COSC367: Artificial Intelligence

This course introduces major concepts and algorithms in Artificial Intelligence. Topics include problem solving, reasoning, games, and machine learning.

Jordan Pyott

2021-05-06 16:57

Contents

Artificial Intelligence

Course Information

The course covers core topics in AI including:

- uninformed and informed graph search algorithms,
- propositional logic and forward and backward chaining algorithms,
- declarative programming with Prolog,
- the min-max and alpha-beta pruning algorithms,
- Bayesian networks and probabilistic inference algorithms,
- classification learning algorithms,
- consistency algorithms,
- local search and heuristic algorithms such as simulated annealing, and population-based algorithms such as genetic search and swarm optimisation.

Grades

Standard Computer science policy applies

- Average 50% over all assessment items
- Average at least 45% on all invigilated assessment items

Grading structure for course

- Assignments (5%)
  - Two Super Quiz's
- Quizzes (16.5%)
  - Weekly Quiz Assessments (1.5% ea)
- Lab Test (20%)
- Final Exam (58.5%)

Textbooks / Resources

- Poole, David L.1958, Mackworth, Alan K; Artificial intelligence : foundations of computational agents; Cambridge University Press, 2010.
- Russell, Stuart J, Norvig, Peter; Artificial intelligence : a modern approach; 3rd ed; Prentice Hall, 2010.

Readings

Lectures

Searching the State Space

What is state?

- A state is a data structure that represents a possible configuration of the world *agent and environment*
- The **state space** is the set of all possible states for that problem
- actions change the state of the world

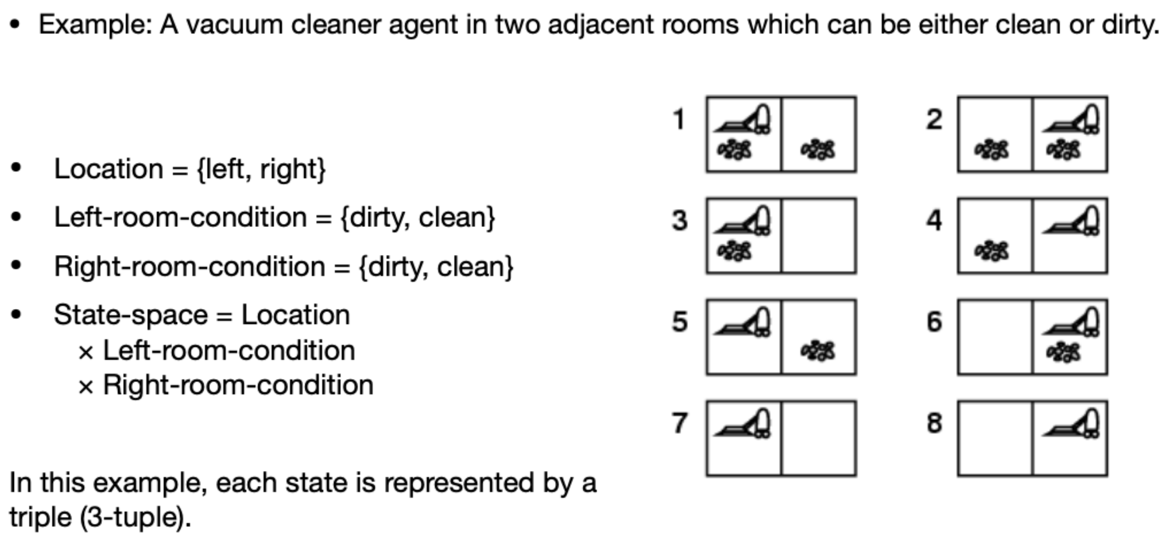

Figure 1: State space example one

State can also be represented as a graph *both directed and undirected*

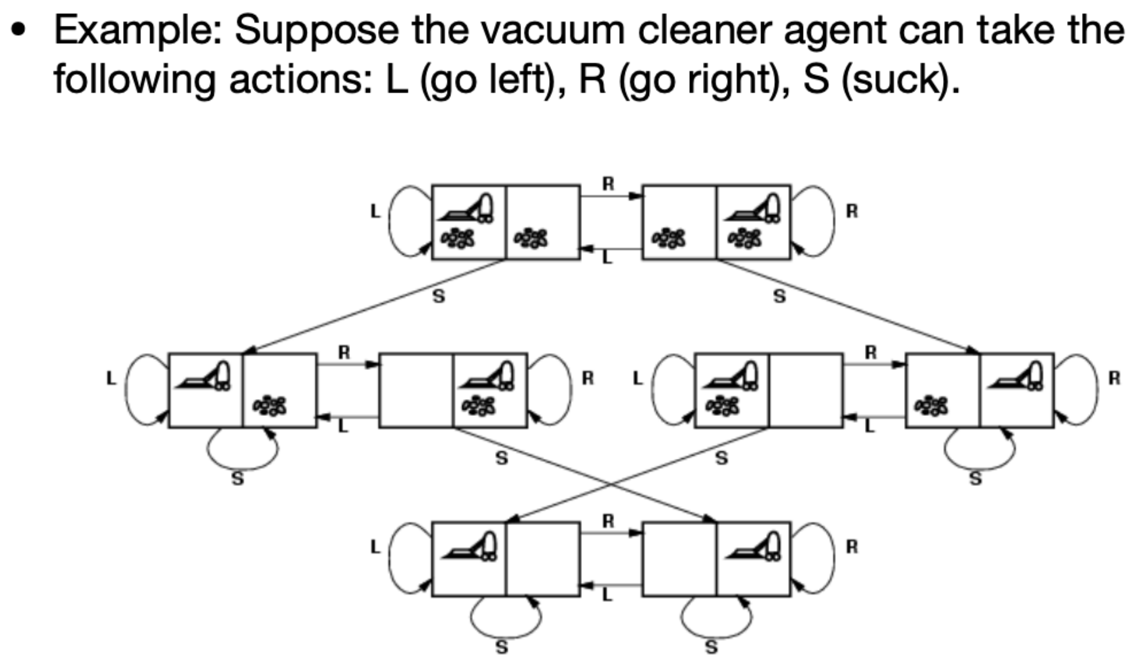

Figure 2: State space graph simplified

- Many problems in AI can be abstracted to the problem of finding a path in a directed graph
- Notation we use is **Nodes** and **arcs** for **vertices** and **edges** in a graph

Explicit vs Implicit graphs

- In **explicit graphs** nodes and arcs are readily available, they are read from the input and stored in a data structure such as an adjacency list/matrix.
  - the entire graph is in memory.
  - the complexity of algorithms are measured in the number of nodes and/or arcs.
- In **implicit graphs** a procedure `outgoing_arcs` is defined that given a node, returns a set of directed arcs that connect node to other nodes.
  - The graph is generated as needed *due to the complexity of the graphs.*
  - The complexity is measured in terms of the depth of the goal state node or *how far do we have to get into the graph to find a solution*.

Explicit graphs in quizzes

- In some exercises we use small explicit graphs to stydy the behaviour of various frontiers
- Nodes are specified in a set

- Edges are specified in a list
    - pairs of nodes, or triples of nodes (in a tuple)

Searching graphs

- We will use generic search algorithms: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a **frontier** of paths that have been explored
    - frontier: paths that we have already explored
- As search proceeds, the frontier is updated and the graph is explored until a goal node is found.
- The order in which paths are removed and added to the frontier defines the search strategy
- A **search tree** is a tree drawn out of all the possible actions in terms of a tree.
    - How do we handle loops? *Covered in next lecture*
    - In the search tree outlined below, you can see that the *end of paths on frontier* represents a BFS relationship note this is not always the case.

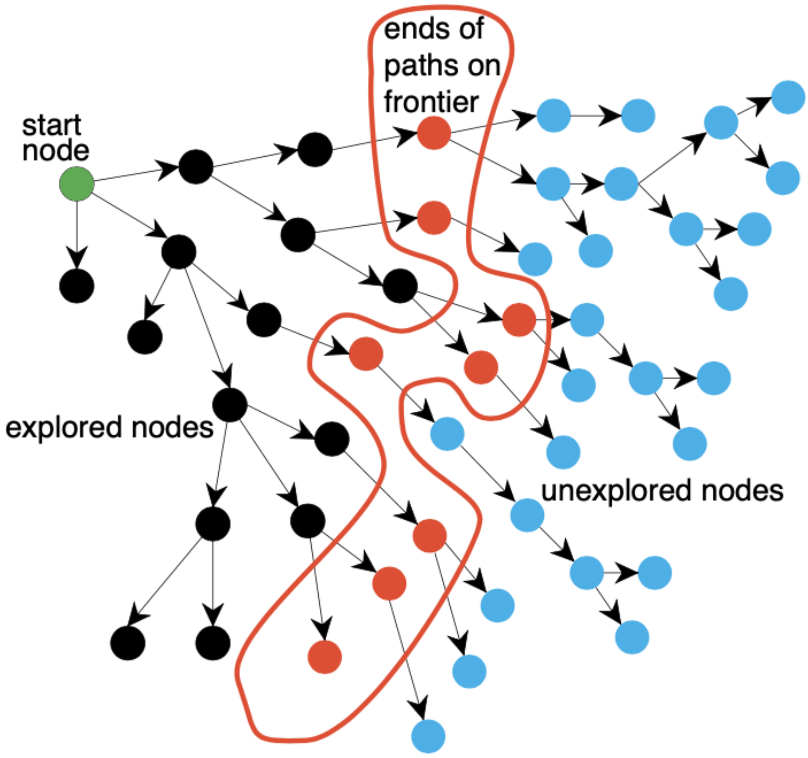

Figure 3: search tree

Generic graph search algorithm

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
  if $goal(n_k)$
   return $\langle n_0, \ldots, n_k \rangle$;
  for every neighbor n of n_k
   add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while

Figure 4: Generic Search

> NOTE: you will have to use what ever data structure for the seach you are using (BFS use a queue), (DFS use a stack).

In the generic algorithm, neighbours are going to use the method `outgoing_arcs`, we are given this algorithm in the form of a python module.

Depth-first search

- In order to perform DFS, the generic graph search must be used with a stack frontier *LIFO*
- If the stack is a python list, where each element is a path, and has the form [..., p, q]
  - *q* is selected and popped
  - of the algorithm continues then paths that extend *q* are pushed (appended) to the stack
  - *p* is only selected when all paths from *q* have been explored.
- As a result, at each stage the algorithm expands the deepest path
- The orange nodes in the graph below are considered the frontier nodes

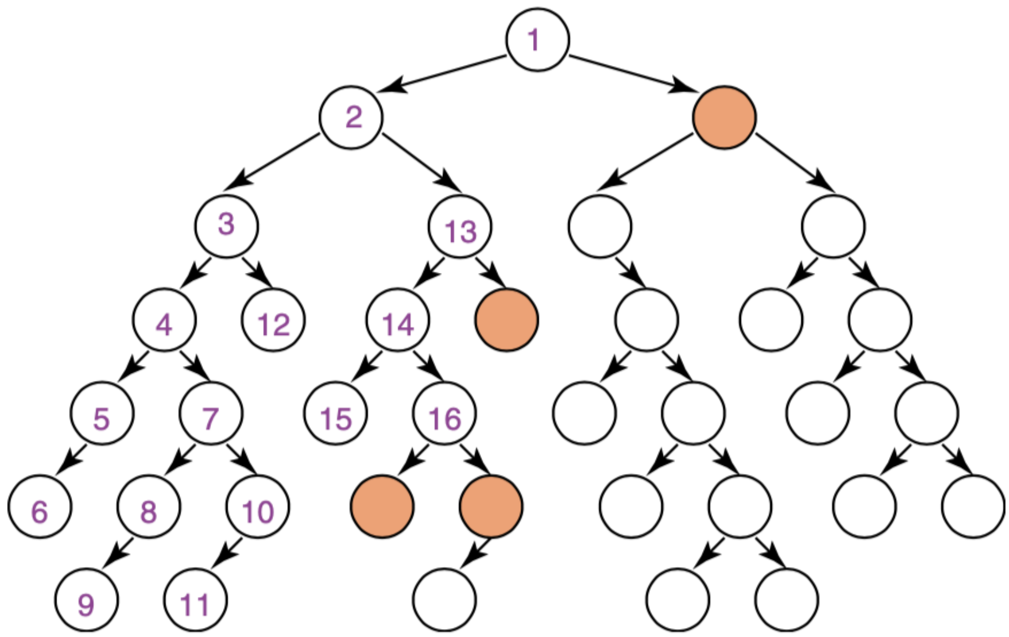

Figure 5: DFS

- DFS does not guarantee a solution without pruning, due to the fact that we can have infinite loops
- It is not guaranteed to complete if it does not use pruning

A note on complexity

Assume a finite search tree of depth *d* and branching factor of *b*:

- What is the time complexity?
  - It will be exponential: $O(b^d)$
- What is the space complexity?
  - It will be linear: $O(bd)$

How do we trace the frontier

- starting with an empty frontier we record all the calls to the frontier: to add or get a path we dedicate one line per call
- When we ask the frontier to add a path, we start the line with a + followed by the path that has been added
- When we ask for a path from the frontier we start the line with a − followed by the path being removed

- When using a priority queue, the path is followed by a comma and then the key *e.g, cost, heuristic, f-value, …*
- The lines of the trace should match the following regular expression `^[+-][a-z]+(,\d+)?!?$`
- We stop when we **remove** a path from the trace

Given the following graph

nodes={a, b, c, d},
edge_list=[(a,b), (a,d), (a, c), (c, d)],
starting_nodes = [a],
goal_nodes = {d}

trace the frontier in depth-first search (DFS).

Answer:

```
+ a
- a
+ ab
+ ad
+ ac
- ac
+ acd
- acd
```

Figure 6: DFS trace using generic algorithm

Breath-first search

- In order to perform BFS, the generic graph search must be used with a queue frontier *FIFO*.
- If the queue is a python deque of the form [p,q,…,r], then
  - p is selected (dequeued)
  - if the algorithm continues then paths that extend *p* are enqueued *appended* to the queue after *r*
- As a result, at each state the algorithm expands the shallowest path.

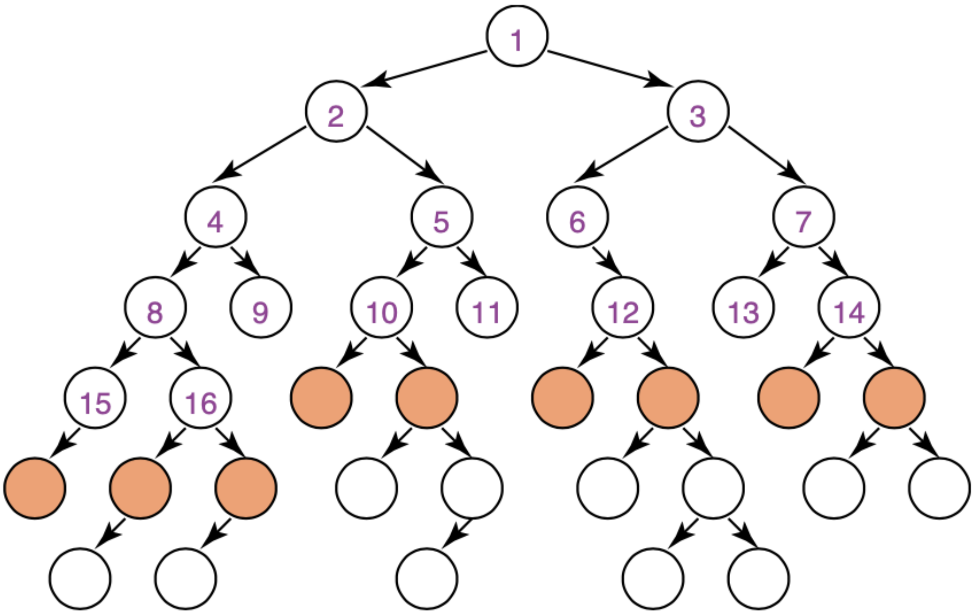

Figure 7: BFS Illustration of search tree

- BFS **does** guarantee to find a solution with the fewest arcs if there is a solution
- It will complete
- It will not halt due to some graphs having *cycles, with no pruning*

A note on complexity

> BFS has higher complexity than DFS

- What is the time complexity?
  - It will be exponential: $O(b^d)$
- What is the space complexity?
  - It will be linear: $O(b^d)$

Given the following graph

nodes={a, b, c, d},
edge_list=[(a,b), (a,d), (a, c), (c, d)],
starting_nodes = [a],
goal_nodes = {d}

trace the frontier in breadth-first search (BFS).

Answer:

```
+ a
- a
+ ab
+ ad
+ ac
- ab
- ad
```

Figure 8: BFS trace using generic algorithm

Lowest-cost-first search

- The cost of a path is the sum of the costs of its arcs
- This algorithm is very similar to Dijkstra's except modified for larger graphs
- LCFS selects a path on the frontier with the lowest cost
- The frontier is a priority queue ordered by path cost
  - A priority queue is a container in which each element has a priority *cost*
  - An element with a higher priority is always selected/removed before an element with a lower priority
  - In python we can use the `heapq` you will need to store objects in a way that these properties hold
- LCFS finds an optimal solution: a least-cost path to a goal node.
- Another name for this algorithm is *uniform-cost search*.

> NOTE: For an example of this queue, see Lecture One: 1:45 time stamp

Given the following graph

nodes={a, b, c, d, g},
edge_lists=[(a,b,4), (a,c,2), (a,d,1),
(b,g,4), (c,g,2), (d,g,4)],
starting_nodes = [a],
goal_nodes = {g}

trace the frontier in lowest-cost-first search (LCFS).

Answer:

```
+ a, 0
- a, 0
+ ab, 4
+ ac, 2
+ ad, 1
- ad, 1
+ adg, 5
- ac, 2
+ acg, 4
- ab, 4
+ abg, 8
- acg, 4
```

Figure 9: LCFS trace generic