Admin

- Q3
  - mean: 26.4
  - median: 27
- Final projects
  - proposals looked pretty good
  - start working
    - plan out exactly what you want to accomplish
    - make sure you have all the data, etc. that you need
    - status 1 still technically due 11/24, but can turn in as late as 11/27
    - status 2 due 12/2 (one day later)

Informed Search
CS457
David Kauchak
Fall 2011

Some material used from:
Sara Owsley Sood and others

Search algorithms

- Last time: search problem formulation
  - state
  - transitions (actions)
  - initial state
  - goal state
  - costs
- Now we want to find the solution!
- Use search techniques
  - Start at the initial state and search for a goal state
- What are candidate search techniques?
  - BFS
  - DFS
  - Uniform-cost search
  - Depth limited DFS
  - Depth-first iterative deepening

Finding the path: Tree search algorithms

- Basic idea:
  - keep a set of nodes to visit next (frontier)
  - pick a node from this set
  - check if it’s the goal state
  - if not, expand out adjacent nodes and repeat

```python
def treeSearch(start):
    add start to the frontier
    while frontier isn’t empty:
        get the next node from the frontier
        if node contains goal state:
            return solution
        else:
            expand node and add resulting nodes to frontier
```
BFS and DFS

How do we get BFS and DFS from this?

def treeSearch(start):
    add start to the frontier
    while frontier isn't empty:
        get the next node from the frontier
        if node contains goal state:
            return solution
        else:
            expand node and add resulting nodes to frontier

Breadth-first search

• Expand shallowest unexpanded node
• Nodes are expanded a level at a time (i.e. all nodes at a given depth)

• Implementation:
  – frontier is a FIFO (queue), i.e., new successors go at end

Depth-first search

• Expand deepest unexpanded node

• Implementation:
  – frontier = LIFO (stack), i.e., put successors at front

Search algorithm properties

• Time (using Big-O)
• Space (using Big-O)
• Complete
  – If a solution exists, will we find it?
• Optimal
  – If we return a solution, will it be the best/optimal solution

• A divergence from data structures
  – we generally won’t use V and E to define time and space. Why?
    • Often V and E are infinite (or very large relative to solution!)  
    • Instead, we often use the branching factor (b) and depth of solution (d)
Activity

• Analyze DFS and BFS according to:
  – time,
  – space,
  – completeness
  – optimality
    (for time and space, analyze in terms of \( b, d \) and \( m \) (max depth); for complete and optimal - simply YES or NO)
  – Which strategy would you use and why?
• Brainstorm improvements to DFS and BFS

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1100</td>
<td>.11 sec</td>
<td>1 MB</td>
</tr>
<tr>
<td>4</td>
<td>111,100</td>
<td>11 sec</td>
<td>106 MB</td>
</tr>
<tr>
<td>6</td>
<td>(10^7)</td>
<td>19 min</td>
<td>10 GB</td>
</tr>
<tr>
<td>8</td>
<td>(10^9)</td>
<td>31 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>(10^{11})</td>
<td>129 days</td>
<td>101 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>(10^{13})</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>(10^{15})</td>
<td>3,523 years</td>
<td>1 exabyte</td>
</tr>
</tbody>
</table>

DFS

• Time: \( O(b^m) \)
• Space: \( O(bm) \)
• Complete = YES, if space is finite (and no circular paths), NO otherwise
• Optimal = NO

BFS

• Time: \( O(b^d) \)
• Space: \( O(b^d) \)
• Complete = YES
• Optimal = YES if action costs are fixed, NO otherwise
Problems with BFS and DFS

• BFS
  – doesn’t take into account costs
  – memory!

• DFS
  – doesn’t take into account costs
  – not optimal
  – can’t handle infinite spaces
  – loops

Uniform-cost search

• Expand unexpanded node with the smallest path cost, \( g(x) \)

• Implementation:
  – \( \text{frontier} \) = priority queue ordered by path cost
  – similar to Dijkstra’s algorithm
  – Equivalent to breadth-first if step costs all equal

Uniform-cost search

• Time? and Space?
  – dependent on the costs and optimal path cost, so cannot be represented in terms of \( b \) and \( d \)
  – Space will still be expensive (e.g. take uniform costs)

• Complete?
  – YES, assuming costs > 0

• Optimal?
  – Yes, assuming costs > 0

• This helped us tackle the issue of costs, but still going to be expensive from a memory standpoint!

Ideas?

Can we combined the optimality and completeness of BFS with the memory of DFS?

+ =

= Superman

= Superman
Depth limited DFS

• DFS, but with a depth limit $L$ specified
  – nodes at depth $L$ are treated as if they have no successors
  – we only search down to depth $L$
• Time?
  – $O(b^L)$
• Space?
  – $O(bL)$
• Complete?
  – No, if solution is longer than $L$  
• Optimal
  – No, for same reasons DFS isn’t

Ideas?

Iterative deepening search

For depth 0, 1, ..., $\infty$
  run depth limited DFS
  if solution found, return result

• Blends the benefits of BFS and DFS
  – searches in a similar order to BFS
  – but has the memory requirements of DFS
• Will find the solution when $L$ is the depth of the shallowest goal

Iterative deepening search $L=0$
Iterative deepening search $L = 1$

Iterative deepening search $L = 2$

Iterative deepening search $L = 3$

Time?

- $L = 0$: 1
- $L = 1$: 1 + b
- $L = 2$: 1 + b + b^2
- $L = 3$: 1 + b + b^2 + b^3
- ...
- $L = d$: 1 + b + b^2 + b^3 + ... + b^d
- Overall:
  - $d(1) + (d-1)b + (d-2)b^2 + (d-3)b^3 + ... + b^d$
  - $O(b^d)$
  - the cost of the repeat of the lower levels is subsumed by the cost at the highest level.
Properties of iterative deepening search

- **Space?**
  - $O(bd)$
- **Complete?**
  - Yes
- **Optimal?**
  - Yes, if step cost = 1

Uninformed search strategies

- **Uninformed** search 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

Repeated states

```python
def treeSearch(start):
    add start to the frontier
    while frontier isn’t empty:
        get the next node from the frontier
        if node contains goal state:
            return solution
        else:
            expand node and add resulting nodes to frontier
```

What is the impact of repeated states?

- **Complete?**
  - Yes
  - Yes
  - No
  - No
  - Yes
- **Space**
  - $O(b^{d+1})$
  - $O(b^{d+1}d)$
  - $O(b^n)$
  - $O(n)$
  - $O(bd)$
- **Optimal?**
  - Yes
  - Yes
  - No
  - No
  - Yes

Summary of algorithms
Can make problems seem harder

Graph search implications?
- We’re keeping track of all of the states that we’ve previously seen
- For problems with lots of repeated states, this is a huge time savings
- The tradeoff is that we blow-up the memory usage
  - Space graphDFS?
    - $O(b^d)$
  - Something to think about, but in practice, we often just use the tree approach

Graph search
- Keep track of nodes that have been visited (explored)
- Only add nodes to the frontier if their state has not been seen before

Solution?

8-puzzle revisited
- The average depth of a solution for an 8-puzzle is 22 moves
- What do you think the average branching factor is?
  - ~3 (center square has 4 options, corners have 2 and edges have 3)
- An exhaustive search would require $3^{22} \approx 3.1 \times 10^{10}$ states
  - BFS: 10 terabytes of memory
  - DFS: 8 hours (assuming one million nodes/second)
  - IDS: ~9 hours
- Can we do better?

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
What would the search algorithms do?

BFS and IDS

We'd like to bias search towards the actual solution.

Ideas?
Informed search

- Order the frontier based on some knowledge of the world that estimates how "good" a node is
  - \( f(n) \) is called an evaluation function

- Best-first search
  - rank the frontier based on \( f(n) \)
  - take the most desirable state in the frontier first
  - different search depending on how we define \( f(n) \)

```python
def treeSearch(start):
    add start to the frontier
    while frontier isn’t empty:
        get the next node from the frontier
        if node contains goal state:
            return solution
        else:
            expand node and add resulting nodes to frontier
```

Heuristic function: \( h(n) \)

- An estimate of how close the node is to a goal
- Uses domain-specific knowledge

- Examples
  - Map path finding?
    - straight-line distance from the node to the goal ("as the crow flies")
  - 8-puzzle?
    - how many tiles are out of place
  - Missionaries and cannibals?
    - number of people on the starting bank

Greedy best-first search

- \( f(n) = h(n) \)
  - rank nodes by how close we think they are to the goal

Heuristic

Merriam-Webster’s Online Dictionary

Heuristic (pron. 'hyu-ris-ik-i): adj. [from Greek heuriskein to discover.] involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods

The Free On-line Dictionary of Computing (15Feb98)

heuristic 1. <programming> A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee feasible solutions and are often used with no theoretical guarantee. 2. <algorithm> approximation algorithm.

Heuristic function: \( h(n) \)

- An estimate of how close the node is to a goal
- Uses domain-specific knowledge

- Examples
  - Map path finding?
    - straight-line distance from the node to the goal ("as the crow flies")
  - 8-puzzle?
    - how many tiles are out of place
  - Missionaries and cannibals?
    - number of people on the starting bank

Greedy best-first search

- \( f(n) = h(n) \)
  - rank nodes by how close we think they are to the goal

Arad to Bucharest
Greedy best-first search

Is this right?
Problems with greedy best-first search

• Time?
  – $O(b^m)$ – but can be much faster

• Space
  – $O(b^m)$ – have to keep them in memory to rank

• Complete?
  – Graph search, yes
  – Tree search, no

Problems with greedy best-first search

• Complete?
  – Graph search, yes
  – Tree search, no

• Optimal?
Problems with greedy best-first search

- Optimal?
  - no, as we just saw in the map example

Sometimes it's too greedy

What is the problem?

A* search

- Idea:
  - don’t expand paths that are already expensive
  - take into account the path cost!
- \[ f(n) = g(n) + h(n) \]
  - \( g(n) \) is the path cost so far
  - \( h(n) \) is our estimate of the cost to the goal
- \( f(n) \) is our estimate of the total path cost to the goal through \( n \)
A* search

Admissible heuristics

- A heuristic function is admissible if it never overestimates
  - if $h^*(n)$ is the actual distance to the goal
    - $h(n) \leq h^*(n)$
- An admissible heuristic is optimistic (it always thinks the goal is closer than it actually is)
- Is the straight-line distance admissible?

A* properties

- Time
  - depends on heuristic, but generally exponential
- Space
  - exponential (have to keep all the nodes in memory/frontier)
- Complete
  - YES
- Optimal
  - YES, if the heuristic is admissible
  - Why?
    - If we could overestimate, then we could find (that is remove from the queue) a goal node that was suboptimal because our estimate for the optimal goal was too large

A point of technicality

- Technically if the heuristic isn’t admissible, then the search algorithm that uses $f(n) = g(n) + h(n)$ is call “Algorithm A”
- A* algorithm requires that the heuristic is admissible
- That said, you’ll often hear the later referred to as A*
- Algorithm A is not optimal
Admissible heuristics

- 8-puzzle
  - $h_1(n)$ = number of misplaced tiles?
  - $h_2(n)$ = manhattan distance?

<table>
<thead>
<tr>
<th>Tile</th>
<th>$h_1$ = 7</th>
<th>$h_2$ = 8</th>
<th>admissible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dominance

- Given two admissible heuristic functions
  - if $h_i(n)$ $\geq$ $h_j(n)$ for all $n$
  - then $h_i(n)$ dominates $h_j(n)$

- A dominant function is always better. Why?
  - It always give a better (i.e. closer) estimate to the actual path cost, without going over

- What about?
  - $h_1(n)$ = number of misplaced tiles
  - $h_2(n)$ = manhattan distance

<table>
<thead>
<tr>
<th>Depth of solution</th>
<th>IDS</th>
<th>$A^*(h_1)$</th>
<th>$A^*(h_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
</tr>
<tr>
<td>14</td>
<td>539</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1301</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3056</td>
<td>363</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>7276</td>
<td>676</td>
<td></td>
</tr>
</tbody>
</table>

average nodes expanded for 8-puzzle problems
Combining heuristics

• Sometimes, we have multiple admissible heuristics, but none dominates
• What then?
  – We can take the max of all the heuristics!
• Why?
  – Since they’re all admissible, we know none overestimate
  – taking the max gives us a closer/better estimate
  – overall, a better heuristic function

Relaxed problems

• A problem with fewer restrictions on the actions is called a relaxed problem
• The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
• How might you create a relaxed problem for the 8-puzzle?
  – a tile can move anywhere (same as \( h_1(n) \))
  – a tile can move to any adjacent square (same as \( h_2(n) \)