Planning and Search

6.01 Spring 2008
Week 10
Tomas Lozano-Perez

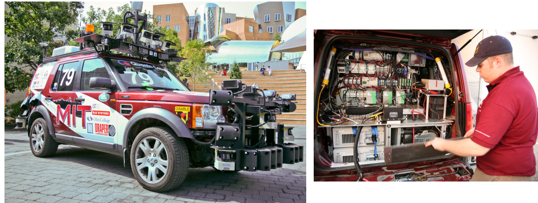
The story so far…

- PCAP framework for building systems
  - Python programs
  - Linear signal systems
  - Circuits
- Models for analysis and design
  - Difference equations
  - System functions
  - Constraint systems

Reaction vs Planning

- Reaction
  - A rule that determines the "action" to take
    - Proportional controller (K*error)
    - Behaviors (non-deterministic, sequential)
- Planning
  - Choose action based on "looking ahead" – exploring the results of alternative action sequences
  - Need a model in the robot's head…

DARPA Urban Grand Challenge



Intersection

Using models to choose actions

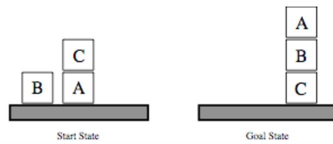
- Given a state-machine model of a system
  - States
  - Actions
  - Transition function
- Given a start state and a goal state
- Find a sequence of actions to reach the goal state from the start state
- Assume states and actions are discrete

Navigation

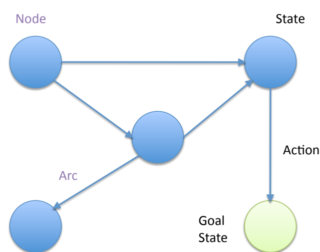
- States
- Actions
- Start state
- Goal test

Manipulation

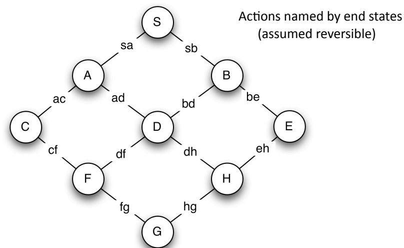
- States
- Actions
- Start state
- Goal test



Abstraction: Labeled graph



Map 1

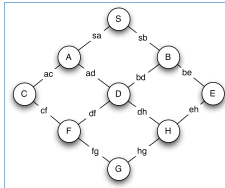


Actions named by end states
(assumed reversible)

Successor

```
map1 = {'S' : [('sa', 'A'), ('sb', 'B')],
        'A' : [('sa', 'S'), ('ac', 'C'), ('ad', 'D')],
        'B' : [('sb', 'S'), ('bd', 'D'), ('be', 'E')],
        'C' : [('ac', 'A'), ('cf', 'F')],
        'D' : [('ad', 'A'), ('bd', 'B'), ('df', 'F'), ('dh', 'H')],
        'E' : [('be', 'B'), ('eh', 'H')],
        'F' : [('cf', 'C'), ('df', 'D'), ('fg', 'G')],
        'H' : [('dh', 'D'), ('eh', 'E'), ('hg', 'G')],
        'G' : [('fg', 'F'), ('hg', 'H')]}

def map1successors(x):
    return map1[x]
```

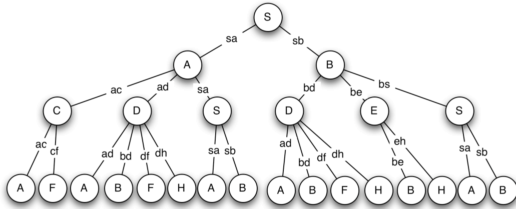


Search for sequences of arithmetic operations

```
def successors (state):
    return [('x*2', state*2),
            ('x+1', state+1),
            ('x-1', state-1),
            ('x**2', state**2),
            ('-x', -state)]
```

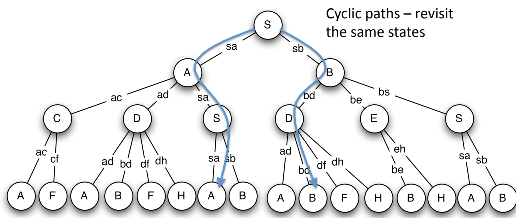
Search: systematic exploration

- Enumerate partial paths in some order
- Stop when we find the goal

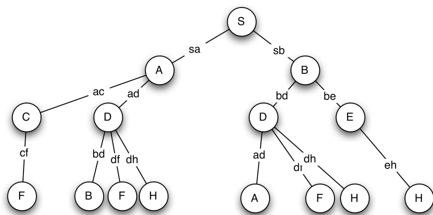


Paths starting at state S

Cyclic paths – revisit the same states



Consider only non-cyclic paths



Search: systematic exploration

- Depth-first
  - Keep extending the most recent path that has successors
- Breadth-first
  - Keep extending the path with the fewest nodes

Generic Search

Agenda: Partially explored paths

S

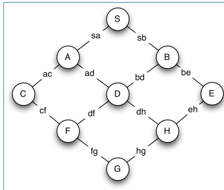
SA SB

SAC SAD SB

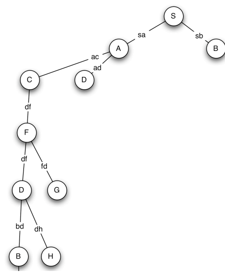
SACF SAD SB

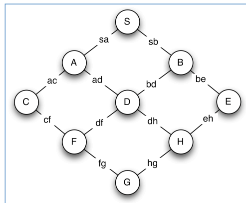
Visit = place on Agenda
Expand = remove from Agenda and visit successors

Stop when you visit a goal state



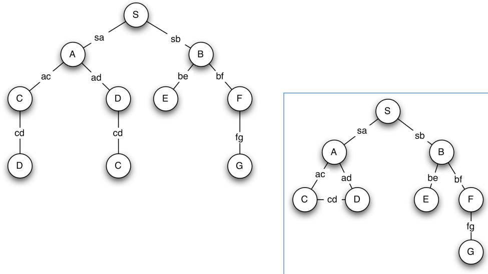
Depth-first search tree: S to H





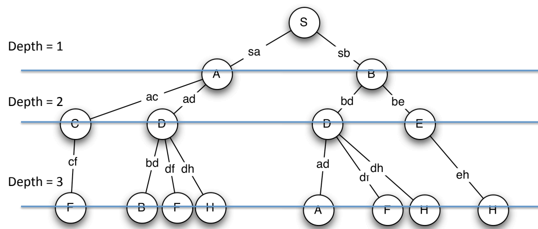
Warning: can go on forever in infinite spaces

Example 2: DFS: S to G



Breadth-first search

Guarantees a shortest path

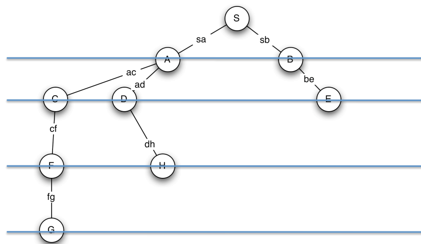


Dynamic Programming Principle

- Shortest path from X to Z that goes through Y
  - Shortest path from X to Y
  - Shortest path from Y to Z
- Don't add paths to states that you've already visited
  - New path is only going to be longer
- **With DP we only visit as many paths as there are states**

Breadth-first search tree with DP

Guarantees a shortest path



Search

```
def search(initialState, goalTest, successors):
```

Search

```
def search(initialState, goalTest, successors):
    initialAS = (None, initialState)  # no action
    if goalTest(initialState): return [initialAS]
    agenda = [[initialAS]]              # a list of a single path
```

Search

```
def search(initialState, goalTest, successors):
    initialAS = (None, initialState)  # no action
    if goalTest(initialState): return [initialAS]
    agenda = [[initialAS]]              # a list of a single path
    while agenda != []:
        path = agenda.pop(0)
        newPaths = []
        for newAS in successors(state(path[-1])):
```

Search

```
def search(initialState, goalTest, successors):
    initialAS = (None, initialState)  # no action
    if goalTest(initialState): return [initialAS]
    agenda = [[initialAS]]              # a list of a single path
    while agenda != []:
        path = agenda.pop(0)
        newPaths = []
        for newAS in successors(state(path[-1])):
            if goalTest(state(newAS)):
                return path + [newAS]
            elif stateInASList(state(newAS), path):
                pass                    # cyclic path
            else:
                newPaths.append(path + [newAS])
```

Search

```
def search(initialState, goalTest, successors):
    initialAS = (None, initialState)  # no action
    if goalTest(initialState): return [initialAS]
    agenda = [[initialAS]]              # a list of a single path
    while agenda != []:
        path = agenda.pop(0)
        newPaths = []
        for newAS in successors(state(path[-1])):
            if goalTest(state(newAS)):
                return path + [newAS]
            elif stateInASList(state(newAS), path):
                pass                    # cyclic path
            else:
                newPaths.append(path + [newAS])
        agenda = agenda + newPaths       # Breadth first
    return None
```

Search

```
def search(initialState, goalTest, successors):
    initialAS = (None, initialState)  # no action
    if goalTest(initialState): return [initialAS]
    agenda = [[initialAS]]              # a list of a single path
    while agenda != []:
        path = agenda.pop(0)
        newPaths = []
        for newAS in successors(state(path[-1])):
            if goalTest(state(newAS)):
                return path + [newAS]
            elif stateInASList(state(newAS), path):
                pass                    # cyclic path
            else:
                newPaths.append(path + [newAS])
        agenda = agenda + newPaths       # Breadth first
    return None
```

SearchDP

```
def searchDP(initialState, goalTest, successors):
    initialAS = (None, initialState)  # no action
    if goalTest(initialState): return [initialAS]
    agenda = [initialAS]                # a list of a single AS
    pathTo = {initialState: [initialAS]}
    while agenda != []:
        AS = agenda.pop(0)
        S = state(AS)
        newASList = []
        for newAS in successors(state(AS)):
            newS = state(newAS)
            if not pathTo.has_key(newS):
                pathTo[newS] = pathTo[S] + [newAS]
                if goalTest(newS):
                    return pathTo[newS]
                else:
                    newASList.append(newAS)
         agenda = agenda + newASList  # Breadth First
    return None
```

Complexity: worst case

- B = max branching factor
- D = max depth of graph
- L = solution depth
- N = state space size (number of states)

- B^D paths at depth D
- N is about B^{D+1} up to depth D

Complexity: worst case – no DP

- Depth-first:
  - may have to search every path (B^{D+1}), but
  - agenda is small (BD)
- Breadth-first:
  - may have to search to depth L (B^{L+1}), and
  - agenda may be as large as B^L

Velodyne