6.01: Introduction to EECS I

Search Algorithms

April 26, 2011

Nano-Quiz Makeups

Wednesday, May 4, 6-11pm, 34-501.

– everyone can makeup/retake NQ 1
– everyone can makeup/retake two additional NQs
– you can makeup/retake other NQs excused by S^3

If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower!

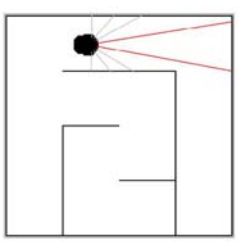
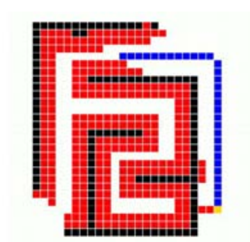
Last Time: Probability

Modeling uncertainty and making robust plans.

Topics: **Bayes' theorem**, search strategies

Lab exercises:

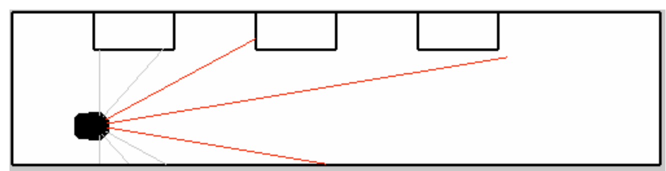
- Mapping: drive robot around unknown space and make map.
- Localization: give robot map and ask it to **find where it is**.
- Planning: plot a route to a goal in a maze

Themes: Robust design in the face of uncertainty

Design Lab 12: One-Dimensional Localizer

As robot drives along hallway with obstacles to its side, estimate its current position based on previous estimates and sonar information.



State S_t: discretized values of distance along the hallway (x).

Transition model $\Pr(S_{t+1} = s' \mid S_t = s)$: conditional distribution of next state given current state.

Observation model $\Pr(O_t = d \mid S_t = s)$: conditional distribution of left-facing sonar readings (y) given state.

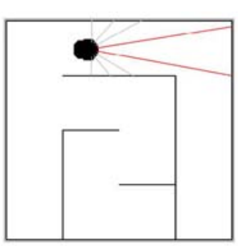
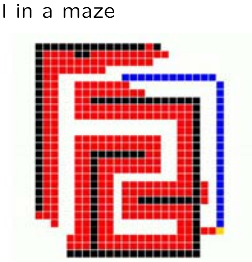
Today: Search Strategies

Modeling uncertainty and making robust plans.

Topics: Bayes' theorem, **search strategies**

Lab exercises:

- Mapping: drive robot around unknown space and make map.
- Localization: give robot map and ask it to find where it is.
- **Planning**: plot a route to a goal in a maze

We will **plan** a route by **searching** through possible alternatives.

Planning

Make a plan by searching.

Example: Eight Puzzle

Start

1	2	3
4	5	6
7	8	

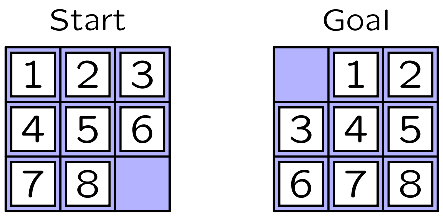
Goal

	1	2
3	4	5
6	7	8

Rearrange board by sequentially sliding tiles into the free spot.

Check Yourself

How many different board configurations (states) exist?



1. $8^2 = 64$
2. $9^2 = 81$
3. $8! = 40320$
4. $9! = 362880$
5. none of the above

Search Algorithm

Develop an algorithm to systematically conduct a search.

Analyze how well the algorithm performs.

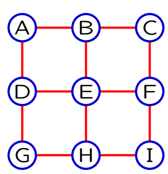
Optimize the algorithm:

– find the "best" solution (i.e., minimum path length)

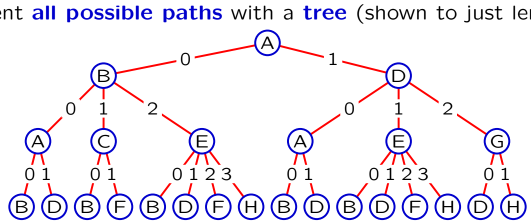
– by considering as few cases as possible.

Algorithm Overview

Find minimum distance path between 2 points on a rectangular grid.



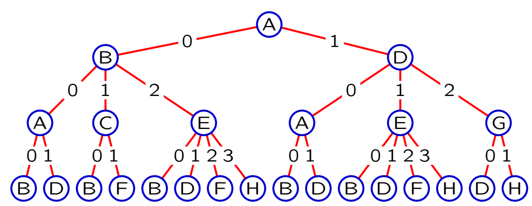
Represent **all possible paths** with a **tree** (shown to just length 3).



Find the shortest **path** from A to I.

Algorithm Overview

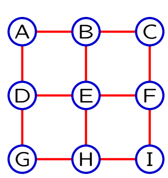
The tree could be infinite.



Therefore, we will construct the tree and search at the same time.

Python Representation

Represent possible locations by **states**: `'A','B','C','D',...'I'`



Represent possible transitions with **successor** procedure

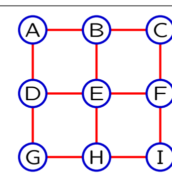
- inputs: current state (location) and action (e.g., up, right, ...)
- output: new state

Define **initialState** (starting location)

Determine if goal has been achieved with **goalTest** procedure

- input: state
- output: `True` if state achieves goal, `False` otherwise.

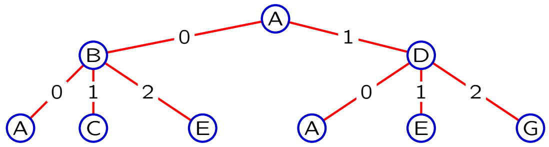
Python Representation




```
successors = { 'A': ['B','D'],          'B': ['A','C','E'],
               'C': ['B','F'],          'D': ['A','E','G'],
               'E': ['B','D','F','H'],  'F': ['C','E','I'],
               'G': ['D','H'],          'H': ['E','G','I'],
               'I': ['F','H'] }
actions = [0, 1, 2, 3]

def successor(s,a):
    if a<len(successors[s]): return successors[s][a]
    else: return s

initialState = 'A'

def goalTest(s):
    return s=='I'
```

Search Trees in Python

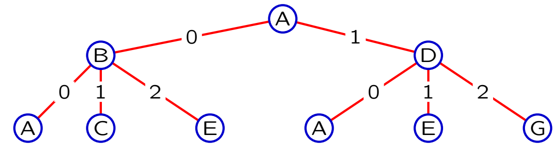
Represent each **node** in the tree as an instance of class **SearchNode**.




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

Search Algorithm

Construct the tree and find the shortest path to the goal.



Algorithm:

- initialize **agenda** (list of nodes being considered) to contain starting node
- repeat the following steps:
  - remove one node from the agenda
  - add that node's children to the agenda

  until **goal is found** or **agenda is empty**
- return resulting path

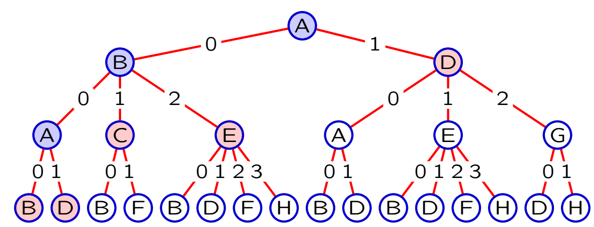
Search Algorithm in Python

Repeatedly (1) **remove** node (parent) from agenda and (2) **add** parent's children until goal is reached or agenda is empty.

```
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
```

Order Matters

Replace first node in agenda by its children:



step Agenda

0: A

1: AB AD

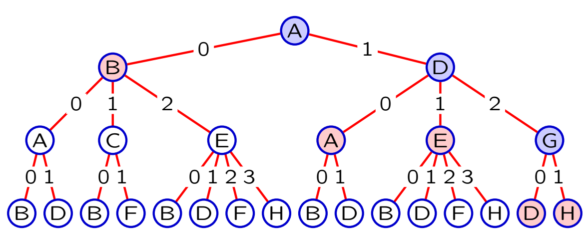
2: ABA ABC ABE AD

3: ABAB ABAD ABC ABE AD

Depth First Search

Order Matters

Replace last node in agenda by its children:



step Agenda

0: A

1: AB AD

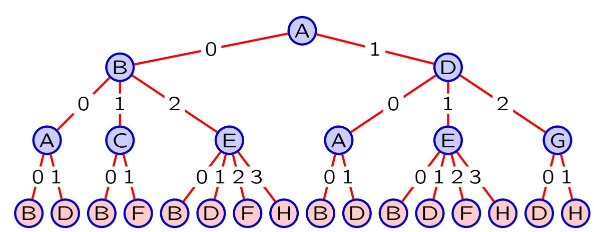
2: AB ADA ADE ADG

3: AB ADA ADE ADGD ADGH

also Depth First Search

Order Matters

Remove first node from agenda. Add its children to end of agenda.



step Agenda

8: ADG ABAB ABAD ABCB ABCF ABEB ABED ABEF ABEH ADAB ADAD ADEB ADED ADEF ADEH

9: ABAB ABAD ABCB ABCF ABEB ABED ABEF ABEH ADAB ADAD ADEB ADED ADEF ADEH ADGD ADGH

Breadth First Search

Order Matters

Replace last node by its children (depth-first search):
– implement with **stack** (last-in, first-out).

Remove first node from agenda. Add its children to the end of the agenda (breadth-first search):
– implement with **queue** (first-in, first-out).

Stack

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
```

Stack Class

Last in, first out.

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

Queue

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
```

Queue Class

First in, first out.

```
class Queue:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop(0)    #NOTE: different argument
    def empty(self):
        return self.data is []
```

Depth-First Search

Replace **getElement**, **add**, and **empty** with **stack** commands.

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

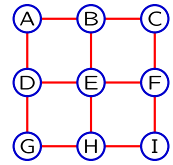
Breadth-First Search

Replace **getElement**, **add**, and **empty** with **queue** commands.

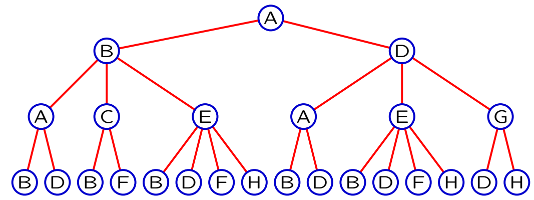
```
def search(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.empty():
        parent = agenda.pop()
        for a in actions:
          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

Too Much Searching

Find minimum distance path between 2 points on a rectangular grid.



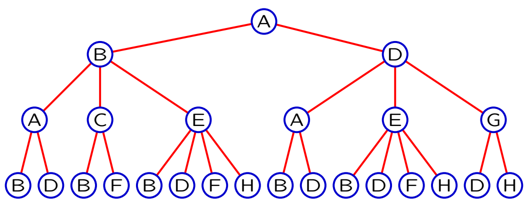
Represent **all possible paths** with a **tree** (shown to just length 3).



Not all of the nodes of this tree must be searched!

Check Yourself

How many of these terminal nodes can be ignored?

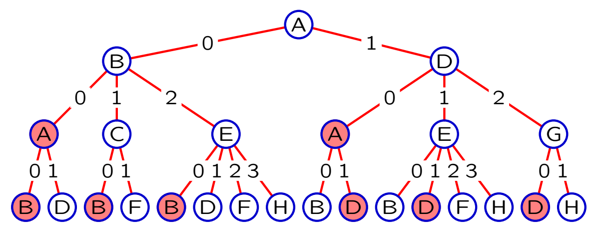


1. 0 2. 2 3. 4 4. 6 5. 8

Pruning

Prune the tree to reduce the amount of work.

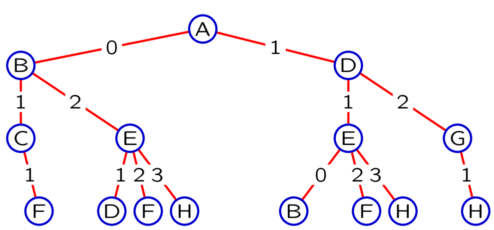
Pruning Rule 1:
Don't consider any path that visits the same state twice.



Pruning

Prune the tree to reduce the amount of work.

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



Pruning Rule 1

Implementation (depth first, switch to **Queue** for breadth first)

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

Pruning Rule 1

Add **inPath** to SearchNode.

```
class SearchNode:
    def __init__(self, action, state, parent):
        self.action = action
 self.state = state
 self.parent = parent
    def path(self):
        if self.parent == None:
            return [(self.action, self.state)]
        else:
            return self.parent.path() + [(self.action, self.state)]
    def inPath(self, state):
        if self.state == state:
            return True
        elif self.parent == None:
            return False
        else:
            return self.parent.inPath(state)
```

Pruning

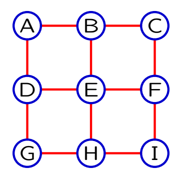
Prune the tree to reduce the amount of work.

Pruning Rule 2:
If multiple actions lead to the same state, consider only one of them.

Pruning Rule 2

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

Depth-First Search Example



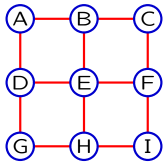
Depth-First Search Properties

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

Breadth-First Search Example

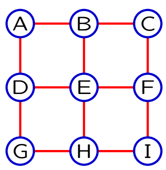


Breadth-First Search Properties

- 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.

Still Too Much Searching

Breadth-first search, visited 16 nodes: but there are only 9 states!



We should be able to reduce the search even further.

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.

We only need to remember the *shortest* path from the start state to each other state!

Dynamic Programming in Breadth-First Search

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

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

Dynamic Programming as a Pruning Technique

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

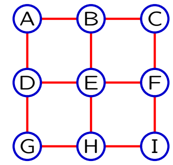
Need to remember the first path we find to each state.

Use dictionary called **visited**

Breadth-First Search with Dynamic Programming

```
def breadthFirstDP(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    visited = {initialState: True}
    while not agenda.empty():
        parent = agenda.pop()
        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
                agenda.push(newN)
    return None
```

Breadth-First with Dynamic Programming Example



Summary

Developed two search algorithms
– depth-first search
– breadth-first search
Developed three pruning rules
– don't consider any path that visits the same state twice
– if multiple actions lead to same state, only consider one of them
– dynamic programming: only consider the first path to a given state

Nano-Quiz Makeup: Wednesday, May 4, 6-11pm, 34-501.
– everyone can makeup/retake NQ 1
– everyone can makeup/retake two additional NQs
– you can makeup/retake other NQs excused by S^3

If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower!