Informed Search

Chapter 3.5 – 3.6, 4.1

Informed Search

• Informed searches use domain knowledge to guide selection of the best path to continue searching

• Heuristics are used, which are informed guesses

• Heuristic means "serving to aid discovery"

Informed Search

• Define a heuristic function, \( h(n) \)
  – uses domain-specific information in some way
  – is computable from the current state description
  – it estimates
    • the "goodness" of node \( n \)
    • how close node \( n \) is to a goal
    • the cost of minimal cost path from node \( n \) to a goal state

Informed Search

• \( h(n) \geq 0 \) for all nodes \( n \)
• \( h(n) \) close to 0 means we think \( n \) is close to a goal state
• \( h(n) \) very big means we think \( n \) is far from a goal state

• All domain knowledge used in the search is encoded in the heuristic function, \( h \)
• An example of a “weak method” for AI because of the limited way that domain-specific information is used to solve a problem
**Best-First Search**

- Sort nodes in the Frontier list by increasing values of an evaluation function, $f(n)$, that incorporates domain-specific information.
- This is a generic way of referring to the class of informed search methods.

**Greedy Best-First Search**

- Use as an evaluation function, $f(n) = h(n)$, sorting nodes in the Frontier by increasing values of $f$.
- Selects the node to expand that is believed to be closest (i.e., smallest $f$ value) to a goal node.

---

**Greedy Best-First Search**

$f(n) = h(n)$

<table>
<thead>
<tr>
<th>expnd. node</th>
<th>Frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S:8)</td>
</tr>
</tbody>
</table>

# of nodes tested: 0, expanded: 0

---

**Greedy Best-First Search**

$f(n) = h(n)$

<table>
<thead>
<tr>
<th>expnd. node</th>
<th>Frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S:8)</td>
</tr>
</tbody>
</table>

# of nodes tested: 1, expanded: 1

---

**Greedy Best-First Search**

$f(n) = h(n)$

<table>
<thead>
<tr>
<th>expnd. node</th>
<th>Frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S:8)</td>
</tr>
</tbody>
</table>

S not goal

---

**Greedy Best-First Search**

$f(n) = h(n)$

<table>
<thead>
<tr>
<th>expnd. node</th>
<th>Frontier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S:8)</td>
</tr>
</tbody>
</table>

---
Greedy Best-First Search

\[ f(n) = h(n) \]

- # of nodes tested: 3, expanded: 2
- expnd. node Frontier
  - S
  - C:3,B:4,A:8
  - G
  - B:4,A:8
- C not goal
  - C:0,B:4,A:8

- path: S,C,G
- cost: 13

• Fast but not optimal

Example: Find Path from Arad to Bucharest
**Heuristic: Straight-Line Distance**

**Greedy Best-First Search**
- Not complete
- Not optimal/admissible

Greedy search finds the left goal (solution cost of 7)
Optimal solution is the path to the right goal (solution cost of 5)

**Beam Search**
- Use an evaluation function $f(n) = h(n)$ as in greedy best-first search, *and* restrict the maximum size of the Frontier to a constant, $k$
- Only keep $k$ best nodes as candidates for expansion, and throw away the rest
- More space efficient than Greedy Search, but may throw away a node on a solution path
- Not complete
- Not optimal/admissible

**Algorithm A Search**
- Use as an evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is minimum cost path from start to current node $n$ (as defined in UCS)
- The $g$ term adds a “breadth-first component” to the evaluation function
- Nodes in Frontier are ranked by the estimated cost of a solution, where $g(n)$ is the cost from the start node to node $n$, and $h(n)$ is the estimated cost from node $n$ to a goal
Is Algorithm A Optimal?

Algorithm A* Search

- Use the same evaluation function used by Algorithm A, except add the constraint that for all nodes \( n \) in the search space, \( h(n) \leq h^*(n) \), where \( h^*(n) \) is the actual cost of the minimum-cost path from \( n \) to a goal.
- The cost to the nearest goal is **never over-estimated**.
- When \( h(n) \leq h^*(n) \) holds true for all \( n \), \( h \) is called an **admissible heuristic function**.
- An admissible heuristic guarantees that a node on the optimal path cannot look so bad so that it is never considered.

Admissible Heuristics are Good for Playing *The Price is Right*
Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ g(n) = \text{actual cost to get to node } n \text{ from start} \]

Example

\[ g(n) = \text{actual cost to get to node } n \text{ from start} \]
Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$g(n) = \text{actual cost to get to node } n \text{ from start}$

Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$g(n) = \text{actual cost to get to node } n \text{ from start}$

Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(n) = \text{estimated cost to get to a goal from node } n$
Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td>0</td>
<td>10/9/13</td>
<td></td>
</tr>
</tbody>
</table>

\(h(n)\) = estimated cost to get to a goal from node \(n\)

Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>3</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td>0</td>
<td>10/9/13</td>
<td>10/9/13</td>
</tr>
</tbody>
</table>

\(f(n) = g(n) + h(n)\)

actual cost to get from start to \(n\) plus estimated cost from \(n\) to goal

Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td>0</td>
<td>10/9/13</td>
<td></td>
</tr>
</tbody>
</table>

\(h^*(n) = true cost of minimum-cost path from \(n\) to a goal\)

Example

<table>
<thead>
<tr>
<th>n</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
<th>h*(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>3</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>G</td>
<td>10/9/13</td>
<td>0</td>
<td>10/9/13</td>
<td>10/9/13</td>
</tr>
</tbody>
</table>

\(h^*(n) = true cost of minimum-cost path from \(n\) to a goal\)
The image appears to be a page from a textbook or a document discussing a problem in graph theory, likely related to the A* algorithm. The table and diagrams illustrate the process of finding the true cost of minimum-cost paths from a node to a goal. The notation $h^*(n) = true\ cost\ of\ minimum-cost\ path\ from\ n\ to\ a\ goal$ is used to denote the cost of the optimal path to a goal from node $n$.
Algorithm A* Search

- Use the same evaluation function used by Algorithm A, except add the constraint that for all nodes $n$ in the search space, $h(n) \leq h^*(n)$, where $h^*(n)$ is the actual cost of the minimum-cost path from $n$ to a goal.
- The cost to the nearest goal is never over-estimated.
- When $h(n) \leq h^*(n)$ holds true for all $n$, $h$ is called an admissible heuristic function.
- An admissible heuristic guarantees that a node on the optimal path cannot look so bad so that it is never considered.

Admissible Heuristic Functions, $h$

- 8-Puzzle example

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

- Which of the following are admissible heuristics?

$$h(n) = \text{number of tiles in wrong position}$$
$$h(n) = 0$$
$$h(n) = 1$$
$$h(n) = \text{sum of “City-block distance” between each tile and its goal location}$$

Note: City-block distance = $L_1$ norm.
Admissible Heuristic Functions, $h$

Which of the following are admissible heuristics?

- $h(n) = h^*(n)$
- $h(n) = \max(2, h^*(n))$
- $h(n) = \min(2, h^*(n))$
- $h(n) = h^*(n) - 2$
- $h(n) = \sqrt{h^*(n)}$

Admissible Heuristic Functions, $h$

Which of the following are admissible heuristics?

- $h(n) = h^*(n)$  **YES**
- $h(n) = \max(2, h^*(n))$  **NO**
- $h(n) = \min(2, h^*(n))$  **YES**
- $h(n) = h^*(n) - 2$  **NO, possibly negative**
- $h(n) = \sqrt{h^*(n)}$  **NO if $h^*(n) < 1$**

When should A* Stop?

- A* should terminate only when a goal is **removed** from the priority queue

When should A* Stop?

- One more complication: A* might revisit a state (in Frontier or Explored), and discover a better path

• Solution: Put $D$ back into the priority queue, with the smaller $g$ value (and path)
A and A* Algorithm for General State-Space Graphs

Frontier = \{S\} where S is the start node
Explored = \{

Loop do

if Frontier is empty then return failure
pick node, n, with min f value from Frontier
if n is a goal node then return solution
foreach each child, n', of n do
if n' is not in Explored or Frontier
then add n' to Frontier
else if g(n') ≥ g(m) then throw n' away
else add n' to Frontier and remove m
Remove n from Frontier and add n to Explored

Note: m is the node in Frontier or Explored that is the same state as n'

Consistency

• A heuristic, h, is called consistent (aka monotonic) if, for every node n and every successor n' of n, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

  c(n, n') ≥ h(n) − h(n')

  or, equivalently: h(n) ≤ c(n, n') + h(n')

• Triangle inequality for heuristics
• Values of f(n) along any path are nondecreasing
• When a node is selected for expansion by A*, the optimal path to that node has been found
• Consistency is a stronger condition than admissibility

Consistency

Is this h consistent?

\[
\begin{array}{c}
S & 1 & B & 2 & D & 999 & G \\
\hline
h=1 & h=1 & h=1 & h=0
\end{array}
\]

\[
\begin{array}{c}
C & 1 & 1 \\
\hline
h=900
\end{array}
\]

f(n) = g(n) + h(n)
# of nodes tested: 0, expanded: 0

expnd. node  Frontier

\[
\begin{array}{c|c}
\text{exnd. node} & \text{Frontier} \\
\hline
[S:0+8] & \{S:0+8\}
\end{array}
\]

A* Search

Is h is admissible and/or consistent?

h(C)=900, h(D)=1, c(C, D) = 1 < 900 − 1, so h is NOT consistent.

h is consistent since h(S) − h(A) = 8 − 8 ≤ 1, etc.
and therefore is also admissible
**A* Search**

\[ f(n) = g(n) + h(n) \]

- # of nodes tested: 1, expanded: 1
  - expnd.
    node
  - Frontier
  - S: 8
  - S not goal
    - {A: 1+8, B: 5+4, C: 8+3}

**A* Search**

\[ f(n) = g(n) + h(n) \]

- # of nodes tested: 2, expanded: 2
  - expnd.
    node
  - Frontier
  - S: 8
  - S (A:9, B:9, C:11)
  - A not goal
    - {B: 5+4+0, G: 1+9+0, C: 11, D: 1+3+∞, E: 1+7+∞}

**A* Search**

\[ f(n) = g(n) + h(n) \]

- # of nodes tested: 3, expanded: 3
  - expnd.
    node
  - Frontier
  - S: 8
  - S (A:9, B:9, C:11)

**A* Search**

\[ f(n) = g(n) + h(n) \]

- # of nodes tested: 4, expanded: 3
  - expnd.
    node
  - Frontier
  - S: 8
  - S (A:9, B:9, C:11)
  - G goal
    - {C: 11, D: 3, E: ∞}
A* Search

\[ f(n) = g(n) + h(n) \]

# of nodes tested: 4, expanded: 3

- Pretty fast and optimal

Example: Find Path from Arad to Bucharest

Heuristic: Straight-Line Distance to Bucharest

The initial state

After expanding Arad
Visualizing Search Methods

- [http://qiao.github.io/PathFinding.js/visual/](http://qiao.github.io/PathFinding.js/visual/)
- BFS, UCS, Greedy Best-First, A*

A*: The Dark Side

- A* can use lots of memory: $O(\text{number of states})$
- For really big search spaces, A* will run out of memory

Devising Heuristics

Heuristics are often defined by relaxing the problem, i.e., computing the exact cost of a solution to a simplified version of problem
- remove constraints: 8-puzzle movement
- simplify problem: straight line distance for 8-puzzle and mazes

Comparing Iterative Deepening with A*

[from Russell and Norvig, Fig 3.29]

<table>
<thead>
<tr>
<th>Depth-First Iterative Deepening (IDS)</th>
<th>4 steps</th>
<th>8 steps</th>
<th>12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>A* search using “number of misplaced tiles” as the heuristic</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
<tr>
<td>A* using “Sum of Manhattan distances” as the heuristic</td>
<td>12</td>
<td>25</td>
<td>73</td>
</tr>
</tbody>
</table>

For 8-puzzle, average number of states expanded over 100 randomly chosen problems in which optimal path is length …
Devising Heuristics

• Ideally, we want an admissible heuristic that is as close to the actual cost without going over

• Also, must be relatively fast to compute

• Trade off: use more time to compute a complex heuristic versus use more time to expand more nodes with a simpler heuristic

Devising Heuristics

If $h_1(n) \leq h_2(n) \leq h^*(n)$ for all $n$, then $h_2$ dominates $h_1$

– $h_2$ is a better heuristic than $h_1$

– $A^*$ using $h_1$ (i.e., $A_1^*$) expands at least as many if not more nodes than using $A^*$ with $h_2$ (i.e., $A_2^*$)

– $A_2^*$ is said to be better informed than $A_1^*$

Devising Heuristics

• If $h(n) = h^*(n)$ for all $n$,
  – only nodes on optimal solution path are expanded
  – no unnecessary work is performed

• If $h(n) = 0$ for all $n$,
  – the heuristic is admissible
  – $A^*$ performs exactly as Uniform-Cost Search (UCS)

• The closer $h$ is to $h^*$, the fewer extra nodes that will be expanded

Devising Heuristics

For an admissible heuristic

– $h$ is frequently very simple

– therefore search resorts to (almost) UCS through parts of the search space
Devising Heuristics

- If optimality is not required, i.e., *satisficing solution* okay, then
- Goal of heuristic is then to get as close as possible, either under or over, to the actual cost
- It results in many fewer nodes being expanded than using a poor, but provably admissible, heuristic

Devising Heuristics

A* often suffers because it cannot venture down a single path unless it is almost continuously having success (i.e., \( h \) is decreasing); any failure to decrease \( h \) will almost immediately cause the search to switch to another path.

Local Searching

*Systematic searching:* Search for a **path** from start state to a goal state, then “execute” solution path’s sequence of operators

- BFS, DFS, IDS, UCS, Greedy Best-First, A, A*, etc.
- **ok** for small search spaces
- **not okay** for NP-Hard problems requiring exponential time to find the (optimal) solution

Traveling Salesperson Problem (TSP)

Question: How would you solve using A or A* Algorithm?

Nodes are cities
Arcs are labeled with distances between cities
Adjacency matrix (notice the graph is fully connected):

```
A B C D E
A 0 5 8 9 7
B 5 0 6 5 5
C 8 6 0 2 3
D 9 5 2 0 4
E 7 5 3 4 0
```
• How to represent a state?
• Successor function
• Heuristics

**Heuristics**
- Is Straight-Line-Distance back to Start City admissible?
- Is Minimum-Spanning-Tree length for all cities admissible?

**Optimization Problems**

*Now a different setting:*
- Each state $s$ has a **score** or **cost**, $f(s)$, that we can compute
- The goal is to find the state with the **highest** (or **lowest**) **score**, or a **reasonably high** (low) **score**
- We do **not** care about the path
- Use **variable-based models**
  - Solution is not a path but an assignment of values for a set of variables
- **Enumerating the states is intractable**
  - Previous search algorithms are too expensive

**Traveling Salesperson Problem (TSP)**

*Classic NP-Hard problem:*
- A salesperson wants to visit a list of cities
  - stopping in each city only **once**
  - (sometimes also must return to the first city)
  - traveling the shortest distance
- $f =$ total distance traveled
Traveling Salesperson Problem (TSP)

Nodes are cities
Arcs are labeled with distances between cities
Adjacency matrix (notice the graph is fully connected):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Traveling Salesperson Problem (TSP)

a solution is a permutation of cities, called a tour

e.g. A – B – C – D – E

assume tours can start at any city and do not return home

How many solutions exist?

\( n! \) where \( n \) = # of cities

- \( n = 5 \) results in 120 tours
- \( n = 10 \) results in 3,628,800 tours
- \( n = 20 \) results in \( \approx 2.4 \times 10^{18} \) tours
Example Problems

• **N-Queens**
  – Place $n$ queens on $n \times n$ checkerboard so that no queen can “capture” another
  – $f =$ number of conflicting queens

• **Boolean Satisfiability**
  – Given a Boolean expression containing $n$ Boolean variables, find an assignment of {T, F} to each variable so that the expression evaluates to True
  – $(A \lor \neg B \lor C) \land (\neg A \lor C \lor D)$
  – $f =$ number of satisfied clauses

---

Example Problem: Chip Layout

Example Problem: Scheduling

Local Searching

• Hard problems can be solved in polynomial time by using either an:
  – **approximate model**: find an exact solution to a simpler version of the problem
  – **approximate solution**: find a non-optimal solution to the original hard problem

• We’ll explore means to search through a **solution space** by **iteratively improving** solutions until one is found that is optimal or near optimal
Local Searching

- **Local searching**: every node is a solution
  - Operators/actions go from one solution to another
  - can stop at any time and have a valid solution
  - goal of search is to find a better/best solution
- No longer searching state space for a solution path and then executing the steps of the solution path
- A* isn't a local search since it searches different partial solutions by looking at the estimated cost of a solution path

An operator/action is needed to transform one solution to another

- TSP: 2-swap operator
  - take two cities and swap their positions in the tour
  - A-B-C-D-E with swap(A,D) yields D-B-C-A-E
  - possible since graph is fully connected
- TSP: 2-interchange operator (aka 2-opt swap)
  - reverse the path between two cities
  - A-B-C-D-E with interchange(A,D) yields D-C-B-A-E

Neighbors: TSP

- state: A-B-C-D-E-F-G-H-A
- \( f = \) length of tour
- 2-interchange

\[
\text{Neighborhood diagram}
\]

Those solutions that can be reached with one application of an operator are in the current solution's neighborhood (aka “move set”)

- **Local search** considers next only those solutions in the neighborhood
- The neighborhood should be much smaller than the size of the search space (otherwise the search degenerates)
Examples of Neighborhoods

- **N-queens**: Move queen in rightmost, most-conflicting column to a different position in that column
- **SAT**: Flip the assignment of one Boolean variable

Neighbors: SAT

- **State**: \( (A=T, B=F, C=T, D=T, E=T) \)
- \( f \) = number of satisfied clauses
- **Neighbor**: flip the assignment of *one* variable

\[
\begin{align*}
(A & = F, B = F, C = T, D = T, E = T) \\
(A & = T, B = T, C = T, D = T, E = T) \\
(A & = T, B = F, C = F, D = T, E = T) \\
(A & = T, B = F, C = T, D = F, E = T) \\
(A & = T, B = F, C = T, D = T, E = F)
\end{align*}
\]

Local Searching

- An evaluation function, \( f \), is used to map each solution/state to a number corresponding to the quality/cost of that solution
- **TSP**: Use the length of the tour; A better solution has a shorter tour length
- Maximize \( f \): called **hill-climbing** (gradient ascent if continuous)
- Minimize \( f \): called or **valley-finding** (gradient descent if continuous)
- Can be used to maximize/minimize some cost

Hill-Climbing (HC)

- **Question**: What’s a neighbor?
  - Problem spaces tend to have structure. A *small change* produces a neighboring state
  - The size of the neighborhood must be small enough for efficiency
  - Designing the neighborhood is critical; This is the real ingenuity – not the decision to use hill-climbing
- **Question**: Pick which neighbor? *The best one* (greedy)
- **Question**: What if no neighbor is better than the current state? *Stop*
Example: Highest Point in Madison

- Elver Park: Sledding Hill (~1 mi south of Beltline)
- Goal: Find the top of the hill by exploring Madison
- Question: How do we define our neighborhood?
- Question: How do we choose which neighbor to pick?
- Question: When do we stop searching?

Hill-Climbing Algorithm

1. Pick initial state $s$
2. Pick $t$ in neighbors($s$) with the largest $f(t)$
3. if $f(t) \leq f(s)$ then stop and return $s$

- Simple
- Greedy
- Stops at a local maximum

Hill-Climbing (HC)

- HC exploits the neighborhood
  - like Greedy Best-First search, it chooses what looks best locally
  - but doesn’t allow backtracking or jumping to an alternative path since there is no Frontier list
- HC is very space efficient
- HC is very fast and often effective in practice

Local Optima in Hill-Climbing

- Useful mental picture: $f$ is a surface (‘hills’) in state space
  - Global optimum, where we want to be
  - But we can’t see the entire landscape all at once. Can only see a neighborhood; like climbing in fog.
Hill-Climbing

Visualized as a 2D surface
- Height is quality/cost of solution \( f = f(x, y) \)
- Solution space is a 2D surface
- Initial solution is a point
- Goal is to find highest point on the surface of solution space
- Hill-Climbing follows the direction of the steepest ascent, i.e., where \( f \) increases the most

Hill-Climbing (HC)

Solution found by HC is totally determined by the starting point; fundamental weakness is getting stuck:
- At a local maximum
- At plateaus

**Global maximum may not be found**

Trade off:
greedily exploiting locality as in HC vs. exploring state space as in BFS

Hill-Climbing with Random Restarts

- Very simple modification:
  1. When stuck, pick a random new starting state and re-run hill-climbing from there
  2. Repeat this \( k \) times
  3. Return the best of the \( k \) local optima

- Can be very effective
- Should be tried whenever hill-climbing is used
- Fast, easy to implement; works well for many applications where the solution space surface is not too “bumpy” (i.e., not too many local maxima)

Escaping Local Maxima

- HC gets stuck at a local maximum, limiting the quality of the solution found
- Two ways to modify HC:
  1. choice of neighborhood
  2. criterion for deciding to move to neighbor
- For example:
  1. choose neighbor randomly
  2. move to neighbor if it is better or, if it isn’t, move with some probability, \( p \)
Variations on Hill-Climbing

• **Question**: How do we make hill-climbing less greedy?
  • **Stochastic Hill-Climbing**
    • Randomly select among the neighbors
    • The better, the more likely

• **Question**: What if the neighborhood is too large to easily compute? (e.g., N-queens if we need to pick both the column and the row within it)
  • **First-choice hill-climbing**
    • Randomly generate neighbors, one at a time
    • If neighbor is better, take the move

Life Lesson #237

• **Sometimes one needs to temporarily step backward in order to move forward**

  • Lesson applied to iterative, local search:
    – Sometimes one needs to move to an *inferior neighbor* in order to escape a local optimum

Simulated Annealing

(Stochastic Hill-Climbing)

1. Pick initial state, $s$
2. Randomly pick state $t$ from neighbors of $s$
3. if $f(t)$ better than $f(s)$
   then $s = t$
   else with small probability $s = t$
4. Goto Step 2 until bored

Simulated Annealing

**Origin:**
The annealing process of heated solids – Alloys manage to find a near global minimum energy state when heated and then slowly cooled

**Intuition:**
By allowing occasional ascent in the search process, we might be able to escape the traps of local minima

**Introduced by Nicholas Metropolis in 1953**
Consequences of Occasional Bad Moves

desired effect (when searching for a global min)

Helps escape local optima

Idea 1: Use a small, fixed probability threshold, say, \( p = 0.1 \)

adverse effect

But might pass global optimum after reaching it

Escaping Local Optima

• Modified HC can escape from a local optimum \( \text{but} \)
  – chance of making a bad move is the same at the beginning of the search as at the end
  – magnitude of improvement, or lack of, is ignored
• Fix by replacing fixed probability, \( p \), that a bad move is accepted, with a probability that \text{decreases} as the search proceeds
• Now as the search progresses, the chance of taking a bad move goes down

Control of Annealing Process

Acceptance decision for a search step (Metropolis Criterion) in Hill-Climbing:

• Let the performance change in the search be:
  \( \Delta E = f(\text{newNode}) - f(\text{currentNode}) \)

• \text{Always accept} an ascending step (i.e., better state)
  \( \Delta E \geq 0 \)

• Accept a descending step only if it passes a test

Escaping Local Maxima

Let \( \Delta E = f(\text{newNode}) - f(\text{currentNode}) < 0 \)

\[ p = e^{\frac{\Delta E}{T}} \] (Boltzman’s equation)

• \( \Delta E \to -\infty, \; p \to 0 \)
  as badness of the move \text{increases}
  probability of taking it \text{decreases} exponentially
• \( T \to 0, \; p \to 0 \)
  as temperature \text{decreases}
  probability of taking bad move \text{decreases}
Control of Annealing Process

**Cooling Schedule:**
- $T$, the *annealing temperature*, is the parameter that controls the frequency of acceptance of bad steps.
- We gradually reduce temperature $T(k)$.
- At each temperature, search is allowed to proceed for a certain number of steps, $L(k)$.
- The choice of parameters $\{T(k), L(k)\}$ is called the *cooling schedule*.

Simulated Annealing (Stochastic Hill-Climbing)

Pick initial state, $s$
$k = 0$

**while** $k < k_{\text{max}}$ **do**

- $T = \text{temperature}(k)$
- Randomly pick state $t$ from neighbors of $s$
  - **if** $f(t) > f(s)$ **then** $s = t$
  - **else** if $(e^{(f(t) - f(s))/T} > \text{random()}$ **then** $s = t$
  - $k = k + 1$

**return** $s$
Simulated Annealing Demo

Searching for a global minimum

Simulated Annealing
- Can perform multiple backward steps in a row to escape a local optimum
- Chance of finding a global optimum increased
- Fast
  - only one neighbor generated at each iteration
  - whole neighborhood isn't checked to find best neighbor as in HC
  - Usually finds a good quality solution in a very short amount of time

Simulated Annealing
- Requires several parameters to be set
  - starting temperature
    - must be high enough to escape local optima but not too high to be random exploration of space
  - cooling schedule
    - typically exponential
  - halting temperature
- Domain knowledge helps set values: size of search space, bounds of maximum and minimum solutions

Implementation of Simulated Annealing
- This is a stochastic algorithm; the outcome may be different at different trials
- Convergence to global optimum can only be realized in an asymptotic sense
- With infinitely slow cooling rate, finds global optimum with probability 1
Simulated annealing is sometimes empirically much better at avoiding local maxima than hill-climbing. It is a successful, frequently-used, algorithm. Worth putting in your algorithmic toolbox.

Sadly, not much opportunity to say anything formal about it (though there is a proof that with an infinitely slow cooling rate, you’ll find the global optimum)

There are mountains of practical, and problem-specific, papers on improvements.

Local Searching

- Iteratively improve solution
- **nodes**: complete solution
- **arcs**: operator changes to another solution
- can stop at any time
- technique suited for:
  - hard problems, e.g., TSP
  - optimization problems

**Summary**

Local Searching

- \( f(n) \) evaluates quality of solution by weakly using domain knowledge
- HC: maximizes \( f(n) \), VF: minimizes \( f(n) \)
  - solution found determined by starting point
  - can get stuck, which prevents finding global optimum
- SA: explores, then settles down
  - bad moves accepted with probability that decreases as the search progress (\( T \) decreases)
  - with the badness of move (\( \Delta E \) worsens)
  - requires parameters to be set