Search

Search Trees

- node = state
- arcs = operators
- search = control

branching factor = # choices of each node
**Search Criteria**

- **Completeness**: guarantees to find a solution when there is one
- **Optimality**: highest quality solution where there are different solutions
- **Efficiency**: time complexity and space complexity

**Goal**: find the right search criterion for a problem

---

**Search Algorithms**

Types of search algorithms in two perspectives:

- Based on information used
  - **Blind Search**: e.g., depth-first, breath-first, bidirectional search
  - **Heuristic Search**: e.g., hill climbing, best first search, A* search

- Based on search direction (more later)
  - **Goal-driven**: (backward chaining): search from goal states to verify given data
  - **Data-driven**: (forward chaining): search from data to find goal states
Blind Search (Uninformed or Brute force search)

- Systematically generates states and test against the goal
- Depends on topology of search tree
- Is distinguished by the order in which nodes are expanded
- Has no information about the number of steps or the path cost from current state to the goal state

Examples:
- Backtracking
- Depth First
- Breadth First

Backtracking

Solution path: ACG
Backtracking Algorithm

CS = current state
SL = state list of current path being evaluated for solution
NSL = new state list
DE = dead end list of states whose descendants failed to contain goal node

Begin
SL := [start]; NSL := [start]; DE := [] ; CS:= Start;
while NSL != [] do
begin
if CS = goal then return SL;
if CS has no children then
begin
while SL is not empty and CS = first element of SL do
begin
if CS = goal then return SL;
add CS to DE;
remove first element from SL;
remove first element from NSL;
CS := first element of NSL
end;
end;
add CS to SL;
end
else begin
place new children of CS on NSL;
CS := first element of NSL;
add CS to SL
end
end;
return FAIL;
end

Tracing SL

<table>
<thead>
<tr>
<th>CS</th>
<th>SL</th>
<th>NSL</th>
<th>DE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>B</td>
<td>BCDA</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>EBA</td>
<td>EFBCDA</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>EBA</td>
<td>EFBIDA</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>FBA</td>
<td>FBCDA</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>CA</td>
<td>CBA</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>GCA</td>
<td>GCDA</td>
<td></td>
</tr>
</tbody>
</table>
Breadth-First Search (BFS)

- Expands the shallowest node first

Algorithm:
1. Let Q = queue contain start state
2. If head of Q is a goal state STOP
3. Generate successors of head and add them at the tail of Q. Remove the head.
4. Go to 2

Solution: trace ancestors - ACG
Node expansion order: ABCDEFG

Depth-First Search (DFS)

- Expands the deepest node first

Algorithm:
1. Let Q = queue contain start state
2. If head of Q is a goal state STOP
3. Generate successors of head and add them at the head of Q. Remove the head.
4. Go to 2

Solution: ABFG
Node expansion order: ABEHIFJG
**Complexity**

Given $b =$ branching factor, $d =$ solution depth

Running time = the time generating nodes at depth 1, 2, ..., $d$

$$= b + b^2 + ... + b^d$$

시간 복잡성은 $O(b^d)$ 로 BFS와 DFS 모두 동일합니다.

Space required = max queue length

For BFS: $= \#$ nodes at depth $d$

$$= b^d$$

공간 복잡성은 $O(b^d)$ 입니다.

For DFS: $= bd$

공간 복잡성은 $O(bd)$ 입니다.

**Characteristics**

BFS
- Shortest path solution
- Optimal if unit cost (since cost = path length)
- Complete
- Time complexity $O(b^d)$
  - Space complexity $O(b^d)$

DFS
- Space efficiency (Space complexity $O(bd)$)
- Not complete (should avoid when space has large/infinite depth)
- Not optimal (see example 2)
**BFS Example 2**

Start
A 9 B 3 D 4 E
C 7 4 5 11 Goal

Solution:
ACE

Shortest path solution

Solution cost = 18

Given cost

Solution path = trace ancestors

**DFS Example 2**

Start
A 9 B 3 D 5 E
C 7 4 11 Goal

Solution:
ABDE

Solution path = trace ancestors

What is optimal solution? What cost?

Given cost

Solution cost = 17
**Other Blind Searches**

- **Uniform-cost Search:** expands the least-cost leaf node first
- **Depth-limited Search:** places a limit on the DFS depth
- **Iterative deepening Search:** calls Depth-limited search iteratively
- **Bidirectional Search:** searches forward (from start state) and backward (from goal state) and stop where two searches meet

---

**Uniform Cost Search**

- Expands the least expensive node first

Given cost

Solution cost = 16

UCF gives solution with min. cost of the path traveled so far

Solution: ACDE
Other Blind Searches

- **Depth-limited Search** (to avoid drawback of DFS)
  - DFS that imposes cutoff on the max depth of search path
  - complete if cutoff depth large enough
  - not optimal (like DFS)

- **(Depth First) Iterative deepening Search**
  (to avoid issue of choosing cutoffs without sacrificing efficiency)
  - calls Depth-limited search iteratively for increasing cutoff depth
  - order of node expansion ~ BFS
  - combines benefits of DFS and BFS
  - good for large search space with unknown solution depth

Other Blind Searches (contd)

- **Bidirectional Search** (to improve efficiency of BFS)
  - searches forward (from start state) and backward (from goal state) and stop where two searches meet
  - Time complexity $O(b^{d/2})$

What are the requirements of problems that are applicable to this search?
### Blind Search (contd)

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Complete?</th>
<th>Optimal?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first Search</td>
<td>yes</td>
<td>yes for unit-cost</td>
</tr>
<tr>
<td>Depth-first Search</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Uniform-cost Search</td>
<td>yes</td>
<td>yes for non-decreasing path cost</td>
</tr>
<tr>
<td>Depth-limited Search</td>
<td>only if deep enough</td>
<td>no (like DFS)</td>
</tr>
<tr>
<td>Iterative deepening Search</td>
<td>yes</td>
<td>yes for unit-cost</td>
</tr>
<tr>
<td>Bidirectional Search</td>
<td>yes</td>
<td>yes for unit-cost</td>
</tr>
</tbody>
</table>

### Blind Search Issues

**Problems** with Brute force methods

- Time complexity grows exponentially with size of the problem
  - Eg. Rubik’s cube: $4 \times 10^{19}$ nodes
  - Chess tree: $10^{120}$ nodes
  - *Combinatorial Explosion!* → Extremely inefficient

**Solution:** add knowledge to reduce complexity
Heuristic Search

Luger, Ch 4 & Reference Texts

Outline

- What are heuristics and heuristic functions?
- Using heuristics in games
- Heuristic search algorithms
- Properties and complexity
Heuristics

The term “Heuristic” means “to discover”

- “rules of thumb” that domain experts use to generate good solutions without exhaustive search (Fiegenbaum and Feldman)
- “A process that may solve a given problem, but offers no guarantees of doing so” (Newell, Shaw and Simon)
- Technique that improves the average-case performance on a problem-solving task, but not necessarily improve the worst-case performance (Russell and Norvig)

Using heuristics ———> Efficiency + Accuracy
(might miss optimal solution)

Heuristics in Search

Heuristic Search (or Informed search): uses

Problem-specific knowledge

Evaluation function
determines the order of nodes to be expanded

Heuristic evaluation function
estimates merit of state with respect to the goal
Heuristic functions

Examples: Let $h(n) =$ heuristic value of state $n$

- Road navigation or route finding
  
  $h_1(n) =$ Euclidean distance from $n$ to goal state
  
  $h_2(n) =$ Expected traveling time

Heuristic functions (contd)

- The 8 puzzle: E.g.,
  
  $h_1(n) =$ the number of tiles that are out of place from the goal
  
  $h_2(n) =$ sum of distances out of place
  
  $h_3(n) =$ $2 \times$ the number of direct tile reversals

<table>
<thead>
<tr>
<th>state n</th>
<th>Goal state</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 8 3</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>0 6 5 7</td>
<td>4 2 - 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$h_1(n)$</th>
<th>$h_2(n)$</th>
<th>$h_3(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

- Which heuristics to choose?
  
  - prefer global to local accounts (e.g., $h_1$ is more global than $h_2$ but more local than $h_3$)
  
  - Selection of the best heuristic function is empirical and based on its performance on problem instances
Heuristic functions (contd)

- A heuristic evaluation function must
  - provide a reasonable estimate of the merit of a node
  - be inexpensive to compute
  - never over estimate the merit of a node (more later)

- There are tradeoffs between
  - evaluation time
  - search time

- Heuristic sacrifices accuracy but why still heuristic?
  - rarely require optimal solution
  - worse case rarely occur
  - learning heuristic helps us understand the problem better

Outline

- What are heuristics and heuristic functions?
- Using heuristics in games
- Heuristic search algorithms
- Properties and issues
Game playing

- Representation: Game tree
  - Nodes: state of the game (e.g. board configuration)
  - Branches: possible moves
  - Leaves: winning states
- Game trees usually have multiple goal states
- Heuristic function evaluates game value or pay-off or utility of each node
Game playing program

- Full game trees are intractable, e.g., TIC-TAC-TOE
  - total # of leaves = 9!
  - reduced states by symmetry (shown in previous slide) = 12 x 7!

- Two main parts of the program:
  - plausible move generator
  - (static) evaluation function

- Search game trees
  - One-person games (e.g., 8 puzzle): A*
  - Two-person games (e.g., checkers, tic-tac-toe): MINIMAX

---

Heuristics for TIC-TAC-TOE

Heuristic function, \( e(n) = x(n) - o(n) \)

- \( x(n) \) = # winning lines for x, x will try to max \( e \)
- \( o(n) \) = # winning lines for o, o will try to min \( e \)
Minimax Search

- A depth-limited search procedure
  - Generate possible moves
  - Apply evaluation function for each node
  - Alternate selection of either max or min child value
  - Repeat process as deep as time allows

Minimax Search (contd.)

Let \( p \) be a search level (ply)

\[
\text{while } p > 0 \text{ do begin}
\text{ if } p \text{'s parent level is MIN then for each parent node pick min value of its children,}
\text{ else pick max value; propagate the selected value to its parent;}
\text{ } p = p - 1
\text{ end.}
\]
Minimax Search (Contd.)

- Search depth could effect results
  - one- ply search $\rightarrow$ second move
  - two- ply search $\rightarrow$ first move

- Issues:
  - limited look- ahead may lead to bad states later
    - remedy: search deeper BUT ....
  - search deeper doesn’t necessary mean better

Alpha Beta Procedure

- Improve efficiency by pruning
  - value of this node = 2
  - cutoff
**Alpha Beta Procedure**

\[ \alpha = \text{lower bound of MAX node} \]
\[ \beta = \text{upper bound of MIN node} \]
\[ \alpha > \beta \rightarrow \text{cutoff} \]

- **Example 1:**
  - Max \( \geq 2 \)
  - Min \( 2 \)
  - \( \alpha = 2 \)
  - \( \beta = 1 \)
  - B is \( \alpha \)-cutoff

- **Example 2:**
  - Min \( \leq 7 \)
  - Max \( 7 \)
  - \( \beta = 7 \)
  - B is \( \beta \)-cutoff

**More Examples: \( \alpha - \beta \) pruning**

Result on Minimax without pruning.
Select first move for a three-ply search to obtain a game value 3
More Examples: $\alpha$-$\beta$ pruning

Init: $\alpha$ (lower bound of MAX node) = $-\infty$ and $\beta$ (upper bound of MIN node) = $\infty$

Save time on evaluation of five nodes.
Result is to choose first move.

More Examples: $\alpha$-$\beta$ pruning (contd)

Save time on evaluation of 11 nodes.
Result is to choose middle move.
**Best case pruning**

Assume left to right evaluation.
Any game tree with better pruning? What’s its characteristic?

Best case: the best child is the first one being evaluated $\sim O(b^{d/2})$

# of evaluations $= \begin{cases} 2b^{d/2} - 1 & \text{if } d \text{ even} \\ b^{(d+1)/2} + b^{(d-1)/2} - 1 & \text{if } d \text{ is odd} \end{cases}$ [Knuth]

---

**Outline**

- What are heuristics and heuristic functions?
- Using heuristics in games
- Heuristic search algorithms
- Properties and issues
Heuristic Search

Basic classifications of heuristic search algorithms
- Iterative improvement algorithms, e.g.,
  - Hill climbing
  - Simulated annealing
- Best-First Search, e.g.,
  - Greedy search
  - A*
- Memory bounded search, e.g.,
  - Iterative deepening A* (IDA*)
  - Simplified memory-bounded A* (SMA*)

Hill Climbing

Use generate and test problem-solving strategy

Algorithm: (Simple) Hill Climbing
1. Let current node be an initial state.
2. If current node is a goal state, STOP (solution found)
3. If current node has no more successor, STOP (no solution found)
4. If successor is better than the current node, make the successor a current node. Go to 2.

Note: “better” can mean lower or higher value. E.g.,
If h(n) = cost from n to a goal state then “better” = lower value
If h(n) = profit in state n then "better" = higher value
**Example: Hill Climbing**

- \( h(n) = \text{profit in state } n \)

![Diagram](image)

Solution: ACDE

Solutions depend on order of node expansion

**Gradient Search**

- A variation of hill climbing (steepest ascent):
  - Replace “better” by “best” in step 4 of the simple hill climbing algorithm.
  - I.e., making the best successor, that is better than current node, a new current node

![Diagram](image)

Solution: ACE

Solutions do not depend on order of node expansion
Hill Climbing

- May miss optimal solution

Hill Climbing: Example 2

What are solution paths from hill climbing algorithms?

Simple hill climbing: ABGJ but not optimal
Optimal solution: ADL
Gradient search: ABHK miss solution
If K is a goal state ... still the solution is non-optimal

Hill climbing returns “locally” optimal solution
Characteristics and Issues

- Hill climbing returns “locally” optimal solution

- Complexity:
  - Time Complexity = $O(d)$, where $d =$ longest path length in the tree
  - Space Complexity ?

- Failures in hill climbing
  - search is too local (more later)
  - bad heuristics

Example: Blockworld

Operators:  Put(X) puts block X on table
            Stack(X, Y) puts block X on top of block Y

Characteristics and Issues (cont)
Characteristics and Issues (cont)

Heuristic 1: (an object is either a block or a table)
- Add one for every block that is attached on the right object
- Subtract one for every block that is attached on the wrong object

```
A   D   C   B
\  \  /  /
|  |  |  |
0  2  0  0
```

Characteristics and Issues (cont)

Heuristic 2: (a support of X is a set of blocks supporting X)
- For each block that has correct support, add one for every block in the support
- For each block that has an incorrect support, subtract one for each block in the support

```
A   D   C   B
\  \  /  /
|  |  |  |
0  0  0  0
```

Heuristic 2: (a support of X is a set of blocks supporting X)
- For each block that has correct support, add one for every block in the support
- For each block that has an incorrect support, subtract one for each block in the support

```
A   D   C   B
\  \  /  /
|  |  |  |
0  0  0  0
```
**Characteristics and Issues (contd)**

**Drawbacks:** situations that no progress can be made

- **Foothill (local maxima):** always better than neighbors but not necessarily better than some other far away
- **Mesa (Plateau):** not possible to determine the best direction to move
- **Zig-zag (Ridges):** local max has a slope, can’t traverse a ridge by single moves

**Remedy:** Random-restart hill-climbing

- Backtrack to earlier node
- Big jumps to new search space
- Move to several directions at once

---

**Simulated Annealing**

- **Or Valley Descending** - a variation of hill climbing where “downhill” moves may be made at the beginning
- **Allows random moves to escape local max**
  - can temporarily move to a worse state to avoid being stuck, i.e.,

  \[
  \text{If state is better then move, else move with prob. } < 1, \text{ where the prob. of the “badness” of the move decreases exponentially}
  \]

- **Analogous to annealing metals to cool molten metal to solid**
  - A transition to a higher energy state in cooling process occurs with probability \( p \), where

  \[
  p = e^{-\Delta E / T}
  \]

  \( \Delta E = \) +ve change in energy level
  \( T = \) Temperature

  **Note:**
  \( p \) is propositional to \( T \)
  \( T \) initially increases then decreases
Simulated Annealing (contd)

Algorithm (sketched)
1. current node = an initial state. Initialize $T$ according to the annealing schedule
2. Best-so-far = current node
3. Repeat
   - Generate a successor X of a current state;
   - If X is a goal state then return it and STOP
   - else if X is better than current then begin current = X; best-so-far = X end
   - else make X a current state only with probability $p = e^{-\Delta E / T}$ where $\Delta E$ = change in values of current node and X;
   - Revise $T$ from the schedule
   Until a solution is found or current state has no more successor
4. Return Best-so-far (ancestors are traced for a solution path)
Heuristic Search

Basic classifications of heuristic search algorithms

- Iterative improvement algorithms, e.g.,
  - Hill climbing
  - Simulated annealing
- Best-First Search, e.g.,
  - Greedy search
  - A*
- Memory bounded search, e.g.,
  - Iterative deepening A* (IDA*)
  - Simplified memory-bounded A* (SMA*)

Best First Search

- Expand the node with lowest cost first to find a low-cost solution
- Uses heuristic cost evaluation function f(n) for node n
- Idea:
  - Loop until goal is found or no node left to generate
    - Choose the best of the nodes generated so far. If it is goal then quit.
    - Generate the successors of the chosen node
    - Add all new nodes to the set of nodes generated so far
**Example 1**

Best First Search Algorithm

**Algorithm:**

1. OPEN = [start]; CLOSED = []
2. If OPEN = [], exit with FAIL
3. Choose best node from OPEN, call it X
4. If X is a goal node, exit with a traced solution path
5. Remove X from OPEN to CLOSED
6. Generate successors of X (record X as their parent)
   - For each successor S of X
     - i) Evaluate f(S)
     - ii) If S is new (i.e., not on OPEN, CLOSED), add it to OPEN
     - iii) If S is not new, compare f(S) on this path to previous one
       - If previous is better or same, keep previous and discard S
       - Otherwise, replace previous by S.
       - If S is in CLOSED, move it back to OPEN
7. Go to 2

**Solution:**

<table>
<thead>
<tr>
<th>X</th>
<th>OPEN</th>
<th>CLOSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>A25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A25</td>
<td>B17,C15</td>
<td>A25</td>
</tr>
<tr>
<td>C15</td>
<td>D10,E4,B17</td>
<td>C15,A25</td>
</tr>
<tr>
<td>E4</td>
<td>F0,D10,B17</td>
<td>E4,C15,A25</td>
</tr>
<tr>
<td>F0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal: Solution ACEF
**Evaluation Cost Functions**

- **Start**
- **Goal**

- \( f(n) = \text{cost of state } n \text{ from start state to goal state} \)
- \( g(n) = \text{cost from start state to state } n \)
- \( h(n) = \text{cost from state } n \text{ to goal state} \)

- In Best First Search, an evaluation cost function of state \( n \)

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

- \( h'(n) \) is a heuristic estimate of \( h(n) \)

**Example 2**

- **Start**
- **Goal**

- \( h'(A) = 25 \)
- \( h'(F) = 0, \text{ etc.} \)

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

- Solution: ACDEF
**Best First Search: Example 2**

**Algorithm:**
1. OPEN = [start]; CLOSED = [];
2. If OPEN = [], exit with FAIL;
3. Choose best node from OPEN, call it X;
4. If X is a goal node, exit with a traced solution path;
5. Remove X from OPEN to CLOSED;
6. Generate successors of X (record X as their parent)
   For each successor S of X
   i) Evaluate \( f(S) = g(S) + h(S) \);
   ii) If S is new (i.e., not on OPEN, CLOSED), add it to OPEN
   iii) If S is not new, compare \( f(S) \) on this path to previous one
       If previous is better or same, keep previous and discard S
       Otherwise, replace previous by S.
   If S is in CLOSED, move it back to OPEN
7. Go to 2

Evaluation Cost Functions & Search

\( f(n) = \text{cost of state } n \text{ from start state to goal state} \)
\( g(n) = \text{cost from start state to state } n \)
\( h(n) = \text{cost from state } n \text{ to goal state} \)

In **Best First Search**, an evaluation cost function of state \( n \)
\( f'(n) = g(n) + h'(n) \), where \( f'(n) \) is estimate of \( f(n) \)
\( h'(n) \) is a heuristic estimate of \( h(n) \)

**Special cases:**
- **Uniform cost search:** \( f'(n) = g(n) \)
- **Breadth First Search:** \( f'(n) = g(n) \), where \( g = \text{depth} \)
- **Greedy search:** \( f'(n) = h'(n) \)
- **A*:** \( f'(n) = g(n) + h'(n) \) where \( h'(n) \) is admissible (later)

\( h(A) = 25, h(F) = 0, \text{ etc.} \)
Outline

- What are heuristics and heuristic functions?
- Using heuristics in games
- Heuristic search algorithms
- Properties and issues

Some properties

- Uniform Cost Search:
  - Minimizes $g(n)$
  - Expands node on the least cost path first
    - Gives solution with minimal distance traveled
  - Doesn’t direct search towards goal
- Greedy Search:
  - Minimizes $h'(n)$
  - Expands node closest to the goal \( \rightarrow \) Reaches the goal fast
  - Directs search towards goal disregarding traveling distance \( \rightarrow \) not optimal
- A*:
  - Minimizes $g(n) + h'(n)$
  - Balances both costs
**Admissibility**

h′(n) is given near the node, g(n) = depth(n)

Using f′(n) = g(n) + h′(n): Solution ACIJ

H is not generated

Suppose F is also a goal: Solution ABDEF

G, H, C, I, J not generated

Solution is not optimal – cost 5 instead of 3

**Problem:** h′(C) > h*(C) – h′ overestimates h

where h*(n) = cost on the optimal path from state n to goal state

Here h*(C) = 2 from an optimal path ACIJ

Suppose h′(C) = 2 → h′(C) ≤ h*(C):

Solution ACIJ – optimal, and F, G, H not generated

If h′(n) ≤ h*(n) ≤ h(n) for all n, then h′ is admissible (e.g., see ACIJ)

**Admissibility (contd)**

- If h′(n) = h*(n) then goal state is converged with no search
- If h′ overestimates h*
  → h′ is worse (larger) than its actual optimal cost
  (i.e., h′(n) > h*(n) for some n)
  → h′ is pessimistic and the node may not get chosen
  → could miss optimal solution (earlier example h′(C) = 4)
- If h′ never overestimates h* (i.e., h′(n) ≤ h*(n) for all n)
  → h′ is better than its actual optimal cost
  → h′ is optimistic and the node gets chosen

- Search satisfies admissible condition is admissible
- Admissibility guarantees optimal solution if it exists
- A* (satisfies admissible conditions) is complete and optimal
**Monotonicity**

- For \( h'(\text{goal}) = 0 \), \( h' \) is monotone if for all \( n \) and its descendent \( m \)
  - \( h'(n) \leq \text{cost}(n,m) + h'(m) \)
- The above implies \( f'(m) \geq f'(n) \)
  and thus (monotone non-decreasing gives) the name
- **Monotonicity** ~ locally admissible
  - reaching each state along optimal path
  - thus, it guarantees optimal path to any state
  - the first time the state is discovered

**Monotonicity (contd)**

- Consequences of monotone heuristics in search algorithms
  - can ignore a state discovered a second time
  - no need to update the path information
- Monotonicity implies admissibility
- Uniform cost search gives optimal solution when it is monotone
  - i.e., when \( \text{cost}(n,m) \geq 0 \)
**Examples**

- \( h_1(n) = 0 \) for all \( n \)
- \( h_2(n) = \) straight line distance from state \( n \) to a goal state
- \( h_1 \) and \( h_2 \) are admissible and monotonic
- The 8 puzzle: E.g.,
  - \( h_3(n) = \) the number of tiles that are out of place from the goal
  - \( h_4(n) = \) sum of distances out of place
  - \( h_5(n) = 2 \times \) the number of direct tile reversals

All are less than or equal to the number of moves required to move the tiles to the goal position (i.e., actual cost) \( \rightarrow \) all are admissible

**Informedness**

- Let \( h_1 \) and \( h_2 \) be heuristics in two A* algorithms, \( h_2 \) is more informed than \( h_1 \) if
  1. \( h_1(n) \leq h_2(n) \) for all \( n \), and
  2. \( h_2(m) < h_2(m) \) for some \( m \)

E.g., \( h_2 \) is more informed than \( h_1 \) for \( h_1(n) = 0 \) for all \( n \), and

\[ h_2(n) = \) estimate distance from state \( n \) to a goal state

- More informed heuristics reduce search space

  Example: Consider two A* searches:
  1. \( f(n) = \) depth\( (n) + h_1(n) \)
  2. \( f(n) = \) depth\( (n) + h_2(n) \)

Search 1 covers entire space
Search 2 covers partial space
Other heuristic searches

- Iterative improvement algorithms
  - Hill climbing: moves to a better state
  - Simulated annealing: random moves to escape local max
- Best-First Search
  - Greedy search: expands the node closest to the goal
  - A*: expands the node on the least-cost solution path
- Other search variations
  - Beam search
  - AO*
  - Memory bounded search
    - Iterative deepening A* (IDA*)
    - Simplified memory-bounded A* (SMA*)

Applications of search algorithms

Two important problem classes:
- Game playing (discussed earlier)
- Constraint Satisfaction Problems (CSP)
CSP

- **States** are defined by values of a set of variables
- **Goal** is a set of constraints of the values
- **Solution** specifies values of all variables such that the constraints are satisfied

**Real-world problems:**
- Design VLSI layout
- Scheduling problems

CSP (contd)

Search in CSP with:

1. \( x \neq 2 \)
2. \( x + y = z \)
3. \( y \neq 3 \)
4. \( x + y + z \neq 12 \)

Start state: \( (x, y, z) \)

- Propagate thru constraint 2: \( (2, y, z) \)
  - Forward checking: no \( y \) satisfies 3)
  - Backtrack

- Propagate thru constraint 2: \( (4, y, z) \)
  - \( y = 2 \)
  - Solution: \( (4, 2, 6) \)

\( x = 4 \)

\( y = 3 \)

\( (4, 3, 7) \)

Violates 4)

**Arc consistency:** delete \( y = 3, z = 7 \)

Backtrack
Techniques required
- Backtracking
- Forward checking
- Arc consistency which exhibits Constraint propagation

Efficiency in searching in CSP depends on selecting
- which variable
- what value

Heuristics in CSP:
- Most-constraining-variable → less future variable choices
- Least-constraining-value → more freedom for future values