Uninformed and Informed Search

Md Rifat Ahmmad Rashid, Associate Professor, EWU

1 Search algorithms

Search algorithms explore the state space by expanding states (applying actions to generate successors) and systematically considering different action sequences.

- **Search tree:** The set of all possible action sequences forms a **search tree** (with the initial state as the root) (Search). B**ecause real-world state** spaces are often enormous, **a key challenge is designing efficient search strategies that find solutions in a reasonable time.**

Basic Idea

- Offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states).

Algorithm 1 Tree Search Algorithm

```
1:  function TREE-SEARCH(problem)
2:      Initialize the frontier using the initial state of problem
3:      while true do
4:          if frontier is empty then
5:              return failure
6:          end if
7:          Choose a leaf node and remove it from the frontier
8:          if the node contains a goal state then
9:              return the corresponding solution
10:         end if
11:         Expand the chosen node, adding the resulting nodes to the frontier
12:     end while
13: end function
```

Frontier: all leaf nodes available for expansion at any given point.

Different data structures (e.g., FIFO, LIFO) for frontier can cause different orders of node expansion and thus produce different search algorithms.

2 Uninformed search

Uninformed search (or *blind search*) strategies use no problem-specific knowledge beyond the problem definition. They treat the state space as a black box, only using the goal test and the available actions. In this section, we discuss several uninformed algorithms: **Breadth-First Search**, **Depth-First Search**, **Uniform-Cost Search**, **Depth-Limited Search**, and **Iterative Deepening Search**. Each algorithm differs in how it explores the search tree, and we analyze their properties (completeness, optimality) and complexity.

2.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) expands the search tree **level by level**, exploring all states at depth 0, then depth 1, then depth 2, and so on

It uses a **first-in-first-out (FIFO) queue** for the frontier of nodes to expand: initially the queue contains the initial state, then at each step BFS removes the oldest node (shallowest unexpanded state)

and expands it by generating all successors, which are added to the back of the queue. Pseudocode for BFS can be given as:

Listing 1: BFS Pseudocode

```
BFS(problem):
    initialize an empty FIFO queue and add problem.initial_state
    while queue is not empty:
        node := dequeue(queue)
        if problem.goal_test(node.state) is true:
            return solution (path) for node
        for each action in problem.actions(node.state):
            child := result of applying action to node.state
            if child.state not yet visited:
                enqueue(child, queue)
    return failure  // if queue empties with no solution found
```

BFS will eventually find a solution if one exists, because it systematically explores shallower levels before deeper ones.

- **Completeness:** BFS is *complete* as long as the branching factor (b, maximum number of successors per state) is finite; it will find a goal state if one exists, since it expands the search tree in breadth until the goal depth
- **Optimality:** If all actions have equal cost (e.g. each step cost = 1), BFS is *optimal* – it always finds the shallowest goal, which is the least number of steps

  (More generally, BFS is optimal if path cost is a non-decreasing function of depth, such as when all step costs are identical

  However, BFS is **not optimal** in domains with varying step costs – it minimizes the number of actions, not the total cost. In such cases, Uniform-Cost Search should be used for optimality.
- **Time complexity:** In the worst case (when the goal is at depth d and all shallower states must be expanded), BFS runs in time $O(b^d)$, growing exponentially with the depth of the solution

  More precisely, it expands all nodes at depth 0, 1, ..., d, which is on the order of

  $$b^0 + b^1 + \cdots + b^d = O(b^d)$$

  (assuming $b > 1$).
- **Space complexity:** BFS must store all nodes in a given level (and the next level) in the frontier, so space complexity is also $O(b^d)$, which can be a limiting factor

This memory requirement (keeping the entire breadth of the tree) is often the bigger drawback of BFS. In summary, BFS is complete and finds shortest paths in terms of steps (optimal for equal step costs), but it can be very slow and memory-intensive for deep or large state spaces due to its exponential complexity.

2.2 Depth-First Search (DFS)

Depth-First Search (DFS) explores the search tree by always expanding the **deepest unexpanded node** first. It uses a **last-in-first-out (LIFO) stack** for the frontier: it goes down one path, expanding nodes deeper and deeper, and backtracks (pops from the stack) when it hits a dead-end or goal. In a recursive formulation, DFS will recursively explore successor states until no further progress can be made, then backtrack to the next alternative.

The behavior of DFS is fundamentally different from BFS. DFS may find a solution without exploring all shallower states, which can be advantageous if solutions are deep. However, it comes with trade-offs in completeness and optimality.

- **Completeness:** The *tree-search* version of DFS (which does not check for repeated states) is **not complete**, because it can get lost going down an infinitely long path or cycle and never return. Even in a finite state space, naive DFS might revisit states endlessly if cycles exist. The *graph-search* version of DFS (which marks visited states to avoid repetition) is complete for finite state spaces , but if the state space is infinite or has paths of unbounded length, DFS can fail to find a solution even if one exists (it could dive into an infinite branch and never explore others).

- **Optimality:** DFS is **not optimal** in general. It does not guarantee the shortest path is found – it will return the first solution it encounters, which could be arbitrarily deep or costly, even if a cheaper solution exists at a shallower depth.
- **Time complexity:** In the worst case, DFS may end up exploring all nodes in the state space up to some maximum depth m. For a finite state space (graph-search DFS), the time complexity is $O(|V| + |E|)$ in graph terms (linear in states and transitions), but if we consider a search tree of maximum depth m with branching factor b, the complexity can be expressed as $O(b^m)$ in the worst case . This can be much worse than BFS if m is significantly larger than the depth of the shallowest solution d.
- **Space complexity:** A major advantage of DFS is its space usage. It only needs to store the current path and some backtracking information. For a search tree of depth m, DFS requires space $O(b \times m)$ (or $O(m)$ if each node only needs to store its parent and depth) . This is **linear space** complexity, much smaller than BFS's exponential space requirement.

In practice, DFS is often preferred when memory is limited and the search space is very large, or when we suspect solutions may be found deep in the tree. However, one must accept the risk that DFS might miss shallower solutions or even fail to terminate if cycles/infinite paths exist.

2.3 Uniform-Cost Search (UCS)

Uniform-Cost Search (UCS) is an uninformed search that extends BFS to handle varying step costs. It explores the state space in order of increasing **path cost** $g(n)$ (the cost accumulated from the start to node n). UCS uses a **priority queue** (min-heap) ordered by path cost. At each step, it expands the frontier node with the lowest total cost so far. This is essentially Dijkstra's algorithm for shortest paths in a graph, and it guarantees finding the least-cost solution.

The algorithm for UCS is similar to BFS except that the queue is prioritized by g instead of depth: initially insert the initial state with cost 0; then repeatedly pop the lowest-cost node from the frontier. If it's a goal, return it; otherwise expand it and enqueue each successor with cumulative cost:

$$g(successor) = g(parent) + \text{cost}(parent, action, successor)$$

If a state is reached via a cheaper path, UCS updates the frontier with the lower cost. UCS applies the goal test when a node is *removed* from the priority queue (selected for expansion), not when it's first generated.

Algorithm 2 Uniform-Cost Search

```
1: procedure UCS(problem)
2:     Initialize an empty priority queue (min-heap)
3:     Insert the initial state with path cost 0
4:     Initialize an empty dictionary costs to store best costs for each state
5:     while priority queue is not empty do
6:         Remove the node with the lowest cost (n, g(n))
7:         if n is a goal state then
8:             return solution path
9:         end if
10:        for each action in problem.actions(n) do
11:            Generate successor node s
12:            Compute new cost g(s) = g(n) + cost(n, action, s)
13:            if s is not in costs or g(s) is lower than stored cost then
14:                Store g(s) in costs
15:                Insert s into priority queue with priority g(s)
16:            end if
17:        end for
18:    end while
19:    return failure
20: end procedure
```

Optimality: UCS is **optimal** – it will always find the least-cost solution (optimal path) as long as all step costs are nonnegative. UCS orders expansions by optimal partial paths.

Time Complexity: The worst-case number of nodes expanded is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

Space Complexity: Similar to time complexity, UCS must store the frontier which in the worst case could have on the order of the same number of nodes as expanded (exponential as well).

In practice, UCS can be more expensive than BFS if a solution with a small number of steps has a large total cost, because UCS will explore many cheaper paths even if they are longer in steps. However, UCS is indispensable for optimal pathfinding in weighted graphs (e.g. routing problems), where BFS would fail to find the true shortest path if costs differ.

Example: If we apply UCS to a route-finding problem in Fig. 1, it will essentially behave like Dijkstra's algorithm. For instance, given a road map with distances as costs, UCS will explore in increasing order of distance. It will expand states in rings outward from the start, but biased by cost – a longer road might be deferred if a shorter road leads elsewhere. UCS will find the shortest driving route, whereas BFS might find a route with fewer segments but longer distance. UCS and BFS coincide when all actions have equal cost.

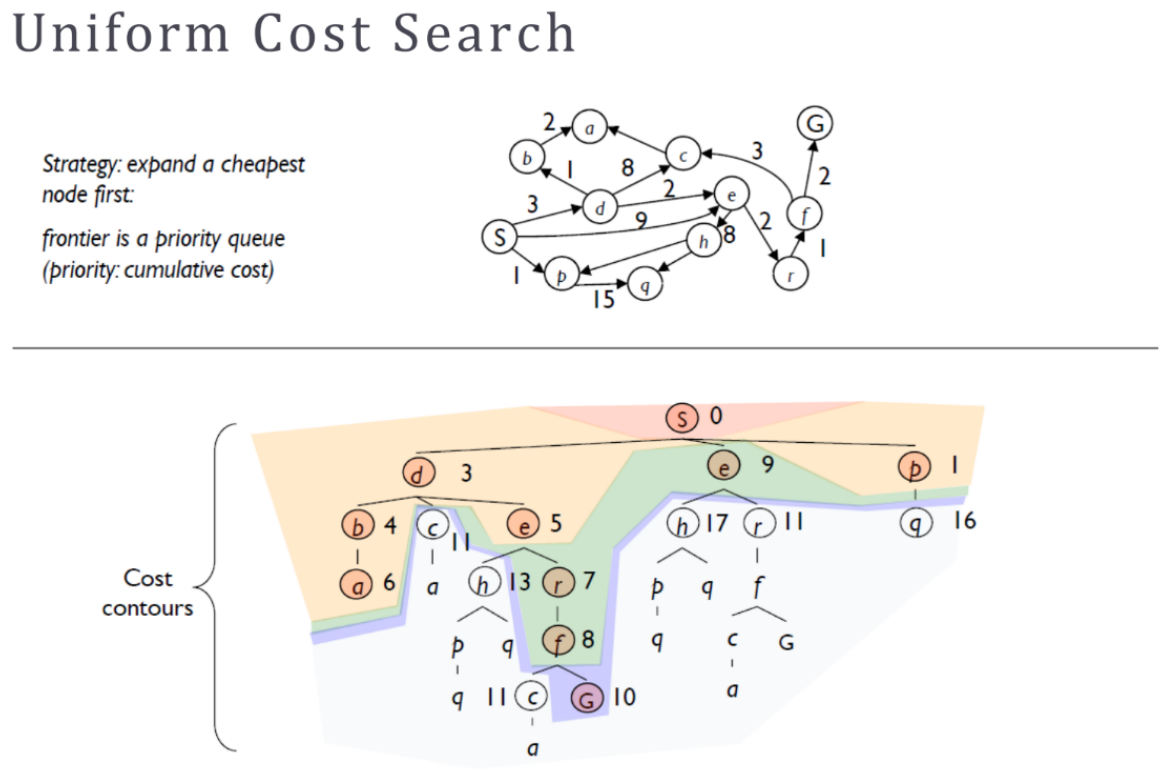

Figure 1: UCS Example

2.4 Depth-Limited Search (DLS)

One way to improve DFS's completeness is to impose a **depth limit**. **Depth-Limited Search (DLS)** performs DFS but cuts off recursion when a specified depth limit ℓ is reached. Nodes at depth ℓ are treated as if they have no successors (the search does not go deeper than ℓ). DLS prevents infinite descent in an unbounded tree, at the cost of possibly missing solutions deeper than the limit.

- If ℓ is at least the depth of a shallowest goal d, and the state space is finite up to that depth, then DLS will find a solution. If $\ell < d$, DLS is **incomplete** (it will terminate without finding the goal if the goal lies below the cutoff depth).
- If $\ell > d$, DLS may find a solution, but it is **not optimal** because it might find a deeper solution if a shallower one exists.
- **Time complexity:** $O(b^\ell)$ in the worst case, since it will generate the entire tree up to depth ℓ if no solution is found.

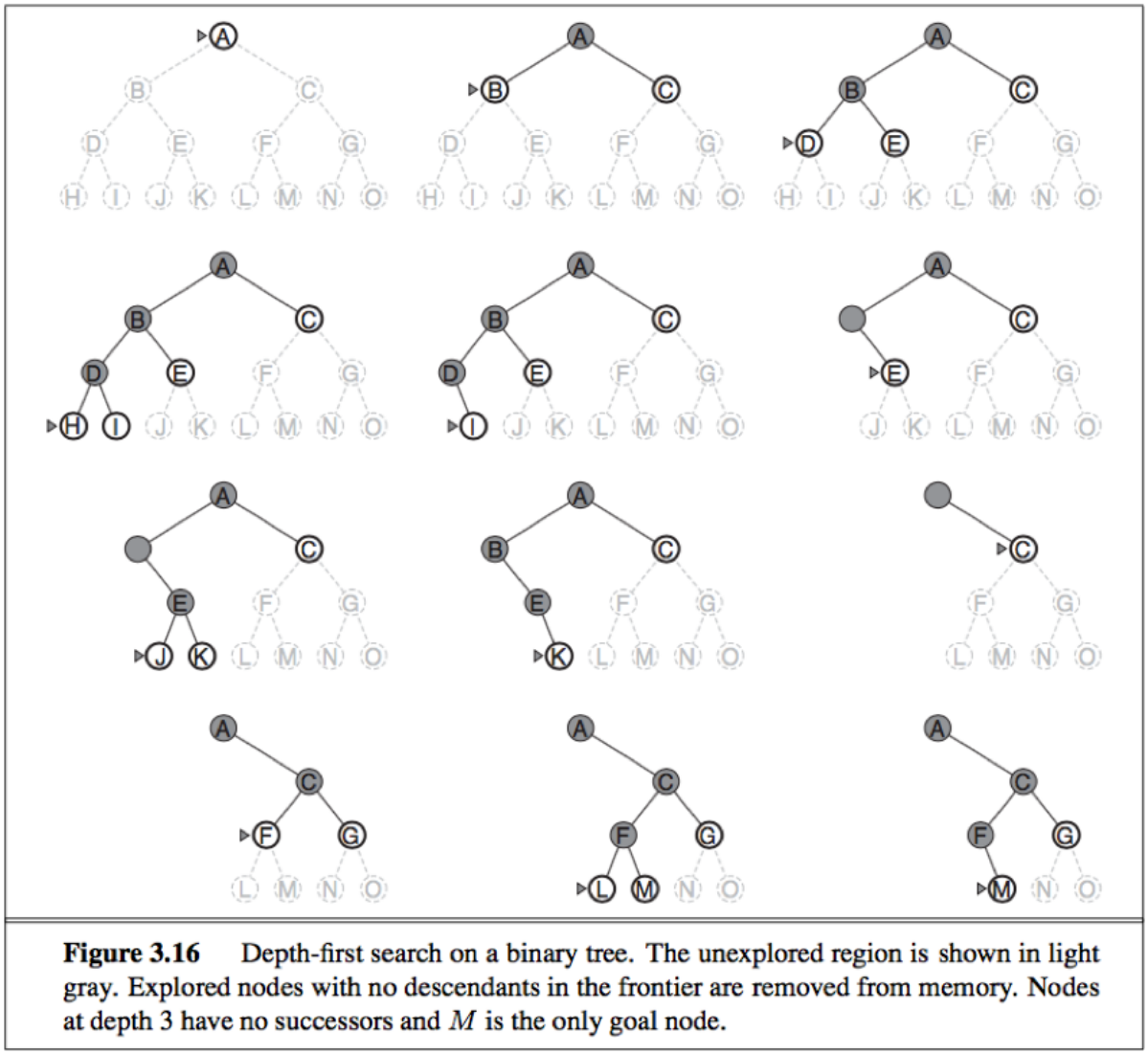

Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Figure 2: Depth Limited Search Example

- **Space complexity:** $O(b \times \ell)$ (or $O(\ell)$ with recursion stack), which is linear in the depth limit. This is significantly better than BFS's space usage.

Depth-limited search(Fig.2) is useful when we have prior knowledge of a reasonable search depth or to avoid infinite loops. However, choosing a good depth limit can be tricky without such knowledge.

2.5 Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) combines the benefits of BFS and DFS. It performs a series of depth-limited searches with increasing depth limits: first depth 0, then depth 1, then 2, and so on, until a solution is found. In effect, **Iterative Deepening DFS** finds the shallowest goal (like BFS) but with the memory footprint of DFS. IDS is **complete** (for finite branching factor) because it will eventually reach depth d where the goal lies. It is also **optimal** when step costs are equal (it finds the shallowest solution, same as BFS)

The time complexity of iterative deepening might seem worse due to repeated work, but it is actually $O(b^d)$ in the same order as BFS for exponential trees. This is because the nodes at the deepest level d dominate the node count. The overhead of re-expanding nodes at smaller depths in each iteration is relatively small (a fraction of the total nodes). In fact, for a branching factor b, the total nodes expanded in an IDS up to depth d is about:

$$b^0 + b^1 + \cdots + b^d = O(b^d)$$

Algorithm 3 Depth-Limited Search

```
1: procedure DLS(node, depth limit)
2:     if goal-test(node) then
3:         return solution path
4:     else if depth limit == 0 then
5:         return failure
6:     else
7:         for each action in problem.actions(node) do
8:             Generate successor node s
9:             Recursive call: DLS(s, depth limit - 1)
10:        end for
11:    end if
12: end procedure
```

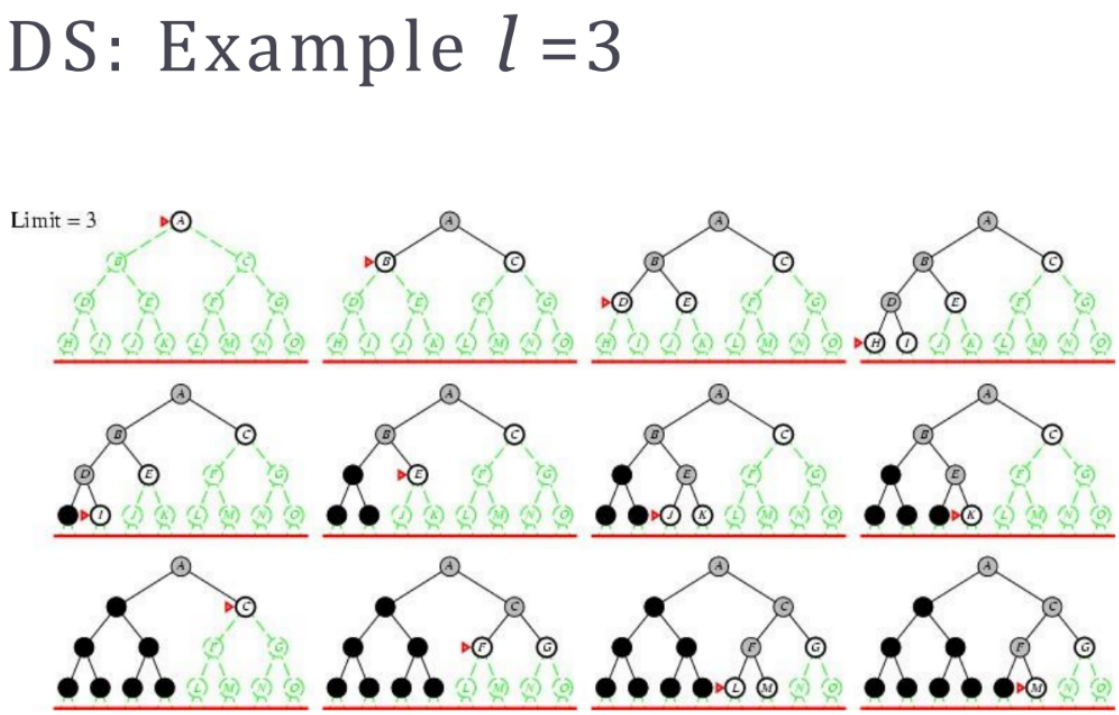

Figure 3: IDS Example with l=3

The constant factor overhead is $\approx \frac{b}{b-1}$ (for $b > 1$), meaning IDS is only about 10% more nodes than BFS if $b = 10$, for example.

Space complexity: IDS, like DFS, uses linear space $O(d)$. It does not need to store all nodes at depth d, only the current path and minimal information per depth.

Iterative deepening 3 is often the preferred uninformed search strategy when the state space is large and the depth of solution is not known. It is complete, optimal (for unit costs), and uses minimal memory. Its time cost is only slightly more than BFS. For these reasons, IDS is commonly used in puzzles like the 8-puzzle or 15-puzzle when no good heuristic is available, and in scenarios where the search depth is not known a prior.

2.6 Comparison of Uninformed Search Strategies

To summarize the properties of different uninformed search strategies:

- **BFS:** Complete; optimal for equal costs; time/space complexity $O(b^d)$.
- **DFS:** Not guaranteed complete (unless finite graph-search); not optimal; time complexity $O(b^m)$; space complexity $O(b \cdot m)$ (linear).

- **UCS:** Complete (with positive costs); optimal; worst-case time/space complexity $O(b^{1+C^*/\epsilon})$.
- **IDS:** Complete; optimal for equal costs; time complexity $O(b^d)$; space complexity $O(d)$. Combines benefits of BFS (optimality) and DFS (low memory).
- **DLS:** DFS with cutoff – incomplete if cutoff is too shallow; can avoid infinite search but must choose limit carefully.

3 Informed Search Algorithms

Uninformed methods do not exploit any information about how close a state is to the goal. **Informed search algorithms** use domain-specific knowledge in the form of a **heuristic function** to guide the search more efficiently. A **heuristic function** $h(n)$ estimates the cost of the cheapest path from a node n to a goal. For example, in a puzzle, it might count the tiles out of place, or in a navigation problem, it could be the straight-line distance to the destination. Informed algorithms prioritize exploring states that seem "closer" to the goal according to $h(n)$.

A good heuristic can dramatically reduce the number of nodes expanded, making search feasible in huge state spaces. We first define properties of heuristics, then discuss specific algorithms:

- An **admissible heuristic** never overestimates the true cost to reach a goal. Formally, $h(n) \leq h^*(n)$ for all nodes n, where $h^*(n)$ is the actual minimal cost from n to a goal. This optimistic property guarantees the heuristic is always a lower bound on the remaining cost.

  For example, the straight-line distance is admissible in route planning because actual road distance is longer or equal.

- A **consistent (or monotonic) heuristic** additionally satisfies the triangle inequality: for every node n and every successor n' reached by action a with cost $c(n, a, n')$, the heuristic obeys

  $$h(n) \leq c(n, a, n') + h(n')$$

  Consistency implies that $h(n)$ is also admissible and that the estimated cost along a path never decreases as you go deeper (so $f(n) = g(n) + h(n)$ is non-decreasing along any path). Consistent heuristics greatly simplify search because they guarantee that once a state has been expanded with the lowest g cost, we've found the best route to it (no need to revisit it later).

- One heuristic **dominates** another if it is always at least as informative (higher values) on every state without violating admissibility. If h_1 and h_2 are both admissible and $h_2(n) \geq h_1(n)$ for all n, then h_2 *dominates* h_1. Using a dominant heuristic will expand **no more nodes** (and usually fewer) than a weaker heuristic, because the search is better guided and prunes more of the state space. Intuitively, a higher heuristic value means the node is judged closer to the goal, so the algorithm will focus its effort more effectively. For example, in the 8-puzzle, the Manhattan distance heuristic dominates the simpler "misplaced tiles" heuristic, because Manhattan distance is always greater or equal to the number of misplaced tiles for any given state (except the goal) – it provides a more accurate estimate of distance to the goal.

The main informed search algorithms we cover are **Greedy Best-First Search** and **A*** (A-star), along with a space-efficient variant of A*, **Iterative Deepening A***.

3.1 Greedy Best-First Search

Greedy Best-First Search selects the next node to expand based solely on the heuristic $h(n)$ – it *greedily* chooses the state that appears to be closest to the goal (smallest h).

In other words, it uses an evaluation function $f(n) = h(n)$. This search strategy uses a priority queue ordered by h. It's like an informed version of DFS/BFS that always follows the "most promising" path first.

Algorithm for Greedy Search

1. Start by inserting the initial state into a priority queue (fringe) with priority h(initial).
2. Pop the state with the lowest h value.

3. If it's a goal, return the solution; otherwise, expand it and insert all its successors with their heuristic values.

4. Repeat until a goal is found or the frontier is empty.

Greedy search can be very efficient in practice, often quickly zeroing in on a goal state. For example, in a route-finding problem, greedy best-first might always move in the direction of the destination (minimizing straight-line distance at each step). However, **greedy search sacrifices completeness and optimality** for the sake of speed:

- It is **not complete** – it can get stuck in loops if it revisits states (it should use a closed set of visited states to avoid this). Even with cycle checking, in an infinite state space, greedy search may fail, and in a finite state space without revisiting, it will eventually find a goal if one exists, but there is no guarantee in more complex spaces (especially if heuristic plateaus or local minima exist).
- It is **not optimal** – greedy search doesn't account for path cost at all. It might take a path that *looks* good (closer to goal) but is actually much longer in cost than an alternative. It finds *a* goal quickly, but not necessarily the cheapest goal.

Complexity Analysis: In the worst case, greedy best-first search can still explore a large portion of the space. Its time complexity is difficult to pin down; in the worst case, it could degenerate to $O(b^m)$ (like DFS) if it chases many misleading paths.

Its space complexity is $O(b^m)$ as well, since it might have to store all generated nodes in the worst case. However, with a good heuristic, greedy search often expands far fewer nodes than BFS or DFS in average cases. The performance depends entirely on the quality of h. A heuristic that is *closer* to the true cost (while remaining admissible if we want to eventually use it in A*) will make greedy search more effective.

Pitfall Example: Consider a maze where the goal is on the far right, and there is a long winding corridor that ultimately leads there. A greedy algorithm using straight-line distance might head East toward the goal, only to hit a dead-end wall because it ignored the fact that it needed to go around. Greedy search might then backtrack and try another route, etc., potentially exploring many dead ends. In contrast, an algorithm like A* (next section) that also considers the distance already traveled $g(n)$ would navigate around obstacles more intelligently.

Greedy best-first search is mainly valued for its speed in finding *a* solution. It's often used in scenarios where a quick, not necessarily optimal, solution is acceptable. It sets the stage for A* search by illustrating the importance of combining $h(n)$ with the path cost.

3.2 A* Search (A-Star)

The **A*** search algorithm is one of the most important and widely used search strategies in AI. It combines uniform-cost strategy with heuristic guidance. A* uses the evaluation function:

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the cost to reach node n from the start (path cost so far), and $h(n)$ is the heuristic estimate of the cost from n to a goal.

Thus, $f(n)$ is the estimated total cost of the path through n to the goal. A* expands the node with the smallest $f(n)$ value first (using a priority queue ordered by f). This balances between the cost already incurred and the estimated cost to go.

3.2.1 A* Algorithm (Graph-Search Version)

1. Initialize the frontier with the initial state (with $g = 0, f = h(\text{initial})$).

2. Repeatedly choose the node n from the frontier with minimum $f(n) = g(n) + h(n)$.

3. If n is a goal, return the solution (we have found a lowest-cost path to a goal).

4. Otherwise, expand n: for each successor s of n with step cost c, compute $g(s) = g(n) + c$; then set $f(s) = g(s) + h(s)$. If s was neither seen before nor in the frontier, add it. If it was seen with a higher g (cost), update its g and f to the better values (and update its position in the queue).
5. Keep a closed set of expanded nodes to avoid re-expanding states. If the heuristic is consistent, this guarantees optimality without reopening closed nodes.
6. Continue until a goal is chosen for expansion (which will be optimal).

3.2.2 Admissibility and Optimality

A* is designed to find optimal solutions given certain conditions on the heuristic:

- **Tree-search A*** (no closed list, might re-expand states) is **guaranteed optimal** if $h(n)$ is **admissible**.

  Intuitively, an admissible heuristic never overestimates, so A* will never ignore a potentially cheaper path.
- **Graph-search A*** (with a closed list to avoid repeats) is **optimal** if $h(n)$ is **consistent (monotonic)**. Consistency ensures that the first time A* reaches a state with the lowest g cost, that is the optimal way to get there.

Under these conditions, A* is not only optimal but also **optimally efficient** in terms of node expansions. This means no other algorithm using the same heuristic information will expand fewer nodes than A* (within a constant factor) to guarantee finding an optimal solution.

3.2.3 Completeness

A* is **complete** if the branching factor is finite and step costs are bounded above zero. With a finite number of nodes of cost $\leq$ any bound, A* will eventually expand all nodes up to the optimal solution.

3.2.4 Complexity

The worst-case time and space complexity of A* are still **exponential** in the search depth (or more precisely, in the difference between the heuristic and the actual cost). In the worst case where the heuristic provides no help (say $h(n) = 0$ for all nodes), A* degenerates to uniform-cost search, which can be exponential. If the heuristic is perfect (i.e., $h(n)$ equals the true remaining cost), A* would expand only the nodes along the optimal path – in this best case, it's linear in the solution length.

Despite exponential worst-case behavior, A* is extremely powerful with good heuristics. In practice, it's used in pathfinding, scheduling, game AI, and more, often solving problems that are otherwise infeasible. The key is to design or obtain a strong heuristic to prune the search space.

3.3 Iterative Deepening A* (IDA*)

Iterative Deepening A* (IDA*) is a variant of A* that uses iterative deepening to reduce memory usage. It performs a series of depth-first searches, each with a cutoff on the $f(n) = g(n) + h(n)$ value rather than depth.

Initially, the cutoff is set as $f(\text{start}) = h(\text{start})$. In each iteration, it expands nodes using DFS but prunes any branch when $f(n)$ exceeds the current threshold. If the goal is not found, it increases the threshold to the smallest f value that exceeded the cutoff in the previous iteration, and repeats.

IDA* retains the **optimality and completeness** of A* (with admissible, consistent heuristics it will find an optimal solution), but uses much less memory – only $O(bd)$ like DFS, since it doesn't store all nodes in a priority queue, just the current recursive stack.

3.3.1 Complexity Analysis

- **Space complexity:** Polynomial, $O(bd)$ where d is the solution depth. This is linear in depth for unit step cost problems.
- **Time complexity:** IDA* often ends up expanding some states multiple times (like iterative deepening search does), but asymptotically it's still exponential in the search depth (same order as A*).

3.3.2 Practical Application

In practice, IDA* is very effective for problems like the 15-puzzle, where A* would require too much memory. It expands more nodes than A* (due to re-expansion), but can solve much larger problems by avoiding the memory blow-up. For example, the optimal solution to the 15-puzzle can be found by IDA* with a Manhattan distance heuristic, where A* would run out of memory. IDA* is complete and optimal given a consistent heuristic, like A*, and usually only modestly increases the node count expanded relative to A* (often within a small factor).

3.3.3 Summary

A* search is the algorithm of choice for many optimal path-finding problems when a good heuristic is available. It guarantees optimality and generally expands far fewer nodes than uninformed methods. Variants like **IDA*** help mitigate its memory usage at the cost of some extra expansions. Good heuristics are critical for A*'s performance, and designing admissible, consistent heuristics is a major part of applying A* effectively.

4 Mathematical Analysis of Search Algorithms

In this section, we compare the algorithms' complexities and formal properties and outline key proofs (such as A* optimality). We also discuss properties of heuristics (admissibility, consistency, dominance) in more depth.

4.1 Complexity and Strategy Comparison

Each search strategy has characteristic **time complexity** (nodes expanded) and **space complexity** (memory usage), as well as guarantees of **completeness** (will it always find a solution if one exists?) and **optimality** (does it find the best solution?).

- **Breadth-First Search (BFS)**:
  - Time: $O(b^d)$, Space: $O(b^d)$ for solution depth d.
  - **Complete** (if finite b).
  - **Optimal** if all step costs are equal (finds shortest path in steps).
- **Depth-First Search (DFS)**:
  - Time: $O(b^m)$ for $m =$ max depth (worst-case could be infinite).
  - Space: $O(b \cdot m)$ (linear) .
  - **Not complete** in infinite spaces (graph-DFS is complete in finite spaces).
  - **Not optimal**.
- **Uniform-Cost Search (UCS)**:
  - Time: $O(b^{1+C*/\varepsilon})$ (exponential in effective depth, where $C*$ is the optimal cost and ε is the minimum step cost).
  - Space: $O(b^{1+C*/\varepsilon})$ similarly.
  - **Complete** (if step costs > 0).
  - **Optimal** (finds least cost path).
- **Depth-Limited DFS (DLS)**:
  - Time: $O(b^\ell)$ for depth limit ℓ.
  - Space: $O(b \cdot \ell)$.
  - **Incomplete** if $\ell < d$ (goal beyond limit).
  - **Non-optimal** if $\ell > d$ (may find non-shallow goal first).
- **Iterative Deepening DFS (IDS)**:

  - Time: $O(b^d)$ (with small overhead).
  - Space: $O(d)$.
  - **Complete** (for finite b).
  - **Optimal** for uniform step costs (finds shallowest goal).

- **Greedy Best-First Search**:
  - Time: worst-case $O(b^m)$ (can be exponential), but typically less with a good heuristic.
  - Space: $O(b^m)$ storing nodes.
  - **Not guaranteed complete or optimal**.

- **A* (with admissible h)**:
  - Time: worst-case exponential in search depth (expands all nodes with $f \leq f^*$, where f^* is the f-value of the optimal solution).
  - Space: exponential (must remember explored and frontier states).
  - **Complete** (with finite branching & positive costs).
  - **Optimal** (with admissible h for tree-search; consistent h for graph-search).

- **Iterative Deepening A* (IDA*)**:
  - Time: $O(b^d)$ asymptotically (like A* exponential), with some overhead due to re-searching.
  - Space: $O(bd) \approx O(d)$ (linear).
  - **Complete and optimal** (if h is admissible).

4.2 Trade-offs and Heuristic Influence

These results highlight trade-offs:

- BFS and IDS guarantee optimality (for unit costs) but may suffer from memory/time blow-up.
- DFS is memory-efficient but risks missing solutions or finding suboptimal ones.
- UCS and A* guarantee optimality and often explore far fewer nodes than BFS by using costs or heuristics, but they can still suffer from exponential complexity and high memory usage.
- Greedy Best-First Search is very fast but does not guarantee completeness or optimality.

In practice, the **effective branching factor** (how the tree expands given the algorithm's order of expansion and pruning) determines actual performance. A good heuristic can bring the effective branching factor close to 1 (ideal), whereas blind search often has an effective branching factor equal to the actual branching factor of the problem.

5 Admissibility and Optimality of A*: Proof Sketch

The Optimality of A* (with an admissible heuristic) is a critical theoretical result. We gave an intuition in the A* section; here is a more formal outline:

5.1 Claim

A* tree-search with an admissible heuristic $h(n)$ will always find an optimal goal (least-cost solution) and never select a suboptimal goal for expansion.

5.2 Proof Sketch

Assume the opposite – that A* has selected a goal node G_{subopt} for expansion, but this goal's cost $g(G_{\text{subopt}}) = C_{\text{subopt}}$ is higher than the optimal solution cost C^*. Let G_{opt} be an optimal goal (cost C^*).

Since A* selected G_{subopt} for expansion, it means that for all nodes n in the frontier,

$$f(n) \geq f(G_{\text{subopt}}) = C_{\text{subopt}}.$$

Consider the initial segment of an optimal path to G_{opt}. Let n be the first node on that optimal path that was not expanded by A* before G_{subopt} (possibly the start state if it never expanded anything on the optimal path). The cost to reach n is $g(n)$, and the remaining optimal cost from n to goal is $h^*(n)$.

Since h is admissible, we have:

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

Thus, the estimated cost function satisfies:

$$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^*.$$

Since we assumed $C^* < C_{\text{subopt}}$, we obtain:

$$f(n) \leq C^* < C_{\text{subopt}}.$$

But A* picks nodes in increasing order of f. This implies A* should have selected n for expansion *before* G_{subopt} (since $f(n)$ was smaller). This is a contradiction – we assumed G_{subopt} was expanded first while n was still waiting.

Hence, our assumption was wrong: A* cannot expand a suboptimal goal before all optimal paths are exhausted.

5.3 Graph-Search A*

For **graph-search A***, the proof is similar but needs to account for the possibility of revisiting states. With a **consistent heuristic**, one can show that each state is expanded at most once with the lowest cost g (any subsequent discovery of a cheaper path would violate consistency, or if allowed, A* would reopen the node).

Therefore, the first time a goal is expanded is optimal. If the heuristic is admissible but not consistent, A* can still be optimal if it reopens nodes – the proof requires showing that even with re-expansions, it won't close off an optimal path.

However, most analyses assume consistency for simplicity, in which case closed set A* is straightforwardly optimal.

6 Heuristic Function Properties

We have defined **admissibility** and **consistency** of heuristics. These properties ensure the optimality of A*, but they also affect efficiency:

- **Admissibility:** An admissible heuristic guarantees no optimal solution is overlooked; it is a safe estimate. Many common heuristics (e.g., misplaced tiles, Manhattan distance for sliding puzzles, straight-line distance in maps) are admissible by design.
- **Consistency (Monotonicity):** A stronger condition that not only guarantees admissibility but also ensures that $f(n)$ values along any path never decrease.

  This implies that the first time A* expands a node, it has found the cheapest path to that node. Consistent heuristics allow A* to avoid reopening closed nodes.

  In practice, most well-designed heuristics for path-finding are consistent. For example, the Manhattan distance heuristic is consistent: moving one step in the puzzle reduces Manhattan distance by at most 1 (equal to the cost of that step).

- **Dominance:** If we have multiple admissible heuristics $h_1, h_2, \ldots$, one heuristic dominates another if it is greater or equal on all states (while remaining admissible). Using a dominant heuristic always improves A* search.

  For instance, consider two heuristics for the 8-puzzle:

  - h_1 = number of misplaced tiles.
  - h_2 = sum of Manhattan distances of each tile.

  Both are admissible; moreover, for any state, Manhattan distance is greater than or equal to the number of misplaced tiles (since if a tile is out of place, its Manhattan distance is at least 1). Thus, h_2 dominates h_1.

  An A* search using h_2 will **expand no more nodes** than it would using h_1 – and usually significantly fewer – because h_2 gives a more accurate estimate and better prioritizes the search.

  In general, if h_2 dominates h_1, then h_2 is strictly better or equal in guiding the search. There is no downside to using the more informed heuristic (aside from possible computational cost).

  All admissible heuristics trivially dominate the zero heuristic $h(n) = 0$ (which corresponds to uninformed uniform-cost search). That's why adding any admissible heuristic can only improve A* efficiency over plain UCS.

6.1 Heuristic Design Strategies

When designing heuristics, one effective approach is to find **multiple** admissible heuristics and take their maximum for each state:

$$h(n) = \max(h_1(n), h_2(n)).$$

Since each h_i is admissible, their maximum is also admissible, and it dominates each individually.

Another strategy is to use **relaxed problems**: a heuristic can be the cost of an optimal solution to a simplified version of the problem (which remains admissible because solving a relaxed problem is at most as hard as solving the original).

For example, Manhattan distance assumes tiles can move freely without being blocked, which is a relaxed version of the 8-puzzle.

6.2 Summary

A good heuristic (**admissible, consistent, and as close as possible to the true cost**) makes A* search highly efficient by drastically reducing the number of nodes expanded. The quality of a heuristic is often measured by how close its evaluations are to the actual cost (without overestimating).

The concept of dominance helps us choose the best among candidate heuristics: always prefer the one that gives higher estimates (while remaining admissible). As long as the heuristic is consistent, A* remains optimal and efficient.

7 Examples

To solidify these concepts, we will go through a few examples of search problems and see how different algorithms operate on them, and how heuristics affect performance.

7.1 Example: Solving the 8-Puzzle with BFS, UCS, and A*

The **8-puzzle** is a classic test for search algorithms. We have a 3×3 board with tiles numbered 1 through 8 and one blank space. The goal is to reach a specific arrangement (often the tiles in numerical order) by sliding tiles into the blank. Let's consider a concrete instance:

Solving an Eight Puzzle — CS160 Reader: An example 8-puzzle instance with a **Starting State** (left) and a **Goal State** (right) (Source).

In the **starting state**, the tiles are arranged as:

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & \_ & 5 \end{bmatrix}$$

The **goal state** in this example is:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & \_ & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

(Note: Another common goal arrangement is 1 2 3 / 4 5 6 / 7 8 _, but the specific goal does not affect the general search concepts.)

7.2 Breadth-First Search (BFS)

BFS explores moves in increasing order of depth. From the start, it generates all states reachable in 1 move, then all in 2 moves, and so on, until it finds the goal. BFS guarantees the **shortest solution in terms of number of moves**.

For this puzzle, the shortest solution might be, say, 5 moves. BFS will systematically try:

- All 1-move possibilities,
- Then all 2-move sequences,
- Then 3-move sequences, and so on.

This ensures an optimal number of moves but causes the number of explored states to grow rapidly.

7.2.1 State Space Complexity

The 8-puzzle has an average branching factor of about 2.7 (not every move is always possible due to the blank being in a corner, edge, or middle). The total number of reachable states is:

$$\frac{9!}{2} = 181,440.$$

In the worst case, BFS may expand thousands of nodes before reaching the goal. Although it guarantees finding the optimal solution, it explores many unnecessary states at the same depth as the solution.

7.2.2 Memory Considerations

BFS is also memory-intensive because it stores a large frontier of states. If the solution depth is 5, BFS could store all states up to depth 4 or 5, which could be on the order of:

$$b^5 \approx 2.7^5 \approx 143 \text{ states.}$$

This is manageable for small problems, but if the depth were larger, memory usage grows exponentially.

7.2.3 Practical Implications

BFS is feasible for the 8-puzzle (which has relatively shallow solutions, often around 20 moves at most), but not for larger puzzles like the 15-puzzle, which can require 50-80 moves, making BFS intractable.

7.3 Uniform-Cost Search (UCS)

Uniform-Cost Search (UCS) expands nodes in order of increasing path cost. In the case of the 8-puzzle, if each tile move has an equal cost of 1 (a common assumption), UCS behaves similarly to BFS.

7.3.1 Path Cost and Expansion Order

Since all moves have the same cost:

- The least-cost solution is also the shortest solution in terms of moves.
- UCS expands nodes in increasing path cost order, which, in this case, means expanding by depth.
- Like BFS, UCS guarantees an optimal solution (minimum number of moves).

7.3.2 When UCS Differs from BFS

The benefit of UCS becomes evident when different actions have different costs. Although this is unusual for the 8-puzzle, hypothetically:

- If moving a tile *up* had a higher cost than moving *down*, UCS would consider this and find the cheapest-cost solution, not just the shortest.
- This makes UCS more flexible than BFS when dealing with variable-cost pathfinding problems.

7.3.3 UCS and BFS Equivalence in the 8-Puzzle

In an equal-cost setting (like the 8-puzzle), UCS behaves exactly like BFS:

- It expands the same states in the same order.
- It incurs the same exponential blowup in state space.
- There is no performance difference between UCS and BFS.

7.3.4 Why Discuss UCS for the 8-Puzzle?

Even though UCS and BFS are identical in uniform-cost scenarios, UCS is often mentioned in the context of the 8-puzzle to highlight that BFS is a special case of UCS when all step costs are equal.

7.4 A* Search

A* search expands nodes based on the evaluation function:

$$f(n) = g(n) + h(n),$$

where:

- $g(n)$ is the cost to reach node n.
- $h(n)$ is a heuristic estimate of the cost from n to the goal.

For the 8-puzzle, two common admissible heuristics are:

- h_1 = number of misplaced tiles (tiles not in their goal position).
- h_2 = Manhattan distance (sum of distances each tile is from its goal position, measured in grid moves).

7.4.1 Example Using Manhattan Distance

Let's use h_2 (Manhattan distance) as it is more informative. For the given starting state, we estimate $h_2(\text{start}) = 6$. A* prioritizes states with the lowest $f = g + h$. Initially:

$$f(\text{start}) = 0 + h(\text{start}) = 6.$$

A* expands the start node, generates its neighbors, and picks the neighbor with the smallest f. Typically:

- Each move increases g by 1.
- h slightly changes based on the new tile arrangement.

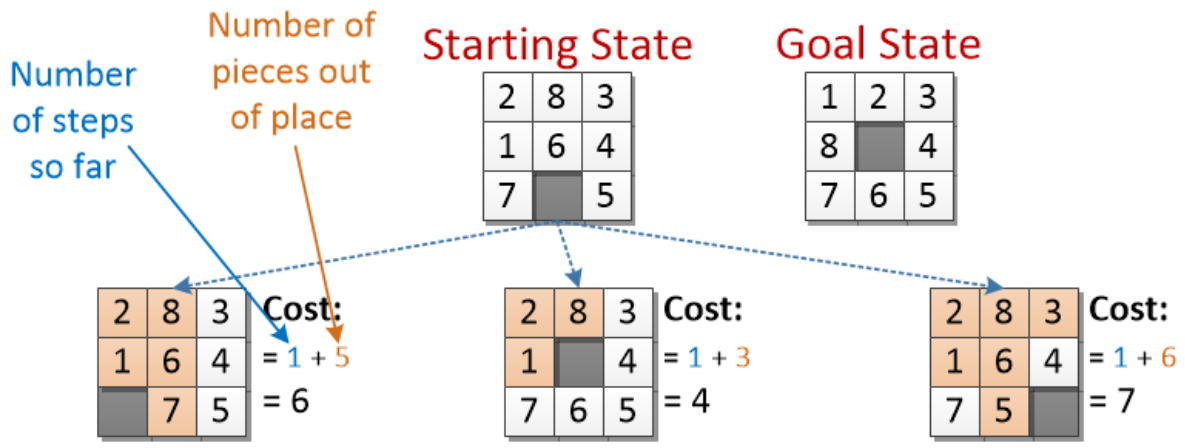

Figure 4: 8-puzzle problem

- A* selects the state with the lowest f value.

Solving an Eight Puzzle — CS160 Reader: A* search (Fig. 4 and fig.5) applied to the 8-puzzle prioritizes the most promising states by computing $f = g + h$. (Source).

7.4.2 Comparison to BFS and UCS

- BFS expands all nodes at a given depth, even if a solution is already found.
- UCS behaves like BFS when all moves have equal cost.
- A* prunes unnecessary states by focusing on the most promising paths.

For example, BFS may explore **all** states up to depth 5 even if the optimal solution is found at depth 5, whereas A* avoids large portions of the search space.

7.4.3 Efficiency of A*

- BFS might expand thousands of nodes for a 20-move solution.
- A* with the Manhattan heuristic might expand only a few hundred nodes.
- If the initial state is 20 moves away from the goal:
  - BFS may explore millions of states.
  - A* using h_2 (Manhattan) may explore only tens of thousands.

7.4.4 Impact of Heuristic Choice

A* will continue expanding nodes with the lowest f until it finds the optimal solution.

- Using h_1 (misplaced tiles) still ensures optimality but expands more nodes than h_2.
- h_1 treats two states as equally good if they have the same number of misplaced tiles, even if one is closer to the goal.
- h_2 differentiates better by considering how far tiles are from their goal.

Conclusion: A* with a better heuristic (one that dominates another) leads to significantly fewer expansions, making the search much more efficient.

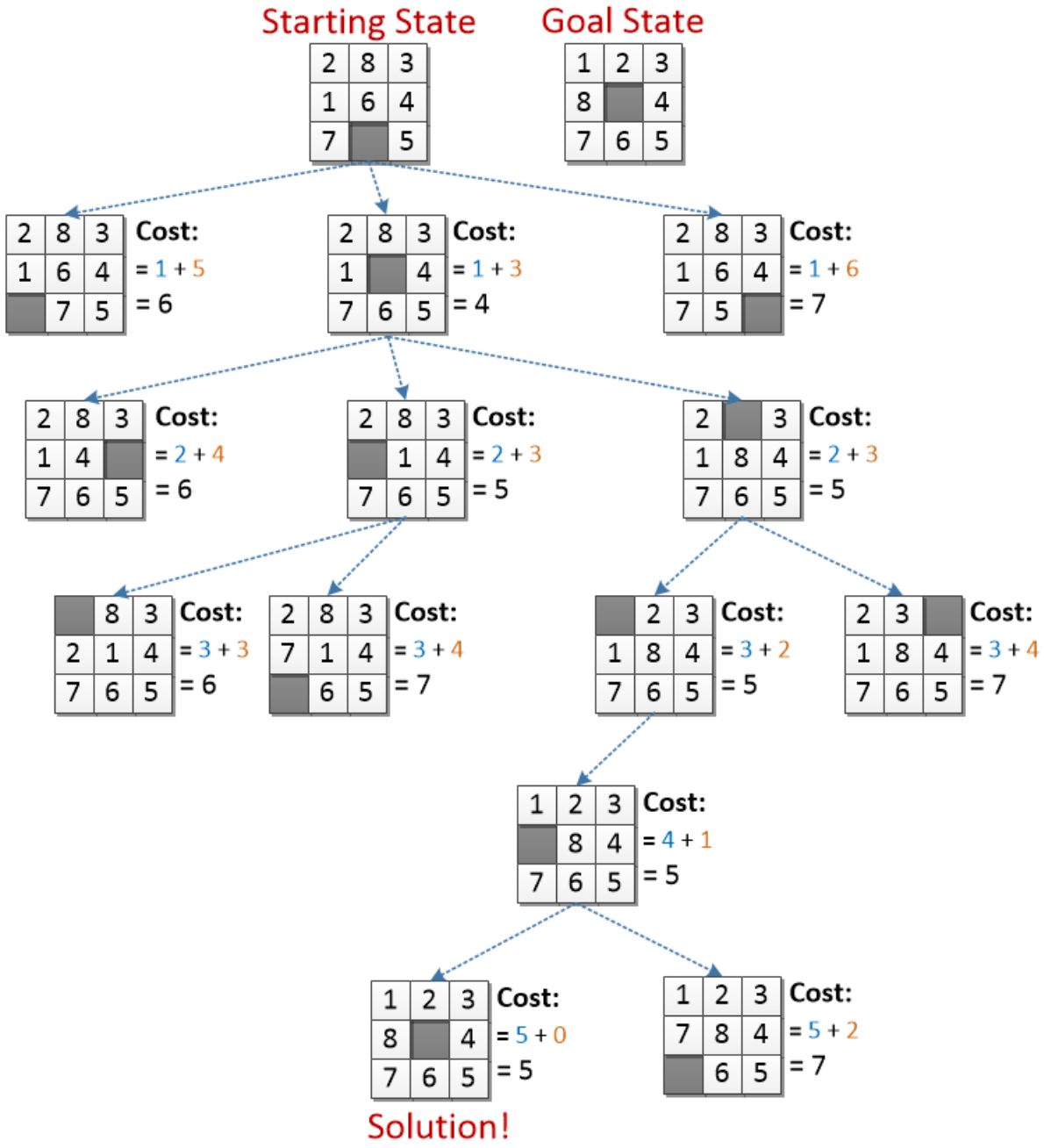

Figure 5: 8-puzzle problem full solution A-star search.

7.5 Summary for 8-Puzzle

Breadth-First Search (BFS) and Uniform-Cost Search (UCS) can certainly solve the 8-puzzle but tend to explore a vast number of states blindly.

- **BFS/UCS:**
  - Guarantee finding the optimal solution (minimum number of moves).
  - Expand states systematically but suffer from exponential growth.
  - Store large frontiers, leading to high memory consumption.
- **A* with Manhattan Distance:**
  - **Drastically reduces** the search effort.
  - Both **optimal and efficient**, finding the minimum-move solution without expanding unnecessary states.
  - Expands significantly fewer nodes than BFS/UCS due to better guidance from heuristics.
- **Larger Puzzles (15-Puzzle, 24-Puzzle):**
  - A* with strong heuristics (e.g., Manhattan Distance, pattern databases) is the **only feasible way** to solve optimally.
  - Uninformed methods such as BFS and UCS become impractical due to their large search space.

Conclusion: A* with an admissible heuristic like Manhattan Distance is the best approach for solving the 8-puzzle efficiently. For larger puzzles, heuristic-driven A* remains essential, as uninformed methods become computationally infeasible.

7.6 Example: Pathfinding with A* vs. Greedy

Consider a **pathfinding problem on a grid**. A robot must move from a start cell to a goal cell in a maze with obstacles (Fig. 6). Each move has a cost of 1. We use the **Manhattan distance** as a heuristic:

$$h(n) = |x_n - x_{\text{goal}}| + |y_n - y_{\text{goal}}|.$$

This heuristic is **admissible** (and **consistent**) because one cannot reach the goal in fewer moves than the Manhattan distance (which assumes no obstacles).

7.6.1 A* Search on the Grid

A* search finds the shortest path to the goal by expanding nodes in a **directed wave** toward the goal:

- If no obstacles exist, A* essentially follows a straight line since it minimizes $f = g + h$.
- With obstacles, A* navigates around them efficiently.
- The heuristic prevents unnecessary exploration: if the goal is eastward, A* primarily explores eastward moves, occasionally adjusting north/south to bypass walls.
- Unlike uniform-cost search (which expands outward in all directions like Dijkstra's algorithm), A* **prunes** the search space by focusing on promising directions.

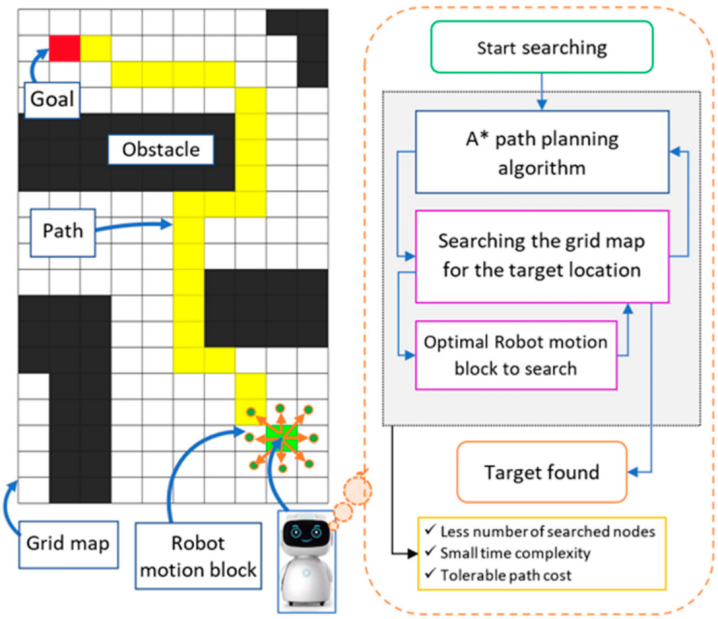

Figure 6: Robot must move from a start cell to a goal cell in a maze with obstacles (reference)

7.6.2 Greedy Best-First Search

Greedy best-first search uses the same heuristic $h(n)$ but ignores $g(n)$, meaning it:

- Always moves toward the goal as fast as possible.
- Can struggle with obstacles. If a large wall blocks the direct path, greedy search moves straight toward the goal until it collides.
- May follow the obstacle edge inefficiently or even get stuck in loops.
- Unlike A*, it does not account for the actual path cost, leading to unnecessary detours.

7.6.3 Comparison of A* vs. Greedy

- A* chooses a slightly longer path if it ultimately leads to a better position.
- Greedy may choose a poor path, increasing total distance despite short-term gains.
- In structured mazes, A* is significantly more reliable for optimal pathfinding.

Conclusion: A* balances heuristic guidance and path cost tracking, ensuring optimal navigation. Greedy search, while fast, often leads to inefficient routes or failures when obstacles are present.

7.7 Pathfinding Example: A* vs. Greedy Search

Scenario: Imagine a robot starting at the bottom-left of a grid and aiming to reach the top-right. A diagonal wall obstructs the direct path, requiring a detour. The **Manhattan distance heuristic**:

$$h(n) = |x_n - x_{\text{goal}}| + |y_n - y_{\text{goal}}|$$

is very informative but does not account for obstacles.

7.7.1 A* Search Behavior

- A* systematically explores nodes in increasing $f = g + h$ order.
- Since the wall forces a detour, A* finds the **optimal** way around.
- A* expands all nodes where $f < C^*$, where C^* is the actual shortest path length.
- It avoids unnecessary expansions by balancing heuristic guidance and path cost tracking.

7.7.2 Greedy Best-First Search Behavior

- Greedy search moves toward the goal using only $h(n)$, ignoring path cost.
- If an obstacle blocks the direct path, greedy search may inefficiently attempt to cut through before being forced to go around.
- The final path may be longer than the optimal solution.
- Greedy might expand fewer nodes in simple cases but can expand many more in difficult cases by pursuing inefficient routes.

7.7.3 Real-World Example: Romania Road Map

In AI textbooks, the **Romania road map** illustrates pathfinding with A* (Fig. 7). Using **straight-line distance** to the destination as the heuristic, A* efficiently finds the shortest route from **Arad to Bucharest**:

- A* might expand: *Arad* → *Sibiu* → *Rimnicu Vilcea* → *Pitesti* → *Bucharest*.
- It prunes less promising paths early because they have a higher f.

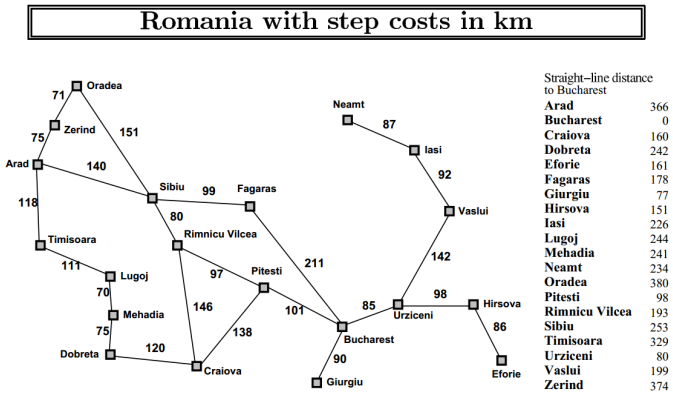

Figure 7: Real-World Example: Romania Road Map

A greedy algorithm might also choose *Sibiu* initially, but if a misleading road appears slightly shorter but is actually much longer, greedy might follow it. A* would consider path cost and likely avoid it in favor of a more efficient route.

7.8 Heuristic Impact

To quantify the impact of heuristics:

- Without a heuristic ($h = 0$), A* behaves like UCS and expands a vast search space.
- With a reasonable heuristic (e.g., Manhattan distance), A* restricts expansions to a corridor around the optimal path.
- A more precise heuristic (e.g., exact step count ignoring only one obstacle) prunes even more nodes.
- If a heuristic were **perfect** (equal to the true remaining cost), A* would expand only the nodes along the optimal path.

7.8.1 Admissibility and Accuracy of Heuristics

- A heuristic must remain **admissible** to guarantee optimality.
- Overestimation (e.g., rounding up Euclidean distance) can break A*'s optimality.
- Small overestimates may still lead to near-optimal solutions, but correctness is no longer assured.
- In theoretical discussions and exams, admissible heuristics are preferred to ensure correctness.

Conclusion: A* provides optimal and efficient pathfinding by considering both the heuristic and past cost. Greedy search, while fast, often takes longer routes or explores inefficient paths in complex environments. The quality of heuristics directly impacts A*'s efficiency, and ensuring admissibility is crucial for correctness..

7.9 Example: Heuristic Quality and Performance

To illustrate the impact of heuristic choice, let's compare two heuristics for solving the same problem:

- h_1: Admissible but not very informed.
- h_2: Admissible and more informed (dominates h_1).

7.9.1 8-Puzzle: Misplaced Tiles vs. Manhattan Distance

Consider the **8-puzzle**. We define:

- h_1 = number of misplaced tiles.
- h_2 = Manhattan distance (sum of distances each tile is from its goal position).

Both heuristics are admissible and consistent. Suppose the optimal solution for a given scramble takes 15 moves. Running A* with:

- h_1 may expand around **1,000 nodes**.
- h_2 may expand only **200 nodes**.

Why? Because h_2 provides a higher average h value for each state, causing $f(n) = g(n) + h(n)$ to grow more slowly as depth increases. This leads A* to focus more tightly on promising states.

7.9.2 Heuristic Dominance and Effective Branching Factor

- The misplaced-tiles heuristic significantly underestimates the true cost since it only counts incorrect tiles but ignores *how far* they are misplaced.
- The Manhattan heuristic is more informative because each tile contributes a movement cost.
- Consequently, h_2 results in a lower **effective branching factor**, making A* expand far fewer nodes.

Empirical studies show that using Manhattan distance instead of misplaced tiles can reduce node expansions by an order of magnitude.

7.9.3 Grid Pathfinding: Manhattan vs. Euclidean Distance

Consider a grid where diagonal moves are allowed. Using:

- Manhattan distance as a heuristic **underestimates** the shortest path cost since diagonal moves are shorter than the Manhattan estimate.
- Euclidean distance would be a better admissible heuristic because it better approximates movement costs.
- The **Octile distance heuristic** (which exactly matches the shortest path cost on a grid with diagonal moves) dominates Manhattan.

As a result, A* expands fewer nodes with Euclidean or Octile heuristics than with Manhattan distance in diagonal-moving grid environments.

7.9.4 Key Takeaways

- The **better (more informed) the heuristic, the less work A* does**.
- However, computing a heuristic has its own cost. In theoretical discussions, we assume heuristic computation is $O(1)$ per state.
- **Heuristic dominance** provides a clear comparison: if one heuristic is dominated, it is never preferable.
- Often, multiple heuristics are combined to create a stronger dominating heuristic.

7.9.5 Final Thoughts

These worked examples highlight why informed search methods are preferred. BFS and UCS guarantee optimality but are computationally expensive, whereas A* achieves optimality with significantly fewer node expansions given a good heuristic. The 8-puzzle example demonstrates **A*'s optimality** and the effect of heuristic accuracy on search efficiency.