IA Planning
Lecture 1: Introduction to Planning

Halim Djerroud

LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ université PARIS-SACLAY

revision: 0.1

Session Outline

1. Motivation and applications of Task Planning
2. Basic notions: states, actions, goals
3. Concrete examples: 8-puzzle, robot navigation on a grid
4. Practical session: first state-search problems (BFS/DFS in Python)

Introduction

Understanding the motivations and applications of task planning in AI.

Chapter objectives:

1. Motivations for task planning in AI
2. Applications

Why task planning in AI?

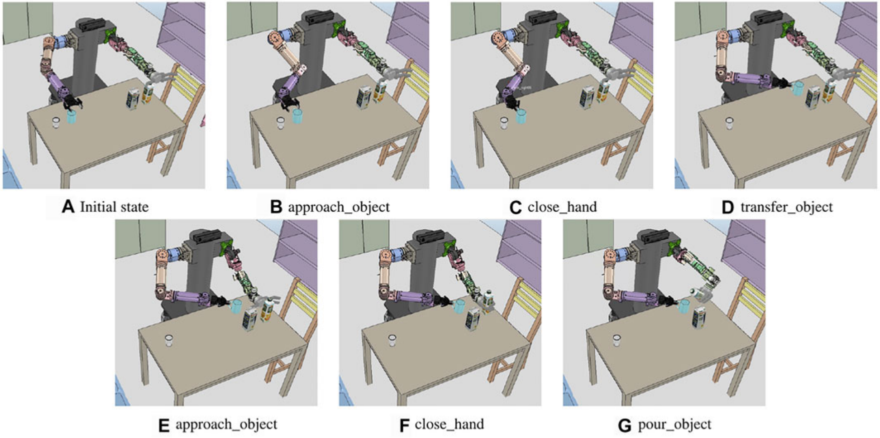
- **Automate sequential decision-making** : Example: a domestic robot that chooses the order to tidy the house (pick up objects, vacuum, then empty the trash).
- **Optimize resource utilization** : Example: a drone that must deliver packages while consuming the least battery possible.
- **Manage the complexity of real environments** :
  - Multiple possible actions: a robot can take different paths to reach a room.
  - Interactions between agents: multiple robots must coordinate their movements to avoid interfering with each other.
  - Safety constraints: a robotic arm must avoid dangerous zones.
- **Generic approach** : The same planner can solve various problems: – Playing chess, – Organizing a schedule, – Planning industrial robotics operations.

LISV Laboratoire d'ingénierie des systèmes de Versailles UVSQ université PARIS-SACLAY

Applications of planning in AI

- **Robotics**
  - Autonomous navigation (e.g., mobile robots).
  - Object manipulation (industrial robots, service robots).
- **Logistics**
  - Delivery organization (Amazon, DHL).
  - Production chain optimization.
- **Video games and simulation**
  - Strategic AI in role-playing or real-time strategy games.
  - Autonomous virtual agents.
- **Scientific and space planning**
  - Space missions (Mars rover, DeepSpace, ...).
  - Laboratory experiment planning.

Example: planning in robotics



Summary: motivations and applications

Key idea

Planning in AI provides a formal framework to **define, generate and evaluate action plans**, enabling artificial agents to solve complex problems in various environments.

- Allows problem abstraction (domain independence).
- Applies to concrete sectors: robotics, logistics, games, sciences.
- First step before introducing formal concepts: *states, actions, objectives*.

Chapter 2: Basic Concepts

Chapter 2: States, actions and objectives

Chapter objectives:

1. Define the fundamental concepts of planning.
2. Understand the relationship between states, actions and objectives.
3. Introduce the concept of state space.

Definition: State

A **state** = a *snapshot of the world*, which completely describes the situation at a given moment.

- Represented by a set of **variables** or **facts**.
- Example 1 (mobile robot on a grid) :

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & R & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \Rightarrow \quad R \text{ is at } (2,2)$$

- Example 2 (8-puzzle) : Here $\square$ represents the empty cell.

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

Common notation: $s_0, s_1, \ldots, s_n$

Variables vs. Facts

Variables

- Describe the world using **attributes** and **values**.
- Each state is defined by an assignment of values to the variables.
- Example (robot on a grid):
  - $PosX = 2$
  - $PosY = 3$
  - $HasBox = \text{False}$

Facts

- Describe the world as **true/false propositions**.
- Each state = a set of true facts.
- Example (robot on a grid):
  - $At(Robot, 2, 3)$ is true
  - $At(Box, 1, 1)$ is true
  - $Carrying(Robot, Box)$ is false

$\Rightarrow$ Variables describe a state through *values*, while facts describe it through *propositions*.

Definition: Action

An **action** = an operation that transforms a state s_i into a new state s_{i+1}.

- Defined by:
  - **Preconditions**: necessary conditions to apply it.
  - **Effects**: modifications produced on the state.
- Example (robot on a grid):

Action $Move(North)$

$$\underbrace{\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & R & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}}_{\text{State before}} \xrightarrow{Move(North)} \underbrace{\begin{bmatrix} \cdot & R & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}}_{\text{State after}}$$

Precondition: no obstacle to the north. Effect: $R(x, y) \mapsto R(x, y+1)$.

Definition: Objective

An **objective** = a state (or set of states) that we seek to reach.

- Represented by a set of **conditions to satisfy**.
- Example 1 (8-puzzle):

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \square \end{bmatrix} \Rightarrow \quad \text{objective reached}$$

- Example 2 (grid navigation):

$$\begin{bmatrix} \cdot & \cdot & G \\ \cdot & R & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \Rightarrow \quad R \text{ must reach } G$$

- A **plan** = sequence of actions that leads from the initial state s_0 to a goal state s_g.

LISV Laboratoire d'ingénierie des systèmes de Versailles

UVSQ université PARIS-SACLAY

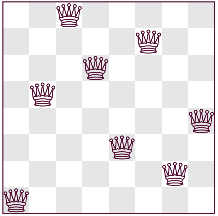
Known vs. Unknown Objectives

Known objective

- The goal state (or conditions) is clearly specified.
- Example: in the 8-puzzle, the objective is to reach the ordered configuration.

Unknown / Implicit objective

- The objective is not directly given; it must be discovered or constructed.
- Example: in the 8-queens problem, the objective is to place 8 queens such that no two attack each other.
- This is a *constraint satisfaction problem*.



Example: 8-Queens problem

$\Rightarrow$ Sometimes the objective is a *target state*, sometimes a set of *constraints*.

Definition: Plan

A **plan** = a sequence of actions that transforms the initial state s_0 into a goal state s_g.

- General form:

$$Plan = \langle a_1, a_2, \ldots, a_n \rangle$$

with $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_g$.

- Example (robot on a grid):

$$\underbrace{\begin{bmatrix} \cdot & \cdot & G \\ \cdot & R & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}}_{\text{Initial state}} \xrightarrow{Move(North)} \begin{bmatrix} \cdot & R & G \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \xrightarrow{Move(East)} \underbrace{\begin{bmatrix} \cdot & \cdot & R \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}}_{\text{Goal state}}$$

- Here, the plan is: $\langle Move(North), Move(East) \rangle$.

Definition: State Space

The **state space** = the set of all possible states, connected by the actions that allow transitioning from one to another.

- Each **node** represents a state.
- Each **edge** represents an action.
- A **plan** corresponds to a path between the initial state s_0 and a goal state s_g.

Example (robot on a 2×2 grid):
Possible states: positions of R.

$$\begin{bmatrix} R & \cdot \\ \cdot & \cdot \end{bmatrix} \xleftrightarrow{Move(East)} \begin{bmatrix} \cdot & R \\ \cdot & \cdot \end{bmatrix}$$

$$\begin{bmatrix} \cdot & \cdot \\ R & \cdot \end{bmatrix} \xleftrightarrow{Move(East)} \begin{bmatrix} \cdot & \cdot \\ \cdot & R \end{bmatrix}$$

$\Rightarrow$ The state space is often represented as a **search graph**.

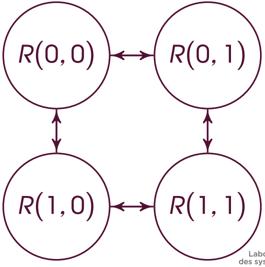
State Space = Search Graph

The state space can be represented by a **graph**:

- The **nodes** = the possible states.
- The **edges** = the actions that connect the states.
- A **plan** = a path in this graph.

Example (robot on a 2×2 grid):

$$\begin{bmatrix} R & \cdot \\ \cdot & \cdot \end{bmatrix}, \begin{bmatrix} \cdot & R \\ \cdot & \cdot \end{bmatrix}, \begin{bmatrix} \cdot & \cdot \\ R & \cdot \end{bmatrix}, \begin{bmatrix} \cdot & \cdot \\ \cdot & R \end{bmatrix}$$



Graphs and Exploration

A **graph** represents the state space:

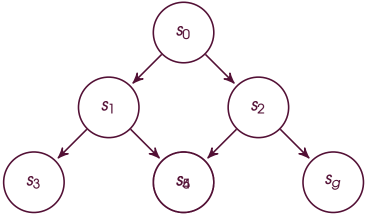
- The **nodes** = the states. The **edges** = the possible actions.
- Finding a plan = searching for a path between the initial state and the goal state.

BFS (Breadth-First Search)

- Explores first all states at distance 1, then 2, etc.
- Finds the shortest path (in number of actions).

DFS (Depth-First Search)

- Explores a path in depth before backtracking.
- Can find a solution quickly, but not necessarily the shortest.



BFS and DFS are two strategies to explore this graph and find a plan.

Chapter 3: Concrete Examples

Chapter 3: 8-puzzle and grid navigation

Objectives of this chapter:

1. Illustrate the notions of states, actions, and goals with classical examples.
2. Understand the construction of a state space.
3. Provide the foundations for planning search algorithms.

Example 1: the 8-puzzle

- **State**: configuration of the 8 tiles + empty cell.
- **Actions**: move an adjacent tile into the empty cell.
- **Goal**: reach the sorted configuration.

Initial state:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 6 \\ 7 & 5 & 8 \end{bmatrix}$$

Goal state:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

State transitions

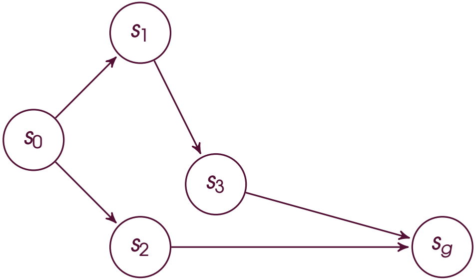
- One action = moving a tile into the empty cell.
- Each action leads to a **new state**.

$$
\begin{tikzcd}
\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 7 & 5 & 8 \end{bmatrix} & & \begin{bmatrix} 1 & 0 & 3 \\ 4 & 2 & 6 \\ 7 & 5 & 8 \end{bmatrix} \\
& \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 6 \\ 7 & 5 & 8 \end{bmatrix} \arrow[ul] \arrow[ur] \arrow[dl] \arrow[dr] & \\
\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{bmatrix} & & \begin{bmatrix} 1 & 2 & 3 \\ 4 & 6 & 0 \\ 7 & 5 & 8 \end{bmatrix}
\end{tikzcd}
$$

State space of the 8-puzzle

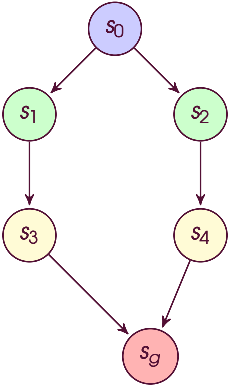
The set of all possible configurations forms a **search graph**.

- Nodes = possible configurations (up to 181,440 reachable).
- Edges = tile moves.
- A plan = a path from the initial state to the goal.



Exploration with BFS

- Explores all states at distance 1, then distance 2, etc.
- Guarantees finding the **shortest solution**.



Exploration with DFS

- Explores one path in depth before backtracking.
- May find a solution quickly, but not necessarily the shortest.

Example 2: Grid navigation

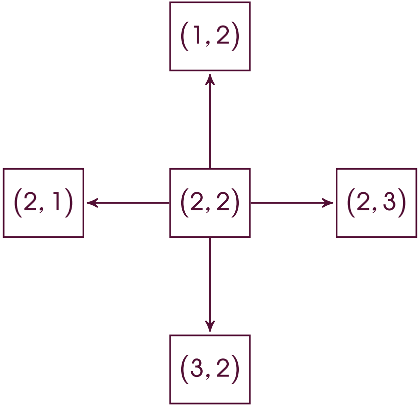
- **State**: robot position (x, y).
- **Actions**: `up`, `down`, `left`, `right` (if no obstacle).
- **Goal**: reach a target cell (x_g, y_g).

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & S & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & G \end{bmatrix}$$

$S =$ start, $G =$ goal, $1 =$ obstacle, $0 =$ free cell

State transitions

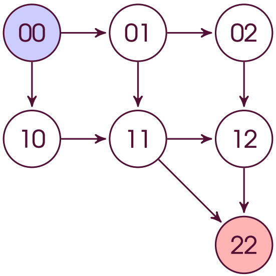
- Each valid move leads to a **new state**.
- Example (robot at $(2,2)$): can move left, right, or down.



State space for navigation

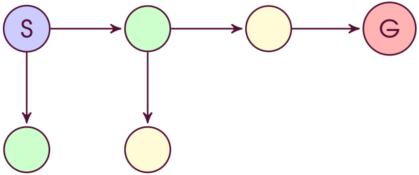
The grid becomes a **graph**:

- Nodes = possible positions.
- Edges = allowed moves.
- Plan = path from start S to goal G.



Exploration with BFS

- Explores the grid in "layers" around the start.
- Finds the shortest path (in number of steps).



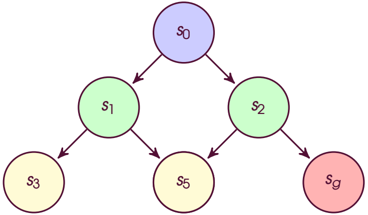
Breadth-First Search (BFS) Algorithm

Pseudo-code (BFS)

Input: Initial state s_0, Goal test
Output: Path from s_0 to s_g if found

Create an empty queue Q;
Enqueue (s_0) into Q;
Mark s_0 as visited;
while *Q not empty* **do**
  Dequeue first path $(s_0 \ldots s_i)$;
  Let $n =$ last node of path;
  if *n is goal* **then**
    return path $(s_0 \ldots s_i)$;
  end
  foreach *successor n' of n* **do**
    if *n' not visited* **then**
      Mark n' visited;
      Enqueue path $(s_0 \ldots s_i, n')$;
    end
  end
end

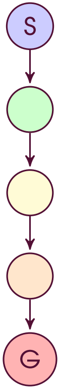
Exploration order (layers)



$\Rightarrow$ BFS explores the graph level by level, guaranteeing the shortest solution (in number of actions).

Exploration with DFS

- Explores one path in depth before backtracking.
- May find a solution quickly, but not necessarily optimal.



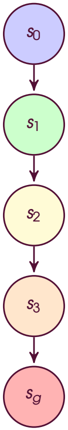
Depth-First Search (DFS) Algorithm

Pseudo-code (DFS)

Input: Initial state s_0, Goal test
Output: Path from s_0 to s_g if found

Create an empty stack S;
Push (s_0) onto S;
Mark s_0 as visited;
while *S not empty* **do**
  Pop last path $(s_0 \ldots s_i)$;
  Let $n =$ last node of path;
  if *n is goal* **then**
    return path $(s_0 \ldots s_i)$;
  end
  foreach *successor n' of n* **do**
    if *n' not visited* **then**
      Mark n' visited;
      Push path $(s_0 \ldots s_i, n')$;
    end
  end
end

Exploration order (depth-first)



$\Rightarrow$ DFS explores one path in depth before backtracking. It may find a solution faster, but not necessarily the shortest one.

Chapter 4: A* Search Algorithm

Chapter 4: The A* Search Algorithm

Optimal Pathfinding with Heuristic Guidance

Learning Objectives:

1. Master the A* algorithm and its evaluation function $f(n) = g(n) + h(n)$
2. Understand admissible heuristics and their critical role in optimality
3. Analyze A* performance compared to BFS, DFS, UCS, and Greedy Best-First
4. Apply A* to real-world pathfinding problems

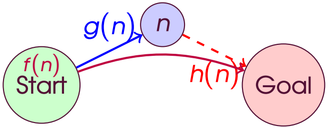
What is A* Search?

Key Insight

A* combines the **actual cost** to reach a node with an **estimated cost** to the goal, making informed decisions about which paths to explore first.

The A* Evaluation Function:

$$\boxed{f(n) = g(n) + h(n)}$$



- $g(n)$ = **actual cost** from start to node n
- $h(n)$ = **heuristic estimate** from n to goal
- $f(n)$ = **estimated total cost** of best path through n

A* (A Star)

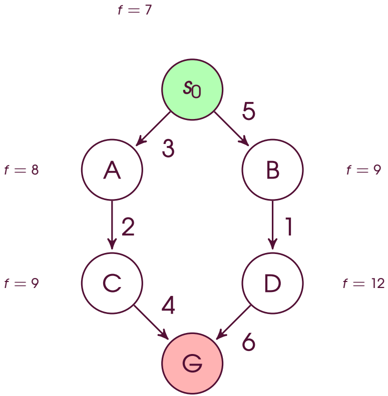
Input: Initial state s_0, Goal test, heuristic h
Output: Optimal path from s_0 to goal

```
Open ← {s0}                      // priority queue by f
Closed ← ∅                       // explored set
g(s0) ← 0; f(s0) ← h(s0)
while Open ≠ ∅ do
    n ← PopMin_f(Open)
    if n is goal then
        return path to n
    end
    add n to Closed
    foreach successor n′ of n do
        // Process successor (see right column)
    end
end
return failure
```

```
ProcessSuccessor(n, n′) if n′ ∈ Closed then
    continue
end
g_new ← g(n) + cost(n, n′)
if n′ ∉ Open or g_new < g(n′) then
    g(n′) ← g_new
    f(n′) ← g(n′) + h(n′)
    parent(n′) ← n
    Insert/DecreaseKey n′ in Open
end
```

Algorithm 1: *

A* Example (Step-by-Step)



Search Order:

1. Expand s_0 ($f = 7$)
2. Expand A ($f = 8$)
3. Expand C ($f = 9$)
4. Goal found: $s_0 \rightarrow A \rightarrow C \rightarrow G$

Key: A* explores the most promising nodes first!

Heuristic Functions: The Heart of A*

Admissible Heuristics

A heuristic $h(n)$ is **admissible** if it never overestimates the true cost: $h(n) \leq h^*(n)$ for all n where $h^*(n)$ is the actual minimum cost from n to goal.

Common Heuristics: 1. Grid Navigation

- **Manhattan Distance** (4-connected):

$$h(n) = |x_n - x_g| + |y_n - y_g|$$

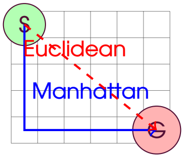
- **Euclidean Distance** (continuous):

$$h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2}$$

- **Chebyshev Distance** (8-connected):

$$h(n) = \max(|x_n - x_g|, |y_n - y_g|)$$

Heuristic Quality Impact:



Heuristic Dominance:
If $h_2(n) \geq h_1(n)$ for all n, then h_2 **dominates** h_1 and will expand fewer nodes.

Properties and Guarantees of A*

Theoretical Properties

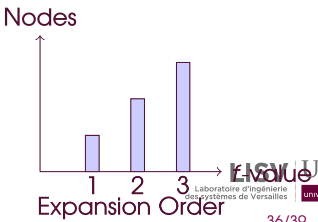
- **Completeness**: Finds solution if one exists (finite branching factor)
- **Optimality**: Guarantees optimal solution when h is admissible
- **Optimally Efficient**: No other algorithm can expand fewer nodes with same heuristic

Complexity Analysis:

- **Time**: $O(b^d)$ worst case, where b = branching factor, d = depth
- **Space**: $O(b^d)$ - stores all generated nodes
- **Performance depends on**: heuristic accuracy, branching factor, solution depth

Why A* is Optimal:

1. A* always expands nodes in order of increasing f-value
2. If h is admissible, f never overestimates true cost
3. When goal is selected, no unexplored node can lead to better solution



Algorithm Comparison: A* vs Others

Comprehensive Comparison

Algorithm	Complete	Optimal	Time	Space	Uses
BFS	Yes	Yes*	$O(b^d)$	$O(b^d)$	Unit costs only
DFS	No**	No	$O(b^m)$	$O(bm)$	Memory efficient
UCS (Dijkstra)	Yes	Yes	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$	No heuristic
Greedy Best-First	No	No	$O(b^m)$	$O(b^m)$	Fast but risky
A*	Yes	Yes***	$O(b^d)$	$O(b^d)$	Best of both worlds

* BFS optimal only for unit costs

** DFS complete in finite spaces

*** A* optimal when heuristic is admissible

When to Choose Each Algorithm:

- **A***: When you have a good admissible heuristic and need optimal solutions
- **UCS**: When no heuristic is available but optimality is required
- **Greedy**: When speed matters more than optimality
- **DFS**: When memory is extremely limited and optimality not required

LISV Laboratoire d'ingénierie des systèmes de Versailles UVSQ université PARIS-SACLAY

Advanced Topics and Extensions

A* Variants:

- **Weighted A***:
  $f(n) = g(n) + w \cdot h(n)$
  - $w > 1$: faster but suboptimal
  - Trade optimality for speed
- **IDA* (Iterative Deepening A*)**:
  - Uses $O(d)$ space instead of $O(b^d)$
  - Good for memory-constrained problems
- **Bidirectional A***:
  - Search from both start and goal
  - Can reduce search space significantly

Heuristic Design Tips:

1. **Relaxation**: Remove constraints from original problem
2. **Pattern Databases**: Precompute costs for subproblems
3. **Landmark Heuristics**: Use intermediate waypoints
4. **Combination**: $h(n) = \max(h_1(n), h_2(n))$

Common Pitfalls

- Non-admissible heuristics $\Rightarrow$ suboptimal solutions
- Poor heuristics $\Rightarrow$ performance similar to Dijkstra
- Inconsistent heuristics $\Rightarrow$ need to reopen nodes

Summary and Key Takeaways

What Makes A* Special?

A* achieves the **best of both worlds**: the optimality guarantees of Dijkstra's algorithm with the efficiency of heuristic-guided search.

Core Concepts to Remember:

1. **Evaluation Function**: $f(n) = g(n) + h(n)$ balances known cost and estimated remaining cost
2. **Admissible Heuristics**: Essential for optimality - never overestimate true cost
3. **Optimal Efficiency**: No algorithm with same heuristic can expand fewer nodes
4. **Trade-offs**: Excellent performance but requires $O(b^d)$ memory

A* = Informed Search

Dijkstra (optimal) Greedy (fast)

Perfect balance of optimality and efficiency

Next Steps: Practice implementing A* with different heuristics and explore advanced variants