Heuristic Search

CSI 4106 Introduction to Artificial Intelligence

Marcel Turcotte
Version: Nov 7, 2025 09:04

Preamble

Message of the Day

7 U of T community members named in Observer A.I. Power Index, 2025-10-08.

Learning Objectives

- **Comprehend** informed search strategies and heuristic functions' role in search efficiency.
- **Implement** and compare BFS, DFS, and Best-First Search using the 8-Puzzle problem.
- **Analyze** performance and optimality of various search algorithms.

Summary

Search Problem Definition

- A collection of **states**, referred to as the **state space**.
- An **initial state** where the agent begins.
- One or more **goal states** that define successful outcomes.
- A set of **actions** available in a given state s.
- A **transition model** that determines the next state based on the current state and selected action.
- An **action cost function** that specifies the cost of performing action a in state s to reach state s'.

Definitions

- A **path** is defined as a sequence of actions.
- A **solution** is a path that connects the initial state to the goal state.
- An **optimal solution** is the path with the lowest cost among all possible solutions.

We assume that the **path cost** is the sum of the individual action costs, and all costs are positive. The state space can be conceptualized as a **graph**, where the nodes represent the states and the edges correspond to the actions.

In certain problems, multiple optimal solutions may exist. However, it is typically sufficient to identify and report a single optimal solution. Providing all optimal solutions can significantly increase time and space complexity for some problems.

Example: 8-Puzzle

In [2]:
```
import random
import matplotlib.pyplot as plt
import numpy as np

random.seed(58)

def is_solvable(tiles):
    # Compter les inversions dans la liste à plat des tuiles (en excluant l'
    inversions = 0
    for i in range(len(tiles)):
        for j in range(i + 1, len(tiles)):
            if tiles[i] != 0 and tiles[j] != 0 and tiles[i] > tiles[j]:
```

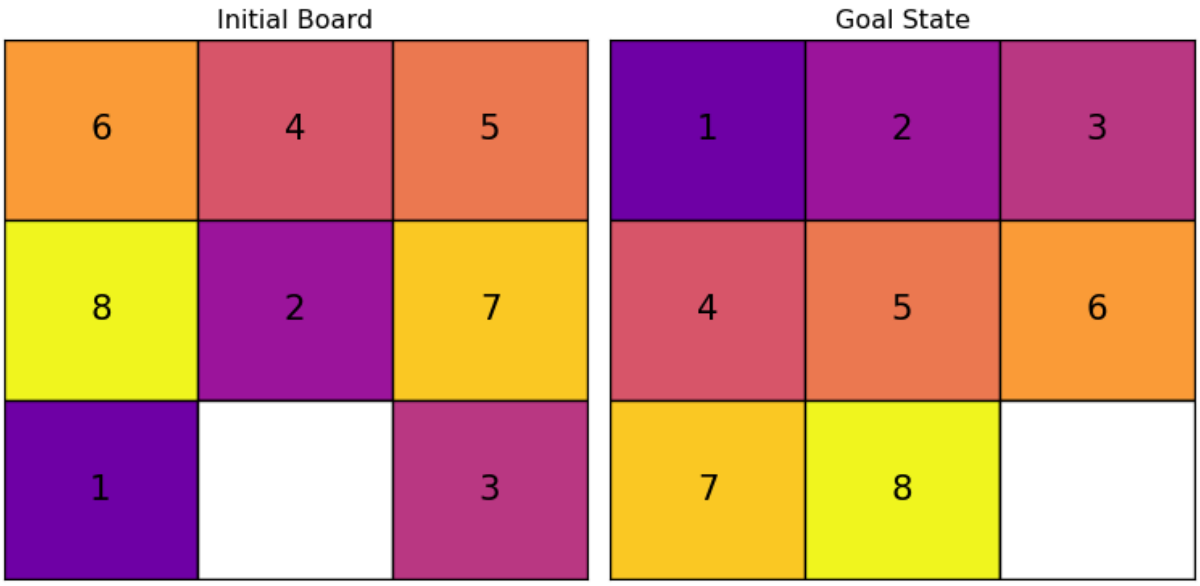
```
                inversions += 1
    return inversions % 2 == 0

def generate_solvable_board():
    # Générer une configuration de plateau aléatoire qui est garantie d'être
    tiles = list(range(9))
    random.shuffle(tiles)
    while not is_solvable(tiles):
        random.shuffle(tiles)

    return tiles

def plot_board(board, title, num_pos, position):
    ax = plt.subplot(1, num_pos, position)
    ax.set_title(title)
    ax.set_xticks([])
    ax.set_yticks([])

    board = np.array(board).reshape(3, 3).tolist() # Reconfigurer en une gri

    # Utiliser une carte de couleurs pour afficher les numéros
    cmap = plt.cm.plasma
    norm = plt.Normalize(vmin=-1, vmax=8)

    for i in range(3):
        for j in range(3):
            tile_value = board[i][j]
            color = cmap(norm(tile_value))
            ax.add_patch(plt.Rectangle((j, 2 - i), 1, 1, facecolor=color, ec
            if tile_value == 0:
                ax.add_patch(plt.Rectangle((j, 2 - i), 1, 1, facecolor='whit
            else:
                ax.text(j + 0.5, 2 - i + 0.5, str(tile_value),
                        fontsize=16, ha='center', va='center', color='black'

    ax.set_xlim(0, 3)
    ax.set_ylim(0, 3)
```

Example: 8-Puzzle

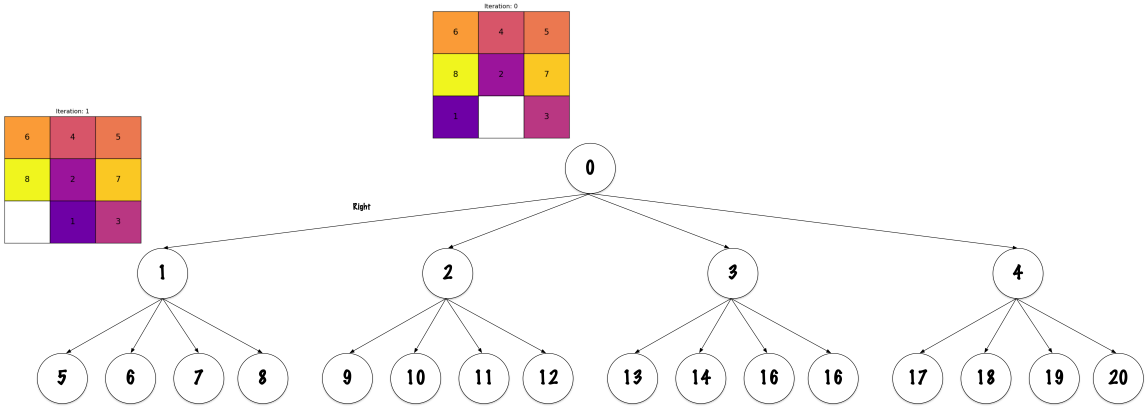
In [3]:

```
def main():
    # Generate initial solvable board
    initial_board = generate_solvable_board()

    # Define goal state
    goal_board = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]

    # Plot both boards
    plt.figure(figsize=(8, 4))
    plot_board(initial_board, "Initial Board", 2, 1)
```

```
    plot_board(goal_board, "Goal State", 2, 2)
    plt.tight_layout()
    plt.show()

main()
```



Search Tree

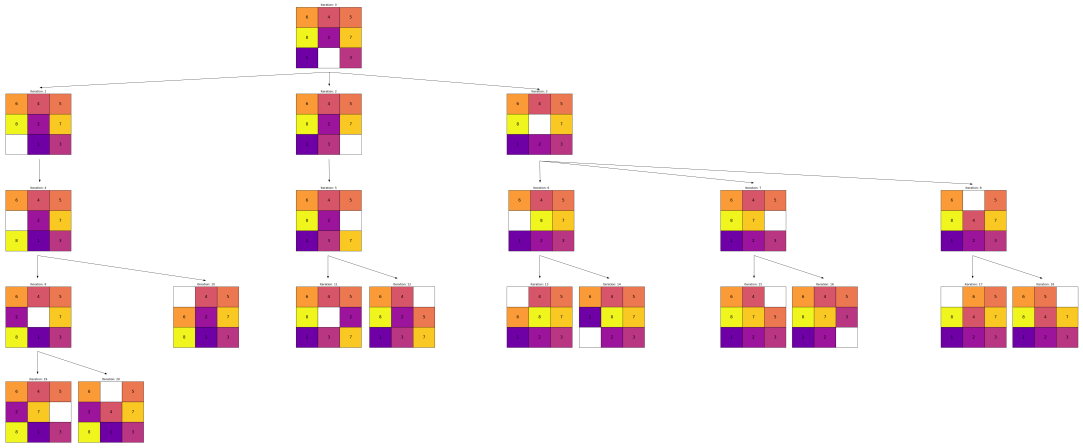


A **search tree** is a *conceptual tree structure* where **nodes** represent *states* in a **state space**, and **edges** represent possible **actions**, facilitating systematic exploration to find a **path** from an **initial state** to a **goal state**.

The search algorithms we examine today construct a search tree, where each node represents a state within the state space and each edge represents an action.

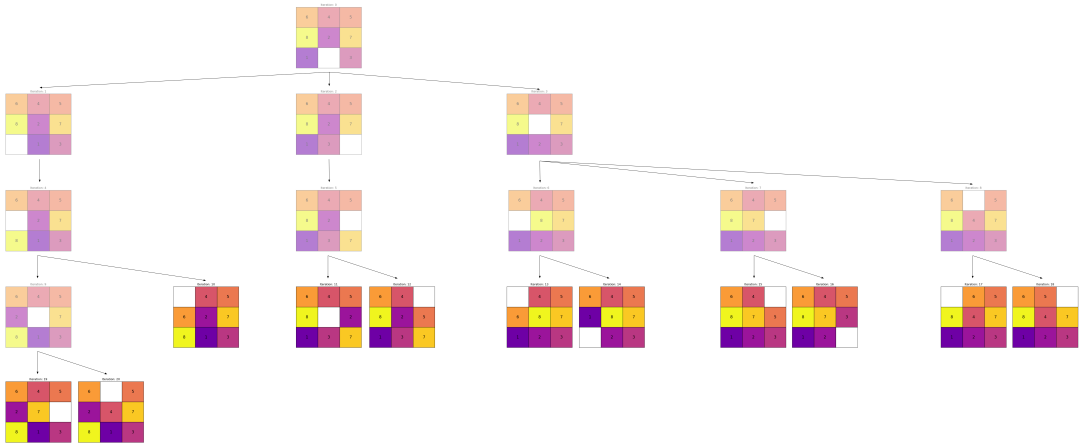
It is important to distinguish between the search tree and the state space, which can be depicted as a graph. The structure of the search tree varies depending on the algorithm employed to address the search problem.

Search Tree



An example of a **search tree** for the **8-Puzzle**. The solution here is incomplete.

Frontier



Any state corresponding to a node in the search tree is considered **reached**. **Frontier** nodes are those that have been **reached** but have **not yet been expanded**. Above, there are **10 expanded nodes** and **11 frontier nodes**, resulting in a total of 21 nodes that have been **reached**.

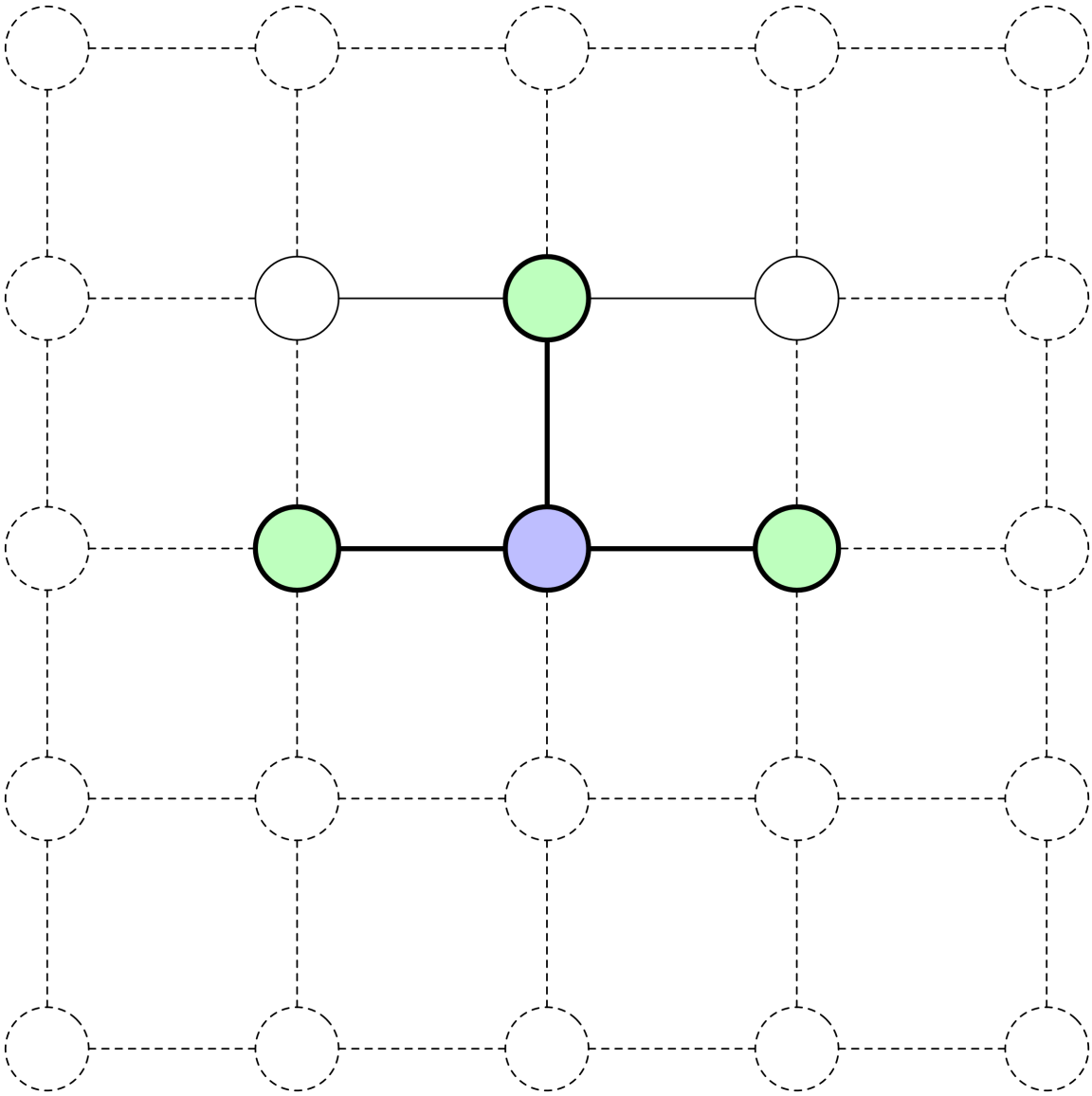
If no solution exists, the algorithm stops when the frontier becomes empty.

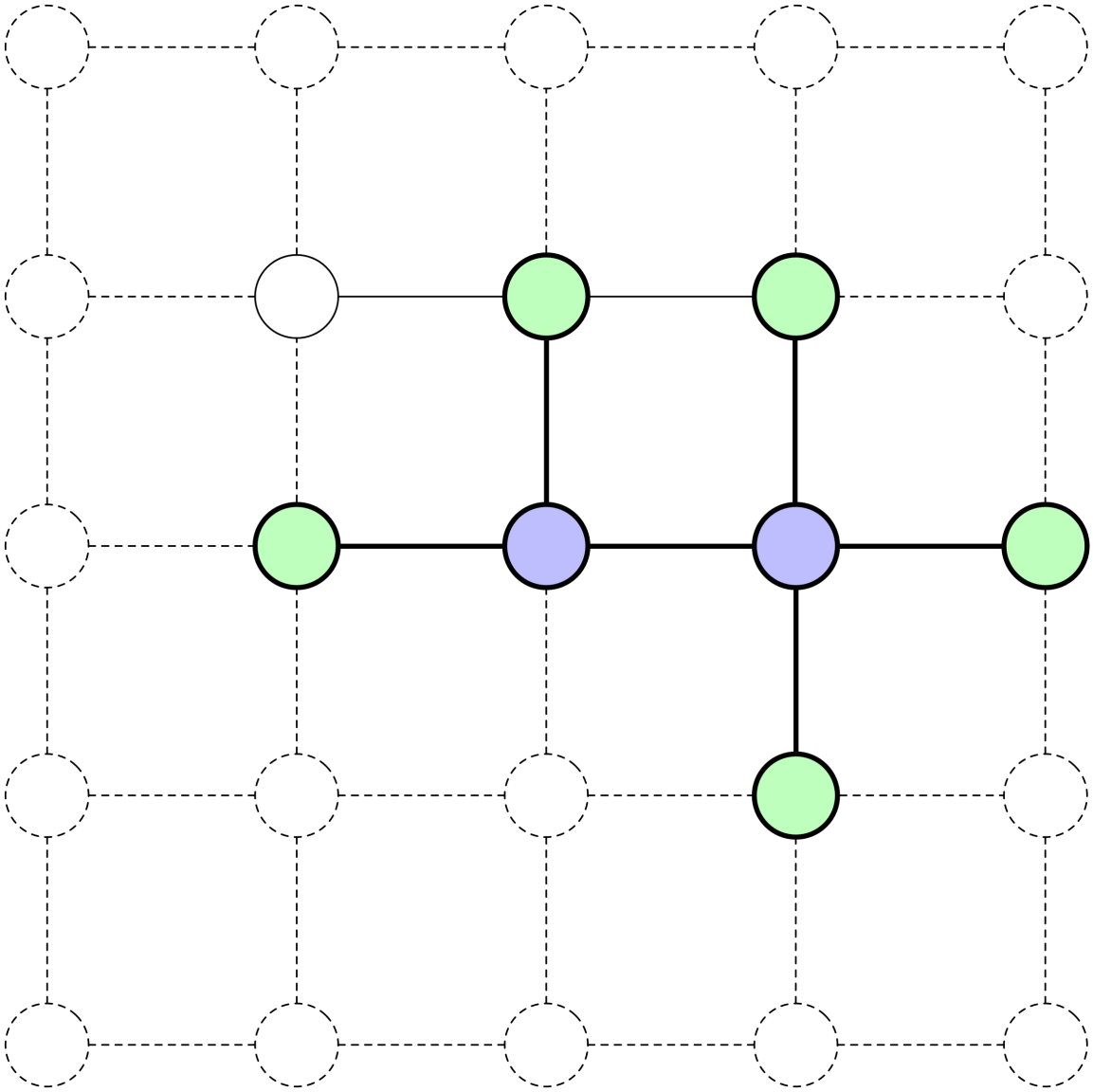
The frontier always contains at least one node that might lead to the goal.

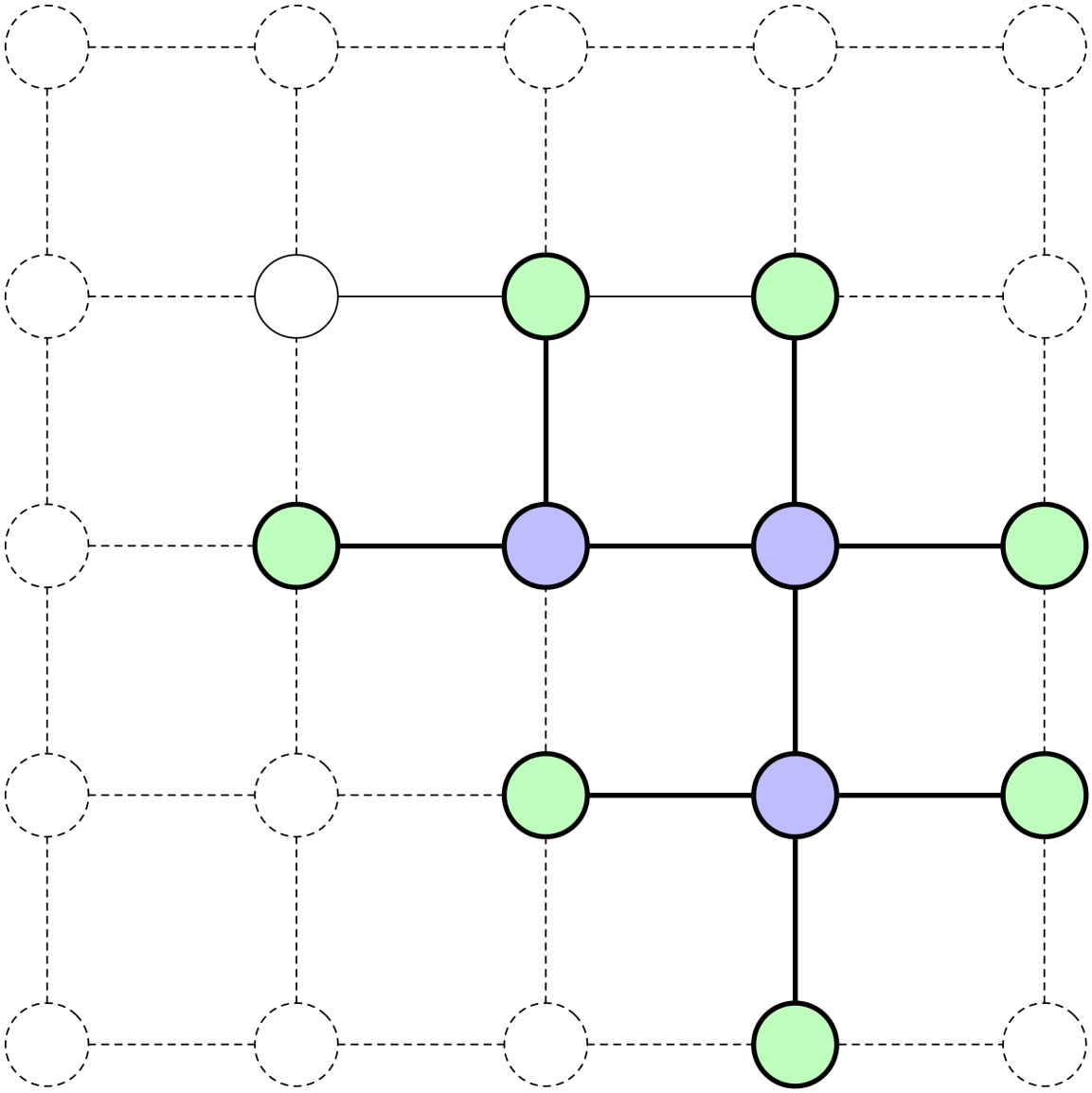
In the context of the 8-puzzle, the generated board admits a solution. From this board, we generate all possible neighbours.

Each time a node n is removed from the frontier, we generate all its neighbours. This operation applies to all nodes, including those located on an optimal path.

Frontier



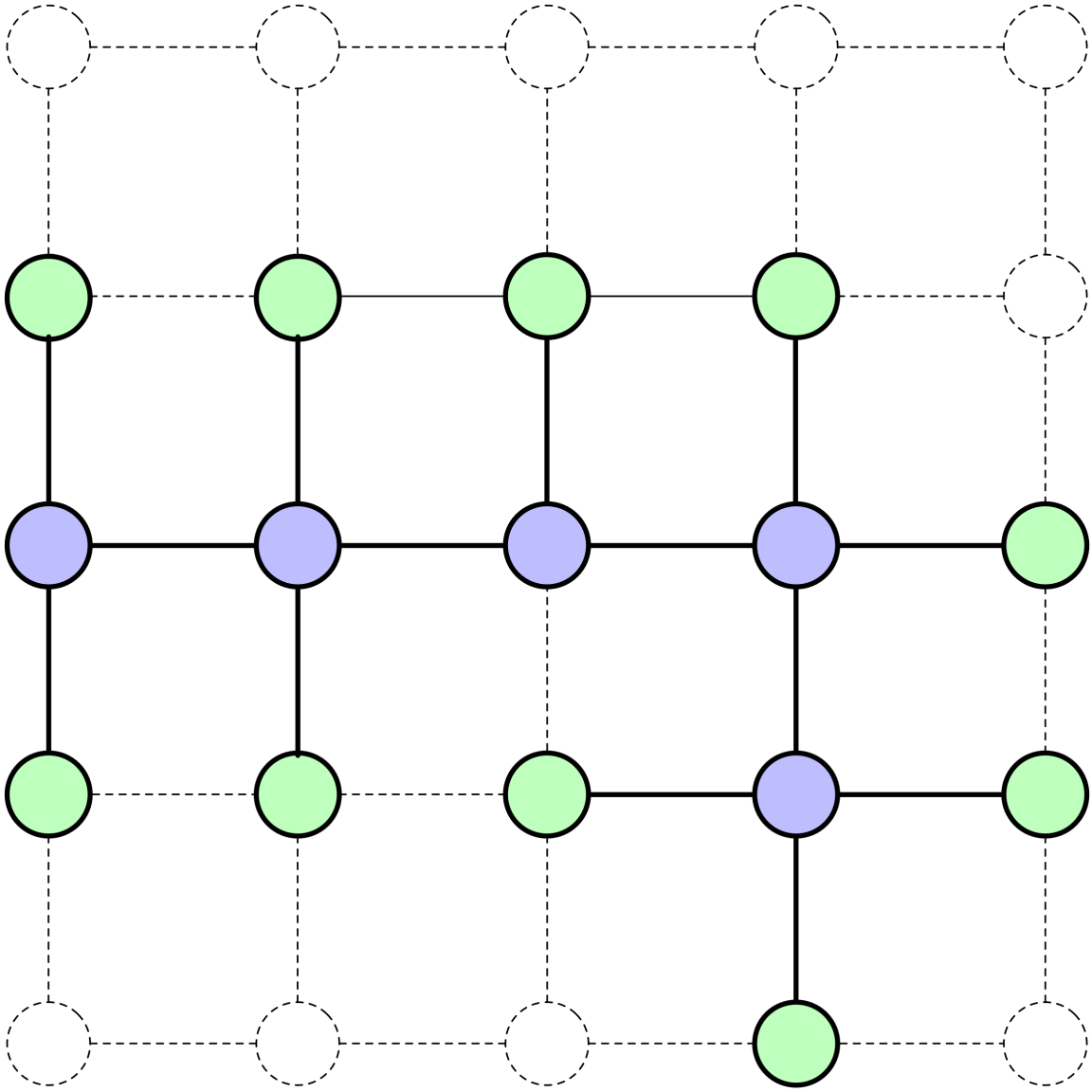


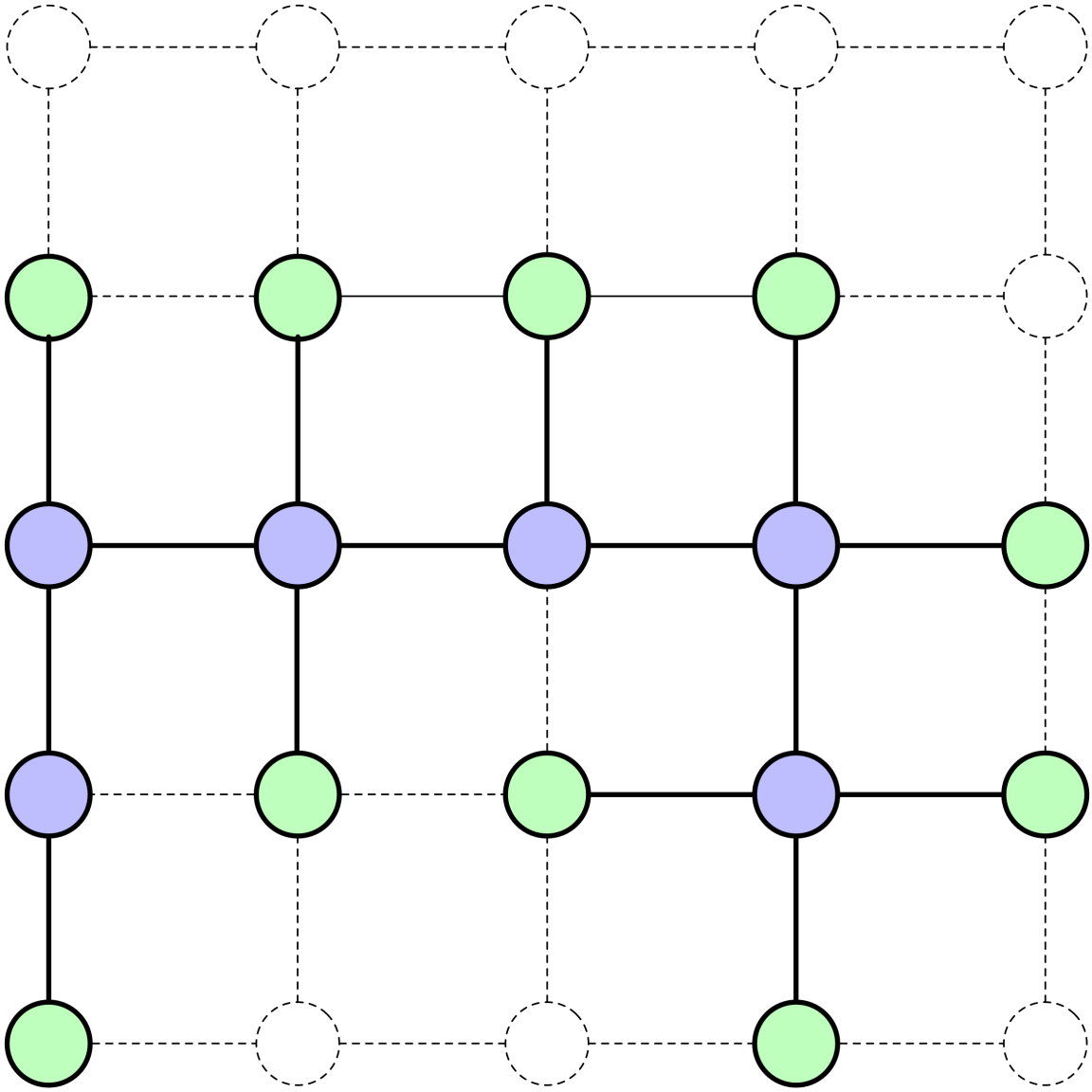


In the 8-Puzzle, four actions are possible: slide **left**, **right**, **up**, or **down**. The search can be visualized on a grid: **purple nodes**: expanded states, **green nodes**: frontier states (reached but not expanded).

The diagrams correspond to the search tree presented on the previous page. For example, the initial state can be expanded using three actions: slide left, right, and up. Node (2, 3) can only be expanded by sliding down, while node (3, 3) can be expanded by sliding left and down.

Frontier





is_empty

In [4]:
```
def is_empty(frontier):
    """Checks if the frontier is empty."""
    return len(frontier) == 0
```

is_goal

In [5]:
```
def is_goal(state, goal_state):
    """Determines if a given state matches the goal state."""
    return state == goal_state
```

Auxilliary method.

expand

In [6]:
```python
def expand(state):
    """Generates successor states by moving the blank tile in all possible d
    size = int(len(state) ** 0.5)  # Determine puzzle size (3 for 8-puzzle,
    idx = state.index(0)  # Find the index of the blank tile represented by
    x, y = idx % size, idx // size  # Convert index to (x, y) coordinates
    neighbors = []

    # Define possible moves: Left, Right, Up, Down
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        # Check if the new position is within the puzzle boundaries
        if 0 <= nx < size and 0 <= ny < size:
            n_idx = ny * size + nx
            new_state = state.copy()
            # Swap the blank tile with the adjacent tile
            new_state[idx], new_state[n_idx] = new_state[n_idx], new_state[i
            neighbors.append(new_state)
    return neighbors
```

print_solution

In [7]:
```python
def print_solution(solution):
    """Prints the sequence of steps from the initial to the goal state."""
    size = int(len(solution[0]) ** 0.5)
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for i in range(size):
            row = state[i*size:(i+1)*size]
            print(' '.join(str(n) if n != 0 else ' ' for n in row))
        print()
```

Breadth-first search

In [8]:
```python
from collections import deque
```

Breadth-first search (BFS) employs a **queue** to manage the frontier nodes, which are also known as the open list.

Breadth-first search

In [9]:
```python
def bfs(initial_state, goal_state):

    frontier = deque()  # Initialize the queue for BFS
    frontier.append((initial_state, []))  # Each element is a tuple: (state,

    explored = set()
    explored.add(tuple(initial_state))
```

```
    iterations = 0 # simply used to compare algorithms

    while not is_empty(frontier):
        current_state, path = frontier.popleft()

        if is_goal(current_state, goal_state):
            print(f"Number of iterations: {iterations}")
            return path + [current_state]  # Return the successful path

        iterations = iterations + 1

        for neighbor in expand(current_state):
            neighbor_tuple = tuple(neighbor)
            if neighbor_tuple not in explored:
                explored.add(neighbor_tuple)
                frontier.append((neighbor, path + [current_state]))

    return None  # No solution found
```

Using `tuple` makes states immutable and hashable, enabling storage in a set.

Depth-First Search

In [10]:
```
def dfs(initial_state, goal_state):

    frontier = [(initial_state, [])]  # Each element is a tuple: (state, pat

    explored = set()
    explored.add(tuple(initial_state))

    iterations = 0

    while not is_empty(frontier):
        current_state, path = frontier.pop()

        if is_goal(current_state, goal_state):
            print(f"Number of iterations: {iterations}")
            return path + [current_state]  # Return the successful path

        iterations = iterations + 1

        for neighbor in expand(current_state):
            neighbor_tuple = tuple(neighbor)
            if neighbor_tuple not in explored:
                explored.add(neighbor_tuple)
                frontier.append((neighbor, path + [current_state]))

    return None  # No solution found
```

Remarks

- **Breadth-first search (BFS)** identifies the optimal solution, 25 moves, in 145,605 iterations.
- **Depth-first search (DFS)** discovers a solution involving 1,157 moves in 1,187 iterations.

How can solutions be discovered **more efficiently**?

Will **Depth-First Search (DFS)** invariably yield sub-optimal solutions?

No, if the optimal solution lies along the path traversed by depth-first search (DFS) within the search tree, then DFS will indeed identify the optimal solution.

Is it possible for DFS to discover solutions superior to the optimal solution?

Certainly not; such solutions would either be invalid (involving impossible moves) or indicate an error in your estimation.

Does this imply that depth-first search (DFS) has no practical applications?

When is it appropriate to use DFS?

Breadth-first search (BFS) expands its frontier systematically in all directions, leading to rapid growth in memory requirements.

In contrast, the memory usage of DFS is constrained by the number of moves needed to reach its backtracking points or the path length of the first solution found. In all scenarios, DFS continues expanding the frontier in one direction.

In certain applications where all possible solutions must be explored, the entire search space must be traversed. Using BFS in these cases would be prohibitively expensive in terms of memory. However, DFS can explore the entire space with minimal memory usage.

The programming language Prolog includes a built-in backtracking algorithm that enumerates all possible solutions. Backtracking is a memory-efficient variant of DFS.

Depth-limited and iterative deepening search would be alternative uninformed search algorithms.

Finding solutions more efficiently requires domain knowledge.

Informed Search

Heuristic Search

Informed search algorithms utilize **domain-specific knowledge** regarding the goal state's location.

Heuristic Search

Let $f(n)$ be a **heuristic function** that estimates the **cost of the cheapest path** from the current state or node n to the goal.

This approach is termed **best-first search**.

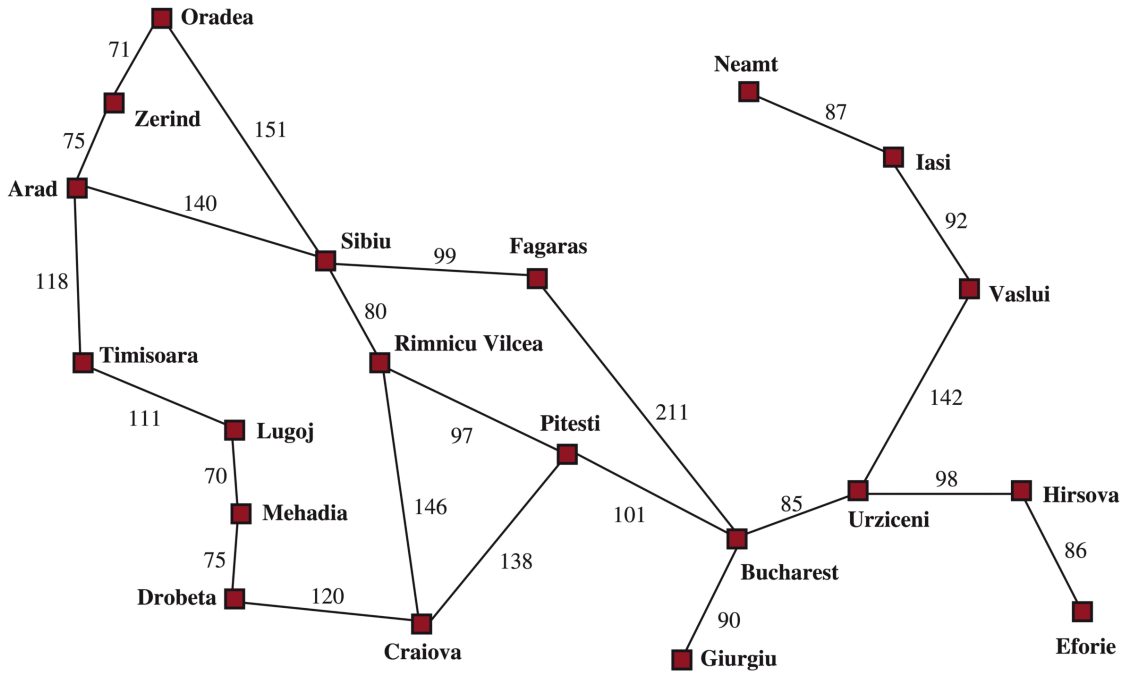
Heuristic Search

In **route-finding problems**, one might employ the straight-line distance from the current location to the destination as a heuristic.

. . .

Although **an actual path may not exist along that straight line**, the algorithm will prioritize expanding the node closest to the destination (goal) based on this straight-line measurement.

Book Example



Source: (Russell and Norvig 2020, fig. 3.1)

Problem: Determine the shortest route between Arad (initial state) and Bucharest (goal state).

Book Example

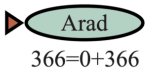
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Source: (Russell and Norvig 2020, fig. 3.16)

We have data on the direct (straight-line, as-the-crow-flies) Euclidean distances between each city and Bucharest.

Book Example

(a) The initial state

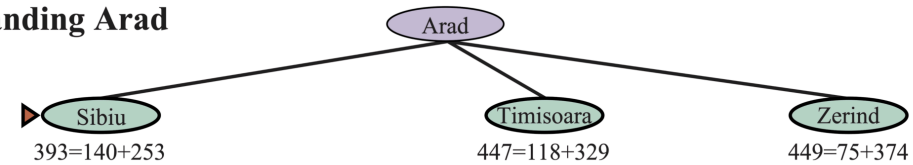


Source: (Russell and Norvig 2020, fig. 3.18)

The initial state is Arad, with a heuristic value of 366. Since no path has been traversed yet, the total estimated cost for this state is the sum of the traveled cost, which is 0, and the direct Euclidean distance to Bucharest, which is 366.

Book Example

(b) After expanding Arad

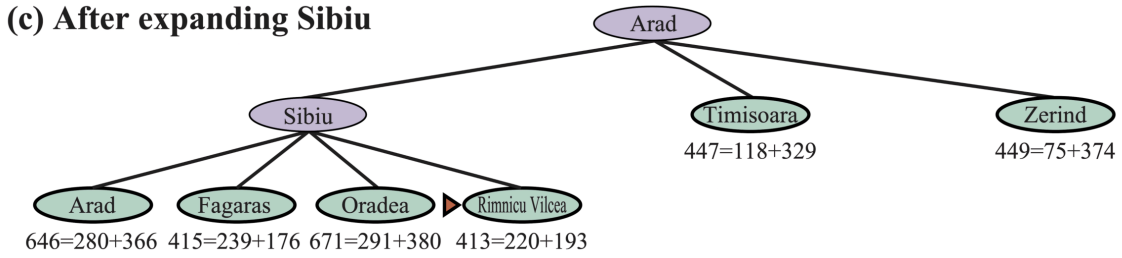


Source: (Russell and Norvig 2020, fig. 3.18)

From Arad, three cities can be reached directly: Sibiu, Timisoara, and Zerind. For each of these destinations, we evaluate the heuristic value by adding the distance traveled so far and the direct Euclidean distance to the final destination.

Sibiu has the lowest heuristic value and will thus be removed from the frontier.

Book Example



Source: (Russell and Norvig 2020, fig. 3.18)

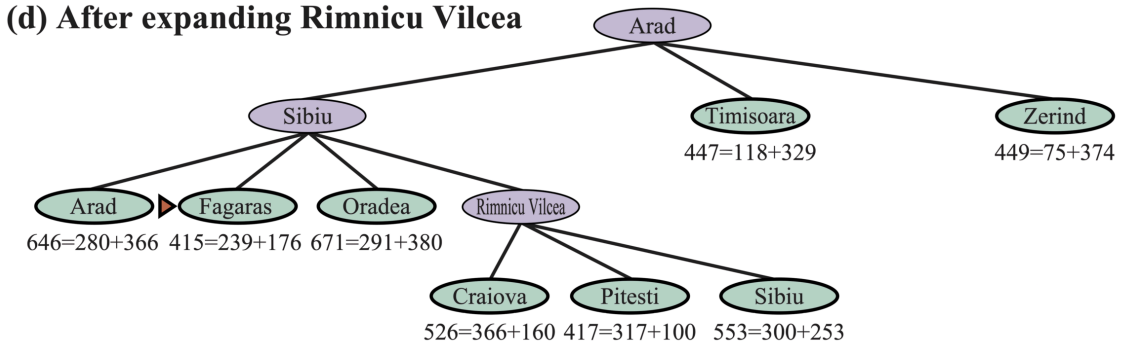
From Sibiu, four cities can be reached directly: Arad, Fagaras, Oradea, and Rimnicu Vilcea. For each of these destinations, we evaluate the heuristic value by adding the distance traveled so far and the direct Euclidean distance to the final destination.

In this example, the authors chose not to detect cycles, so Arad is added to the frontier.

Which city will be removed from the frontier next?

Rimnicu Vilcea has the lowest heuristic value and will thus be removed from the frontier.

Book Example



Source: (Russell and Norvig 2020, fig. 3.18)

From Rimnicu Vilcea, three cities are directly accessible: Craiova, Pitesti, and Sibiu. To evaluate the potential of each destination, we calculate a heuristic value by adding the distance already traveled and the direct Euclidean distance to the final destination.

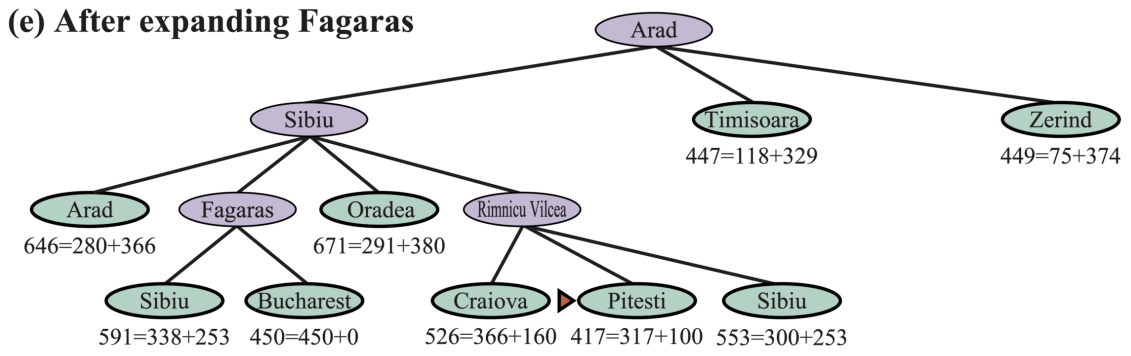
It is important to note that Sibiu is reintroduced into the frontier. Its current heuristic value is 553, resulting from the sum of the path traveled (300) and the direct Euclidean distance to Bucharest (253). When first introduced to the frontier, the path traveled was 140, yielding a heuristic value of 393. Although Sibiu presents a better potential than

alternative routes like Arad to Sibiu then to Arad, or Arad to Sibiu then to Oradea, it is not a good option compared to other possibilities.

Which city will be removed from the frontier next?

The next city to be removed from the frontier will be Fagaras, as it has the lowest heuristic value.

Book Example



Source: (Russell and Norvig 2020, fig. 3.18)

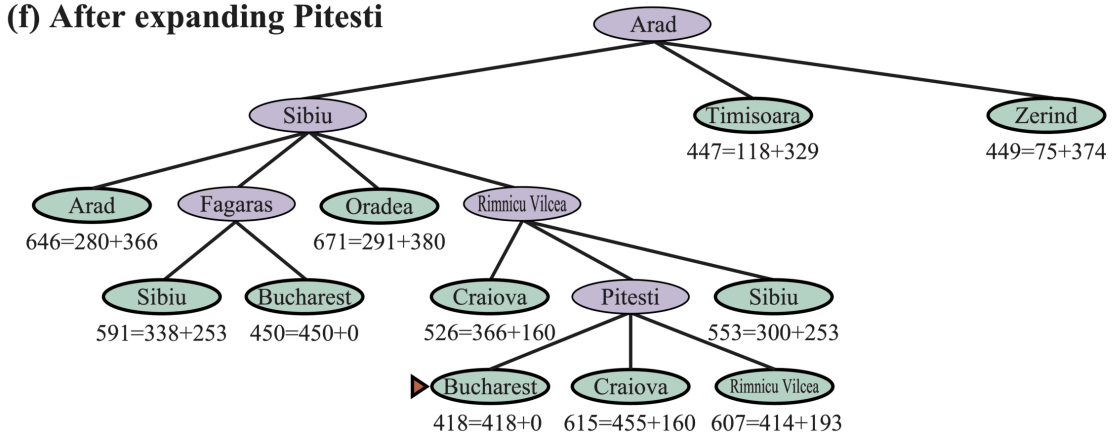
From Fagaras, two cities can be reached directly: Sibiu and Bucharest. For each of these destinations, we evaluate the heuristic value by adding the distance traveled so far and the direct Euclidean distance to the final destination.

Although Bucharest is the final goal, the algorithm does not stop immediately. After calculating the heuristic values, this path may not be the most promising. The heuristic value associated with Bucharest is 450, which corresponds only to the distance already traveled, since it is also the final destination. The direct Euclidean distance from the last node of the current path to the destination serves as an estimate. Although this estimate is initially imprecise, it becomes more accurate as we approach the destination.

A crucial aspect of the algorithm is that it does not terminate upon reaching the goal state; instead, it continues until the goal state is removed from the frontier. This indicates that no alternative path remains that appears more promising.

The nodes Pitesti, Timisoara, and Zerind, in this order, present more favorable options than Bucharest. Pitesti, having the lowest heuristic value, will therefore be selected to be removed from the frontier.

Book Example



Source: (Russell and Norvig 2020, fig. 3.18)

From Pitesti, three cities can be reached directly: Bucharest, Craiova, and Rimnicu Vilcea. For each of these destinations, we evaluate the heuristic value by adding the distance traveled so far and the direct Euclidean distance to the final destination.

The next city to be removed from the frontier will be Bucharest, as it has the lowest heuristic value. It is also the goal state, thus stopping the process. Arad, Sibiu, Rimnicu Vilcea, Pitesti, Bucharest is the returned solution.

Is this solution optimal in cost?

Implementation

- How can the existing **breadth-first** and **depth-first search** algorithms be modified to implement **best-first search**?
    - This can be achieved by employing a **priority queue**, which is sorted according to the values of the **heuristic function** $h(n)$.

. . .

In [11]:
```
import heapq
```

Remark

Breadth-first search can be interpreted as a form of **best-first search**, where the heuristic function $f(n)$ is defined as the **depth of the node within the search tree**, corresponding to the path length.

Is this solution viable? The answer is nuanced. It is useful for examining the properties of the algorithm, but using a queue will likely provide a more efficient implementation.

Can you think of a way to implement depth-first-search as a best-first-search?

A-star

$A^\star$ (a-star) is the **most common informed search**.

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

where

- $g(n)$ is the path cost from the *initial state* to n.
- $h(n)$ is an **estimate** of the cost of the shortest path from n to the *goal state*.

Hart, Nilsson, and Raphael (1968)

It is clear that $g(n)$ is a known value and not an estimate. Consequently, the accuracy of $f(n)$ improves as the execution progresses.

Admissibility

A heuristic is **admissible** if *it never overestimates the true cost to reach the goal from any node in the search space*.

. . .

This ensures that the $A^\star$ algorithm finds an optimal solution, as it guarantees that the estimated cost is always a lower bound on the actual cost.

What would happen if a heuristic were to **overestimate the cost** of the shortest path from n to the goal?

If a heuristic were to overestimate the cost of the shortest path from n to the goal, then $A^\star$ might ignore or delay expanding nodes that actually lie on the optimal path, because their f values look artificially expensive. The algorithm may still find a solution, but not necessarily the shortest one!

Admissibility

Formally, a heuristic $h(n)$ is admissible if:

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

where:

- $h(n)$ is the heuristic estimate of the cost from node n to the goal.
- $h^\star(n)$ is the actual cost of the optimal path from node n to the goal.

Cost Optimality

Cost optimality refers to an algorithm's ability to find the least-cost solution among all possible solutions.

. . .

In the context of search algorithms like $A^\star$, cost optimality means that the algorithm will identify the path with the lowest total cost from the start to the goal, assuming an admissible heuristic is used.

Theorem

Let h be **admissible**, i.e., $0 \leq h(n) \leq h^\star(n)$ for all nodes n, where $h^\star(n)$ is the true cost from n to a goal.

. . .

Assume non-negative action costs and that $A^\star$ **terminates when a goal is selected for expansion** (i.e., removed from the **frontier**).

. . .

Then $A^\star$ returns an optimal solution.

See also: Berkeley, CS 188, *Fall 2022 — Lecture Note 02: "Informed Search"*, September 2, 2022.

Proof

1. **Assume for contradiction**

Suppose $A^\star$ returns a suboptimal goal G with cost $C > C^\star$, where C^* is the optimal solution cost.

When $A^\star$ halts, G has just been selected from the frontier with $f(G) = g(G) = C$.

Proof

1. **Lower bound along any optimal path**

Consider any node n on an optimal path to an optimal goal $G^\star$ (cost $C^\star$).

By admissibility,

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

Thus every node on an optimal path has $f(n) \leq C^\star$.

Proof

1. **The first-unexpanded node is in frontier**

When G is chosen, walk from the start along an optimal path to $G^\star$ until you reach the **first node n that has not yet been expanded**.

. . .

Its parent on that path has been expanded (by definition of "first"), so n has been **generated** and is therefore in **frontier**.

. . .

From Step 2, $f(n) \leq C^\star$. Since $C^\star < C = f(G)$, $A^\star$'s best-first rule should choose n (or another node with $f \leq C^\star$) before G, a contradiction.

Proof

1. **Conclude optimality**

The contradiction shows $A^\star$ cannot return a goal with cost $> C^\star$; hence the returned solution is optimal.

Q.E.D.

8-Puzzle

Can you think of a heuristic function, $h(n)$, for the 8-Puzzle?

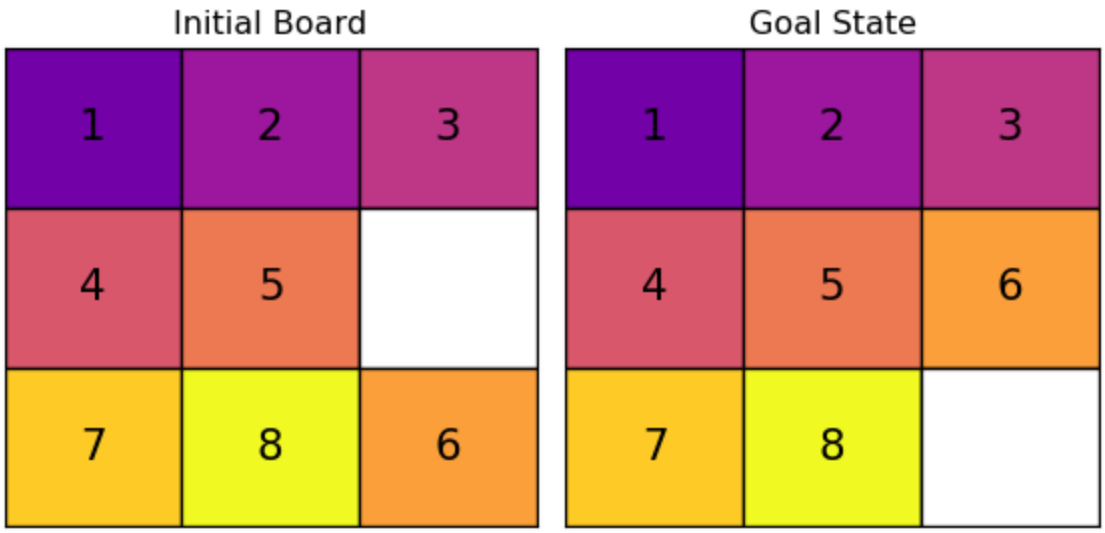
Number of Inversions Distance

Can the number of inversions be used as a heuristic function, $h(n)$, for the 8-Puzzle?

. . .

In [12]:
```
# Define the initial state
initial_board = [
    [1, 2, 3],
    [4, 5, 0],
    [7, 8, 6]
]

# Define goal state
goal_board = [
    [1, 2, 3],
```

```
    [4, 5, 6],
    [7, 8, 0]
]

# Plot both boards
plt.figure(figsize=(6, 3))
plot_board(initial_board, "Initial Board", 2, 1)
plot_board(goal_board, "Goal State", 2, 2)
plt.tight_layout()
plt.show()
```



- Calculate $h(s)$ and $h^\star(s)$.
- What do you conclude?

Intuitively, counting the number of inversions in the 8-Puzzle seems like it could serve as a heuristic — configurations with many inversions appear farther from the goal, which has none. However, since each move changes the inversion count by 0 or 2, it fails to provide a consistent or admissible measure of progress toward the goal.

For the initial board configuration, the heuristic function $h(s)$ has a value of 2, while the optimal cost, denoted $h^\star(s)$, is 1. Hence, $h(s) = 2 > h^\star(s) = 1$. As the inversion count can overestimate the true cost, it cannot be considered an admissible heuristic. This is because a single move can alter the relative order of the moved tile with respect to multiple other tiles, potentially decreasing the inversion count by more than 1 per move. Consequently, the inversion count fails to provide a valid lower bound on the number of actions required.

Be careful when constructing examples to reason about the problem. For instance, it may seem that the permutation “1, 2, 3, 4, 5, 6, 0, 8, 7” is only one move away from the goal, since it contains just one inversion. However, recall that the parity of the inversion count is invariant — each move changes it by 0 or 2. Because this configuration has an odd number of inversions, no sequence of actions can transform it into the goal state (the identity permutation).

Misplaced Tiles Distance

In [13]:
```
def  misplaced_tiles_distance(state, goal_state):

    # Count the number of misplaced tiles

    misplaced_tiles = sum(1 for s, g in zip(state, goal_state) if s != g and

    return misplaced_tiles
```

Is this heuristic admissible?

Consider any single legal move. It slides exactly one tile into the blank. That move can fix **at most one tile** that was previously misplaced; all other tiles keep their "misplaced vs. placed" status.

Therefore, along any solution path that uses k moves, the number of misplaced tiles can drop by at most k. If the start has $h(x)$ misplaced tiles, any solution must have length at least $h(s)$ steps.

Hence $h(s) \leq h^{\star}(s)$, i.e. the heuristic is admissible.

8-Puzzle

In [14]:
```
plt.figure(figsize=(8, 2))

initial_state_8a = [1, 2, 3,
                    4, 0, 6,
                    7, 5, 8]

initial_state_8b = [6, 4, 5,
                    8, 2, 7,
                    1, 0, 3]

goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

distance_a = misplaced_tiles_distance(initial_state_8a, goal_state_8)

distance_b = misplaced_tiles_distance(initial_state_8b, goal_state_8)

plot_board(initial_state_8a, f"h(n) = {distance_a}", 3, 1)

plot_board(goal_state_8, "Goal State", 3, 2)

plot_board(initial_state_8b, f"h(n) = {distance_b}", 3, 3)

plt.tight_layout()
plt.show()
```

Best-First Search

In [15]:
```python
def best_first_search(initial_state, goal_state):

    frontier = []  # Initialize the priority queue
    initial_h = misplaced_tiles_distance(initial_state, goal_state)
    # Push the initial state with its heuristic value onto the queue
    # (f(n), g(n), state, path)
    heapq.heappush(frontier, (initial_h, 0, initial_state, []))

    explored = set()
    explored.add(tuple(initial_state))

    iterations = 0

    while not is_empty(frontier):
        f, g, current_state, path = heapq.heappop(frontier)

        if is_goal(current_state, goal_state):
            print(f"Number of iterations: {iterations}")
            return path + [current_state]  # Return the successful path

        iterations = iterations + 1

        for neighbor in expand(current_state):
            if tuple(neighbor) not in explored:
                new_g = g + 1  # Increment the path cost
                h = misplaced_tiles_distance(neighbor, goal_state)
                new_f = new_g + h  # Calculate the new total cost
                # Push the neighbor state onto the priority queue
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [cu
                explored.add(tuple(neighbor))  # Mark neighbor as explored

    return None  # No solution found
```

Simple Case

In [16]:
```python
plt.figure(figsize=(8, 2))

initial_state_8 = [1, 2, 3,
                   4, 0, 6,
                   7, 5, 8]
goal_state_8 = [1, 2, 3,
```

```
                4, 5, 6,
                7, 8, 0]

solutions = best_first_search(initial_state_8, goal_state_8)

for i, solution in enumerate(solutions):
    plot_board(solution, f"Step: {i}", 3, i+1)

plt.tight_layout()
plt.show()
```

```
Number of iterations: 2
```

In [17]:
```
initial_state_8 = [1, 2, 3,
                   4, 0, 6,
                   7, 5, 8]
goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

best_first_search(initial_state_8, goal_state_8)
```

```
Number of iterations: 2
[[1, 2, 3, 4, 0, 6, 7, 5, 8],
 [1, 2, 3, 4, 5, 6, 7, 0, 8],
 [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

Challenging Case

In [18]:
```
initial_state_8 = [6, 4, 5,
                   8, 2, 7,
                   1, 0, 3]
goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

print("Solving 8-puzzle with best_first_search...")

solution_8_bfs = best_first_search(initial_state_8, goal_state_8)

if solution_8_bfs:
    print(f"Best_first_search Solution found in {len(solution_8_bfs) - 1} mo
    print_solution(solution_8_bfs)
else:
    print("No solution found for 8-puzzle using best_first_search.")
```

```
Solving 8-puzzle with best_first_search...
Number of iterations: 29005
Best_first_search Solution found in 25 moves:
Step 0:
6 4 5
8 2 7
1   3

Step 1:
6 4 5
8 2 7
  1 3

Step 2:
6 4 5
  2 7
8 1 3

Step 3:
6 4 5
2   7
8 1 3

Step 4:
6   5
2 4 7
8 1 3

Step 5:
  6 5
2 4 7
8 1 3

Step 6:
2 6 5
  4 7
8 1 3

Step 7:
2 6 5
4   7
8 1 3

Step 8:
2 6 5
4 1 7
8   3

Step 9:
2 6 5
4 1 7
  8 3

Step 10:
2 6 5
  1 7
```

```
4 8 3

Step 11:
2 6 5
1   7
4 8 3

Step 12:
2 6 5
1 7
4 8 3

Step 13:
2 6 5
1 7 3
4 8

Step 14:
2 6 5
1 7 3
4   8

Step 15:
2 6 5
1   3
4 7 8

Step 16:
2   5
1 6 3
4 7 8

Step 17:
2 5
1 6 3
4 7 8

Step 18:
2 5 3
1 6
4 7 8

Step 19:
2 5 3
1   6
4 7 8

Step 20:
2   3
1 5 6
4 7 8

Step 21:
  2 3
1 5 6
4 7 8
```

```
Step 22:
1 2 3
  5 6
4 7 8

Step 23:
1 2 3
4 5 6
  7 8

Step 24:
1 2 3
4 5 6
7   8

Step 25:
1 2 3
4 5 6
7 8
```

8-Puzzle

In [19]:
```python
def manhattan_distance(state, goal_state):
    distance = 0
    size = int(len(state) ** 0.5)
    for num in range(1, len(state)):
        idx1 = state.index(num)
        idx2 = goal_state.index(num)
        x1, y1 = idx1 % size, idx1 // size
        x2, y2 = idx2 % size, idx2 // size
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```

$$h_{\text{Manathan}}(s) = \sum_{t \in \{1,\ldots,8\}} (|x_t - x_t^\star| + |y_t - y_t^\star|)$$

Calculates the **Manhattan distance** heuristic for a given state. Is this heuristic **admissible**?

Consider a relaxed version of the 8-puzzle in which a tile may move to any adjacent square (up/down/left/right) without needing the blank. In that relaxed puzzle, the exact number of moves needed from a state is precisely the total Manhattan distance (each move reduces one tile's distance by 1 and you can always move the needed tile directly). The optimal cost of any relaxed problem is a lower bound on the optimal cost of the original problem, therefore the Manathan distance is admissible for the real 8-puzzle.

8-Puzzle

In [20]:
```
plt.figure(figsize=(8, 2))

initial_state_8a = [1, 2, 3,
                    4, 0, 6,
                    7, 5, 8]

initial_state_8b = [6, 4, 5,
                    8, 2, 7,
                    1, 0, 3]

goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

distance_a = manhattan_distance(initial_state_8a, goal_state_8)

distance_b = manhattan_distance(initial_state_8b, goal_state_8)

plot_board(initial_state_8a, f"h(n) = {distance_a}", 3, 1)

plot_board(goal_state_8, "Goal State", 3, 2)

plot_board(initial_state_8b, f"h(n) = {distance_b}", 3, 3)

plt.tight_layout()
plt.show()
```

8-Puzzle

- Compare **Manhattan** vs. **Misplaced Tiles** heuristics.
- Which is more effective?
- Significant run time differences?

8-Puzzle

In [21]:
```
plt.figure(figsize=(8, 2))

initial_state_8a = [1, 2, 3,
                    4, 0, 6,
                    7, 5, 8]

initial_state_8b = [6, 4, 5,
```

```
                    8, 2, 7,
                    1, 0, 3]

goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

distance_a_mis = misplaced_tiles_distance(initial_state_8a, goal_state_8)
distance_b_mis = misplaced_tiles_distance(initial_state_8b, goal_state_8)

distance_a_man = manhattan_distance(initial_state_8a, goal_state_8)
distance_b_man = manhattan_distance(initial_state_8b, goal_state_8)

plot_board(initial_state_8a, f"a = {distance_a_mis}, b = {distance_a_man}",

plot_board(goal_state_8, "Goal State", 3, 2)

plot_board(initial_state_8b, f"a = {distance_b_mis}, b = {distance_b_man}",

plt.tight_layout()
plt.show()
```

where

- **a** = misplaced tiles distance
- **b** = Manathan distance

8-Puzzle

In [22]:
```
plt.figure(figsize=(8, 2))

initial_state_8a = [3, 1, 2,
                    4, 5, 6,
                    7, 8, 0]

initial_state_8b = [8, 2, 3,
                    4, 5, 6,
                    1, 0, 7]

goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

distance_a_mis = misplaced_tiles_distance(initial_state_8a, goal_state_8)
distance_b_mis = misplaced_tiles_distance(initial_state_8b, goal_state_8)
```

```
distance_a_man = manhattan_distance(initial_state_8a, goal_state_8)
distance_b_man = manhattan_distance(initial_state_8b, goal_state_8)

plot_board(initial_state_8a, f"a = {distance_a_mis}, b = {distance_a_man}",

plot_board(goal_state_8, "Goal State", 3, 2)

plot_board(initial_state_8b, f"a = {distance_b_mis}, b = {distance_b_man}",

plt.tight_layout()
plt.show()
```

where

- **a** = misplaced tiles distance
- **b** = Manathan distance

`misplaced_tiles_distance` does not take into account how far a tile is from its expected final location, whereas `manhattan_distance` does.

Thus, one can expect the algorithm using the Manhattan distance to select the next node to explore more wisely.

Best-First Search

In [23]:
```
def best_first_search_revised(initial_state, goal_state):

    frontier = []  # Initialize the priority queue
    initial_h = manhattan_distance(initial_state, goal_state)
    # Push the initial state with its heuristic value onto the queue
    heapq.heappush(frontier, (initial_h, 0, initial_state, []))  # (f(n), g(

    explored = set()

    iterations = 0

    while not is_empty(frontier):
        f, g, current_state, path = heapq.heappop(frontier)

        if is_goal(current_state, goal_state):
            print(f"Number of iterations: {iterations}")
            return path + [current_state]  # Return the successful path
```

```
        iterations = iterations + 1

        explored.add(tuple(current_state))

        for neighbor in expand(current_state):
            if tuple(neighbor) not in explored:
                new_g = g + 1  # Increment the path cost
                h = manhattan_distance(neighbor, goal_state)
                new_f = new_g + h  # Calculate the new total cost
                # Push the neighbor state onto the priority queue
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [cu
                explored.add(tuple(neighbor))  # Mark neighbor as explored

    return None  # No solution found
```

When expanding a node in the search tree, it is crucial to note that we do not immediately evaluate its descendant nodes to determine if they are goal nodes. Instead, these descendants are added to the frontier, which is a priority queue. This allows their $f(n)$ values to be compared with those of other nodes in the frontier, ensuring that the most promising nodes are considered first based on their estimated total cost.

Consequently, it is possible for the frontier to include nodes that are goal nodes. However, their $f(n)$ values may be higher than those of other nodes deemed more promising, as these other nodes might lead to a shorter path to the goal.

Simple Case

In [24]:
```
plt.figure(figsize=(8, 2))

initial_state_8 = [1, 2, 3,
                   4, 0, 6,
                   7, 5, 8]
goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

solutions = best_first_search_revised(initial_state_8, goal_state_8)

for i, solution in enumerate(solutions):
    plot_board(solution, f"Step: {i}", 3, i+1)

plt.tight_layout()
plt.show()
```

```
Number of iterations: 2
```

In [25]:
```
initial_state_8 = [1, 2, 3,
                   4, 0, 6,
                   7, 5, 8]
goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

best_first_search_revised(initial_state_8, goal_state_8)
```

```
Number of iterations: 2
[[1, 2, 3, 4, 0, 6, 7, 5, 8],
 [1, 2, 3, 4, 5, 6, 7, 0, 8],
 [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

Challenging Case

In [26]:
```
initial_state_8 = [6, 4, 5,
                   8, 2, 7,
                   1, 0, 3]
goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

print("Solving 8-puzzle with best_first_search...")

solution_8_bfs = best_first_search_revised(initial_state_8, goal_state_8)

if solution_8_bfs:
    print(f"Best_first_search Solution found in {len(solution_8_bfs) - 1} mo
    print_solution(solution_8_bfs)
else:
    print("No solution found for 8-puzzle using best_first_search.")
```

```
Solving 8-puzzle with best_first_search...
Number of iterations: 2255
Best_first_search Solution found in 25 moves:
Step 0:
6 4 5
8 2 7
1   3

Step 1:
6 4 5
8 2 7
  1 3

Step 2:
6 4 5
  2 7
8 1 3

Step 3:
6 4 5
2   7
8 1 3

Step 4:
6   5
2 4 7
8 1 3

Step 5:
  6 5
2 4 7
8 1 3

Step 6:
2 6 5
  4 7
8 1 3

Step 7:
2 6 5
4   7
8 1 3

Step 8:
2 6 5
4 1 7
8   3

Step 9:
2 6 5
4 1 7
  8 3

Step 10:
2 6 5
  1 7
```

```
4 8 3

Step 11:
2 6 5
1   7
4 8 3

Step 12:
2 6 5
1 7
4 8 3

Step 13:
2 6 5
1 7 3
4 8

Step 14:
2 6 5
1 7 3
4   8

Step 15:
2 6 5
1   3
4 7 8

Step 16:
2   5
1 6 3
4 7 8

Step 17:
2 5
1 6 3
4 7 8

Step 18:
2 5 3
1 6
4 7 8

Step 19:
2 5 3
1   6
4 7 8

Step 20:
2   3
1 5 6
4 7 8

Step 21:
  2 3
1 5 6
4 7 8
```

```
Step 22:
1 2 3
  5 6
4 7 8

Step 23:
1 2 3
4 5 6
  7 8

Step 24:
1 2 3
4 5 6
7   8

Step 25:
1 2 3
4 5 6
7 8
```

Experiments

In [28]:

```python
def best_first_search_count(initial_state, goal_state):
    frontier = []  # Initialize the priority queue
    initial_h = misplaced_tiles_distance(initial_state, goal_state)
    # Add the initial state with its heuristic value to the queue
    heapq.heappush(frontier, (initial_h, 0, initial_state, []))  # (f(n), g(

    explored = set()
    explored.add(tuple(initial_state))

    iterations = 0

    while not is_empty(frontier):
        f, g, current_state, path = heapq.heappop(frontier)

        if is_goal(current_state, goal_state):
            return len(path), iterations

        iterations = iterations + 1

        for neighbor in expand(current_state):
            if tuple(neighbor) not in explored:
                new_g = g + 1  # Increment the path cost
                h = misplaced_tiles_distance(neighbor, goal_state)
                new_f = new_g + h  # Compute the new total cost
                # Add the neighboring state to the priority queue
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [cu
                explored.add(tuple(neighbor))  # Mark the neighbor as explor

    return None, None  # No solution found
```

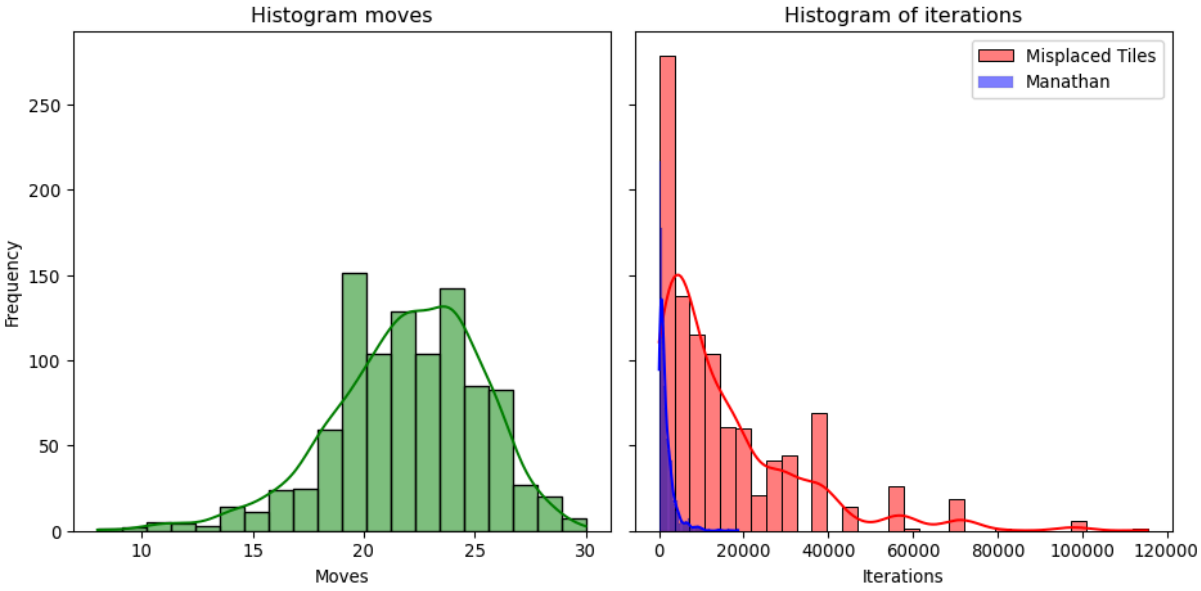
In [29]:
```
def best_first_search_revised_count(initial_state, goal_state):
    frontier = []  # Initialize the priority queue
    initial_h = manhattan_distance(initial_state, goal_state)
    # Add the initial state with its heuristic value to the queue
    heapq.heappush(frontier, (initial_h, 0, initial_state, []))  # (f(n), g(

    explored = set()
    explored.add(tuple(initial_state))

    iterations = 0

    while not is_empty(frontier):
        f, g, current_state, path = heapq.heappop(frontier)

        if is_goal(current_state, goal_state):
            return len(path), iterations

        iterations = iterations + 1

        for neighbor in expand(current_state):
            if tuple(neighbor) not in explored:
                new_g = g + 1  # Increment the path cost
                h = manhattan_distance(neighbor, goal_state)
                new_f = new_g + h  # Compute the new total cost
                # Add the neighboring state to the priority queue
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [cu
                explored.add(tuple(neighbor))  # Mark the neighbor as explor

    return None, None  # No solution found
```

We introduce two novel functions: each returning the length of the solution as well as the number of iterations performed. These functions will be used in our upcoming experimental analyses.

1000 Experiments

In [30]:
```
goal_state_8 = [1, 2, 3,
                4, 5, 6,
                7, 8, 0]

moves = []
iterations_a = []
iterations_b = []

for i in range(1000):
    initial_state_8 = generate_solvable_board()
    nb_moves, nb_iterations_a = best_first_search_count(initial_state_8, goa
    nb_moves, nb_iterations_b = best_first_search_revised_count(initial_stat
    moves.append(nb_moves)
    iterations_a.append(nb_iterations_a)
    iterations_b.append(nb_iterations_b)

import seaborn as sns
```

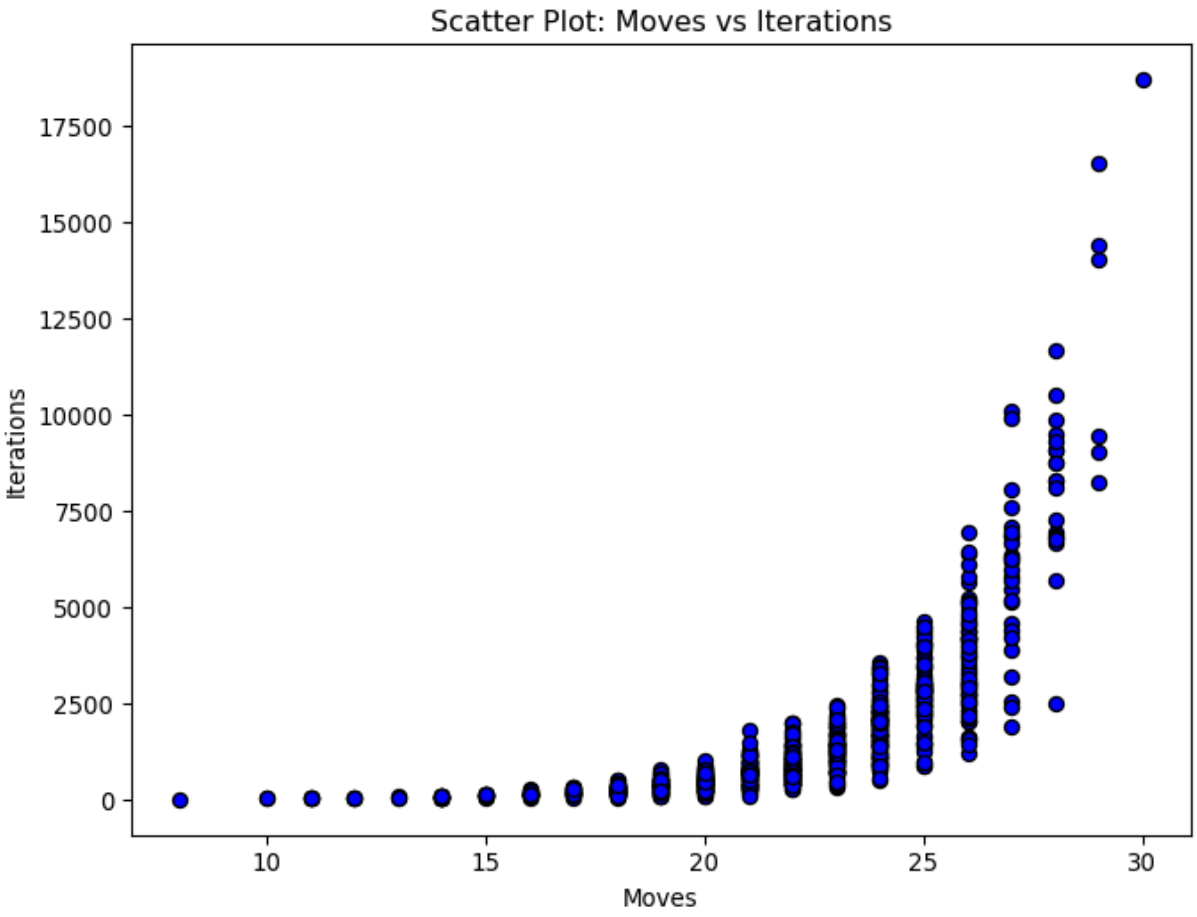
```
import matplotlib.pyplot as plt

# Set up the subplot structure
fig, axes = plt.subplots(1, 2, figsize=(10, 5), sharey=True)

# Plot histogram for `moves`
sns.histplot(moves, bins=20, kde=True, color='green', ax=axes[0], edgecolor=
axes[0].set_title("Histogram moves")
axes[0].set_xlabel("Moves")
axes[0].set_ylabel("Frequency")

# Plot histogram for `iterations`
sns.histplot(iterations_a, kde=True, color='red', ax=axes[1], edgecolor='bla

# Plot histogram for `iterations`
sns.histplot(iterations_b, kde=True, color='blue', ax=axes[1], edgecolor='bl
axes[1].set_title("Histogram of iterations")
axes[1].set_xlabel("Iterations")

# Show the plot
plt.legend()
plt.tight_layout()
plt.show()
```



Scatter Plot (Manathan)

In [31]:
```
# Create scatter plot

plt.figure(figsize=(8, 6))
plt.scatter(moves, iterations_b, color='blue', edgecolor='black')

# Add titles and labels
plt.title("Scatter Plot: Moves vs Iterations")
plt.xlabel("Moves")
plt.ylabel("Iterations")
```

```
# Display the plot
plt.show()
```



Exploration

Breadth-first search (BFS) is guaranteed to find the shortest path, or lowest-cost solution, assuming all actions have unit cost.

Develop a program that performs the following tasks:

1. Generate a random configuration of the 8-Puzzle.
2. Determine the shortest path using breadth-first search.
3. Identify the optimal solution using the $A^\star$ algorithm.
4. Compare the costs of the solutions obtained in steps 2 and 3. There should be no discrepancy if $A^\star$ identifies cost-optimal solutions.
5. Repeat the process.

Exploration

The heuristic $h(n) = 0$ is considered admissible, yet it typically results in inefficient exploration of the search space. Develop a program to investigate this concept. Demonstrate that when all actions are assumed to have unit cost, both $A^\star$ and breadth-

first search (BFS) explore the search space similarly. Specifically, they examine all paths of length one, followed by paths of length two, and so forth.

Remarks

- **Breadth-first search (BFS)** identifies the optimal solution, 25 moves, in 145,605 iterations.
- **Depth-first search (DFS)** discovers a solution involving 1,157 moves in 1,187 iterations.
- **Best-First Search ** using the Manathan distance identifies the optimal solution, 25 moves, in 2,255 iterations.

Measuring Performance

- **Completeness**: Does the algorithm **ensure** that a solution will be found if one exists, and accurately indicate failure when no solution exists?
- **Cost Optimality**: Does the algorithm identify the (a) solution with the **lowest path cost** among all possible solutions?
- Are all three algorithms complete? What are the necessary conditions?
- Do all three algorithms guarantee cost optimality?

Measuring Performance

- **Time Complexity**: How does the **time** required by the algorithm scale with respect to the **number of states** and **actions**?
- **Space Complexity**: How does the **space** required by the algorithm scale with respect to the **number of states** and **actions**?

Alternatively, complexity can be evaluated based on the **depth** (d) and **branching factor** (b) of the search tree, instead of the number of states (nodes) and actions (edges) in the state space.

- What is the time and space complexity of breadth-first-search? $\mathcal{O}(b^d)$

Videos by Sebastian Lague

- A* Pathfinding (E01: algorithm explanation) posted on 2014-12-16.
- A* Pathfinding (E02: node grid) posted on 2014-12-18.

- A* Pathfinding (E03: algorithm implementation) posted on 2014-12-19.
- A* Pathfinding (E04: heap optimization) posted on 2014-12-24.
- A* Pathfinding (E05: units) posted on 2015-01-06.
- A* Pathfinding (E06: weights) posted on 2015-01-11.
- A* Pathfinding (E07: smooth weights) posted on 2016-12-30.
- A* Pathfinding (E08: path smoothing 1/2) posted on 2017-01-31.
- A* Pathfinding (E09: path smoothing 2/2) posted on 2017-01-31.
- A* Pathfinding (E10: threading) posted on 2017-02-03.
- A* Pathfinding Tutorial (Unity) (play list)

A resource dedicated to $A^\star$

- Amit's A* Pages

Prologue

Summary

- Informed Search and Heuristics
  - Best-First Search
- Implementations

- **Informed Search and Heuristics:**
  - Introduced the concept of **heuristic functions** (`h(n)`) to estimate costs.
  - **Best-First Search:**
    - Uses heuristics to prioritize nodes that seem closer to the goal.
    - Implemented with a priority queue sorted by estimated cost.
  - **Manhattan Distance Heuristic:**
    - Calculates the sum of the distances of tiles from their goal positions.
    - Used in the 8-Puzzle to guide the search more efficiently.
- **Comparative Analysis:**
  - **BFS:** Optimal solution in 145,605 iterations (25 moves).
  - **DFS:** Suboptimal solution in 1,187 iterations (1,157 moves).
  - **Best-First Search:** Optimal solution in 2,255 iterations (25 moves).
  - Demonstrated that informed search algorithms can find optimal solutions more efficiently.

Next lecture

- We will examine additional search algorithms.

References

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. 1968. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” *IEEE Transactions on Systems Science and Cybernetics* 4 (2): 100–107. https://doi.org/10.1109/tssc.1968.300136.

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. http://aima.cs.berkeley.edu/.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa