6.01: Introduction to EECS I

Search Algorithms

Week 12 *November 24, 2009*

Looking Ahead

- **Reaction:** Use a rule to determine the 'action' to take, as a direct function of the state
  - wall-following
  - proportional controller
- **Planning:** Choose action based on 'looking ahead': exploring alternative sequences of actions

Methods for planning require us to specify an explicit model of the effects of our actions in the world.

Using models to choose actions

Assume states and actions are discrete.

Given

- A state-machine model of the world
- A start state
- A goal test

Find a sequence of actions (inputs to the state machine) to reach a goal state from the start state.

Application: Navigation

What are good definitions of states, actions?

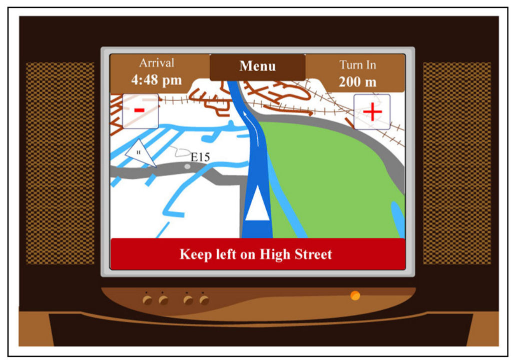
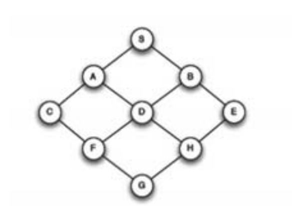


Figure by MIT OpenCourseWare.

What makes a path good?

Abstraction: Labeled graph



Lots of possible paths! Could enumerate them and evaluate each one. Too hard...

Assume *additive cost*. For now, each segment has a cost of one. We want to find the shortest path.

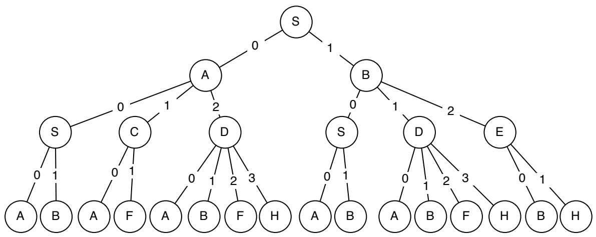
Formal model

- States: `{'S', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'}`.
- Starting state is `'S'`.
- Goal test: `lambda x: x == 'H'`
- Legal actions and successors:

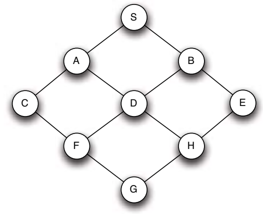
```
map1 = {'S' : ['A', 'B'],
        'A' : ['S', 'C', 'D'],
        'B' : ['S', 'D', 'E'],
        'C' : ['A', 'F'],
        'D' : ['A', 'B', 'F', 'H'],
        'E' : ['B', 'H'],
        'F' : ['C', 'D', 'G'],
        'H' : ['D', 'E', 'G'],
        'G' : ['F', 'H']}
map1LegalActions = [0, 1, 2, 3]

def map1successors(s, a):
    if a < len(map1[s]): return map1[s][a]
    else: return s
```

Search tree encodes all possible paths

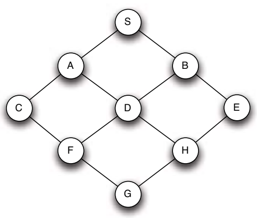


Check yourself



How many "shortest" paths are there from S to G?

Check yourself



How many "shortest" paths are there from S to G?

6

A numeric example

- States: integers
- Start state: 1
- Legal actions (and successors) in state n: $\{2n, n+1, n-1, n^2, -n\}$
- Goal test: $x = 10$

How long is the longest path in this domain?

Search trees in Python

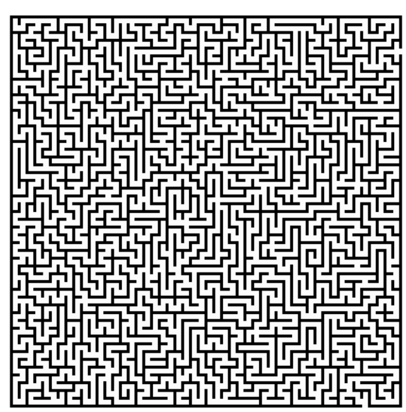
Node in search tree, not the same as a state!

```
class SearchNode:
    def __init__(self, action, state, parent):
        self.state = state
        self.action = action
        self.parent = parent
```

Search node encodes a whole path

```
def path(self):
    if self.parent == None:
        return [(self.action, self.state)]
    else:
        return self.parent.path() + \
                  [(self.action, self.state)]
```

Finding your way



Search algorithm

Until we find the goal or the agenda is empty:

- Extract a node from the agenda
- Expand it (find its children)
- Add its children to the agenda (visit its children)

Goal: Search as few nodes as possible while guaranteeing that we still find the shortest path.

Return a path:
$[(None, S_0), (A_1, S_1), (A_2, S_2), ..., (A_n, S_n)]$
where S_n satisfies the goal test.

Search in Python

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
```

Search in Python

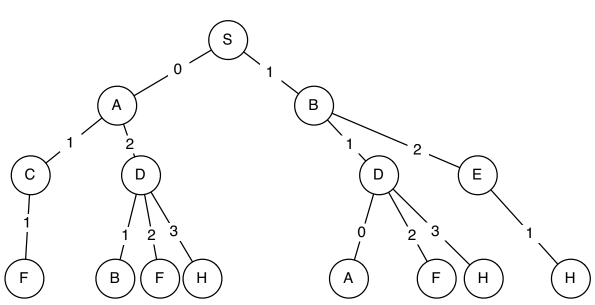
```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while not empty(agenda):
```

Search in Python

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while not empty(agenda):
        parent = getElement(agenda)
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                add(newN, agenda)
    return None
```

Don't be totally stupid!

Pruning Rule 1. Don't consider any path that visits the same state twice.

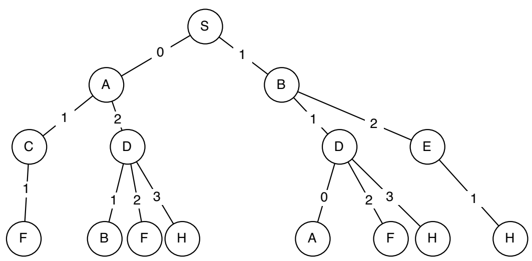


Not being totally stupid, in Python

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while not empty(agenda):
        parent = getElement(agenda)
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif parent.inPath(newS):
                pass
            else:
                add(newN, agenda)
    return None
```

Another pruning rule

Pruning Rule 2. If there are multiple actions that lead from a state r to a state s, consider only one of them.



Stack and Queue using Lists

```
class Stack:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop()
    def isEmpty(self):
        return self.data is []

class Queue:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop(0)
    def isEmpty(self):
        return self.data is []
```

Stack data structure

Last in, first out

```
>>> s = Stack()
>>> s.push(1)
>>> s.push(9)
>>> s.push(3)
>>> s.pop()
3
>>> s.pop()
9
>>> s.push(-2)
>>> s.pop()
-2
```

Queue data structure

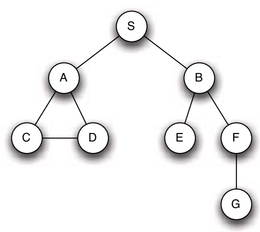
First in, first out

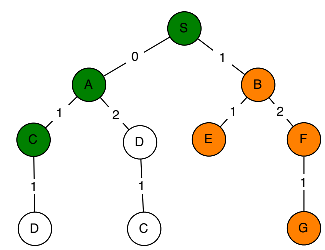
```
>>> q = Queue()
>>> q.push(1)
>>> q.push(9)
>>> q.push(3)
>>> q.pop()
1
>>> q.pop()
9
>>> q.push(-2)
>>> q.pop()
3
```

Depth-First search

```
def depthFirstSearch(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.isEmpty()
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:  # pruning rule 2
                pass
            elif parent.inPath(newS):     # pruning rule 1
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

DFS: From S to C





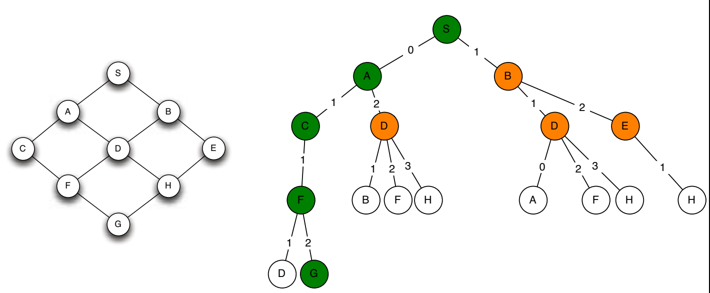
DFS properties

- May run forever if we don't apply pruning rule 1.
- May run forever in an infinite domain.
- Doesn't necessarily find the shortest path.
- Efficient in the amount of space it requires to store the agenda.

Breadth-First search

```
def breadthFirstSearch(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.isEmpty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

BFS: From S to G

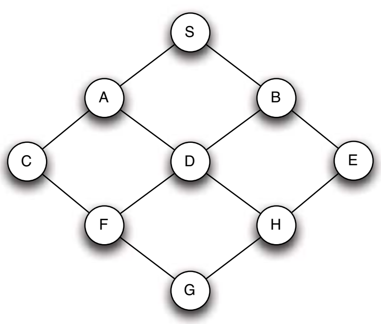


BFS properties

- Always returns a shortest path to a goal state, if a goal state exists in the set of states reachable from the start state.
- May run forever in an infinite domain if there is no solution.
- Requires more space than depth-first search.

Dynamic Programming

When happened when we did BFS in this city with goal G?



Visits 16 nodes, but there are only 9 states!!

Dynamic Programming Principle

The *shortest* path from X to Z that goes through Y is made up of

- the *shortest* path from X to Y and
- the *shortest* path from Y to Z.

So, we only need to remember the *shortest* path from the start state to each other state.

DP in breadth-first search

The *first* path that BFS finds from start to X is the *shortest* path from start to X.

So, we only need to remember the *first* path we find from the start state to each other state.

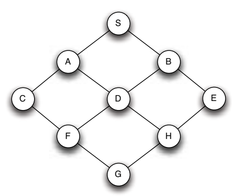
DP as a pruning technique

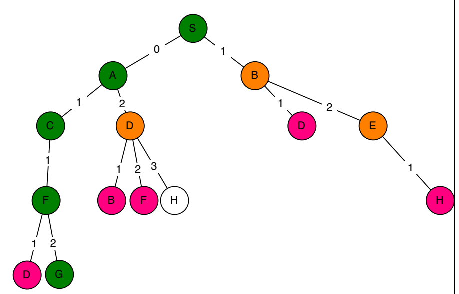
Pruning Rule 3. Don't consider any path that visits a state that you have already visited via some other path.

BFS with DP

```
def breadthFirstDP(initialState, goalTest, actions, successor):
    ...
    agenda.push(SearchNode(None, initialState, None))
    visited = {initialState: True}
    while not agenda.isEmpty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions:
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif visited.has_key(newS): # rules 1, 2, 3
                pass
            else:
                visited[newS] = True:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

BFS-DP: From S to G





Visits 9 states.

Can never expand more nodes than there are states.

Can be used with DFS as well.

State machines as world models

Add two new features:

- `done(self, state)`: returns `True` if the machine has terminated; we can use this as a goal test
- `legalInputs`: list of possible legal inputs to the machine; we can use this as the set of possible actions

For now, we will ignore the output of the state machine. Later we will put it to good use.

Planning in a state machine

Question: Given a state machine in its initial state, what sequence of inputs can we feed to it, in order to cause it to enter a done state?

Planning in a state machine

Question: Given a state machine in its initial state, what sequence of inputs can we feed to it, in order to cause it to enter a done state?

```
def smSearch(smToSearch, initialState = None, goalTest = None,
             maxNodes = 10000, depthFirst = False, DP = True):
   if initialState == None:
       initialState = smToSearch.startState
   if goalTest == None:
       goalTest = smToSearch.done
   return search(initialState, goalTest,
                 smToSearch.legalInputs,
                 lambda s, a: smToSearch.getNextValues(s, a)[0],
                 maxNodes = maxNodes,
                 depthFirst=depthFirst, DP=DP)
```

A numeric example

- States: integers
- Start state: 1
- Legal actions (and successors) in state n: $\{2n, n+1, n-1, n^2, -n\}$
- Goal test: $x = 10$

A numeric example – state machine

```
class NumberTestSM(sm.SM):
    startState = 1
    legalInputs = ['x*2', 'x+1', 'x-1', 'x**2', '-x']
    def __init__(self, goal):
        self.goal = goal
    def nextState(self, state, action):
        if action == 'x*2':
            return state*2
        elif action == 'x+1':
            return state+1
        elif action == 'x-1':
            return state-1
        elif action == 'x**2':
            return state**2
        elif action == '-x':
            return -state
    def getNextValues(self, state, action):
        nextState = self.nextState(state, action)
        return (nextState, nextState)
    def done(self, state):
        return state == self.goal
```

Numeric – Breadth First

```
>>> smSearch(NumberTestSM(10), initialState = 1,
             depthFirst = False, DP = False)
   expanding:  1
   expanding:  1-x*2->2
   expanding:  1-x-1->0
   expanding:  1--x->-1
   expanding:  1-x*2->2-x*2->4
   expanding:  1-x*2->2-x+1->3
   expanding:  1-x*2->2--x->-2
   expanding:  1-x-1->0-x-1->-1
   expanding:  1--x->-1-x*2->-2
   expanding:  1--x->-1-x+1->0
   expanding:  1-x*2->2-x*2->4-x*2->8
   expanding:  1-x*2->2-x*2->4-x+1->5
33  states visited
[(None, 1), ('x*2', 2), ('x*2', 4), ('x+1', 5), ('x*2', 10)]
```

Numeric – Breadth First with DP

```
>>> smSearch(NumberTestSM(10), initialState = 1,
             depthFirst = False, DP = True)
   expanding:  1
   expanding:  1-x*2->2
   expanding:  1-x-1->0
   expanding:  1--x->-1
   expanding:  1-x*2->2-x*2->4
   expanding:  1-x*2->2-x+1->3
   expanding:  1-x*2->2--x->-2
   expanding:  1-x*2->2-x*2->4-x*2->8
   expanding:  1-x*2->2-x*2->4-x+1->5
17  states visited
[(None, 1), ('x*2', 2), ('x*2', 4), ('x+1', 5), ('x*2', 10)]
```

Numeric – Depth First

Limit numbers to be in range (-20, 20)

```
>>> smSearch(NumberTestFiniteSM(10, 20), initialState = 1,
             depthFirst = True, DP = False)
   expanding:  1
   expanding:  1--x->-1
   expanding:  1--x->-1-x+1->0
   expanding:  1--x->-1-x*2->-2
   expanding:  1--x->-1-x*2->-2--x->2
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3--x->-3
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3--x->-3-x**2->9
20  states visited
[(None, 1), ('-x', -1), ('x*2', -2), ('-x', 2), ('x+1', 3), ('-x', -3),
('x**2', 9), ('x+1', 10)]
```

Computational complexity

Let

- b be the *branching factor* of the graph; that is, the number of successors a node can have.
- d be the *maximum depth* of the graph; that is, the length of the longest path in the graph.
- l be the *solution depth* of the problem; that is, the length of the shortest path from the start state to the shallowest goal state.
- n be the *state space size* of the graph; that is the total number of states in the domain.

There are b^d paths at depth d.

The number of nodes in the tree of depth d is about b^{d+1}.

Without dynamic programming

- Depth first:
  - may have to search every path (b^{d+1} nodes), but
  - agenda is small (bd)
- Breadth first:
  - may have to search to depth l (b^{l+1} nodes),
  - agenda may be as large as b^l

With dynamic programming

Visit at most n states!

Sometimes $n << b^l$ (in a road network, for example), sometimes not (small problem in large space).

DP is almost always an improvement in running time.

6.01 Introduction to Electrical Engineering and Computer Science I
Fall 2009