CSUG

- Tutoring: bit.ly/csug-tutoring
- League of Legends LAN Party:
  - Sat 2/2 @ 2PM in CSB 209
  - $2 to benefit Big Brothers Big Sisters
- bit.ly/URLoL
ULW

Topics due to me by tomorrow
First draft due Mar 1
Questions, more feedback, etc.?
Get in touch early!
Project 1

- Code posted on BB
- Everyone should have it working
Local Search
States + Actions + Transition Model
= State Space

The set of all states reachable from the initial state by some sequence of actions
Solution graphSearch(Problem p) {
    Set<Node> frontier = new Set<Node>(p.getInitialState());
    Set<Node> explored = new Set<Node>();
    while (true) {
        if (frontier.isEmpty()) {
            return null;
        }
        Node node = frontier.selectOne();
        if (p.isGoalState(node.getState())) {
            return node.getSolution();
        }
        explored.add(node);
        for (Node n : node.expand()) {
            if (!explored.contains(n)) {
                frontier.add(n);
            }
        }
    }
}
Search Strategies

Uninformed

- No additional information about states

Informed (Heuristic)

- Can identify “promising” states
## Search Strategies

<table>
<thead>
<tr>
<th>Complet</th>
<th>BFS</th>
<th>DFS</th>
<th>IDS</th>
<th>Greedy</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓†</td>
</tr>
<tr>
<td>Optimal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓†</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^d)$</td>
<td>$O(b^m)$</td>
<td>$O(b^d)$</td>
<td>$O(b^m)$</td>
<td>$O(b^{ed})$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^d)$</td>
<td>$O(bm)$</td>
<td>$O(bd)$</td>
<td>$O(b^m)$</td>
<td>$O(b^d)$</td>
</tr>
</tbody>
</table>

* If step costs are identical
† With an admissible heuristic
Systematic Search

• Enumerates paths from initial state
• Records what alternatives have been explored at each point in the path

Good: Systematic $\rightarrow$ Exhaustive

Bad: Exponential time and/or space
Local Search
N-Queens as State-Space Search Problem

- State
- Actions
- Transition Model
- Initial State
- Goal State(s)/Test
- Step costs
Local Search

• Evaluates and modifies a small number of current states

• Does not record history of search (paths, explored set, etc.)

Good: Very little (constant) memory

Bad: May not explore all alternatives

=> Incomplete
State localSearch(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {

        if (p.isGoalState(node.getState())) {
            return node.getState();
        }

        for (Node n : node.expand()) {

        }
    }
}

State localSearch(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {

        if (p.isGoalState(node.getState())) {
            return node.getState();
        }

        for (Node n : node.expand()) {
            ???
        }
    }
}
Solution graphSearch(Problem p) {
    Set<Node> frontier = new Set<Node>(p.getInitialState());
    Set<Node> explored = new Set<Node>();
    while (true) {
        if (frontier.isEmpty()) {
            return null;
        }
        Node node = frontier.selectOne();
        if (p.isGoalState(node.getState())) {
            return node.getSolution();
        }
        explored.add(node);
        for (Node n : node.expand()) {
            if (!explored.contains(n)) {
                frontier.add(n);
            }
        }
    }
}

State localSearch(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {
        if (p.isGoalState(node.getState())) {
            return node.getState();
        }
        for (Node n : node.expand()) {
            if (p.value(n) >= p.value(node)) {
                node = n;
            }
        }
    }
}
State localSearch(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {
        if (p.isGoalState(node.getState())) {
            return node.getState();
        }

        for (Node n : node.expand()) {
            if (p.value(n) >= p.value(node)) {
                node = n;
            }
        }
    }
}
State localSearch(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {
        if (p.isGoalState(node.getState())) {
            return node.getState();
        }
        Node next = null;
        for (Node n : node.expand()) {
            if (p.value(n) >= p.value(node)) {
                next = n;
            }
        }
        if (next == null) {
            return node.getState();
        } else {
            node = next;
        }
    }
}
State localSearch(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {
        Node next = null;
        for (Node n : node.expand()) {
            if (p.value(n) >= p.value(node)) {
                next = n;
            }
        }
        if (next == null) {
            return node.getState();
        } else {
            node = next;
        }
    }
}
Hill-climbing Search

- Move through state space in the direction of increasing value ("uphill")

State-space landscape
State: $[r_0, \ldots, r_7]$

Action: $<i, r_i>$

$h(n) = \# \text{ of pairs of queens attacking each other}$
\[ h(n) = 17 \]
\[ h(n) = 17 \]
\[ h(n) = 12 \]
State hillClimb(Problem p) {
    Node node = new Node(p.getInitialState());
    while (true) {
        Node next = null;
        for (Node n : node.expand()) {
            if (p.value(n) >= p.value(node)) {
                next = n;
            }
        }
        if (next == null) {
            return node.getState();
        } else {
            node = next;
        }
    }
}
\[ h(n) = 1 \]
current state

objective function

shoulder

global maximum

local maximum

“flat” local maximum

state space
If at first you don’t succeed, try again.
State randomRestart(Problem p) {
    while (true) {
        p.setInitialState(new random State);
        State solution = hillClimb(p);
        if (p.isGoal(solution)) {
            return solution;
        }
    }
}

Does it work? Yes (but)

How well does it work?
Prob of success = $p$  Expected # of tries = $1/p$
  = 0.14  $\approx 7$
State randomRestart(Problem p) {
    while (true) {
        p.setInitialState(new random State);
        State solution = hillClimb(p);
        if (p.isGoal(solution)) {
            return solution;
        }
    }
}
Randomness in Search

Pure random walk
Complete, but horribly slow

Greedy local search
Incomplete, but fast
Simulated Annealing

• Follow landscape down towards global minimum of state cost function

• Occasionally allow an upward move ("shake") to get out of local minima

• Don’t shake so hard that you bounce out of global minimum
Annealing

• A heat treatment that alters the microstructure of a material causing changes in properties such as strength and hardness and ductility

• High temp => Atoms jumping around

• Low temp => Atoms settle into position
State simulatedAnnealing(Problem p, Schedule schedule) {
    Node node = new Node(p.getInitialState());
    for (t=1; true; t++) {
        Number T = schedule(t);
        if (T == 0) {
            return node;
        }
        Node next = randomly selected successor of node
        Number deltaE = p.cost(node) - p.cost(next);
        if (deltaE > 0 || Math.exp(-deltaE/T) > new Random(1)) {
            node = next;
        }
    }
}

\[
e = \frac{\Delta E}{T}
\]
$y = e^{-\Delta E}$
Simulated Annealing

Complete? No.

Optimal? No.

“But if the schedule lowers $T$ slowly enough, simulated annealing will find a global minimum with probability approaching one.”
Local Search

- Evaluates and modifies a small number of current states
- Does not record history of search

Good: Very little (constant) memory

Bad: May not explore all alternatives

=> Incomplete
Local Beam Search

• During hill-climbing:
  • Keep track of $k$ states rather than just one
  • At each step, generate all successors of all $k$ states ($k^*b$ of them)
  • Keep the most promising $k$ of them
Local Search
Parallel Local Search
Parallel Local Search

-8  -5  -7  -2

4   6   9   1

-3  3   0   -9
Local Beam Search
Local Beam Search
“And this report just in. ... Apparently, the grass is greener on the other side.”
Local Beam Search

- During hill-climbing:
  - Keep track of $k$ states rather than just one
  - At each step, generate all successors of all $k$ states ($k^b$ of them)
  - Keep the most promising $k$ of them
Mitosis

DNA replication → Mitosis → Two diploid cells
Meiosis

Interphase

Meiosis I

Homologous Chromosomes

Daughter Nuclei

Meiosis II

Daughter Nuclei II

Nuclei II

Wikipedia
Genetic Algorithms

• Start with \( k \) random states
• Select pairs of states and have them “mate” to produce “offspring”
• Most fit (highest-scoring) individuals reproduce more often
Genetic Algorithms

• States encoded as “chromosomes” (linear sequences, a.k.a. strings)

• During mating:
  • Crossover: swap chunks of code
  • Mutation: randomly change bits of code
Genetic Algorithms

• States encoded as “chromosomes” (linear sequences, a.k.a. strings)

• During mating:
  • Crossover: swap chunks of code
  • Mutation: randomly change bits of code
GA Summary

• A version of stochastic local beam search with a special way to generate successor states (motivated by a naive biology analogy)

• “Much work remains to be done to identify the conditions under which genetic algorithms perform well.”
- **Evaluates and modifies a small number of current states**
- **Does not record history of search**

**Hill-climbing**

**Good:** Very little (constant) memory

**Bad:** May not explore all alternatives

=> Incomplete

**Local Beam Search**

**Simulated Annealing**

**Genetic Algorithms**

- Evaluates and modifies a small number of current states
- Does not record history of search

**Good:** Very little (constant) memory

**Bad:** May not explore all alternatives

=> Incomplete
For next time:
AIMA 5.0–5.2.2

Quiz?

Project 1: Get Going!