# Summary of Bidirectional Search and Backtracking Search

1. **Bidirectional Search Algorithm**

* **Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.**
* **Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.**

**Advantages**

* Bidirectional search is fast.
* Bidirectional search requires less memory
* The graph can be extremely helpful when it is very large in size and there is no way to make it smaller. In such cases, using this tool becomes particularly useful.
* The cost of expanding nodes can be high in certain cases. In such scenarios, using this approach can help reduce the number of nodes that need to be expanded.

**Disadvantages**

* Implementation of the bidirectional search tree is difficult.
* In bidirectional search, one should know the goal state in advance.
* Finding an efficient way to check if a match exists between search trees can be tricky, which can increase the time it takes to complete the task.
* **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
* **Time Complexity:** Time complexity of bidirectional search using BFS is **O(bd)**.
* **Space Complexity:** Space complexity of bidirectional search is **O(bd)**.
* **Optimal:** Bidirectional search is Optimal.

1. **Backtracking Search Algorithm**

Backtracking is a search technique for solving complex problems by recursively exploring combinations of possible choices to arrive at a solution. It is widely applied to solve problems in search, optimization, planning, and gaming. The algorithm is based on a depth-first search approach, systematically exploring options until a solution is found or all possibilities are exhausted.

**Algorithm Steps**

1. Begin with an initial candidate solution.

2. Attempt to extend the candidate by adding a valid step.

3. If constraints are violated:

* + **Backtrack**: Remove the last step and explore other options.

4. If a valid solution is found:

- Return the solution.

5. If no solution exists:

* Report failure.

This process continues until all possibilities are exhausted or a solution is identified.

**Features**

* Backtracking is often implemented using recursion.
* It is particularly effective for problems with large search spaces where constraints can eliminate infeasible paths.

**Advantages**

* It eliminates invalid candidates early, thereby reducing unnecessary computational efforts.
* It is applicable to a wide range of problems, including puzzles, constraint satisfaction problems (CSPs), and optimization tasks.
* The recursive nature of the implementation makes the algorithm relatively straightforward to implement.

**Disadvantages**

* The worst-case time complexity is (O(b^d)).
* In the absence of effective heuristics, the algorithm may explore a large number of unnecessary paths, leading to inefficiency.
* Deep recursion can result in significant memory consumption due to the recursive call stack

Applications include solving puzzles, addressing graph problems, optimizing with the knapsack problem, and handling scheduling and resource allocation.

* **Completeness:** Backtracking is complete for finite search spaces.
* **Optimality**: It does not guarantee optimality unless combined with heuristics or additional logic.
* **Space** **Complexity**: Proportional to the depth of the recursion (O(d)).