Computer science course notes (/csci)

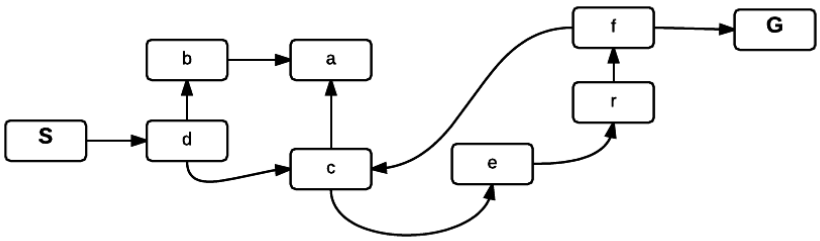
Home

Lecture 3 - Searching for a plan (/csci/index.php/9-csci-3202-lecture-notes/25-lecture-3-searching-for-a-plan)

Details
Written by Rhonda Hoenigman
Published: 26 August 2015
Hits: 474

State space graph

Once the state space is defined, the transitions between states can be represented using a state-space graph. This only works on small problems, or sub-sets of larger problems, where the transitions between states can be drawn. In the following directed graph, the vertices are the states and the arrows represent the successor function. Details of the successor function are not specified.



Question: Find a path from state S to state G.

There is only one option:

S->d->c->e->r->f->G

Searching for the path through the graph

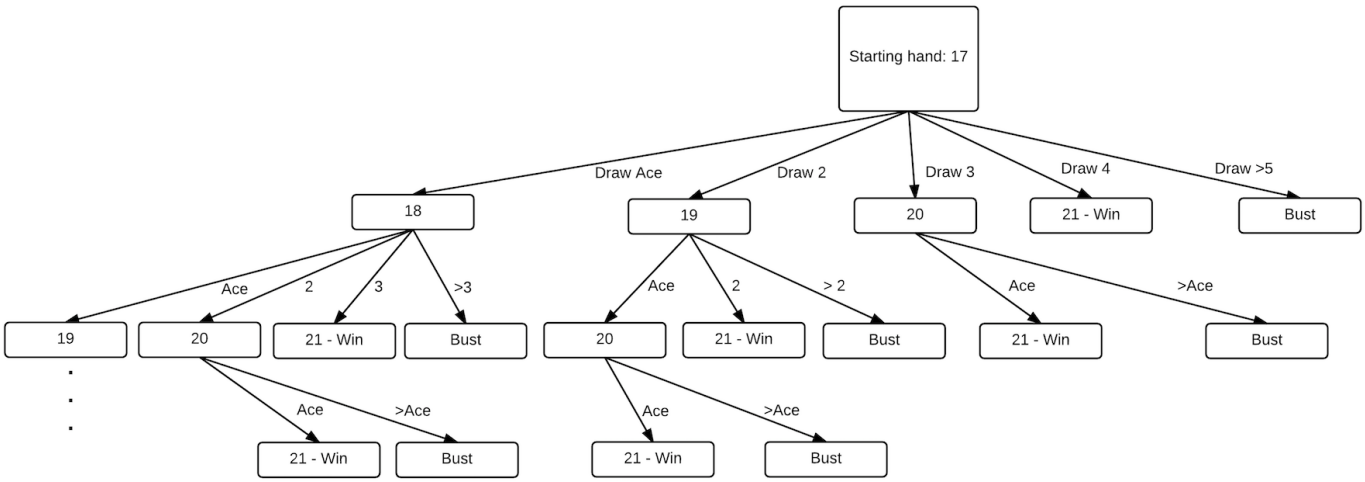
Search algorithms find paths through a graph without having to build the graph first. The states can be represented in a tree, where the current state is a node in the tree, and the next states are children of that node.

Top of the tree, aka the root of the tree, is the start state or current state.

The children of a node are the states that can be reached from that state by one action, or one application of the successor function.

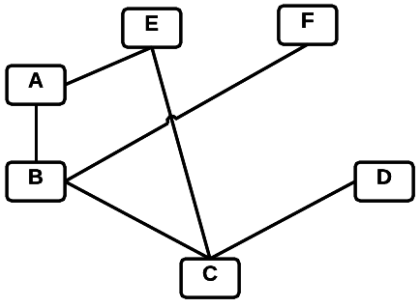
The children's children are two actions from a node, or two applications of a successor function.

The search tree grows exponentially as you move down the tree. Consider, for example, the tree for a move in the game blackjack.

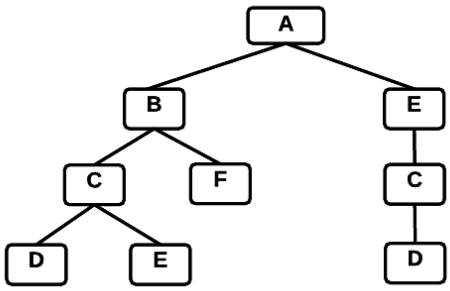


The starting value for the hand is 17, which is given as the root node. The children of the root node show all possible next hands, or states, that can be achieved by a successor function of drawing another card. Two steps from the root are the hands that result from drawing two cards, and so on. Each node in this tree represented a plan for drawing cards and the outcome from that plan. A node that is two steps from the root includes the plan for the first card drawn and the second card drawn.

Example: In the following graph, starting from A, where can you go in 1, 2, 3 steps? Is there a path from A to D?



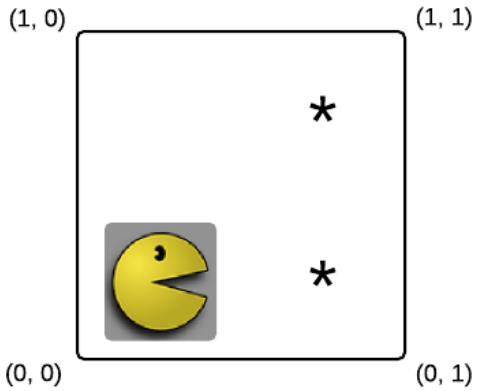
The vertex A becomes the root of the tree, and the vertices connected to A become its children. We can build this tree, starting from vertex A. Nodes are added to the tree if they haven't been visited on the current path.



Where can we go from A in one step? We can determine this by looking at A's children. To know where we can get in two steps, we look at A's children's children. We're only adding nodes to the tree that are not part of the current path.

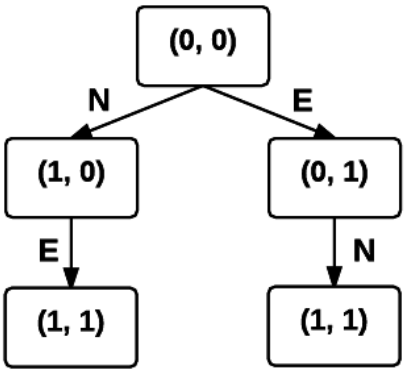
Example: Pacman path planning

Consider a start state that looks like the following:



Pacman is at (0, 0). Assume you want to find a path for pacman to (1,1). The start state is (0, 0) and a successor function that calculates the next state given pacman's direction would show that there are two next states: (0, 1) if pacman goes E and (1, 0) if pacman goes N. From each of these states, pacman goes from (1, 0) to (1, 1) by going E and pacman goes from (0, 1) by going N.

If we build a tree from the start state to the end state, it looks like:



Each of the nodes in the tree is a state.

Each of the paths from the start state (shown as the root of the tree) to a leaf is a solution. In this example, there are two solutions that can be shown by listing the actions taken to move between states:

N->E and E->N

The solution can also include the nodes in the tree, where the node can include the state and action information.

Notice that the solution is the action that moves the agent between states. In this example, both solutions are equally good. But, with a more complicated problem, there would be some paths that are shorter than others.

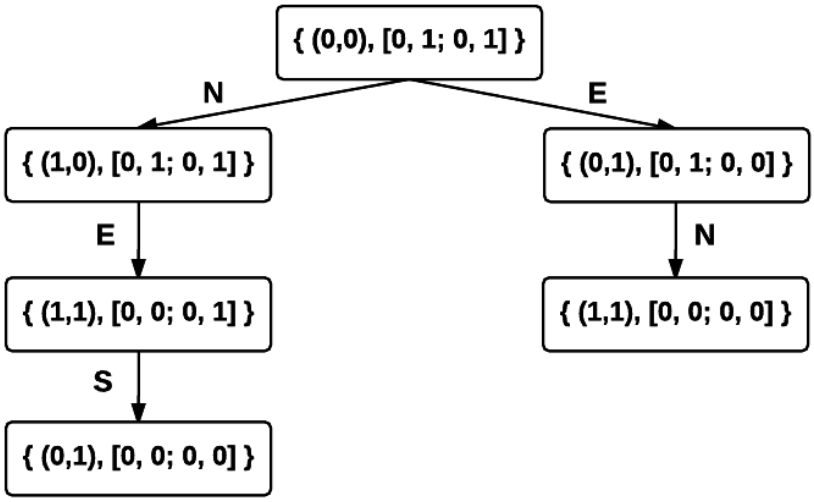
Example: Pacman eat all of the dots

State: (x,y) location and matrix of dots remaining

Next state: current location + step in NSEW direction and matrix of dots remaining after movement

Goal test: All dots gone

This example also uses the location, but also keeps track of how many dots are remaining. There is still nothing intelligent about how dots are consumed. The successor function moves pacman one step from its current location in a specified direction, and also updates the dot map. Starting from (0,0) in the simple example shown above, the state sequences generated by the successor function would look like:



Neither of these Pacman problems include information about the positions of the ghosts. However, if the goal also included staying alive, and the ghosts could attack, then ghost position would also need to be included and the size of the state space would increase.

Search strategy

How the tree is traversed, i.e. which states are evaluated and when, to find a solution is set by a search strategy. There are four criteria for evaluating search strategies:

1. Completeness: Is the strategy guaranteed to find a solution if there is one?

2. Time Complexity: How long does it take to find a solution?

3. Space Complexity: How much memory does the strategy need?

4. Optimality: Does the strategy find the highest quality solution when there are several available?

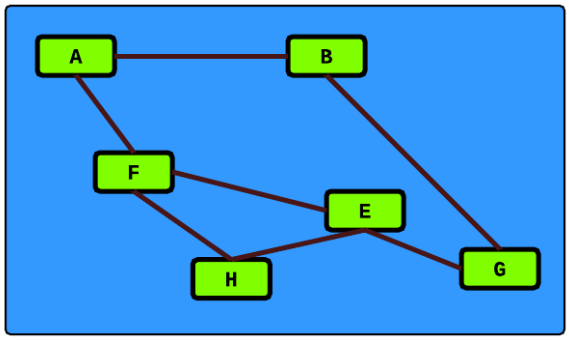
The first two search strategies we'll discuss are considered uninformed. We have no additional information than the current state and the adjacent states.

Breadth-first search

Breadth-first search (BFS) is a search algorithm that identifies solutions by evaluating the vertices in at one level in a graph before going deeper into the graph. If a graph is represented as a tree, the search order of the nodes in the tree is as follows:

BFS is used to find the shortest path in an unweighted graph. In the following BFS algorithm, a queue is used to keep track of the vertices in the graph. When a vertex is visited, its children are enqueued. They can then be dequeued and processed, which involves enqueueing their children. This continues until there are no vertices left to evaluate and the queue is empty. The BFS algorithm uses a *visited* property to keep track of whether a node has already been evaluated.

Example: Imagine that the following image shows a bunch of small islands connected by bridges. Starting from A, how many vertices (islands) are visited to reach G, and in what order?



Starting from A, we add A to the queue. Once it's added, the queue isn't empty, so we dequeue A and loop through each of A's children, which are B and F. If the vertices haven't been visited, we mark each as visited and add 1 to the distance to reach that node, and then check if it's the goal state. The next vertex to dequeue is B, and we then enqueue its unvisited child G. The distance to G is the distance to B + 1. Since G is the goal state, the search exits.

```
BFS(Graph, start, value)
1.  for each vertex in Graph.Vertices
2.       vertex.visited = false
3.  start.visited = true //visited
4.  start.distance = 0 //distance to source
5.  start.parent = NULL
6.  Queue = empty //initialize empty queue
7.  Queue.enqueue(start)
8.  while Queue not empty
9.     u = Queue.dequeue()
10.    for each vertex in Graph.Adjacent[u] //each vertex adjacent to u
11.         if vertex.visited == false
12.            vertex.distance = u.distance + 1 //u is the parent, dequeued on line 8
13.            vertex.parent = u
14.            if vertex.value == value
15.                return vertex
16.            else
17.                vertex.visited = true
18.                Queue.enqueue(vertex)
19. return NULL
```

The BFS algorithm will return the vertex if it's found and NULL if it's not. Stored in the vertex structure is its distance back to the starting vertex *start*.

BFS Evaluated:

Completeness: Yes

Optimality: If all edge weights the same and the optimal solution is non-decreasing function of depth of node.

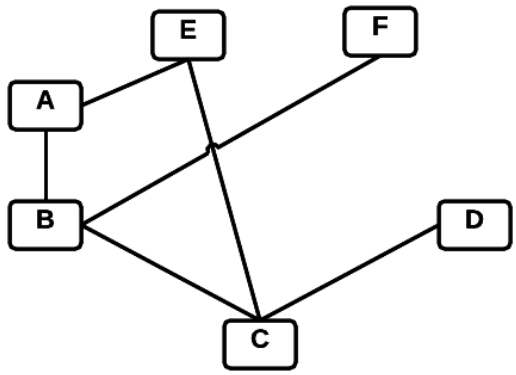
Time complexity: $O(b^d)$: b is branching factor and d is depth of node.

Space complexity: $O(b^d)$: b is branching factor and d is depth of node.

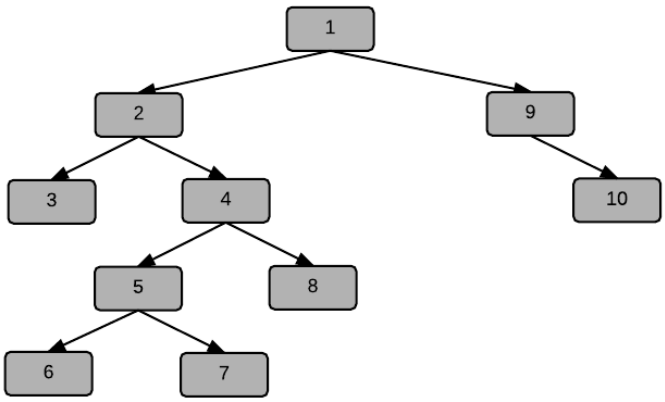
Depth-first search

Another ordering for searching the vertices in a graph, called depth-first search (DFS), evaluates the vertices along one path before evaluating other paths. DFS is also used in the tree-traversal algorithms for binary trees that print the nodes in the tree. Those algorithms recursively traverse all the way to the leaf nodes in the tree, following the left or right branch, before evaluating any of the other branches in the tree. For the graph shown below, a DFS that starts at vertex A

and selects the next adjacent vertex alphabetically, would evaluate vertices in the order *A*-*B*-*C*-*D* backing up and selecting a different path to evaluate vertices *E* and *F*. This ordering differs from a breadth-first search, which would evaluate vertices *B* and *E* before evaluating vertices *C* and *D*.



The evaluation order of vertices in a graph, assuming the graph has been converted to a search tree, is shown here:



Once the bottom of the left branch in a tree is reached, which is equivalent to following a path in a graph until there are no unvisited, adjacent vertices on that path to evaluate, DFS will evaluate all nodes in the right branch. In a graph, following a different branch means selecting a different vertex at the last decision point.

DFS can be handled using a recursive and a non-recursive algorithm. Non-recursive implementations of DFS typically use a stack data structure to store the vertices as they are visited. The stack generates an ordering where the most-recently visited vertices are popped off the stack and processed before vertices that were encountered at higher levels in the tree.

In both versions of the DFS algorithm, shown here, the parent of each node is stored when it is visited. (Note: in this algorithm, we assume the parent of the original vertex has been set elsewhere.)

```
DFS(vertex)
    vertex.visited = true
    for each v in vertex.adjacent
         if(!v.visited)
             v.parent = vertex
             print(v.key)
             DFS(v)
```

DFS evaluated

Completeness: yes, if the algorithm includes cycle checking to only visit unvisited nodes.

Optimality: no. It will return a solution if one exists, but not necessarily the optimal one without additional processing.

Time complexity: $O(b^h)$: h is height of tree.

Space complexity: O(bh): h is height of the tree. Algorithm only needs to store pointers back to the root.

< Prev (/csci/index.php/9-csci-3202-lecture-notes/27-lecture-5-6-searching-for-a-plan-ii)

Next > (/csci/index.php/9-csci-3202-lecture-notes/24-lecture-2-agents-and-the-environment)

© 2016 Computer science course notes

Back to Top