Chapter 3
Problem Solving using Search

"First, they do an on-line search"

Pac-Man as an Agent

SCORE: 0
The CSE 473 Pac-Man Projects

Originally developed at UC Berkeley:
http://www-inst.eecs.berkeley.edu/~cs188/pacman/pacman.html

Project 1: Search

Goal:
• Help Pac-man find its way through the maze

Techniques:
• Search: breadth-first, depth-first, etc.
• Heuristic Search: Best-first, A*, etc.
Project 2: Game Playing

Goal:
Build a rational Pac-Man agent!

Techniques:
Adversarial Search: minimax, alpha-beta, expectimax, etc.

Project 3: Planning and Learning

Goal:
Help Pac-Man learn about its world

Techniques:
• Planning: MDPs, Value Iteration
• Learning: Reinforcement Learning
Project 4: Ghostbusters

Goal:
Help Pac-man hunt down the ghosts

Techniques:
• Probabilistic models: HMMs, Bayes Nets
• Inference: State estimation and particle filtering

Problem Solving using Search

Example 1: The 8-puzzle

<table>
<thead>
<tr>
<th>Start</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>8 4</td>
<td>4 5 6</td>
</tr>
<tr>
<td>7 6 5</td>
<td>7 8</td>
</tr>
</tbody>
</table>
Example 2: Route Planning

Example 3: N Queens

4 Queens problem

(Place queens such that no queen attacks any other)
State-Space Search Problems

General problem:
Find a path from a start state to a goal state given:

- A goal test: Tests if a given state is a goal state
- A successor function (transition model): Given a state and action, generate successor state

Variants:

- Find any path vs. a least-cost path (if each step has a different cost i.e. a “step-cost”)
- Goal is completely specified, task is to find a path or least-cost path
  - Route planning
- Path doesn’t matter, only finding the goal state
  - 8 puzzle, N queens, Rubik’s cube
Example: Simplified Pac-Man

Input:
- State space
- Successor function
- Start state
- Goal test

Search Trees

A search tree:
- Root contains Start state
- Children = successor states
- Edges = actions and step-costs
- Path from Root to a node is a “plan” to get to that state
- For most problems, we can never actually build the whole tree (why?)
**State Space Graph versus Search Trees**

**State Space Graph**
(graph of states with arrows pointing to successors)

**Search Tree**
Search Tree for 8-Puzzle

Implementation: states vs. nodes

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree
includes parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree
includes parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!
Searching with Search Trees

Search:
- Expand out possible nodes
- Maintain a fringe or frontier of as yet unexpanded nodes
- Try to expand as few tree nodes as possible

Implementation: general tree search

```plaintext
function TREE-SEARCH(problem, fringe) returns a solution, or failure
fringe ← INSERT(MAKE-NODE(INITIAL-STATE(problem)), fringe)
loop do
  if fringe is empty then return failure
  node ← REMOVE-FRONT(fringe)
  if GOAL-TEST[problem] applied to STATE(node) succeeds return node
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```
Handling Repeated States

Failure to detect repeated states (e.g., in 8 puzzle) can cause infinite loops in search

```
function Tree-Search( problem, fringe ) returns a solution, or failure
    fringe ← Insert(Make-Node([INITIAL-STATE][problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test[problem] applied to State(node) succeeds return node
        fringe ← InsertAll(Expand(node, problem), fringe)
    
function Expand( node, problem ) returns a set of nodes
    successors ← the empty set
    for each action, result in Successor-Fn[problem][State[node]] do
        s ← a new Node
        Parent-Node[s] ← node; Action[s] ← action; State[s] ← result
        Path-Cost[s] ← Path-Cost[node] + Step-Cost(node, action, s)
        Depth[s] ← Depth[node] + 1
        add s to successors
    return successors
```

Graph Search algorithm: Augment Tree-Search to store expanded nodes in a set called explored set (or closed set) and only add new nodes not in the explored set to the fringe
Search strategies

A strategy is defined by picking the order of node expansion.

Strategies are evaluated along the following dimensions:
- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of
- $b$—maximum branching factor of the search tree
- $d$—depth of the least-cost solution
- $m$—maximum depth of the state space (may be $\infty$)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
**Breadth-first search**

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end

---

![Tree Diagram](tree_diagram.png)

---

**Breadth-first search**

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end

---

![Tree Diagram](tree_diagram.png)
Breadth-first search

Expand shallowest unexpanded node

Implementation:

- fringe is a FIFO queue, i.e., new successors go at end

```
A
 / \  
B   C
 / \   
D   E  (F) (G)
```

Breadth-first search

Expand shallowest unexpanded node

Implementation:

- fringe is a FIFO queue, i.e., new successors go at end

```
A
 / \  
B   C
 /   
D   E  (F) (G)
```
Properties of breadth-first search

Complete??

Yes (if b is finite)

Time??
Properties of breadth-first search

**Complete??** Yes (if \( b \) is finite)

**Time??** \( b + b^2 + b^3 + \cdots + b^d = O(b^d) \) i.e. exp in \( d \)

**Space??** 

---

Properties of breadth-first search

**Complete??** Yes (if \( b \) is finite)

**Time??** \( b + b^2 + b^3 + \cdots + b^d = O(b^d) \) i.e. exp in \( d \)

**Space??** \( O(b^d) \)

**Optimal??**
Properties of breadth-first search

**Complete??** Yes (if $b$ is finite)

**Time??** $b + b^2 + b^3 + \cdots + b^d = O(b^d)$ i.e. exp in $d$

**Space??** $O(b^d)$

**Optimal??** Yes if all step costs are equal. Not optimal in general.

Space and time are big problems for BFS.

Example: $b = 10$ with 1,000,000 nodes/sec, 1000 Bytes/node

d = 2 $\Rightarrow$ 110 nodes, 0.11 millisecs, 107KB

d = 4 $\Rightarrow$ 11,110 nodes, 11 millisecs, 10.6 MB

d = 8 $\Rightarrow$ $10^8$ nodes, 2 minutes, 103 GB

d = 16 $\Rightarrow$ $10^{16}$ nodes, 350 years, 10 EB (1 billion GB)

---

**What if the step costs are not equal?**

Can we modify BFS to handle any step cost function?
**Uniform-cost search**

Expand least-cost unexpanded node

Implementation:

\[ \text{fringe} = \text{queue ordered by path cost } g(n) \]

(Use priority queue)

Equivalent to breadth-first if step costs all equal

**Complete??** Yes, if step cost \( \geq \epsilon \)

**Time??** \# of nodes with \( g \leq \) cost of optimal solution, \( O\left(b^{\left\lfloor \frac{C^*}{\epsilon} \right\rfloor + 1}\right) \)

where \( C^* \) is the cost of the optimal solution

**Space??** \# of nodes with \( g \leq \) cost of optimal solution, \( O\left(b^{\left\lfloor \frac{C^*}{\epsilon} \right\rfloor + 1}\right) \)

**Optimal??** Yes—nodes expanded in increasing order of \( g(n) \)

---

**Can we do better?**

Next time: depth first search, depth limited search, iterative deepening search, bidirectional search

All these methods are slow (because they are “blind”)

Solution \( \rightarrow \) use problem-specific knowledge to guide search (“heuristic function”)

\( \rightarrow \) “informed search” (next lecture)

**To Do**

- Start Project #1
- Read Chapter 3