# *CSL020U4E: Artificial Intelligence*
# *Lecture* 02 *(State Space Search)*

Sumit Kumar Pandey

August, 2025

## State Space Search

- Problem solving basically involves doing the right thing at the right time.

Given a problem to solve, the task is to select the right moves that would lead to the solution.

## *Some Problems*

- The Map Colouring Problem.
- The $N$-Queens Problem.
- The SAT Problem.
- The 8-Puzzle Problem.
- River Crossing Puzzle.
- The Water Jug Problem.
- The Travelling Salesman Problem.
- The Rubik Cube Problem.

## State Space Search

- We use the term *agent* to refer to the problem solver.
- The problem solver, or the agent, operates on a representation of the space in which the problem has to be solved.
- And also a representation of the moves or decisions that the agent has to choose from to solve the problem.
- Our endeavour will be to write the algorithms in a domain-independent manner.
- Then the algorithms will be general purpose in nature and could be adapted to solve problems in different domains.
- A user would only have to impement the domain description and call the domain specific functions from the general program.

## *The State Space*

- The **state space** is the space of all possible **states**.
- A **state** is a description of the world in some language and corresponds to a node in the **state space**.
- The problem solver begins in a given or **start** state and has some desired state as **goal** state.
- The **desired state** can be a single state that is completely specified, or it may be a set of states described by a common property.

## *The State Space*

- The agent has a access to a neighbourhood function, which defines all the **moves** that can be made to transform any state in some way.
- We will use the name MoveGen function for the neighbourhood function, to signify that it *generates the moves* in a given state.
  - It takes a state as input and returns the set of its neighbours.
  - The MoveGen function will have to be supplied by the user for our solvers.
- We will also assume that the user has defined a GoalTest function that takes a state as input and determines whether it is a goal state or not.
  - This will provide a termination criterion for the search algorithm.

## *The State Space and* MoveGen *Function*

- The state space can be seen as a graph which is defined by the MoveGen function.
- The edges in a state space may be directed when the moves are not reversible, and undirected when they are.
- In some problems, some moves may be reversible and some are not. For example, the water jug problem.
- The MoveGen function captures all these variations.

For any node as input, the MoveGen function returns the set of neighbour nodes reachable in one move.

```
MoveGen($N$)
  Input: a node $N$
  Output: the neighbours of node $N$
```

## *Generate and Test*

- A search algorithm navigates the state space generated by the MoveGen function.
- It will terminate when it has reached the goal state.

We require the user to supply another function, GoalTest, that will identify a goal state.

GoalTest($N$)

  Input: a node $N$

  Output: *true* if $N$ is a goal state, and *false* otherwise.

## *A Search Algorithm*

- The high level search algorithm maintains a set of candidate states, which is traditionally called *OPEN*, and repeatedly picks some node $N$ from *OPEN*, till it picks a goal node. *OPEN* is initialized to the start stae $S$.
- This search algorithm returns the goal state if it finds one, else it returns *fail*.

```
SimpleSearch()
  $OPEN \leftarrow \{S\}$
  while $OPEN$ is not empty
    do pick some node $N$ from $OPEN$
      $OPEN \leftarrow OPEN - \{N\}$
      if GoalTest($N$) = TRUE
        then return $N$
      else $OPEN \leftarrow OPEN \cup$ MoveGen($N$)
  return FAILURE
```

## *Performance of a Search Algorithm*

We will evaluate the performance of our search algorithms on the following four criteria:

1. Completeness
2. Quality of Solution
3. Space Complexity
4. Time Complexity

## *Performance of a Search Algorithm*

Completeness:

- An algorithm is said to be complete if it is guaranteed to find a goal state if one is reachable.
- We also call such algorithms as being systematic. By this we mean that the algorithm searches the entire space and returns fail only if a goal state is not reachable.

Quality of Solution:

- We begin with the length of the path found as a measure of quality.
- Later, we will associate edge costs with each move, and then the total cost of the path will be the measure.

## *Performance of a Search Algorithm*

Space Complexity:

- This looks at the amount of space the algorithm requires to execute.
- We will see that this will be a critical measure, as the number of candidates in search space often grows exponentially.

Time Complexity:

- This describes the amount of time needed by the algorithm, measured by the number of candidates inspected.
- The most desirable complexity will be linear in path length, but one will have to often contend with exponential complexity.

## *Kinds of Problems*

1. Planning Problems: Planning problems are problems in which the sequence of moves to the goal state is of interest. Often in such problems the goal state is clearly specified, and one needs to find a path to a goal state.
2. Configuration Problems: Configuration problems have only one abstract description of the desired state, and the task is to find a state that conforms to the description. The path to the goal state is generally of no interest.

## *Configuration Problems*

- The Map Colouring Problem.
- The $N$-Queens Problem.
- The SAT Problem.

## *Planning Problems*

- The 8-Puzzle Problem.
- River Crossing Puzzle.
- The Water Jug Problem.
- The Travelling Salesman Problem.
- The Rubik Cube Problem.

## Blind Search

## *Blind Search*

- We devise algorithms for navigating the implicit search space and look at their properties.
- One distinctive feature of the algorithms in our discussion is that they are all **blind** or **uninformed**.
- This means that the way the algorithms search the space is always the same irrespective of the problem instance being solved.

## SimpleSearch() - Version 2

- We consider *OPEN* as a list now.

```
SimpleSearch()
  $OPEN \leftarrow [S]$
  while $OPEN$ is not empty
    $N \leftarrow$ head $OPEN$
    $OPEN \leftarrow$ tail $OPEN$
    if GoalTest($N$) = TRUE
      then return $N$
    else Combine MoveGen($N$) with $OPEN$
  return FAILURE
```

## *Search Trees*

- The candidate that the search algorithm picks from *OPEN* determines its behaviour.
- Algorithm SimpleSearch picks the node at the head of *OPEN*.
- The question now is where the new nodes generated by MoveGen are added.
- The search space the algorithm will explore in some order is a tree, which we call the **search tree**.
- The search tree depicts the space as viewed by a given search algorithm. Each variation of the algorithm generates its own search tree.
- Given that we have decided to extract the next candidate from the head of the *OPEN* list, we need to decide how to add the new nodes to the *OPEN*.

## *Depth First Search (DFS) and Breadth First Search (BFS)*

DFS: Treat *OPEN* as a **stack** data structure.

- When new nodes are added to the head of *OPEN*, the newest nodes are the first ones to be inspected.
- The stack data structure is characterized by the last-in-first-out (LIFO) behaviour.

BFS: Treat *OPEN* as a **queue** data structure.

- When new nodes are added to the rear of *OPEN*, the newest nodes are the last ones to be inspected.
- The queue data structure is characterized by the first-in-first-out (FIFO) behaviour.

## Depth First Search (DFS)

```
SimpleSearchDFS()
  $OPEN \leftarrow [S]$
  while $OPEN$ is not empty
    $N \leftarrow$ head $OPEN$
    $OPEN \leftarrow$ tail $OPEN$
    if GoalTest($N$) = TRUE
      then return $N$
    else $OPEN \leftarrow$ MoveGen($N$) ++ $OPEN$
  return FAILURE
```

## Breadth First Search (BFS)

```
SimpleSearchBFS()
    $OPEN \leftarrow [S]$
    while $OPEN$ is not empty
        $N \leftarrow$ head $OPEN$
        $OPEN \leftarrow$ tail $OPEN$
        if GoalTest($N$) = TRUE
            then return $N$
        else $OPEN \leftarrow OPEN$ ++ MoveGen($N$)
    return FAILURE
```

## *How to stop going around in circles*

- We implement this idea by maintaining another list apart from *OPEN*, that stores the nodes already inspected. This list is traditionally called *CLOSED*.
- Every time we remove a node *N* from *OPEN* for inspection, we add it to *CLOSED*.
- And before adding the neighbours of *N* generated by the MoveGen function, we remove those nodes that are already present in *CLOSED*.

## RemoveSeen *Function*

- RemoveSeen accepts a list of nodes, two lists *OPEN* and *CLOSED*, and filters out those nodes in the *nodeList* that are present in either *CLOSED* or *OPEN*.
- RemoveSeen calls function OccursIn(*node*, *LIST*) to check if a given *node* occurs somewhere in the *LIST*.

```
RemoveSeen(nodeList, OPEN, CLOSED)
  if nodeList is empty
    return empty list
  else node ← head nodeList
    if OccursIn(node, OPEN) or OccursIn(node, CLOSED)
      return RemoveSeen(tail nodeList, OPEN, CLOSED)
    else return node: RemoveSeen(tail nodeList, OPEN,
                                                CLOSED)
```

## OccursIn *Function*

```
OccursIn(node, LIST)
  if LIST is empty
    return FALSE
  elseif node = head LIST
    return TRUE
  else return OccursIn(node, tail LIST)
```

## *Complete Algorithm - Using DFS*

```
SimpleSearchDFS()
  $OPEN \leftarrow [S]$
  $CLOSED \leftarrow [\,]$
  while $OPEN$ is not empty
    $N \leftarrow$ head $OPEN$
    if GoalTest($N$) = TRUE
      then return $N$
    else $CLOSED \leftarrow N : CLOSED$
      $children \leftarrow$ MoveGen($N$)
      $newNodes \leftarrow$ RemoveSeen($children, OPEN, CLOSED$)
      $OPEN \leftarrow newNodes ++$ (tail $OPEN$)
  return FAILURE
```

## *Complete Algorithm - Using BFS*

```
SimpleSearchBFS()
  $OPEN \leftarrow [S]$
  $CLOSED \leftarrow [\,]$
  while $OPEN$ is not empty
    $N \leftarrow$ head $OPEN$
    if GoalTest($N$) = TRUE
      then return $N$
    else $CLOSED \leftarrow N : CLOSED$
      $children \leftarrow$ MoveGen($N$)
      $newNodes \leftarrow$ RemoveSeen($children$, $OPEN$, $CLOSED$)
      $OPEN \leftarrow$ (tail $OPEN$) ++ $newNodes$
  return FAILURE
```

## *Reconstructing the Path*

- In a **configuration problem**, it suffices to find a goal node that matches the description.
- But in a **planning problem**, we require our search algorithm to return the path to the goal node.
- We do this by modifying the **node representation** in the search tree, while the problem space remains as given to us.

## *Node Representation*

- Each node in the search tree keeps track of the parent it was generated from.
- The node in the search tree, which we call a *nodePair* is a pair of two nodes, the node itself and its parent node, i.e. $nodePair = (Node, Parent)$.

## ReconstructPath *Function*

- ReconstructPath function accepts the *nodePair* containing the goal node and constructs the path by tracing the parents via the *nodePairs* stored in *CLOSED*.
- It uses an ancillary function FindLink(*node*, *CLOSED*) which fetches the *nodePair* in which node is the first element.

```
ReconstructPath(nodePair, CLOSED)
  (node, parent) ← nodePair
  path ← node:[ ]
  while parent is not null
    path ← parent: path
    (_, parent) ← FindLink(parent, CLOSED)
  return path
```

## FindLink *Function*

```
FindLink($node$, $CLOSED$)
  if $node$ = first head $CLOSED$
    return head $CLOSED$
  else return FindLink($node$, tail $CLOSED$)
```

## *Complete Algorithm - Using DFS*

```
SimpleSearchDFS()
  $OPEN \leftarrow (S, null) : [\,]$
  $CLOSED \leftarrow [\,]$
  while $OPEN$ is not empty
    $nodePair \leftarrow$ head $OPEN$
    $(N, \_) \leftarrow nodePair$
    if GoalTest($N$) = TRUE
      return ReconstructPath($nodePair$, $CLOSED$)
    else $CLOSED \leftarrow nodePair : CLOSED$
      $children \leftarrow$ MoveGen($N$)
      $newNodes \leftarrow$ RemoveSeen($children$, $OPEN$, $CLOSED$)
      $newPairs \leftarrow$ MakePairs($newNodes$, $N$)
      $OPEN \leftarrow newPairs$ ++ (tail $OPEN$)
  return empty list
```

## Complete Algorithm - Using BFS

```
SimpleSearchBFS()
  $OPEN \leftarrow (S, null) : [\,]$
  $CLOSED \leftarrow [\,]$
  while $OPEN$ is not empty
    $nodePair \leftarrow$ head $OPEN$
    $(N, \_) \leftarrow nodePair$
    if GoalTest($N$) = TRUE
      return ReconstructPath($nodePair$, $CLOSED$)
    else $CLOSED \leftarrow nodePair : CLOSED$
      $children \leftarrow$ MoveGen($N$)
      $newNodes \leftarrow$ RemoveSeen($children$, $OPEN$, $CLOSED$)
      $newPairs \leftarrow$ MakePairs($newNodes$, $N$)
      $OPEN \leftarrow$ (tail $OPEN$) ++ $newPairs$
  return empty list
```

## MakePairs *Function*

- MakePairs recurses down a list of nodes in *nodeList*, for each making a pair with the parent node. For example, $\textsc{MakePairs}([A, B, C, D], S) = [(A, S), (B, S), (C, S), (D, S)]$

```
MakePairs(nodeList, parent)
  if nodeList is empty
    return empty list
  else return (head nodeList, parent): MakePairs(tail
                                                  nodeList, parent)
```

## *Comparing DFS and BFS*

There are four criteria we have stated for analysing a search algorithm.

1. Completeness
2. Quality of Solution
3. Space Complexity
4. Time Complexity

## *Completeness*

BFS

- If a path exists from the start state to the goal state, then BFS will find a path.
- This will happen even if the search tree is infinite.
- Not only that, it will find the shortest path to the goal node.
- The moment a goal node appears in a new layer, it will be found.

DFS

- DFS can go down an **infinite path** even though there is a path to the goal.

DFS and BFS both are complete for finite state spaces.

## *Quality of Solution*

BFS

- BFS always finds a shortest path.
- This is a direct consequence of its conservative tendency of sticking closer to the start node.
- It pushes into the search tree level by level.
- When it first encounters a goal node, it is the earliest occurrence, and hence a shortest path.
- This is true even if the state space, and correspondingly the search space, is infinite.

DFS

- DFS may not give you a shorest path.
- It has no affinity to the start node.
- It strives to be as far away as possible,since it always picks the deepest node from *OPEN* in the search tree.

## *Space Complexity*

Traditionally, space complexity is estimated by the size of *OPEN*, which stores the candidate nodes yet to be inspected and defines the search frontier.

<u>BFS</u>

- If the branching factor is $d$, then the number of nodes in *OPEN* at depth $d$ (root node is at depth 0) is

$$|OPEN_{BFS}| = b^d$$

## *Space Complexity*

DFS

- DFS is a clear winner here as search goes deeper into the tree, *OPEN* grows linearly with a constant number of nodes being added at every level.
- If the branching factor is $d$, then the number of nodes in *OPEN* at depth $d$ (root node is at depth 0) is

$$|OPEN_{DFS}| = d(b-1)+1$$

## *Time Complexity*

Time complexity can be measured in terms of the number of times the GoalTest function is applied, or the numer of nodes inspected by the algorithm. This is the size of *CLOSED*.

- The time taken by the two search algorithms depends on where the goal node lies.
- If the goal node lies deep into the tree on the left side, then DFS will find it early.
- If the goal node lies near to the start node at the right of the tree, then BFS will find it early.

## *Time Complexity*

For the sake of comparison, assume that the goal node always occurs at depth $d$ in a tree of depth $d$. Let $L$ be on the left end and $R$ be on the right end of the tree at depth $d$.
We compute the time complexity at the two ends for both algorithms and take their average for the purpose of comparing DFS with BFS. Let $N_{DFS}$ be the number of nodes inspected by DFS and $N_{BFS}$ be the number of nodes inspected by BFS.
When the goal node is at $L$,

$$N_{DFS} = (d+1), \quad N_{BFS} = 1+b+b^2+\cdots+b^{d-1}+1 = (b^d-1)/(b-1)+1$$

When the goal node is at $R$,

$$N_{DFS} = N_{BFS} = (b^{d+1}-1)/(b-1).$$

## Time Complexity

On average,

$$N_{DFS} = ((d+1) + (b^{d+1} - 1)/(b-1))/2 \approx b^d/2 \text{ for large } b$$
$$N_{BFS} = ((b^d - 1)/(b-1) + 1 + (b^{d+1} - 1)/(b-1))/2$$
$$\approx b^d(b+1)/2(b-1) \text{ for large } b$$

Thus the ratio of the time taken by BFS and DFS is

$$N_{BFS}/N_{DFS} = (b+1)/(b-1) \text{ for large } b.$$

We observe that for large $b$, BFS has a slightly higher time complexity than DFS. But the important observation is that both are exponential.

## *Depth First Iterative Deepening (DFID)*

- BFS did better on the quality measure, guaranteeing the shortest path.
- DFS did better on space complexity, requiring only linear space to store *OPEN*.

We next look at an algorithm that gives us the best of these two worlds, with surprisingly low extra cost.

## *Depth First Iterative Deepening (DFID)*

- This algorithm is essentially DFS but with a bound on the depth it cannot go beyond.
- Depth First Iterative Deppening (DFID) function uses Depth bounded depth first search (DB-DFS) function.
- DB-DFS takes the depth bound as an argument, *depthBound*, and executes DFS only within the bound.

```
DFID(start)
   depthBound $\leftarrow$ 0
   while TRUE
     do DB-DFS(start, depthBound)
     depthBound $\leftarrow$ depthBound $+1$
```

## *Depth Bounded Depth First Search (DB-DFS)*

- DB-DFS is a modification to restrict DFS to a bound received as an argument. We extend the *nodePair* to store the depth value as a third parameter, even though we still call it a pair.

## Depth Bounded Depth First Search (DB-DFS)

```
DB-DFS($S$, depthBound)
  $OPEN \leftarrow (S, \textbf{null}, 0) : [\,]$
  $CLOSED \leftarrow [\,]$
  while $OPEN$ is not empty
    $nodePair \leftarrow$ head $OPEN$
    ($N$, _, depth) $\leftarrow nodePair$
    if GoalTest($N$) = TRUE
      return ReconstructPath($nodePair$, $CLOSED$)
    else $CLOSED \leftarrow nodePair : CLOSED$
      if depth < depthBound
        $children \leftarrow$ MoveGen($N$)
        $newNodes \leftarrow$ RemoveSeen($children$, $OPEN$, $CLOSED$)
        $newPairs \leftarrow$ MakePairs($newNodes$, $N$, depth+1)
        $OPEN \leftarrow newPairs$ ++ tail $OPEN$
      else $OPEN \leftarrow$ tail $OPEN$
  return empty list
```

## *Performance of DFID*

Space Complexity:

- Beneath the hood of DFID is DFS which, as we have seen earlier has linear space requirement. Ergo, DFID requires linear space.

Time Complexity:

- While it pushes into the state space layer by layer, like BFS, in reality it revisits the older nodes again and again repeatedly.
- For every fresh layer it explores, it does the extra work of revisiting all the earlier layers completely.
- Where BFS would have inspected $L$ nodes in the newest layer, DFID again explores, in addition, the entire $I$ internal nodes again.

## *Performance of DFID*

Time Complexity:

When BFS would have inspected $L$ nodes, DFID inspects $(L + I)$ nodes. Therefore,

$$N_{DFID}/N_{BFS} = (L + I)/L$$

Now, for a full tree with branching factor $b$, the following holds:

$$L = (b - 1)I + 1.$$

This gives us

$$N_{DFID}/N_{BFS} = (bI + 1)/((b - 1)I + 1) \approx b/(b - 1) \text{ for large } b.$$

Thus, the extra work done by DFID is not significant for large $b$.

## *Performance of DFID*

Quality of Solution:

- DFID may not find a shortest path. (Exercise!!)

Completeness:

- It will definitely terminate if the search space if finite.

Will DFID be able to find a shortest path if CLOSED is removed? (Exercise!!)

Thank You