Week 2
State Space Search, DFS, BFS and DFID

Notes by Baskaran Sankaranarayanan

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 an input STATE and applies all possible MOVES to the input STATE to produce a set of NEIGHBOURING STATES (that are 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 the set of NEIGHBOURING STATES. MoveGen: STATE --> SET OF NEIGHBOURING STATES.

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

In state space search, a solution is found by **exploring** the state space with the help of the MoveGen function: Begin by inspecting the start state, and keep generating candidate states until a goal state is found. Generating neighbours of a state and adding them as candidates is called expanding the state.

State spaces are used to represent two kinds of problems. In **configuration problems** the task is to find a goal state that satisfies some properties. 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. 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, finite, or large-and-finite. Branching factor refers to the maximum neighbours a node may have.

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 have regions that are reversible, and regions that are not.

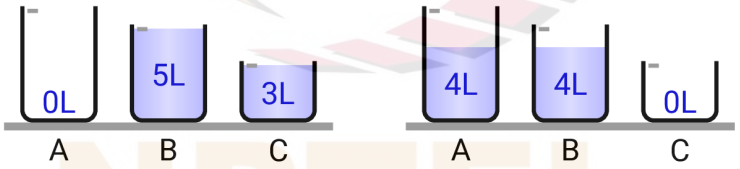
Connectedness: The whole state space may be completely connected, or may have several connected components which are mutually disjoint. A search algorithm can only explore the connected component in which the start node lies. The 8-puzzle, for example, has two disjoint connected components, and the Rubik's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

Edge costs: The moves in the 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 a quality measure used.

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

Water Jug Puzzle

There are 3 jugs A, B and C of capacities 6L, 5L and 3L, respectively. Jug A is empty and jugs B and C are filled with water as shown in the initial-state in the figure below. Only two types of actions/operations/moves are permitted: empty a jug into another jug, or fill a jug to its brim. Use these moves (in any order and any number of times) to divide the water into two equal halves as shown in the goal state in the figure below.



Model this puzzle as a state-space search problem. A state is represented by a tuple (x, y, z), where x, y and z are the amounts of water in jugs A, B and C, respectively. In every state, x+y+z = 8L.

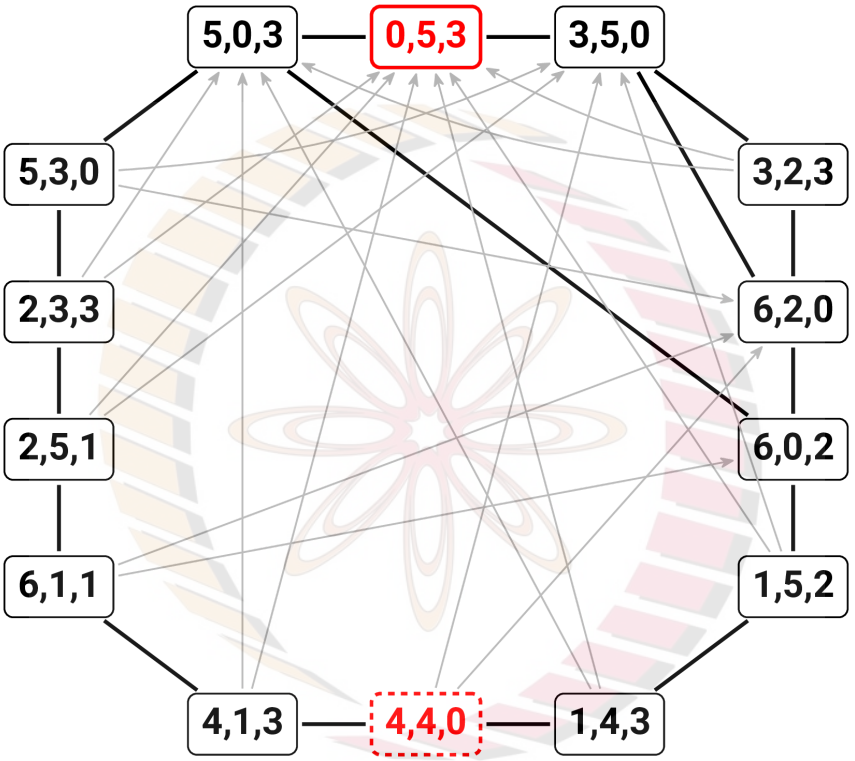
The initial state is (0, 5, 3), if we transfer water from B to A, we reach (5, 0, 3), from this state if we transfer water from C to A, we reach (6, 0, 2). So, we have made two moves: (0, 5, 3) --> (5, 0, 3) --> (6, 0, 2). And these two moves are reversible: (6, 0, 2) --> (5, 0, 3) --> (0, 5, 3).

A state-space expresses all valid states and their transitions. Now, start from the initial state (0, 5, 3) and build the state-space for the water jug puzzle to answer the questions in this week's assignment Group 1.

Reachable Subspace: in this assignment, to reduce manual effort, we restrict the state space to contain only those states that are reachable from (0, 5, 3) via zero, one or more moves. For example, from (2, 4, 2) we can reach (0, 5, 3) in two moves, but starting from (0, 5, 3) we cannot reach (2, 4, 2) and hence it is excluded from the state space.

Reachable Subspace of Water Jug Puzzle

The subspace reachable from (0, 5, 3) is shown below. The edges without arrowheads indicate two way transitions and the edges with arrowheads indicate one way transitions.



MUST FOLLOW:

For all algorithms presented in this course, study the order in which nodes are added to and removed from OPEN and CLOSED lists (and other lists as well). The solution to questions depends on this order. When the node order is ambiguous, follow the tie-breaking rule stated in the question.

Also understand how lists (used in the algorithms) are constructed, accessed and printed. Understand the cons and append operator, pay attention to the order in which the nodes are cons-ed (added) and appended to lists.

WARNING: Often, participants understand the concepts correctly and sometimes they get the node order wrong (**and get zero marks**) because they follow a different variation of an algorithm and/or follow different implementations of cons and append operations.

Algorithms DFS and BFS

Use the DFS and BFS versions given below to answer questions in 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
```

Algorithm DFID-1

DFID-1 opens only new nodes (nodes not already present in OPEN or CLOSED). And does not reopen any nodes.

```
DFID-1(S)
1  count ← −1
2  path ← empty list
3  depthBound ← 0
4  repeat
5       previousCount ← count
6       (count, path) ← DB-DFS-1(S, depthBound)
7       depthBound ← depthBound + 1
8  until (path is not empty) or (previousCount = count)
9  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.
 1  count ← 0
 2  OPEN ← (S, null, 0) : []
 3  CLOSED ← empty list
 4  while OPEN is not empty
 5       nodePair ← head OPEN
 6       (N, _, depth) ← nodePair
 7       if GoalTest(N) = true
 8            return (count, ReconstructPath(nodePair, CLOSED))
 9       else CLOSED ← nodePair : CLOSED
10            if depth < depthBound
11                 neighbours ← MoveGen(N)
12                 newNodes ← RemoveSeen(neighbours, OPEN, CLOSED)
13                 newPairs ← MakePairs(newNodes, N, depth + 1)
14                 OPEN ← newPairs ++ tail OPEN
15                 count ← count + length newPairs
16            else OPEN ← tail OPEN
17  return (count, empty list)
```

Algorithm DFID-2

DFID-2 opens new nodes (nodes not already present in OPEN or CLOSED) and *also* reopens nodes present in CLOSED and 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.
 1  count ← 0
 2  OPEN ← (S, null, 0) : []
 3  CLOSED ← empty list
 4  while OPEN is not empty
 5       nodePair ← head OPEN
 6       (N, _, depth) ← nodePair
 7       if GoalTest(N) = true
 8            return (count, ReconstructPath(nodePair, CLOSED))
 9       else CLOSED ← nodePair : CLOSED
10            if depth < depthBound
11                 neighbours ← MoveGen(N)
12                 newNodes ← RemoveSeen(neighbours, OPEN, [])
13                 newPairs ← MakePairs(newNodes, N, depth + 1)
14                 OPEN ← newPairs ++ tail OPEN
15                 count ← count + length newPairs
16            else OPEN ← tail OPEN
17  return (count, empty list)
```