Introduction

- Ch.3 searches – good building blocks for learning about search
- But vastly inefficient eg:
- Can we do better?

<table>
<thead>
<tr>
<th>Time</th>
<th>Breadth</th>
<th>Depth</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B^D</td>
<td>B^M</td>
<td>&gt;B^D (?)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space</th>
<th>Breadth</th>
<th>Depth</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B^D</td>
<td>B^M</td>
<td>&gt;B^D (?)</td>
</tr>
</tbody>
</table>

| Optimal? | Y       | N     | Y       |
| Complete?| Y       | N     | Y       |
(Quick Partial) **Review**

- Previous algorithms differed in how to select next node for expansion eg:
  - Breadth First
    - Fringe nodes sorted old -> new
  - Depth First
    - Fringe nodes sorted new -> old
  - Uniform cost
    - Fringe nodes sorted by path cost: small -> big
- Used little (no) “external” domain knowledge

---

**Overview**

- Heuristic Search
  - Best-First Search Approach
    - Greedy
    - A*
  - Heuristic Functions
- Local Search and Optimization
  - Hill-climbing
  - Simulated Annealing
  - Local Beam
  - Genetic Algorithms
Informed Searching

- An *informed search* strategy uses knowledge beyond the definition of the problem
- The knowledge is embodied in an *evaluation function* $f(n)$

Best-First Search

- An algorithm in which a node is selected for expansion based on an evaluation function $f(n)$
  - Fringe nodes ordered by $f(n)$
  - Traditionally the node with the lowest evaluation function is selected
  - Not an accurate name...expanding the best node first would be a straight march to the goal.
  - Choose the node that *appears* to be the best
Best-First Search

- Remember: Uniform cost search
  - $F(n) = g(n)$
- Best-first search:
  - $F(n) = h(n)$
- Later, a-star search:
  - $F(n) = g(n) + h(n)$

Best-First Search (cont.)

- Some BFS algorithms also include the notion of a heuristic function $h(n)$
- $h(n) =$ estimated cost of the cheapest path from node $n$ to a goal node
- Best way to include informed knowledge into a search
- Examples:
  - How far is it from point A to point B
  - How much time will it take to complete the rest of the task at current node to finish
Greedy Best-First Search

- Expands node estimated to be closest to the goal
  - \( f(n) = h(n) \)
- Consider the route finding problem.
  - Can we use additional information to avoid costly paths that lead nowhere?
  - Consider using the straight line distance (SLD)
Route Finding: Greedy Best First

- **Arad**: $f(n) = 366$

Route Finding: Greedy Best First

- **Arad**: $f(n) = 366$
- **Sibiu**: 253
- **Timisoara**: 329
- **Zerind**: 374
Route Finding: Greedy Best First

Arad
  f(n) = 366
  Sibiu 253
  Timisoara 329
  Zerind 374
  Arad 366
  Fagaras 176
  Oradea 380
  Rimnicu Vilcea 193

Route Finding: Greedy Best First

Arad
  f(n) = 366
  Sibiu 253
  Timisoara 329
  Zerind 374
  Arad 366
  Fagaras 176
  Oradea 380
  Rimnicu Vilcea 193
  Bucharest 0
  Sibiu 253
Exercise

So is Arad->Sibiu->Fagaras->Bucharest optimal?

Greedy Best-First Search

- Not optimal.
- Not complete.
  - Could go down a path and never return to try another.
  - e.g., Iasi → Neamt → Iasi → Neamt → ...
- Space Complexity
  - $O(b^m)$ – keeps all nodes in memory
- Time Complexity
  - $O(b^m)$ (but a good heuristic can give a dramatic improvement)
Heuristic Functions

- Example: 8-Puzzle
  - Average solution cost for a random puzzle is 22 moves
  - Branching factor is about 3
    - Empty tile in the middle -> four moves
    - Empty tile on the edge -> three moves
    - Empty tile in corner -> two moves
  - $3^{22}$ is approx $3.1 \times 10^{10}$
    - Get rid of repeated states
    - 181,440 distinct states

- $h_1$ = number of misplaced tiles
- $h_2$ = sum of distances of tiles to goal position.
Heuristic Functions

- $h_1 = 7$
- $h_2 = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

Admissible Heuristics

- A heuristic function $h(n)$ is **admissible** if it **never overestimates** the cost to reach the goal from $n$.
- Another property of heuristic functions is **consistency**
  - $h(n) \leq c(n,a,n') + h(n')$ where:
    - $c(n,a,n')$ is the cost to get to $n'$ from $n$ using action $a$.
    - Consistent $h(n)$ → the values of $f(n)$ along any path are non-decreasing.
- Graph search is optimal if $h(n)$ is consistent.
Heuristic Functions

- Is $h_1$ (#of displaced tiles)
  - admissible?
  - consistent?
- Is $h_2$ (Manhattan distance)
  - admissible?
  - consistent?

Dominance

- If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible)
  - then $h_2$ dominates $h_1$
  - $h_2$ is better for search

Typical search costs (average number of nodes expanded):

- $d=12$  
  - IDS = 3,644,035 nodes
  - $A^*(h_1) = 227$ nodes
  - $A^*(h_2) = 73$ nodes

- $d=24$  
  - IDS = too many nodes
  - $A^*(h_1) = 39,135$ nodes
  - $A^*(h_2) = 1,641$ nodes
Heuristic Functions

- Heuristics are often obtained from *relaxed problem*
  - Simplify the original problem by removing constraints
  - The cost of an optimal solution to a relaxed problem is an admissible heuristic.

8-Puzzle

- **Original**
  - A tile can move from \( A \) to \( B \) if \( A \) is horizontally or vertically *adjacent* to \( B \) and \( B \) is *blank*.

- **Relaxations**
  - Move from \( A \) to \( B \) if \( A \) is adjacent to \( B \) (remove "blank")
    - \( h_2 \) by moving each tile in turn to destination
  - Move from \( A \) to \( B \) (remove "adjacent" and "blank")
    - \( h_1 \) by simply moving each tile directly to destination
How to Obtain Heuristics?

- Ask the domain expert (if there is one)
- Solve example problems and generalize your experience on which operators are helpful in which situation (particularly important for state space search)
- Try to develop sophisticated evaluation functions that measure the closeness of a state to a goal state (particularly important for state space search)
- Run your search algorithm with different parameter settings trying to determine which parameter settings of the chosen search algorithm are “good” to solve a particular class of problems.
- Write a program that selects “good parameter” settings based on problem characteristics (frequently very difficult) relying on machine learning

A* Search

- The greedy best-first search does not consider how costly it was to get to a node.
  - \( f(n) = h(n) \)
- Idea: avoid expanding paths that are already expensive
- Combine \( g(n) \), the cost to reach node \( n \), with \( h(n) \)
  - \( f(n) = g(n) + h(n) \)
  - estimated cost of cheapest solution through \( n \)
A* Search

- When $h(n) = $ actual cost to goal
  - Only nodes in the correct path are expanded
  - Optimal solution is found
- When $h(n) < $ actual cost to goal
  - Additional nodes are expanded
  - Optimal solution is found
- When $h(n) > $ actual cost to goal
  - Optimal solution can be overlooked

A* Search

- Complete
  - Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time
  - Exponential in $[\text{relative error of } h \times \text{length of soln}]$
  - The better the heuristic, the better the time
    - Best case $h$ is perfect, $O(d)$
    - Worst case $h = 0$, $O(b^d)$ same as BFS
- Space
  - Keeps all nodes in memory and save in case of repetition
  - This is $O(b^d)$ or worse
  - A* usually runs out of space before it runs out of time
- Optimal
  - Yes, cannot expand $f_{i+1}$ unless $f_i$ is finished
**Route Finding**

![Route Finding Diagram](image)

**A* Example**

<table>
<thead>
<tr>
<th>Straight-line distance to Bucharest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
</tr>
<tr>
<td>Bucharest</td>
</tr>
<tr>
<td>Craiova</td>
</tr>
<tr>
<td>Dobrogea</td>
</tr>
<tr>
<td>Eforie</td>
</tr>
<tr>
<td>Fagaras</td>
</tr>
<tr>
<td>Giurgiu</td>
</tr>
<tr>
<td>Hurzova</td>
</tr>
<tr>
<td>Iași</td>
</tr>
<tr>
<td>Lugoj</td>
</tr>
<tr>
<td>Mehadia</td>
</tr>
<tr>
<td>Neamț</td>
</tr>
<tr>
<td>Oradea</td>
</tr>
<tr>
<td>Pitești</td>
</tr>
<tr>
<td>Râmnicu Vâlcea</td>
</tr>
<tr>
<td>Sibiu</td>
</tr>
<tr>
<td>Timișoara</td>
</tr>
<tr>
<td>Urziceni</td>
</tr>
<tr>
<td>Vâlcea</td>
</tr>
<tr>
<td>Zerind</td>
</tr>
</tbody>
</table>
### A* Search

- **Arad**: \( f(n) = 0 + 366 \)
- **Sibiu**: 393
- **Timisoara**: 447
- **Zerind**: 449
- **Arad**: 646
- **Fagaras**: 415
- **Oradea**: 671
- **Rimnicu Vilcea**: 413

**Things are different now!**

### A* Search Continued

- **Arad**: 646
- **Fagaras**: 415
- **Oradea**: 671
- **Rimnicu Vilcea**: 413
- **Bucharest**: 450
- **Sibiu**: 591
- **Craiova**: 526
- **Pitesti**: 417
- **Sibiu**: 553
- **Bucharest**: 418
- **Craiova**: 615
- **Rimnicu Vilcea**: 607
A* Properties review

- Complete
  - Yes, unless there are infinitely many nodes with \( f \leq f(G) \)
- Time
  - Exponential in \([\text{relative error of } h \times \text{length of soln}]\)
  - The better the heuristic, the better the time
    - Best case \( h \) is perfect, \( O(d) \)
    - Worst case \( h = 0, O(b^d) \) same as BFS
- Space
  - Keeps all nodes in memory and save in case of repetition
  - This is \( O(b^d) \) or worse
  - A* usually runs out of space before it runs out of time
- Optimal
  - Yes, cannot expand \( f_{i+1} \) unless \( f_i \) is finished

A* Exercise

<table>
<thead>
<tr>
<th>1st Expansion</th>
<th>City</th>
<th>P+Val</th>
<th>Fringe</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>City</th>
<th>Carnegie</th>
<th>Unknowntown</th>
</tr>
</thead>
<tbody>
<tr>
<td>P+Val</td>
<td>16</td>
<td>54</td>
</tr>
<tr>
<td>Fringe</td>
<td></td>
<td>Fringe</td>
</tr>
</tbody>
</table>
A* Exercise

1st Expansion

2nd Expansion

CS 2710 – Informed Search
A* Search; complete

- A* is complete.
  
  A* builds search “bands” of increasing $f(n)$
  At all points $f(n) < C^*$
  Eventually we reach the “goal contour”

- Optimally efficient
- Most times exponential growth occurs

Memory Bounded Heuristic Search

- Ways of getting around memory issues of A*:
  - IDA* (iterative deepening algorithm)
    - Cutoff = $f(n)$ instead of depth
  - Recursive Best First Search
    - Mimic standard BFS, but use linear space!
    - Keeps track of best $f(n)$ from alternate paths
RBFS

- F-limit: keeps track of the f-value of the best alternative path available
- F-value replacement: as the recursion unwinds, replaces f-value of each node with the best f-value of its children.

RBFS Exercise

<table>
<thead>
<tr>
<th>1st Expansion</th>
<th>City</th>
<th>Pgh HVal</th>
<th>F-Limit</th>
<th>RBFS</th>
<th>City</th>
<th>Carnegie HVal</th>
<th>F-Val</th>
<th>F-Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Carnegie</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F-Val 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F-Limit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RBFS Exercise

2nd Expansion

<table>
<thead>
<tr>
<th>City</th>
<th>Pgh</th>
<th>H-Val</th>
<th>F-Val</th>
<th>F-Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carnegie</td>
<td>4</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniontown</td>
<td>3</td>
<td>14</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wash.</td>
<td>0</td>
<td>20</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H-Val</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F-Val</td>
<td>11</td>
<td>5</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>F-Limit</td>
<td>12</td>
<td>9</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

CS 2710 – Informed Search 41

RBFS Exercise

3rd Expansion

<table>
<thead>
<tr>
<th>City</th>
<th>Pgh</th>
<th>H-Val</th>
<th>F-Val</th>
<th>F-Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carnegie</td>
<td>4</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniontown</td>
<td>3</td>
<td>14</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wash.</td>
<td>0</td>
<td>20</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>City</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H-Val</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F-Val</td>
<td>11</td>
<td>5</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>F-Limit</td>
<td>12</td>
<td>9</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

CS 2710 – Informed Search 42
RBFS Review

- F-limit: keeps track of the f-value of the best alternative path available
- F-value replacement: as the recursion unwinds, replaces f-value of each node with the best f-value of its children.
- Disad’s: excessive node regeneration from recursion
- Too little memory! → use memory-bounded approaches
  - Cutoff when memory bound is reached and other constraints

Local Search / Optimization

- Idea is to find the best state.
- We don’t really care how to get to the best state, just that we get there.
- The best state is defined according to an objective function
  - Measures the “fitness” of a state.
- Problem: Find the optimal state
  - The one that maximizes (or minimizes) the objective function.
**State Space Landscapes**

- Objective Function
- Global max
- Local max
- Shoulder

**Problem Formulation**

- Complete-state formulation
  - Start with an approximate solution and perturb
- n-queens problem
  - Place n queens on a board so that no queen is attacking another queen.
Problem Formulation

- Initial State: n queens placed randomly on the board, one per column.
- Successor function: States that obtained by moving one queen to a new location in its column.
- Heuristic/objective function: The number of pairs of attacking queens.

n-Queens

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Local Search Algorithms

- Hill climbing
- Simulated annealing
- Local beam search
- Genetic Algorithms

Hill Climbing (or Descent)
Hill Climbing Pseudo-code

• "Like climbing Everest in thick fog with amnesia"

function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
neighbor ← a highest-valued successor of current
if VALUE[neighbor] > VALUE[current] then return STATE[current]
current ← neighbor

Hill Climbing Problems

Objective Function

State Space
n-Queens

What happens if we move 3rd queen?

Possible Improvements

- Stochastic hill climbing
  - Choose at random from uphill moves
  - Probability of move could be influenced by steepness
- First-choice hill climbing
  - Generate successors at random until one is better than current.
- Random-restart
  - Execute hill climbing several times, choose best result.
  - If $p$ is probability of a search succeeding, then expected number of restarts is $1/p$. 
Simulated Annealing

- Similar to stochastic hill climbing
  - Moves are selected at random
  - If a move is an improvement, accept
  - Otherwise, accept with probability less than 1.
- Probability gets smaller as time passes and by the amount of “badness” of the move.

Simulated Annealing Algorithm

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs
  problem, a problem
  schedule, a mapping from time to “temperature”
local variables
  current, a node
  next, a node
  T, a “temperature” controlling the probability of downward steps

current ← MAKE-NODE(INITIAL-STATE(problem))
for t ← 1 to go do
  T ← schedule(t)
  if T = 0 then return current
  next ← a randomly selected successor of current
  ΔE ← VALUE[next] - VALUE[current]
  if ΔE > 0 then current ← next
  else current ← next only with probability $e^{-ΔE/T}$
```

Success
Traveling Salesperson Problem

- Tour of cities
- Visit each one exactly once
- Minimize distance/cost/etc.

Local Beam Search

- Keep $k$ states in memory instead of just one
- Generate successors of all $k$ states
- If one is a goal, return the goal
- Otherwise, take $k$ best successors and repeat.
Local Beam Search

- Initial k states may not be diverse enough
  - Could have clustered around a local max.
- Improvement is *stochastic beam search*
  - Choose k states at random, with probability of choice an increasing function of its value.
Genetic Algorithms

- Variant of stochastic beam search
- Successor states are generated by combining two parent states
  - Hopefully improves diversity
- Start with k states, the population
- Each state, or individual, represented as a string over a finite alphabet (e.g. DNA)
- Each state is rated by a fitness function
- Select parents for reproduction using the fitness function

Taken from [http://www.cs.qub.ac.uk/~M.Sullivan/ga/ga_index.html](http://www.cs.qub.ac.uk/~M.Sullivan/ga/ga_index.html)
A Genetic Algorithm

```plaintext
function Genetic-Algorithm(population, fitness-FN) returns an individual
inputs: population, a set of individuals
        fitness-FN, a function that measures the fitness of an individual

repeat
    new_population ← empty set
    for i from 1 to size(population) do
        g ← Random-Selection(population, fitness-FN)
        y ← Random-Selection(population, fitness-FN)
        child ← Reproduce(g, y)
        if (small random probability) then child ← Mutate(child)
        add child to new_population
    population ← new_population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to fitness-FN

function Reproduce(x, y) returns an individual
inputs: x, y, parent individuals

γ ← Length(x)

return Append(Substring(x, 1, γ), Substring(y, γ + 1, n))
```

CS 2710 – Informed Search 63