AI SMPS 2023 Week 2 Algorithms

Prepared by S. Baskaran

Node Order is 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**.

State Space Search

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.

   The MoveGen function embodies all the single-step operations/actions/rules/moves that are possible in a given STATE. The output of MoveGen is a set of NEIGHBOURING STATES.

   MoveGen: STATE $\rightarrow$ 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 neighbour 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 number of neighbours a node can 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 reversible.

Connectedness: the whole state space may be fully connected or may have two or more connected components. A search algorithm can only explore the connected component in which the start node lives. 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 is a good measure of quality.

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. This estimate should ideally be correlated to the actual cost of the solution.

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

Algorithms: DFS, BFS

Use this version of DFS (Depth First Search) and BFS (Breadth First Search) to solve the assignment problems. **These will be used in the final exam.**

Observe that a node is added to OPEN only if it is not already present in OPEN or CLOSED.

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

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
```

DB-DFS(S, depthBound)

```
1  OPEN ← (S, null, 0) : []
2  CLOSED ← empty list
3  while OPEN is not empty
4      nodePair ← head OPEN
5      (N, _, depth) ← nodePair
6      if Goal-Test(N) = true
7          return Reconstruct-Path(nodePair, CLOSED)
8      else CLOSED ← nodePair : CLOSED
9          if depth < depthBound
10             neighbours ← Move-Gen(N)
11             newNodes ← Remove-Seen(neighbours, OPEN, CLOSED)
12             newPairs ← Make-Pairs(newNodes, N, depth + 1)
13             OPEN ← newPairs ++ tail OPEN
14         else OPEN ← tail OPEN
15 return empty list
```

Make-Pairs(nodeList, parent)

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

Remove-Seen(nodeList, OPEN, CLOSED)

```
1  if nodeList is empty
2      return empty list
3  else node ← head nodeList
4      if Occurs-In(node, OPEN) or Occurs-In(node, CLOSED)
5          return Remove-Seen(tail nodeList, OPEN, CLOSED)
6      else return node : Remove-Seen(tail nodeList, OPEN, CLOSED)
```

Occurs-In(node, nodePairs)

```
1  if nodePairs is empty
2      return FALSE
3  else if node = first head nodePairs
4      return TRUE
5  else return Occurs-In(node, tail nodePairs)
```

Reconstruct-Path(nodePair, CLOSED)

```
1  Skip-To(parent, nodePairs)
2      if parent = first head nodePairs
3          return nodePairs
4      else return Skip-To(parent, tail nodePairs)

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

Algorithm: DB-DFS

Use this version of DB-DFS (Depth-Bound Depth First Search) to solve the assignment problems. **This algorithm will be used in the final exam.**

Observe that a node is added to OPEN only if it is not already present in OPEN or CLOSED.

Use the ancillary functions given for DFS/BFS, modify them to handle triples.

```
DB-DFS(S, depthBound)
 1  OPEN ← (S, null, 0) : []
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4      nodePair ← head OPEN
 5      (N, _, depth) ← nodePair
 6      if GOAL-TEST(N) = TRUE
 7          return RECONSTRUCT-PATH(nodePair, CLOSED)
 8      else CLOSED ← nodePair : CLOSED
 9          if depth < depthBound
10              neighbours ← MOVE-GEN(N)
11              newNodes ← REMOVE-SEEN(neighbours, OPEN, CLOSED)
12              newPairs ← MAKE-PAIRS(newNodes, N, depth + 1)
13              OPEN ← newPairs ++ tail OPEN
14          else OPEN ← tail OPEN
15  return empty list
```

Algorithm: DFID-N

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

Note: DFID-N and DFID-C are identical except for Line 21 in the DB-DFS-N/C procedures.

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


DB-DFS-N(S, depthBound)
   ▷ Opens only new nodes, i.e., nodes neither in OPEN nor in CLOSED,
   ▷ and does not reopen any nodes.
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, CLOSED )
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)
```

Algorithm: DFID-C

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

Note: DFID-N and DFID-C are identical except for Line 21 in the DB-DFS-N/C procedures.

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


DB-DFS-C(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)
```

Ancillary Functions for DFID-N/C

```
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
```