Heuristic Search
(Informed Search)
Heuristic Search

• In **uninformed search**, we don’t try to evaluate which of the nodes on the frontier/OPEN are most promising. We never “look-ahead” to the goal.

  E.g., in uniform cost search we always expand the cheapest path. We don’t consider the cost of getting to the goal from the end of the current path.

• Often we have some other knowledge about the merit of nodes, e.g., going the wrong direction in Romania.
Heuristic Search

**Merit** of a frontier/OPEN node: different notions of merit.

- If we are concerned about the *cost of the solution*, we might want a notion of merit of how costly it is to get to the goal from that search node.
- If we are concerned about *minimizing computation* in search we might want to consider how easy it is to find the goal from that search node.
- We will focus on the “*cost of solution*” notion of merit.
Heuristic Search

• The idea is to develop a domain specific heuristic function $h(n)$.

• $h(n)$ guesses the cost of getting to the goal from node $n$ (the cost of completing the path that is captured by the state of node $n$).

• There are different ways of guessing this cost in different domains. I.e., heuristics are domain specific.
“As the crow flies” – Straight line heuristic

On the map, the numbers between cities represent the driving distance between cities on potentially wiggly roads, even though they are drawn as straight lines. Contrast this to the line-of-sight/``as the crow flies” distance which ignores wiggles in the road, cliffs, bridges, and assumes you can just drive in a straight line from one city to another.
Planning a path from Arad to Bucharest, we can utilize the straight line distance from each city to our goal as a heuristic/guess of the actual distance. This lets us plan our trip by picking cities at each time point that minimize the distance to our goal.
Heuristic Search

• If $h(n_1) < h(n_2)$ this means that we guess that it is cheaper to get to the goal from $n_1$ than from $n_2$.

• We require that
  • $h(n) = 0$ for every node $n$ whose state satisfies the goal.
  • Zero cost of getting to a goal node from $n$. 
Using only $h(n)$: Greedy best-first search (Greedy BFS)

- We use $h(n)$ to rank the nodes on the frontier/OPEN.
  - Always expand node with lowest $h$-value.
  - We are greedily trying to achieve a low cost solution.

- However, this method **ignores the cost of getting to $n$**, so it can be lead astray exploring nodes that cost a lot to get to but seem to be close to the goal:

  - step cost = 10
  - step cost = 100

\[
\begin{align*}
\text{[S]} & \\
\text{[n3,n1]} & \\
\text{[Goal, n1]} & \\
\h(n1) = 70 & \\
\h(n3) = 50 & 
\end{align*}
\]
Using only $h(n)$: Greedy best-first search (Greedy BFS).

- We use $h(n)$ to rank the nodes on the frontier.
  - Always expand node with lowest $h$-value.
  - We are greedily trying to achieve a low cost solution.

- However, this method **ignores the cost of getting to $n$**, so it can be lead astray exploring nodes that cost a lot to get to but seem to be close to the goal:

  - step cost = 10
  - step cost = 100

  \[
  h(n1) = 70 \\
  h(n3) = 50
  \]

  (Greedy BFS is
  - Incomplete
  - not optimal)
Greedy best-first search example

When you’re at Sibiu and contemplating whether to go to Fagaras or RV, the heuristic value of the successor nodes, i.e., the h value guess of the cost is: \( h(\text{Fagaras}) = 178 \) and \( h(\text{RV}) = 193 \), so Fagaras looks like the better choice, but ...

Actual Cost (Arad-Sibiu-RV-Pitesli-Bucharest):  
\[ 140 + 80 + 97 + 101 = 140 + 278 = 418 \]

Actual Cost (Arad-Sibiu-Fagaras-Bucharest):  
\[ 140 + 99 + 211 = 140 + 310 = 450 \]
A* search

- Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from n.

- Define an evaluation function $f(n)$
  
  $f(n) = g(n) + h(n)$
  
  - $g(n)$ is the cost of the path to node $n$
  - $h(n)$ is the heuristic estimate of the cost of getting to a goal node from $n$.

- Always expand the node with lowest $f$-value on the frontier.

- The $f$-value is an estimate of the cost of getting to the goal via this node (path).
A* example

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

= actual cost to n + heuristic estimate of cost from n to the goal
**A* example**

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

= actual cost to n + heuristic estimate of cost from n to the goal
A* example

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

= actual cost to n + heuristic estimate of cost from n to the goal
A* example

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

= actual cost to n + heuristic estimate of cost from n to the goal
A* example

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

= actual cost to n + heuristic estimate of cost from n to the goal
A* Example

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

= actual cost to n + heuristic estimate of cost from n to the goal
A* search

• Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from n.

• Define an evaluation function \( f(n) \)
  \[
  f(n) = g(n) + h(n)
  \]
  • \( g(n) \) is the cost of the path to node n
  • \( h(n) \) is the heuristic estimate of the cost of getting to a goal node from n.

• Always expand the node with lowest f-value on the frontier.

• The f-value is an estimate of the cost of getting to the goal via this node (path).
Conditions on $h(n)$

• We want to analyze the behavior of the resultant search.
  • Completeness, time and space, optimality?

• To obtain such results we must put some further conditions on the heuristic function $h(n)$ and the search space.
Conditions on $h(n)$: Admissible

- We always assume that $c(n_1 \rightarrow n_2) \geq \epsilon > 0$. The cost of any transition is greater than zero and can’t be arbitrarily small.

- Let $h^*(n)$ be the cost of an optimal path from $n$ to a goal node ($\infty$ if there is no path). Then an admissible heuristic satisfies the condition

  $$h(n) \leq h^*(n)$$

  Admissible heuristic $h$ always underestimates the true cost to reach the goal. i.e., it is optimistic 😊

- Hence
  - $h(g) = 0$, for any goal note, $g$
  - $h^*(n) = \infty$ if there is not path from $n$ to a goal node
Consistency (aka monotonicity)

- Is a stronger condition than \( h(n) \leq h^*(n) \).

- A monotone/consistent heuristic satisfies the triangle inequality (for all nodes \( n_1, n_2 \)):
  \[
  h(n_1) \leq c(n_1 \rightarrow n_2) + h(n_2)
  \]

- Note that there might be more than one transition (action) between \( n_1 \) and \( n_2 \), the inequality must hold for all of them.

- Note that monotonicity implies admissibility.
  - \((\forall n_1, n_2)\ h(n_1) \leq c(n_1 \rightarrow n_2) + h(n_2) \Rightarrow (\forall n)\ h(n) \leq h^*(n)\)
Intuition behind admissibility

$h(n) \leq h^*(n)$ means that the search won’t miss any promising paths.

- If it really is cheap to get to a goal via $n$ (i.e., both $g(n)$ and $h^*(n)$ are low), then $f(n) = g(n) + h(n)$ will also be low, and the search won’t ignore $n$ in favour of more expensive options.

- This can be formalized to show that admissibility implies optimality.
Intuition behind monotonicity

\[ h(n_1) \leq c(n_1 \rightarrow n_2) + h(n_2) \]

- This says something similar, but in addition one won’t be “locally” mislead. See next example.
Consistency $\Rightarrow$ Admissible

• Assume consistency: $h(n_1) \leq c(n_1 \rightarrow n_2) + h(n_2)$

Prove admissible: $h(n) \leq h^*(n)$

Proof:
If no path exists from $n$ to a goal then $h^*(n) = \infty$ and $h(n) \leq h^*(n)$
Else let $n \rightarrow n_1 \rightarrow \ldots \rightarrow n^*$ be an OPTIMAL path from $n$ to a goal. Note the cost of this path is $h^*(n)$, and each subpath ($n_i \rightarrow \ldots \rightarrow n^*$) has cost equal to $h^*(n_i)$.
Prove $h(n) \leq h^*(n)$ by induction on the length of this optimal path.

**Base Case: $n = n^*$**

[optimal path length = 0]

By our conditions on $h$, $h(n) = 0 \leq h(n^*) = 0$

**Induction Hypothesis: $h(n_1) \leq h^*(n_1)$**

$h(n) \leq c(n \rightarrow n_1) + h(n_1)$

$\leq c(n_1 \rightarrow n_1) + h^*(n_1)$

$= h^*(n)$

[consistency]

[defn $h^*$]
Example: admissible but nonmonotonic

The following $h$ is not consistent (i.e., not monotone) since $h(n2) > c(n2 \rightarrow n4) + h(n4)$. But it is admissible.

→ step cost = 200
→ step cost = 100

We do find the optimal path as the heuristic is still admissible. But we are mislead into ignoring $n2$ until after we expand $n1$. 

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

\[
\{S\} \rightarrow \{n1 \ [200+50=250], \ n2 \ [200+100=300]\}\n\rightarrow \{n2 \ [100+200=300], \ n3 \ [400+50=450]\}\n\rightarrow \{n4 \ [200+50=250], \ n3 \ [400+50=450]\}\n\rightarrow \{\text{goal} \ [300+0=300], \ n3 \ [400+50=450]\}\]
Example: admissible but nonmonotonic

The following $h$ is **not consistent (i.e., not monotone)** since $h(n2) > c(n2 \rightarrow n4) + h(n4)$. **But it is admissible.**

- step cost = 200
- step cost = 100

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

- $\{S\} \rightarrow \{n1 \ [200+50=250], \ n2 \ [100+200=300]\}$
- $\rightarrow \{n2 \ [100+200=300], \ n3 \ [400+50=450]\}$
- $\rightarrow \{n4 \ [200+50=250], \ n3 \ [400+50=450]\}$
- $\rightarrow \{\text{goal} \ [300+0=300], \ n3 \ [400+50=450]\}$

We **do find** the optimal path as the heuristic is still admissible. **But** we are mislead into ignoring $n2$ until after we expand $n1$. 
“As the crow flies” – Straight line heuristic
• Most admissible heuristics are also monotone. (Indeed it’s hard to find an admissible heuristic that is not monotone!)
Consequences of monotonicity

1. The f-values of nodes along a path must be non-decreasing.

Let <Start → n1 → n2... → nk> be a path.

We claim that
\[ f(n_i) \leq f(n_{i+1}) \]

Proof:
\[
\begin{align*}
f(n_i) &= c(\text{Start} \rightarrow \cdots \rightarrow n_i) + h(n_i) \\
&\leq c(\text{Start} \rightarrow \cdots \rightarrow n_i) + c(n_i \rightarrow n_{i+1}) + h(n_{i+1}) \quad \text{[monotonicity]} \\
&= c(\text{Start} \rightarrow \cdots \rightarrow n_i \rightarrow n_{i+1}) + h(n_{i+1}) \\
&= g(n_{i+1}) + h(n_{i+1}) \\
&= f(n_{i+1}).
\end{align*}
\]
Consequences of monotonicity

2. If n2 is expanded after n1, then \( f(n_1) \leq f(n_2) \)
   (the f-value increases monotonically)

Proof (2 cases):

- If \( n_2 \) was on the frontier/OPEN when \( n_1 \) was expanded,
  then \( f(n_1) \leq f(n_2) \) otherwise we would have expanded \( n_2 \).
- If \( n_2 \) was added to the frontier/OPEN after \( n_1 \)'s expansion,
  then let \( n \) be an ancestor of \( n_2 \) that was present when \( n_1 \) was
  being expanded (this could be \( n_1 \) itself). We have \( f(n_1) \leq f(n) \)
  since A* chose \( n_1 \) while \( n \) was present in the frontier/OPEN.
  Also, since \( n \) is along the path to \( n_2 \), by property (1) we have
  \( f(n) \leq f(n_2) \). So, we have \( f(n_1) \leq f(n_2) \).

1) The f-values of nodes along a path must be non-decreasing.
2) If \( n_2 \) is expanded after \( n_1 \), then \( f(n_1) \leq f(n_2) \)
**Consequences of monotonicity**

**Corollary:** the sequence of f-values of the nodes expanded by A* is non-decreasing. I.e, If n2 is expanded after (not necessarily immediately after) n1, then f(n1) ≤ f(n2)

(the f-value of expanded nodes is monotonic non-decreasing)

**Proof:**

- If n2 was on frontier/OPEN when n1 was expanded, then f(n1) ≤ f(n2) otherwise we would have expanded n2.
- If n2 was added to frontier/OPEN after n1's expansion, then let n be an ancestor of n2 that was present when n1 was being expanded (this could be n1 itself). We have f(n1) ≤ f(n) since A* chose n1 while n was present on frontier/OPEN. Also, since n is along the path to n2, by property (1) we have f(n) ≤ f(n2). So, we have f(n1) ≤ f(n2).
Consequences of monotonicity

3. When n is expanded every path with lower f-value has already been expanded.

- **Proof:** Assume by contradiction that there exists a path <Start, n0, n1, ni-1, ni, ni+1, ..., nk> with f(nk) < f(n) and ni is its last expanded node.
  - ni+1 must be on the frontier/OPEN while n is expanded, so
    a) by (1) f(ni+1) ≤ f(nk) since they lie along the same path.
    b) since f(nk) < f(n) (given) so we have f(ni+1) < f(n) (from a)
    c) by (2) f(n) ≤ f(ni+1) because n is expanded before ni+1.
  - **Contradiction from b&c!**

1) The f-values of nodes along a path must be non-decreasing.
2) If n2 is expanded after n1, then f(n1) ≤ f(n2)
3) When n is expanded every path with lower f-value has already been expanded.
Consequences of monotonicity

4. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Proof:

- Let PATH1 = <Start, n0, n1, ..., nk, n> be the first path to n found. We have f(path1) = c(PATH1) + h(n).
- Let PATH2 = <Start, m0, m1, ..., mj, n> be another path to n found later. we have f(path2) = c(PATH2) + h(n).
- By property (3) and its corollary, f(path1) ≤ f(path2)
- hence: c(PATH1) ≤ c(PATH2)

1) The f-values of nodes along a path must be non-decreasing.
2) If n2 is expanded after n1, then f(n1) ≤ f(n2)
3) When n is expanded every path with lower f-value has already been expanded.

Corollary: the sequence of f-values of the nodes expanded by A* is non-decreasing. I.e, If n2 is expanded after (not necessarily immediately after) n1, then f(n1) ≤ f(n2)

(the f-value of expanded nodes is monotonic non-decreasing)
Consequences of monotonicity

Complete.

- Yes, consider a least cost path to a goal node
  - SolutionPath = <Start → n1 → ... → G> with cost c(SolutionPath)
  - Since each action has a cost ≥ ε > 0, there are only a finite number of paths that have cost ≤ c(SolutionPath).
  - All of these paths must be explored before any path of cost > c(SolutionPath).
  - So eventually SolutionPath, or some equal cost path to a goal must be expanded.

Time and Space complexity.

- When h(n) = 0, for all n, h is monotone. (a very *un*informative heuristic!!!)
  - A* becomes uniform-cost search!
  - It can be shown that when h(n) > 0 for some n, the number of nodes expanded can be no larger than uniform-cost.
  - Hence the same bounds as uniform-cost apply. (These are worst case bounds). Still exponential unless we have a very good h!
  - In real world problems, we run out of time and memory! IDA* can sometimes be used to address memory issues, but IDA* isn’t very good when many cycles are present.
Consequences of monotonicity

Optimality

- Yes, by (4) the first path to a goal node must be optimal.

4. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Cycle Checking

- We can use a simple implementation of cycle checking (multiple path checking)---just reject all search nodes visiting a state already visited by a previously expanded node. By property (4) we need keep only the first path to a node, rejecting all subsequent paths.
Search generated by monotonicity

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Inside each counter, the f values are less than or equal to counter value!

- For uniform cost search, bands are "circular".
- With more accurate heuristics, bands stretch out more toward the goal.
Admissibility without monotonicity

When “h” is admissible but not monotonic.

- Time and Space complexity remain the same. Completeness holds.
- Optimality still holds (without cycle checking), but need a different argument: don’t know that paths are explored in order of cost.

Proof (by contradiction) of optimality (without cycle checking):

- Assume the goal path <S,...,G> found by A* has cost bigger than the optimal cost: i.e. $C^*(G) < f(G)$.
- There must exist a node n in the optimal path that is still in the frontier.
- We have: $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^*(G) < f(G)$
- Therefore, f(n) must have been selected before G by A*. contradiction!
Admissibility without monotonicity

What about Cycle Checking?

- No longer guaranteed we have found an optimal path to a node *the first time* we visit it.

- So, cycle checking might not preserve optimality.
  - To fix this: for previously visited nodes, must remember cost of previous path. If new path is cheaper must explore again.

- contours of monotonic heuristics don’t hold.
Space Problems with A*

- A* has the same potential space problems as BFS or UCS

- IDA* - Iterative Deepening A* is similar to Iterative Deepening Search and similarly addresses space issues.
Objective: reduce memory requirements for A*

- Like iterative deepening, but now the “cutoff” is the f-value (g+h) rather than the depth.
- At each iteration, the cutoff value is the smallest f-value of any node that exceeded the cutoff on the previous iteration.
- Avoids overhead associated with keeping a sorted queue of nodes.
- Two new parameters:
  - `curBound` (any node with a bigger f-value is discarded)
  - `smallestNotExplored` (the smallest f-value for discarded nodes in a round) when frontier/OPEN becomes empty, the search starts a new round with this bound.
  - Easier to expand all nodes with f-value EQUAL to the f-limit. This way we can compute “smallestNotExplored” more easily.
Constructing Heuristics
Building Heuristics: Relaxed Problem

- One useful technique is to consider an easier problem, and let \( h(n) \) be the cost of reaching the goal in the easier problem.

8-Puzzle

- Can move a tile from square A to B if
  - A is adjacent (left, right, above, below) to B
  - and B is blank
Building Heuristics: Relaxed Problem

8-Puzzle moves (continued)

- Can move a tile from square A to B if
  - A is adjacent (left, right, above, below) to B
  - and B is blank

- Can relax some of these conditions
  1. can move from A to B if A is adjacent to B (ignore whether or not position is blank)
  2. can move from A to B if B is blank (ignore adjacency)
  3. can move from A to B (ignore both conditions).
# Building Heuristics: Relaxed Problem

- **#3** “can move from A to B (ignore both conditions)”.
  leads to the **misplaced tiles heuristic**.
  - To solve the puzzle, we need to move each tile into its final position.
  - Number of moves = number of misplaced tiles.
  - Clearly $h(n) = $ number of misplaced tiles $\leq h^*(n)$ the cost of an optimal sequence of moves from $n$.

- **#1** “can move from A to B if A is adjacent to B (ignore whether or not position is blank)”
  leads to the **manhattan distance heuristic**.
  - To solve the puzzle we need to slide each tile into its final position.
  - We can move vertically or horizontally.
  - Number of moves = sum over all of the tiles of the number of vertical and horizontal slides we need to move that tile into place.
  - Again $h(n) = $ sum of the manhattan distances $\leq h^*(n)$
    - in a real solution we need to move each tile at least that far and we can only move one tile at a time.
Building Heuristics: Relaxed Problem

The **optimal** cost to nodes in the relaxed problem is an **admissible heuristic** for the original problem!

**Proof Idea:** the optimal solution in the original problem is a solution for relaxed problem, therefore it must be at least as expensive as the optimal solution in the relaxed problem.

So admissible heuristics can sometimes be constructed by finding a relaxation whose optimal solution can be **easily computed**.
Building Heuristics: Relaxed Problem

Comparison of IDS and A* (average total nodes expanded):

<table>
<thead>
<tr>
<th>Depth</th>
<th>IDS</th>
<th>A*(Misplaced) h1</th>
<th>A*(Manhattan) h2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>47,127</td>
<td>93</td>
<td>39</td>
</tr>
<tr>
<td>14</td>
<td>3,473,941</td>
<td>539</td>
<td>113</td>
</tr>
<tr>
<td>24</td>
<td>---</td>
<td>39,135</td>
<td>1,641</td>
</tr>
</tbody>
</table>

Let $h_1$=Misplaced, $h_2$=Manhattan

- Does $h_2$ always expand fewer nodes than $h_1$?
  - Yes! Note that $h_2$ dominates $h_1$, i.e. for all $n$: $h_1(n) \leq h_2(n)$.
  - Therefore, among several admissible heuristic the one with highest value expands the fewest nodes. Is it the fastest?
Building Heuristics: Pattern databases.

- Admissible heuristics can also be derived from solution to **subproblems:** Each state is mapped into a partial specification, e.g. in 15-puzzle only **position of specific tiles matters.**

| Here are goals for two sub-problems (called Corner and Fringe) of 15-puzzle. |
| Note the location of BLANK! |

![Fig. 2. The Fringe and Corner Target Patterns.](image)

- By searching **backwards** from these goal states, we can compute the distance of any configuration of these tiles to their goal locations. We are ignoring the identity of the other tiles.

- For any state n, the number of moves required to get these tiles into place form a lower bound on the cost of getting to the goal from n.
Building Heuristics: Pattern databases.

• These configurations are stored in a database, along with the number of moves required to move the tiles into place.

• The maximum number of moves taken over all of the databases can be used as a heuristic.

• On the 15-puzzle
  • The fringe data base yields about a 345 fold decrease in the search tree size.
  • The corner data base yields about 437 fold decrease.

• Sometimes disjoint patterns can be found, then the number of moves can be added rather than taking the max (if we only count moves of the target tiles).
Local Search

• So far, we keep the paths to the goal.
• For some problems (like 8-queens) we don’t care about the path, we **only care about the solution.** Many real problem like Scheduling, IC design, and network optimizations are of this form.
• Local search algorithms operate using a single *current state* and generally move to neighbors of that state.
• There is an **objective function** that tells the value of each state. The goal has the highest value (global maximum).
• Algorithms like **Hill Climbing** try to move to a neighbour with the highest value.
• Danger of being stuck in a *local maximum*. So some randomness is added to “shake” out of local maxima.
• **Simulated Annealing:** Instead of the best move, take a random move and if it improves the situation then always accept, otherwise accept with a probability <1.
• [If interested read these two algorithms from the R&N book].