# 2 Heuristic Search

| Strategy | Frontier Selection | Halts? | Space | Time |
| --- | --- | --- | --- | --- |
| Depth-first | Last node added | No | Linear | Exp |
| Breadth-first | First node added | Yes | Exp | Exp |
| Lowest-cost-first | min $cost(n)$ | Yes | Exp | Exp |
| Greedy Best-first | min $h(n)$ | No | Exp | Exp |
| A* | min $cost(n) + h(n)$ | Yes | Exp | Exp |

*2 Heuristic Search-20250525140945825.webp*

# TL;DR

For A*:

- the heuristic $h(n)$ is an estimate of the cost from $n$ to the goal node
- any heuristic must at least be **admissible** -> it must never overestimate the true cost
- if a heuristic is **consistent**, then $h(a) - h(b) \leq \text{edge}(a, b)$ for all $a$, $b$
  - this is the monotone condition
  - consistency -> admissibility
  - A* no longer revisits nodes
- the algorithm itself is just Dijkstra's with a heuristic

# Heuristic function $h(n)$

A search heuristic $h(n)$ is an estimate of the cost of the cheapest path from node $n$ to a goal node.

A good $h(n)$ has these properties:

1. Problem specific
2. Non-negative
3. $h(\text{goal node}) = 0$
4. $h(n)$ must be easy to compute

# Lowest Cost First Search (LCFS)

Prioritize expanding the node with the lowest path cost $g(n)$ from the start.

This is just Dijkstra's algorithm. See Dijkstra's algorithm.

# Greedy Best First Search

BFS but the queue is a heap ordered by the heuristic $h(n)$.

```
def greedy_best_first_search(start, goal, neighbors_fn, h_fn):
    # Priority queue with (h(n), node)
    open_set = [(h_fn(start), start)]
```

```python
    came_from = {start: None}
    visited = set()

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            return reconstruct_path(came_from, current)

        visited.add(current)

        for neighbor in neighbors_fn(current):
            if neighbor not in visited:
                visited.add(neighbor)
                came_from[neighbor] = current
                heapq.heappush(open_set, (h_fn(neighbor), neighbor))

    return None  # No path found
```

**Pros**:

- Super easy to implement and fast if the heuristic is accurate
- Can use less memory than regular BFS or A* (not guaranteed)

**Cons**:

- Not optimal - if the heuristic is misleading it may pick a worse path
- Not complete - may end up in a cycle or dead end

# A*

> "It's just Dijkstra's but we also factor in the heuristic so that we don't explore paths that are obviously farther away from the goal."

A* is similar to Dijkstra's but when we calculate costs we also factor in a heuristic so $c(n) = g(n) + h(n)$. Where $g(n)$ is the normal Dijkstra's cost.

```
open_heap = []
g_score = default_inf()
g_score[start] = 0
heapq.heappush(open_heap, (h(start, goal), start))  # f = g + h = h at start

while open_heap:
    f_curr, current = heapq.heappop(open_heap)

    if current == goal:
        return reconstruct_path(...)

    # stale entry check: if f_curr > g_score[current] + h(current, goal): continue

    for neighbor in neighbors(current):
        tentative_g = g_score[current] + cost(current, neighbor)
        if tentative_g < g_score[neighbor]:
```

```
            g_score[neighbor] = tentative_g
            came_from[neighbor] = current
            f_neighbor = tentative_g + h(neighbor, goal)
            heapq.heappush(open_heap, (f_neighbor, neighbor))
```

Implementation notes:

- **`f_cost` is used only to order the min heap**
- We compare only `g_cost` scores when deciding to update path parents
- In practice, a minimum `g_cost` tracker is needed as well to detect stale heap entries (this must be in a dict and in each heap entry)
  - We may push multiple entries of the same node into the heap
  - Only the lowest cost one is up to date (valid)

---

**How do we choose a heuristic function?**

1. The heuristic MUST be admissible.
2. The heuristic SHOULD be consistent.

**The heuristic function $h(n)$ is always defined as an estimate of the cheapest cost from $n$ to the goal node.**

# Admissibility

The heuristic function *must be admissible* - it must never **over** estimate the cost of the cheapest path from $n$ to the goal.

In other words: $h(n) \leq \text{true\_optimal\_cost}(n, \text{goal node}) = h^*(n)$

*If the heuristic is admissible, then A* is optimal*. (Theorem of A* Optimality)

---

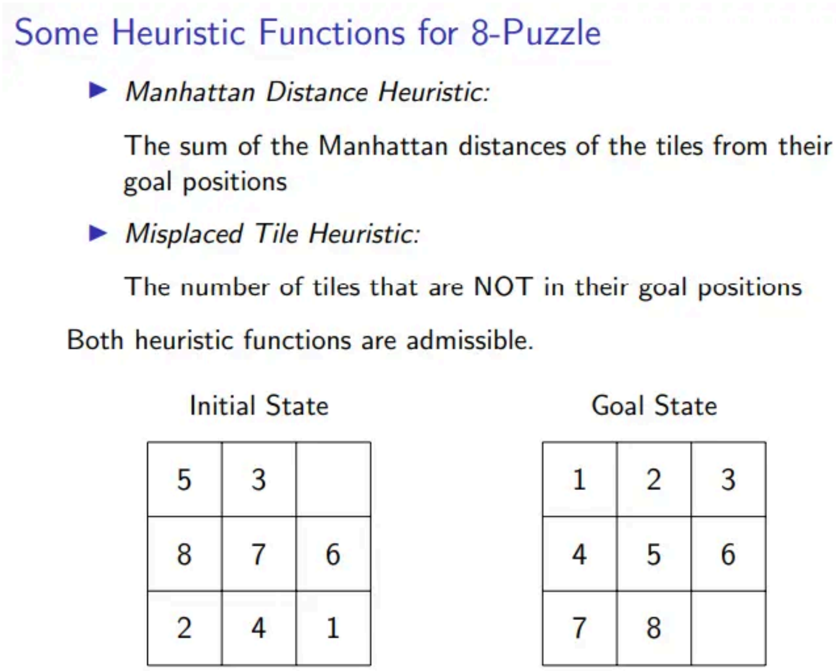
What happens if it over estimates?

- A* thinks a good path is bad
- Explores a bad path first, ignoring a good one

## Constructing an admissible heuristic

1. **Define a relaxed problem** - simplify or remove constraints on the original
2. **Solve the relaxed problem without search**
3. The cost of the optimal solution to the relaxed problem is an admissible heuristic for the original problem
   - **Why?** The cost of a solution with less constraints should usually be smaller than that of the original problem, which satisfies the admissibility requirement $h(n) \leq h^*(n)$
   - We should also prefer heuristics that are very different for different states

---




*2 Heuristic Search-20250526175602959.webp*

### Dominating heuristic

Assume we have two admissible heuristics $h_a(n)$ and $h_b(n)$.

$h_a(n)$ **dominates** $h_b(n)$ if the heuristic output for $h_a(n)$ is $\geq$ the output from $h_b(n)$ for all possible $n$.

**If a heuristic A dominates another heuristic B, the dominating heuristic A is better.**

## Consistency

A heuristic is consistent if it satisfies the *monotone condition* - for any two neighbor nodes $n$ and $m$:

$$h(m) - h(n) \leq \text{edge\_cost(m,n)}$$

A* with multi-path pruning is only optimal if there is consistent heuristic function.

Consistency => admissibility

Admissibility != consistency (but usually, admissible heuristics are consistent)

### Implications of consistency

If you find a heuristic which is consistent (and thus also admissible), then you get the following guarantees:

1. A* never needs to revisit nodes
2. A* becomes simpler and much more optimal

# Cycle pruning

Check that the nodes we are trying to visit are not already on the path.

- Time complexity: Linear if using a list, $O(1)$ using a list and stack

# Multipath pruning

Discard new paths to a node if we already found one.

- Use a visited set for $O(1)$ pruning
- Saves computation but increases space consumption

Can multipath pruning cause a search algorithm to fail to find the optimal solution? Say we keep the first path and prune the rest, but the first path is not the most optimal.

- **LCFS - NO**. Dijkstra's will always find the least cost path first
- **A* - YES**. The first path may not be optimal - A* with multi-path pruning is NOT optimal

## Making A* work with multipath pruning

To make A* work with multipath pruning, the heuristic must be consistent (satisfies the monotone condition)

**Monotone condition**: for all neighbor pairs $m$ and $n$,

$$h(m) - h(n) \leq \text{edge\_cost(m,n)}$$

---

Why this is the case:

### When does multi-path pruning not work?

Assuming we have a frontier $(s \to n, \cdots, s \to n')$, and we are exploring node $n$.

- If there exists another path through $n'$ to $n$ with lower f-value.
- 1) we have $h(n) + cost(n) > h(n) + cost(n') + cost(n', n)$, e.g. $cost(n) - cost(n') > cost(n', n)$
- 2) node $n$ is already explored, so $h(n) + cost(n) \leq h(n') + cost(n')$
- Combine these two, we have $h(n') - h(n) \geq cost(n) - cost(n') > cost(n', n)$
- Such scenario only happens when there exists two nodes $n$ and $n'$ with $h(n') - h(n) > cost(n', n)$.

*2 Heuristic Search-20250526181534955.webp*