Artificial Intelligence

Lecture 3
Vibhav Gogate
The University of Texas at Dallas

Some material courtesy of Luke Zettlemoyer, Dan Klein, Dan Weld, Alex Ihler and Stuart Russell
Review: Rational Agents

- An **agent** is an entity that *perceives* and *acts*.
- A **rational agent** selects actions that maximize its **utility function**.
- Characteristics of the **percepts**, **environment**, and **action space** dictate techniques for selecting rational actions.

Search -- the environment is: fully observable, single agent, deterministic, episodic, discrete
Reflex Agents

• Reflex agents:
  – Choose action based on current percept (and maybe memory)
  – Do not consider the future consequences of their actions
  – Act on how the world IS

• Can a reflex agent be rational?
Famous Reflex Agents

NEW!
Goal Based Agents

• Goal-based agents:
  – Plan ahead
  – Ask “what if”
  – Decisions based on (hypothesized) consequences of actions
  – Must have a model of how the world evolves in response to actions
  – Act on how the world WOULD BE
Search thru a Problem Space / State Space

- **Input:**
  - Set of states: the agent’s model of the world
  - Operators [and costs]: actions which the agent can take to move from one state to another
  - Start state
  - Goal state [test]: a goal is defined as a desirable state for an agent

- **Output:**
  - Path: start $\Rightarrow$ a state satisfying goal test
  - [May require shortest path] - Optimality
  - [Sometimes just need state passing test] - Solution
Example: Romania

• On holiday in Romania; currently in Arad.
• Flight leaves tomorrow from Bucharest

• **Formulate goal:**
  – be in Bucharest

• **Formulate problem:**
  – **states**: various cities
  – **actions**: drive between cities

• **Find solution:**
  – sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
Problem types

Classifying the environment:

- **Single agent/Multi-agent**
  - Previous problem was a Single agent problem

- **Deterministic / Stochastic**
  - Previous problem was deterministic: no new percepts were necessary, we can predict the future perfectly

- **Observable / Partially Observable / Unobservable**
  - Previous problem was observable: it knew the initial state, &c

- **Discrete / continuous**
  - Previous problem was discrete: we can enumerate all possibilities
State-Space Problem Formulation

A **problem** is defined by four items:

- **initial state** e.g., "at Arad"

- **actions** or **successor function** \( S(x) = \) set of action–state pairs
  - e.g., \( S(\text{Arad}) = \{<\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}>, \ldots \} \)

- **goal test**, (or goal state)
  e.g., \( x = "\text{at Bucharest}" \), \( \text{Checkmate}(x) \)

- **path cost** (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - \( c(x,a,y) \) is the **step cost**, assumed to be \( \geq 0 \)

A **solution** is a sequence of actions leading from the initial state to a goal state
Abstraction/Modeling

Process of removing irrelevant detail to create an abstract representation: ``high-level'', ignores irrelevant details

• Definition of Abstraction:
• Navigation Example: how do we define states and operators?
  – First step is to abstract “the big picture”
    • i.e., solve a map problem
    • nodes = cities, links = freeways/roads (a high-level description)
    • this description is an abstraction of the real problem
  – Can later worry about details like freeway onramps, refueling, etc

• Abstraction is critical for automated problem solving
  – must create an approximate, simplified, model of the world for the computer to deal with: real-world is too detailed to model exactly
  – good abstractions retain all important details
Robot block world

• Given a set of blocks in a certain configuration,
• Move the blocks into a goal configuration.
• Example:
  – (c,b,a) $\rightarrow$ (b,c,a)
Operator Description

Effects of Moving a Block
The state-space graph

- **Graphs:**
  - nodes, arcs, directed arcs, paths

- **Search graphs:**
  - States are nodes
  - operators are directed arcs
  - solution is a path from start to goal

- **Problem formulation:**
  - Give an abstract description of states, operators, initial state and goal state.

- **Problem solving activity:**
  - Generate a part of the search space that contains a solution
Example: Simplified Pac-Man

• Input:
  – A state space
  – A successor function
  – A start state
  – A goal test

• Output:
Example: 8-Queens

Place as many queens as possible on the chess board without capture

- states?
- initial state?
- actions?
- goal test?
- path cost?
Example: 8-Queens

Place as many queens as possible on the chess board without capture

• **states?**  - any arrangement of n<=8 queens
• **initial state?** no queens on the board
• **actions?**  - add queen to any empty square
• **goal test?** 8 queens on the board, none attacked.
• **path cost?** 1 per move
The sliding tile problem

- **8-Puzzle**

- **States**
  - Configurations of the tiles

- How many states in an n-Puzzle?
  
  \[
  \frac{n!}{2} =
  \]

- 8-puzzle: 181,440 states
- 15-puzzle: 1.3 trillion
- 24-puzzle: \(10^{25}\)
The sliding tile problem

8-puzzle

- Start & Goal configurations:

```
2 8 3
1 6 4
7 ■ 5
```

```
1 2 3
4 5 6
7 8 ■
```

- Actions (successor function):

```
2 8 3
1 6 4
7 ■ 5
```

```
2 8 3
1 ■ 4
7 6 5
```

```
2 8 3
1 2 3
4 5 6
7 8 ■
```
The sliding tile problem

Start State

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Goal State

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
State Space Graphs

- State space graph:
  - Each node is a state
  - The successor function is represented by arcs
  - Edges may be labeled with costs
- We can rarely build this graph in memory (so we don’t)

Ridiculously tiny search graph for a tiny search problem
Search Strategies

• Blind Search
  • Depth first search
  • Breadth first search
  • Iterative deepening search
  • Uniform cost search

• Informed Search

• Constraint Satisfaction

• Adversary Search
Search Trees

• A search tree:
  – Start state at the root node
  – Children correspond to successors
  – Nodes contain states, correspond to PLANS to those states
  – Edges are labeled with actions and costs
  – For most problems, we can never actually build the whole tree
Example: Tree Search

State Graph:

What is the search tree?
We construct both on demand – and we construct as little as possible.

Each NODE in the search tree is an entire PATH in the problem graph.
Building Search Trees

• Search:
  – Expand out possible plans
  – Maintain a fringe of unexpanded plans
  – Try to expand as few tree nodes as possible
General Tree Search

function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end

• Important ideas:
  – Fringe
  – Expansion
  – Exploration strategy

• Main question: which fringe nodes to explore?

Detailed pseudocode is in the book!
**Strategy:** expand deepest node first

**Implementation:**
Fringe is a LIFO queue (a stack)
Review: Depth First Search

Expansion ordering:

\[(d, b, a, c, a, e, h, p, q, q, r, f, c, a, G)\]
Review: Breadth First Search

**Strategy:** expand shallowest node first

**Implementation:** Fringe is a FIFO queue
Review: Breadth First Search

Expansion order:

\((S,d,e,p,b,c,e,h,r,q,a,a,h,r,p,q,f,p,q,f,q,c,G)\)
Review: Search

• Search Problem formulation
  – Take a real world problem and formulate it as a search problem
  – **Remove irrelevant detail to create an abstract representation: ``high-level”**, ignores irrelevant details
  – State space, Successor function, Initial state, Goal test, Cost path

• Search = Method used to find a solution
Review: Search

- Tree Search algorithm (Fringe, Expansion Strategy)

```plaintext
function TREE-SEARCH( problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end
```

- DFS: Deepest node first
- BFS: Shallowest node first
Search Algorithm Properties

- Complete? Guaranteed to find a solution if one exists?
- Optimal? Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

Variables:

<table>
<thead>
<tr>
<th>$n$</th>
<th>Number of states in the problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>The maximum branching factor (the maximum number of successors for a state)</td>
</tr>
<tr>
<td>$C^*$</td>
<td>Cost of least cost solution</td>
</tr>
<tr>
<td>$d$</td>
<td>Depth of the shallowest solution</td>
</tr>
<tr>
<td>$m$</td>
<td>Max depth of the search tree</td>
</tr>
</tbody>
</table>
## DFS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## DFS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>No</td>
<td>No</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>
**DFS**

- Infinite paths make DFS incomplete...
  - How can we fix this? Path Checking
  - Check new nodes against path from S
- Infinite search spaces still a problem

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>No</td>
<td>No</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
</tbody>
</table>
**DFS**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS w/ Path Checking</td>
<td>Y if finite</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
</tbody>
</table>
### BFS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DFS</strong> w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td><strong>BFS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **d tiers**
  - 1 node
  - $b$ nodes
  - $b^2$ nodes
  - $b^d$ nodes
  - $b^m$ nodes
**BFS**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
</tbody>
</table>

---

- **d tiers**
- 1 node
- $b$ nodes
- $b^2$ nodes
- $b^d$ nodes
- $b^m$ nodes
Memory a Limitation?

• Suppose:
  • 4 GHz CPU
  • 6 GB main memory
  • 100 instructions / expansion
  • 5 bytes / node

• 400,000 expansions / sec
  • Memory filled in 300 sec ... 5 min
Comparisons

• When will BFS outperform DFS?

• When will DFS outperform BFS?
Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.
   ....and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>$Y^*$</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Iterative Deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>(O(b^m))</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>(Y^*)</td>
<td>(O(b^d))</td>
<td>(O(b^d))</td>
</tr>
</tbody>
</table>
Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>O(b^m)</td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td>Y</td>
<td>Y*</td>
<td>O(b^d)</td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>ID</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(bd)$</td>
</tr>
</tbody>
</table>
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.
Uniform Cost Search

Expand cheapest node first:
Fringe is a priority queue
Uniform Cost Search

Expansion order:

\((S, p, d, b, e, a, r, f, e, G)\)
Priority Queue Refresher

A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pq.push(key, value)</code></td>
<td>inserts (key, value) into the queue.</td>
</tr>
<tr>
<td><code>pq.pop()</code></td>
<td>returns the key with the lowest value, and removes it from the queue.</td>
</tr>
</tbody>
</table>

You can decrease a key’s priority by pushing it again.

Unlike a regular queue, insertions aren’t constant time, usually $O(\log n)$.

We’ll need priority queues for cost-sensitive search methods.
### Uniform Cost Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>w/ Path Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>UCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which goal will UCS find?
- Shallowest one?
- The goal with min cost?
### Uniform Cost Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>UCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **C***: Cost of Cheapest goal
- **ε**: Min Cost of an action

$C^*/\varepsilon$ tiers
# Uniform Cost Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS w/ Path Checking</td>
<td>Y</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y*</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>UCS</td>
<td>Y*</td>
<td>Y</td>
<td>$O(b^{C*/\varepsilon})$</td>
<td>$O(b^{C*/\varepsilon})$</td>
</tr>
</tbody>
</table>

\(C^{*/\varepsilon}\) tiers
Uniform Cost Issues

• Remember: explores increasing cost contours

• The good: UCS is complete and optimal!

• The bad:
  – Explores options in every “direction”
  – No information about goal location
Uniform Cost: Pac-Man

- Cost of 1 for each action
- Explores all of the states, but one
Search Heuristics

- Any *estimate* of how close a state is to a goal
- Designed for a particular search problem

- Examples: Manhattan distance, Euclidean distance
Greedy Search

Best first with \( f(n) = \) heuristic estimate of distance to goal
Greedy Search

Expand the node that seems closest...

What can go wrong?
Greedy Search

Expand the node that seems closest…

What can go wrong?
Greedy Search

• A common case:
  – Best-first takes you straight to the (wrong) goal

• Worst-case: like a badly-guided DFS in the worst case
  – Can explore everything
  – Can get stuck in loops if no cycle checking

• Like DFS in completeness (if finite # states w/ cycle checking)
A* Search

Hart, Nilsson & Rafael 1968

Best first search with \( f(n) = g(n) + h(n) \)

- \( g(n) \) = sum of costs from start to \( n \)
- \( h(n) \) = estimate of lowest cost path \( n \rightarrow \text{goal} \)
  
  \( h(\text{goal}) = 0 \)

If \( h(n) \) is admissible then A* is optimal
A* Example

Arad

366=0+366
A* Example
A* Example

- Arad
  - Fagaras: 415=239+176
  - Oradea: 671=291+380
  - Rimnicu Vîlcea

- Sibiu

- Timisoara: 447=118+329

- Zerind: 449=75+374
A* Example

Arad
646=280+366

Fagaras
415=239+176

Oradea
671=291+380

Rimnicu Vilcea

Craiova
526=366+160

Pitesti
417=317+100

Sibiu
553=300+253

Timisoara
447=118+329

Zerind
449=75+374
A* Example
A* Example

Diagram showing the A* algorithm with distances and costs for cities such as Arad, Sibiu, Fagaras, Oradea, Bucharest, Craiova, Pitesti, Sibiu, Timisoara, and Zerind.
Optimality of A*: Blocking

Notation:
• $g(n) = \text{cost to node } n$
• $h(n) = \text{estimated cost from } n \text{ to the nearest goal (heuristic)}$
• $f(n) = g(n) + h(n) = \text{estimated total cost via } n$
• $G^* : \text{a lowest cost goal node}$
• $G : \text{another goal node}$
Optimality of A*: Blocking

Proof:
• What could go wrong?
• We’d have to have to pop a suboptimal goal G off the fringe before G*

- This can’t happen:
  - For all nodes n on the best path to G*
    - \( f(n) < f(G) \)
  - So, G* will be popped before G

\[
\begin{align*}
    f(n) &= g(n) + h(n) \\
g(n) + h(n) &\leq g(G^*) \\
g(G^*) &< g(G) \\
g(G) &= f(G) \\
f(n) &< f(G)
\end{align*}
\]
Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics.

- Often, admissible heuristics are solutions to *relaxed problems*, where new actions are available.

- Inadmissible heuristics are often useful too (why?)
Creating Heuristics

8-puzzle:

- What are the states?
- How many states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?
8 Puzzle I

- Heuristic: Number of tiles misplaced

  \( h(\text{start}) = 8 \)

- Is it admissible?

<table>
<thead>
<tr>
<th>Average nodes expanded when optimal path has length…</th>
<th>…4 steps</th>
<th>…8 steps</th>
<th>…12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS</td>
<td>112</td>
<td>6,300</td>
<td>3.6 x 10^6</td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
</tbody>
</table>
8 Puzzle I

- Heuristic: Number of tiles misplaced
  - $h(\text{start}) = 8$

- Is it admissible?
  - Any tile that is out of place must be moved at least once

<table>
<thead>
<tr>
<th></th>
<th>Average nodes expanded when optimal path has length…</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>…4 steps</td>
</tr>
<tr>
<td>UCS</td>
<td>112</td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
</tr>
</tbody>
</table>
8 Puzzle II

• What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?

• Total Manhattan distance

• $h(\text{start}) = 3 + 1 + 2 + \ldots = 18$

– Admissible?

![Start State](image1)

![Goal State](image2)

Average nodes expanded when optimal path has length…

<table>
<thead>
<tr>
<th></th>
<th>…4 steps</th>
<th>…8 steps</th>
<th>…12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
<tr>
<td>MANHATTAN</td>
<td>12</td>
<td>25</td>
<td>73</td>
</tr>
</tbody>
</table>
8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total Manhattan distance
- \( h(\text{start}) = 18 \)
  - Admissible?
  - All any move can do is move one tile closer to the goal!

<table>
<thead>
<tr>
<th>TILES</th>
<th>13</th>
<th>39</th>
<th>227</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANHATTAN</td>
<td>12</td>
<td>25</td>
<td>73</td>
</tr>
</tbody>
</table>
How about using the *actual cost* as a heuristic?

- Would it be admissible?
- Would we save on nodes expanded?
- What’s wrong with it?

With A*: a trade-off between quality of estimate and work per node!
# Effectiveness of A* Search Algorithm

Average number of nodes expanded

<table>
<thead>
<tr>
<th>d</th>
<th>IDS</th>
<th>A*(h1)</th>
<th>A*(h2)</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>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>227</td>
<td>73</td>
</tr>
<tr>
<td>14</td>
<td>3473941</td>
<td>539</td>
<td>113</td>
</tr>
<tr>
<td>20</td>
<td>-------</td>
<td>7276</td>
<td>676</td>
</tr>
</tbody>
</table>

Average over 100 randomly generated 8-puzzle problems

- \( h_1 \) = number of tiles in the wrong position
- \( h_2 \) = sum of Manhattan distances
Trivial Heuristics, Dominance

- Dominance: \( h_a \geq h_c \) if
  \[
  \forall n : h_a(n) \geq h_c(n)
  \]

- Heuristics form a hierarchy or lattice:
  - Max of admissible heuristics is admissible
    \[
    h(n) = \max(h_a(n), h_b(n))
    \]

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic
A* Applications

• Pathing / routing problems
• Resource planning problems
• Robot motion planning
• Language analysis
• Machine translation
• Speech recognition
• ...
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work. Why?
Tree Search: Extra Work!

• Failure to detect repeated states can cause exponentially more work. Why?
Graph Search

• In BFS, for example, we shouldn’t bother expanding some nodes (which, and why?)
Graph Search

• In BFS, for example, we shouldn’t bother expanding some nodes (which, and why?)
Graph Search

• Idea: never **expand** a state twice

• How to implement:
  – Tree search + list of expanded states (closed list)
  – Expand the search tree node-by-node, but...
  – Before expanding a node, check to make sure its state is new

  ▪ Python trick: **store the closed list as a set, not a list**
  ▪ **Can graph search wreck completeness? Why/why not?**
  ▪ **How about optimality?**
Graph Search

- Very simple fix: never expand a state type twice

```python
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    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, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

- Can this wreck completeness? Why or why not?
- How about optimality? Why or why not?
A* Graph Search

State space graph

Will it find the optimal solution?
A* Graph Search Gone Wrong

State space graph

Search tree

S (0+2)

A (1+4)

B (1+1)

C (2+1)

C (3+1)

G (5+0)

G (6+0)
Optimality of A* Graph Search

• Consider what A* does:
  – Expands nodes in increasing total f value (f-contours)
    Reminder: $f(n) = g(n) + h(n) = \text{cost to } n + \text{heuristic}$
  – Proof idea: the optimal goal(s) has the lowest f value, so it must get expanded first

\begin{align*}
f &\leq 1 \\
f &\leq 2 \\
f &\leq 3
\end{align*}
Optimality of A* Graph Search

• Consider what A* does:
  – Expands nodes in increasing total f value (f-contours)
    Reminder: $f(n) = g(n) + h(n) = \text{cost to } n + \text{heuristic}$
  – Proof idea: the optimal goal(s) has the lowest f value, so it must get expanded first

There’s a problem with this argument. What are we assuming is true?
Consistency

- Wait, how do we know we expand in increasing f value?
- Couldn’t we pop some node \( n \), and find its child \( n' \) to have lower f value?
Consistency

- Wait, how do we know we expand in increasing f value?
- Couldn’t we pop some node n, and find its child n’ to have lower f value?
- YES:
  - What can we require to prevent these inversions?

\[
g = 10
\]

\[
h = 10
\]

\[
3
\]
Consistency

- Wait, how do we know we expand in increasing f value?
- Couldn’t we pop some node $n$, and find its child $n'$ to have lower f value?
- YES:
  - What can we require to prevent these inversions?
  - Consistency: $c(n, a, n') \geq h(n) - h(n')$
  - Real cost must always exceed reduction in heuristic
Optimality

• Tree search:
  – A* optimal if heuristic is admissible (and non-negative)
  – UCS is a special case (h = 0)

• Graph search:
  – A* optimal if heuristic is consistent
  – UCS optimal (h = 0 is consistent)

• Consistency implies admissibility

• In general, natural admissible heuristics tend to be consistent
Complexity of A*

• A* is optimally efficient (Dechter and Pearl 1985):
  – It can be shown that all algorithms that do not expand a node which A* did expand (inside the contours) may miss an optimal solution

• A* worst-case time complexity:
  – is exponential unless the heuristic function is very accurate

• If h is exact (h = h*)
  – search focus only on optimal paths

• Main problem: space complexity is exponential

• Effective branching factor:
  – logarithm of base (d+1) of average number of nodes expanded.
Summary: A*

• A* uses both backward costs ($g(n)$) and estimates of forward costs ($h(n)$)

• A* is optimal with admissible heuristics

• Heuristic design is key: often use relaxed problems