CS 440: Introduction to Artificial Intelligence

Lecture 8

Matthew Stone

September 30, 2015

Recap— Challenge of Search

Applying knowledge in new situations

- Actual situations are *complex*.
  You usually haven't seen exactly the same thing before.
  Often: a novel mix of familiar features.
- Useful knowledge needs to be *generalizable*
  It describes large classes of situations
  in terms of underlying features.
- Consequence:
  New situations require a creative synthesis of existing knowledge

Search

Way to creatively synthesize pieces of knowledge

- Symbol structures represent current information
- Applying a piece of knowledge extends this information
- Can test whether current information solves your problem
- Systematically explore all the alternatives

Ingredients of search problems

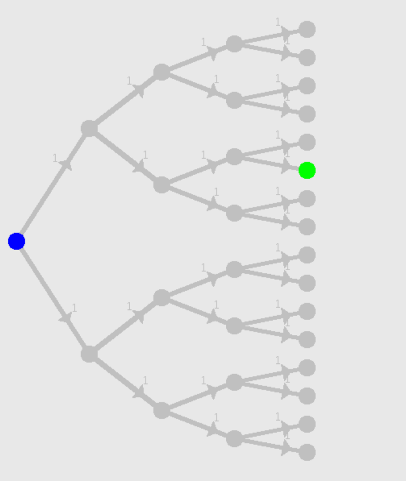
- Initial state
- Possible actions in each state
- Transition model:
  Takes state and action and gives new state
- Goal test
  Describes whether state is what you want
- Path cost
  Says how easy or hard action sequence is

General Case

State space is a tree

- Each node has a set of children
  obtained by considering different actions
- Each action sequence leads to a new state
- Visualization and demo

General Case



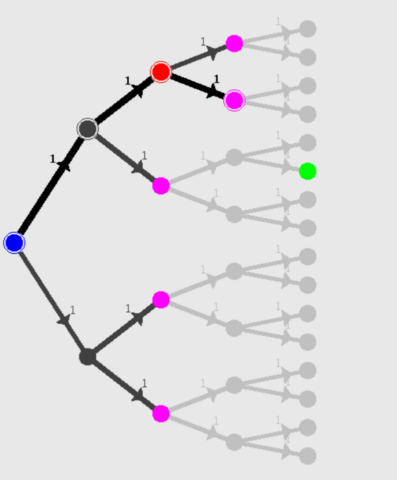
Search tree
Start in blue
Goal in green

Breadth-first search

Simple regime for exploring

- Gradually "fan out" into the search space
- Explore level by level
- Consider all the nodes at level n first
- Then consider nodes at level $n + 1$ (and so on)
- Visualization and demo

Snapshot during breadth-first search



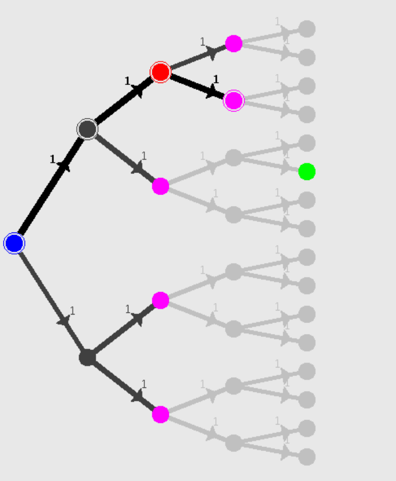
Implementing search

Key data elements

- states
  circles in demo,
  including start (blue), goal (green)
- nodes
  paths in demo, distinguished by ending state
  (only one path per end state!)
  current node has bold path, red end state

Snapshot during breadth-first search

Search tree
Start state in blue
Goal state in green
Current node in red
with bold path



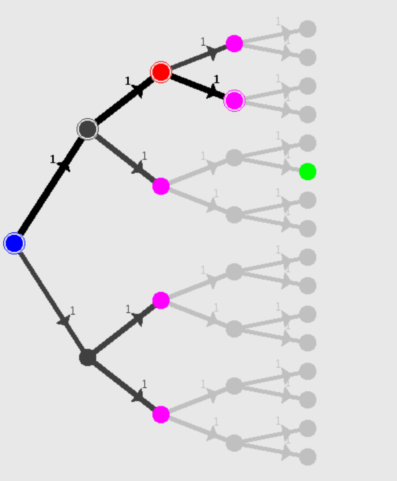
Implementing search

Key data elements

- frontier nodes
  end states with purple highlights in demo
  nodes you have created but have not yet explored
- explored set
  blacked-out end states in demo nodes you have created and explored

Snapshot during breadth-first search

Search tree
Start state in blue
Goal state in green
Current node in red
with bold path
Frontier nodes at purple states
Explored nodes at black ones



Implementing search

```
function StartSearch(problem)
  initialize frontier:
     new node for initial state
  initialize explored set:
     empty
```

Implementing search

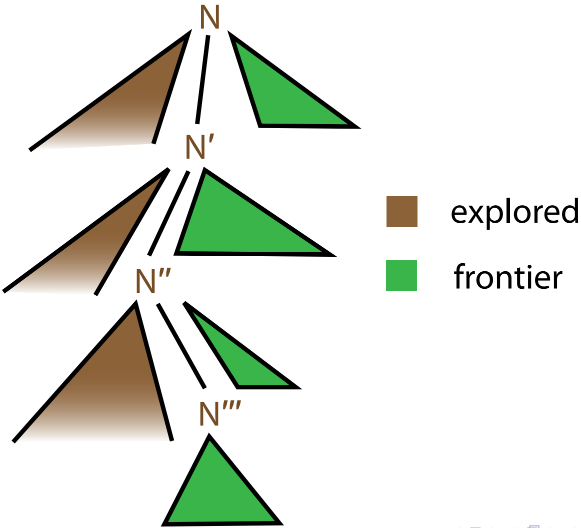
```
function ContinueSearch(problem)
return solution or done
loop
  if frontier is empty return done
  pop next node from frontier
  add to explored set
  if it is a goal return it
  expand it and add resulting nodes to frontier
      (if no node for state already in frontier
       or already in explored set)
```

Depth-first search

- Implement frontier as a stack
- Small space requirements
- Efficient realization through function calls
- Not always shortest path first
- Related idea: "iterative deepening"

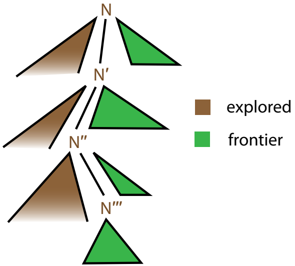
Demo

Schematic view of DFS in tree



DFS in tree

- Forget explored nodes
- $O(kd)$ nodes in frontier (siblings of nodes on path)



DFS with native function calls

```
function DFS(node)
   if node is a goal
      return node
   else
      for a in actions
         solution = DFS(next(node, a))
         if solution is not done
            return solution
      return done
```

DFS with native function calls

Gives only one solution.

- Must save call stack to resume search

No visit check.

- Can add if `next` accesses existing data
- Saving nodes ruins good memory performance

Obvious problem: infinite paths.

Go-to variant: Iterative deepening

Limit depth on any run of DFS

```
function LDFS(node,depth)
    if node is a goal
        return node
    else if depth < 1
        return done
    else
        for a in actions
            solution = LDFS(next(node, a), depth-1)
            if solution is not done
                return solution
        return done
```

Iterative Deepening

Strategy

- Do search in rounds
- In each round, limit number of steps allowed
- Increase limit to find additional solutions

Properties

- Low memory requirement
- Constant factor additional time
  compared to single DFS run to least solution depth
- Gives shortest path result

Search spaces

Convenient to think in terms of tree

- Number of actions: “branching factor” k
- Complexity of solution: “depth” d
- Size of tree: $O(k^d)$

Corrolary

Search is intractable (in depth/size of solution)

- if k is (asymptotically) bigger than 1
- if (asymptotically) constant fraction of space is explored

So why are we learning it?

Two answers

- Boring answer: no alternative
  - NP-complete problems
  - Sometimes we have to solve them anyway
- Interesting answer: AI methodology
  - Search is a response to AI problem
  - Programmer has only partial information
  - System must decide what to do on its own

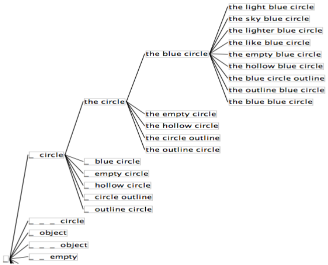
Good Practical Design for Easy Problems

Can make local decisions

- No "dead ends" – you can always say more
- Partial meaning is a good guide to progress

Explore only tiny part of search space

- Look at all actions in each state
- Pick the best and never look back
- $O(kd)$



Good Practical Design

- Program factored into knowledge and goals
- Knowledge
  - Grammatical structures
  - Words and their meanings
  - Shared context
- Goals
  - Complete sentence
  - Right meaning
  - Unambiguous
  - Natural

Good Practical Design

- Program factored into knowledge and goals
- Get flexible, general decision making without rules
- Can learn knowledge from other data sources
- Can adapt goals to new objectives
- Same code works