal

return pick\_unvisited(node.children) or node

# function for the result of the simulation

def rollout(node):

while non\_terminal(node):

node = rollout\_policy(node)

return result(node)

# function for randomly selecting a child node

def rollout\_policy(node):

return pick\_random(node.children)

# function for backpropagation

def backpropagate(node, result):

if is\_root(node) return

node.stats = update\_stats(node, result)

backpropagate(node.parent)

# function for selecting the best child

# node with highest number of visits

def best\_child(node):

pick child with highest number of visits

## Complexity:

| Algorithm | Time Complexity | Space Complexity |
| --- | --- | --- |
| DFS | Worst-case: O(bd)O(b^d), where bb = branching factor, dd = depth of solution | O(d)O(d) (stack depth for recursion) |
| MCTS | Approx. O(n×tsim)O(n \times t\_{sim}), nn = iterations, tsimt\_{sim} = simulation time per rollout | O(n)O(n) (nodes stored in the search tree) |

* DFS Complexity: Exponential in the depth of the solution and the branching factor, but memory use is linear in depth due to stack.
* MCTS Complexity: Depends on the number and length of simulations; more simulations improve accuracy but increase computational cost.

## Applications

### DFS Applications

* Solving puzzles (e.g., Sudoku, mazes).
* Graph traversal tasks such as cycle detection.
* Generating mazes and topological sorting.
* Scenarios requiring complete exploration with limited memory.

### MCTS Applications

* AI in complex board games (Go, Chess, Hex).
* Automated planning and decision making under uncertainty.
* Robotics navigation and control where full modeling is infeasible.
* Real-time strategy games requiring adaptive, on-the-fly decisions.

## Conclusion

Depth-First Search and Monte Carlo Tree Search represent two ends of the search algorithm spectrum: DFS is a deterministic, exhaustive, and memory-efficient method that blindly searches the state space, while MCTS is a heuristic, probabilistic search that strategically focuses computation on promising areas using simulations.

DFS’s simplicity makes it effective for small, well-defined problems but limits scalability in large or complex search spaces. In contrast, MCTS excels at handling large, uncertain, or stochastic environments by balancing exploration and exploitation, providing better performance when resources allow more computation.

In practice, the choice depends on the problem domain: DFS fits problems with modest search space and where solution depth is manageable, whereas MCTS is suited for complex domains requiring approximate, adaptive search strategies.