# Week 2: State Space Search, DFS, BFS and DFID

Notes by S. Baskaran

Solving a problem by search is solving a problem by trial and error. Several real life problems can be modeled as a **state-space search** problem. But how?

1. Choose your problem and determine what constitutes a STATE (a symbolic representation of the state-of-existence).
2. Identify the START STATE and the GOAL STATE(S).
3. Identify the MOVES (single-step operations/actions/rules) that cause a STATE to change.
4. Write a function that takes a STATE and applies all possible MOVES to that STATE to produce a set of NEIGHBOURING STATES (exactly one move away from the input state). Such a function (state-transition function) is called MoveGen.

   MoveGen embodies all the single-step operations/actions/rules/moves possible in a given STATE. The output of MoveGen is a set of NEIGHBOURING STATES. MoveGen: STATE --> SET OF NEIGHBOURING STATES.

   From a graph theoretic perspective the state space is a graph, implicitly defined by a MoveGen function. Each state is a node in the graph, and each edge represents one move that leads to a neighbouring state.

   Generating the neighbours of a state and adding them as candidates for inspection is called “expanding a state”.

In state space search, a solution is found by **exploring** the state space with the help of a MoveGen function, i.e., expand the start state and expand every candidate until the goal state is found.

State spaces are used to represent two kinds of problems: configuration and planning problems.

1. In **configuration problems** the task is to find a goal state that satisfies some properties.
2. In **planning problems** the task is to find a path to a goal state. The sequence of moves in the path constitutes a plan.

State spaces have properties:

**Extent**: state spaces may be finite or infinite.

**Exponential Growth**: finite state spaces may be very very large — exponential and beyond in the number of states. And the **search space** (the number of candidate nodes in the "search" space) associated with a search algorithm may increase exponentially with depth.

**Branching Factor:** may be constant, bounded, or finite. Branching factor refers to the maximum neighbours a node may have. Note: we explicitly discard infinite branching state spaces.

**Reversible**: some state spaces are reversible, i.e., EVERY MOVE (single step operation/action) is reversible. Most real world problems do not have this property, but contain regions that are reversible, and regions that are not.

**Connectedness**: The whole state space may be completely connected, or may have two or more connected components which are mutually disjoint. A search algorithm can only explore the connected component in which the start node lies. The 8-puzzle, for example, has two disjoint connected components, and the Rubik's cube has twelve.

**Edge Costs:** The moves in a state space may have costs associated with them, stored as edge costs. The sum of the edge costs in a plan determines the quality of the solution. In the absence of edge costs, the number of moves in a plan may be used as a quality measure.

**Metric Spaces**: States (and/or transitions) may provide a metric (say Euclidean distance, Manhattan distance, etc.) as an estimate of distance to a goal state, etc. This estimate should ideally be correlated to the actual cost of the solution starting from that node.

**Reachable Subspace:** Given a state space (via a MoveGen) and a start state, all the states reachable via zero, one or more moves from the start state constitutes a subspace, a reachable subspace of the start node.

## Algorithms: DFS, BFS

Use the DFS and BFS versions given below to solve the assignments. These algorithms will be used in the final exam as well. Observe that a node is added to OPEN only if it is not already present in OPEN or CLOSED.

```
DFS(S)
 1  OPEN ← (S, null) : []
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4      nodePair ← head OPEN
 5      (N, _) ← nodePair
 6      if GoalTest(N) = TRUE
 7          return ReconstructPath(nodePair, CLOSED)
 8      else CLOSED ← nodePair : CLOSED
 9          neighbours ← MoveGen(N)
10          newNodes ← RemoveSeen(neighbours, OPEN, CLOSED)
11          newPairs ← MakePairs(newNodes, N)
12          OPEN ← newPairs ++ (tail OPEN)
13  return empty list
```

```
BFS(S)
 1  OPEN ← (S, null) : []
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4      nodePair ← head OPEN
 5      (N, _) ← nodePair
 6      if GoalTest(N) = TRUE
 7          return ReconstructPath(nodePair, CLOSED)
 8      else CLOSED ← nodePair : CLOSED
 9          neighbours ← MoveGen(N)
10          newNodes ← RemoveSeen(neighbours, OPEN, CLOSED)
11          newPairs ← MakePairs(newNodes, N)
12          OPEN ← (tail OPEN) ++ newPairs
13  return empty list
```

**Note:** DFS and BFS are identical except for Line 12.

## Ancillary Functions

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

```
RemoveSeen(nodeList, OPEN, CLOSED)
1   if nodeList is empty
2       return empty list

3   else node ← head nodeList
4       if OccursIn(node, OPEN) or OccursIn(node, CLOSED)
5           return RemoveSeen(tail nodeList, OPEN, CLOSED)
6       else return node : RemoveSeen(tail nodeList, OPEN, CLOSED)
```

```
ReconstructPath(nodePair, CLOSED)
 1  SkipTo(parent, nodePairs)
 2      if parent = first head nodePairs
 3          return nodePairs
 4      else return SkipTo(parent, tail nodePairs)

 5  (node, parent) ← nodePair
 6  path ← node : []
 7  while parent is not null
 8      path ← parent : path
 9      CLOSED ← SkipTo(parent, CLOSED)
10      (_, parent) ← head CLOSED
11  return path
```

## Algorithm DFID-1

DFID-1 opens only new nodes (nodes not already present in OPEN/CLOSED), and does not reopen any nodes.

```
DFID-1(S)
count ← $-1$
path ← **empty list**
depthBound ← 0
**repeat**
    previousCount ← count
    (count, path) ← DB-DFS-1(S, depthBound)
    depthBound ← depthBound + 1
**until** (path **is not empty**) **or** (previousCount = count)
**return** path
```

```
DB-DFS-1(S, depthBound)
    ▷ Opens only new nodes, i.e., nodes neither in OPEN nor in CLOSED,
    ▷ and does not reopen any nodes.
count ← 0
OPEN ← (S, **null**, 0) : []
CLOSED ← **empty list**
**while** OPEN **is not empty**
    nodePair ← **head** OPEN
    (N, _, depth) ← nodePair
    **if** GoalTest(N) = TRUE
        **return** (count, ReconstructPath(nodePair, CLOSED))
    **else** CLOSED ← nodePair : CLOSED
        **if** depth < depthBound
            neighbours ← MoveGen(N)
            newNodes ← RemoveSeen(neighbours, OPEN, CLOSED)
            newPairs ← MakePairs(newNodes, N, depth + 1)
            OPEN ← newPairs ++ **tail** OPEN
            count ← count + **length** newPairs
        **else** OPEN ← **tail** OPEN
**return** (count, **empty list**)
```

## Algorithm DFID-2

DFID-2 opens new nodes (nodes not already present in OPEN/CLOSED) and **also reopens nodes** present in CLOSED but not present in OPEN.

```
DFID-2(S)
1   count ← −1
2   path ← **empty list**
3   depthBound ← 0
4   **repeat**
5       previousCount ← count
6       (count, path) ← DB-DFS-2(S, depthBound)
7       depthBound ← depthBound + 1
8   **until** (path **is not empty**) **or** (previousCount = count)
9   **return** path
```

```
DB-DFS-2(S, depthBound)
    ▷ Opens new nodes, i.e., nodes neither in OPEN nor in CLOSED,
    ▷ and reopens nodes present in CLOSED and not present in OPEN.
10  count ← 0
11  OPEN ← (S, **null**, 0) : []
12  CLOSED ← **empty list**
13  **while** OPEN **is not empty**
14      nodePair ← **head** OPEN
15      (N, _, depth) ← nodePair
16      **if** GOALTEST(N) = TRUE
17          **return** (count, RECONSTRUCTPATH(nodePair, CLOSED))
18      **else** CLOSED ← nodePair : CLOSED
19          **if** depth < depthBound
20              neighbours ← MOVEGEN(N)
21              newNodes ← REMOVESEEN(neighbours, OPEN, [])
22              newPairs ← MAKEPAIRS(newNodes, N, depth + 1)
23              OPEN ← newPairs ++ **tail** OPEN
24              count ← count + **length** newPairs
25          **else** OPEN ← **tail** OPEN
26  **return** (count, **empty list**)
```

**Note:** DFID-1 and DFID-2 are identical except for Line 21 in DB-DFS-* procedures.

## Ancillary Functions for DFID-2

```
MAKEPAIRS(nodeList, parent, depth)
1   if nodeList is empty
2       return empty list
3   else nodePair ← (head nodeList, parent, depth)
4       return nodePair : MAKEPAIRS(tail nodeList, parent, depth)
```

```
RECONSTRUCTPATH(nodePair, CLOSED)
 1  SKIPTO(parent, nodePairs, depth)
 2      if (parent, _, depth) = head nodePairs
 3          return nodePairs
 4      else return SKIPTO(parent, tail nodePairs, depth)

 5  (node, parent, depth) ← nodePair
 6  path ← node : []
 7  while parent is not null
 8      path ← parent : path
 9      CLOSED ← SKIPTO(parent, CLOSED, depth – 1)
10      (_, parent, depth) ← head CLOSED
11  return path
```

## Node Order (IMPORTANT)

**Participants who are planning to take the final exam please pay attention to node order and tie-breaking rules.**

For each algorithm discussed in this course, study the order in which nodes are added-to and removed-from OPEN/CLOSED lists and other data structures.

It is important to understand how list structures are constructed, accessed and printed. Study the ‘cons’ and ‘append’ operators and pay attention to the order in which items are cons-ed (added) and appended to lists.

**WARNING:** while answering auto-graded short-answer type questions, it is important to maintain the algorithm specific node order, otherwise, **the auto-grader will mark the answer as wrong** and there are no partial marks.

**WARNING:** often, participants understand the concepts correctly and they may follow a different variation of an algorithm or follow a different implementation of cons and append operators and arrive at a practically viable solution but with a different node order, in this case **the auto-grader will mark the answer as wrong**.

## Tie Breaking (IMPORTANT)

When node order is not crystal clear (because multiple nodes simultaneously qualify) then follow the tie-breaking rule stated in the question, if no rule is stated, then break the tie based on node label.

When multiple nodes simultaneously qualify for inspection (which is possible in cost based algorithms when two or more nodes have the same optimal cost) then sort the qualified nodes in the alphabetical order of the node labels and select nodes from the head of the sorted list.

In some cases, the node selected by a tie-breaker may not lead to a solution, but another qualified node may. For the purpose of auto-grading we will stick to the node returned by the tie-breaker.

In real world problems, finding a solution is important, so we may try all possible orderings of the qualified nodes. But for a MOOC (Massive Open Online Course) a simple tie-breaking rule simplifies the auto-grading process.