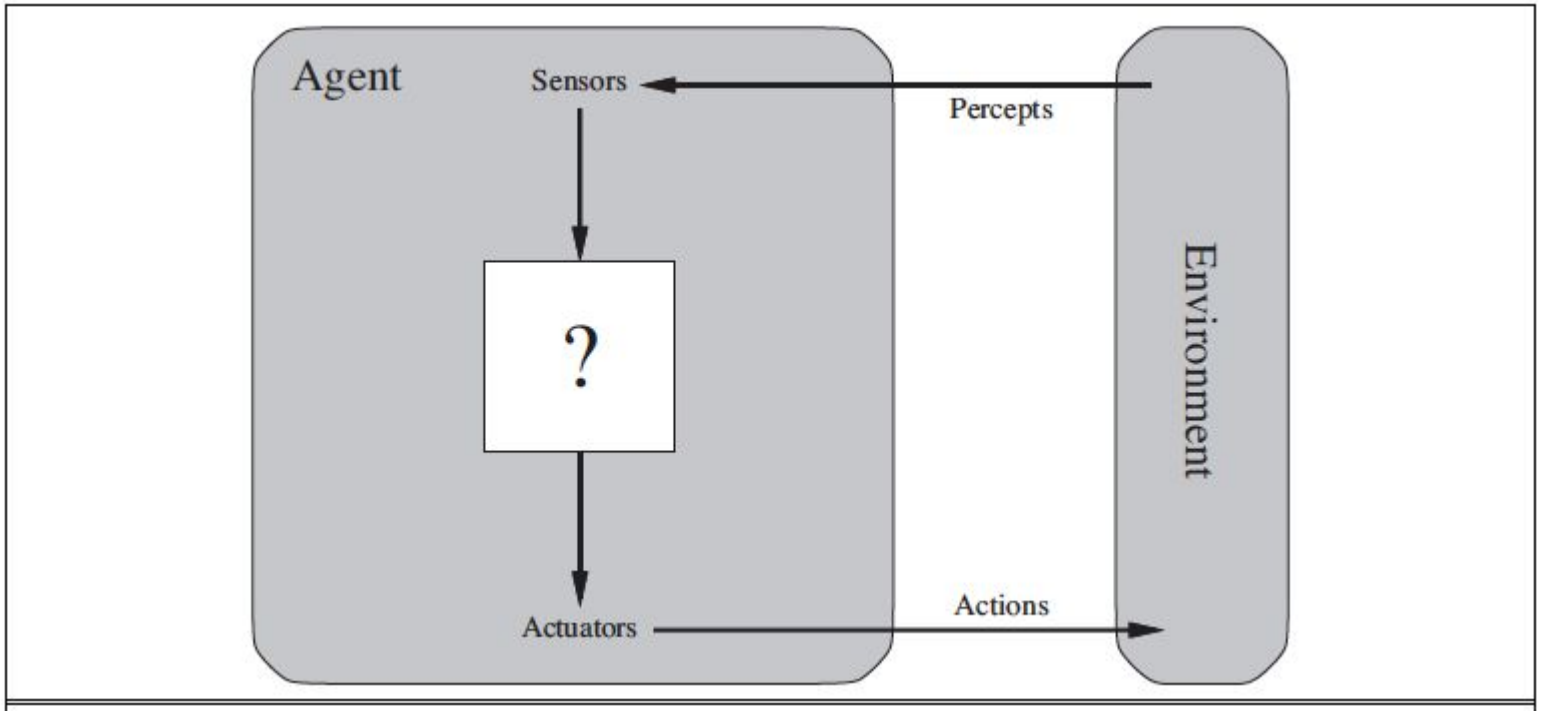
Intelligent Agents

Agent

- Formally “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.”

- We use the term percept to refer to the agent’s perceptual inputs at any given instant.
- An agent’s percept sequence is the complete history of everything the agent has ever perceived.
- “In general, an agent’s choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn’t perceived.”

- An agent’s behavior is described by the agent function that maps any given percept sequence to an action.



Agents interact with environments through sensors and actuators.

PEAS (Performance, Environment, Actuator and Sensor) description

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

PEAS description of the task environment for an automated taxi

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Examples of agent types and their PEAS descriptions.

Rational Agent

A rational agent is one that does the right thing. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing?

Moral philosophy has developed several different notions of the “right thing,” but AI has generally stuck to one notion called consequentialism: we evaluate an agent’s behavior by its consequences.

This notion of desirability is captured by a performance measure that evaluates any given sequence of environment states.

Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a definition of a rational agent:

"For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has."

What is an Intelligent Agent?

An Intelligent Agent (IA) is

- a system that perceives its environment and takes actions to maximize its chances of success.
- These agents can be software-based, hardware-based, or a combination of both,
- and they operate autonomously or semi-autonomously to achieve specific goals.

Characteristics of Intelligent Agents

- **Autonomy:** Agents operate without direct human intervention and have control over their actions and internal state.
  - Self-Driving Cars
- **Perception:** They perceive their environment through sensors or data inputs.
  - Security Surveillance Systems
- **Action:** Agents take actions that affect their environment.
  - Robotic Vacuum Cleaners

Characteristics of Intelligent Agents

- **Rationality:** They act rationally to achieve their goals, which means they make decisions that maximize their expected utility.
  - Automated Trading Systems
- **Learning:** Some agents can improve their performance over time through learning from experiences.
  - Recommendation Systems

Agent Environment

- Definition: The environment in which agents operate.
- Examples:
  - Real-world environments, virtual environments, simulated environments.
- Types of Environments:
  - Fully Observable vs. Partially Observable
  - Deterministic vs. Stochastic
  - Episodic vs. Sequential
  - Static vs. Dynamic
  - Discrete vs. Continuous
  - Single-agent vs. Multi-agent
  - Known vs. Unknown

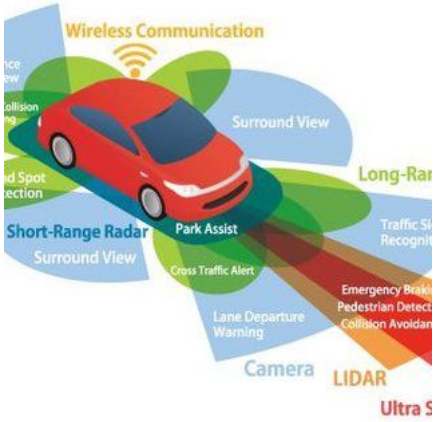
Fully Observable vs. Partially Observable

Fully Observable Environment:

- The agent has access to the complete state of the environment at each point in time.
- There is no hidden information, allowing the agent to make fully informed decisions.

Partially Observable Environment:

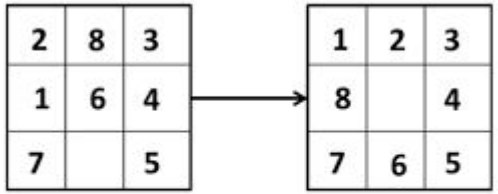
- The agent only has partial information about the state of the environment due to limitations in sensors or inherent characteristics of the environment.



Deterministic vs. Stochastic

Deterministic Environment:

- The next state of the environment is completely determined by the current state and the actions performed by the agent.
- There is no uncertainty in the outcome of actions.



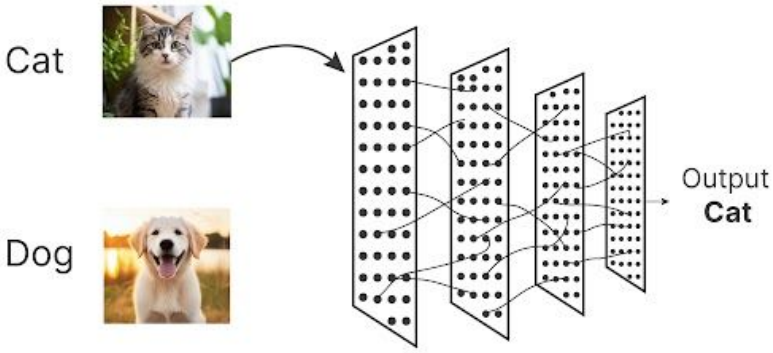
Stochastic Environment:

- The next state of the environment is not fully predictable due to random elements or inherent uncertainty.
- Actions may lead to different outcomes with certain probabilities.

Episodic vs. Sequential

Episodic Environment:

- The agent's actions are divided into discrete episodes.
- Each episode is independent of the others, and the outcome of one episode does not affect the next.



Sequential Environment:

- The current decision could affect all future decisions.
- The agent's actions are part of a continuous sequence, and each action influences subsequent actions.

Static vs. Dynamic

Static Environment:

- The environment remains unchanged while the agent is making a decision.
- The agent does not need to worry about the environment changing in the middle of its decision process.

Dynamic Environment:

- The environment can change while the agent is making a decision, requiring the agent to adapt to changes in real-time.

Discrete vs. Continuous

Discrete Environment:

- The environment consists of a finite number of distinct states and actions.
- The agent's decisions are based on a limited set of possibilities.

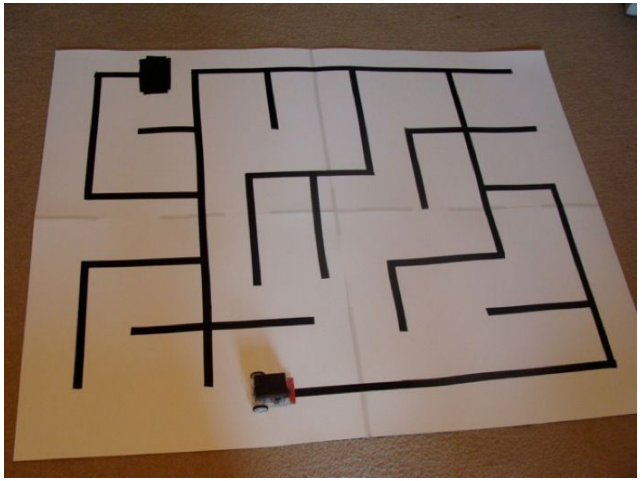
Continuous Environment:

- The environment has a continuous range of states and actions.
- The agent must handle a potentially infinite number of possibilities.

Single-agent vs. Multi-agent

Single-agent Environment:

- Only one agent is operating in the environment,
- and there are no other agents to interact with or compete against.



Multi-agent Environment:

- Multiple agents operate within the environment, interacting with each other.
- These interactions can be competitive or cooperative.

Known vs. Unknown

Known Environment:

- The agent has complete knowledge of the environment’s dynamics, including how actions affect the state of the environment.
- This allows for precise planning and decision-making.

Unknown Environment:

- The agent has no prior knowledge of the environment's dynamics
- and must learn how actions affect states through exploration and interaction.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Examples of task environments and their characteristics

Types of Agents

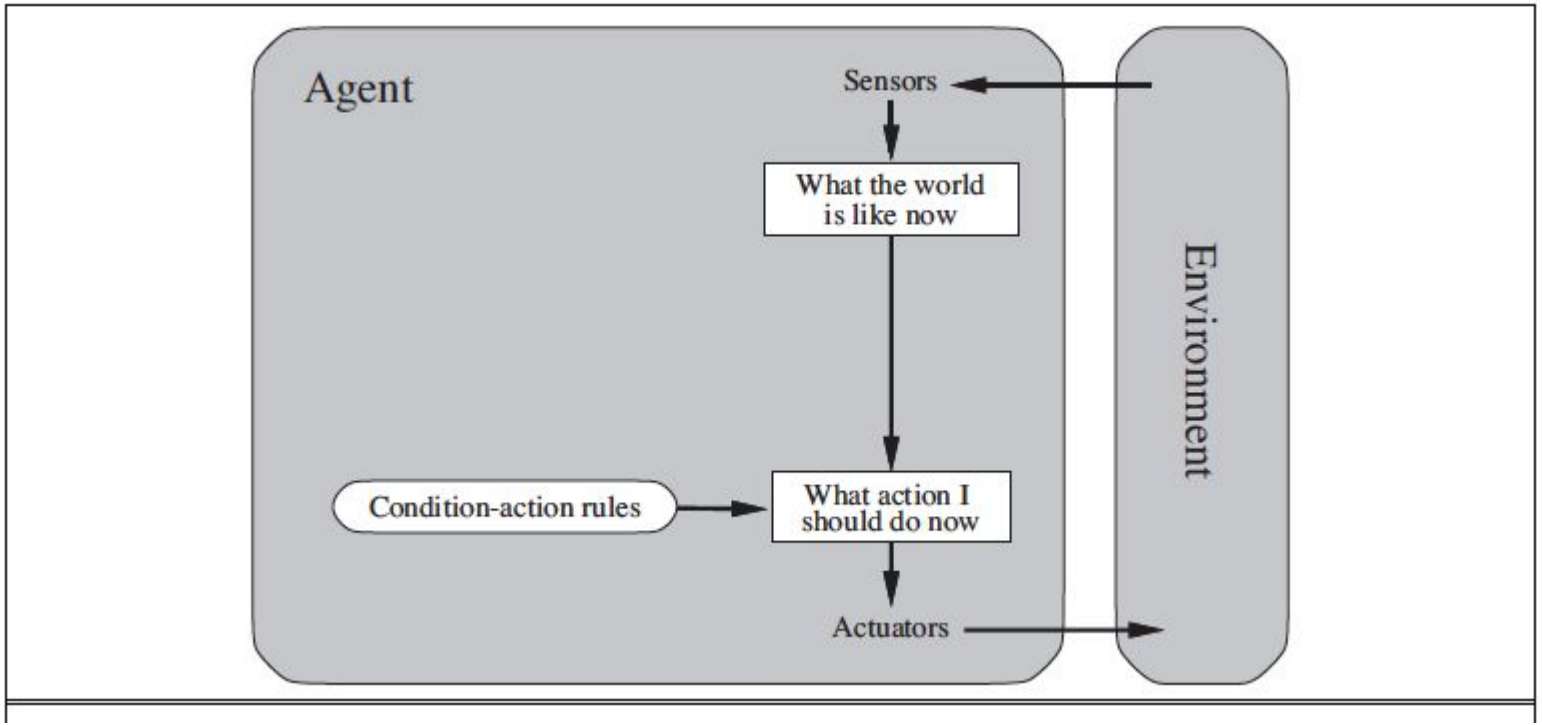
- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents

Simple Reflex Agents (SRA)

Simple Reflex Agents operate by selecting actions based solely on the current percept (i.e., what they sense at the moment), ignoring the rest of the percept history. They follow condition-action rules (if-then statements) to make decisions. They do not consider the history of percepts or the internal state of the environment.

How Simple Reflex Agents Work

- Perception: The agent perceives the environment through its sensors.
- Condition-Action Rules: The agent evaluates the current percept against a set of predefined rules.
- Action: Based on the matching condition, the agent performs the corresponding action.



Schematic diagram of a simple reflex agent.

SRA: Characteristics

1. **Direct Response**:
   - Simple reflex agents respond directly to percepts with actions. They do not maintain any internal state or history of previous percepts.
2. **Condition-Action Rules**:
   - The behavior of these agents is governed by a set of rules, often referred to as "if-then" rules. For example, "if the car in front is braking, then decelerate."
3. **No Memory**:
   - These agents do not have memory; they do not keep track of past percepts or actions. Their decisions are made purely on the basis of the current percept.
4. **Limited Applicability**:
   - Simple reflex agents are effective in environments that are fully observable and predictable. However, they are less effective in complex or partially observable environments where decisions need to consider past events or future consequences.

SRA: Strengths and Limitations

Strengths:

- **Simplicity**: Simple reflex agents are easy to design and implement due to their straightforward nature.
- **Efficiency**: In well-defined, predictable environments, they can operate efficiently without the need for complex processing.

Limitations:

- **Lack of Adaptability**: Without memory or consideration of the history, these agents cannot adapt to changes or learn from past experiences.
- **Limited Scope**: They are not suitable for complex environments where the current percept alone does not provide enough information to make an optimal decision.

SRA Example

Example: A basic thermostat:

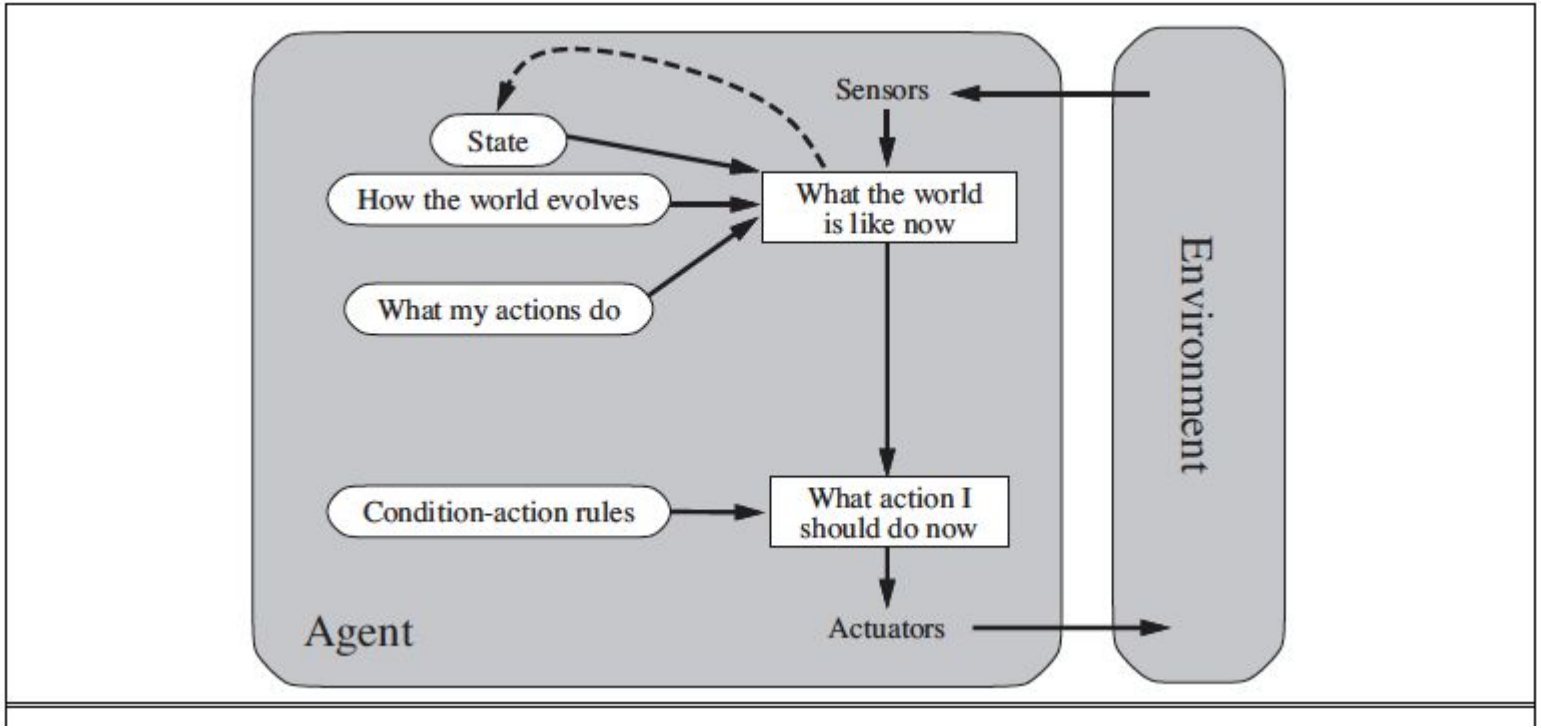
- **Percept**: Current room temperature.
- **Condition-Action Rule**: If the temperature is below 20°C, turn on the heater; if the temperature is above 25°C, turn off the heater.
- **Action**: Turning the heater on or off based on the current temperature.

Model-Based Reflex Agents (MBRA)

Model-Based Reflex Agents maintain an internal state that depends on the percept history and a model of how the world works. This model helps the agent to keep track of parts of the world it can't see at the moment and to make decisions based on a combination of current percepts and stored information.

How Model-Based Reflex Agents Work

1. **Perception**: The agent perceives the environment through its sensors.
2. **Update State**: The agent updates its internal state based on the current percept and its internal model.
3. **Condition-Action Rules**: The agent evaluates its internal state and current percept against a set of predefined rules.
4. **Action**: Based on the matching condition, the agent performs the corresponding action.



Schematic diagram of a model-based reflex agent

MBRA: Characteristics

1. **Internal State**:
   - Model-based reflex agents maintain an internal state that depends on the percept history. This internal state captures relevant aspects of the environment that the agent cannot directly perceive at any given time.
2. **World Model**:
   - These agents use a model of the world that describes how the world evolves in response to actions and how percepts are generated. This model helps the agent update its internal state based on new percepts.
3. **Condition-Action Rules**:
   - Like simple reflex agents, model-based reflex agents use condition-action rules. However, these rules are applied to the internal state rather than directly to the percepts.
4. **State Updating**:
   - The agent updates its internal state using the model of the world whenever it receives a new percept. This process involves incorporating new information into the existing state.

MBRA: Strengths and Limitations

Strengths:

- **Memory**: The ability to maintain an internal state allows these agents to handle more complex and dynamic environments.
- **Increased Applicability**: Model-based reflex agents can function effectively in partially observable environments where not all relevant information is available at once.
- **Improved Decision Making**: By using an internal model, these agents can anticipate the outcomes of their actions better than simple reflex agents.

Limitations:

- **Complexity**: Maintaining and updating an internal state and world model increases the complexity of the agent.
- **Computation**: The need to update the internal state and apply condition-action rules to this state can require more computational resources.

MBRA: Example

Example: A robotic vacuum cleaner:

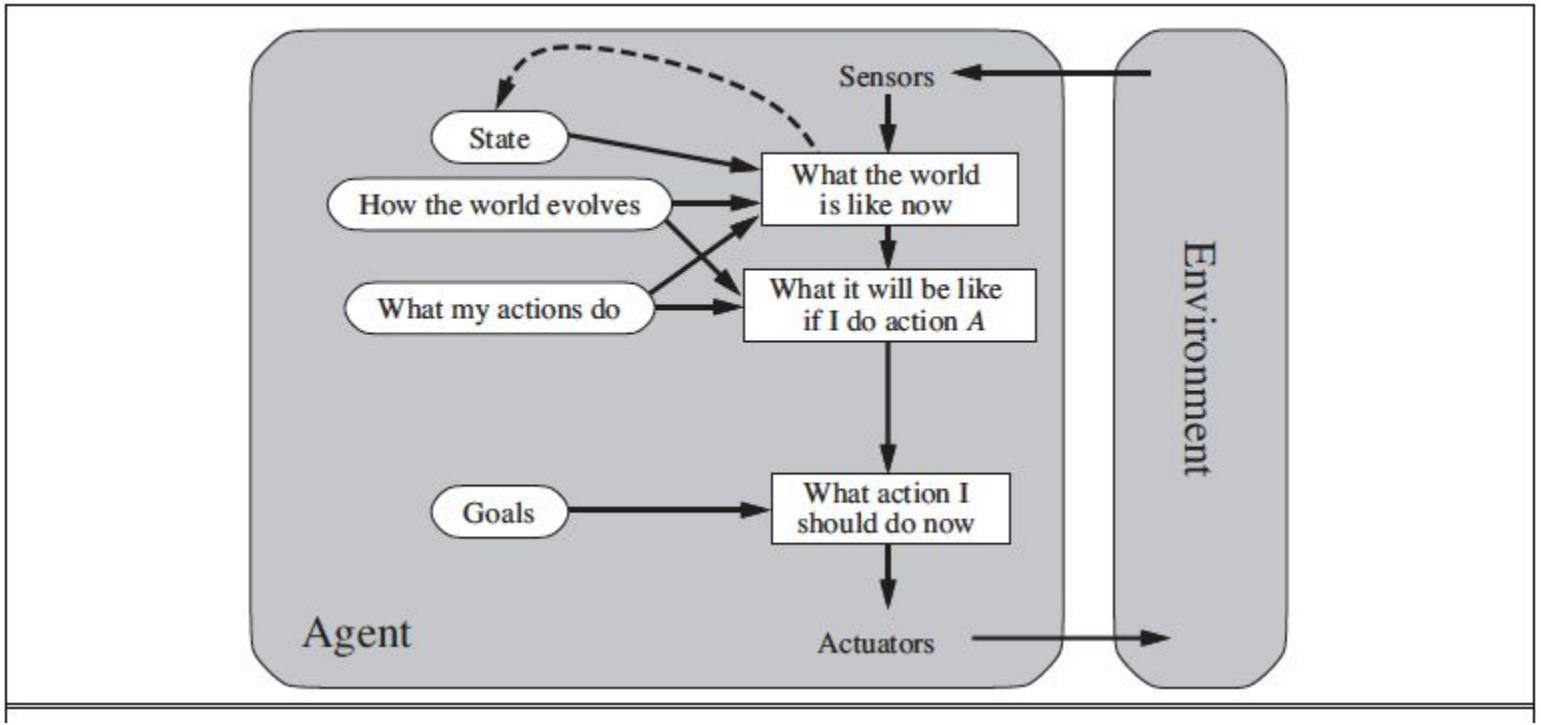
- **Percept**: Current position, obstacles detected.
- **Internal State**: Map of the environment, battery level, cleaning status.
- **Condition-Action Rule**: If the battery level is low and near the charging station, return to the charging station; if an obstacle is detected, change direction.
- **Action**: Moving to the charging station, changing direction, continuing cleaning.

Goal-Based Agents (GBA)

Goal-Based Agents make decisions based on achieving specific goals. These agents evaluate different actions by considering how well each action will help them achieve their goals. They often use planning and search algorithms to find sequences of actions that lead to the desired outcome.

How Goal-Based Agents Work

1. **Perception**: The agent perceives the environment through its sensors.
2. **Goal Identification**: The agent has one or more goals to achieve.
3. **Planning**: The agent evaluates possible actions and sequences of actions to determine which ones will lead to achieving the goal.
4. **Action**: The agent performs the action that is most likely to achieve its goal.



A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

GBA Characteristics

1. **Goals**:
   - Goals are specific desired outcomes or states that the agent aims to achieve. These goals provide a purpose and direction for the agent's actions.
2. **Search and Planning**:
   - Goal-based agents often employ search and planning algorithms to determine a sequence of actions that will lead them from their current state to the goal state. This involves exploring possible future states and evaluating the best path to the goal.
3. **Decision Making**:
   - These agents consider not just the immediate effects of their actions but also their long-term consequences. This forward-thinking approach enables them to handle more complex tasks and environments.
4. **Adaptability**:
   - Goal-based agents can adapt to changes in the environment by re-evaluating their plans and making new decisions based on updated information.

GBA: Strengths and Limitations

Strengths:

- **Effective in Complex Environments**: Goal-based agents can handle complex, dynamic environments where planning and foresight are necessary.
- **Purpose-Driven**: The use of goals provides clear criteria for success, enabling agents to focus their efforts effectively.
- **Flexible and Adaptive**: These agents can re-plan and adapt to changes in the environment, maintaining their focus on achieving their goals.

Limitations:

- **Computational Complexity**: Planning and search algorithms can be computationally intensive, especially in large or complex environments.
- **Requires Clear Goals**: The effectiveness of a goal-based agent depends on well-defined, achievable goals. Ambiguous or conflicting goals can hinder performance.

GBA Example

Example: A GPS Navigation System

- **Perception**: Current location, traffic data, map data.
- **Goal Identification**: Destination address.
- **Planning**: Calculating the optimal route considering current traffic conditions.
- **Action**: Providing turn-by-turn directions to follow the optimal route.

Use Cases:

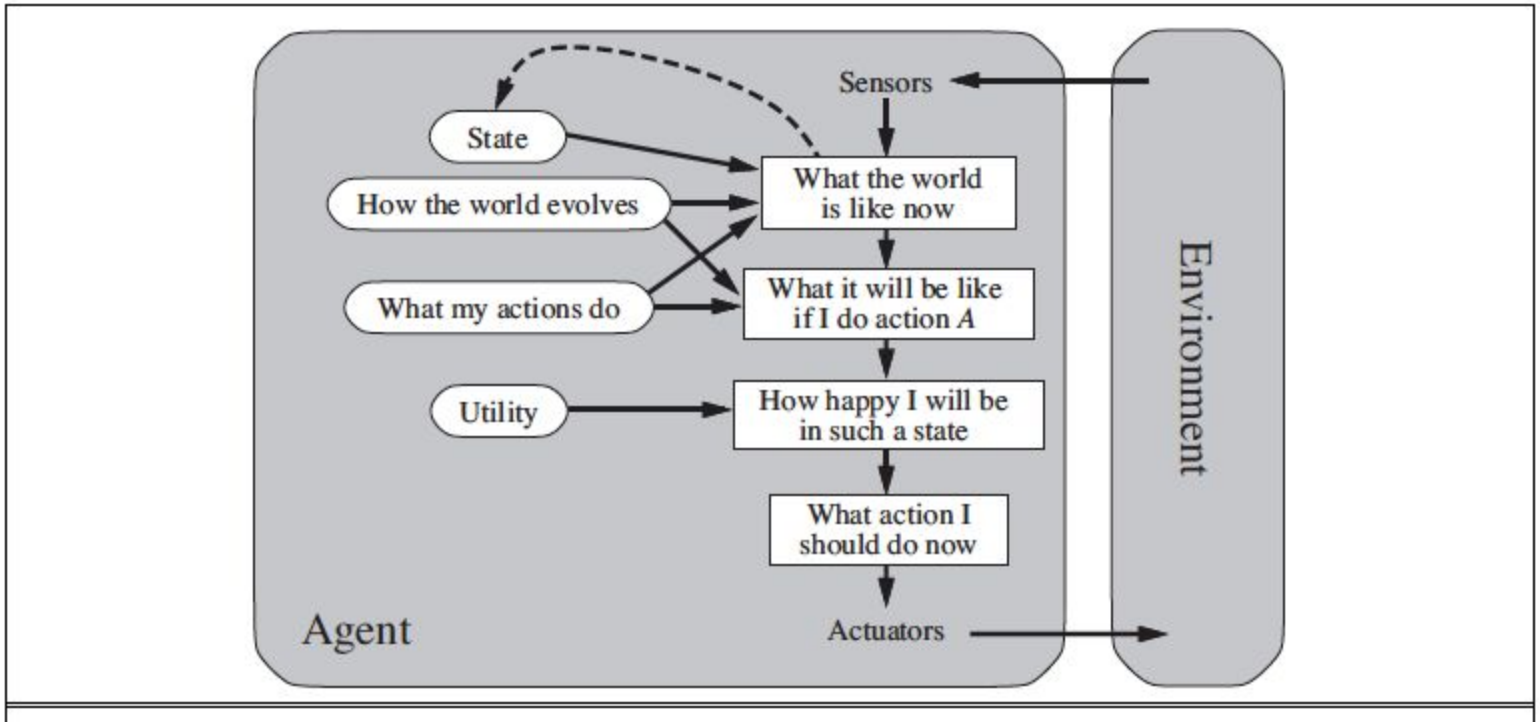
- **Robotics**: A warehouse robot tasked with retrieving an item and delivering it to a specific location.
- **Game AI**: Characters in video games that need to achieve specific objectives, such as capturing a flag or completing a mission.
- **Autonomous Vehicles**: Cars that plan routes to reach destinations efficiently.

Utility-Based Agents (UBA)

Utility-Based Agents extend goal-based agents by considering not only whether goals are achieved but also the desirability (utility) of different states. They use a utility function to evaluate the desirability of different states or outcomes and choose actions that maximize their expected utility.

How Utility-Based Agents Work

1. **Perception**: The agent perceives the environment through its sensors.
2. **Utility Calculation**: The agent uses a utility function to evaluate the desirability of different states or outcomes.
3. **Decision Making**: The agent evaluates possible actions and chooses the one that maximizes expected utility.
4. **Action**: The agent performs the action that maximizes its utility.



A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

UBA: Characteristics

1. **Utility Function**:
   - A utility function is a mathematical representation that assigns a numerical value (utility) to each possible outcome. This value reflects the agent's preference for that outcome.
2. **Decision Making**:
   - Utility-based agents use the utility function to evaluate and compare different outcomes, choosing the action that maximizes the expected utility. This involves considering not only the likelihood of different outcomes but also their desirability.
3. **Handling Trade-offs**:
   - These agents can handle trade-offs between conflicting goals or objectives. For example, an agent might balance the trade-off between speed and safety in a navigation task.
4. **Risk Management**:
   - Utility-based agents can manage uncertainty and risk by considering the expected utility of actions, which accounts for both the probability and the utility of different outcomes.

UBA: Strengths and Limitations

Strengths:

- **Optimal Decision Making**: Utility-based agents aim to make optimal decisions by maximizing the overall utility, leading to better outcomes.
- **Flexibility**: These agents can adapt to different situations and trade-offs by adjusting their utility function.
- **Handling Complex Environments**: Utility-based agents are well-suited for complex, dynamic environments where multiple factors need to be considered.

Limitations:

- **Complexity**: Defining and computing the utility function can be complex, especially in environments with many variables and uncertainties.
- **Subjectivity**: The utility function is subjective and must be carefully designed to reflect the agent's preferences accurately.

UBA: Example

Example: An Automated Trading System

- **Perception**: Market data, historical trends, economic indicators.
- **Utility Calculation**: Evaluating potential trades based on expected return and risk.
- **Decision Making**: Choosing trades that maximize expected utility (profit adjusted for risk).
- **Action**: Executing buy or sell orders.

Use Cases:

- **Healthcare**: Systems that recommend treatments by balancing efficacy, side effects, and patient preferences.
- **Finance**: Investment algorithms that choose portfolios by balancing expected returns and risks.
- **Resource Management**: Systems that allocate resources (like compute power in a data center) to maximize overall efficiency and performance.

Learning Agents

Learning Agents are capable of improving their performance over time by learning from their experiences. They adapt their behavior based on past outcomes, feedback, and interactions with the environment. Learning agents are designed to handle dynamic and complex environments where predefined rules might not be sufficient.

Key Components of Learning Agents

1. **Learning Element**: Responsible for making improvements and acquiring new knowledge or skills based on experience.
2. **Performance Element**: Executes actions based on the knowledge it has acquired. This component is similar to the decision-making part of non-learning agents.
3. **Critic**: Provides feedback to the learning element about the agent's performance. It evaluates the actions taken and helps the learning element to make adjustments.
4. **Problem Generator**: Suggests actions that lead to new experiences and opportunities for learning. It helps the agent explore new possibilities and gather diverse experiences.

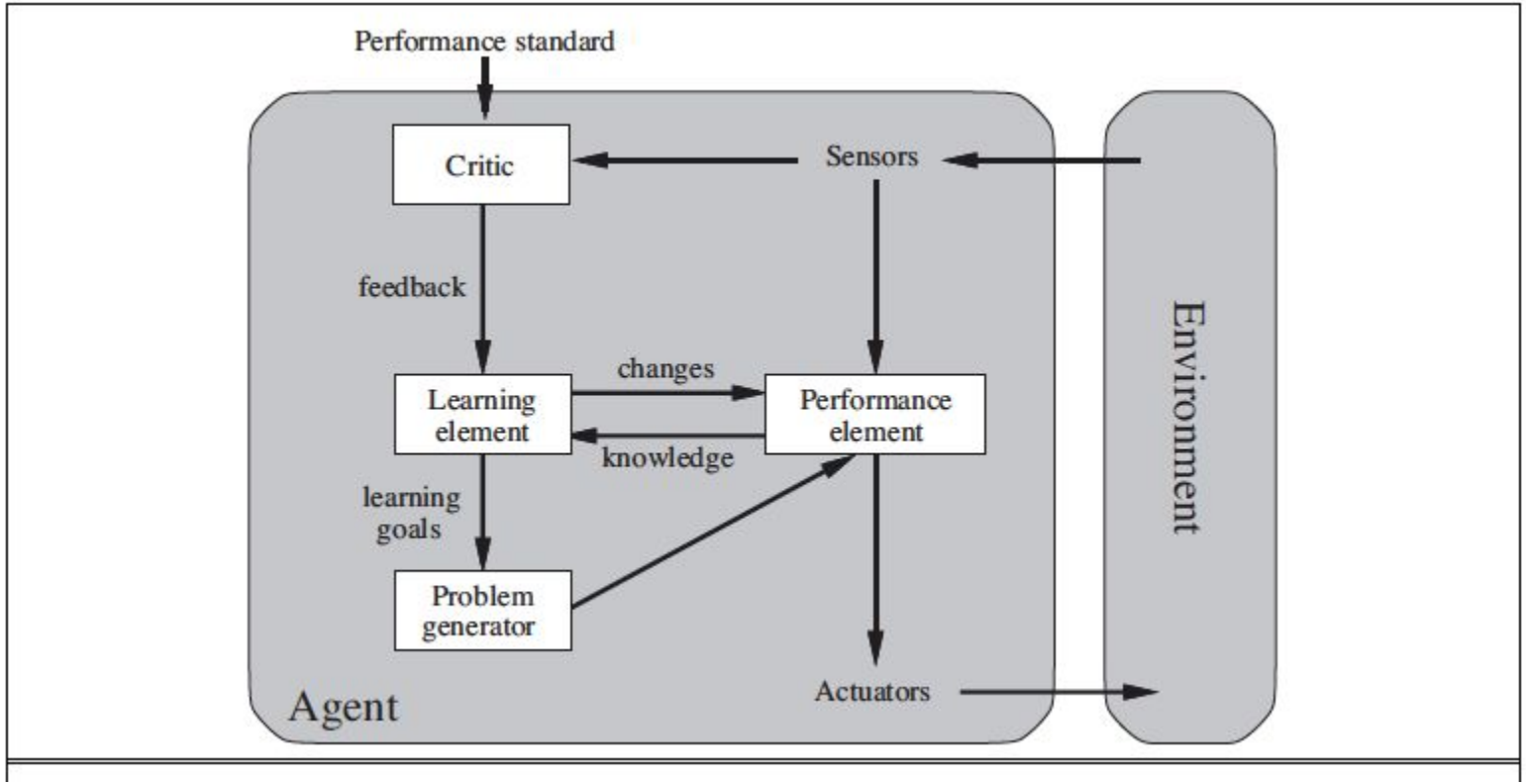
Learning Agents

How Components Contribute to Learning

1. **Learning Element**:
   - **Function**: Updates the agent's knowledge base or policies based on feedback and new data.
   - **Contribution**: Enables the agent to adapt and improve by refining its understanding of the environment and the consequences of its actions.
2. **Performance Element**:
   - **Function**: Uses the current knowledge to make decisions and take actions in the environment.
   - **Contribution**: Acts as the operational part of the agent, implementing the learned strategies and behaviors.

Learning Agents

3. **Critic**:
    - **Function**: Evaluates the outcomes of the agent's actions and provides feedback.
    - **Contribution**: Helps the learning element understand the effectiveness of different actions and identify areas for improvement.
4. **Problem Generator**:
    - **Function**: Encourages exploration by suggesting new actions or strategies that the agent has not tried before.
    - **Contribution**: Promotes learning by exposing the agent to new experiences and challenges, preventing it from getting stuck in suboptimal behaviors.



A general learning agent

Types of Learning in Agents:

Supervised Learning:

- In supervised learning, the agent is provided with labeled training data that includes input-output pairs. The agent learns to map inputs to outputs based on this data, refining its model to improve accuracy.
- Example: A learning agent that recognizes handwritten digits might be trained on a dataset of images labeled with the correct digit.

Unsupervised Learning:

- Unsupervised learning involves training the agent on data without explicit labels. The agent must identify patterns, clusters, or structures within the data on its own.
- Example: A learning agent tasked with customer segmentation might analyze purchase data to identify groups of similar customers without predefined categories.

Types of Learning in Agents:

Reinforcement Learning:

- In reinforcement learning, the agent learns by interacting with its environment and receiving feedback in the form of rewards or punishments. The goal is to learn a policy that maximizes cumulative rewards over time.
- Example: A robot navigating a maze might receive a reward for reaching the goal and penalties for hitting walls, learning to optimize its path to the goal.

Online Learning:

- Online learning refers to the process where the agent learns incrementally as new data arrives, updating its model continuously rather than relying on a fixed dataset.
- Example: A recommendation system that adapts in real-time based on users' interactions with recommended content.

Examples of Learning Agents

1. Supervised Learning Agents

Example: Spam Email Classifier

- **Description**: A spam email classifier learns to differentiate between spam and non-spam emails based on labeled training data.
- **Components**:
  - **Learning Element**: Uses labeled examples to update its classification model.
  - **Performance Element**: Classifies incoming emails based on the learned model.
  - **Critic**: Evaluates the classifier's accuracy using feedback from users (e.g., marking emails as spam or not spam).
  - **Problem Generator**: Incorporates new types of emails into the training set to improve classification.

Examples of Learning Agents

2. Unsupervised Learning Agents

Example: Customer Segmentation

- **Description**: A marketing agent that segments customers into different groups based on purchasing behavior without labeled data.
- **Components**:
  - **Learning Element**: Uses clustering algorithms to group customers based on similarities in data.
  - **Performance Element**: Applies the learned segments to target marketing campaigns.
  - **Critic**: Analyzes the success of marketing campaigns to refine the segmentation.
  - **Problem Generator**: Identifies new customer attributes to consider for segmentation.

Examples of Learning Agents

3. Reinforcement Learning Agents

Example: AlphaGo

- **Description**: AlphaGo, developed by DeepMind, uses reinforcement learning to play the game of Go. It learns optimal strategies by playing numerous games against itself and other opponents, constantly improving based on the outcomes.
- **Components**:
  - **Learning Element**: Updates its policy based on the results of games played.
  - **Performance Element**: Executes moves during gameplay.
  - **Critic**: Evaluates the outcome of each game to determine the success of different strategies.
  - **Problem Generator**: Explores new strategies and move sequences to enhance learning.

Challenges and Considerations

Exploration vs. Exploitation: Learning agents must balance exploration (trying new actions to gather more information) and exploitation (using known information to maximize rewards). This balance is critical to avoid local optima and ensure robust learning.

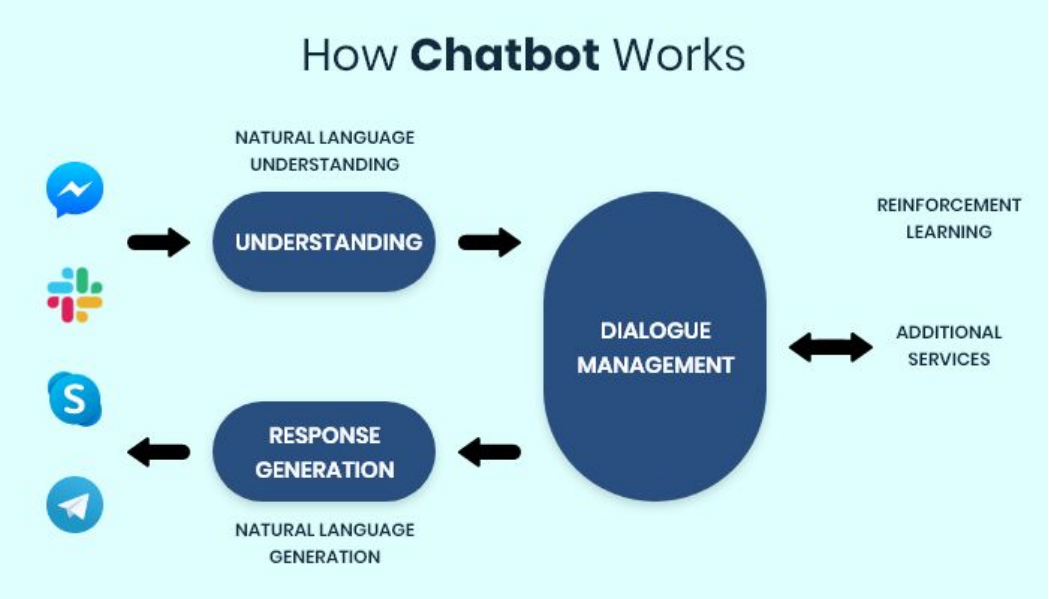
Scalability: As the complexity of the environment increases, the agent must handle larger state and action spaces, which can pose computational challenges.

Transfer Learning: Learning agents may need to transfer knowledge from one domain or task to another, requiring them to generalize and apply what they've learned in new contexts.

Ethical Considerations: In some applications, learning agents must be designed with ethical considerations in mind, especially when their decisions impact human well-being, such as in healthcare or autonomous vehicles.

Case Study: Virtual Personal Assistants

- **Learning Element**: The learning element in a chatbot is responsible for improving the chatbot's ability to understand and generate responses.
- **Performance Element**: The performance element involves the actual operation of the chatbot during interactions with users. It selects responses based on the current state of the conversation
- **Critic**: The critic evaluates how well the chatbot performs in its interactions. (User feedback, sentiment analysis)
- **Problem Generator**: The problem generator in a chatbot might involve testing new types of interactions or exploring different conversation paths to expand the chatbot capabilities.



Other Examples of Intelligent Agents

- **Smart Home Systems**: Systems like Nest smart thermostats and Philips Hue smart lighting adapt to user preferences and environmental conditions. They learn from user behavior to optimize energy usage and provide comfort.
- **Robotics**: Robots in manufacturing, healthcare, and service industries perform tasks autonomously or semi-autonomously. For instance, robotic arms in factories assemble products, and robotic surgery systems assist surgeons in performing precise operations.
- **Game AI**: Non-player characters (NPCs) in video games exhibit intelligent behavior, such as planning, learning, and adapting to player actions. Examples include AI opponents in strategy games like StarCraft or role-playing games like Skyrim.
- **E-commerce Recommendations**: Online retailers like Amazon use intelligent agents to recommend products to users based on their browsing and purchase history. These agents analyze large datasets to identify patterns and suggest items that users are likely to be interested in.

How the components of agent programs work

In agent programs, representations of the environment and internal state can be categorized into atomic, factored, and structured representations. Each type of representation has different implications for how an agent perceives, reasons about, and interacts with the world.

The representations are:

- Atomic
- Factored
- Structured

Atomic Representation

In atomic representations, each state of the environment is treated as a unique, indivisible entity without any internal structure.

Components:

1. **State Representation**:
   - Each state is a distinct and unique identifier.
   - No internal features or attributes are used to describe the state.
2. **Perception**:
   - The agent perceives the environment as a set of distinct states.
   - Each perception corresponds to a specific state.
3. **Decision Making**:
   - The decision-making process involves mapping each state directly to an action.
   - Simple lookup tables or state-action pairs are commonly used.
4. **Action**:
   - Actions are selected based on the current state as perceived.
   - No internal structure within states means actions are typically predefined for each state.

Example:

- **Chess**: Each board configuration is a unique state without any further breakdown of individual pieces or positions.

Factored Representation

In factored representations, states are described by a set of variables (features), each of which can take on different values. This allows for a more detailed and flexible representation of the environment.

Components:

1. **State Representation**:
   - A state is represented as a vector of variables (features).
   - Each variable can take on different values, providing a structured description of the state.
2. **Perception**:
   - The agent perceives the environment by assigning values to the relevant variables.
   - Sensor data is mapped to these variables to create a factored state description.
3. **Decision Making**:
   - Decision-making involves evaluating the values of different variables to determine the best action.
   - Techniques such as decision trees, Bayesian networks, or linear programming can be used.
4. **Action**:
   - Actions are selected based on the values of the state variables.
   - Policies or rules are defined in terms of these variables.

Example:

- **Self-Driving Car**: The state might include variables like the car's position, speed, nearby obstacles, and traffic signals. Each variable can take on different values, such as specific locations, velocities, or statuses of traffic lights.

Structured Representation

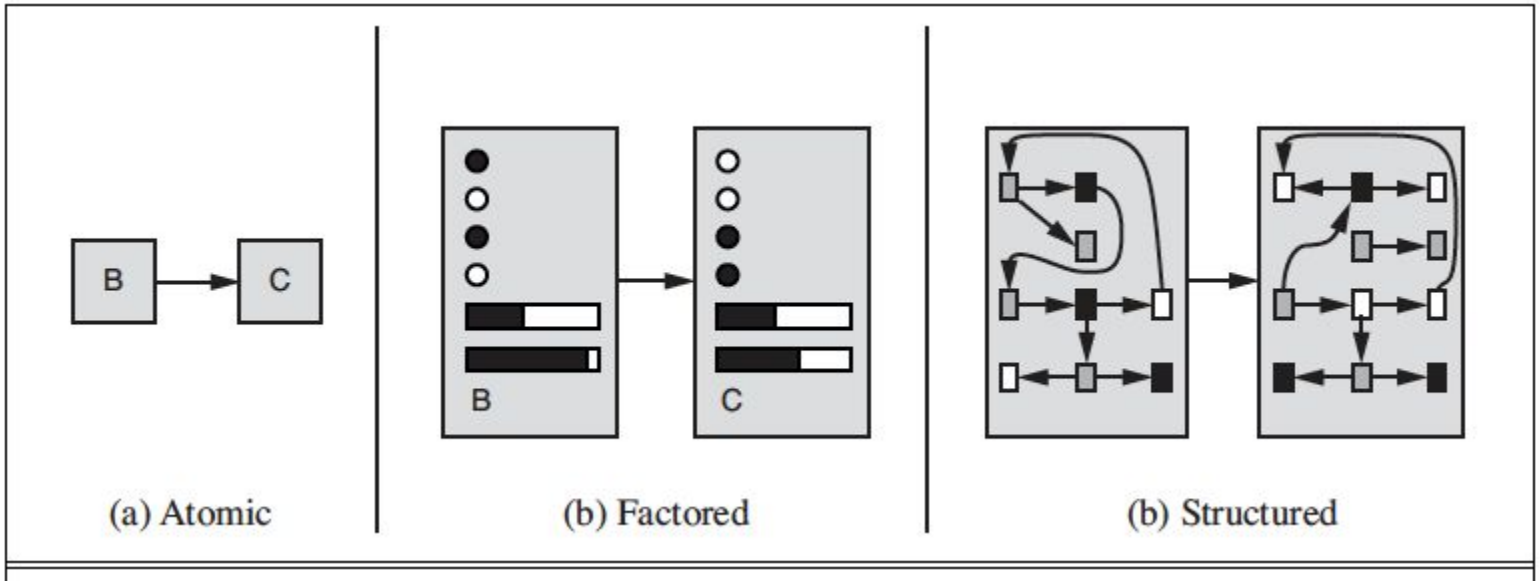
Structured representations go a step further by incorporating relationships between objects and attributes within the state. This allows for complex and rich descriptions of the environment.

Components:

1. **State Representation**:
    - States are represented as a collection of objects, each with its own attributes and relationships to other objects.
    - This is often modeled using graphs, relational databases, or logical representations.
2. **Perception**:
    - The agent perceives the environment by identifying objects, their attributes, and the relationships between them.
    - Advanced perception systems, like those using computer vision or natural language processing, are often required.
3. **Decision Making**:
    - Decision-making involves reasoning about the objects and their interrelationships.
    - Techniques such as knowledge graphs, ontologies, and first-order logic can be used to infer actions.
4. **Action**:
    - Actions are selected based on the detailed structure of the state.
    - Complex planning algorithms like STRIPS or PDDL might be used to generate action sequences.

Example:

- **NLP:** A sentence might be represented using a parse tree, where words are nodes connected by syntactic relationships.



Three ways to represent states and the transitions between them.

Problem Solving by Search

Solving Problems by Searching

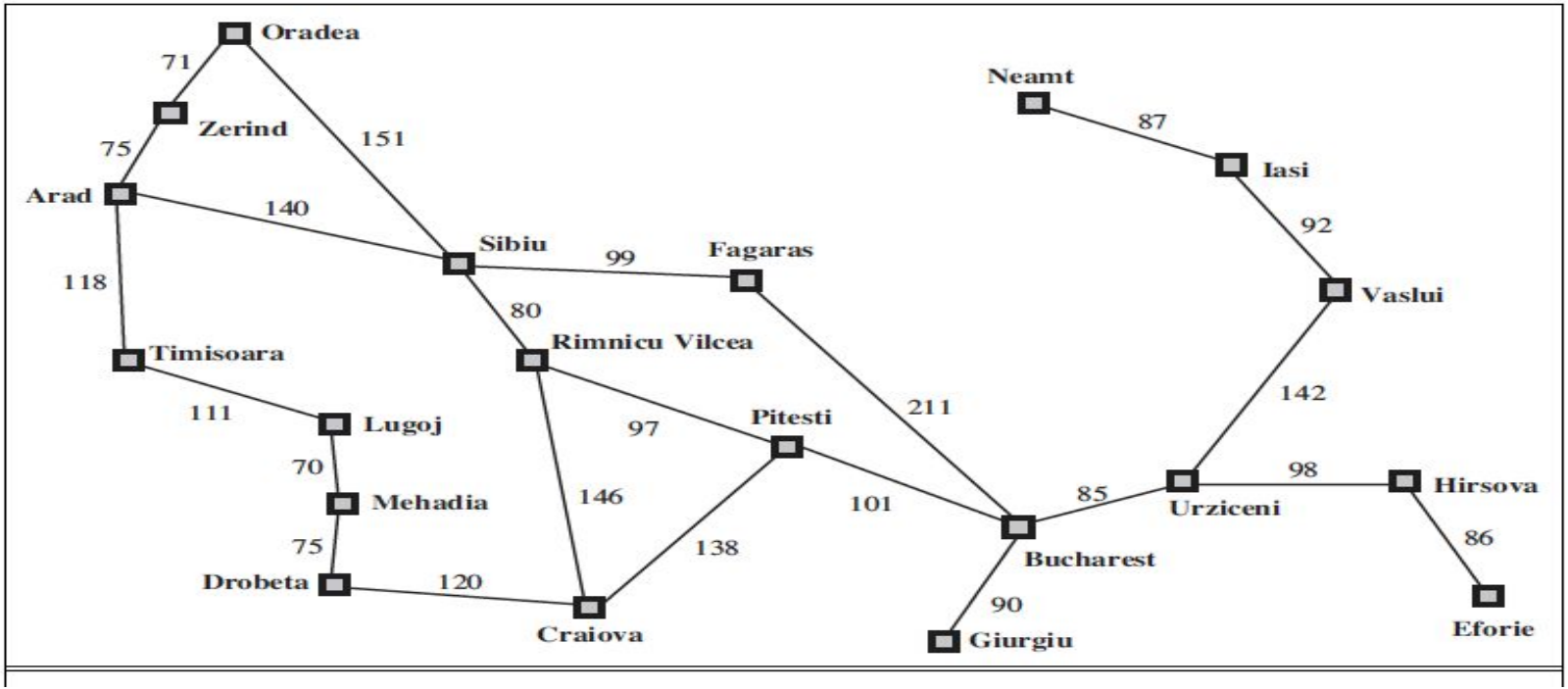
- When the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state.
- Such an agent is called a problem-solving agent, and the computational process it undertakes is called search.
- Problem-solving agents use atomic representations, that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms.
- Agents that use factored or structured representations of states are called planning agents.

An Example

- Imagine an agent enjoying a touring vacation in Romania.
- The agent's performance measure contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife (such as it is), avoid hangovers, and so on.
- The decision problem is a complex one involving many tradeoffs and careful reading of guidebooks.
- Now suppose the agent is currently in the city of Arad and has a non refundable ticket to fly out of Bucharest the following day.
- In that case, it makes sense for the agent to adopt the goal of getting to Bucharest.
- The agent observes street signs and sees that there are three roads leading out of Arad: one to Sibiu, one to Timișoara, and one to Zerind.

An Example

- None of these are the goal, so unless the agent is familiar with the geography of Romania it will not know which road to follow.
- If the agent has no additional information, i.e., if the environment is unknown then the agent can do no better than to execute one of the actions at random.
- In this class, we will assume our agents always have access to information about the world, such as the map.
- With that information, the agent can follow the following four-phase problem solving process.



A simplified road map of part of Romania

Four-Phase Problem-Solving Process

- Goal Formulation:
  - The agent adopts the goal of getting to Bucharest.
  - Courses of action that don't reach Bucharest on time can be rejected without further consideration
  - and the agent's decision problem is greatly simplified.
  - Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
  - Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

Four-Phase Problem-Solving Process

- Problem Formulation:
  - We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied.
  - The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.
  - Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider.
  - If it were to consider actions at the level of "move the left foot forward an inch" or "turn the steering wheel one degree left," the agent would probably never find its way out of the parking lot, let alone to Bucharest,
  - because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution.
  - Problem formulation is the process of deciding what actions and states to consider, given a goal.

Four-Phase Problem-Solving Process

- Search:
  - Before taking any action in the real world, the agent simulate sequences of actions in its model, searching until it find the sequence of actions that reaches the goal.
  - Such a sequence is called a solution.
  - The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution or it will find that no solution is possible.
  - The process of looking for a sequence of actions that reaches the goal is called search.
  - A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- Execution:
  - Once a solution is found, the actions it recommends can be carried out.
  - This is called the execution phase.
- Thus, we have a simple “formulate, search, execute” design for the agent.

- It is an important property that in a fully observable, deterministic, known environment *the solution to any problem is a fixed sequence of actions*: i.e, drive to Sibiu, then Fagaras, then Bucharest.
- If the model is correct then once the agent has found the solution it can ignore its percepts while it is executing the action— closing it's eyes, so to speak— because the solution is guaranteed to lead to the goal.
- Control theorist call this an open-loop system: ignoring the percepts break the loop between agent and environment.
- If there is a chance that the model is incorrect or the environment is non deterministic then the agent would be safer using a closed-loop approach that monitors the percepts.

Summary

1. Goal Formulation

Objective: Define what you want to achieve.

2. Problem Formulation

Objective: Translate the goal into a well-defined problem.

3. Search

Objective: Explore the sequence of actions to find a solution path from the initial point to the goal.

4. Execution

Objective: Execute the solution path to achieve the goal.

Distinction Between Well-Defined and Ill-Defined Problems

1. **Clarity and Completeness of Problem Specification:**
   - **Well-Defined:** All elements of the problem (initial state, goal state, operators, constraints) are clearly and completely specified.
   - **Ill-Defined:** One or more elements of the problem are vague, incomplete, or subjective.
2. **Objective vs. Subjective Criteria:**
   - **Well-Defined:** Success criteria are objective and measurable.
   - **Ill-Defined:** Success criteria may be subjective and open to interpretation.
3. **Boundedness of State Space:**
   - **Well-Defined:** The state space is finite and well-delineated.
   - **Ill-Defined:** The state space may be infinite or not clearly defined.

Distinction Between Well-Defined and Ill-Defined Problems

Example Distinction:

- **Well-Defined Problem:** Solving a Sudoku puzzle.
  - **Initial State:** The starting grid with some numbers filled in.
  - **Goal State:** A completed grid that follows the rules of Sudoku.
  - **Operators:** Filling in the blank cells with numbers 1-9 following Sudoku rules.
  - **Constraints:** Each number 1-9 must appear exactly once in each row, column, and 3x3 subgrid.
  - **Criteria for Success:** A fully completed and correct Sudoku grid.
- **Ill-Defined Problem:** Designing a marketing campaign.
  - **Initial State:** General knowledge about the product and target audience.
  - **Goal State:** Increased brand awareness and sales (vague and subjective).
  - **Operators:** Various marketing strategies and tactics (not clearly defined).
  - **Constraints:** Budget limits, market trends (may be ambiguous).
  - **Criteria for Success:** Subjective measures like brand perception, customer engagement, and sales growth.

Notion of State

States are used to represent different possible situations or steps in the journey toward finding a solution to a problem.

Key Concepts:

1. **State Representation:**
   - A state is a snapshot of all relevant information at a particular point in the problem-solving process. For example, in a chess game, a state would include the positions of all the pieces on the board.
2. **Initial State:**
   - This is the starting point of the search. It's the state from which the search begins. In a maze-solving problem, the initial state would be the starting position of the agent in the maze.
3. **Goal State:**
   - The goal state is the condition or configuration that the problem-solving process aims to reach. For instance, in a puzzle, the goal state is the final solved configuration.

State Space

Definition of State Space:

- The state space of a problem is the complete set of all possible states that can be generated from the initial state by applying a sequence of operations or actions. Each state in the state space represents a unique configuration of the system at a specific point in the problem-solving process.
- For example, in a chess game, each possible arrangement of pieces on the board corresponds to a different state. The state space would thus include every conceivable board configuration that could arise during the game.

2. State Space as a Graph:

- The state space can be visualized as a graph where:
  - **Nodes (Vertices):** Represent individual states.
  - **Edges:** Represent transitions between states, which occur as a result of applying an action or operator.
- This graph may be finite or infinite depending on the problem. For example, in a game with a limited number of moves, the state space is finite. However, in problems like robot navigation in an unbounded environment, the state space could potentially be infinite.

Path in the State Space:

- A path in the state space is a sequence of states connected by transitions (edges). The search process is essentially about finding a path from the initial state to one of the goal states.
- Example: In a maze-solving problem, a path is the sequence of steps taken to move from the entrance to the exit of the maze.

State Space Size:

- The size of the state space can vary significantly depending on the problem:
  - **Finite State Space:** Problems like the 8-puzzle or chess have a finite number of states. In the 8-puzzle, there are 9! = 362,880 possible states.
  - **Infinite State Space:** Problems like mathematical optimization or robot path planning in an open field might have an infinite state space.

State Space Complexity:

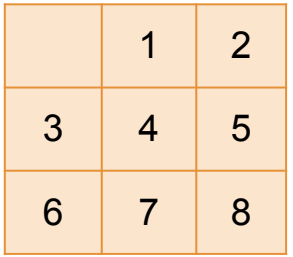
- The complexity of a search problem is often measured by the size of its state space and the branching factor (the average number of children per node). Problems with large state spaces or high branching factors are generally more difficult to solve.

Example: The 8 Puzzle

- States? Locations of tiles
- Actions? Move blank left, right, up, down
- Goal test? = goal state (given)
- Path cost? 1 per move

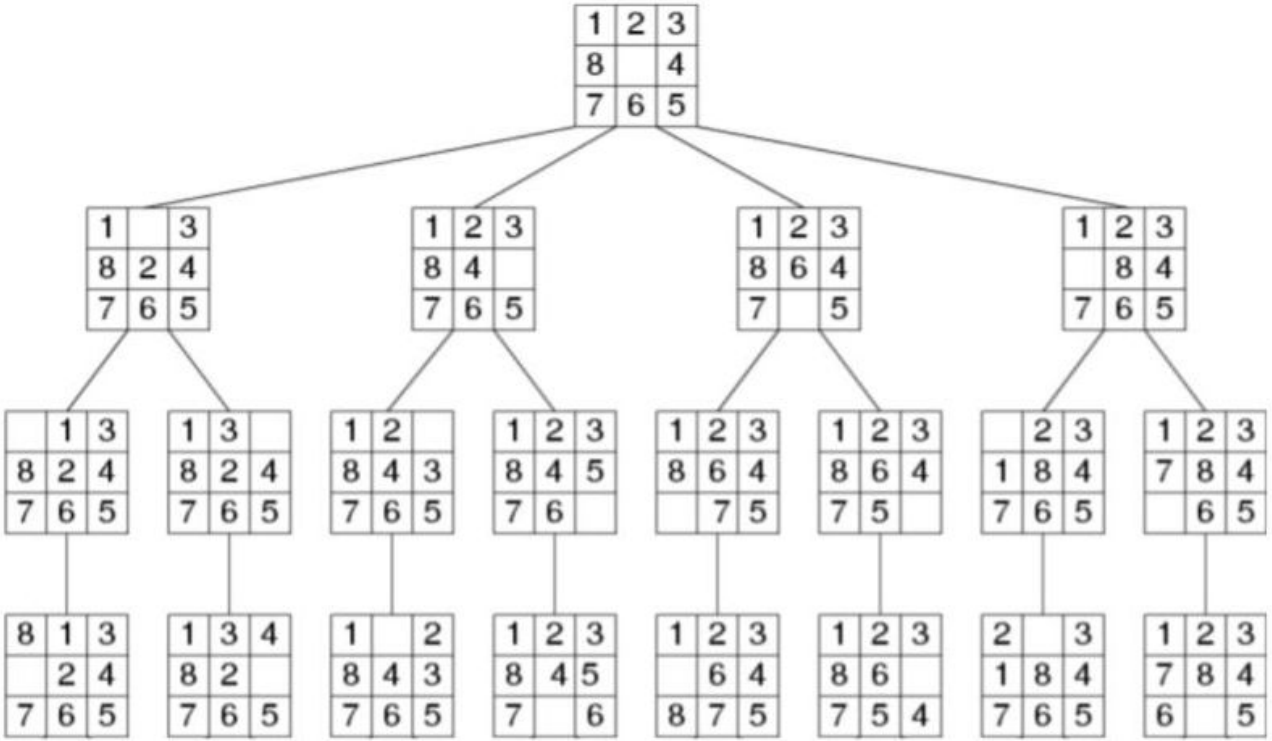
- [Note: optimal solution of n-Puzzle family is NP-hard]

Start State



Goal State

State Space of 8 Puzzle Problem



More Problem: Missionaries and Cannibals

Description: Three missionaries and three cannibals must cross a river using a boat that can carry at most two people. At no point should cannibals outnumber missionaries on either side of the river.

State Space: Each state is defined by the number of missionaries and cannibals on each side of the river and the boat's location.

State Representation:

Each state can be represented as a tuple `(M1, C1, B, M2, C2)`, where:

- `M1` = Number of missionaries on the starting side.
- `C1` = Number of cannibals on the starting side.
- `B` = Location of the boat (1 for the starting side, 0 for the other side).
- `M2` = Number of missionaries on the other side.
- `C2` = Number of cannibals on the other side.

The initial state is `(3, 3, 1, 0, 0)`, and the goal state is `(0, 0, 0, 3, 3)`.

Rules:

1. The boat can carry 1 or 2 people.
2. At no point should the number of cannibals exceed the number of missionaries on either side (if there are missionaries present).

State Transitions:

The boat can move between the two sides, transferring missionaries and cannibals according to the following rules:

- `(1M, 0C)` or `(0M, 1C)` or `(1M, 1C)` or `(2M, 0C)` or `(0M, 2C)`

Search Problem

A search problem can be defined formally as follows:

- A set of possible states that the environment can be in. We call this the state space.
- The initial state that the agent starts in.
- A set of one or more goal states. Sometimes there is one goal state (e.g., Bucharest), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number).
    - We can account for all three of these possibilities by specifying an *Is-GOAL* method for a problem.
- The actions available to the agent. Given a state S, Action(S) returns a finite set of actions that can be executed in S. We say that each of these actions is applicable in S.
    - For example, Action(Arad) = {ToSibiu, ToTimisoara, ToZerind}

Search Problem

- A transition model, which describes what each action does. Result(S,A) returns the state that results from doing action A in state S.
  - For example, Result(Arad, ToZerind) = Zerind
- An action cost function, denote by Action-Cost(S,A,S') that gives the numeric cost of applying action A in state S to reach state S'.
- A sequence of actions forms a path, and a solution is a path from the initial state to a goal state.
  - We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs.
- An optimal solution has the lowest path cost among all solutions.

Abstraction in Problem Solving

Abstraction is a fundamental concept in computer science and artificial intelligence. It involves simplifying a problem by reducing the amount of detail in the representation, focusing on the essential aspects that are necessary to solve the problem. This process allows us to manage complexity more effectively and can make problem-solving more efficient.

Key Aspects of Abstraction:

1. **Removing Detail:**
   - **Definition:** Abstraction involves stripping away unnecessary details from a problem to create a simplified model that retains the core components relevant to solving the problem.
   - **Example:** In a navigation problem, instead of considering every street and building, we might abstract the city as a grid where intersections are nodes and streets are edges.
2. **Validity of Abstraction:**
   - **Definition:** An abstraction is valid if any solution derived in the abstract model can be translated back into a valid solution in the original, more detailed problem space.
   - **Example:** If we solve the navigation problem using our simplified grid model, we must be able to convert the path found in this model back into actual streets and turns in the real city.

Abstraction in Problem Solving

- **Significance:** Validity ensures that the abstraction does not omit crucial details that would render the abstract solution unusable in the real world.

2. **Usefulness of Abstraction:**
   - **Definition:** An abstraction is useful if solving the problem in the abstract model is easier or more efficient than solving it in the original detailed model.
   - **Example:** Solving a navigation problem on a simplified grid is typically faster and computationally less expensive than accounting for every street, traffic light, and pedestrian in the real city.
   - **Significance:** Usefulness ensures that the abstraction provides practical benefits, such as reduced computational resources or time savings, making it a valuable tool in problem-solving.

Importance of Abstraction in Problem Solving

Managing Complexity:

- By focusing on essential details and ignoring irrelevant ones, abstraction helps manage the complexity of a problem. This makes it easier to understand, analyze, and solve.
- Example: In software development, we might abstract the concept of a "user" to focus only on relevant attributes like username and password, ignoring other details such as the user's physical address or phone number.

Improving Efficiency:

- Simplifying a problem often leads to more efficient algorithms. Abstract models typically require fewer computational resources, which can lead to faster solutions.
- Example: In pathfinding algorithms, abstracting a large and detailed map into a simpler graph reduces the number of nodes and edges, speeding up the search process.

Importance of Abstraction in Problem Solving

Facilitating Reusability:

- Abstract solutions and models can often be reused across different problems with similar structures. This can lead to more generalized solutions that apply to a broader range of issues.
- Example: The A* algorithm is a general-purpose pathfinding algorithm that works on various abstract graph models, from game maps to network routing.

Enhancing Focus:

- Abstraction allows problem solvers to concentrate on high-level strategies without getting bogged down by low-level details. This focus can lead to better strategic thinking and innovation.
- Example: In strategic planning for businesses, abstracting the market environment helps leaders focus on major trends and forces, rather than getting lost in minor fluctuations.

Example Problems

A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.

A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.

- **Vacuum world**
- **Sokoban puzzle**
- **Sliding-tile puzzle**

Example Problems

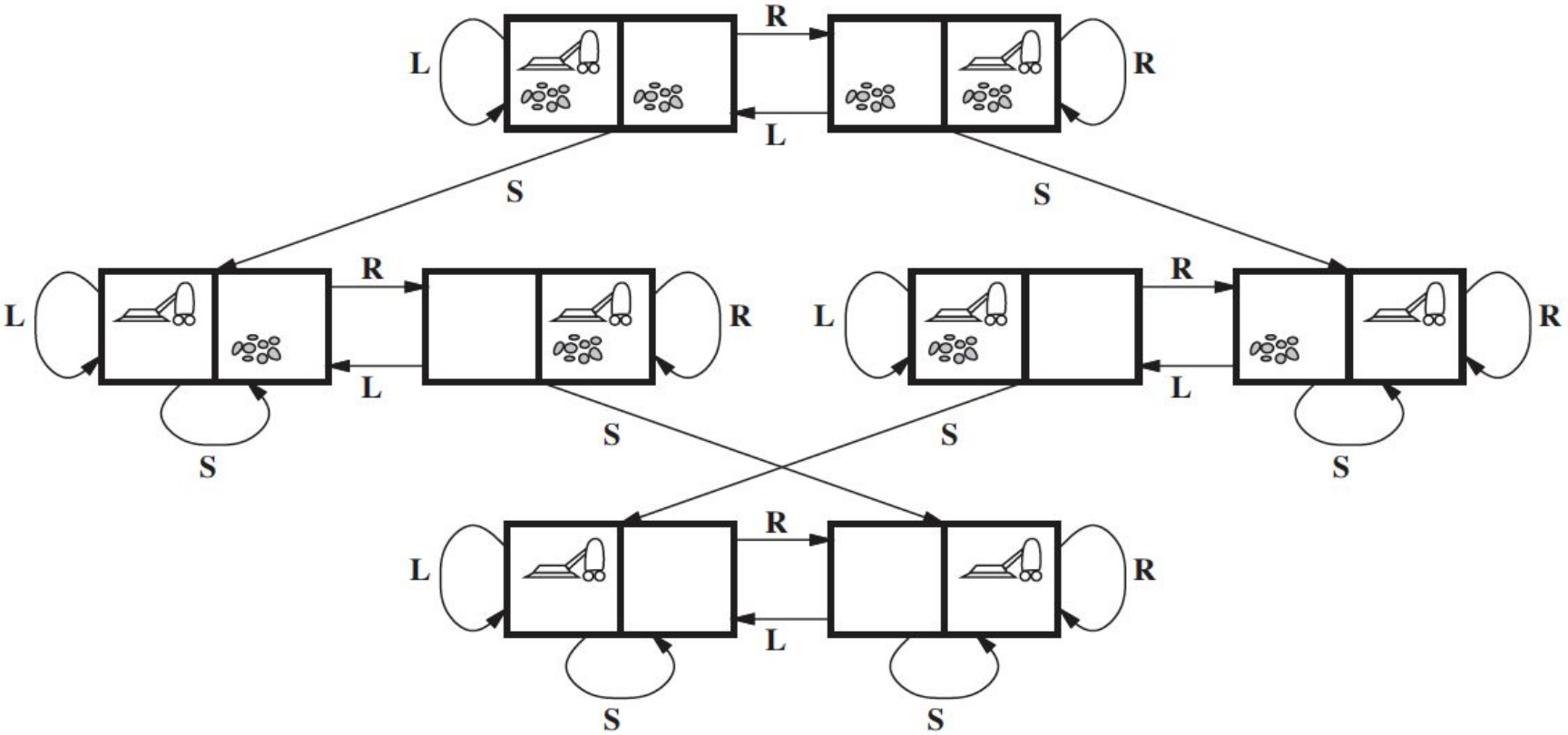
A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

- **Route-finding problem**
- **Touring problems**
- **Traveling salesperson problem (TSP)**
- **VLSI layout problem**
- **Robot navigation**
- **Automatic assembly sequencing**

Vacuum World

States:

- **Definition:** The state is determined by the agent's (vacuum cleaner's) location and the status of the dirt in each location.
- **Representation:** In a simple environment with 2 locations (left and right), the agent can be in either location, and each location can be clean or dirty.
- **Possible States Calculation:** Since the agent can be in one of 2 locations and each location can be either clean or dirty, we have $2 \times 2^2 = 8$ possible states.
    - Example States:
        1. Agent in the left location, both locations clean.
        2. Agent in the left location, left location dirty, right location clean.
        3. Agent in the left location, left location clean, right location dirty.
        4. Agent in the left location, both locations dirty.
        5. Agent in the right location, both locations clean.
        6. Agent in the right location, left location dirty, right location clean.
        7. Agent in the right location, left location clean, right location dirty.
        8. Agent in the right location, both locations dirty.


L
R
L
R
S
S
L
R
L
R
S
L
R
L
R
S
S
S
L
R
L
R
S
S

Vacuum World

Initial State:

- **Definition:** The initial state is the starting configuration of the environment.
- **Flexibility:** Any of the possible states can be designated as the initial state.
    - Example: The agent starts in the left location with both locations dirty.

Actions:

- **Available Actions:** In this simple environment, the agent has three possible actions:
    - **Left:** Move to the adjacent left location.
    - **Right:** Move to the adjacent right location.
    - **Suck:** Clean the current location.

Vacuum World

Transition Model:

- **Expected Effects:** Actions generally have the anticipated outcomes, but there are specific exceptions:
  - **Boundaries:**
    - Moving Left in the leftmost location has no effect.
    - Moving Right in the rightmost location has no effect.
  - **Cleaning:**
    - Sucking in a clean location has no effect.
- **State Transitions:** The transition model describes how each action changes the state of the environment.

Goal State

- The states in which every cell is clean.

Path Cost:

- **Definition:** The path cost is a measure of the total cost to reach the goal state from the initial state.
- **Metric:** In this problem, each action has a cost of 1.
  - **Calculation:** The path cost is the number of actions taken (steps) to reach the goal state.
  - **Example:** If it takes 4 actions to clean all locations, the path cost is 4.

Real-World Problems

Route-finding problems involve navigating specified locations and transitions between them. While some applications, like driving directions on websites or in-car systems, are straightforward, others—such as routing video streams, military operations planning, and airline travel-planning—require more complex algorithms.

Consider the airline travel problems that must be solved by a travel-planning Website:

- **States:** Each state includes the current location (e.g., an airport) and time. It also records historical aspects like previous flights, fare types, and whether they were domestic or international, as these affect the cost and availability of future actions.
- **Initial State:** Defined by the user's query, specifying the starting point, time, and other preferences.
- **Actions:** Take any available flight from the current location, considering seat class, departure time, and required transfer times within the airport.
- **Transition Model:** Taking a flight updates the state to the flight's destination and arrival time.
- **Goal State:** The goal is reached when the user arrives at the specified final destination. Sometimes the goal can be more complex, such as "arrive at the destination on a nonstop flight."
- **Path Cost:** Calculated based on factors like monetary cost, waiting and flight times, customs procedures, seat quality, frequent-flyer benefits, and more.

More Real-World Problems

- **Touring Problems:** Touring problems describe a set of location that must be visited, rather than a single goal destination.
  - The travelling salesman problem is a touring problem in which every city on a map must be visited. The aim is to find the shortest tour.
  - Touring problems are similar to route-finding problems but with a key distinction. Instead of simply finding a route, the goal is to visit every specified location at least once. The state space is more complex, as each state must track both the current location and the set of locations the agent has already visited.
- A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: cell layout and channel routing.
- **Robot navigation** is a generalization of the route-finding problem. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite.

Search Algorithms

- A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.
- We consider algorithms that superimpose a **search tree** over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state.
- Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.
- The **state space** describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another.
- The **search tree** describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).
- The **frontier** separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

Search data structures

Three kinds of queues are used in search algorithms:

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, ***f***. It is used in best-first search.

- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.

- A **LIFO queue** or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.

A **node** in the tree is represented by a data structure with four components

- node.S: the state to which the node corresponds;
- node.Parent: the node in the tree that generated this node;
- node.Action: the action that was applied to the parent's state to generate this node;
- node.Path-Cost: the total cost of the path from the initial state to this node.

We need a data structure to store the **frontier**. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:

- Is-Empty(frontier) returns true only if there are no nodes in the frontier.
- Pop(frontier) removes the top node from the frontier and returns it.
- Top(frontier) returns (but does not remove) the top node of the frontier.
- Add(node,frontier) inserts node into its proper place in the queue.

Redundant Paths

Redundant Paths:

- **Definition:** Redundant paths refer to different sequences of actions that lead to the same state multiple times within a search tree. Essentially, the same state is reached through different paths, but the state itself has already been explored earlier.
- **Impact:** Redundant paths do not provide new information and can unnecessarily increase the size of the search tree, making the search process less efficient.

Repeated States:

- **Definition:** A repeated state occurs when the same state is encountered multiple times during the search process, but through different paths. This can lead to the exploration of the same state more than once, resulting in inefficiency.
- **Impact:** Repeated states can slow down the search process, especially in large state spaces, as the algorithm may end up exploring the same state multiple times.

Cycles:

- **Definition:** A cycle occurs when a path leads back to a state that has already been visited earlier in the current path, forming a loop. Cycles can cause the search to go in circles, potentially leading to infinite loops if not handled properly. A cycle is a special case of a redundant path.
- **Impact:** Cycles can significantly hamper the efficiency of a search algorithm, and in the worst case, can cause the algorithm to never reach a solution if it gets stuck in an infinite loop.

We call a search algorithm a graph search if it checks for redundant paths and a tree search if it does not check.

Measuring problem-solving performance

1. Completeness

- **Definition**: Completeness refers to whether the algorithm is guaranteed to find a solution if one exists and to correctly report failure if no solution is available.
- **Importance**: A complete algorithm ensures that all possible paths are explored (or sufficient paths, depending on the method), making it reliable for finding solutions in search spaces.

2. Optimality

- **Definition**: Optimality measures whether the algorithm can find the solution with the lowest path cost among all possible solutions.
- **Importance**: This is crucial in applications where cost minimization is essential, such as route finding in transportation networks. An optimal algorithm guarantees that the best solution is found, which can have significant implications for resource utilization.

3. Time Complexity

- **Definition**: Time complexity assesses how the execution time of the algorithm grows with the size of the input or the search space.
- **Importance**: Understanding time complexity helps predict performance and efficiency. Algorithms with lower time complexity are preferred, especially for large search spaces, as they can provide solutions more quickly.

4. Space Complexity

- **Definition**: Space complexity measures the amount of memory required by the algorithm to perform the search, including both the space needed to store the search space and any auxiliary data structures.
- **Importance**: Space complexity is crucial for determining how well an algorithm can operate within memory constraints. Algorithms with high space complexity might struggle with larger inputs, making them less practical in memory-limited environments.

Uninformed Search

Uninformed Search Strategies

Uninformed search strategies are search algorithms that operate without additional information about the goal or the nature of the problem beyond the basic definition of the state space.

Key Characteristics of Uninformed Search Strategies:

- **No Heuristic Information**: Uninformed search strategies do not employ any domain-specific knowledge or heuristics to prioritize or evaluate paths in the search space.
- **Exhaustive Exploration**: They often explore the entire search space, which can lead to inefficiencies, especially in large or complex problems.
- **Optimality and Completeness**: Some uninformed search strategies guarantee finding an optimal solution or ensuring completeness (finding a solution if one exists), while others may not.

Types of Uninformed Search Strategies

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. Depth-Limited Search
4. Iterative Deepening Search
5. Uniform Cost Search
6. Bidirectional Search

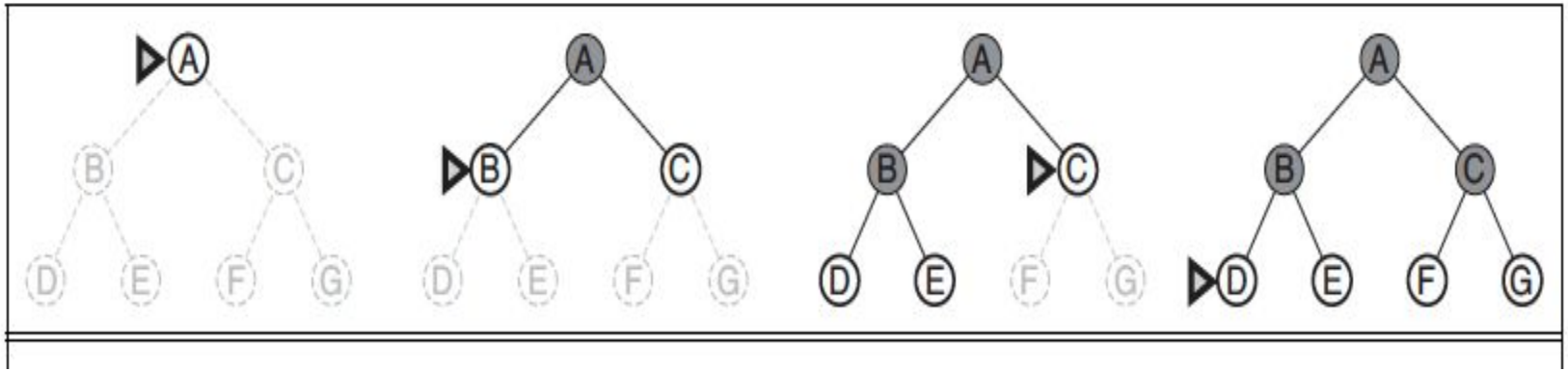
Breadth-First Search (BFS)

Breadth-First Search (BFS) is an uninformed search algorithm used for traversing or searching tree or graph data structures. It explores all the neighbor nodes at the present depth level before moving on to nodes at the next depth level. This strategy is particularly useful for finding the shortest path in unweighted graphs.

Key Characteristics of BFS:

1. **Level Order Exploration**: BFS explores nodes level by level. It starts from the root (or starting node) and explores all adjacent nodes before moving deeper into the tree or graph.
2. **Queue Data Structure**: BFS uses a queue to keep track of nodes that need to be explored. The first node added to the queue will be the first one to be processed (FIFO - First In, First Out).

Example



BFS Algorithm Steps

1. **Initialize**:
   - Create a queue and enqueue the starting node.
   - Maintain a set to keep track of visited nodes to prevent cycles.
2. **Process Nodes**:
   - While the queue is not empty:
     - Dequeue the front node from the queue.
     - If the dequeued node is the goal node, return it as the solution.
     - Otherwise, enqueue all unvisited neighboring nodes of the current node, marking them as visited.
3. **Terminate**:
   - If the queue becomes empty without finding the goal node, report that no solution exists.

Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

BFS Performance Measure

Completeness: BFS is complete, meaning that if there is a solution, it will find it (assuming the search space is finite).

Optimality: BFS is optimal for unweighted graphs, as it guarantees finding the shortest path (in terms of the number of edges) to the goal node.

Time Complexity: The time complexity of BFS is $O(b^d)$, where b is the branching factor (the average number of children per node) and d is the depth of the shallowest solution. This means that BFS can become inefficient for large search spaces.

Space Complexity: The space complexity is also $O(b^d)$ because all nodes at the current depth level need to be stored in memory before moving to the next level.

Formal Definition of Branching Factor (*b*) and Depth (*d*)

Branching Factor (b):

- The branching factor ***b*** of a graph/tree is defined as the average number of children (or adjacent nodes) each node has. In other words, for a node ***v***, the branching factor ***b*** represents the number of immediate neighbors (children) that can be reached from ***v***.
- Mathematically, if ***v*** is a node and ***N(v)*** represents the set of neighbors (children) of ***v***, then the branching factor ***b*** is the average of ***|N(v)|*** across all nodes:

$$b = \frac{1}{|V|} \sum_{v \in V} |N(v)|$$

where ***|V|*** is the total number of nodes in the graph.

Depth (d):

- The depth ***d*** of a solution in a search tree or graph is defined as the number of edges in the shortest path from the root node (or starting node) to the goal node. It represents the "level" at which the goal node is located in the search tree.
- For a search problem, ***d*** is the depth of the shallowest goal node. It is the minimum number of steps required to reach the goal from the starting point.

Derivation of Time Complexity $O(b^d)$ for BFS

Number of Nodes Explored:

- At depth 0 (root level), BFS explores 1 node (the root node).
- At depth 1, BFS explores up to b nodes (the children of the root).
- At depth 2, BFS explores up to b^2 nodes (the children of the nodes at depth 1).
- Continuing this pattern, at depth d, BFS explores up to b^d nodes.

The total number of nodes explored by BFS up to depth **d** can be approximated by summing the nodes at each level:

$$\text{Total nodes explored} = 1 + b + b^2 + \cdots + b^d$$

- The sum of this geometric series can be approximated by its largest term when b>1 and d is sufficiently large:

$$1 + b + b^2 + \cdots + b^d \approx b^d$$

Therefore, the time complexity of BFS, which is proportional to the number of nodes explored, is: **$O(b^d)$**

This represents the worst-case time complexity, where BFS has to explore all nodes up to the depth d before finding the goal node.

Proof of Completeness

To prove the completeness of BFS, we argue that BFS explores all possible nodes at each depth level before moving on to the next level. Here's how it works:

- **Step 1:** BFS starts at the root node (or starting node) and explores all its immediate neighbors (nodes at depth 1).
- **Step 2:** BFS then moves on to explore all neighbors of the nodes at depth 1 (nodes at depth 2).
- **Step 3:** This process continues level by level until BFS either finds the goal node or exhausts all possible nodes in the graph.
- **Assumption:** Suppose there is a solution, i.e., a path from the starting node ***s*** to the goal node ***g***.
- **Observation:** Since BFS explores nodes level by level, it will eventually reach the depth where the goal node ***g*** resides. BFS will examine all nodes at this depth, including ***g***, assuming the graph is finite and the goal node exists.

Conclusion: BFS is guaranteed to find the goal node if one exists. Therefore, BFS is complete for finite graphs.

Proof of Optimality

Proof by Contradiction:

- **Assumption:** Assume BFS is not optimal. This implies that there exists a path ***P*** from the start node ***s*** to the goal node ***g*** found by BFS that is not the shortest path. Let the actual shortest path be P_{short}, with a length (number of edges) of ***d***.
- **Observation:** BFS explores all nodes at depth 1 before moving to depth 2, all nodes at depth 2 before moving to depth 3, and so on. Therefore, BFS will explore all paths of length ***d*** before it explores any paths of length greater than ***d***.
- **Contradiction Setup:** Since P_{short} is the shortest path, its length is ***d***. According to the BFS algorithm, all paths of length ***d*** will be explored before any longer paths, including ***P***. Hence, BFS should have found P_{short} before considering ***P***, which contradicts the assumption that BFS found a longer path first.
- **Contradiction Conclusion:** The assumption that BFS is not optimal leads to a contradiction. Therefore, BFS must be optimal, finding the shortest path in an unweighted graph.

BFS Applications

Finding the Shortest Path in Unweighted Graphs: BFS is commonly used to find the shortest path in unweighted graphs, such as in social networks or maps, where all edges are considered equal.

Level-Order Traversal in Trees: BFS performs level-order traversal, which is useful for printing tree nodes level by level, and is essential in various tree algorithms.

Finding Connected Components in Graphs: BFS can identify connected components in undirected graphs, helping to analyze the structure and connectivity of networks.

Web Crawlers: BFS is utilized in web crawlers to explore web pages systematically, ensuring that all linked pages are visited in a structured manner.

Pathfinding in AI: BFS is employed in AI for pathfinding in game environments or mazes, helping characters find the shortest route from a starting point to a target.

Limitations of BFS

- **Exponential Growth:** The most significant limitation of BFS is its exponential growth in time and space complexity due to the $O(b^d)$ complexity. As the depth ***d*** and branching factor ***b*** increase, the number of nodes BFS must explore becomes infeasible.

- **Memory Consumption:** BFS requires storing all nodes at the current depth level in memory, leading to high memory consumption. This can be a bottleneck in large graphs.

- **Irrelevance in Weighted Graphs:** BFS is optimal for unweighted graphs, but in weighted graphs, algorithms like Dijkstra's algorithm or A* search are preferred as they account for varying edge weights.

Scenarios Where BFS Might Be Inefficient

1. Graphs with Large Branching Factors

- **Inefficiency:** In graphs where each node has a large number of children (high branching factor), BFS can quickly generate an overwhelming number of nodes to explore at each level.
- **Example:** Consider a social network graph where each person (node) has hundreds of connections (edges). At each level, BFS needs to explore a vast number of connections, leading to high memory consumption and slower performance.
- **Example Analysis:** If b=10 and d=5, BFS would need to explore and store up to 10^5=100,000 nodes, which can be computationally expensive and memory-intensive.
- **Impact:** The exponential growth in the number of nodes explored at each level can lead to excessive memory usage and longer computation times.

2. Deep Graphs

- **Inefficiency:** In very deep graphs, where the shortest path to the goal node lies at a significant depth, BFS might take a long time to reach the goal since it explores all nodes at each depth level before moving deeper.
- **Example:** In a game tree where the solution is several moves deep, BFS will systematically explore all possible moves at each level, which can be inefficient compared to depth-first strategies that dive deeper more quickly.
- **Impact:** For deep solutions, BFS may explore a large number of irrelevant nodes at shallower levels, leading to unnecessary computation.

3. Memory-Intensive Searches

- **Inefficiency:** BFS stores all nodes at the current level in memory before proceeding to the next level. In large graphs, this can lead to high memory usage, which can be a significant limitation on systems with limited memory resources.
- **Example:** In a large maze represented as a graph, BFS needs to store all possible paths at each level, which can quickly consume large amounts of memory.
- **Impact:** The requirement to store all nodes at each level can make BFS infeasible for very large graphs or on systems with constrained memory.

4. Uniformly Unweighted Graphs

- **Inefficiency:** While BFS is optimal for finding the shortest path in unweighted graphs, it can be inefficient in scenarios where the graph is large and uniformly unweighted, meaning there are many equally valid paths. BFS will explore all paths level by level without any prioritization, leading to excessive exploration.
- **Example:** In a large, uniform network where all connections are equally viable, BFS will explore all paths indiscriminately, leading to redundant exploration.
- **Impact:** In uniformly unweighted graphs, BFS may spend unnecessary time exploring multiple paths that all lead to the same result.

5. BFS in Infinite Graphs

- **Inefficiency:** In infinite or very large graphs, BFS is impractical because it must explore all nodes at each depth level, which is impossible in an infinite structure.
- **Example:** Consider an infinite grid where the goal node is far away. BFS will attempt to explore all nodes at each distance from the starting point, which is not feasible.
- **Impact:** BFS is not suitable for infinite graphs, as it will never terminate unless the goal is extremely close to the starting point.

6. Graphs with Cycles

- **Inefficiency:** In graphs with cycles, BFS must keep track of visited nodes to avoid re-exploration, which adds overhead. If the graph contains many cycles, this tracking can become complex and resource-intensive.
- **Example:** In a network with redundant connections, BFS must ensure it doesn't revisit the same nodes through different paths, which can slow down the algorithm.
- **Impact:** The need to manage cycles increases both memory and computational overhead, making BFS less efficient.

7. Finding Multiple Paths

- **Inefficiency:** BFS is designed to find the shortest path but is less efficient when the goal is to find multiple paths or all paths to a goal. BFS will redundantly explore all possibilities, leading to unnecessary computation.
- **Example:** In a network routing scenario where multiple paths are needed, BFS will explore all nodes at each level, even after finding a valid path.
- **Impact:** For scenarios requiring multiple solutions, BFS's level-by-level exploration can lead to excessive computation compared to more targeted search algorithms.

8. Sparse Graphs

- **Inefficiency:** In sparse graphs, where the number of edges is much lower than the maximum possible number of edges, BFS can become inefficient due to the large distances between connected nodes. BFS explores all nodes level by level, and in a sparse graph, this can mean traversing many levels where only a few nodes are connected, leading to wasted exploration.
- **Example:** Consider a sparse social network where most people have very few connections. BFS will explore each level, but since few connections exist, it will spend a lot of time processing nodes that do not lead to the goal.
- **Impact:** The level-by-level exploration in sparse graphs can lead to BFS traversing many irrelevant or empty levels, increasing the time and computational resources required to find the goal.

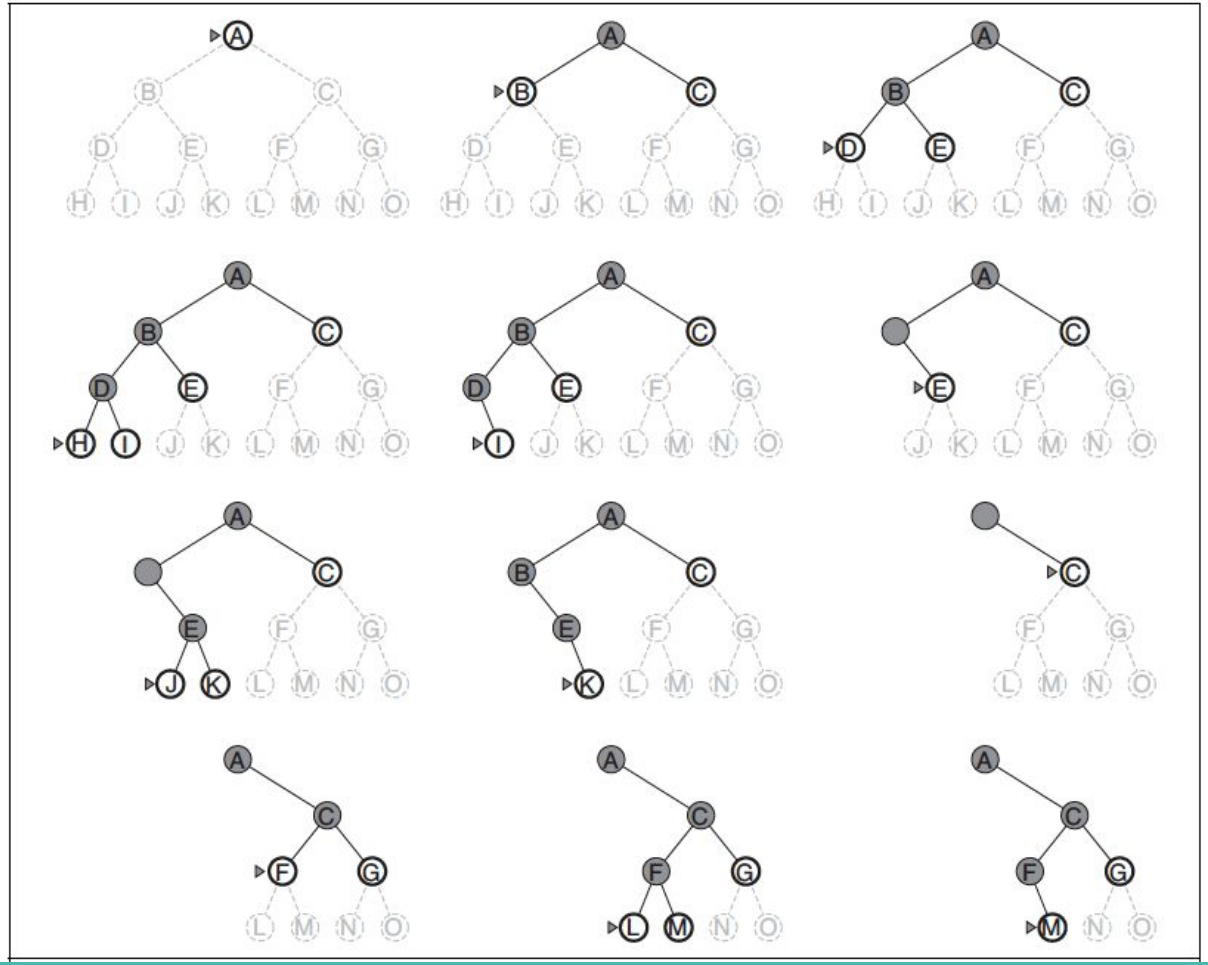
Depth-First Search (DFS)

Depth-First Search (DFS) is an uninformed search algorithm used for traversing or searching tree or graph data structures. It explores as far down a branch as possible before backtracking, making it distinct from other search strategies like Breadth-First Search (BFS).

Key Characteristics of DFS:

1. **Stack Data Structure**: DFS uses a stack to keep track of the nodes to be explored. This can be implemented using a recursive approach (which utilizes the call stack) or explicitly with an iterative approach using a manual stack.
2. **Backtracking**: When DFS reaches a node with no unvisited neighbors, it backtracks to the most recent node with unexplored neighbors. This characteristic makes it suitable for exploring all possible paths in a search space.

Example


A
B C
D E F G
H I J K L M N O
A
B C
D E F G
H I J K L M N O
A
B C
D E F G
H I J K L M N O
A
B C
D E F G
H I J K L M N O
A
B C
D E F G
H I J K L M N O
A
B C
D E F G
H I J K L M N O
A
C
E F G
J K L M N O
A
C
E F G
J K L M N O
A
B C
E F G
K L M N O
A
C
F G
L M N O
A
C
F G
L M N O
A
C
F G
L M N O
A
C
F G
L M N O
A
C
F G
L M N O

DFS Algorithm Steps

1. **Initialize**:
   - Create a stack to keep track of nodes to explore.
   - Push the starting node onto the stack.
   - Maintain a set or list to keep track of visited nodes to avoid cycles.
2. **Process Nodes**:
   - While the stack is not empty:
     - Pop a node from the top of the stack.
     - If the popped node is the goal node, return it as the solution.
     - Otherwise, mark the node as visited.
     - Push all unvisited neighbors of the current node onto the stack.
3. **Terminate**:
   - If the stack becomes empty without finding the goal node, report that no solution exists.

NOTE: Algorithm based on the steps to be done by YOU!

DFS Performance Measure

Completeness:

- **Finite Graphs:** DFS is complete in finite graphs, meaning that if there is a solution, DFS will find it eventually. This is because DFS explores as far as possible along each branch before backtracking, ensuring that all nodes will eventually be explored.
- **Infinite Graphs:** DFS is not complete in infinite graphs, as it may get stuck in an infinitely deep branch without ever finding the solution. For example, if a graph contains an infinite path and the solution lies elsewhere, DFS might never reach the solution.

Optimality:

- DFS is **not optimal** in general. Since DFS explores each branch to its fullest depth before backtracking, it does not necessarily find the shortest path to the goal. The first solution found by DFS might not be the optimal one. For instance, if there is a shorter path that lies in a different branch, DFS may miss it and find a longer path first.

Time Complexity:

- The time complexity of DFS depends on the number of vertices (V) and edges (E) in the graph.
- In an **adjacency list** representation, the time complexity of DFS is: O(V+E)
- This is because DFS visits every vertex and every edge in the graph exactly once during the search process.

Space Complexity:

- The space complexity of DFS is mainly determined by the depth of the recursion stack (in the recursive implementation) or the size of the stack data structure (in the iterative implementation).
- In the **worst case**, the space complexity is O(d), where d is the maximum depth of the search tree. In the worst-case scenario, DFS might explore a path that is as deep as the maximum depth of the graph.
- If the graph is very deep or has many branches, the space complexity can be significant, potentially leading to stack overflow in the recursive implementation.

Step-by-Step Analysis of Time Complexity

Vertex Exploration

- Each vertex ***v*** in the graph is visited exactly once by DFS. When ***v*** is visited, it is marked as visited, and all its neighbors are explored.
- **Total Time for Vertex Exploration:** Since each vertex is visited once, the total time for vertex exploration is ***O(V)***.

2. Edge Exploration

- For each vertex ***v***, DFS iterates through all its neighbors (i.e., all vertices connected to ***v*** by an edge).
- The key point is that each edge ***(u,v)*** is explored exactly once during the entire DFS process:
  - When DFS visits vertex ***u***, it explores edge ***(u,v)*** to check if ***v*** is visited.
  - Similarly, when DFS visits vertex ***v***, it explores edge ***(v,u)*** to check if ***u*** is visited.
- **Total Time for Edge Exploration:** Since each edge is checked once, the total time for edge exploration is ***O(E)***.

3. Overall Time Complexity

- The time complexity of DFS is the sum of the time required for vertex exploration and edge exploration.
- **Total Time Complexity:**

  O(Vertex Exploration) + O(Edge Exploration) = O(V)+O(E)

  Time Complexity of DFS = O(V+E)

- This time complexity applies to both connected and disconnected graphs. For disconnected graphs, DFS may be initiated multiple times, once for each disconnected component, but the overall complexity remains ***O(V+E)***.

Proof of Optimality

Proof by Contradiction

To mathematically prove that DFS is not optimal, we'll use a proof by contradiction.

Assumptions:

1. Let ***G=(V,E)*** be an unweighted graph where ***V*** is the set of vertices and ***E*** is the set of edges.
2. Assume that DFS is optimal, meaning that it always finds the shortest path from the start node ***s*** to the goal node ***g***.
3. Let the depth of a node be defined as the number of edges from the start node ***s*** to that node.

Setup:

Consider the following situation:

- Let there be two paths from the start node ***s*** to the goal node ***g***.
    1. **Path 1** has a length of d_1 edges.
    2. **Path 2** has a length of $d_2$edges.
- Assume without loss of generality that $d_1 < d_2$, meaning Path 1 is shorter than Path 2.

DFS Execution:

- DFS explores paths by diving deep into the graph, meaning it will explore nodes by moving along the edges until it can no longer proceed or it finds the goal node ***g***.
- Let's assume that DFS explores Path 2 first and finds the goal node ***g*** at depth d_2.

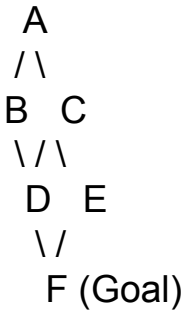
Contradiction Argument:

1. **By our assumption**, DFS is optimal and should find the shortest path, which is Path 1 with length d_1.
2. **However**, since DFS explores Path 2 first and reaches the goal node ***g*** at depth d_2, DFS will return Path 2 as the solution.
3. This contradicts the assumption that DFS is optimal because it found a path of length d_2 instead of the shorter path of length d_1.

Conclusion:

- Since our assumption that DFS is optimal led to a contradiction, DFS cannot be optimal in general.
- **Thus, DFS is not guaranteed to find the shortest path** from ***s*** to ***g*** in an unweighted graph.

Example



Edges:

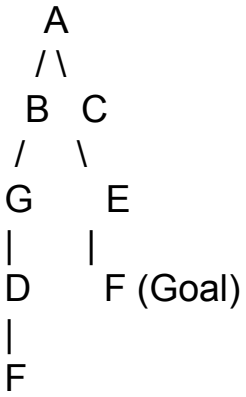
- A -> B
- A -> C
- B -> D
- C -> D
- C -> E
- D -> F
- E -> F

Paths from A to F:

1. **Path 1:** A -> B -> D -> F (Length = 3)
2. **Path 2:** A -> C -> D -> F (Length = 3)
3. **Path 3:** A -> C -> E -> F (Length = 3)

The path DFS finds is A→B→D→F

The path A→B→D→F is of length 3



Edges:

- A -> B
- A -> C
- B -> G
- G -> D
- D -> F
- C -> E
- E -> F

Paths from A to F:

1. **Path 1:** A -> B -> G -> D -> F (Length = 4)
2. **Path 2:** A -> C -> E -> F (Length = 3)

If DFS starts at node A and chooses to explore the left branch first (i.e., goes from A→B), it will follow the longer path A→B→G→D→F, which has a length of 4.

The shorter path A→C→E→F with length 3 might be ignored if the left branch is fully explored first.

Proof of Completeness

Proof by Contradiction

To mathematically prove the completeness of DFS, we will use a proof by contradiction.

Assumptions:

1. Let ***G=(V,E)*** be a finite graph, where ***V*** is the set of vertices (nodes) and ***E*** is the set of edges.
2. Assume that there exists a path ***P*** from the start node ***s*** to the goal node ***g***.
3. Assume that DFS is not complete, meaning that it fails to find the solution (i.e., it does not find ***g***).

DFS Algorithm Characteristics:

- DFS explores nodes by traversing as deep as possible along each branch before backtracking.
- DFS uses a stack (either explicitly or via recursion) to keep track of the nodes to be explored.
- DFS visits each node at most once.

Contradiction Setup:

1. **Finite Graph:** Since ***G*** is finite, there are a finite number of vertices ***|V|*** and edges ***|E|*** in ***G***.
2. **Path Existence:** Let ***P = (s,v_1,v_2,…,g)*** be a valid path from ***s*** to ***g*** in ***G***. The path ***P*** has a finite length ***k***, where ***k*** is the number of edges in ***P***.

DFS Behavior:

- DFS will start at node ***s*** and explore one of the paths.
- If DFS does not immediately find ***g***, it will backtrack and explore alternative paths.
- Since ***G*** is finite, DFS will eventually exhaust all possible paths from ***s*** to any other reachable vertex.

Contradiction Argument:

1. **Assume DFS does not find g.**
   - This implies that DFS fails to explore the path ***P*** from ***s*** to ***g***, despite ***P*** being a valid path in the graph.
2. **Since G is finite, DFS will eventually visit every vertex reachable from s.**
   - Let $V' \subseteq V$ be the set of all vertices reachable from ***s***. Since ***P*** is a valid path from ***s*** to ***g***, ***g*** must be in ***V'***.
   - By the nature of DFS, every vertex in ***V'*** will eventually be visited because DFS continues until all paths have been explored or the goal is found.
3. **DFS must eventually explore the path P entirely.**
   - As DFS explores all vertices in ***V'***, it will visit each vertex in the path $P=(s,v_1,v_2,\ldots,g)$ sequentially.
   - Since ***g*** is the last vertex in ***P***, DFS must eventually visit ***g***.
4. **Contradiction:**
   - The assumption that DFS does not find ***g*** contradicts the fact that DFS visits every vertex in ***V'***, which includes ***g***.
   - Therefore, our assumption that DFS is not complete must be false.

DFS Applications

Topological Sorting: Used to order tasks in directed acyclic graphs (DAGs) based on dependencies.

Cycle Detection: Employed to detect cycles in both directed and undirected graphs.

Finding Connected Components: Identifies connected components in undirected graphs, helping analyze graph structure and connectivity.

Solving Puzzles: Useful for solving various puzzles, such as mazes and the N-Queens problem, by exploring all possible configurations.

Generating Mazes: DFS can be used to generate mazes by starting at a point and randomly carving paths until all cells are visited, resulting in a maze-like structure.

Limitations of Depth-First Search (DFS)

1. Non-Optimality

- **Limitation:** DFS does not guarantee finding the shortest path in an unweighted graph. It might return a longer path even when a shorter one exists.
- **Impact:** This makes DFS less suitable for applications where the optimal (shortest) solution is required, such as routing and navigation systems.

2. Incompleteness in Infinite Graphs

- **Limitation:** DFS is not complete in infinite graphs, meaning it may fail to find a solution even if one exists. This occurs because DFS can get stuck in an infinitely deep branch without ever exploring other potentially valid paths.
- **Impact:** In problems with infinite or very large state spaces, such as certain types of puzzle-solving or game AI, DFS may not reliably find a solution.

3. Memory Usage in Deep Recursion

- **Limitation:** In cases where the graph or tree is very deep, the recursive implementation of DFS can lead to a stack overflow due to excessive memory usage. This is especially problematic in environments with limited stack space.
- **Impact:** In deeply nested structures or highly recursive problems, DFS may cause program crashes or excessive memory consumption.

4. Sensitivity to Graph Structure

- **Limitation:** DFS's performance can vary significantly depending on the structure of the graph. In unbalanced trees or graphs with long paths, DFS may waste time exploring deep, irrelevant branches before finding the goal.
- **Impact:** This sensitivity makes DFS less predictable and less efficient in cases where the graph structure is not well understood or controlled.

5. Difficulty in Handling Cycles

- **Limitation:** In graphs with cycles, DFS requires careful handling to avoid revisiting nodes, which can lead to infinite loops and wasted computational effort. Although this can be mitigated by marking visited nodes, it adds complexity and increases space requirements.
- **Impact:** In complex graphs with many cycles, managing these challenges can reduce the efficiency and simplicity of using DFS.

6. Limited Applicability in Weighted Graphs

- **Limitation:** DFS does not take edge weights into account, making it unsuitable for problems where paths have different costs. In such scenarios, algorithms like Dijkstra's or A* are more appropriate.
- **Impact:** For problems involving cost optimization or weighted paths, DFS cannot provide the best solution.

Scenarios Where DFS Might Be Inefficient

1. Graphs with Long Paths and Deep Recursion

- **Inefficiency:** DFS explores as deep as possible before backtracking. In graphs with very long paths (or deep trees), DFS may traverse a long, unproductive path before finding the goal. This deep recursion can lead to inefficiency in both time and space.
- **Example:** In a deep tree where the goal node is located near the root but in a different branch, DFS might explore a very long branch that doesn't lead to the goal, resulting in wasted computation.
- **Technical Issue:** If the graph has a deep structure, the recursive DFS implementation may lead to a stack overflow, causing the program to crash or run inefficiently. This is especially problematic in languages with limited stack sizes.

2. Graphs with Cycles

- **Inefficiency:** In graphs that contain cycles, DFS can become inefficient if it revisits nodes, potentially getting stuck in an infinite loop. This is particularly problematic if the graph is large and contains many cycles.
- **Example:** Consider a graph representing a maze with loops. If DFS does not properly mark visited nodes, it might keep revisiting the same nodes, wasting time and computational resources.
- **Mitigation:** While this can be mitigated by marking visited nodes, the need for additional memory to track visited nodes increases the algorithm's space complexity.

3. Graphs with High Branching Factors

- **Inefficiency:** In graphs with high branching factors (i.e., each node has many neighbors), DFS can become inefficient because it might explore a large number of nodes in a deep branch that does not contain the goal.
- **Example:** In a graph where each node has hundreds of neighbors, DFS might explore a long and irrelevant path, causing the algorithm to perform unnecessary computations.
- **Comparison:** In such cases, BFS might be more efficient as it explores all nodes at the current depth level before moving to the next, ensuring that the shortest path is found.

4. Unbalanced Trees

- **Inefficiency:** In unbalanced trees, where some branches are much deeper than others, DFS can become inefficient because it might explore the deeper branches unnecessarily while the goal node might be located in a shallower branch.
- **Example:** In an unbalanced binary tree, if the left branch is very deep and the right branch is shallow (with the goal node), DFS might waste time exploring the deep left branch first.
- **Impact:** This results in longer execution times and increased memory usage, as DFS will explore many nodes that do not contribute to finding the goal.

5. Sparse Graphs with Distant Goals

- **Inefficiency:** In sparse graphs where the goal node is far from the start node, DFS might explore many unnecessary paths that do not lead to the goal. This is because DFS does not have any heuristic to guide it towards the goal, leading to inefficient exploration.
- **Example:** In a graph representing a road network, if the goal is on the opposite side of the graph, DFS might explore many distant and irrelevant paths before finding the correct one.
- **Performance:** This inefficiency becomes more pronounced in large graphs, where the number of potential paths can be enormous.

6. Finding the Shortest Path in Unweighted Graphs

- **Inefficiency:** DFS is not guaranteed to find the shortest path in an unweighted graph, as it may explore a long path first, even when a shorter path exists. This makes DFS inefficient in scenarios where finding the shortest path is crucial.
- **Example:** In an unweighted graph representing social connections, DFS might find a long path between two people, even when a much shorter path exists.
- **Alternative:** Breadth-First Search (BFS) would be more efficient in such cases, as it guarantees finding the shortest path in unweighted graphs.

7. Search in Infinite Graphs

- **Inefficiency:** DFS is not complete in infinite graphs, meaning that it might not find a solution even if one exists. This is because DFS can get stuck exploring an infinitely deep branch without ever finding the goal.
- **Example:** Consider a graph representing possible configurations in a puzzle. If the graph is infinite, DFS might explore an infinitely long sequence of moves without ever backtracking to find the correct solution.
- **Risk:** This makes DFS inefficient and potentially ineffective in applications involving infinite or very large state spaces.

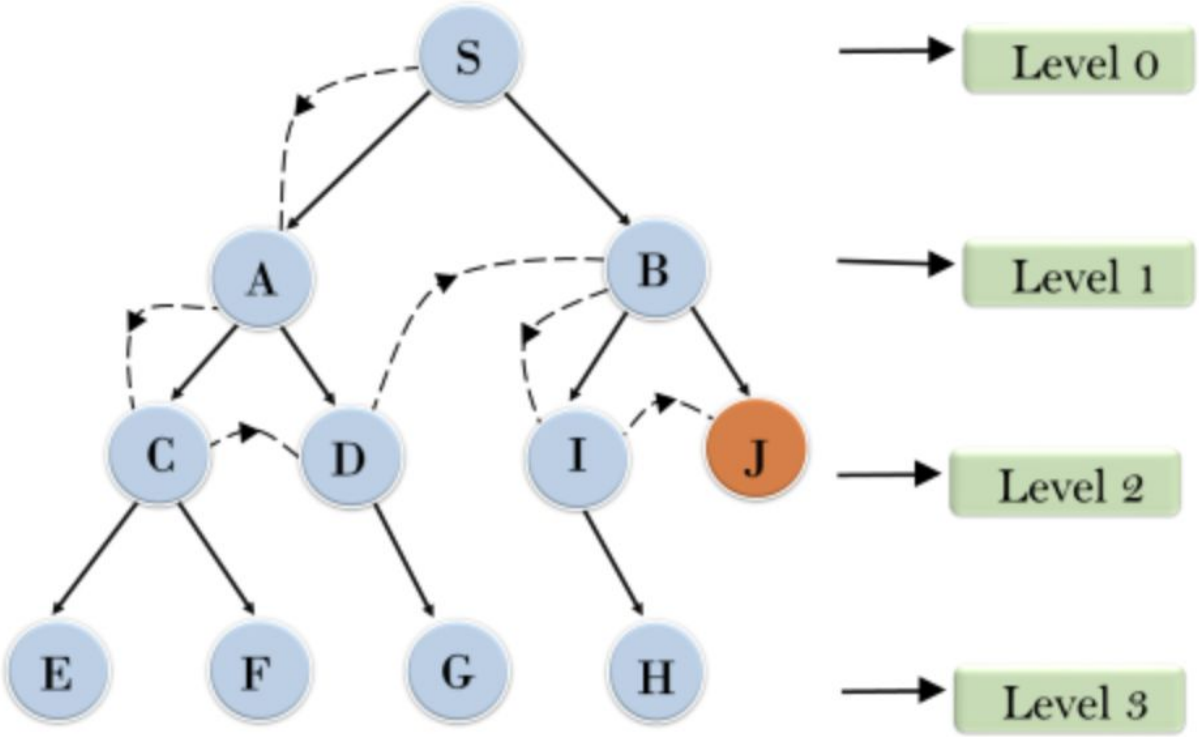
Depth-Limited Search

Depth-Limited Search (DLS) is a variation of the Depth-First Search (DFS) algorithm that imposes a maximum depth limit to prevent the search from going too deep into the search space. This approach is particularly useful in scenarios where infinite paths may exist, such as in certain graph or tree structures.

Key Characteristics of Depth-Limited Search:

1. **Depth Limit**: DLS introduces a parameter that specifies the maximum depth to which the search can explore. If the depth limit is reached, the search backtracks without exploring further.
2. **Controlled Exploration**: By limiting the depth, DLS avoids the risk of running indefinitely in deep or infinite structures, making it more efficient in certain contexts.
3. **Stack-Based**: Like DFS, DLS uses a stack (either explicitly or through recursion) to manage the nodes being explored.

Example



DLS Algorithm Steps

1. **Initialize**:
    - Create a stack to manage nodes to explore.
    - Push the starting node onto the stack along with its depth (initially 0).
    - Define a maximum depth limit.
2. **Process Nodes**:
    - While the stack is not empty:
        - Pop a node and its depth from the stack.
        - If the depth is equal to the limit, backtrack (do not explore further).
        - If the node is the goal node, return it as the solution.
        - Otherwise, mark the node as visited.
        - Push all unvisited neighbors of the current node onto the stack, incrementing their depth by one.
3. **Terminate**:
    - If the stack becomes empty without finding the goal node, report that no solution exists within the depth limit.

Algorithm

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Time Complexity

Depth-Limited Search (DLS) is a variation of Depth-First Search (DFS) where the search is limited to a specified depth ***l***. The time complexity of DLS depends on the branching factor ***b*** (the average number of child nodes for each node) and the depth limit ***l***.

1. Understand the Tree Structure:

In DLS, the search tree can be visualized as a complete tree where:

- **Root node:** At depth 0.
- **Level 1:** Contains ***b*** nodes.
- **Level 2:** Contains b^2 nodes.
- ...
- **Level l:** Contains b^l nodes.

The total number of nodes in a tree up to depth ***l*** is the sum of the nodes at each level:

$$\textbf{Total nodes} = 1 + b + b^2 + \cdots + b^l$$

2. Calculate the Sum of the Series:

This sum represents the total number of nodes that the DLS algorithm will potentially explore. The sum of a geometric series can be calculated using the formula:

$$S = \frac{b^{l+1} - 1}{b - 1}$$

3. Simplifying for Large l

For large ***l***, the term b^{l+1} dominates the sum ***S***, making the smaller terms negligible. This means the sum ***S*** can be approximated by just considering the largest term in the series, which is b^l:

$$S \approx \frac{b^{l+1} - 1}{b - 1} \approx \frac{b^{l+1}}{b - 1}$$

Now, to derive the time complexity, we focus on the dominant term that contributes to the overall growth of the function. The b^{l+1} term is the most significant, and since it dominates, the time complexity is approximated as:

$$O(b^{l+1}) \text{ where } b^{l+1} = b \cdot b^l = O(b^l)$$

Why l+1 Becomes l:

- **Mathematical Insight:** When considering time complexity, constants and lower-order terms are typically ignored because they do not affect the asymptotic behavior. In the expression b^{l+1}, the base ***b*** is constant, so increasing the exponent by 1 (which is what happens when you go from $l-1$ to ***l***) does not fundamentally change the asymptotic growth rate. Thus, both b^{l+1} and b^l grow at a similar exponential rate.
- **Simplification:** We drop the constant factor and focus on the term that defines the exponential growth, which is b^l. Hence, the time complexity is simplified to $O(b^l)$.

Completeness of DLS

- **Scenario 1: The Goal is Within the Depth Limit l:**
  - If the goal node is at a depth ***d ≤ l***, Depth-Limited Search (DLS) will find the goal. This is because DLS will explore all nodes up to the depth limit ***l***. Since DLS systematically explores every possible path within the depth limit, it is guaranteed to reach the goal if it exists within that limit.
- **Scenario 2: The Goal is Deeper than the Depth Limit l:**
  - If the goal node is at a depth ***d > l***, DLS will not find the goal because it will not explore nodes beyond depth ***l***. Therefore, in this case, DLS is not complete because it fails to find the solution that lies beyond its depth limit.

Mathematical Expression of Completeness:

Let ***d*** be the depth at which the goal node ***g*** is located. DLS is complete if:

$$d \leq l$$

where ***l*** is the depth limit. If ***d>l***, DLS is incomplete.

Conclusion on Completeness:

- **Complete when d ≤ l :** If the goal lies within the depth limit, DLS is complete.
- **Incomplete when d > l:** If the goal lies beyond the depth limit, DLS is not complete.

Optimality of DLS

- DLS explores nodes at increasing depths up to the limit ***l***, similar to Depth-First Search (DFS). However, DLS is not guaranteed to find the shortest path if there are multiple paths to the goal.
- DLS might find a solution at a greater depth before discovering a shallower path because it explores deeply before considering alternative, potentially shorter paths at shallower levels.

Mathematical Proof of Non-Optimality:

Let there be two paths to the goal ***g***:

1. **Path 1:** Length d_1
2. **Path 2:** Length d_2

Assume $d_1 < d_2$

- If DLS explores Path 2 first because the nodes along Path 2 are explored before those along Path 1, it will find the solution at depth d_2 before discovering the shorter path of length d_1.
- Therefore, DLS can return a solution that is not optimal because it does not consider the depth or cost of the paths it explores.

Conclusion on Optimality:

- **Not Optimal:** DLS is not guaranteed to find the shortest path, as it may find a solution at a greater depth before exploring all shallower paths.

DLS Performance Measure

Time Complexity: The time complexity of DLS is $O(b^l)$, where b is the branching factor and l is the depth limit. This can be more efficient than standard DFS if the limit is set appropriately.

Space Complexity: The space complexity remains $O(l)$, where l is the depth limit, as only nodes up to that depth need to be stored in memory.

Completeness: DLS is complete only if the solution lies within the specified depth limit. If the solution is deeper than the limit, DLS will fail to find it.

Optimality: DLS does not guarantee an optimal solution, as it may not explore all possible paths if they exceed the depth limit.

DLS Applications

Search Space Pruning: DLS can be used in optimization problems where certain solutions are known to be infeasible beyond a certain depth. By applying depth limits, DLS can prune large portions of the search space early, improving efficiency.

Hierarchical Problem Solving: In hierarchical problem-solving environments, DLS can be employed to navigate through different levels of abstraction. For example, it can be used to explore various levels of a decision tree where only a limited number of decisions need to be considered at once.

Algorithm Testing and Validation: DLS can be useful in testing algorithms that operate in recursive environments by restricting the depth of recursion, allowing developers to verify behavior and performance without risking stack overflow or excessive computation.

Interactive Game Development: In interactive gaming, DLS can help limit the depth of AI decision-making processes to maintain responsiveness. This ensures that AI characters make timely decisions without evaluating all potential future states.

State Space Exploration: In fields like robotics or automated planning, DLS can be used to explore state spaces where certain states may require excessive resources to evaluate. By limiting the depth, robots can focus on immediate and feasible actions while ignoring less relevant deep states.

Limitations of Depth-Limited Search (DLS)

1. Risk of Incompleteness

Limitation:

- **Missing the Solution:** DLS is not guaranteed to find a solution if the goal node is deeper than the specified depth limit. If the solution exists beyond the depth limit, DLS will terminate without finding the goal, leading to incompleteness.

Scenarios Where This is Inefficient:

- **Deep Search Spaces:** In problems where the solution may exist at an unknown or deep level, such as deep puzzle-solving or long-path routing problems, setting an arbitrary depth limit may cause DLS to miss the solution entirely.
- **Unknown Solution Depth:** In situations where the depth of the solution is not known in advance, DLS might be inefficient because it might miss the goal simply due to an inadequate depth limit.

2. Sensitivity to Depth Limit Selection

Limitation:

- **Optimal Depth Limit is Unknown:** The efficiency of DLS heavily depends on choosing an appropriate depth limit. If the limit is set too low, the algorithm may fail to find the solution even if it exists. If the limit is set too high, the search may explore unnecessary paths, reducing efficiency.

Scenarios Where This is Inefficient:

- **Dynamic or Complex Environments:** In environments like AI decision-making or dynamic systems where the depth of the goal can vary, selecting an appropriate depth limit can be challenging, leading to inefficiencies.
- **Exploratory Problems:** In problems where the search space is complex or poorly understood, and the optimal depth limit is unclear, DLS might waste time exploring irrelevant paths or fail to reach the goal.

3. Inefficiency in Large and Wide Search Trees

Limitation:

- **High Branching Factor:** In search spaces with a high branching factor, where each node has many children, DLS can become inefficient. The algorithm might spend a considerable amount of time exploring wide portions of the tree at shallow depths without making significant progress toward deeper nodes where the goal might lie.

Scenarios Where This is Inefficient:

- **Large Search Trees:** In large decision trees, such as those found in game theory or combinatorial optimization problems, DLS might explore many unnecessary branches at shallow levels, leading to inefficient use of time and resources.
- **Wide Networks:** In network routing or communication systems with many possible paths, DLS might be inefficient because it cannot focus on deeper nodes where the goal might be located.

4. Lack of Optimality

Limitation:

- **Non-Optimal Solutions:** DLS does not guarantee that it will find the optimal solution, even if it finds the goal within the depth limit. The first solution found may not be the shortest or least costly path, as DLS does not prioritize paths based on cost or distance.

Scenarios Where This is Inefficient:

- **Pathfinding in Weighted Graphs:** In scenarios where finding the least-cost path is critical, such as in transportation or logistics planning, DLS may return a suboptimal path, making it less suitable for such applications.
- **Optimization Problems:** In problems like resource allocation or project scheduling, where the optimal solution is essential, DLS may provide a solution, but it might not be the best one available.

5. Potential for Redundant Exploration

Limitation:

- **Repeated Node Visits:** In DLS, the same nodes can be revisited multiple times across different recursive calls if the depth limit is not carefully managed. This redundancy can lead to wasted computational effort.

Scenarios Where This is Inefficient:

- **Cyclic Graphs:** In cyclic graphs, where nodes can be revisited through different paths, DLS might waste time exploring the same nodes repeatedly, leading to inefficiency.
- **Large Repetitive Structures:** In search spaces with repetitive structures or patterns, such as certain types of mazes or puzzles, DLS may revisit the same states multiple times, slowing down the search process.

6. Memory Usage Concerns in Deep Searches

Limitation:

- **Stack Overflow Risk:** Although DLS is more memory-efficient than BFS, it still relies on recursion or a stack to manage the depth-first exploration. In very deep searches, the risk of stack overflow becomes significant, particularly in environments with limited memory resources.

Scenarios Where This is Inefficient:

- **Deep Recursion:** In problems requiring deep exploration, such as complex puzzle-solving or long-term planning, DLS can run into stack overflow issues, especially on systems with limited stack size.
- **Embedded Systems:** In memory-constrained environments like embedded systems or IoT devices, DLS might be inefficient due to its reliance on recursive depth-first exploration, which can exhaust available memory.

7. Limited Applicability to Real-Time Systems

Limitation:

- **Time Constraints:** DLS is not well-suited for real-time systems where decisions need to be made within strict time limits. The algorithm's reliance on exploring nodes up to a certain depth can lead to unpredictable delays, especially if the depth limit is not well-calibrated.

Scenarios Where This is Inefficient:

- **Real-Time Decision-Making:** In applications like robotics or real-time game AI, where timely responses are critical, DLS may introduce delays due to its depth-limited exploration, making it less suitable for real-time environments.
- **Interactive Systems:** In interactive applications where the system needs to respond quickly to user input, DLS might be inefficient if the depth limit causes unnecessary delays or fails to find a timely solution.

8. Incomplete in Infinite Graphs

Limitation:

- **Incompleteness in Infinite Spaces:** DLS is incomplete in infinite graphs or search spaces. If the graph has an infinite structure and the solution lies beyond the depth limit, DLS will not find it, making the algorithm unsuitable for such problems.

Scenarios Where This is Inefficient:

- **Infinite State Spaces:** In problems like certain types of puzzles or theoretical computer science problems where the search space is infinite, DLS might terminate prematurely, failing to find a solution that exists at a greater depth.
- **Open-Ended Problems:** In open-ended AI tasks, where the search space is not well-defined or bounded, DLS may struggle to find solutions if they are located beyond the arbitrarily set depth limit.

Iterative Deepening Search

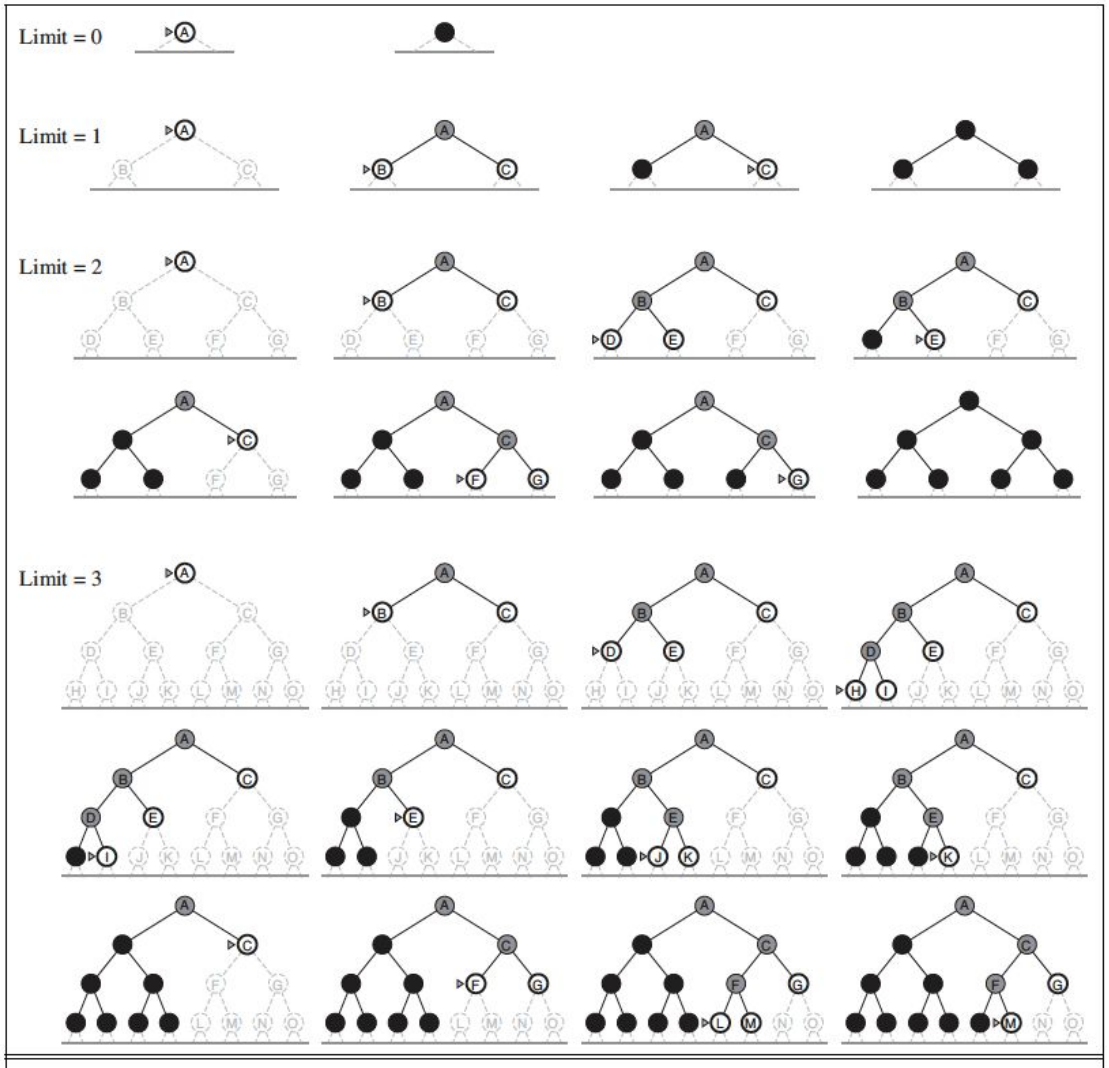
Iterative Deepening Search (IDS) is a search algorithm that combines the benefits of Depth-First Search (DFS) and Breadth-First Search (BFS). It performs a series of depth-limited searches, incrementally increasing the depth limit with each iteration until a solution is found. This approach is particularly useful for searching large or infinite state spaces while ensuring optimality and completeness.

Key Characteristics of Iterative Deepening Search:

1. **Combination of DFS and BFS**: IDS utilizes the depth-limited search methodology, effectively exploring the search space like DFS but doing so iteratively with increasing depth limits.
2. **Memory Efficiency**: IDS uses much less memory than BFS because it only needs to store the nodes along the current path and the depth limit, making it suitable for large state spaces.

Example

Four iterations of iterative deepening search on a binary tree.



IDS Algorithm Steps

Initialize:

- Set the initial depth limit (usually starting at 0).
- Create a loop that will run until a solution is found.

Depth-Limited Search:

- Perform a depth-limited search for the current depth limit:
  - Start from the initial node and explore its children.
  - If the depth of the current node equals the limit, stop exploring further down that branch.
  - If the current node is the goal node, return it as the solution.
  - Otherwise, push unvisited neighbors onto the stack with their depth incremented by one.

Increment Depth Limit:

- If the depth-limited search fails to find a solution, increment the depth limit by one and repeat the process.

Terminate:

- Continue this process until a solution is found or the search space is exhausted.

Proof of Optimality

Argument for Optimality in Unweighted Graphs:

- **Shallowest Goal First:** In an unweighted graph, the cost of reaching a node is proportional to the depth of the node. Therefore, the optimal solution is the one found at the shallowest depth.
- **Mechanism:** IDS performs a complete search at each depth level before moving to the next level. This means that IDS will explore all nodes at depth ***d*** before exploring any nodes at depth ***d+1***.
- **Guarantee:** Since IDS systematically searches at increasing depths and finds the goal at the shallowest possible depth, it is guaranteed to find the optimal solution (the shallowest goal node) if multiple goals exist.

Mathematical Reasoning:

- Let $g_1, g_2, \ldots, g_n$ be goal nodes at depths $d_1, d_2, \ldots, d_n$, where $d_1 < d_2 < \cdots < d_n$.
- IDS will find g_1 during the iteration with depth limit $l = d_1$.
- Since IDS does not explore deeper levels until all nodes at the current depth have been explored, g_1 will be found before any deeper goals like $g_2, g_3, \ldots$
- **Conclusion:** IDS is optimal in unweighted graphs because it finds the shallowest goal first, ensuring that the least-cost solution is found.

Optimality in Weighted Graphs:

- **Important Note:** IDS is not optimal in weighted graphs where the cost is not solely determined by depth. To guarantee optimality in such scenarios, a different algorithm like Uniform-Cost Search or A* would be necessary.

Proof of Completeness

Argument for Completeness:

- **Mechanism:** IDS performs a series of Depth-Limited Searches (DLS), starting from a depth limit of 0 and incrementally increasing the depth limit by 1 in each iteration.
- **Process:** In each iteration, IDS explores all nodes at the current depth before increasing the limit. This means that, for a finite state space, IDS will eventually explore all possible depths in the search tree.
- **Guarantee:** Since IDS increments the depth limit with each iteration, it will eventually reach the depth at which the goal node is located, provided that the goal exists within the state space.
- **Handling Infinite State Spaces:** In the case of an infinite state space, if a solution exists at some finite depth ***d***, IDS will find it after ***d*** iterations.

Mathematical Reasoning:

- Let ***d*** be the depth of the shallowest goal node in the search tree.
- IDS will perform a DLS with depth limits l = 0,1,2,…,d
- When the depth limit reaches ***l = d***, IDS will find the goal node because it explores all nodes at that depth.

Conclusion: IDS is complete because it systematically explores deeper levels until the goal is found. If the goal exists at any finite depth, IDS will find it.

IDS Performance Measure

Optimality: IDS is optimal for unweighted graphs, meaning it will always find the shortest path (in terms of the number of edges) to the goal node.

Completeness: IDS is complete, ensuring that if a solution exists, it will be found eventually, even in infinite search spaces.

Time Complexity: The time complexity is $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution. Although this is similar to BFS, IDS performs better in terms of memory usage.

Space Complexity: The space complexity is O(d), where ddd is the maximum depth. This is much more efficient than BFS, which has a space complexity of $O(b^d)$.

IDS Applications

Distributed Systems: In distributed systems, IDS can be used to explore state spaces across multiple nodes or processors while minimizing communication overhead. Each node can handle its own depth-limited search, incrementally sharing results to reach a global solution.

AI Planning: IDS can be applied in automated planning systems where actions can lead to exponentially growing state spaces. By incrementally deepening the search, it can effectively manage the exploration of possible plans and their associated consequences.

Exploration in Robotics: IDS can be utilized in robotic exploration tasks, where robots systematically explore unknown environments. By controlling the depth of exploration, robots can cover areas thoroughly without exceeding memory constraints.

Software Model Checking: In software verification and model checking, IDS can be used to exhaustively explore state spaces of finite-state systems. By incrementally increasing depth, it can verify properties of systems without running out of memory.

Genetic Algorithms: In hybrid approaches that combine genetic algorithms with search techniques, IDS can be used to guide the search process through a population of solutions. This allows for focused exploration of the most promising areas of the solution space while managing computational resources effectively.

Limitations of Iterative Deepening Search (IDS)

1. Redundant Exploration of Nodes

Limitation:

- **Redundant Searches:** IDS repeatedly explores the same nodes at shallower depths during each iteration. For example, nodes at depth 1 are explored in every iteration until the depth limit exceeds 1. This redundancy can lead to inefficiencies, particularly when the search space is large.

Scenarios Where This is Inefficient:

- **Large Search Spaces:** In search problems with a large number of nodes, such as complex combinatorial problems, the repeated exploration of nodes can result in significant overhead, making IDS slower than other algorithms that avoid such redundancy.
- **Problems with Shallow Goals:** In scenarios where the goal is located at a shallow depth, IDS's repeated exploration of shallower nodes adds unnecessary computation, making it less efficient compared to a single BFS.

2. High Computational Overhead in Deep Searches

Limitation:

- **Exponential Time Complexity:** The time complexity of IDS is exponential in the depth of the solution because it revisits nodes multiple times across different depth limits. As the depth of the search increases, the number of nodes revisited grows exponentially.

Scenarios Where This is Inefficient:

- **Deep Solutions:** In search problems where the goal is located at a significant depth, such as deep game trees or long-path puzzles, IDS may become inefficient due to the large number of nodes that need to be revisited across multiple iterations.
- **Complex Planning Problems:** In complex AI planning problems where solutions require exploring deep decision trees, IDS's computational overhead can make it less suitable, especially when compared to more direct depth-first approaches.

3. Memory Usage and Resource Constraints

Limitation:

- **Space Complexity vs. Efficiency:** While IDS is designed to be memory-efficient (similar to DFS), it still requires enough memory to handle repeated depth-limited searches. The memory efficiency can be a trade-off against the time complexity, which may not be ideal in all situations.

Scenarios Where This is Inefficient:

- **Resource-Constrained Environments:** In environments with strict memory or computational constraints, such as embedded systems or mobile devices, the overhead of repeated searches may not be justifiable, leading to inefficiency.
- **Real-Time Applications:** In real-time systems, where quick decision-making is critical (e.g., robotics or online pathfinding), the time required by IDS to revisit nodes might introduce delays, making it less suitable for time-sensitive tasks.

4. Difficulty in Handling Weighted Graphs

Limitation:

- **Not Optimal for Weighted Graphs:** IDS is optimal in unweighted graphs, where the goal is to find the shortest path in terms of the number of edges. However, it does not account for varying edge weights and therefore might not find the least-cost path in weighted graphs.

Scenarios Where This is Inefficient:

- **Weighted Pathfinding:** In scenarios such as road networks or transportation logistics where path costs vary, IDS may explore paths that are longer in terms of cost, even if they are shorter in terms of depth. This makes IDS less effective compared to algorithms like Uniform-Cost Search (UCS) or A*.
- **Cost-Sensitive Problems:** In AI applications where cost considerations are crucial, such as resource allocation or economic modeling, IDS's lack of cost awareness can lead to suboptimal solutions.

5. Inefficiency in Infinite or Very Large Search Spaces

Limitation:

- **Challenges with Infinite Spaces:** IDS is complete in finite search spaces, but in infinite or very large search spaces, it may not be practical. The algorithm might continue searching indefinitely without finding a solution if the goal is located far from the start.

Scenarios Where This is Inefficient:

- **Infinite State Spaces:** In problems with infinite state spaces, such as certain types of puzzles or AI decision-making trees that have no bound, IDS may not terminate in a reasonable time frame, making it impractical for these scenarios.
- **Extremely Large Search Spaces:** Even in finite but extremely large search spaces, IDS may become inefficient as it requires an enormous amount of time to explore deep nodes, especially when the goal is located at a great distance.

6. Lack of Heuristic Guidance

Limitation:

- **Uninformed Search Strategy:** IDS is an uninformed search algorithm, meaning it does not use any domain-specific knowledge or heuristics to guide the search towards the goal more efficiently. It explores all nodes at each depth level without prioritization.

Scenarios Where This is Inefficient:

- **Heuristic-Driven Searches:** In problems where heuristics can provide significant guidance (e.g., in A* search), IDS's lack of direction can make it less efficient, as it does not take advantage of heuristic information to prune the search space.
- **Complex Problem Domains:** In complex domains like AI planning or navigation in large environments, where heuristics can dramatically reduce the search space, IDS may expand many irrelevant nodes, leading to inefficiency.

7. Suboptimal Performance in Wide Search Trees

Limitation:

- **Wide Branching Factors:** IDS performs poorly in search trees with high branching factors, where each node has many children. The algorithm must explore all children at each depth level, leading to an exponential growth in the number of nodes explored.

Scenarios Where This is Inefficient:

- **High Branching Factor Problems:** In problems like large-scale decision-making or certain types of games with many possible moves at each step, IDS can become overwhelmed by the sheer number of nodes, making it less efficient compared to algorithms that can prioritize certain branches.
- **Network Pathfinding:** In network pathfinding scenarios with many alternative routes at each step, IDS may become bogged down by the number of options it must explore, leading to inefficiency.

Uniform Cost Search

Uniform Cost Search (UCS) is a search algorithm that is used to find the least-cost path from a given starting node to a goal node in a weighted graph. It is a variant of Dijkstra's algorithm and is a form of best-first search that expands the node with the lowest path cost first. UCS is particularly useful for finding optimal solutions in scenarios where path costs vary.

Key Characteristics of Uniform Cost Search:

1. **Cost-Based Expansion**: UCS expands nodes based on the cumulative cost to reach them from the start node. It always chooses the node with the lowest total path cost for expansion.
2. **Data Structures**: UCS uses a priority queue (often implemented as a min-heap) to manage the nodes based on their cumulative costs. This allows for efficient retrieval of the next node to expand.

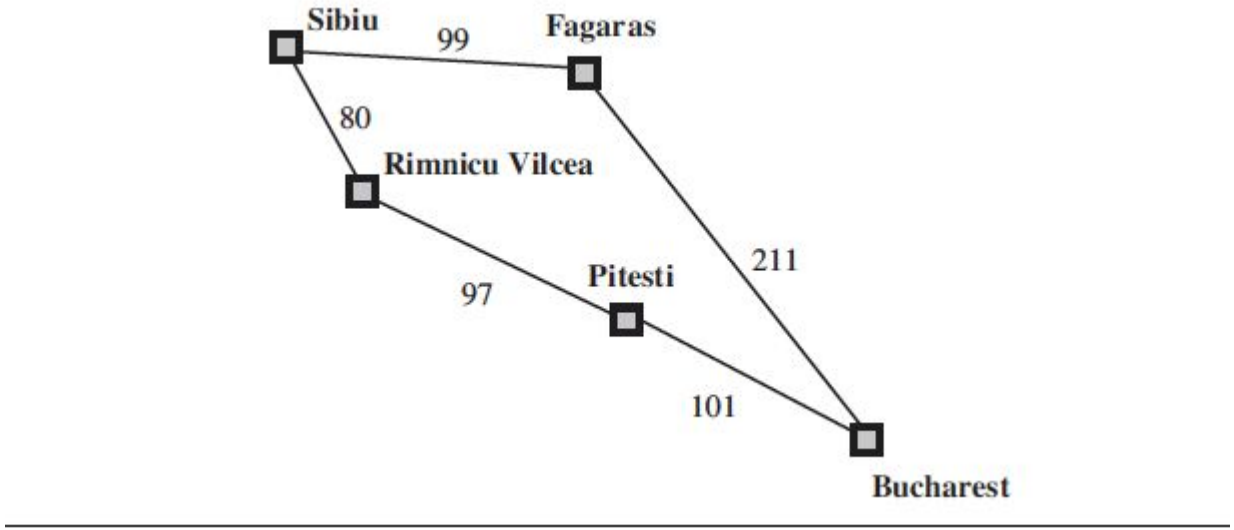
Contrast with BFS and DFS:

- **BFS:** Explores level by level and finds the shortest path in terms of the number of edges (in unweighted graphs).
- **DFS:** Explores deeply before backtracking but does not guarantee finding the shortest or least-cost path.
- **UCS:** Explores paths based on their cumulative cost, ensuring the discovery of the least-cost path in weighted graphs.

Real-world analogy: Imagine you're navigating a city using public transport. You want to reach your destination using the least expensive route, regardless of how many stops it takes. UCS helps you find this optimal route.

Example

Part of the Romania state space, selected to illustrate uniform-cost search.



Dijkstra's Algorithm: A specific case of UCS where all edge costs are non-negative, commonly used in graph-based pathfinding.

UCS Algorithm Steps

Initialize the Priority Queue:

- Create a priority queue (often implemented as a min-heap) to store the frontier. The frontier is the set of all leaf nodes available for expansion at any given point.
- Start by inserting the initial node (start state) into the priority queue with a cumulative cost of 0.

Initialize the Explored Set:

- Create an empty set to keep track of the nodes (states) that have already been expanded (explored) to avoid redundant processing.

Begin the Search Loop:

- **While the priority queue is not empty**, repeat the following steps:

Pop the Node with the Lowest Cost:

- Extract the node with the lowest cumulative cost from the priority queue. This node represents the current path with the smallest cost.
- Let this node be called `current_node`, and its associated cost be `current_cost`.

UCS Algorithm Steps

Goal Test:

- Check if `current_node` is the goal state.
- If `current_node` is the goal state, the algorithm terminates and returns the solution, which includes the path and the total cost to reach the goal.

Add the Current Node to the Explored Set:

- If `current_node` is not the goal state, add it to the explored set to ensure that it is not expanded again.

Expand the Current Node:

- For each child (or successor) of `current_node`:
  - **Calculate the Cumulative Cost:** Determine the cumulative cost to reach the child by adding the edge cost from `current_node` to the child's cumulative cost.
  - **Check if the Child is in the Explored Set:**
    - If the child is not in the explored set or the priority queue, add it to the priority queue with its cumulative cost.
    - If the child is already in the priority queue but with a higher cumulative cost, update the priority queue with the new lower cost.
  - **Note:** UCS prioritizes paths with lower costs, so the priority queue ensures that the node with the lowest cumulative cost is expanded next.

Termination:

- If the priority queue becomes empty and the goal has not been found, this indicates that no solution exists within the given state space, and the algorithm returns failure.

UCS Algorithm

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Proof of Optimality

Step 1: Node Expansion and Path Costs

- Consider the moment UCS expands the goal node ***g***. Let ***C(g)*** be the cumulative cost associated with reaching ***g*** along the path UCS followed.
- Assume, for the sake of contradiction, that there exists another path to ***g*** with a lower cost, denoted as ***C'(g) < C(g)***.

Step 2: Contradiction by Assumption

- Since UCS always expands the node with the lowest cumulative cost next, it must have expanded all nodes with cumulative costs less than ***C(g)*** before expanding ***g***.
- If there were a path to ***g*** with a cumulative cost ***C'(g) < C(g)***, UCS would have expanded the nodes along this cheaper path to ***g*** first.
- Therefore, UCS would have reached ***g*** via the cheaper path before considering the path with cost ***C(g)***.

Step 3: Conclusion

- The assumption that there is a cheaper path ***C'(g) < C(g)*** contradicts the way UCS operates, as UCS would have found and expanded this cheaper path first.
- Since UCS expands nodes in order of their cumulative path cost, when UCS expands the goal node ***g***, the path ***C(g)*** must be the minimum possible cost to reach ***g***.
- Therefore, no other path to ***g*** with a cost less than ***C(g)*** exists.

Step 4: Generalization

- This reasoning holds for all nodes in the graph, not just the goal node. Thus, UCS guarantees that when it expands any node, it has found the least-cost path to that node.
- When UCS expands the goal node, it has found the optimal solution to the problem.

Proof of Completeness

Step 1: Initialization and Node Expansion

- UCS starts by initializing the priority queue with the start node ***s*** and a path cost of 0.
- Nodes are expanded in order of increasing cumulative cost, meaning that the node with the lowest cost is expanded first.

Step 2: Handling Non-Negative Costs

- Each edge in the graph has a non-negative cost, meaning that the cumulative cost of any path increases as nodes are expanded.
- Because UCS always expands the node with the smallest cumulative cost, it systematically explores paths in increasing order of their total cost.

Step 3: Exploring All Paths

- Suppose a solution (a path from ***s*** to ***g***) exists.
- UCS will continue expanding nodes and exploring new paths until the goal node ***g*** is reached.
- Because all edge costs are non-negative, the cumulative cost associated with paths cannot decrease, ensuring that UCS doesn't revisit nodes with lower costs unnecessarily.

Step 4: No Infinite Loop or Incomplete Exploration

- Since UCS expands nodes in order of increasing cost, it avoids getting stuck in loops or revisiting nodes excessively.
- If the graph is finite, UCS will eventually exhaust all possible paths leading from sss and will reach ***g*** because it explores every possible path with increasing cost until the goal is found.
- If the graph is infinite but has finite cumulative costs, UCS will still find the goal ***g*** by exploring paths with increasingly higher costs. If a path exists, UCS will eventually expand the goal node.

Step 5: Termination

- UCS terminates as soon as it expands the goal node ***g***. Since UCS prioritizes paths with lower costs, the first time UCS expands ***g***, it does so using the least-cost path.
- Since UCS expands nodes based on cumulative path costs, it ensures that all possible paths to ***g*** are considered, and ***g*** will be reached if reachable.

Conclusion

- **Completeness:** UCS is complete because it systematically explores all possible paths from the start node ***s*** in increasing order of cost. If a path to the goal node ***g*** exists, UCS will eventually find and expand ***g***, ensuring that the goal is reached.

Time Complexity

Assumptions:

- Let ***b*** be the branching factor, which is the average number of children each node has.
- Let ***C**** be the cost of the optimal (least-cost) solution.
- Let ϵ be the smallest positive edge cost in the graph.
- The state space is assumed to be finite, and all edge costs are non-negative.

Basic Operation of UCS

- UCS explores nodes in increasing order of their cumulative path cost.
- Each time a node is expanded, UCS adds its children to the priority queue with their cumulative path costs.
- The priority queue is used to ensure that the node with the lowest cumulative path cost is expanded next.

Number of Nodes Expanded by UCS

- The key to understanding the time complexity is estimating how many nodes UCS might need to expand to find the optimal solution.

Cost Boundaries and Path Costs:

- Let ***C(n)*** be the cumulative cost of the path from the start node ***s*** to a node ***n***.
- For UCS to explore a node ***n*** before finding the goal node ***g***, the cost ***C(n)*** must be less than or equal to ***C**** , where ***C**** is the cost of the optimal solution.
- Since UCS only expands nodes with costs $C(n) \leq C^*$, the number of nodes expanded depends on how many nodes have path costs within this range.

Depth and Branching Factor Consideration:

- The number of nodes that UCS might consider expands exponentially with the depth of the tree. However, in UCS, "depth" is more appropriately measured in terms of path cost rather than the number of edges.
- For each depth level in terms of cost, UCS could potentially explore all nodes whose cumulative costs fall within that level.
- Each node ***n*** can generate up to ***b*** children. If UCS explores nodes up to a cumulative cost ***C****, the number of nodes expanded is related to the total number of nodes within that cost bound.

Estimating the Total Number of Nodes Expanded

- The total number of nodes UCS can expand is determined by how many nodes have a cumulative path cost less than or equal to ***C****.
- Consider the worst-case scenario where UCS might expand nodes even with slightly larger costs ***C(n) > C**** due to ties or small variations in costs. However, in the limit, the number of nodes with path costs within ***C**** is proportional to $b^{C^*/\epsilon}$.
- Therefore, the time complexity of UCS is dominated by the number of nodes with cumulative costs within ***C****, which is approximately:

$$\textbf{Time Complexity} = O(b^{C^*/\epsilon})$$

Conclusion

- The time complexity of UCS is $O(b^{C^*/\epsilon})$, where:
  - ***b*** is the branching factor,
  - ***C**** is the cost of the optimal solution, and
  - ϵ is the smallest positive edge cost in the graph.
- This complexity reflects the exponential growth in the number of nodes UCS must expand as the cost C* increases, relative to the smallest cost increment ϵ.

UCS Performance Measure: Summary

Optimality: UCS is optimal because it expands nodes in order of their cumulative cost, ensuring that the first time it reaches the goal, it does so via the least-cost path.

Completeness: UCS is complete as long as all edge costs are non-negative. It will always find a solution if there is one because it explores all possible paths in order of increasing cost.

Time Complexity: UCS has a time complexity of $O(b^{C^*/\epsilon})$, where b is the branching factor, C^* is the cost of the optimal solution, and ϵ is the smallest positive edge cost. The time complexity grows exponentially with the cost of the optimal path.

Space Complexity: UCS has a space complexity of $O(b^{C^*/\epsilon})$, as it must store all paths in the priority queue. This is similar to its time complexity, meaning UCS can be memory-intensive, especially in large search spaces.

UCS Applications

Logistics and Supply Chain Management: UCS can optimize transportation routes, determining the least-cost path for moving goods from warehouses to distribution centers while considering fluctuating transportation costs.

Telecommunication Network Design: UCS is used to find optimal routing paths for data packets in networks, minimizing latency and costs while maximizing bandwidth and reliability.

AI in Robotics: UCS assists in robotic path planning by navigating complex environments with varying movement costs, allowing robots to conserve energy and maximize efficiency in tasks.

Medical Treatment Pathways: UCS optimizes treatment plans by evaluating different options based on cost-effectiveness and resource allocation, helping identify the most efficient pathway to achieve desired health outcomes.

Urban Planning: UCS helps evaluate construction routes for infrastructure development, allowing urban planners to minimize expenses and environmental impact by prioritizing paths with lower costs.

Limitations of UCS

1. High Memory Usage

Limitation:

- **Space Complexity:** UCS requires significant memory to store all the nodes in the priority queue until the search reaches the goal. Since UCS must store every explored path in the queue to ensure that the least-cost path is expanded first, the space complexity can become problematic, particularly in large or dense graphs.

Scenarios Where This is Inefficient:

- **Large Graphs:** In applications such as mapping large cities or vast networks, where the number of nodes and edges is very high, UCS can consume large amounts of memory, potentially exhausting available resources.
- **Dense Graphs:** In graphs where most nodes are connected by edges, the number of potential paths that UCS needs to store increases rapidly, leading to excessive memory usage.

2. Slow Performance in Graphs with Uniform Costs

Limitation:

- **Time Complexity:** When edge costs are relatively uniform (or identical), UCS behaves similarly to Breadth-First Search (BFS), exploring many unnecessary nodes at the same level of cost. This can lead to slow performance because UCS does not take advantage of cost variations to prune the search space effectively.

Scenarios Where This is Inefficient:

- **Uniformly Weighted Graphs:** In scenarios like grid-based pathfinding where all moves have the same cost, UCS will explore a broad set of nodes, making it slower compared to algorithms that exploit heuristics.
- **Flat Terrain Navigation:** In robotics, if a robot is navigating on a flat surface where all paths are equally costly, UCS might explore many unnecessary paths before finding the goal, leading to inefficiency.

3. Sensitivity to Small Variations in Edge Costs

Limitation:

- **Minimal Edge Cost ϵ\epsilonϵ:** The performance of UCS is sensitive to the smallest edge cost in the graph. If the smallest edge cost ϵ\epsilonϵ is very small, UCS may have to explore a large number of paths before distinguishing between slightly different cumulative costs, resulting in high computational overhead.

Scenarios Where This is Inefficient:

- **Graphs with Minor Cost Differences:** In applications such as financial modeling or logistics planning where small cost differences between paths matter, UCS might expand many similar-cost paths, increasing the computational burden.
- **Network Routing with Variable Costs:** In network routing where data packet delivery costs vary slightly due to congestion or traffic, UCS might inefficiently explore many near-optimal paths, increasing processing time.

4. Inefficiency in Deep Search Spaces

Limitation:

- **Exponential Growth:** UCS's time complexity is exponential in the depth of the least-cost solution, especially in deep search spaces. If the optimal solution is located deep in the search tree, UCS must potentially explore many levels of nodes before finding the goal.

Scenarios Where This is Inefficient:

- **Deep Decision Trees:** In AI decision-making processes, such as game playing where the solution lies deep in the decision tree, UCS can be slow because it needs to explore all possible paths up to a certain depth.
- **Long-Path Planning:** In robotics or planning problems where the optimal solution requires many steps (e.g., complex multi-stage manufacturing processes), UCS might take a long time to reach the goal due to the depth of the search space.

5. Difficulty Handling Negative Edge Costs

Limitation:

- **Incompatibility with Negative Costs:** UCS assumes non-negative edge costs. If the graph contains negative edge costs, UCS can fail to find the optimal solution, as it does not handle negative cycles properly. Negative costs can lead to situations where the cumulative cost of a path keeps decreasing, causing UCS to re-expand nodes and potentially enter an infinite loop.

Scenarios Where This is Inefficient:

- **Economic Modeling:** In economic scenarios where certain paths might have negative costs (e.g., subsidies, rebates), UCS is not suitable because it cannot accurately model and solve such problems.
- **Resource Allocation with Penalties:** In scenarios where penalties (negative costs) apply to certain decisions, UCS may struggle to find the optimal allocation due to its inability to handle negative costs correctly.

6. Inability to Leverage Heuristics

Limitation:

- **Lack of Heuristic Guidance:** UCS is an uninformed search algorithm, meaning it does not use heuristics to guide the search. This can lead to inefficiencies in large search spaces where heuristics could significantly reduce the number of nodes explored by focusing the search on promising areas.

Scenarios Where This is Inefficient:

- **Complex AI Planning:** In applications like AI planning or complex decision-making, where domain-specific knowledge could guide the search towards the goal more efficiently, UCS is less effective because it cannot incorporate this information.
- **Pathfinding in Large, Complex Environments:** In large maps or terrains where the goal is far away, UCS might explore many irrelevant paths that could be avoided with heuristic guidance, making it less efficient than algorithms like A*.

Bidirectional Search

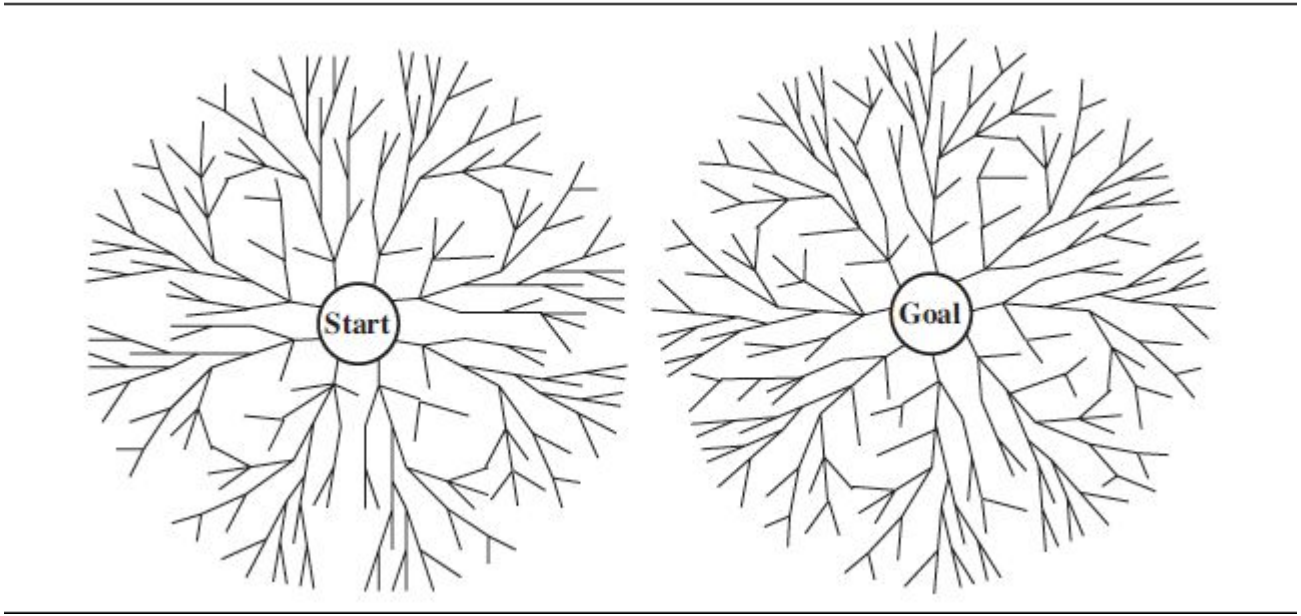
Bidirectional Search is a search algorithm that simultaneously explores the search space from both the initial state (start node) and the goal state, aiming to meet in the middle. This approach can significantly reduce the time complexity of finding a solution compared to unidirectional search methods, particularly in large state spaces.

Key Characteristics of Bidirectional Search:

1. **Two Fronts**: The algorithm maintains two separate searches: one that starts from the initial state and another that starts from the goal state. These searches progress concurrently.
2. **Meeting Point**: The searches continue until they intersect, which indicates that a solution has been found. This meeting point is crucial for determining the optimal path.

The rationale behind Bidirectional Search is that searching from two directions simultaneously can significantly reduce the number of nodes that need to be explored compared to a unidirectional search. In the best case, this approach can cut down the search space to roughly the square root of what would be explored by a single BFS.

Example



A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

BS Algorithm Steps

- **Initialization:**
  a. Two search fronts are initialized: one starting from the start node and one from the goal node.
  b. Each search front maintains its own queue (like in BFS) or priority queue (in weighted graphs).
  c. Two sets of explored nodes are maintained: one for each search direction.
- **Search Execution:**
  a. In each iteration, alternate between expanding the forward search and the backward search.
  b. Expand the node with the smallest cost (or next in line, in the case of BFS) from each frontier.
  c. For the forward search, explore all unvisited neighbors of the current node and add them to the frontier.
  d. For the backward search, do the same, but considering paths leading to the goal node.
- **Termination:**
  a. The search terminates when a node is found that has been expanded by both the forward and backward searches. This node lies on the shortest path between the start and goal nodes.
  b. The shortest path can be reconstructed by combining the paths from the start node to the meeting point (found by the forward search) and from the goal node to the meeting point (found by the backward search).

BS Algorithm

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Time Complexity

1. Understanding the Problem

Let:

- ***b*** be the branching factor (the average number of children per node).
- ***d*** be the depth of the solution (the number of edges in the shortest path from the start node to the goal node).

2. Time Complexity of Unidirectional BFS

In a traditional Breadth-First Search:

- The search expands nodes level by level.
- At depth ***d***, the total number of nodes that could be expanded is the sum of the nodes at all levels from 0 to ***d***.
- The number of nodes at each level ***i*** is approximately b^i.

The total number of nodes expanded by BFS up to depth ***d*** is:

$$\textit{Total nodes} = b^0 + b^1 + b^2 + \cdots + b^d \approx O(b^d)$$

3. Time Complexity of Bidirectional Search

In Bidirectional Search:

- The search is conducted from both the start node and the goal node, meeting somewhere in the middle.
- Instead of searching to depth ***d*** in one direction, Bidirectional Search goes to depth ***d/2*** from the start and ***d/2*** from the goal.

Nodes Expanded by Each Search

For each half of the search:

- The number of nodes expanded by the forward search up to depth ***d/2*** is $b^{d/2}$.
- Similarly, the number of nodes expanded by the backward search up to depth ***d/2*** is $b^{d/2}$.

Since the search is happening in two directions, the total number of nodes expanded by Bidirectional Search is the sum of the nodes expanded in both directions:

$$\textbf{Total nodes in Bidirectional Search} = b^{d/2} + b^{d/2} = 2b^{d/2}$$

4. Comparing the Complexities

- **Unidirectional BFS:** $O(b^d)$
- **Bidirectional Search:** $O(2b^{d/2})$

Since the factor of 2 is a constant, it can be ignored in big-O notation, so the time complexity of Bidirectional Search simplifies to: $\mathbf{O(b^{d/2})}$

Proof of Optimality

To prove the optimality of Bidirectional Search, we will argue that:

1. **Paths Found by Both Searches Are Optimal Up to the Meeting Point:**
   - The forward search finds the shortest path from ***S*** to ***M***.
   - The backward search finds the shortest path from ***G*** to ***M***.
2. **Combination of Paths Forms the Optimal Path:**
   - The path ***S***→***M***→***G*** formed by combining the two optimal paths ***S***→***M*** and ***G***→***M*** is the shortest possible path from ***S*** to ***G***.

Proof of Optimality

Step 1: Forward and Backward Searches Find Optimal Paths

- **Forward Search Optimality:**
  - The forward search from ***S*** expands nodes in order of increasing path length or cost.
  - When the forward search reaches ***M***, it must have found the shortest path from ***S*** to ***M*** because if there were a shorter path, it would have been expanded first.
- **Backward Search Optimality:**
  - Similarly, the backward search from ***G*** expands nodes in order of increasing path length or cost.
  - When the backward search reaches ***M***, it must have found the shortest path from ***G*** to ***M*** because if there were a shorter path, it would have been expanded first.

Step 2: Combining the Two Paths

- **Path from S to G via M:**
  - The path ***S*→*M*** found by the forward search is the shortest path from ***S*** to ***M***.
  - The path ***M*→*G*** found by the backward search is the shortest path from ***G*** to ***M***.
  - Therefore, the combined path ***S*→*M*→*G*** must be the shortest path from ***S*** to ***G*** because it consists of two optimal subpaths.

Step 3: No Shorter Path Exists

- **Suppose there exists a shorter path P′ from S to G that does not pass through M.**
  - Such a path would imply that one of the two searches (either forward or backward) missed a shorter subpath, which contradicts the optimality of the individual searches.
  - Since the forward search finds the shortest path from ***S*** to ***M*** and the backward search finds the shortest path from ***G*** to ***M***, no shorter alternative path ***P′*** can exist.

Conclusion

- **Optimality:** Bidirectional Search is optimal because it finds the shortest path from the start node ***S*** to the goal node ***G***. The two searches meet at a node ***M***, and the combination of the shortest paths from ***S*** to ***M*** and ***G*** to ***M*** forms the overall shortest path from ***S*** to ***G***.
- **No Shorter Path Exists:** If a shorter path existed, it would have been found by one of the two searches before they met, leading to a contradiction. Thus, the path found by Bidirectional Search is guaranteed to be the shortest.

Proof of Completeness

Step 1: Both Searches Explore All Reachable Nodes

- **Forward Search:**
  - The forward search starts from ***S*** and expands nodes level by level (in breadth-first fashion or based on cumulative cost if the graph is weighted).
  - It explores all nodes that can be reached from ***S***, ensuring that no reachable node is skipped.
- **Backward Search:**
  - Similarly, the backward search starts from ***G*** and expands nodes level by level.
  - It explores all nodes that can be reached from ***G***, ensuring that no reachable node is skipped.

Step 2: Searches Must Meet if a Path Exists

- **Assumption of a Path:**
  - Assume that there is at least one path from ***S*** to ***G***. This means there exists a sequence of nodes $S \rightarrow N1 \rightarrow N2 \rightarrow \cdots \rightarrow G$.
- **Meeting of Searches:**
  - As the forward search explores all nodes reachable from ***S*** and the backward search explores all nodes reachable from ***G***, the two searches are guaranteed to meet at some node along the path $S \rightarrow G$.
  - The node where the two searches meet can be any node ***M*** along the path, including ***S***, ***G***, or any intermediate node.

Proof of Completeness

Step 3: Termination at a Solution

- **Meeting Node M:**
  - When the forward and backward searches meet at node ***M***, the forward search has found the path ***S*→*M*** and the backward search has found the path ***G*→*M*** (in reverse).
- **Path Formation:**
  - The algorithm can then combine these two paths to form the full path ***S*→*M*→*G***.
- **Solution Guarantee:**
  - Since both searches are guaranteed to explore all reachable nodes, they will necessarily meet if a path exists. Therefore, a solution is guaranteed to be found.

Handling Edge Cases

- **No Path Exists:**
  - If no path exists between ***S*** and ***G***, one of the searches will eventually exhaust all possible nodes without finding a meeting point. In this case, Bidirectional Search correctly identifies that no solution exists.
- **Graph with a Single Node:**
  - If ***S*=*G***, Bidirectional Search trivially finds the solution without expanding any other nodes.

Conclusion

- **Completeness:** Bidirectional Search is complete because it guarantees finding a path from the start node ***S*** to the goal node ***G*** if such a path exists. The algorithm achieves this by ensuring that both the forward and backward searches explore all reachable nodes from their respective starting points, leading to a meeting point that forms the solution.

BS Performance Measure

Time Complexity:

- **Best Case**: **$O(b^{d/2})$** – In Bidirectional Search, the time complexity is significantly reduced compared to unidirectional search methods like Breadth-First Search (BFS). Since both searches (forward and backward) progress simultaneously and ideally meet in the middle, the depth ddd is effectively halved. Therefore, the time complexity is $O(b^{d/2})$ instead of $O(b^d)$, where b is the branching factor and d is the depth of the solution.
- **Worst Case**: In some cases, if the searches do not intersect early or if one of the search directions encounters a more complex path, the time complexity could approach $O(b^d)$, especially if the intersection happens closer to one end rather than in the middle.

Space Efficiency:$O(b^{d/2})$ – The space complexity of Bidirectional Search is also $O(b^{d/2})$, which is a significant improvement over BFS's $O(b^d)$. This reduction in space complexity occurs because each search only needs to store nodes up to the midpoint of the search space, rather than the entire depth.

Optimality

- **Guaranteed Optimal Solution**:
  - Bidirectional Search guarantees finding an optimal solution if it is implemented with both searches expanding nodes in the same way and if the costs are uniform or if the search algorithm is adapted to handle varying costs (e.g., using a priority queue similar to Uniform Cost Search).
- **Challenges with Non-uniform Costs**: In cases where the edge costs are non-uniform, special care must be taken to ensure that the meeting point between the two searches does not result in a suboptimal solution. This typically involves more complex coordination between the two searches.

Completeness

- **Guaranteed Solution**:
  - Bidirectional Search is complete, meaning it will find a solution if one exists, provided that the search space is finite and fully connected.
- **Handling Disconnected Graphs**: If the search space has disconnected components, Bidirectional Search might fail to find a solution. In such cases, the algorithm needs to ensure that the search directions can indeed meet.

BS Applications

DNA Sequence Alignment: In bioinformatics, Bidirectional Search can be used to align DNA sequences. By starting from both ends of the sequences, the algorithm can efficiently find the optimal alignment that minimizes the difference between the sequences, which is crucial in tasks like genome assembly or comparing genetic data.

Social Network Analysis: In social networks, Bidirectional Search can efficiently find the shortest path between two users. This is useful for determining degrees of separation, finding influencers, or analyzing the spread of information or diseases across a network.

Natural Language Processing (NLP): In NLP tasks like parsing or sentence diagramming, Bidirectional Search can be applied to efficiently match syntactic structures from both the start and end of a sentence, ensuring that grammatical rules are followed and reducing the complexity of processing long sentences.

Reverse Engineering: Bidirectional Search can be used in reverse engineering to match high-level descriptions of a system (like software specifications) with low-level implementations (like code). By searching from both the specification and the implementation, it can efficiently identify where discrepancies or optimizations occur.

Cultural Heritage and Archaeology: In reconstructing ancient texts or artifacts, Bidirectional Search can help match fragmented pieces. By starting the search from both known start points (beginning of a text or structure) and known endpoints (ending), researchers can efficiently piece together incomplete historical data.

Limitations of Bidirectional Search

1. Difficulty in Finding a Meeting Point

Limitation:

- **Synchronization Challenge:** One of the key challenges in Bidirectional Search is ensuring that the two searches (forward and backward) meet. In complex or irregular graphs, it may be difficult to identify a common node where the two searches intersect.

Scenarios Where This is Inefficient:

- **Irregular Graphs:** In graphs with an irregular structure, where nodes are not symmetrically connected, the forward and backward searches might explore vastly different parts of the graph before meeting, leading to inefficiencies.
- **Sparse Graphs:** In sparse graphs, where connections between nodes are limited, the searches may need to cover a large portion of the graph before they intersect, reducing the efficiency gains of bidirectional search.

2. Increased Memory Usage

Limitation:

- **Space Complexity:** While Bidirectional Search reduces the depth of the search, it still requires maintaining two separate frontiers (one for each direction). This can lead to high memory usage, especially in large or dense graphs.

Scenarios Where This is Inefficient:

- **Large Graphs:** In very large graphs, the memory required to store the frontiers from both directions can become substantial, potentially exhausting available resources.
- **Dense Graphs:** In graphs with a high branching factor, the number of nodes added to the frontier at each step can grow rapidly, leading to significant memory consumption as both frontiers expand.

3. Difficulty in Implementing in Weighted Graphs

Limitation:

- **Handling Edge Weights:** In weighted graphs, ensuring that Bidirectional Search remains optimal requires careful management of the priority queues for both searches. The algorithm needs to guarantee that the path cost is minimized when the searches meet, which complicates implementation.

Scenarios Where This is Inefficient:

- **Graphs with Varying Weights:** In graphs where edge weights vary significantly, managing the two priority queues to ensure that both searches proceed optimally can be complex and error-prone.
- **Graphs with Negative Weights:** If the graph contains negative weights, additional mechanisms are required to prevent infinite loops or suboptimal paths, further complicating the implementation.

4. Asymmetry in the Search Space

Limitation:

- **Uneven Exploration:** If the search space is asymmetrical, meaning that the start and goal nodes are not symmetrically located within the graph, one of the searches might expand far more nodes than the other. This uneven exploration can negate the efficiency gains of Bidirectional Search.

Scenarios Where This is Inefficient:

- **Asymmetric Graphs:** In graphs where the start and goal nodes are located in areas with very different branching factors or densities, one search might progress much faster than the other, leading to inefficiencies.
- **Skewed Search Spaces:** In environments like certain game maps or terrain models, where obstacles or barriers create asymmetry, one search might be forced to explore a much larger area, reducing the overall efficiency.

5. Not Suitable for All Graph Types

Limitation:

- **Applicability:** Bidirectional Search is most effective in unweighted graphs or graphs with uniform edge weights. In other types of graphs, particularly those with complex structures or varying costs, other algorithms may be more appropriate.

Scenarios Where This is Inefficient:

- **Graphs with Multiple Goals:** In scenarios where there are multiple goal nodes, Bidirectional Search may need to adjust its strategy or restart multiple times, leading to inefficiencies compared to algorithms specifically designed for multi-goal search problems.
- **Non-Uniform Graphs:** In graphs where edge weights vary widely or are not uniform, Bidirectional Search may struggle to efficiently manage the two searches, leading to inefficiencies compared to heuristic-based searches like A*.

6. Coordination and Management Complexity

Limitation:

- **Algorithmic Complexity:** Coordinating the two searches and ensuring they proceed in a balanced manner can be complex. Managing two sets of data structures, ensuring synchronization, and correctly handling the meeting point adds to the complexity of the implementation.

Scenarios Where This is Inefficient:

- **Real-Time Applications:** In real-time systems, such as robotics or online pathfinding, the overhead of managing two synchronized searches might introduce latency, making Bidirectional Search less suitable.
- **Distributed Systems:** In distributed environments where the search might be spread across multiple machines or processes, coordinating the two searches can introduce significant complexity and overhead, reducing efficiency.

7. Limited Improvement in Small or Simple Graphs

Limitation:

- **Marginal Gains:** In small or simple graphs, where the start and goal nodes are relatively close or the graph structure is straightforward, the benefits of Bidirectional Search may be minimal. The overhead of managing two searches might outweigh the performance gains.

Scenarios Where This is Inefficient:

- **Small Graphs:** In small graphs, the overhead of running two searches may not be justified, as a single BFS or DFS could find the solution with less complexity.
- **Simple Graphs:** In simple, well-connected graphs where paths between nodes are direct and obvious, Bidirectional Search might offer little advantage over traditional unidirectional searches.

Performance Summary

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit.

Superscript caveats are as follows: a complete if b is finite; b complete if step costs ≥ ϵ for positive ϵ; c optimal if step costs are all identical; d if both directions use breadth-first search.

AI Related Inefficiency

Natural Language Processing (NLP)

1.1 Language Parsing and Syntactic Analysis

- **Application:** In NLP, parsing sentences to understand their syntactic structure often involves exploring different possible parse trees.
- **Inefficiencies:**
  - **BFS and DFS:** Both can be inefficient when parsing sentences with ambiguous or complex structures. BFS might explore many irrelevant paths, leading to high memory consumption, while DFS might dive deeply into incorrect parses, wasting computational resources.
  - **DLS:** If the depth limit is set too low, DLS might miss correct parses that require deeper exploration, leading to incomplete or incorrect syntactic analysis.
  - **IDS:** Although it addresses some DFS limitations, IDS's repeated exploration of nodes can be inefficient in languages with long sentences or complex grammars.

1.2 Semantic Search and Question Answering

- **Application:** Semantic search engines and question-answering systems need to explore large knowledge graphs to retrieve the most relevant information.
- **Inefficiencies:**
  - **UCS:** In large knowledge graphs, UCS can become inefficient when the cost (relevance) of connections varies widely. It may explore many low-cost but irrelevant paths, leading to slow response times.
  - **BF:** Bidirectional Search could be inefficient in unstructured knowledge graphs where the start and goal nodes are not symmetrically located, making it hard to find a meeting point.

1.3 Machine Translation

- **Application:** In machine translation, search algorithms are used to explore possible translations by generating and evaluating different sentence structures.
- **Inefficiencies:**
  - **DFS:** DFS might generate long and complex sentence structures that are syntactically valid but semantically incorrect, leading to poor translation quality.
  - **BFS:** BFS could consume excessive memory by exploring all possible translations at each level, making it inefficient for real-time translation tasks.
  - **DLS:** If the translation requires deeper exploration (more complex sentence structures), an inappropriate depth limit in DLS might result in incomplete or inaccurate translations.

Computer Vision (CV)

2.1 Object Recognition and Scene Understanding

- **Application:** Object recognition tasks in CV often involve searching through different possible interpretations of an image to correctly identify objects.
- **Inefficiencies:**
  - **BFS:** In high-resolution images with many possible interpretations, BFS can be memory-intensive and slow, as it explores all possibilities level by level.
  - **DFS:** DFS might explore incorrect object interpretations deeply before backtracking, leading to inefficiencies in real-time applications like autonomous driving.
  - **DLS:** In complex scenes requiring deeper interpretation, DLS might miss correct identifications if the depth limit is insufficient.
  - **IDS:** While mitigating some DFS inefficiencies, IDS's repetitive exploration can still be slow in high-dimensional image spaces.

2.2 Image Segmentation

- **Application:** Image segmentation involves dividing an image into meaningful segments by exploring different pixel groupings.
- **Inefficiencies:**
  - **UCS:** UCS may struggle with varying costs (e.g., edge strength or color differences) between segments, leading to inefficiencies in finding the optimal segmentation.
  - **BF:** In large images, Bidirectional Search might be inefficient if the search spaces from different segments are asymmetrical, leading to imbalanced exploration.

2.3 Pathfinding for Object Tracking

- **Application:** Object tracking in CV involves following an object through a sequence of frames, which requires efficient pathfinding.
- **Inefficiencies:**
  - **BFS:** In complex environments with many possible object trajectories, BFS can quickly consume large amounts of memory, slowing down real-time tracking.
  - **DFS:** DFS might follow incorrect object trajectories deeply before backtracking, leading to poor tracking performance in fast-moving scenes.
  - **DLS:** A poorly chosen depth limit in DLS could result in incomplete tracking if the correct path lies beyond the limit.
  - **IDS:** Repeated node exploration in IDS can slow down tracking, especially in scenarios requiring quick adaptation to changing object movements.

Related Fields

3.1 Robotics (Path Planning and Navigation)

- **Application:** Robots often use search algorithms to plan paths and navigate through environments.
- **Inefficiencies:**
  - **BFS:** In large or cluttered environments, BFS can become memory-intensive and slow, as it explores all possible paths level by level.
  - **DFS:** DFS might lead robots into dead ends or inefficient paths, wasting time and energy.
  - **UCS:** UCS can be inefficient in environments where path costs vary widely, as it may explore many low-cost but suboptimal paths.
  - **DLS:** In complex environments, an incorrect depth limit in DLS could result in incomplete paths, causing the robot to fail in reaching its destination.
  - **IDS:** IDS's repetitive exploration of nodes can be inefficient in dynamic environments where quick pathfinding is necessary.

3.2 Game AI (Decision-Making and Strategy)

- **Application:** Game AI uses search algorithms to make decisions and plan strategies.
- **Inefficiencies:**
  - **BFS:** In games with a large state space, BFS might explore too many irrelevant states, consuming memory and slowing down decision-making.
  - **DFS:** DFS might pursue a poor strategy deeply before recognizing its inefficiency, leading to suboptimal gameplay.
  - **DLS:** DLS might fail to explore deep, effective strategies if the depth limit is set too low, resulting in weak AI performance.
  - **IDS:** The repeated exploration of states in IDS can slow down strategic planning in fast-paced games where decisions need to be made quickly.
  - **BF:** Bidirectional Search can be inefficient in asymmetric game state spaces, where one search direction may progress much faster than the other.

3.3 Automated Theorem Proving

- **Application:** Automated theorem proving involves exploring logical proofs to verify theorems.
- **Inefficiencies:**
  - **DFS:** DFS might explore long, complex proofs deeply before backtracking, leading to inefficiencies in finding shorter, more elegant proofs.
  - **BFS:** BFS could become memory-intensive when exploring large logical spaces, slowing down the proving process.
  - **DLS:** An inappropriate depth limit in DLS might cause it to miss valid proofs that require deeper exploration, leading to incomplete theorem proving.
  - **IDS:** The repeated exploration of proof steps in IDS can lead to inefficiencies in large logical spaces where quick verification is essential.

Informed Search

Search everywhere!!!

Planning and Scheduling

- **Example:** Automated planning for tasks in robotics, logistics, or project management.
- **Search Problem:** The initial state is the starting set of tasks, and the goal state is the completion of all tasks within constraints (time, resources). The search involves finding an optimal or feasible sequence of actions to achieve the goal.

Natural Language Processing (NLP)

- **Example:** Sentence generation, machine translation, or speech recognition.
- **Search Problem:** The initial state is the beginning of a sentence or phrase, and the goal state is the complete, correct sentence or translation. The search explores possible word sequences or grammar structures to form valid sentences.

Optimization Problems

- **Example:** Resource allocation, network design, or maximizing profits in business.
- **Search Problem:** The initial state is the current allocation or configuration, and the goal state is the optimal allocation that maximizes or minimizes a certain objective. The search explores possible configurations to find the optimal one.

Machine Learning Model Training

- **Example:** Hyperparameter tuning or feature selection.
- **Search Problem:** The initial state is the model with default parameters or features, and the goal state is the model with the best performance. The search explores different parameter values or feature combinations to optimize the model's performance.

General Tree Search Paradigm

```
function tree-search(root-node)
        frontier ← successors(root-node)
        while (notempty(frontier))
                { node ← remove-first(frontier)   // lowest f value
                        state ← state(node)
                        if goal-test(state) return solution(node)
                        frontier ← insert-all(successors(node),frontier) }
        return failure
end tree-search
```

General Graph Search Paradigm

```
function tree-search(root-node)
        frontier ← successors(root-node)
        explored ← empty
        while (notempty(frontier))
            { node ← remove-first(frontier)   // lowest f value
                state ← state(node)
                if goal-test(state) return solution(node)
                explored ← insert(node, explored)
                frontier ← insert-all(successors(node),frontier, if node not in explored)
            }
        return failure
end tree-search
```

Best-first Search

- A search strategy is defined by picking the order of node expansion
- Idea: use an evaluation function f(n) for each node
  - Estimate of “desirability”
  - Expand most desirable unexpanded node
- Implementation:
  - Order the nodes in frontier/fringe in decreasing order of desirability

Old Friends

- Breadth First
  - Best First
  - With f(n) = depth(n)
  - BFS considers nodes closer to the start as better, so f(n) could be seen as the depth of the node in the tree. Nodes at shallower depths (closer to the start) are explored first, which means f(n) is lower for these nodes.

- Uniform cost search
  - Best First
  - With f(n) = the sum of edge costs from start to n = g(n)

Three functions

- f(n) = is the evaluation function that best-first search uses to figure out which frontier node to expand
- g(n) = cost from the start node to the current node
- h(n) = guess of the distance/cost from current node to the goal node

f(n) - best node overall cumulatively

g(n) - closer node from the start

h(n) - how far is the goal from me

Informed Search Strategies

Informed search strategies in artificial intelligence (AI) refer to search algorithms that use additional information beyond the basic problem definition to guide the search process towards a solution more efficiently. This additional information typically comes in the form of a heuristic, which is a function that estimates the cost or distance from a given state to the goal state. The purpose of using a heuristic is to make the search process more directed and, ideally, faster by focusing on the most promising paths in the search space.

Key Characteristics of Informed Search Strategies:

1. **Heuristics**: The defining feature of informed search strategies is the use of heuristics. A heuristic is a rule of thumb or an educated guess that helps to estimate how close a current state is to the goal state. For example, in a pathfinding problem, the straight-line distance (Euclidean distance) to the goal can be used as a heuristic.
2. **Efficiency**: By using heuristics, informed search strategies aim to explore fewer nodes than uninformed strategies (like Breadth-First Search or Depth-First Search), making them more efficient in terms of time and memory.

3. **Goal-Directed Search**: Informed search strategies are more goal-directed because they use the heuristic to prioritize which paths to explore first, typically focusing on those that seem most likely to lead to a solution.

4. **Optimality and Completeness**: Some informed search algorithms, are both optimal (they find the best possible solution) and complete (they will find a solution if one exists), provided the heuristic used is admissible (never overestimates the cost to reach the goal).

Heuristics

A **heuristic** is a rule of thumb, a practical method, or a shortcut that helps in decision-making, problem-solving, or discovery. It is not guaranteed to be perfect or optimal, but it is often effective for reaching a satisfactory solution in a reasonable amount of time. In AI, heuristics are used to speed up the process of finding a good solution, especially when the problem space is vast and exhaustive search would be computationally expensive or infeasible.

Characteristics of Heuristics:

1. **Simplification**: Heuristics simplify complex problems by focusing on the most relevant aspects, ignoring less critical details.
2. **Efficiency**: They provide quick, approximate solutions that are "good enough" for the task at hand, even if they are not optimal.
3. **Domain-Specific**: Heuristics are often tailored to specific problems or domains. For example, a heuristic for a chess-playing AI might involve evaluating board positions based on the value of the pieces.
4. **Guidance**: In search algorithms, heuristics guide the search process by estimating which paths are more promising, reducing the number of states that need to be explored.

Heuristics Function

A **heuristic function** (often denoted as `h(n)`) is a specific type of heuristic used in search algorithms. It is a function that estimates the cost or distance from a given node (or state) `n` in the search space to the goal state. The heuristic function helps to prioritize which nodes should be explored next based on their estimated proximity to the goal.

Properties of Heuristic Functions:

1. **Admissibility**:
   - A heuristic function is **admissible** if it never overestimates the true cost to reach the goal from any node. In other words, `h(n)` is always less than or equal to the actual lowest cost to reach the goal.
   - Admissible heuristics are crucial in ensuring that search algorithms like A* find the optimal solution.
2. **Consistency (or Monotonicity)**:
   - A heuristic function is **consistent** if, for every node `n` and every successor `n'` of `n`, the estimated cost of reaching the goal from `n` is no greater than the cost of reaching `n'` plus the estimated cost of reaching the goal from `n'`. Mathematically, this means: $h(n) \leq c(n,n') + h(n')$ where `c(n, n')` is the actual cost of moving from node `n` to node `n'`.
   - Consistency ensures that the heuristic function never decreases along a path, which helps in simplifying the implementation of search algorithms.
3. **Informedness**:
   - The **informedness** of a heuristic function refers to how accurately it estimates the true cost to the goal. A more informed heuristic provides estimates closer to the actual costs, making the search more efficient by reducing unnecessary exploration of non-promising paths.

Examples of Heuristic Functions

Manhattan Distance: In grid-based pathfinding problems, such as navigating through a maze, the Manhattan distance (sum of the absolute differences in the x and y coordinates) between the current node and the goal is a common heuristic. It is admissible and often used in algorithms like A*.

$$h(n) = |x_2 - x_1| + |y_2 - y_1|$$

Euclidean Distance: In problems where the cost to move from one point to another is based on straight-line (as-the-crow-flies) distance, the Euclidean distance (the direct distance between two points) can be used as a heuristic.

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Examples of Effective Heuristics

Pathfinding in Grid-Based Environments

- **Heuristic:** Manhattan Distance
- **Why It's Effective:** The Manhattan distance provides a heuristic for grid-based movement where diagonal moves are not allowed. It is simple to compute and provides a good estimate of the actual path cost.

Sliding Puzzle (8-puzzle, 15-puzzle)

- **Heuristic:** Number of Misplaced Tiles or Manhattan Distance
- **Why It's Effective:** Both heuristics provide estimates for the minimum number of moves required to solve the puzzle. The number of misplaced tiles heuristic is simple, while the Manhattan distance gives a more refined estimate by accounting for how far each tile is from its goal position.

Traveling Salesman Problem (TSP)

- **Heuristic:** Minimum Spanning Tree (MST)
- **Why It's Effective:** The MST heuristic estimates the lower bound on the cost of completing the tour by connecting all cities without returning to the start.

Real-World Examples of Heuristics

Navigational Heuristics (GPS and Mapping Applications):

- **Example:** When using a GPS to find the shortest route from your current location to a destination, the system might use the straight-line (Euclidean) distance between your location and the destination as a heuristic.
- **Purpose:** This heuristic helps the GPS prioritize routes that move closer to the destination, even though the actual travel path might involve turns and detours.

Medical Diagnosis:

- **Example:** A doctor might use heuristics when diagnosing a patient based on symptoms. For instance, if a patient has a sore throat, fever, and swollen lymph nodes, the doctor might quickly lean towards diagnosing strep throat based on these common indicators.
- **Purpose:** This heuristic allows the doctor to make a preliminary diagnosis quickly, which can then be confirmed with further testing.

Online Search Algorithms:

- **Example:** Search engines like Google use heuristics to rank web pages. For instance, pages that contain the exact keywords being searched for are ranked higher because they are more likely to be relevant.
- **Purpose:** This heuristic helps the search engine return the most relevant results quickly, improving the user experience.

Shopping and Decision-Making:

- **Example:** When shopping online, a heuristic might be to choose a product with the highest number of positive reviews. While not a guarantee of quality, this heuristic simplifies the decision-making process by using crowd wisdom.
- **Purpose:** This approach helps consumers make quicker decisions without analyzing every available option in detail.

Heuristics in Chess (Game AI):

- **Example:** In chess, a common heuristic is to control the center of the board. While not a strict rule, it's generally advantageous to control the center, leading to better mobility and control over the game.
- **Purpose:** This heuristic helps players (and AI) make strategic decisions more quickly by focusing on a key aspect of gameplay rather than calculating every possible move.

Software Development (Rule of Thumb for Bug Fixes):

- **Example:** A common heuristic in software development is that the earlier a bug is found and fixed, the cheaper it is to address. Therefore, developers prioritize writing tests and catching bugs during the development phase rather than after deployment.
- **Purpose:** This heuristic helps in managing resources effectively by addressing issues at a stage where they are easier and less costly to fix.

Types of Informed Search Strategies

1. Greedy Best-First Search
2. A* Search
3. Bidirectional A* Search
4. Iterative Deepening A* Search
5. Memory-Bounded A* Search
6. Weighted A* Search
7. Recursive Best-First Search
8. Beam Search

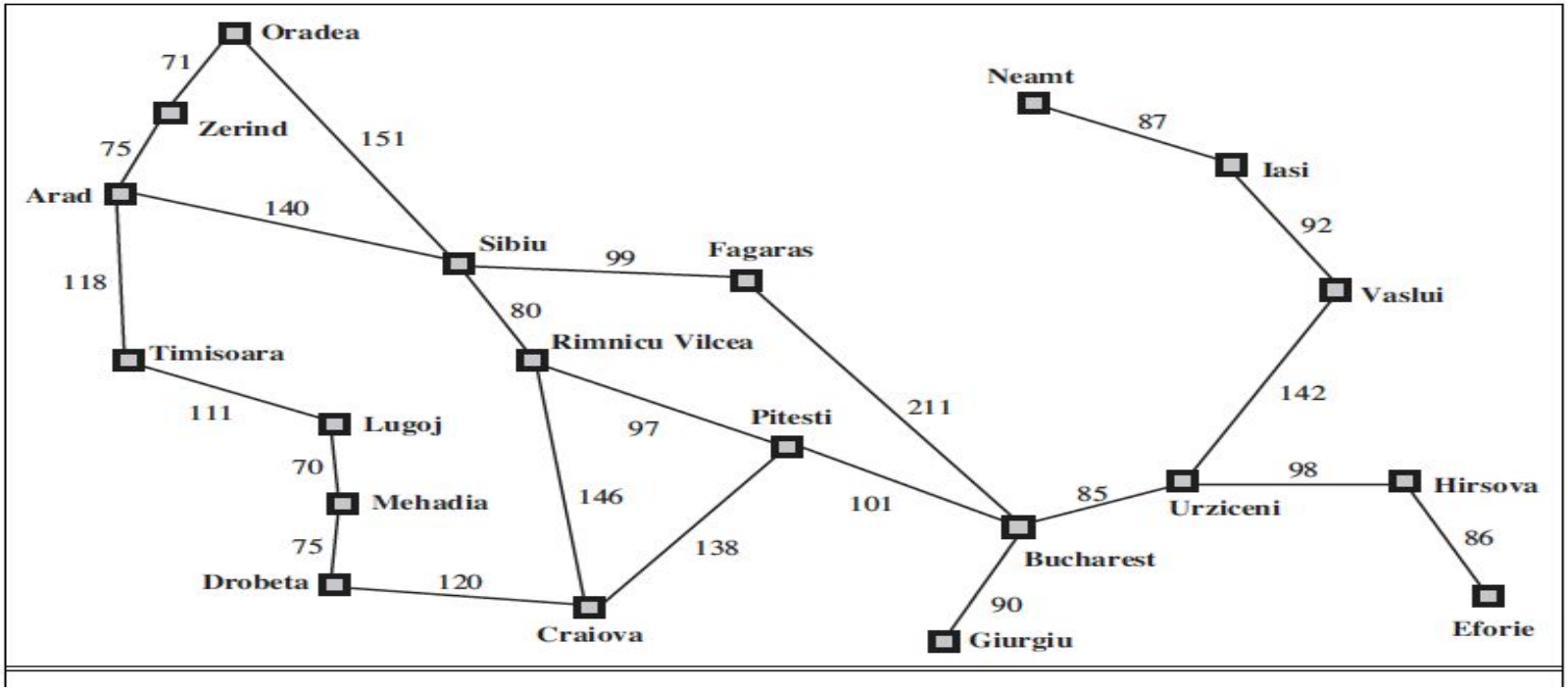
Greedy Best-First Search

Greedy Best-First Search (GBFS) is an informed search algorithm that aims to find the most promising path to a goal by expanding the node that appears closest to the goal based solely on a heuristic function. Unlike other search algorithms that consider both the cost to reach a node and the estimated cost to reach the goal, Greedy Best-First Search focuses only on the heuristic estimate, making it "greedy" in its approach.

Here, f(n) = h(n).

How Greedy Best-First Search Works

1. **Initialization**:
   - Start with an initial node, usually the starting state of the problem.
   - Add this node to a priority queue (often implemented as a min-heap) where nodes are prioritized based on their heuristic value `h(n)`.
2. **Node Expansion**:
   - Extract the node with the lowest heuristic value from the priority queue (this is the "greedy" part, as it selects the node that seems closest to the goal).
   - If this node is the goal, the search ends successfully.
   - If not, expand the node by generating all its successors (i.e., possible states that can be reached from the current state).
3. **Adding Successors to the Queue**:
   - For each successor, compute its heuristic value `h(n)` and add it to the priority queue.
   - The priority queue is then updated to ensure that the node with the lowest `h(n)` will be expanded next.
4. **Repeat**:
   - Continue expanding nodes by repeatedly extracting the node with the lowest `h(n)` and adding its successors to the queue until the goal is found or the queue is empty (indicating that no solution exists).



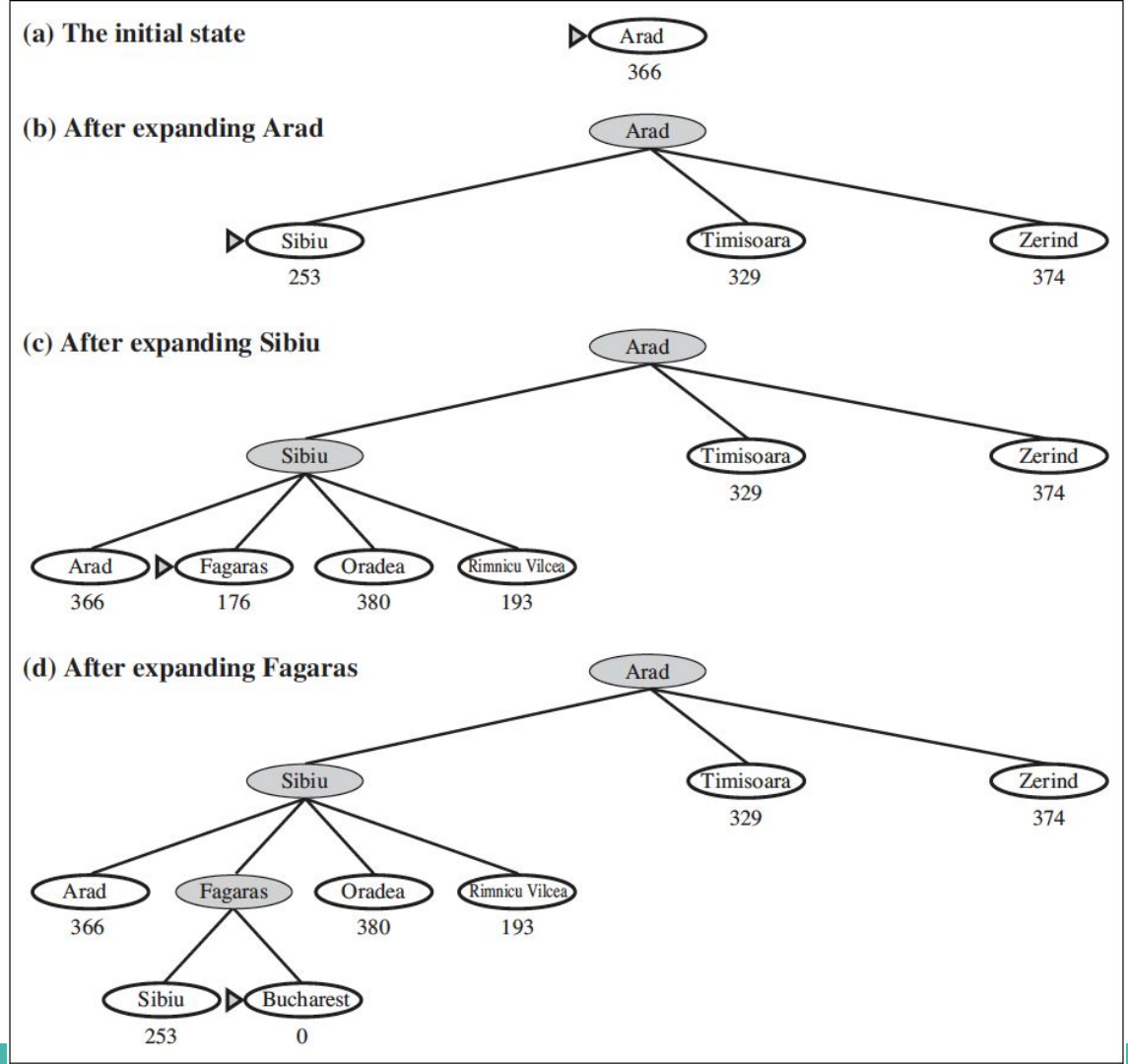
A simplified road map of part of Romania

Values of h_{SLD}—straight-line distances to Bucharest

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

Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.



Properties of Greedy Best-First Search

1. **Heuristic-Driven**: GBFS relies entirely on the heuristic function `h(n)` to make decisions. It does not consider the actual cost to reach a node, making it greedy.
2. **Not Optimal**: Since GBFS only considers the heuristic and ignores the actual cost to reach nodes (`g(n)`), it is not guaranteed to find the optimal solution. It may choose a path that looks promising (based on the heuristic) but turns out to be suboptimal.
3. **Not Complete**: GBFS is not complete, meaning it may fail to find a solution even if one exists, especially if the heuristic misguides the search into dead ends or cycles in case of tree-search paradigm, but complete in case of graph-search paradigm.
4. **Efficiency**: GBFS can be faster than algorithms like A* because it often expands fewer nodes. However, this speed comes at the cost of potentially finding suboptimal or incomplete solutions.
5. **Memory Usage**: GBFS stores all generated nodes in memory, like A*. However, because it doesn't keep track of the actual cost `g(n)`, it might require less memory than A* in some cases.

Advantages of Greedy Best-First Search

- **Speed**: GBFS can be very fast, especially in problems where the heuristic is well-chosen and closely approximates the true cost to the goal.
- **Simplicity**: The algorithm is relatively simple to implement, focusing solely on the heuristic to guide the search.

Limitations of Greedy Best-First Search

1. Suboptimal Solutions

- **Description:** Greedy Best-First Search is not guaranteed to find the shortest or optimal path to the goal. It focuses solely on the heuristic value, ignoring the actual cost to reach a node from the start.
- **Example in AI:** In a pathfinding AI, GBFS might find a path that looks promising based on the heuristic (e.g., straight-line distance), but it could turn out to be much longer than the optimal path due to obstacles or higher costs not accounted for by the heuristic.

2. Susceptibility to Local Minima

- **Description:** GBFS can get "stuck" in local minima, where it chooses a path that seems best based on the heuristic but is not actually the best overall path. This happens because it doesn’t backtrack or explore alternative paths once a promising-looking route is chosen.
- **Example in AI:** In game AI, GBFS might choose a series of moves that seem to bring the player closer to a win (based on a heuristic), but the strategy leads to a dead end or a poor overall position because it doesn't consider the broader game context.

3. Heuristic Dependence

- **Description:** The effectiveness of GBFS is highly dependent on the quality of the heuristic. If the heuristic is poorly chosen or misleading, the algorithm's performance can degrade significantly, leading to inefficient searches or incorrect solutions.
- **Example in AI:** In natural language processing (NLP) tasks like sentence generation, a heuristic that poorly estimates the likelihood of a phrase fitting into a sentence context can lead to grammatically incorrect or nonsensical outputs.

4. Incomplete Search

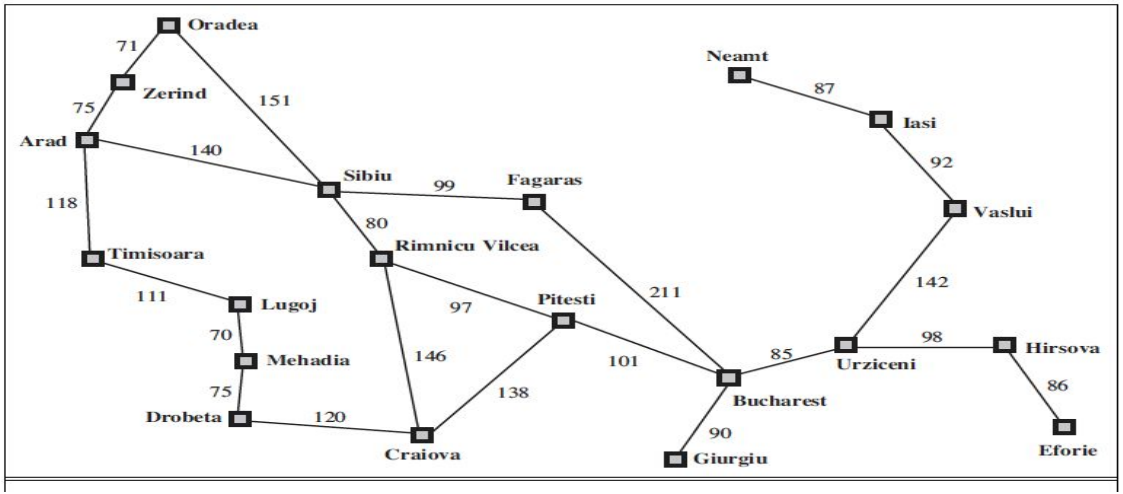
- **Description:** GBFS is incomplete in certain cases, meaning it may fail to find a solution even if one exists. This can occur in large or infinite search spaces where the algorithm might get stuck in a loop or continue exploring suboptimal paths indefinitely.
- **Example in AI:** In complex planning tasks like robot navigation, GBFS might endlessly explore a series of moves that seem promising but fail to reach the destination because it doesn’t explore alternative routes that may be longer initially but lead to the goal more effectively.

5. Lack of Global Awareness

- **Description:** GBFS lacks global awareness since it only considers the immediate heuristic value without accounting for the overall structure of the problem. This can lead to inefficient exploration of the search space, especially in problems where the solution requires understanding the broader context or making trade-offs.
- **Example in AI:** In search-based problem-solving like puzzle-solving (e.g., Sudoku), GBFS might focus on filling one row or column without considering how this will affect the overall solution, potentially leading to conflicts later that are hard to resolve.

In Greedy Best First Search

- We are ignoring the cost to the current path and only looking to the extra cost to give to the goal
- In GBFS, we don't consider g(n)
- f(n) = h(n)
- Is this ideal?



A simplified road map of part of Romania

- Our objective is to go from the start state to the goal, not from the goal.
- The state ***n*** is somewhere in the middle, but our objective is to go from the start state to the goal and
- as we are taking the minimum, we should not myopically only take the minimum concerning how much cost we are going to pay in the future, but also take minimum over the cost that we have paid so far.

A* Search

A* is a widely used search algorithm in computer science, particularly for pathfinding and graph traversal problems. It is an informed search algorithm that balances the need to find the shortest path (optimality) with the need to find it efficiently (speed).

Idea: **avoid expanding paths that are already expensive**

The Combined Cost Function

At the heart of the A* algorithm is the combined cost function ***f(n)***, which determines which node to expand next:

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

Where:

- **g(n):** The actual cost from the start node to the current node n.
- **h(n):** The heuristic estimate of the cost from the current node n to the goal node. This is an estimate based on domain-specific knowledge.
- **f(n):** The total estimated cost of the cheapest solution that passes through the current node nnn, combining both the known cost to reach nnn and the estimated cost from nnn to the goal.

The A* algorithm efficiently balances the exploration of nodes by considering both the actual cost incurred so far and an estimate of the remaining cost to reach the goal:

Role of g(n) (Actual Cost):

- **Accurate Path Cost:**
  - The g(n) term ensures that the algorithm accounts for the real cost of reaching a particular node from the start node. This means that paths that have lower cumulative costs are favored, helping to ensure that the algorithm does not choose paths that are deceptively cheap in the short term but expensive overall.
- **Accumulation of Costs:**
  - g(n) is calculated by summing the costs of all edges along the path from the start node to the current node n. As the algorithm progresses, g(n) accumulates, representing the true cost of the path taken so far.

Role of h(n) (Heuristic Estimate):

- **Guidance Toward the Goal:**
  - The heuristic h(n) provides an estimate of the remaining cost to reach the goal. It guides the search process by directing it toward nodes that appear closer to the goal, thus reducing the number of nodes that need to be explored.
- **Influence on Efficiency:**
  - The quality of the heuristic h(n) greatly affects the efficiency of the A* algorithm. A good heuristic (one that closely approximates the true remaining cost) will speed up the search by focusing on the most promising paths. However, if the heuristic is poor, the algorithm may end up exploring unnecessary nodes, reducing efficiency.

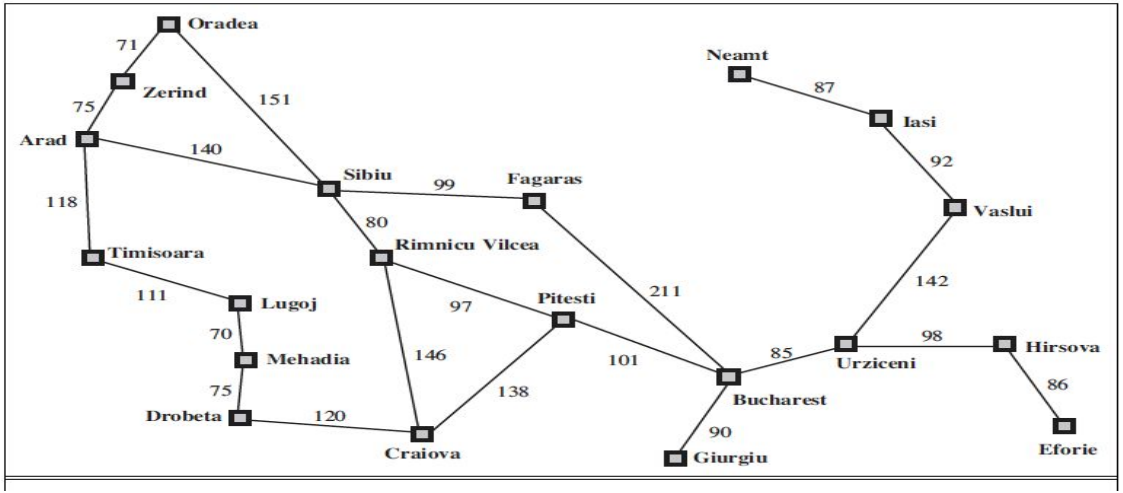
The Balance:

- **Optimality vs. Speed:**
  - A* achieves a balance between g(n) and h(n) by combining them into the f(n) function. This combination allows A* to find the shortest path efficiently by considering both the actual cost of reaching a node and an estimate of the cost to complete the path.
  - The algorithm prioritizes nodes with lower f(n) values, effectively balancing exploration of new paths (guided by h(n)) and exploitation of known paths (tracked by g(n)).
- **Admissibility and Consistency:**
  - For A* to be both optimal and complete, the heuristic h(n) must be admissible (never overestimating the true cost to reach the goal) and consistent (ensuring that the estimated cost from any node to the goal is less than or equal to the cost of reaching a neighbor plus the estimated cost from the neighbor to the goal).

It is not only looking at h, which is greedy best-first search. It is not only looking at g, which is a uniform cost search

A* for Romanian Shortest Path

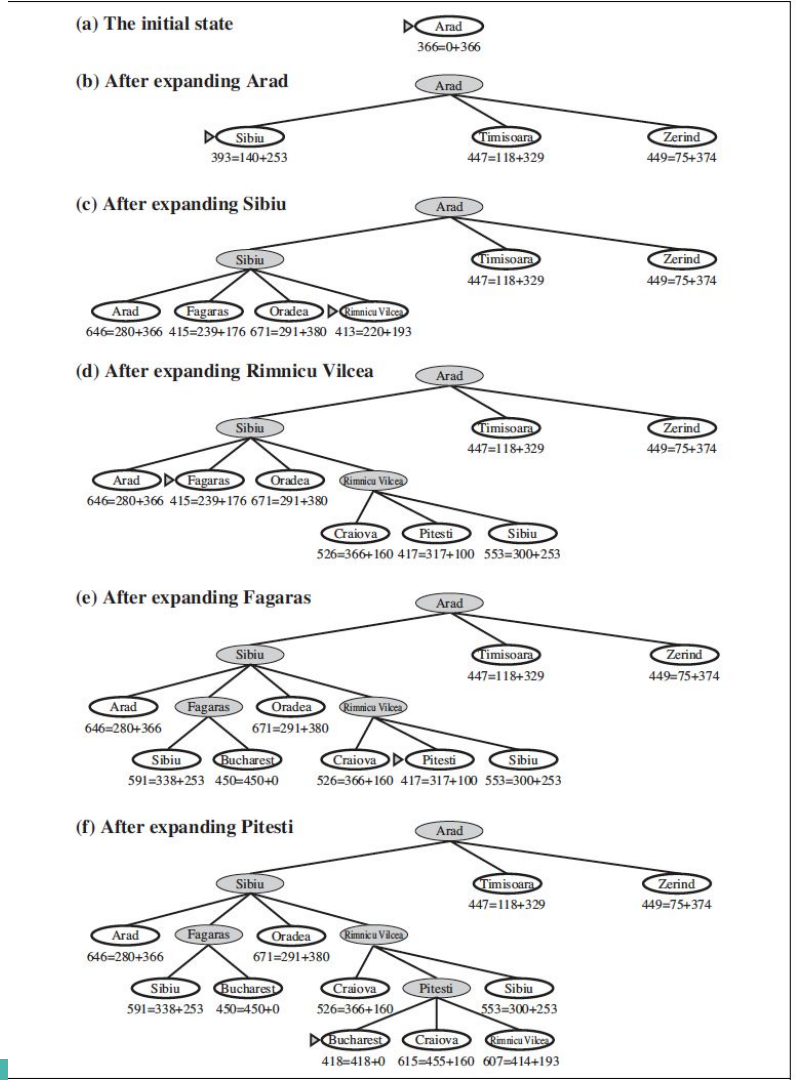
Romanian Map



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

Values of h_{SLD}—straight-line distances to Bucharest

Stages in an A* search for Bucharest. Nodes are labeled with f = g +h. The h values are the straight-line distances to Bucharest



Algorithm Steps

Step 1: Initialize the Open and Closed Lists

- **Open List:** This is a priority queue that contains the nodes to be explored. The nodes in the open list are sorted based on their f(n)=g(n)+h(n) value, where g(n) is the actual cost from the start node to node n, and h(n) is the heuristic estimate of the cost from n to the goal.
- **Closed List:** This is a set that contains the nodes that have already been explored and should not be revisited.

Step 2: Add the Start Node to the Open List

- Initialize g(start)=0 because the cost to reach the start node from itself is zero.
- Calculate h(start), the heuristic estimate of the cost from the start node to the goal.
- Set f(start) = g(start) + h(start).
- Add the start node to the open list.

Step 3: Repeat Until the Goal is Reached or the Open List is Empty

While the open list is not empty, perform the following steps:

Step 3.1: Select the Node with the Lowest f(n) Value

- Remove the node n from the open list that has the lowest f(n) value.
- If n is the goal node, the algorithm terminates, and the path is reconstructed from the goal to the start node by following parent pointers.

Step 3.2: Generate Successors

- For each neighbor (successor) of the current node n:
  - Calculate g(neighbor) = g(n) + cost(n,neighbor), where cost(n,neighbor) is the actual cost to move from n to the neighbor.
  - Calculate h(neighbor), the heuristic estimate of the cost from the neighbor to the goal.
  - Set f(neighbor)=g(neighbor)+h(neighbor).

Step 3.3: Check if the Neighbor Should be Added to the Open List

- If the neighbor is already in the closed list, skip it.
- If the neighbor is not in the open list, add it and set the current node n as its parent.
- If the neighbor is in the open list but the new path to it has a lower g(neighbor) value, update its g(neighbor) and f(neighbor) values and change its parent to the current node n.

Step 3.4: Add the Current Node to the Closed List

- After all neighbors have been evaluated, add the current node n to the closed list to ensure it is not revisited.

Step 4: Reconstruct the Path

- If the goal node was reached, reconstruct the path by following the parent pointers from the goal node back to the start node.

Goodness of Heuristic Functions

- Admissibility
- Consistency

Admissible Heuristics: Definition of Admissibility

A heuristic function $h(n)$ is said to be **admissible** if it never overestimates the true cost of reaching the goal from any node n in the search space.

Mathematically, this is expressed as:

$$h(n) \leq h^*(n) \quad \text{for all nodes } n$$

Where:

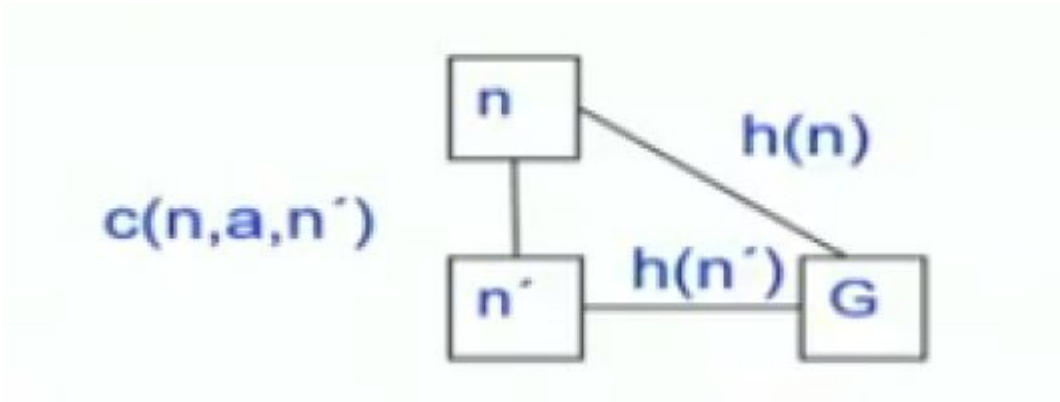
- $h(n)$ is the heuristic estimate of the cost to reach the goal from node n.
- $h^*(n)$ is the actual cost of the optimal path from node n to the goal.

- An admissible heuristic is **Optimistic**
- Example: h_{SLD} (never overestimates the actual road distance)
- What is most optimistic?
  - What will be the heights of optimism for the Node?
  - The Node will not need to search for the Goal (I am goal) as it is goal itself, right?
  - In other words, h will be equal to 0 if I am the goal, then the distance to myself is 0.
  - So the most optimistic heuristic would be 0.
  - However, what would be the meaning of optimism? The meaning of optimism would be that the cost to the goal is estimated as less than what it is. You feel that I will have to pay less cost. Then you end up paying. That is the meaning of admissibility.
- If, however, you are in a place where you want to make money and higher is better, what is an admissible heuristic one that makes you believe that you will earn more, not less.
- Admissible heuristic is not always that “less is optimal”. It is less than or less than equal to optimal for minimization problems, but it is greater than equal to optimal for a maximization problem.

- **Theorem**: If h(n) is admissible, then the Tree-Search version of A* is optimal.
- The interesting thing is that if it's the graph search A* version, then admissibility is insufficient.
- It is a necessary condition but not a sufficient condition.
- When would the graph search version of A* be optimal?
- New condition is needed called **consistency**

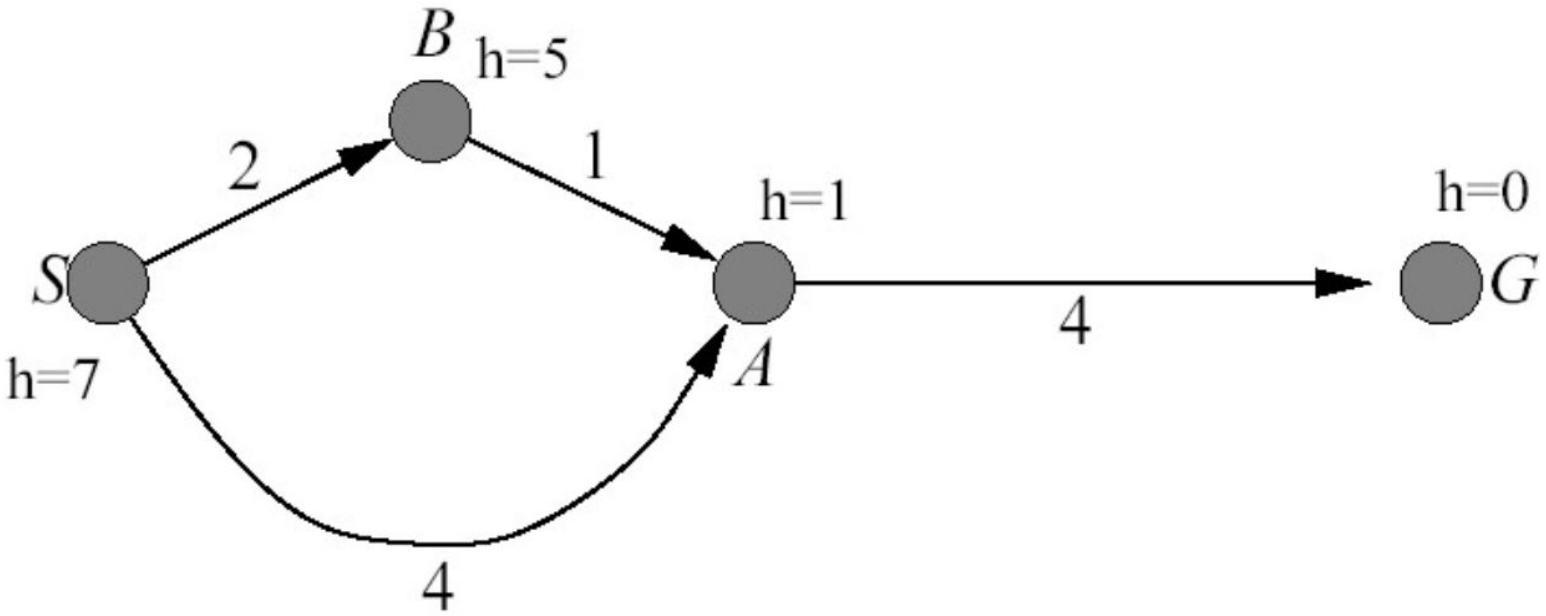
Consistent Heuristics (Triangle Inequality)

- h(n) is consistent if
  - for every node **n**
  - for every successor **n'** due to legal action **a**
  - **h(n) <= c(n,a,n') + h(n')** (triangle inequality)



- Every consistent heuristic is also admissible, i.e, consistency subsumes admissibility
- Theorem: If h(n) is consistent, A* using Graph-Search is optimal.

Example: Graph-Search version of A* having Admissible Heuristics



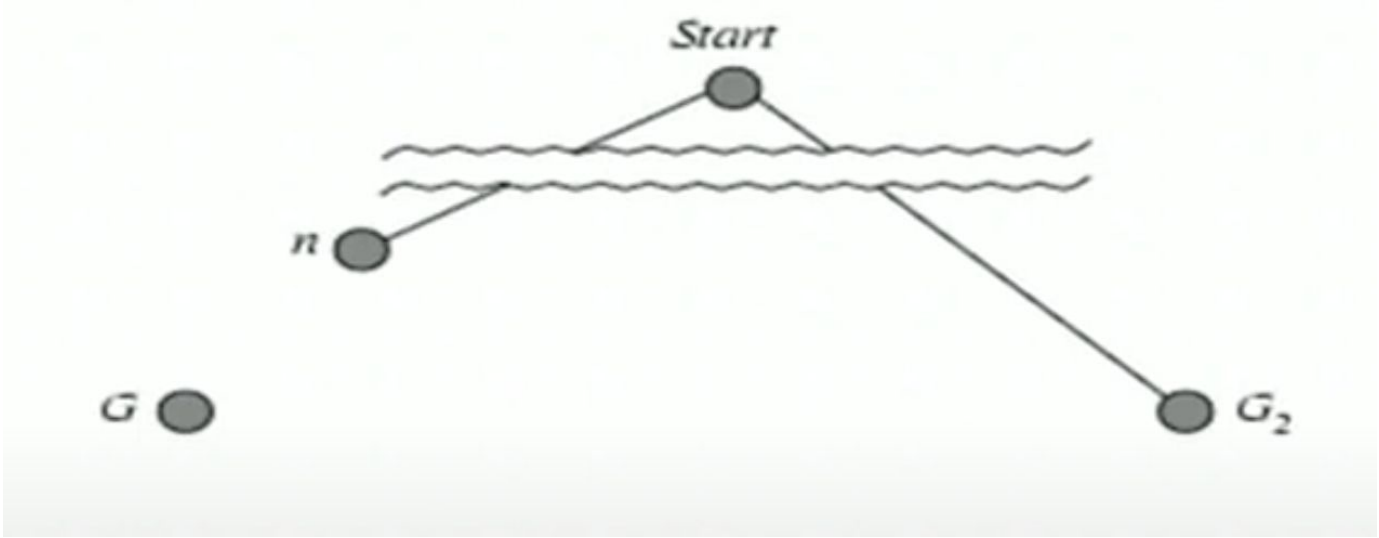
https://stackoverflow.com/questions/25823391/suboptimal-solution-given-by-a-search

Proof of Optimality of A* (Tree Version)

- We are trying to look at how to prove optimality for the A^* algorithm when the heuristic is admissible.
- Assume ***h()*** is admissible
- Say some sub-optimal goal state G_2 has been generated and is on the frontier and we are about to remove it.
- Currently, we have a start node ***S*** we are about to remove G_2 from the fringe.
- We believe that the algorithm should finish, and we have found the optimal path.

- Let us say that it is not the optimal path, i.e., there is another path which is the optimal path.
- If it is another path, it would have it's goal node, so let us say the optimal goal now that we are looking for is G.
- And either G may be in the fringe or some other node to that path has to be in the fringe at this point.
- Let ***n*** be an unexpanded state such that ***n*** is on the optimal path to the optimal goal ***G***.

- Let us say that path is this one in the diagram, and let us say that we have to have some node in the fringe that leads us to this goal ***G***.
- We chose to remove G_2 and not remove n.
- So if the algorithm was right, it should have removed ***n***, but the algorithm tried to remove G_2
- That means, for some reason, the algorithm thought that G_2 was the right node to remove, which means its F value of G_2 was less than the F value of ***n***.



Let's focus on G_2,

- $f(G_2) = g(G_2)$ [since $h(G_2) = 0$]
- $g(G_2) > g(G)$ [since G_2 is suboptimal]

Now focus on G,

- $f(G) = g(G)$ [since $h(G) = 0$]
- $f(G_2) > f(G)$ [substitution in above 3 equations]

Now let's focus on ***n***,

- $h(n) \leq h^*(n)$ [since h is admissible]
- $g(n) + h(n) \leq g(n) + h^*(n)$ [algebra]
- $f(n) = g(n) + h(n)$ [by definition]
- $f(G) = g(n) + h^*(n)$ [by assumption]
- $f(n) \leq f(G)$ [substitution using $f(G_2) > f(G)$]

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion (therefore it is a contradiction)

Proof of Completeness

Theorem: A* is complete, meaning it will find a solution if one exists, provided the search space is finite and the heuristic h(n) is admissible.

Proof:

- **Assumption:** The search space is finite, and the heuristic h(n) is admissible, i.e., $h(n) \leq h^*(n)$ for all nodes n, where $h^*(n)$ is the true cost from n to the goal.
- **Proof by Contradiction:**
  - Suppose A* is not complete, i.e., it fails to find a solution even though one exists.
  - This would imply that A* exhausts all nodes in the open list without finding the goal, which means the open list eventually becomes empty.
  - For A* to fail, it must be the case that all possible paths to the goal have been pruned or ignored.
  - However, because the heuristic h(n) is admissible, A* will not ignore any potential path that could lead to the goal. The admissible heuristic ensures that A* explores all feasible paths until the goal is found, as the estimated cost $f(n) = g(n) + h(n)$ will always lead A* to explore paths that can reach the goal.
- **Conclusion:** Since A* will eventually explore every possible path that could lead to the goal, it must find the goal if one exists, proving that A* is complete.

Properties of A* Search

1. **Optimality**:
   - A* guarantees finding the least-cost path if the heuristic h(n) is admissible and consistent. This means it never overestimates the actual cost to reach the goal.
2. **Completeness**:
   - A* is complete; it will find a solution if one exists, provided there is enough memory to keep track of all nodes.
3. **Efficiency(Time)**:
   - A* can be very efficient if a good heuristic is used, leading to fewer nodes being expanded compared to uninformed search strategies. In worst case, all nodes are expanded.
4. **Memory Usage(Space)**:
   - A* stores all generated nodes in memory, which can lead to high memory consumption, especially in large search spaces.

Advantages of A* Search

1. **Optimality and Completeness**: A* guarantees finding the optimal solution, making it a reliable choice for many applications.
2. **Flexibility**: A* can be adapted with different heuristic functions to fit various problem domains.
3. **Wide Applicability**: It is widely used in pathfinding, game development, robotics, and various optimization problems.

Disadvantages of A* Search

1. **Memory Intensive**: A* can require significant memory to store all generated nodes, particularly in large or complex search spaces.
2. **Performance Depends on Heuristic**: If the heuristic is poor or poorly tuned, A* can behave like a breadth-first search, resulting in inefficient performance.
3. **Computational Overhead**: The overhead of calculating and maintaining g(n) and h(n) values can add to the computational cost.

Bidirectional A* Search

Bidirectional A* Search is an extension of the A* search algorithm that seeks to improve search efficiency by simultaneously exploring paths from both the start node and the goal node. This approach aims to reduce the search space and the number of nodes that need to be expanded, making it especially useful in large search spaces.

Key Features of Bidirectional A* Search

1. **Two Search Frontiers**: The algorithm maintains two open lists: one for the forward search (from the start node) and one for the backward search (from the goal node).
2. **Meeting Point**: The searches continue until the two frontiers meet, at which point the algorithm can construct a solution path.
3. **Heuristic Function**: Like A*, Bidirectional A* uses heuristic functions to guide both the forward and backward searches, ensuring that both searches are informed.

How Bidirectional A* Search Works

1. **Initialization**:
   - Start with two open lists: one for the forward search from the start node and one for the backward search from the goal node.
   - Initialize the cost values g and heuristic values h for both searches.
2. **Node Expansion**:
   - Perform the following steps until one of the open lists is empty:
     - Expand the node with the lowest f(n)=g(n)+h(n) from the forward search.
     - If this node is found in the backward search's open list, a path has been discovered.
     - If not, generate its successors and add them to the forward open list.
3. **Simultaneous Search**:
   - In the same iteration, expand the node with the lowest f(n) from the backward search.
   - If this node is found in the forward search's open list, a path has been discovered.
4. **Meeting Point**:
   - When a node from either search connects to a node from the other search (i.e., the same state is found), a solution path can be constructed.
   - The total cost of the solution can be computed by combining the costs from both searches.
5. **Path Construction**:
   - Trace back from the meeting node to construct the complete path from the start node to the goal node by combining the paths from both searches.

Properties of Bidirectional A* Search

1. **Efficiency**:
   - By searching from both the start and goal simultaneously, Bidirectional A* can significantly reduce the search space, especially in large graphs or grids. It effectively narrows down the number of nodes expanded.
2. **Optimality**:
   - Bidirectional A* is guaranteed to find the optimal solution as long as both searches use admissible heuristics.
3. **Completeness**:
   - The algorithm is complete; it will find a solution if one exists, provided there is sufficient memory.
4. **Memory Usage**:
   - Bidirectional A* requires maintaining two open lists, which can increase memory usage compared to a unidirectional A* search. However, it may still use less memory overall since it expands fewer nodes.

Advantages of Bidirectional A* Search

1. **Reduced Search Time**: The simultaneous search can lead to faster solutions, particularly in large state spaces.
2. **Improved Time Complexity:** The reduced search space often translates into a reduction in time complexity. By expanding fewer nodes, Bidirectional A* can reach the goal faster than unidirectional A*.
3. **Potential for Faster Convergence:** Since each search frontier only needs to reach approximately halfway to the goal, the chances of the two frontiers meeting quickly are higher. This can lead to faster discovery of the optimal path.
4. **Balanced Exploration:** By balancing exploration between the forward and backward searches, Bidirectional A* can prevent the algorithm from getting "stuck" in certain parts of the search space. If one search direction becomes less promising, the other can compensate.

Limitations of Bidirectional A*

Heuristic Consistency and Symmetry:

- The success of Bidirectional A* heavily depends on the heuristics being consistent and symmetric (i.e., $h_f(n)=h_b(n)$). If the heuristics are not well-matched, one of the searches may dominate, reducing the benefit of bidirectional search.

Meeting Point Detection:

- Detecting the exact meeting point where the two searches should merge can be complex, especially in cases where the search frontiers do not meet at a single node but rather in a region of nodes.

Space Complexity:

- Although bidirectional search reduces time complexity, the space complexity remains high. The algorithm still requires storing all expanded nodes in both directions, which can be memory-intensive.

Implementation Complexity:

- Implementing bidirectional search is more complex than unidirectional search. Managing two separate frontiers, ensuring heuristic consistency, and correctly merging the paths add to the implementation difficulty.

Path Quality in Weighted Graphs:

- In weighted graphs, ensuring that the path found by bidirectional A* is truly optimal can be challenging if the forward and backward costs are not carefully managed. The weights might distort the balance between the two searches.

Iterative Deepening A* Search

Iterative Deepening A* (IDA*) is an informed search algorithm that combines the benefits of A* search and iterative deepening search. It is particularly useful for problems where the search space is large and the depth of the solution is not known in advance. IDA* systematically explores paths to a certain depth, using a heuristic to guide the search, while iteratively increasing the depth limit until a solution is found.

Key Features of Iterative Deepening A* Search

1. **Combination of Depth-First and Best-First Search**: IDA* uses a depth-first search approach to explore the search space while incorporating a heuristic to evaluate the cost to reach the goal.
2. **Memory Efficiency**: IDA* uses significantly less memory compared to A*, as it does not store all generated nodes in memory. Instead, it only keeps track of the current path.
3. **Optimality**: IDA* is guaranteed to find the optimal solution if the heuristic used is admissible.
4. **Completeness**: IDA* is complete; it will find a solution if one exists, provided that the search space is finite.

How Iterative Deepening A* Search Works

IDA* follows a systematic approach to explore the search space. The main components include:

1. **Evaluation Function**:
   - Similar to A*, IDA* uses an evaluation function f(n) defined as:

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

   - Where:
     - g(n): The actual cost from the start node to node n.
     - h(n): The heuristic estimate of the cost from node n to the goal.
2. **Depth-Limited Search**:
   - The algorithm conducts a depth-first search with a depth limit defined by the current threshold. This threshold is incrementally increased after each complete iteration.
3. **Threshold Management**:
   - The algorithm starts with a threshold based on the heuristic value of the start node, f(start)=h(start).
   - If a node exceeds this threshold during the search, the algorithm returns and increments the threshold, repeating the process until a solution is found.
4. **Recursive Exploration**:
   - During each iteration, the algorithm recursively explores nodes up to the current depth limit. If a node is reached that exceeds the threshold, it backtracks and tries other nodes.
   - Nodes that lead to a solution are recorded, allowing the algorithm to construct the solution path once the goal is found.

Steps of Iterative Deepening A* Search

1. **Initialization**:
   - Start with the initial node (the starting state).
   - Set the initial threshold to f(start) (this can be zero or based on h(start)).
   - Initialize the solution variable to keep track of the best cost found.
2. **Depth-Limited Search**:
   - Perform a depth-first search with the current threshold:
     - Expand the node with the lowest f(n) value.
     - If the goal node is reached, update the solution and store the cost.
     - If a node's f(n) exceeds the threshold, backtrack and return the threshold value.
3. **Update Threshold**:
   - After completing the depth-limited search for the current threshold:
     - If a solution is found, the algorithm terminates.
     - If not, increment the threshold to the lowest f(n) value of the nodes that exceeded the threshold during the search and repeat the depth-limited search.
4. **Repeat**:
   - Continue this process, incrementing the threshold until a solution is found.

Properties of Iterative Deepening A* Search

1. **Optimality**:
   - IDA* guarantees finding the optimal solution if the heuristic is admissible and consistent.
2. **Completeness**:
   - The algorithm is complete, meaning it will find a solution if one exists within a finite search space.
3. **Memory Efficiency**:
   - Unlike A*, IDA* does not need to store all nodes, making it suitable for large search spaces.
4. **Performance**:
   - IDA* may perform better than A* in some scenarios, especially when the solution depth is shallow compared to the size of the search space.

Advantages of Iterative Deepening A* Search

1. **Memory Usage**: IDA* uses much less memory than A*, making it feasible for large problems.
2. **Guaranteed Optimality**: The algorithm guarantees finding the optimal solution when using an admissible heuristic.
3. **Simplicity**: IDA* can be easier to implement than maintaining a priority queue in A*.

Limitations of IDA*

Repeated Work:

- A major drawback of IDA* is that it performs repeated work across iterations. Nodes near the root of the search tree are expanded multiple times, which can significantly increase the time complexity compared to A*.

Performance on Complex Heuristics:

- For problems where the heuristic function is complex or expensive to compute, the repeated evaluations in each iteration can make IDA* inefficient.

Lack of Path Reconstruction:

- Unlike A*, which can reconstruct the path as it searches, IDA* only finds the path after reaching the goal. This can make it more difficult to provide intermediate solutions or progress updates.

Threshold Management:

- Managing the threshold updates can be tricky, particularly in problems with varying step costs. If the increments are too small, many unnecessary iterations may occur; if too large, the search may miss the optimal solution.

Memory-Bounded A* Search

Memory-Bounded A* search algorithms are designed to address the high memory requirements of the standard A* algorithm by limiting the amount of memory used during the search process. The basic idea is to retain the benefits of A* (such as finding optimal paths) while ensuring that the algorithm can run even when memory is limited.

Key Concepts

1. **Memory Limitation:**
   - Unlike A*, which keeps all generated nodes in memory, Memory-Bounded A* algorithms impose a limit on the number of nodes that can be stored at any given time. This is crucial in environments where memory is a critical constraint.
2. **Node Pruning:**
   - When the memory limit is reached, Memory-Bounded A* algorithms must prune or discard nodes to make room for more promising ones. The choice of which nodes to prune is key to maintaining optimality and efficiency.
3. **Re-expansion of Nodes:**
   - If a pruned node turns out to be necessary for finding the optimal path, Memory-Bounded A* algorithms may need to re-expand it. This re-expansion can lead to increased computational effort, but it's necessary to ensure that the best path is found.

Variants of Memory-Bounded A*

1. MA* (Memory-Bounded A*)

MA* is an early form of Memory-Bounded A* that introduces a simple mechanism for handling memory constraints:

- **Memory Limit:** MA* keeps track of the nodes in memory and discards the least promising ones when the memory limit is reached. The nodes are typically evaluated based on their ***f(n)=g(n)+h(n)*** value, where ***g(n)*** is the cost to reach the node and ***h(n)*** is the heuristic estimate to reach the goal.
- **Node Pruning:** When pruning is necessary, MA* removes the nodes with the highest ***f(n)*** values (i.e., the least promising nodes). These nodes might be needed later, so their best ***f*** values are stored as backup.
- **Re-expansion:** If the search later determines that a pruned node should be revisited (e.g., because it led to a potentially better solution), MA* must re-expand that node. This can result in increased time complexity due to the repeated expansion of nodes.
- **Optimality:** MA* can find the optimal solution if given enough time, but the need for re-expansion can lead to inefficiencies.

2. SMA* (Simplified Memory-Bounded A*)

SMA* improves upon MA* by offering a more sophisticated approach to managing memory:

- **Memory Management:** SMA* maintains a fixed amount of memory and uses it to store the most promising nodes, based on their ***f(n)*** values. Unlike MA*, SMA* tries to minimize the need for re-expansion by keeping track of backup paths.
- **Pruning Strategy:** When the memory limit is reached, SMA* removes the least promising nodes, similar to MA*. However, SMA* also keeps track of the best alternative path for each pruned node, which reduces the need for re-expansion.
- **Backtracking:** SMA* can backtrack when it encounters a dead end or when all remaining paths have ***f(n)*** values higher than those of previously pruned nodes. This backtracking allows SMA* to explore alternative paths without significantly increasing memory usage.
- **Optimality:** SMA* is optimal, meaning it will find the best solution within the given memory constraints. It is designed to handle the memory limit more efficiently than MA*, leading to better performance in practice.

How Memory-Bounded A* Works

1. **Initialization:**
   - The algorithm starts by initializing the open list (frontier) with the start node, just like A*.
   - The memory limit is defined, which restricts the number of nodes that can be stored in the open list at any time.
2. **Node Expansion:**
   - Nodes are expanded based on their f(n)=g(n)+h(n) value. The node with the lowest f(n) is expanded first, following the same principle as A*.
3. **Memory Check:**
   - After expanding a node, the algorithm checks whether the number of nodes in memory exceeds the predefined limit.
   - If the memory limit is not reached, the algorithm continues expanding nodes.
4. **Pruning:**
   - When the memory limit is reached, the algorithm must prune nodes to make room for more promising ones.
   - The least promising node (the one with the highest f(n) value) is pruned first.
   - The pruned node's f value is stored in case the node needs to be re-expanded later.
5. **Backtracking (SMA):***
   - If all paths from the current node lead to dead ends or paths with higher f(n) values than those of pruned nodes, SMA* will backtrack to explore alternative paths.
   - SMA* can re-expand pruned nodes if necessary, but it tries to minimize this by keeping track of backup paths.
6. **Goal Test:**
   - The algorithm continues expanding and pruning nodes until it finds the goal node.
   - When the goal is found, the path from the start node to the goal is reconstructed.

Limitations of Memory-Bounded A*

Increased Time Complexity:

- The need to prune and potentially re-expand nodes can lead to increased computational effort, making Memory-Bounded A* slower in some cases compared to standard A*.

Implementation Complexity:

- Managing memory and ensuring optimality within constrained memory limits can be complex, requiring careful implementation.

Suboptimality in MA:*

- MA* may not always find the optimal path if re-expansion is not handled correctly, leading to potential suboptimal solutions.

Weighted A* Search

Weighted A* is a variant of the standard A* search algorithm that introduces a trade-off between optimality and search efficiency. By weighting the heuristic function, Weighted A* can prioritize faster searches at the expense of finding the absolute shortest path. This approach is particularly useful in situations where finding a solution quickly is more important than finding the optimal solution.

Key Concepts

1. **Heuristic Weighting:**
   - In standard A*, the evaluation function is f(n)=g(n)+h(n, where g(n) is the cost to reach node n from the start node, and h(n) is the heuristic estimate of the cost from n to the goal.
   - In Weighted A*, the evaluation function is modified to f′(n)=g(n)+w×h(n), where w is a weight factor greater than 1.
   - The weight w amplifies the heuristic function, making the search algorithm more "greedy" by placing more emphasis on estimated distance to the goal rather than the cost incurred so far.
2. **Trade-off Between Optimality and Speed:**
   - By increasing the weight w, Weighted A* focuses more on reaching the goal quickly, potentially at the expense of bypassing the optimal path.
   - When w=1, Weighted A* is equivalent to standard A* and will find the optimal path if the heuristic is admissible.
   - When w>1, the path found may be suboptimal, but the search may complete faster by exploring fewer nodes.
3. **Bounded Suboptimality:**
   - Weighted A* guarantees that the cost of the path found will not exceed w times the cost of the optimal path. This means that the solution is within a known factor of the optimal solution, providing a controlled trade-off between search time and path quality.

A* search:	$g(n)+h(n)$	$(W=1)$
Uniform-cost search:	$g(n)$	$(W=0)$
Greedy best-first search:	$h(n)$	$(W=\infty)$
Weighted A* search:	$g(n)+W\times h(n)$	$(1<W<\infty)$

How Weighted A* Works

1. **Initialization:**
   - Begin with the start node S and calculate its heuristic estimate h(S).
   - Set the initial threshold as $f'(S)=g(S) + w \times h(S)$.
2. **Node Expansion:**
   - Similar to standard A*, expand the node with the lowest $f'(n)$ value, where $f'(n)=g(n)+w \times h(n)$.
   - Add the successors of the current node to the open list (frontier).
3. **Goal Test:**
   - If the goal node G is reached, the search stops, and the path from S to G is reconstructed.
4. **Termination:**
   - The algorithm continues expanding nodes until the goal is reached or no more nodes can be expanded.

Advantages of Weighted A*

1. **Faster Search:**
   - By increasing the weight w, Weighted A* can reach the goal faster, making it particularly useful in real-time applications or when quick decisions are required.
2. **Controlled Suboptimality:**
   - Weighted A* provides a way to control the trade-off between search speed and path optimality. By adjusting the weight w, users can decide how much they are willing to compromise on path quality to gain speed.
3. **Flexibility:**
   - The ability to tune the weight www allows Weighted A* to be adapted to different scenarios, balancing between speed and optimality depending on the needs of the application.
4. **Same Implementation Framework:**
   - Weighted A* can be implemented using the same framework as standard A*, with minimal modifications, making it easy to integrate into existing systems.

Limitations of Weighted A*

Suboptimal Paths:

- The primary drawback of Weighted A* is that it may produce suboptimal paths, especially as w increases. This can be problematic in applications where optimality is crucial.

Heuristic Dependence:

- The effectiveness of Weighted A* heavily depends on the quality of the heuristic. If the heuristic is poor, increasing the weight might lead to exploring misleading paths, reducing the overall efficiency.

Bounding Factor:

- While the path cost is bounded by w × optimal cost, in practice, the actual suboptimality might be closer to this upper bound, particularly with high weights.

Risk of Missing Optimal Paths:

- In some search spaces, emphasizing the heuristic too much might cause Weighted A* to miss shorter but less obvious paths, leading to significantly suboptimal solutions.

Real-life applications of A* search

Air Traffic Management:

- **Application:** A* is used in air traffic management systems to optimize flight paths, helping avoid collisions and manage airspace efficiently.
- **Example:** A* can help in planning optimal flight paths for aircraft to avoid no-fly zones, minimize fuel consumption, and ensure timely arrivals.

Autonomous Vehicles:

- **Application:** A* is used in the path planning systems of autonomous vehicles, helping them navigate safely through urban environments.
- **Example:** Self-driving cars use A* to plan routes that avoid obstacles, follow traffic rules, and adapt to changing road conditions. time.

Natural Language Processing (NLP):

- **Application:** A* can be used in NLP tasks that require searching through possible sequences, such as in parsing sentences or finding the most likely sequence of words.
- **Example:** In speech recognition, A* can help find the most likely word sequence that matches the given sound input.

Virtual Assistants and Chatbots:

- **Application:** A* can be employed in virtual assistants and chatbots to determine the best sequence of actions or responses based on user input and context.
- **Example:** A chatbot may use A* to determine the most appropriate sequence of responses to guide a user through a troubleshooting process or a decision-making tree.

Medical Diagnosis Systems:

- **Application:** A* can be used in decision support systems for medical diagnosis, helping to find the most probable diagnosis or treatment path based on symptoms and medical history.
- **Example:** A medical expert system might use A* to suggest a series of diagnostic tests that lead to an accurate diagnosis in the shortest possible time.

Recursive Best-First Search

Recursive Best-First Search (RBFS) is an informed search algorithm that combines the benefits of best-first search and depth-first search while maintaining a low memory footprint. It is designed to address the memory limitations of traditional best-first search algorithms, such as A*, by using recursion and a limited memory strategy to explore paths.

Key Features of Recursive Best-First Search

1. **Memory Efficiency**: RBFS uses memory more efficiently than traditional best-first search algorithms like A*, as it only retains a single path from the root to the current node, along with some additional information about the best alternative paths.
2. **Optimality**: RBFS can be optimal if it uses an admissible heuristic function (i.e., one that does not overestimate the cost to reach the goal).
3. **Completeness**: RBFS is complete; it will find a solution if one exists in a finite search space.
4. **Recursive Nature**: The algorithm is inherently recursive, simplifying its implementation and reducing the need for explicit stack management.

How Recursive Best-First Search Works

RBFS operates using a recursive function that expands nodes based on their estimated cost while managing a limited memory footprint. The main components of the algorithm include:

1. **Evaluation Function**:
   - Similar to A* and other best-first search algorithms, RBFS uses an evaluation function:

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

   - Where:
     - g(n): The cost from the start node to node n.
     - h(n): The heuristic estimate of the cost from node n to the goal.
2. **Recursive Search**:
   - RBFS recursively explores nodes by keeping track of the current node's cost and the best alternative path cost.
   - The recursive function calls itself to explore child nodes, maintaining the path from the start to the current node.
3. **Path and Cost Management**:
   - Each recursive call keeps track of the current best path and its cost. If the best path cost exceeds a threshold, the algorithm backtracks and explores alternative paths.
4. **Backtracking**:
   - If a leaf node is reached or a node's f(n) exceeds the current best cost, the algorithm backtracks to explore other nodes, updating the best cost as necessary.

Steps of Recursive Best-First Search

1. **Initialization**:
   - Start with the initial node (the starting state).
   - Set an initial threshold based on the heuristic value f(start)=h(start).
2. **Recursive Exploration**:
   - Call the recursive function with the current node:
     - Expand the node by generating its successors.
     - For each successor, calculate its f(n) value.
     - If a successor is the goal, return the path and cost.
     - If the best successor's f(n) exceeds the threshold, backtrack and update the threshold.
3. **Threshold Management**:
   - Keep track of the lowest f(n) value among all successors.
   - If a better path is found, update the best path and continue searching.
4. **Repeat**:
   - Continue the recursive process until a solution is found or all nodes have been explored.

Properties of Recursive Best-First Search

1. **Optimality**:
   - RBFS guarantees optimality if the heuristic is admissible and the cost function is consistent.
2. **Completeness**:
   - The algorithm is complete, meaning it will find a solution if one exists.
3. **Memory Usage**:
   - RBFS uses less memory compared to traditional best-first search algorithms because it does not store all nodes, just the current path and some additional information.
4. **Performance**:
   - The performance of RBFS can be comparable to A* in some scenarios, particularly when the heuristic is well-designed.

Advantages of Recursive Best-First Search

1. **Low Memory Usage**: RBFS is particularly useful in situations where memory is a constraint, as it only retains the current path.
2. **Simple Implementation**: The recursive nature of RBFS simplifies the implementation of the algorithm, avoiding explicit stack management.
3. **Flexible Heuristic Use**: RBFS can be adapted with different heuristics to fit various problems.

Disadvantages of Recursive Best-First Search

1. **Repeated Node Expansions**: RBFS may explore some nodes multiple times, especially in cases where paths are revisited.
2. **Time Complexity**: While RBFS uses less memory, it may take longer to find solutions due to its recursive nature and potential repeated explorations.
3. **Dependence on Heuristic Quality**: The efficiency of RBFS heavily relies on the quality of the heuristic function used.

Memory Bounded A* vs RBFS

Memory Management:

- **SMA***: Operates within a fixed memory limit, pruning least promising nodes when memory is full.
- **RBFS**: Uses linear memory relative to the search depth, only keeps current path and best alternative path.

Node Pruning:

- **SMA***: Prunes nodes when memory is full and stores the best f-value of pruned nodes for possible regeneration.
- **RBFS**: Does not prune; instead, it discards all but the current path and the best alternative, leading to potential re-expansion of nodes.

Backtracking:

- **SMA***: Backtracks by re-expanding pruned nodes based on stored f-values, balancing memory use with exploration.
- **RBFS**: Backtracks by revisiting the best alternative path, potentially causing redundant re-expansions without memory of previously explored paths.

Optimality:

- **SMA***: Attempts to maintain optimality within memory limits but may sacrifice optimality if key nodes are pruned.
- **RBFS**: Can find optimal solutions if the heuristic is admissible, but frequent backtracking might lead to inefficiency.

Efficiency:

- **SMA***: More efficient in large, complex spaces due to its memory management strategy, which reduces unnecessary re-expansions.
- **RBFS**: Less efficient due to potential repeated work during backtracking, as it lacks memory of pruned nodes.

Implementation Complexity:

- **SMA***: More complex due to the need for memory management and node regeneration mechanisms.
- **RBFS**: Simpler to implement as it primarily focuses on depth-first search with limited memory overhead.

Beam Search

Beam Search is a heuristic search algorithm that is often used in scenarios where the search space is vast, such as natural language processing (NLP), machine translation, speech recognition, and other AI applications. It is a breadth-first search algorithm but with a limited memory footprint. Instead of exploring all possible paths like in standard breadth-first search, Beam Search keeps track of only the most promising paths at each level of the search tree, defined by a fixed beam width.

Key Concepts

1. **Beam Width:**
    - The beam width is a parameter that determines the number of top candidates (nodes) that the algorithm keeps at each level of the search tree.
    - For example, if the beam width is set to 3, Beam Search will keep only the top 3 most promising paths at each level.

2. **Breadth-First Search with Pruning:**

   - Beam Search explores the search tree level by level, similar to breadth-first search.
   - However, instead of keeping all nodes at each level, it only retains a fixed number (equal to the beam width) of the most promising nodes, discarding the rest.

3. **Evaluation Function:**

   - The algorithm uses an evaluation function (often similar to the one used in A*) to rank the nodes. Typically, the evaluation function is f(n)=g(n)+h(n), where:
     - g(n) is the cost to reach node n.
     - h(n) is the heuristic estimate of the cost from nnn to the goal.
   - In cases like language generation, the evaluation function might be based on the likelihood of generating the next word or token.

4. **Pruning Less Promising Paths:**

   - At each level of the search tree, Beam Search selects the top candidates based on the evaluation function.
   - Paths that do not rank within the top k (where k is the beam width) are discarded.

5. **Termination:**

   - The search continues until it reaches a predefined stopping condition, such as finding a goal state, reaching a maximum depth, or completing a fixed number of iterations.

How Beam Search Works

1. **Initialization:**
   - Start with the initial node, often called the root node. The algorithm calculates its evaluation function and adds it to the list of active candidates.
2. **Expand Candidates:**
   - From the list of current candidates (initially just the root node), expand each candidate by generating its successors (children nodes).
3. **Evaluate and Prune:**
   - For each successor, calculate the evaluation function.
   - Sort the successors based on their evaluation scores.
   - Select the top k (beam width) successors and prune the rest.
4. **Repeat:**
   - Treat the selected successors as the new list of current candidates and repeat the process of expansion, evaluation, and pruning.
5. **Termination:**
   - The algorithm continues this process until it either reaches the goal state or satisfies another termination condition (e.g., reaching a certain depth in the search tree).

Key Characteristics and Use Cases

Memory Efficiency:

- Beam Search is more memory-efficient than exhaustive search methods like breadth-first search because it only retains a fixed number of paths (equal to the beam width) at each level.

Scalability:

- Beam Search scales well in large search spaces, such as language generation or sequence prediction tasks, because it limits the number of paths explored.

Controlled Complexity:

- The beam width k allows control over the trade-off between search thoroughness and computational complexity. A larger beam width means more paths are considered, increasing the chances of finding a better solution but at the cost of higher computational resources.

Differences from Other Search Algorithms

- **Beam Search vs. Breadth-First Search:**
  - **Breadth-First Search**: Explores all nodes at each level of the search tree, leading to high memory usage.
  - **Beam Search**: Prunes all but the top k nodes at each level, reducing memory usage and making the search more manageable in large spaces.
- **Beam Search vs. A:***
  - **A***: Keeps all nodes in memory and always expands the most promising node based on a combination of path cost and heuristic (optimal but memory-intensive).
  - **Beam Search**: Focuses on keeping only a fixed number of promising paths, sacrificing optimality for efficiency.
- **Beam Search vs. Greedy Best-First Search:**
  - **Greedy Best-First Search**: Expands nodes based solely on the heuristic, potentially ignoring the true cost to reach a node.
  - **Beam Search**: Can be configured to balance between the heuristic and the path cost, but with a focus on maintaining a limited set of candidates.

Limitations

1. **Suboptimal Solutions:**
    - Beam Search is not guaranteed to find the optimal solution because it might prune paths that could lead to a better outcome later on.
2. **Heuristic Dependence:**
    - The performance of Beam Search heavily depends on the quality of the evaluation function. A poorly designed heuristic can lead to suboptimal or even incorrect solutions.
3. **Limited Exploration:**
    - By restricting the number of paths considered at each step, Beam Search may miss better solutions that would be found with a broader or exhaustive search.

Applications

1. Machine Translation: Beam Search is a core component in sequence-to-sequence models used in machine translation systems, such as Google Translate. When translating a sentence from one language to another, the model generates possible translations for each word or phrase, and Beam Search is used to explore the most promising translation paths.

2. Speech Recognition: In speech recognition systems like those used by virtual assistants (e.g., Siri, Google Assistant), Beam Search is used to find the most likely sequence of words given a sequence of audio inputs. The algorithm processes the audio data to generate possible word sequences, then uses Beam Search to select the most likely sequence based on a language model.

3. Text Generation and Summarization: In generative models like GPT, Beam Search is used to generate coherent and contextually appropriate text by selecting the best sequence of words or tokens. This is also applied in automatic text summarization, where the system generates a concise summary from a longer document.

4. Path Planning in Robotics: Beam Search is used in robotics for path planning, where a robot must navigate from a start position to a goal while avoiding obstacles. The algorithm evaluates multiple possible paths at each step and selects the most promising ones to explore further.

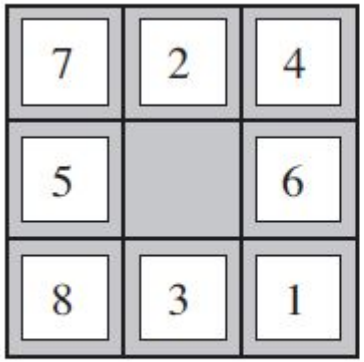
5. DNA Sequence Alignment: In bioinformatics, Beam Search is used for aligning DNA sequences to identify regions of similarity, which can indicate functional, structural, or evolutionary relationships between sequences.

6. Dialogue Systems and Chatbots: In dialogue systems, Beam Search is used to generate the most appropriate response in a conversation by exploring multiple possible replies and selecting the one that best matches the context of the conversation.

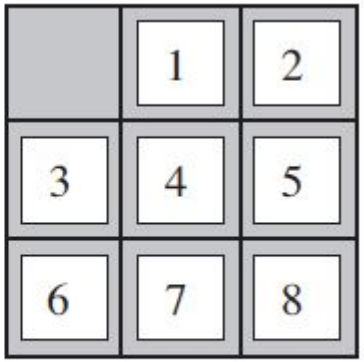
Summary of Benefits Over Other Search Algorithms:

- **Efficiency:** Beam Search is more efficient than exhaustive search methods because it prunes less promising paths early, reducing computational complexity.
- **Memory Usage:** It uses significantly less memory than algorithms like A* since it only retains a fixed number of paths (defined by the beam width) at each level of the search tree.
- **Quality:** While it may sacrifice some optimality compared to exhaustive searches, Beam Search often provides better results than greedy algorithms by considering multiple possibilities at each step.
- **Scalability:** Beam Search is scalable to large problems, making it suitable for real-world applications where search spaces can be vast, such as language processing, speech recognition, and robotics.

Admissible Heuristics: An Example



Start State



Goal State

$h_1(n)$ = number of misplaced tiles

$h_1(S)$ = ?

$h_1(S)$ = 8

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)

$h_2(S)$ = ?

$h_2(S)$ = 3+1+2+2+2+3+3+2 = 18

Heuristic Property: Dominance

Dominance occurs when one heuristic is consistently more effective than another in estimating the true cost to reach the goal. Specifically, a heuristic h_1 is said to **dominate** another heuristic h_2 if, for every possible state n, the estimate provided by $h_1(n)$ is closer to the actual cost $h^*(n)$ of reaching the goal than the estimate provided by $h_2(n)$.

Mathematically, this can be expressed as: $h_1(n) \geq h_2(n)$ for all states n, and $h_1(n)$ is closer to $h^*(n)$ than $h_2(n)$.

Dominance and Admissibility:

1. **Admissibility** of a heuristic means that the heuristic never overestimates the true cost to reach the goal (i.e., $h(n) \leq h^*(n)$ for all n).
    - If h_1 dominates h_2, and both heuristics are admissible, then h_1 is guaranteed to be more effective in terms of the search process, as it will guide the search more directly towards the goal, potentially expanding fewer nodes.
2. **Example of Dominance:**
    - **Heuristic Example:**
        - Suppose we have two heuristics for a pathfinding problem on a grid:
            - $h_1(n)$ = straight-line distance from the current node to the goal.
            - $h_2(n)$ = Manhattan distance (the sum of the absolute differences in the horizontal and vertical distances to the goal).
        - In this case, h_1 might dominate h_2 because the straight-line distance is always equal to or greater than the Manhattan distance (assuming all movements cost the same). Thus, h_1 gives a more accurate estimate of the true cost in a direct path scenario.
    -

Why Dominance Matters

- **Search Efficiency:**
  - Using a dominant heuristic can significantly reduce the number of nodes explored during the search process, leading to faster solution times and less computational effort.
- **Algorithm Performance:**
  - In practice, dominance is an essential property to consider when choosing or designing heuristics for a particular problem. A heuristic that dominates others can make the difference between a search algorithm that performs efficiently and one that is prohibitively slow.
- **Combining Heuristics:**
  - In some cases, if h_1 dominates h_2, a combined heuristic such as $h(n) = \max(h_1(n), h_2(n))$ can be used, which still preserves admissibility while benefiting from the best features of both heuristics.

Relaxed Problems

Relaxed Problem:

- A relaxed problem is a modified version of an original problem where some of the constraints are either loosened or removed entirely. The relaxation typically results in a problem that is easier to solve or analyze.
- The solution to the relaxed problem provides an estimate or a bound that can be used as a heuristic in solving the original, more complex problem.

They are critically important in computing admissible heuristics. Why?

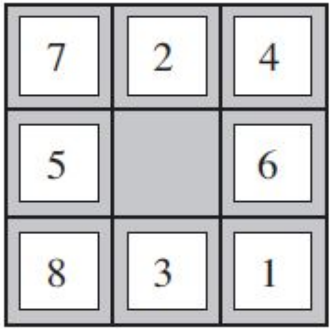
Purpose of Relaxation:

- The primary purpose of creating a relaxed problem is to derive a heuristic that can guide a search algorithm efficiently towards the solution of the original problem.
- Relaxed problems are used to generate admissible heuristics, which never overestimate the true cost of reaching the goal in the original problem.

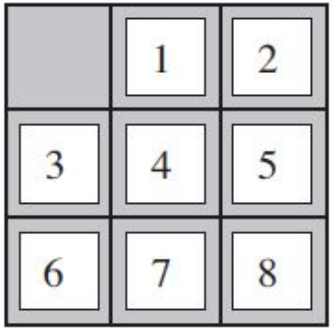
Example: Imagine we have to plan the best route for a delivery truck that has to stop at many different places. The original problem is very complex because there are a lot of rules to follow, like which roads are open, traffic conditions, and the order of deliveries.

- Now, let's say we simplify the problem by ignoring some of those rules. For example, you pretend that there's no traffic or that the truck can take any road, even if it's closed. This simplified version of the problem is what we call a "relaxed problem."
- When you solve this relaxed problem, you're essentially looking at all possible ways to solve the puzzle, including some that wouldn't work in the real world. Because you're considering more options, the solution you find in this relaxed world will either be cheaper (in terms of time, distance, or cost) or at least no more expensive than the best possible solution to the original, harder problem.
- This is useful because the relaxed problem can give you a good estimate or starting point for solving the original problem. It helps you figure out a strategy that might not be perfect but gets you close to the best solution faster.

8 Puzzle: Relaxation



Start State



Goal State

The 8-Puzzle Problem:

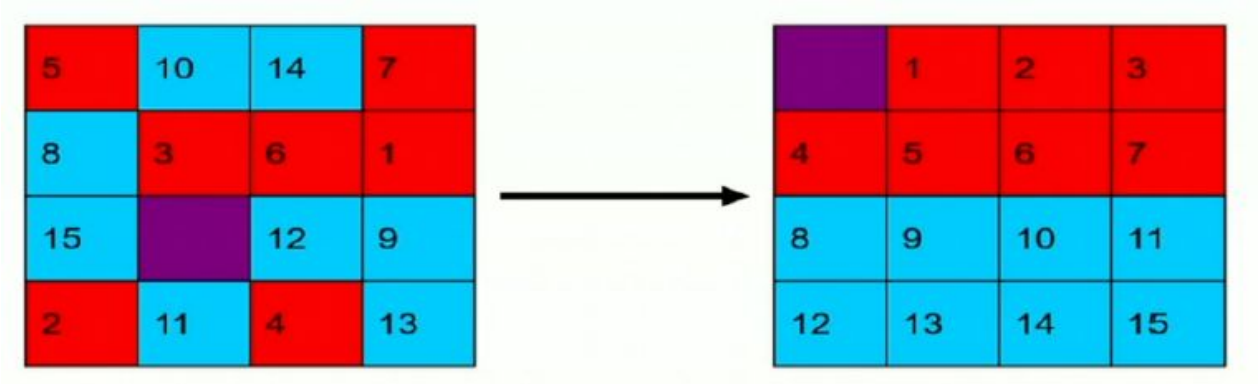
- **Original Problem:** Constraints
  a. Swap with the space
  b. Swap with the neighbour
- **Relaxation:**
  a. For, h_1, I remove the 2nd constraint i.e., swap with neighbour but keep the 1st one that it can only swap with gaps, then I am allowing the tiles to fly.
    i. So here my total cost will be 8, which is the cost of h_1 hence, we find h_1 to be an admissible heuristic.
  b. For h_2, I relax the constraint of swapping again and allow the tiles to move over one another.
    i. Eg., allow 4 to share the tile with 6 before moving to its actual position i.e., the gap.
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.

Pattern Database Heuristics

Pattern Database Heuristics (PDBs) are a specific type of heuristic used in search algorithms, particularly in solving complex combinatorial problems like sliding tile puzzles, Rubik's Cube, and other similar problems. The core idea behind pattern database heuristics is to precompute and store the exact solutions for simplified versions of the problem—referred to as "patterns"—and then use these precomputed solutions as heuristics during the actual search process.

Definition of Pattern Database Heuristics

- **Pattern:**
  - A pattern in this context refers to a subset of the problem's components (e.g., a subset of tiles in a puzzle) while ignoring the rest. The pattern represents a smaller, simpler version of the original problem.
- **Pattern Database (PDB):**
  - A pattern database is a large lookup table that stores the optimal solution (e.g., minimum number of moves) for all possible configurations of a given pattern.
  - This database is created by exhaustively solving the subproblem represented by the pattern and recording the cost to reach the goal configuration for each possible state of the pattern.
- **Heuristic Calculation:**
  - During the actual search in the original problem, the current state is matched against the patterns. The heuristic value is obtained by looking up the pattern database to get the precomputed cost for that pattern, which gives an estimate of how far the problem is from the goal.



Advantages:

- **Efficiency:** Pattern database heuristics can significantly speed up the search process by providing accurate, precomputed estimates that reduce the number of states the algorithm needs to explore.
- **Scalability:** For large and complex problems, where calculating exact solutions on-the-fly would be computationally expensive, pattern database heuristics offer a practical way to handle the complexity.
- **Improved Search Performance:** When integrated with search algorithms like A*, pattern database heuristics can dramatically reduce the search space, making it feasible to solve problems that would otherwise be intractable.

Disadvantages:

- **Memory Usage:** Storing the pattern database can require significant amounts of memory, especially for larger patterns or more complex problems.
- **Precomputation Time:** Creating the pattern database requires an exhaustive search of all possible configurations of the pattern, which can be time-consuming.
- **Complexity of Design:** Designing effective pattern databases involves carefully choosing patterns that provide meaningful heuristics while balancing memory usage and computational complexity.