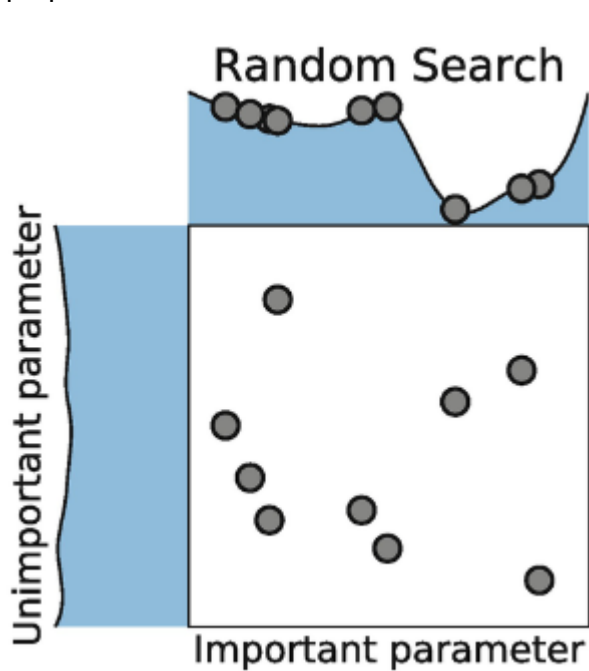


Unit-2: Introduction to AI, Part-2:

Random Search:

Search algorithms are one of the most important areas of Artificial Intelligence. Random Search is a way to optimize the performance of machine learning algorithms by randomly selecting combinations of hyperparameters. This technique can be used in discrete, continuous, and mixed settings and is especially effective when the optimization problem has a low intrinsic dimensionality.

Random Search is also embarrassingly parallel, which means that it can be easily distributed across multiple processors or machines. This allows for faster optimization and better scaling to larger problems. In addition to its simplicity and scalability, Random Search also allows for the inclusion of prior knowledge by specifying the distribution from which to sample. This can be useful when there is some knowledge or intuition about what values the hyperparameters should take on.



It is a powerful technique for hyperparameter optimization in machine learning. It offers a simple and scalable alternative to exhaustive search methods like Grid search, and can outperform these methods when the optimization problem has a low intrinsic dimensionality. With its ability to handle discrete, continuous, and mixed settings, Random Search is a versatile tool that can be applied to a wide range of machine learning problems.

Time Complexity: $O(n)$ (linear time complexity)

Space Complexity: $O(1)$ (constant space complexity)

Search with Open and Closed List:

Open List:

1. The open list is like a waiting area for states or nodes that the algorithm has discovered but hasn't explored yet.
2. It holds nodes that are potential candidates for further investigation.
3. Nodes in the open list are often sorted based on some criteria, like their cost or priority.

Closed List:

1. The closed list keeps track of nodes or states that have already been explored.
2. Once a node is expanded or visited, it is moved from the open list to the closed list.
3. This helps prevent revisiting the same state, avoiding loops in the search process.

Purpose:

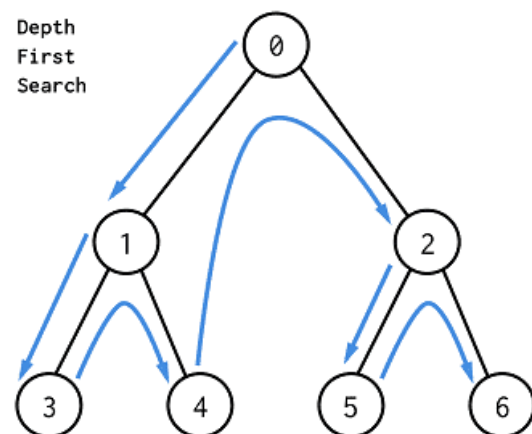
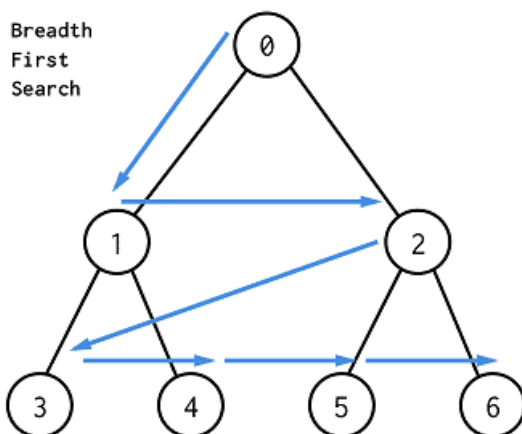
1. Open and closed lists help manage the search efficiently by keeping track of what has been explored and what still needs to be examined.
2. They prevent redundant exploration of the same states, making the search process more focused and avoiding unnecessary computations.

Benefits:

1. These lists contribute to the efficiency of search algorithms by guiding the exploration process.
2. They are common in various search algorithms, not limited to a specific one, ensuring adaptability to different problem-solving scenarios.

The use of open and closed lists is a fundamental strategy in search algorithms, helping to organize the exploration of states or nodes and improving the overall efficiency of the search process.

Depth First and Breadth First Search:



Depth First Search:

DFS adds the new candidates at the head of the list. The reason it is called depth first is that from the search tree, it selects a node from the OPEN list that is deepest or farthest from the start node. The candidates are inspected in the last-in-first-out order. This way its focus is on the newer arrivals first, and consequently its characteristic behavior is that it dives headlong into the search space. The blow figure illustrates the search tree as seen by DFS, represented by the two lists OPEN and CLOSED.

```
DepthFirstSearch()
1 open ← ((start NIL))
2 closed ← {}
3 while not Null(open)
4   do nodePair ← Head(open)
5     node ← Head(nodePair)
6     if GoalTest(node) = TRUE
7       then return ReconstructPath(nodePair, closed)
8     else closed ← Cons(nodePair, closed)
9       children ← MoveGen(node)
10      noLoops ← RemoveSeen(children, open, closed)
11      new ← MakePairs(noLoops, node)
12      open ← Append(new, Tail(open))
13 return "No solution found"

RemoveSeen(nodeList, openList, closedList)
1 if Null(nodeList)
2   then return {}
3   else n ← Head(nodeList)
4     if (OccursIn(n, openList) OR OccursIn(n, closedList))
5       then return RemoveSeen(Tail(nodeList), openList, closedList)
6       else return Cons(n, RemoveSeen(Tail(nodeList), openList, closedList))

OccursIn(node, listOfPairs)
1 if Null(listOfPairs)
2   then return FALSE
3   else if n = Head(Head(listOfPairs))
4     then return TRUE
5     else return OccursIn(node, Tail(listOfPairs))

MakePairs(list, parent)
1 if Null(list)
2   then return {}
3   else return Cons(MakeList(Head(list), parent),
                    MakePairs(Tail(list), parent))
```

Time Complexity:

DFS can have varying time complexities based on the implementation and the structure of the graph. In the worst case, it can be $O(V + E)$, similar to BFS, if it explores all vertices and edges.

Space Complexity:

The space complexity of DFS is influenced by the recursion stack or an explicit stack used to store nodes. In the worst case, it can be $O(V)$ for the recursion stack, where V is the number of vertices. If an explicit stack is used, the space complexity can be $O(V + E)$.

Breadth First Search:

BFS on the other hand, is very conservative. It inspects the nodes generated on a first come, first served basis. That is, the nodes from a queue to be inspected. The data structure Queue implements the first in, first out order. The BFS algorithm is only a small modification on the DFS algorithm. Only the order in the append statement where OPEN is updated needs to be changed.

```

BreadthFirstSearch()
1  open ← ((start, NIL))
2  closed ← ()
3  while not Null(open)
4    do nodePair ← Head(open)
5      node ← Head(nodePair)
6      if GoalTest(node) = TRUE
7        then return ReconstructPath(nodePair, closed)
8      else closed ← Cons(nodePair, closed)
9        children ← MoveGen(node)
10       noLoops ← RemoveSeen(children, open, closed)
11       new ← MakePairs(noLoops, node)
12       open ← Append(Tail(open), new)
13 return "No solution found"

```

Time Complexity:

BFS explores nodes level by level, starting from the source. In the worst case, it visits all nodes, so the time complexity is $O(V + E)$, where V is the number of vertices (nodes) and E is the number of edges in the graph.

Space Complexity:

BFS typically requires additional space to store the visited nodes and the queue data structure. In the worst case, the space complexity is $O(V)$, where V is the number of vertices.

Comparison between DFS and BFS:

Depth First Search	Breath First Search
DFS requires less memory since only the nodes on the current path are stored.	BFS guarantees that the space of possible moves is systematically examined; this search requires considerable memory resources.
By chance, DFS may find a solution without examining much of the search space at all. Then it finds solution faster.	The search systematically proceeds testing each node that is reachable from a parent node before it expands to any child of those nodes.
If the selected path does not reach to the solution node, DFS gets stuck into a blind alley.	BFS will not get trapped exploring a blind alley.
Does not guarantee to find solution. Backtracking is required if wrong path is selected.	If there is a solution, BFS is guaranteed to find it.

Heuristic Search:

Heuristic search is a strategy that incorporates additional information, called heuristics, to guide the search algorithm towards the most promising candidates for exploration. Unlike the systematic exploration of BFS or DFS, heuristic search aims to prioritize nodes based on their estimated likelihood of leading to a solution.

Heuristic search selects nodes for exploration based on a priority determined by a heuristic function. This function estimates the desirability of a node, considering its potential to lead to a solution. Similar to DFS, heuristic search might prioritize nodes that appear deeper or farther from the initial state, based on the heuristic information. The focus is on exploring nodes that seem more promising according to the heuristic. The order of exploration might not strictly adhere to a last-in-first-out order, as this depends on the specific heuristic and the data structure used. However, the idea is to prioritize newer arrivals or nodes that seem more likely to lead to a solution. Heuristic search maintains OPEN and CLOSED lists similar to DFS, with the OPEN list containing nodes to be explored, prioritized by their heuristic values.

In summary, heuristic search differs from traditional searches like BFS and DFS by incorporating heuristic information to efficiently guide the exploration process towards more promising solutions, aiming for optimality in a quicker manner.