Artificial Intelligence

For naturally intelligent beings

CSE422 Lecture Notes

First Edition

Ipshita Bonhi Upoma

Departmant of Computer Science and Engineering
School of Data and Sciences

NOTE TO STUDENTS

Welcome to our course on Artificial Intelligence (AI)! This course is designed to explore the core strategies and foundational concepts that power intelligent systems, solving complex, real-world problems across various domains. From navigating routes to optimizing industrial processes, and from playing strategic games to making predictive models, the use of AI is pivotal in creating efficient and effective solutions.

These lecture notes have been prepared by Ipshita Bonhi Upoma (Lecturer, BRAC University) based on the book "Artificial Intelligence: A Modern Approach" by Peter Norvig and Stuart Russell. The credit for the initial typesetting of these notes goes to Arifa Alam.

The purpose of these notes is to provide additional examples to enhance the understanding of how basic AI algorithms work and the types of problems they can solve.

This is a work in progress, and as it is the first time preparing such notes, many parts need citations and rewriting. Several graphs, algorithms and images are directly borrowed from Russell and Norvig's book. The next draft will address these citations.

As this is the first draft, please inform us of any typos, mathematical errors, or other technical issues so they can be corrected in future versions.

Lecture Plan (Central)

Our Reference book (Artificial Intelligence: A Modern Approach): https://drive.google.com/file/d/16pK--SRAKZzkVs8NcxxYCDFfxtvE0pmZ/view?usp=sharing

CSE422 OBE Outline: https://docs.google.com/document/d/1SJFBkfkL0wHUokhqfcBRZLhNqtw6Qhjt/edit?usp=sharing&ouid=107280348520227181657&rtpof=true&sd=true

CSE422 Tentative Lecture Plan: https://docs.google.com/document/d/1fAsXap2Qjm-QdAgAXKb2J2bAO
edit?usp=sharing&ouid=107280348520227181657&rtpof=true&sd=true

Marks Distribution

- Class Task 5%
- Quiz 10%(4 questions, 3 will be counted)

- Assignment 10% (4 Assignsments, Bonus assignments?)
- Lab 20%
- Mid 25%
- Final 30%

Class rules for Section 13 and 14:

1. Classwork on the lecture will be given after the lecture and attendance will be counted based on the classwork.
2. Feel free to bring coffee/ light snacks for yourself. Make sure to not cause any noise in the class.
3. If you are not enjoying the lecture you are free to leave the lecture, but in no way you should do anything that disturbs me or the other students.
4. If you want me to consider a leave of absence, email with valid reason. Classwork must be submitted even if leave is considered.
5. 3 of 4 quizzes will be counted.
6. All assignments will be counted. 30% penalty for late submission.
7. Cheating in any form will not be tolerated and will result in a 100% penalty.
8. If bonus assignments are given, the marks of bonus will be added after completion of all other assessments.
9. Lab marks are totally up to the Lab instructor.
10. No grace marks will be given for any grade bump. Such requests will not be taken nicely.

CONTENTS

Part I

CHAPTER 1

INTRODUCTION: PAST, PRESENT, FUTURE

1.1 Some of the earliest problems that were solved using Artificial Intelligence.

In the 1950s, the field of Artificial Intelligence (AI) began to take shape as researchers pondered the question: Can machines think? Alan Turing, one of the pioneers in computing, proposed the Turing Test as a way to measure if a machine could think like a human. This sparked an interest in developing computers that could simulate human reasoning, laying the groundwork for AI. Researchers focused on translating human problem-solving abilities into mathematical rules and algorithms, forming the very foundation of the field.

AI research quickly pushed forward, especially in Bayesian networks and Markov Decision Processes, which allowed machines to reason under uncertainty. Games also became a testing ground for AI. In 1951, Turing designed a theoretical chess program, followed soon by Arthur Samuel's checkers program, one of the earliest examples of a machine that could learn from experience—a major step towards machine learning.

AI wasn't just about games, though. In 1956, The Logic Theorist program, created by Allen Newell, Herbert Simon, and Cliff Shaw, was designed to mimic human reasoning, even proving mathematical theorems from Principia Mathematica. This breakthrough demonstrated AI's ability to solve complex logical tasks. Meanwhile, machine translation experiments, like the Georgetown project in 1954, successfully converted Russian sentences into English, showcasing AI's potential for language processing.

Even early speech recognition made strides in this decade, with Bell Labs developing Audrey, a system that recognized spoken numbers. These advancements in the 1950s laid a strong foundation for AI's growth.

In the 1960s, AI research advanced further, focusing on problem-solving algorithms and early neural networks like Perceptrons. During this time, control theory also became part of

robotics, helping machines perform tasks efficiently and adaptively.

The 1970s saw AI face skepticism. Reports like the Lighthill Report highlighted AI's limitations, but this decade also saw graph theory grow in importance, allowing for knowledge representation in early expert systems.

The 1980s brought significant advances with expert systems that made decisions using rule-based logic, as well as the backpropagation algorithm, which greatly improved neural networks.

In the 1990s, AI reached new heights. IBM's Deep Blue made history by defeating a world chess champion, and statistical learning models like Support Vector Machines gained popularity, leading to a data-driven approach in AI.

The 2000s introduced game theory for multi-agent systems, and deep learning brought renewed interest in neural networks for complex pattern recognition.

In the 2010s, AI achieved remarkable feats with IBM's Watson and Google DeepMind's AlphaGo, showcasing AI's ability to understand language and excel in strategic games, demonstrating the depth of AI's problem-solving abilities.

Large Language Models (LLMs) like GPT and BERT are significant developments in natural language processing. Starting with early neural network foundations in the 2000s, advancements such as Word2Vec laid the groundwork for sophisticated word embeddings. The 2017 introduction of the Transformer architecture revolutionized NLP by efficiently handling long-range dependencies in text.

In 2018, Google's BERT and OpenAI's GPT used the Transformer model to understand and generate human-like text, with BERT improving context understanding through bidirectional training, and GPT enhancing generative capabilities. Recent iterations like GPT-3 and GPT-4 have scaled up in size and performance, expanding their application range from content generation to conversational AI.

Today, in the 2020s, AI is focusing on issues like fairness and bias and exploring the potential of quantum computing to revolutionize the field even further.

1.2 Problems we are trying to solve using these days:

Healthcare—

Disease Diagnosis: AI algorithms analyze medical imaging data to detect and diagnose diseases early, such as cancer or neurological disorders.

Personalized Medicine: AI helps tailor treatment plans to individual patients based on their genetic makeup and specific health profiles.

Transportation—

Autonomous Vehicles: AI powers self-driving cars, aiming to reduce human error in driving and increase road safety.

Traffic Management: AI optimizes traffic flow, reduces congestion, and enhances public transport systems through predictive analytics and real-time data processing.

Finance—

Fraud Detection: AI systems analyze transaction patterns to identify and prevent fraudulent activities in real time.

Algorithmic Trading: AI uses complex mathematical formulas to make high-speed trading decisions to maximize investment returns.

Retail—

Customer Personalization: AI enhances customer experience by providing personalized recommendations based on past purchases and browsing behaviors.

Inventory Management: AI predicts future product demand, optimizing stock levels and reducing waste.

Education—

Adaptive Learning Platforms: AI tailors educational content to the learning styles and pace of individual students, improving engagement and outcomes.

Automated Grading: AI systems grade student essays and exams, reducing workload for educators and providing timely feedback.

Environment—] Climate Change Modeling: AI analyzes environmental data to predict changes in climate patterns, helping in planning and mitigation strategies.

Wildlife Conservation: AI assists in monitoring and protecting wildlife through pattern recognition in animal migration and population count.

Manufacturing—] Predictive Maintenance: AI predicts when equipment will require maintenance, preventing unexpected breakdowns and saving costs.

Quality Control: AI automatically inspects products for defects, ensuring high quality and reducing human error.

Cybersecurity— Threat Detection: AI monitors network activities to detect and respond to security threats in real time.

Vulnerability Management: AI predicts which parts of a software system are vulnerable to attacks and suggests corrective actions.

Entertainment— Content Recommendation: AI algorithms power recommendation systems in streaming services like Netflix and Spotify to suggest movies, shows, and music based on user preferences.

Game Development: AI is used to create more realistic and intelligent non-player characters (NPCs) and to enhance gaming environments.

Legal— Document Analysis: AI helps in reviewing large volumes of legal documents to identify relevant information, reducing the time and effort required for legal research.

Case Prediction: AI analyzes past legal cases to predict outcomes and provide guidance on legal strategies.

CHAPTER 2

SOLVING PROBLEMS WITH AI

2.1 Solving problems with artificial intelligence

In this course, we explore three distinct strategic domains of artificial intelligence: Searching Strategies, Constraint Satisfaction Problems (CSP), and Machine Learning.

Solving any problem first requires abstraction/problem formulation of the problem so that the problem can be tackled using an algorithmic solution. Based on that abstraction we choose a suitable strategy to solve the problem.

Real-world AI challenges are rarely straightforward. They often need to be broken down into smaller parts, with each part solved using a different strategy. For example, in creating an autonomous vehicle, informed search may help us find a route to destination, adversarial search helps us predict other drivers' actions, while machine learning helps the vehicle understand road signs.

Thankfully in this course, we'll focus on learning each strategy separately. This approach lets us dive deep into each area without worrying about combining them.

2.1.1 Searching Strategies

Informed Search

Why— Informed search strategies, such as A* and Best-First Search, utilize heuristics (we will come back to this later) to efficiently find solutions, focusing the search towards more promising paths. These strategies are essential in scenarios like real-time pathfinding for autonomous vehicles.

Local Search

Why— Local search methods are crucial for tackling optimization problems where finding an optimal solution might be too time-consuming. These methods, which include Simulated Annealing and Hill Climbing, are invaluable for tasks such as resource allocation and scheduling where a near-optimal solution is often sufficient.

Adversarial search

Why—Adversarial search techniques are essential for environments where agents compete against each other, such as in board games or market competitions. Understanding strategies like Minimax and Alpha-Beta Pruning allows one to predict and counter opponents' moves effectively.

2.1.2 Constraint Satisfaction Problems (CSP)

Constraint Satisfaction Problems (CSP)

Why—CSPs are studied to solve problems where the goal is to assign values to variables under strict constraints. Techniques like backtracking and constraint propagation are fundamental for solving puzzles, scheduling problems, and many configuration problems where all constraints must be satisfied simultaneously.

2.1.3 Machine Learning Techniques

Probabilistic Reasoning

Why—We delve into probabilistic reasoning to equip students with methods for making decisions under uncertainty. Techniques such as Bayesian Networks are vital for applications ranging from diagnostics to automated decision-making systems.

Decision Tree

Why—Decision trees are introduced due to their straightforward approach to solving classification and regression problems. They split data into increasingly precise subsets using simple decision rules, making them suitable for tasks from financial forecasting to clinical decision support.

Gradient Descent

Why—The gradient descent algorithm is essential for optimizing machine learning models, particularly in training deep neural networks. Its ability to minimize error functions makes it indispensable for developing applications like voice recognition systems and personalized recommendation engines.

2.2 Some keywords you will hear every now and then

Agent: In AI, an agent is an entity that perceives its environment through sensors and acts upon that environment using actuators. It operates within a framework of objectives, using its perceptions to make decisions that influence its actions.

Rational Agent: A rational agent acts to achieve the best outcome or, when there is uncertainty, the best expected outcome. It is "rational" in the sense that it maximizes its performance measure, based on its perceived data from the environment and the knowledge it has.

Autonomous Agent: An autonomous agent is a type of rational agent that can *learn from its own experiences and actions.* It can adjust its behavior based on new information, making up for any initial gaps or inaccuracies in its knowledge. Essentially, an autonomous agent operates independently and adapts effectively to changes in its environment.

Task Environment: In AI, the environment refers to everything external to the agent that it interacts with or perceives to make decisions and achieve its goals. It includes all factors, conditions, and entities that can influence or be influenced by the agent's actions.

2.3 Properties of Task Environment

Understanding these properties helps in the design and development of AI systems, tailoring the AI's architecture, algorithms, and decision-making processes to the specific nature of the environment it will operate in. This enables more effective, efficient, and appropriate responses to varying conditions and objectives.

Fully Observable vs. Partially Observable

Fully Observable: In these environments, the agent's sensors provide access to the complete state of the environment at all times. This allows the agent to make decisions with full knowledge of the world. For example, a chess game where the agent (player) can see the entire board and all the pieces at all times.

Partially Observable: Here, the agent only has partial information about the environment due to limitations in sensor capabilities or because some information is inherently hidden. Agents must often infer or guess the missing information to make decisions.For instance, in a poker game, players cannot see the cards of their opponents.

Deterministic vs. Stochastic

Deterministic: The outcome of any action by the agent is completely determined by the current state and the action taken. There is no uncertainty involved. For example, a tic-tac-toe game where each move reliably changes the board in a predictable way.

Stochastic: In these environments, actions have probabilistic outcomes, meaning the same action taken under the same conditions can lead to different results. Agents must deal with uncertainty and probability. For example, in stock trading, buying a stock does not guarantee profit due to market volatility.

Episodic vs. Sequential

Episodic: The agent's experience is divided into distinct episodes, where the action in each episode does not affect the next. Each episode consists of the agent perceiving and then performing a single action. An example is image classification tasks where each image is processed independently.

Sequential: Actions have long-term consequences, and thus the current choice affects all future decisions. Agents need to consider the overall potential future outcomes when deciding their actions. Navigation tasks where an agent (like a robot or self-driving car) must continuously make decisions based on past movements exemplify this.

Static vs. Dynamic

Static: The environment does not change while the agent is deliberating. This simplicity allows the agent time to make a decision without worrying about the environment moving on. An example is a Sudoku puzzle, where the grid waits inertly as the player strategizes.

Dynamic: The environment can change while the agent is considering its actions. Agents need to adapt quickly and consider the timing of actions. For example, in automated trading systems where market conditions can change in the midst of computations.

Discrete vs. Continuous

Discrete: Possible states, actions, and outcomes are limited to a set of distinct, clearly defined values. For example, a chess game has a finite number of possible moves and positions.

Continuous: The environment may change continuously, and the number of possible states or actions is infinite. For instance, driving a car involves navigating through a continuous range of positions and speeds.

Competitive vs. Cooperative

Competitive: Agents operate in environments where other agents might have conflicting objectives, like in strategic games such as in a game of chess where each player aims to defeat the other.

Cooperative: Agents work together towards a common goal, which may involve communication and shared tasks. For example, a collaborative robotics setting where multiple robots work together to assemble a product.

2.4 Types of Agents

In artificial intelligence, agents can be categorized based on their operational complexity and capabilities.

Simple Reflex Agents

These agents act solely based on the current perception, ignoring the rest of the perceptual history. They operate on a condition-action rule, meaning if a condition is met, an action is taken.

Example: A room light that turns on when it detects motion. It does not remember past movements; it only responds if it detects motion currently.

Model-Based Reflex Agents

These agents maintain some sort of internal state that depends on the percept history, allowing them to make decisions in partially observable environments. They use a model of the world to decide their actions.

Example: A thermostat that controls a heating system. It uses the history of temperature changes and the current temperature to predict and adjust the heating to maintain a set temperature.

Goal-Based Agents

These agents act to achieve goals. They consider future actions and evaluate them based on whether they lead to the achievement of a set goal.

Example: A navigation system in a car that plans routes not only based on the current location but also on the destination the user wants to reach.

Utility-Based Agents:

These agents aim to maximize their own perceived happiness or satisfaction, expressed as a utility function. They choose actions based on which outcome provides the greatest benefit according to this utility.

Example: An investment bot that decides to buy or sell stocks based on an algorithm designed to maximize the expected return on investment, weighing various financial indicators and market conditions.

2.4.1 Learning Agent

A learning agent in artificial intelligence is an agent that can improve its performance over time based on experience. This type of agent typically consists of four components: a learning element that updates knowledge, a performance element that makes decisions, a critic that assesses how well the agent is doing, and a problem generator that suggests challenging situations to learn from.

Example: Self-Driving Car

A self-driving car is a learning agent that adapts and improves its driving decisions based on accumulated driving data and experiences.

Performance Element: This part of the agent controls the car, making real-time driving decisions such as steering, accelerating, and braking based on current traffic conditions and sensor inputs.

Learning Element: It processes the data gathered from various sensors and feedback from the performance element to improve the decision-making algorithms. For example, it learns to recognize stop signs better or understand the nuances of merging into heavy traffic.

Critic: It evaluates the driving decisions made by the performance element. For instance, if a particular maneuver led to a near-miss, the critic would flag this as suboptimal.

Problem Generator: This might simulate challenging driving conditions that are not frequently encountered, such as slippery roads or unexpected obstacles, to prepare the car for a wider range of scenarios.

Over time, by learning from both successes and failures, a self-driving car improves its capability to drive safely and efficiently in complex traffic environments, demonstrating how learning agents adapt and enhance their performance based on experience.

2.5 Problem Formulation and choice of strategies

Problem formulation is introduced at the outset because it sets the stage for all AI strategies. It involves defining the problem in a way that a computer can process—identifying what the inputs are, what the desired outputs should be, and the constraints and environment within which the problem exists. This is crucial for effectively applying any AI technique, as a well-formulated problem can significantly simplify the solution process. It is foundational in areas such as robotics, where tasks need to be defined clearly before they can be automated.

2.6 Steps of Problem Formulation

1. **Define the Goal**

   Start by clearly identifying what needs to be achieved. This involves understanding the desired outcome and what constitutes a solution to the problem. *Example:* For an autonomous vacuum cleaner, the goal might be to clean the entire floor space of a house without retracing any area unnecessarily. For a puzzle this could be the configuration of the puzzle when solved 2.1.

2. **Identify the Initial State**

   Determine the starting point of the problem.

   Example: In a chess game, the initial state is the standard starting position of all pieces on the chessboard.

   For an engineering task, the current measurements of a system.

   For a machine learning model, the initial data set from which the model will learn.

3. **Determine the Possible Actions**

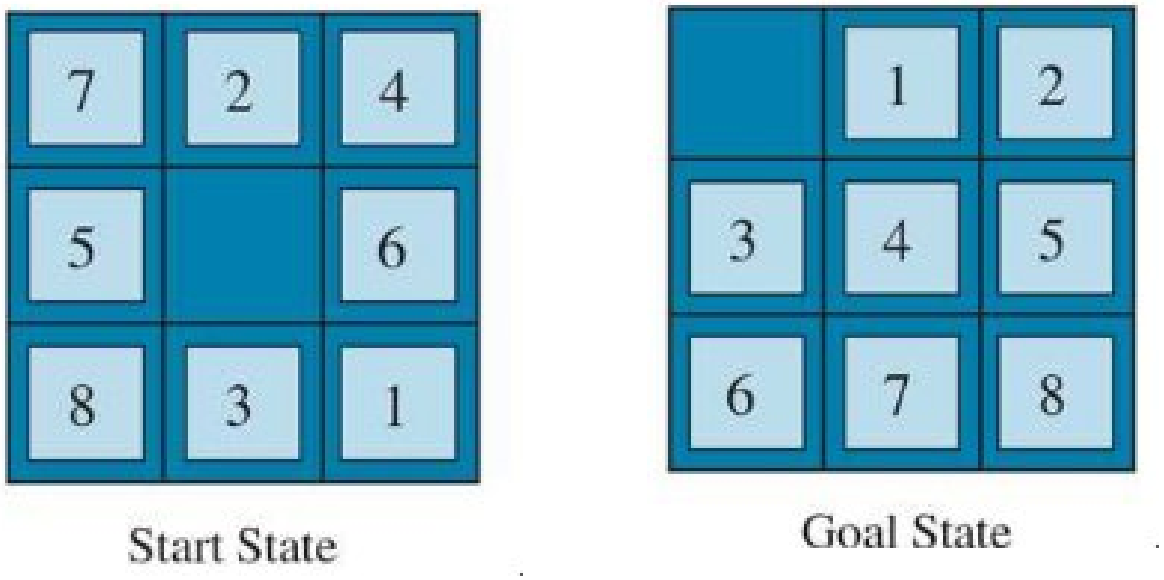


Figure 2.1: Initial and Goal State of a 8-Puzzle Game (Russel and Norvig, *Artificial Intelligence: A Modern Approach*)

List out all possible actions that can be taken from any given state.

Example: In an online booking system, actions could include selecting dates, choosing a room type, and adding guest information.

In a navigation problem, for example, these actions could be the different paths or turns one can take at an intersection.

For a sorting algorithm, actions might be the comparisons or swaps between elements.

4. **Define the Transition Model**

   Establish how each action affects the state. The transition model describes what the new state will be after an action is taken from a current state.

   Example: In a stock trading app, the transition model would define how buying or selling stocks affects the portfolio's state, including changes in cash reserves and stock quantities.

   In a chess game, moving a pawn will change the state of the board.

5. **Establish the Goal Test**

   Create a method to determine whether a given state is a goal state. This test checks if the goal has been achieved.

   Example: In a puzzle like Sudoku, the goal test checks if the board is completely filled without any repeating numbers in any row, column, or grid.

   In a maze-solving problem, the goal test would verify whether the current location is the exit of the maze.

6. **Define the Path Cost**

   Decide how to measure the cost of a path. The path cost function will calculate the numerical cost of any given path from the start state to any state at any point. This is often critical in optimization problems where you want to find not just any solution, but the most cost-effective one.

   Example: For a route optimization problem, the path cost could include factors like total distance, travel time, and toll costs.

7. **Consider Any Constraints**

   Identify any constraints that must be considered. Constraints are limitations or restrictions on the possible solutions. For example, in scheduling, constraints could be the availability of resources or time slots.

   Example: In university class scheduling, constraints include classroom capacities, instructor availability, and specific time blocks when certain classes can or cannot be held.

8. **Select the Suitable AI Technique**

   Based on the problem's characteristics, such as whether the environment is deterministic or stochastic, static or dynamic, discrete or continuous, select the most appropriate AI technique. This could range from simple rule-based algorithms to complex machine learning models.

   Example: For a predictive maintenance system in a factory, the suitable AI technique might involve using machine learning models like decision tree to predict equipment failures based on historical sensor data. On the other hand, in a game of chess, we use adversarial search to decide our moves by considering what the opponent might do next for certain moves.

2.7 Examples of problem formulation

1. **City Traffic Navigation (Solved by Informed Search)**
   - **Problem Formulation:**
     - **Goal:** To find the quickest route from a starting point (origin) to a destination (end point) while considering current traffic conditions.
     - **States:** Each state represents a geographic location within the city's road network.
     - **Initial State:** The specific starting location of the vehicle.
     - **Actions:** From any given state (location), the actions available are the set of all possible roads that can be taken next.
     - **Transition Model:** Moving from one location to another via a chosen road or intersection.

- **Goal Test:** Determines whether the current location is the destination.
- **Path Cost:** Each step cost can be a function of travel time, which depends on factors such as distance and current traffic. The total path cost is the sum of the step costs, representing the total travel time.

◦ **Heuristics Used:**

- **Time Estimation:** An estimate of time from the current location to the destination, possibly using historical traffic data and real-time conditions.
- **Distance:** Straight-line distance (Euclidean or Manhattan distance) to the goal, which helps prioritize closer locations during the search process.

2. **Power Plant Operation (Solved by Local Search)**

◦ **Problem Formulation:**

- **Goal:** To optimize the power output while minimizing fuel usage and adhering to safety regulations.
- **States:** Each state represents a specific configuration of the power plant's operational settings (e.g., temperature, pressure levels, valve positions).
- **Initial State:** The current operational settings of the plant.
- **Actions:** Adjustments to the operational settings such as increasing or decreasing temperature, adjusting pressure, and changing the mix of fuel used.
- **Transition Model:** Changing from one set of operational settings to another.
- **Goal Test:** A set of operational conditions that meet all efficiency, safety, and regulatory requirements.
- **Path Cost:** Typically involves costs related to fuel consumption, wear and tear on equipment, and potential safety risks. The cost function aims to minimize these while maximizing output efficiency.

◦ **Heuristic Used:**

- **Efficiency Metrics:** Estimations of how changes in operational settings will affect output efficiency and resource usage. This might include predictive models based on past performance data.

3. **University Class Scheduling (Solved by CSP)**

◦ **Problem Formulation:**

- **Goal:** To assign time slots and rooms to university classes in a way that no two classes that share students or instructors overlap, and all other constraints are satisfied.
- **States:** Each state represents an assignment of classes to time slots and rooms.
- **Initial State:** No courses are assigned to any time slots or rooms.

- **Actions:** Assign a class to a specific time slot in a specific room.
- **Transition Model:** Changing from one assignment to another by placing a class into an available slot and room.
- **Goal Test:** All classes are assigned to time slots and rooms without any conflicts with other classes.
- **Path Cost:** Path cost is not typically a factor in CSP for scheduling; instead, the focus is on fulfilling all constraints.

◦ **Constraints:**

- **Room Capacity:** Each class must be assigned to a room that can accommodate all enrolled students.
- **Time Conflicts:** No instructor or student can be required to be in more than one place at the same time.
- **Resource Availability:** Some classes require specific resources (e.g., laboratories or audio-visual equipment).
- **Instructor Preferences:** Some instructors may have restrictions on when they can teach.

4. **Disease Diagnosis (Solved by Decision Trees)**

◦ **Problem Formulation:**

- **Goal:** To accurately diagnose diseases based on symptoms, patient history, and test results.
- **States:** Each state represents a set of features associated with a patient, including symptoms presented, medical history, demographic data, and results from various medical tests.
- **Initial State:** The initial information gathered about the patient, which includes all initial symptoms and available medical history.
- **Actions:** jActions are not typically modeled in decision trees as they are used for classification rather than processes involving sequential decisions.
- **Transition Model:** Not applicable for decision trees since the process does not involve moving between states.
- **Goal Test:** The diagnosis output by the decision tree, determining the most likely disease or condition based on the input features.
- **Path Cost:** In decision trees, the cost is not typically measured in terms of path, but accuracy, specificity, and sensitivity of the diagnosis can be considered as metrics for evaluating performance.

◦ **Features Used:**

- **Symptoms:** Patient-reported symptoms and observable signs.
- **Test Results:** Quantitative data from blood tests, imaging tests, etc.

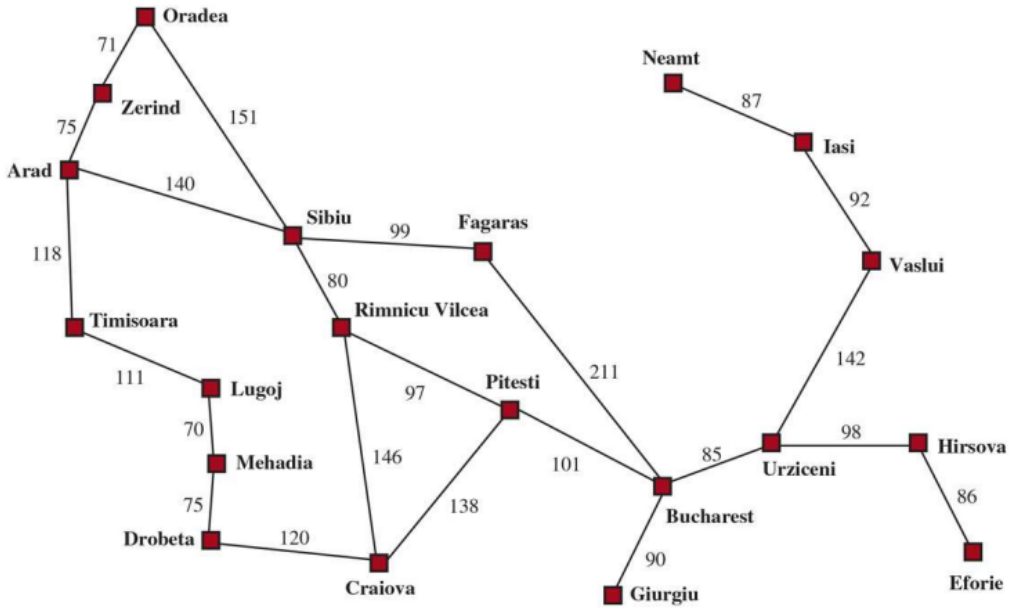
- **Demographic Data:** Age, sex, genetic information, lifestyle factors.
- **Medical History:** Previous diagnoses, treatments, family medical history.

CHAPTER 3

INFORMED SEARCH ALGORITHMS

Note: This lecture closely follows Chapter 3.6 (Heuristic Functions) and 3.5 (Informed Search) to 3.5.1 (Greedy Best First Search) of Russel and Norvig, *Artificial Intelligence: A Modern Approach*. The images are also borrowed from these chapters

As computer science students, you are already familiar with various search algorithms such as Breadth-First Search, Depth-First Search, and Dijkstra's/Best-First Search. These strategies fall under the category of Uninformed Search or Blind Search, which means they rely solely on the information provided in the problem definition.



A simplified road map of part of Romania, with road distances in miles.

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.

For example, consider a map of Romania where we want to travel from Arad to Bucharest. The map indicates that Arad is connected to Zerind by 75 miles, Sibiu by 140 miles, and Timișoara by 118 miles. Using a blind search strategy, the next action from Arad would be chosen based solely on the distances to these connected cities. This approach can be slower and less efficient as it may explore paths that are irrelevant to reaching the goal efficiently.

In this course, we will focus on informed search strategies, also known as heuristic search. Informed Search uses additional information—referred to as heuristics—to make educated guesses about the most promising direction to pursue in the search space. This approach often results in faster and more efficient solutions because it avoids wasting time on less likely paths. We will study Greedy Best-First Search and A* search extensively. But first, let's explore the concept of heuristics.

3.1 Heruistic Function

In the context of informed search algorithms, a heuristic is a technique that helps the algorithm estimate the cost (often the shortest path or least costly path) from a current state (or node) to the goal state. It's essentially a function that provides guidance on which direction the search should take in order to find the most efficient path to the goal. This guidance allows informed search algorithms to perform more efficiently than uninformed search algorithms, which do not have knowledge of the goal state as they make their decisions.

3.1.1 Key Characteristics of Heuristics in Search Algorithms

> **Estimation:** A heuristic function estimates the cost to reach the goal from a current node. This estimate does not need to be exact but should never overestimate.

Returning to the example of traveling to Bucharest from Arad: A heuristic function can estimate the shortest distance from any city in Romania to the goal. For instance, we might use the straight-line distance as a measure of the heuristic value for a city. The straight-line distance from Arad to Bucharest is 366 miles, although the optimal path from Arad to

Bucharest actually spans 418 miles. Therefore, the heuristic value for Arad is 366 miles. For each node (in this problem, city) in the state space (in this problem, the map of Romania) the heuristic value will be their straight line distance from the goal state (in this case Bucharest).

3.1.2 Why do we use heuristics?

Guidance: The heuristic guides the search process, helping the algorithm prioritize which nodes to explore next based on which seem most promising—i.e., likely to lead to the goal with the least cost.

Efficiency: By providing a way to estimate the distance to the goal, heuristics can significantly speed up the search process, as they allow the algorithm to focus on more promising paths and potentially disregard paths that are unlikely to be efficient.

3.2 Heuristic functions to solve different problems

Generating a heuristic function for use in informed search algorithms involves a process where the function must effectively estimate the cost, usually the shortest path, from any node or state in the search space to the goal. The design of the heuristic function is based on the problem domain, which requires an understanding of the rules, constraints, and ultimate goal of the problem. Here are some examples of heuristic functions for different types of problems.

8-Puzzle Game

The number of misplaced tiles (blank not included): For the figure above, all eight tiles are out of position, so the start state has $h_1 = 8$.

The sum of the distances of the tiles from their goal positions: Because tiles cannot move along diagonals, the distance is the sum of the horizontal and vertical distances- sometimes called the city-block distance or Manhattan distance.

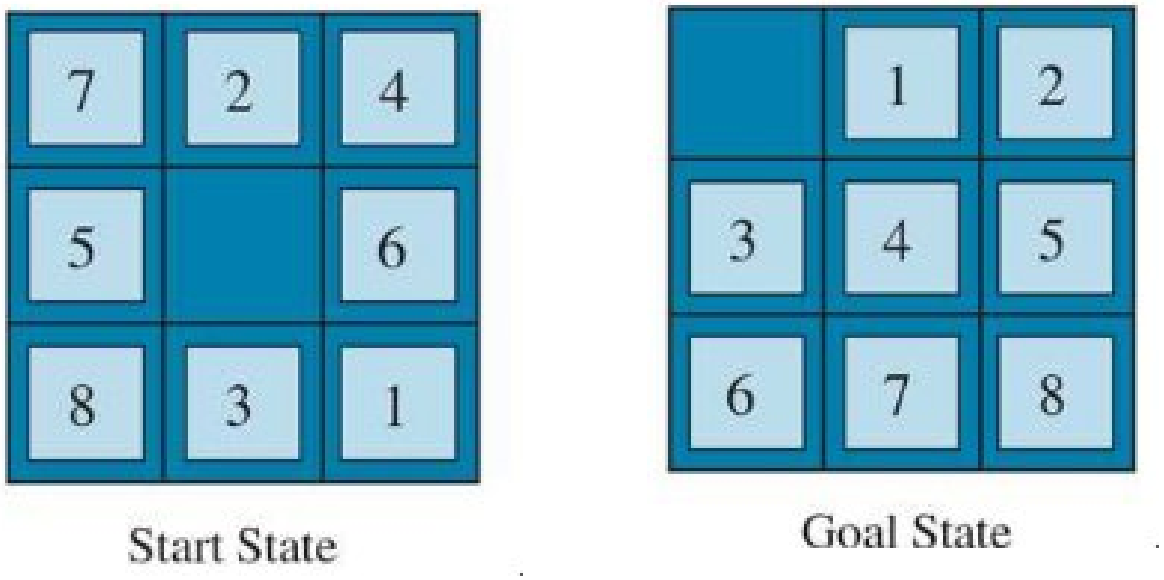


Figure 3.1: Initial and Goal State of a 8-Puzzle Game (Russel and Norvig, *Artificial Intelligence: A Modern Approach*)

Pathfinding in Maps and GPS Systems

Straight-line Distance: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ where (x_1, y_1) and (x_2, y_2) are the coordinates of the current position and the goal.

Travel Time: Estimating the time needed to reach the goal based on average speeds and road types. $t = \frac{d}{v}$ where d is the distance and v is the average speed.

Traffic Patterns: Using historical or real-time traffic data to estimate the fastest route. Could involve a weighting factor, w based on traffic data, modifying the travel time: $t_{adjusted} = t \times w$.

Game AI (e.g., Chess, Go)

Material Count: Sum of the values of all pieces. For example, in chess, pawns = 1, knights/bishops = 3, rooks = 5, queen = 9.

Positional Advantage: A score based on piece positions. E.g., control of center squares in chess might be given additional points.

Mobility: Number of legal moves available M for a player at a given turn.

Web Search Engines

Keyword Frequency: The number of times a search term appears on a webpage.

$$F = \frac{\text{Number of occurrences of keyword}}{\text{Total number of words in document}}$$

Page Rank: Evaluating the number and quality of inbound links to estimate the page's importance.

Domain Authority: The reputation and reliability of the website hosting the information. Often a proprietary metric, but generally a combination of factors like link profile, site age, traffic, etc.

Robotics and Path Planning

Distance to Goal: Estimating the remaining distance to the target location. Same as straight-line distance in GPS systems.

Obstacle Proximity: Distance to the nearest obstacle to avoid collisions. $O = \min(\text{distance to each obstacle})$.

Energy Efficiency: Estimating the most energy-efficient path, important for battery-powered robots.

$$E = \sum \text{energy per unit distance} \times \text{path distance}$$

Natural Language Processing (NLP)

Word Probability: $P(w \ |\text{context})$ where w is the word and context represents the surrounding words.

Semantic Similarity: How closely words or phrases match in meaning. Often uses cosine similarity,

$$\text{similarity} \ = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

where (A), and (B) are vector representations of words or sentences.

Language Consistency: Ensuring the text follows grammatical and syntactical norms of the target language. Can be quantified using perplexity in language models.

Recommendation Systems

User Behavior Tracking: Score items based on frequency and recency of user interactions. Analyzing past purchases or viewing habits to predict future interests.

Item Similarity: Recommending products similar to those a user has liked or purchased. Cosine similarity or other distance measures between item feature vectors.

Collaborative Filtering: Using preferences of similar users to recommend items. Matrix factorization techniques or neighbor-based algorithms to predict user preferences.

3.3 Greedy Best First Search- Finally an algorithm

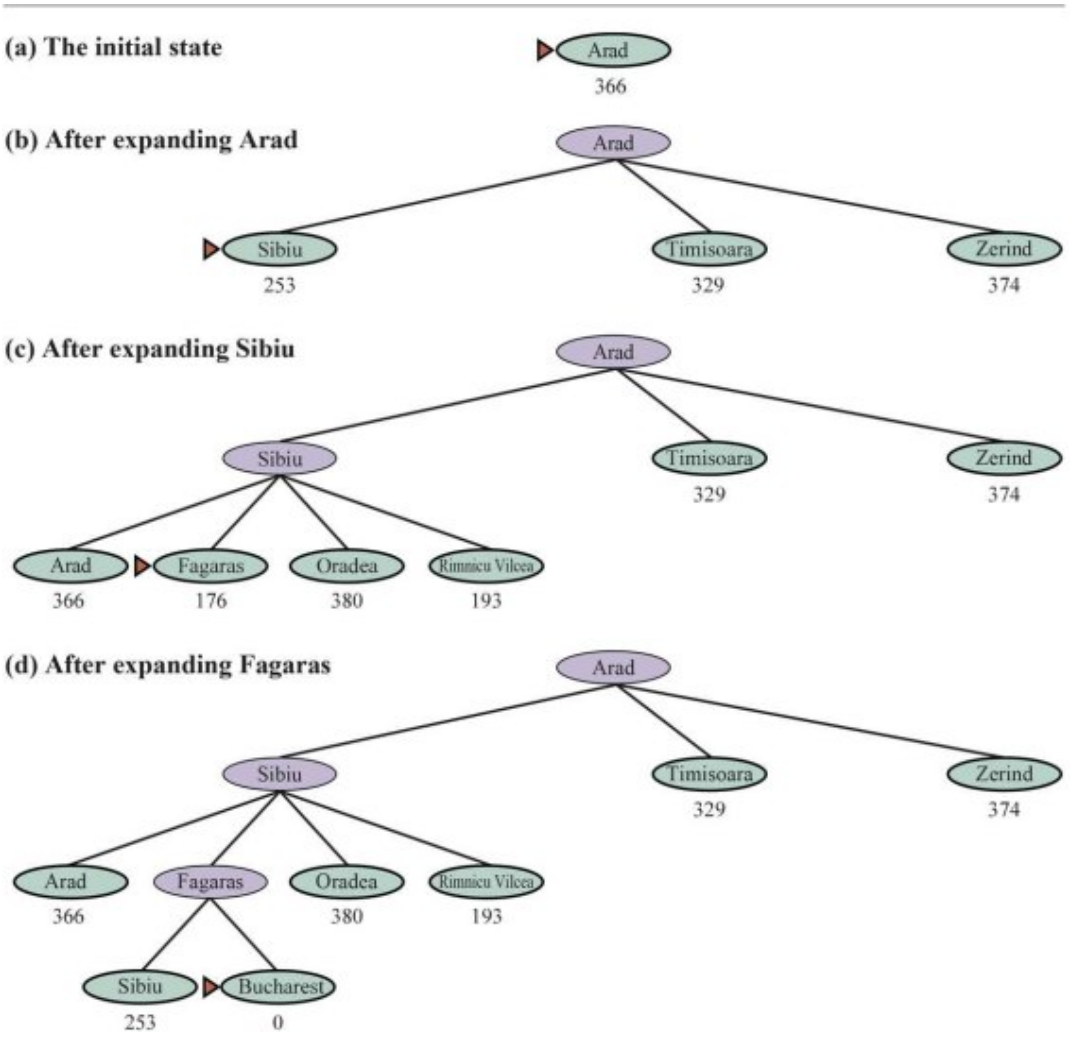
Greedy best-first search is a form of best-first search that expands first the node with the lowest $h(n)$ value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function $f(n) = h(n)$.

Let us see how this works for route-finding problems in Romania; we use the **straight-line-distance** heuristic, which we will call h_{SLD} . If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in the figure below. For example, $h_{SLD}(Arad) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself (that is, the `ACTIONS` and `RESULT` functions). Moreover, it takes a certain amount of world knowledge to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

The next figure shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than is either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path. The solution it found does not have optimal cost, however: the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This is why the algorithm is called "greedy"—on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful.

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

Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is $O(|\ V\ |)$. With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching $O(bm)$.

3.3.1 Algorithm: Greedy Best-First Search

Algorithm 1 Greedy Best First Search Algorithm

- `Input:`
  - `start:` The target node of the search
  - `goal:` The target node to reach
  - `heuristic(node):` A function that estimates the cost from node to the goal
- `Output:`
  - The path from `start` to `goal` if one exists, otherwise `None`.
- `Procedure`
  - Initialize:
    - ☐ Create a priority queue and insert the start node along with its heuristic value `heuristic(start)`.
    - ☐ Define a `visited` set to keep track of all visited nodes to avoid cycles and redundant paths.
  - Search:
    - ☐ While the priority queue is not empty:
      - Remove the node `current` with the lowest heuristic value from the priority queue.
      - If `current` is the goal, return the path that led to current.
      - Add `current` to the `visited` set.
      - For each neighbor `n` of `current`:
        - If `n` is not in `visited`:
          - Calculate the heuristic value `heuristic(n)`.
          - Add `n` to the priority queue with the priority set to `heuristic(n)`.
  - Failure to find the goal:
    - ☐ If the priority queue is exhausted without finding the `goal`, return `None`.

3.3.2 Key Points

Heuristic Function: This function is crucial as it determines the search behavior. A good heuristic can dramatically increase the efficiency of the search.

Completeness and Optimality: Greedy Best-First Search does not guarantee that the shortest path will be found, making it neither complete nor optimal. It can get stuck in loops or dead ends if not careful with the management of the visited set.

Data Structures: The algorithm typically uses a priority queue for the frontier and a set for the visited nodes. This setup helps in efficiently managing the nodes during the search process.

Greedy Best-First Search is particularly useful when the path's exact length is less important than quickly finding a path that is reasonably close to the shortest possible. **It is well-suited for problems where a good heuristic is available.**

3.4 A* Search algorithm

The most common informed search algorithm is A* Search (pronounced "A-star search"), a best-first search that uses the evaluation function $f(n) = g(n) + h(n)$ where $g(n)$ is the path cost from the initial state to node n, and $h(n)$ is the estimated cost of the shortest path from n to a goal state, so we have $f(n)$ estimated cost of the best path that continues from n to a goal.

3.4.1 Algorithm: A* Search

Algorithm 2 A* Search Algorithm

- **Input:**
  - `start`: The starting node of the search.
  - `goal`: The target node to reach.
  - `neighbors(node)`: A function that returns the neighbors of `node`.
  - `cost(current, neighbor)`: A function that returns the cost of moving from `current` to `neighbor`.
  - `heuristic(node)`: A function that estimates the cost from `node` to the `goal`.
- **Output:**
  - The path from `start` to `goal` if one exists, otherwise `None`.
- **Procedure**
  - Initialize
    - ☐ Create a priority queue and insert the `start` node.

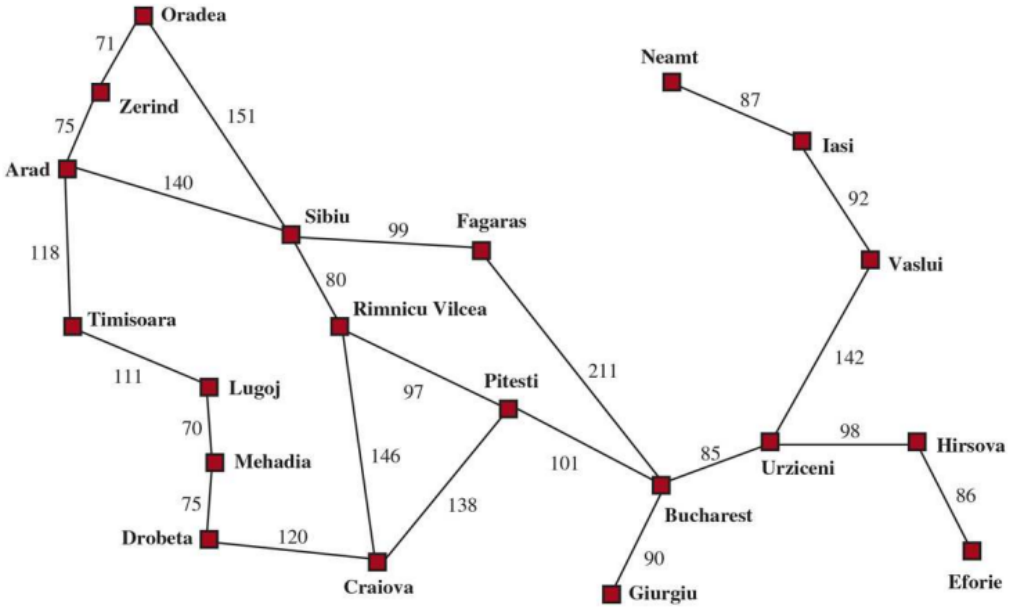
- ☐ Set `gScore[start]` to `0` (cost from start to `start`).
- ☐ Set `fScore[start]` = 0 + `heuristic(start)` (total estimated cost from start to goal through `start`).
- ☐ Define `cameFrom` to store the path reconstruction data.

– Search:

- ☐ While the priority queue is not empty:
  * Remove the node `current` with the lowest `fScore` value from the priority queue.
  * If `current` is the goal, reconstruct and return the path from start to goal using `cameFrom`.
  * For each neighbor of current:
  * For each neighbor of current:
    - Calculate `tentative-gScore` as `gScore[current] + cost(current, neighbor)`.
    - If `tentative-gScore` is less than `gScore[neighbor]` (or neighbor is not in `priority queue`):
      – Update `cameFrom[neighbor]` to `current`.
      – Update `gScore[neighbor]` to `tentative-gScore`.
      – Update `fScore[neighbor]` to `tentative-gScore + heuristic(neighbor)`.
      – If `neighbor` is not in the priority queue, add it.

– Failure to find the goal:

- ☐ If the priority queue is exhausted without reaching the `goal`, return `None`.



A simplified road map of part of Romania, with road distances in miles.

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.

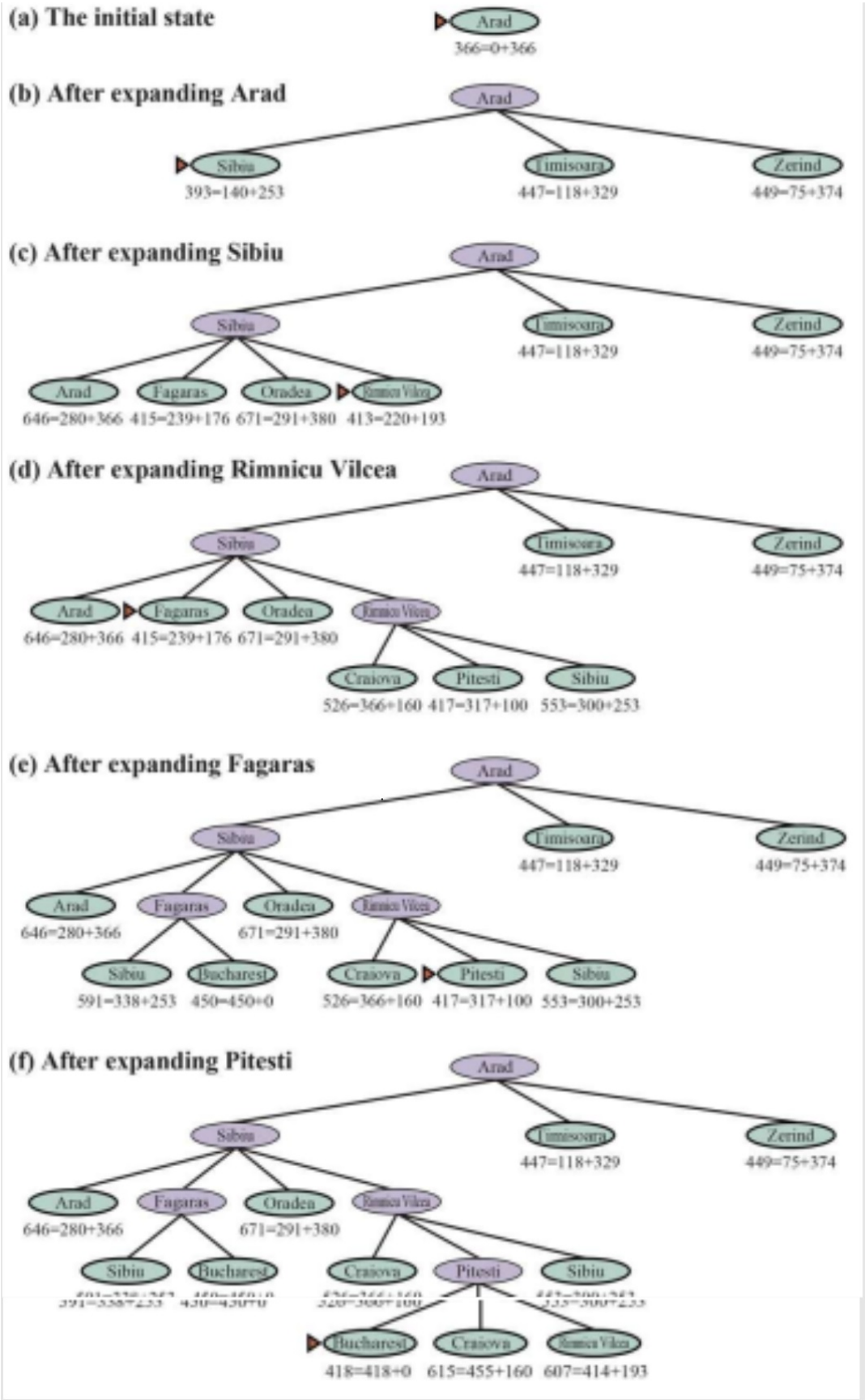


Figure 3.2: Simulation of A* Algorithm to find a path from Arad to Bucharest.

Path Reconstruction: The path is reconstructed from the cameFrom map, which records where each node was reached from.

Notice that Bucharest first appears on the frontier at step (e), but isn't selected for expansion as it isn't the lowest-cost node at that moment, with a cost of 450 compared to Pitesti's lower cost of 417. The algorithm prioritizes exploring potentially cheaper routes, such as through Pitesti, before settling on higher-cost paths. By step (f), a more cost-effective path to Bucharest, costing 417, becomes available and is subsequently selected as the optimal solution.

So we can say that the A* algorithm has the following properties,

- **Cost Focus:** Prioritizes nodes with the lowest estimated total cost, $f(n) = g(n)+h(n)$
- **Guarantees Optimality:** Ensures the solution is the least expensive by expanding the cheapest node first.
- **Resource Efficiency:** Avoids exploring more expensive paths when cheaper options are available.
- **Adapts Based on New Information:** Adjusts path choices dynamically as new cost information becomes available.
- **Heuristic Importance:** Relies on the heuristic to guide the search efficiently by estimating costs.

A* Algorithm is always complete, meaning that if a solution exists then the algorithm will find the path.

However, the A* algorithm only returns optimal solutions when the heuristic has some specific properties.

3.5 Condition on heuristics for A* to be optimal:

Whether A* Search is optimal depends on two key properties of the heuristic. These are **Admissibility** and **Consistency**.

Definition 3.5.1: Admissibility

An admissible heuristic never overestimates the cost to reach the goal. This makes it optimistic about the path costs.

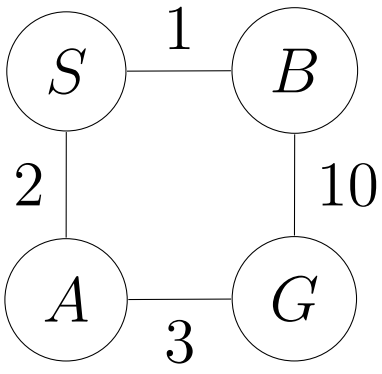
Proof of Cost-Optimality (via Contradiction)

- Assume the optimal path cost is $C*$ but $A*$ returns a path with a cost greater than $C*$.

- There must be a node n on the optimal path that $A*$ did not expand.
- If $f(n)$ which is the estimated cost of the cheapest solution through n were less or equal to $C*$ then n would have been expanded.
- By definition and admissibility, $f(n)$ is $g(n) + h(n)$ and should be less or equal to $g*(n)+$ cost(n to goal) $= C*$ as n is on the optimal path and $h(n)$ is less than or equal cost(n to goal) due to admissibility
- This is contradicting our assumption that $f(n) > C*$.
- Thus, $A*$ must indeed return the cost-optimal path.

Example: Where Violation of admissibility Leads to a Suboptimal Solution

- **Nodes:** Start (S), A, B, Goal (G)
- **Edges with costs:**
  - S to $A = 2$
  - A to $G = 3$
  - S to $B = 1$
  - B to $G = 10$



Heuristic, $h(node)$ Estimates to the Goal, G:

- $h(S) = 3$ admissible as $h(S) <$ the optimal path-cost from S to G.
- $h(A) = 10$ is in-admissible as $h(A) >$ optimal path-cost from A to G
- $h(B) = 2$ admissible.
- $h(G) = 0$ admissible.

A* Algorithm Execution:

- **Start at S:**
$$f(S) = g(S) + h(S) = 0 + 3 = 3.$$
- **Expand S:** Adding Neighbors (A and B) to the queue:
  - For A:
$$g(A) = 2, \quad \text{so, } f(A) = g(A) + h(A) = 2 + 10 = 12.$$
  - For B:
$$g(B) = 1, \quad \text{so, } f(B) = g(B) + h(B) = 1 + 2 = 3$$
- Node A and Node B are currently in queue.
- **Node B selected** from the queue for expansion because of **lower f-value** despite leading to a higher cost path.
- **Expand B:**, Adding Neighbor (G):
  - For G via B
$$g(G) = 11, \quad \text{so, } f(G) = g(G) + h(G) = 11 + 0 = 11$$
- Node A and Node G are currently in queue.
- **Node G selected** from the queue for expansion because of lower f-value.
- The algorithm returns path S-B-G which is sub-optimal.

The inadmissibility of the the heuristic causes the algorithm to prefer a sub-optimal path.

Definition 3.5.2: Consistency

A heuristic h is consistent if for every node n and every successor n' of n, the estimated cost from n to the goal, denoted $h(n)$, does not exceed the cost from n to n' plus the estimated cost from n' to the goal, $h(n')$. Mathematically, this is represented as: $h(n) \leq c(n, n') + h(n')$ where $c(n, n')$ is the cost to reach n' from n.

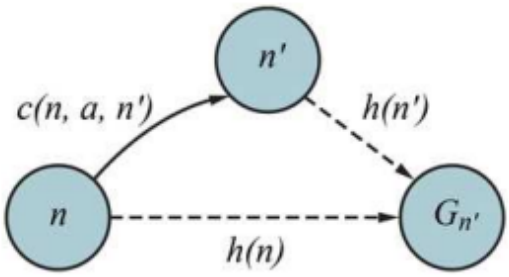
A consistent heuristic, also known as a monotonic heuristic, is a stronger condition than admissibility and plays a crucial role in ensuring that A* search finds the optimal path. Here's how it ensures that A* returns an optimal path:

1. **Path Cost Non-Decreasing:**
   - Consistency ensures that the f-value (total estimated cost) of a node n calculated as $f(n) = g(n) + h(n)$ does not decrease as the algorithm progresses from the start node to the goal. This is because for any node n and its successor n': $g(n') = g(n) + c(n, n')$

- Simplifying, we find:

$$f(n) = g(n) + h(n) \leq g(n) + c(n, n') + h(n') = g(n') + h(n') = f(n')$$

- Therefore, f-values along a path do not decrease, preventing any re-exploration of nodes already deemed sub-optimal, hence streamlining the search towards the goal.



Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

2. **Closed Set Invariance:**

- When a node n is expanded, its f-value is finalized. Due to the non-decreasing nature of f-values, any path rediscovered through n will have an f-value at least as large as when n was first expanded.
- This prevents the algorithm from revisiting nodes unnecessarily, thereby ensuring efficiency in path finding.

3. **Optimal Path Discovery:**

- Given the consistency condition, once the goal node g is reached and its $f(g)$ calculated, there can be no other path to g with a lower f-value that has not already been considered.
- Since $h(g) = 0$ (by definition at the goal), $f(g) = g(g)$, meaning that the path cost $g(g)$ represents the total minimal cost to reach the goal from the start node.

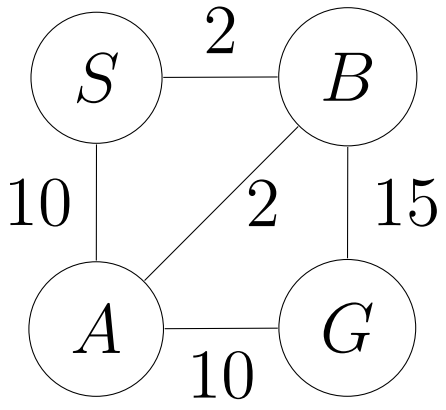
4. **Optimality Guarantee:**

- The search terminates when the goal is popped from the priority queue for expansion, and due to the non-decreasing nature of f-values, this means that no other path with a lower cost can exist that has not already been evaluated.
- Thus, the path to g with cost $g(g)$ must be optimal.

By adhering to these principles, A* search with a consistent heuristic not only finds a solution but ensures it is the optimal one.

Example: Checking inconsistency

- **Nodes:** Start (S), A, B, Goal (G)
- **Edges with costs:**
  - S to $A = 2$
  - A to $G = 3$
  - S to $B = 1$
  - B to $G = 10$



- **Heuristic (h) Estimates to the Goal (G):**
  - $h(S) = 12$
  - $h(A) = 7$
  - $h(B) = 10$
  - $h(G) = 0$
- **Checking consistency for each node:** First we find the optimal path to Goal from each node.
  - Optimal path from Node S is $S - B - A - G = 14$.
  - Optimal path from Node B is $B - A - G = 12$.
  - Optimal path from Node A is $A - G = 10$.

– Optimal path from Node G is $G- = 0$.

For any given node, n the heuristic, $h(n)$ for that node is lower or equal than the optimal path-cost from n. So we can say that the given heuristics are admissible. An inadmissible heuristic automatically leads to inconsistent heuristic. But admissible heuristic does not guarantee consistency.

It is better to start checking consistency from the Goal node.

- We see $h(G) = 0$ is admissible and consistent.
- Next, from Node A, there is a direct path to Goal node, G with cost 10 which is the optimal path. There is also a path via B. So to be consistent,

$$\begin{aligned} h(A) &\leq Cost(A, G) + h(G) \\ &= 10 + 0 = 10 \\ h(A) &= 7 \end{aligned}$$

  So, node A is consistent with node G.

$$\begin{aligned} h(A) &\leq Cost(A, B) + h(B) \\ &= 2 + 15 = 17 \\ h(A) &= 7 \end{aligned}$$

  So, heuristic of node A is consistent with node B.

- Next, from Node B, there is a direct path to Goal node, G with cost 15 and a path via nod A costing 12. In this case, G and A are the child of B. So to be consistent,

$$\begin{aligned} h(B) &\leq Cost(B, G) + h(G) \\ &= 15 + 0 = 15 \\ h(B) &= 10 \end{aligned}$$

  $h(B)$ is consistent with node G. However,

$$\begin{aligned} h(B) &\leq Cost(B, A) + h(A) \\ &= 2 + 7 = 9 \\ h(B) &= 10 \end{aligned}$$

  So, heuristic of node B with node A is inconsistent. Similarly for node S,

$$\begin{aligned} h(S) &\leq cost(S, A) + h(A) = 17 \\ h(S) &\leq cost(S, B) + h(B) = 12 \\ h(S) &= 12 \end{aligned}$$

  Making, $h(S)$ consistent with node A and B.

3.6 How to choose a better Heuristic:

In the previous lecture, we have seen that there are many possible ways to design the heuristics for a problem space. We want to know which heuristic function should be selected when we have more than one way of computing admissible and consistent heuristics.

1. **Effective Branching Factor:** The effective branching factor **(EBF)** is a measure used in tree search algorithms to provide a quantitative description of the tree's growth rate. It reflects how many children each node has, on average, in the search tree that needs to be generated to find a solution. Mathematically, the EBF is defined as the branching factor b for which:

   $$N + 1 = 1 + b + b^2 + b^3 + ... + b^d$$

   where N is the total number of nodes generated in the search tree and d is the depth of the shallowest solution.

   The EBF gives an insight into the efficiency of the search process, influenced heavily by the heuristic used:

   - **Lower EBF:** A lower EBF suggests that the heuristic is effective, as it leads to fewer nodes being expanded. This usually indicates a more directed and efficient search.
   - **Higher EBF:** A higher EBF suggests a less effective heuristic, as more nodes are being generated, indicating a broader search, which is generally less efficient.

   Calculating the EBF can help evaluate the practical performance of a heuristic. An ideal heuristic would reduce the EBF to the minimum necessary to find the optimal solution, indicating a highly efficient search strategy.

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A* with
h_1
(misplaced tiles), and A* with
h_2
(Manhattan distance). Data are averaged over 100 puzzles for each solution length
d
from 6 to 28.

In the above figure, Stuart and Russel generated random 8-puzzle problems and solved them with an uninformed breadth-first search and with A* search using both and reporting the average number of nodes generated and the corresponding effective branching factor for each search strategy and for each solution length. The results suggest that h_2 is better than h_1 and both are better than no heuristic at all.

2. **Dominating heuristic:** For two heuristic functions, h_1 and h_2, we say that h_1 dominates h_2 if for every node n in the search space, the following condition holds:

   $h_1(n) \geq h_2(n)$ and there is at least one node n where $h_1(n) > h_2(n)$ then we say that h_1 dominates h_2 or in other words, h_1 is the dominating heuristic.

3.6.1 Why is a dominant heuristic for efficient?

By now we should have an understanding that in A* algorithm every node, n with

$$f(n) \leq C^*,$$
$$\text{or, } h(n) \leq C^* - g(n) \text{ [}C^* \text{ is optimal path cost]}$$

is surely expanded.
Now let us consider, two heuristics, h_1 and h_2, both admissible and consistent. Where,

$$h_2 \geq h_1.$$

If node, n is on the optimal path, it will be expanded with A* algorithm for both the heuristics. Meaning,

$$h_1(n) \leq h_(2) \leq C^* - g(n).$$

If node, n is not on the optimal path, we have three possibilities,

- **Possibility 1:**
$$C^* - g(n) \leq h_1 \leq h_2,$$
in which case, node n will not be expanded at all, whichever heuristic we use.

- **Possibility 2:**
$$h_1(n) \leq h_(2) \leq C^* - g(n).$$
in this case, node, n will be expanded for both the heuristics.

- **Possibility 3:**
$$h_1(n) \leq C^* - g(n) \leq h_2(n),$$
where, node n will be expanded when h_1 is used but not when h_2 is used.

So, we see that the using h_1 may expand the same nodes that are expanded when h_2 is used. But it may end up expanding more unnecessary nodes than those expanded by using the dominant heuristic h_2.

You may want to go back to the last example in section 3.3 and notice that the number of nodes generated by using the dominant heuristic is significantly less for the 8-puzzle problem.

So, given that the heuristic is consistent and the computation time is not too long, it is generally a better idea to use higher valued heuristic.