Artificial Intelligence

Instructor	:	Ipshita Bonhi Upoma
Course Code	:	CSE422
Initial	:	IBU
Mail	:	ipshita.upoma@bracu.ac.bd
Time	:	Thursday/Saturday: 2:00 pm- 3:20 pm. Thursday/Saturday: 3:30 pm - 4:50 pm.
Office	:	CSE / MNS (beside CSE conference room)
Consultation hour	:	Thursday 5:00 pm to 6pm (Email beforehand)

Departmant of Computer Science and Engineering
BRAC University

Contents

Introduction

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.

Lecture Plan (Central):

https://docs.google.com/document/d/1SJFBkfkL0wHUokhqfcBRZLhNqtw6Qhjt/edit

Marks Distribution

- Class Task 5%
- Quiz 15%(4 questions, 3 will be counted)
- Assignment 10% (4 Assignsments, Bonus assignments?)
- Lab 20%
- Mid 20%
- Final 30%

Class rules:

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.

1 Lecture 1: Overview of the history of mathematical advances in Artificial Intelligence:

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

1. *1950s*

   - **Turing Test:** Introduced computational logic as a measure of machine intelligence.
   - **Foundational Theories:** AI's roots were grounded in mathematical logic and algorithmic theories.
   - Early Bayesian networks and the development of the Markov Decision Processes in the 1950s laid the groundwork for probabilistic reasoning in AI.

   Artificial Intelligence (AI) has a rich history with its roots in several foundational problems that researchers initially sought to solve using computational methods. Here are some of the earliest problems that AI was used to address:

Logic Puzzles and Games—

Chess: One of the first and most famous applications of AI was in playing chess. The idea was to create a machine that could challenge human intellect in this strategic game. In 1951, Alan Turings first program, designed to play chess, was one of the earliest instances.

Checkers: Arthur Samuel developed one of the first AI programs to play the game of checkers in the 1950s. This program learned from experience using a method that is now recognized as a form of machine learning.

Mathematical Theorems Proving—

Logic Theorist: Developed by Allen Newell, Herbert A. Simon, and Cliff Shaw in 1956, the Logic Theorist was designed to mimic human problem-solving skills and was able to prove 38 of the first 52 theorems in Whitehead and Russells Principia Mathematica.

Language Understanding and Translation—

Machine Translation: Early AI research also focused on translating texts from one language to another. The Georgetown experiment in 1954 was an early demonstration where more than sixty Russian sentences were automatically translated into English.

Problem-Solving Programs—

General Problem Solver (GPS): Developed by Newell and Simon in 1957, GPS was designed as a universal problem solver that could find solutions to a wide range of problems through heuristic search and problem decomposition.

Pattern Recognition—

Perceptrons: Frank Rosenblatts Perceptron, developed in 1957, was an early neural network used for simple pattern recognition tasks. This marked an early attempt to model human brain processes using machines.

Speech Recognition—

Audrey: Developed in the 1950s by Bell Laboratories, Audrey was an early speech recognition system that could recognize spoken digits from 0 to 9.

2. *1960s*

   - **Problem Solving Algorithms:** Development of algorithms like Dijkstras algorithm for efficient problem-solving.
   - **Perceptrons:** Early neural networks analyzed mathematically, highlighting their limitations.
   - **Control Theory:** Started being used in robotics and automated systems, integrating with AI to develop systems capable of adaptive and optimal control strategies.

3. *1970s*

- **AI Criticisms:** Mathematical criticisms such as the Lighthill Report highlighted the unfulfilled promises of AI based on its current mathematical capabilities.
- **Graph theory:** Became prominent with the development of semantic networks and the rise of expert systems which used graph-based data structures to represent knowledge.

4. *1980s*
    - **Expert Systems:** Utilized rule-based deduction, which relies heavily on logical programming.
    - **Backpropagation Algorithm:** Introduced for training multi-layer neural networks, significantly improving their performance and capabilities.
5. *1990s*
    - **IBM's Deep Blue:** Employed complex algorithms including minimax and evaluation functions for strategic gameplay in chess.
    - **Fourier transforms and signal processing techniques:** Became crucial in the 1980s and 1990s for feature extraction in AI applications like speech and image recognition, enabling AI to interpret complex sensory data more effectively.
    - **Rise of Statistical Learning:** Support Vector Machines and other machine learning models became popular, emphasizing data-driven AI approaches.
6. *2000s*
    - **Game Theory:** As artificial intelligence began tackling more complex scenarios involving multiple decision-makers, such as in economics and multi-agent systems, game theory provided strategies for adversarial and cooperative interactions.
    - **Deep Learning:** Re-introduction and emphasis on multi-layer neural networks capable of handling complex pattern recognition.
    - **Probabilistic Models:** Bayesian networks and other probabilistic models became integral for improving AIs decision-making capabilities.
7. *2010s*
    - **IBM's Watson:** Featured advanced natural language processing and machine learning algorithms, including logistic regression for pattern recognition.
    - **Google DeepMind's AlphaGo:** Employed Monte Carlo tree search and deep reinforcement learning, showcasing superior problem-solving skills in the game of Go.
8. *2020s*
    - **Algorithmic Fairness and Bias Mitigation:** Focused on developing mathematical techniques to address and mitigate bias in AI algorithms.
    - **Quantum Computing:** Emerging discussions on how quantum computing could revolutionize AI computation methodologies.

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.

1.3 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.

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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.

Problem Formulation and choice of strategies

Why— 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 processidentifying 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.

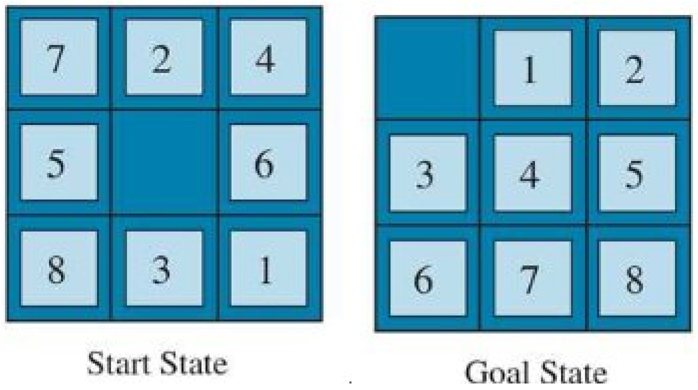
2 Lecture 2: Problem Formulation and Informed Search

2.1 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 initial configuration of a puzzle.



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**

   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.

2.2 Examples of problem formulation

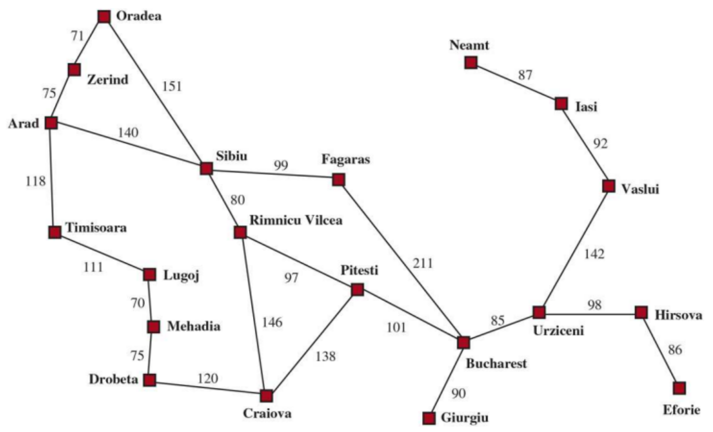
1. **City Traffic Navigation (Solved by Informed Search)**
   - **Problem Formulation:**
     - **Goal:** To find the quickest route from a starting point (origin) to a destination (end point) while considering current traffic conditions.
     - **States:** Each state represents a geographic location within the citys 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 plants 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.

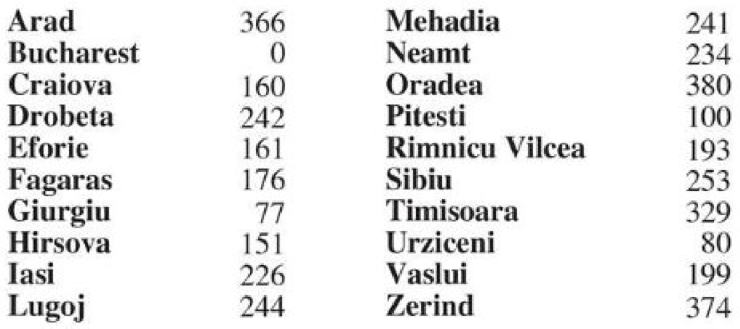
4. **Disease Diagnosis (Solved by Decision Trees)**
  - **Problem Formulation:**
    - **Goal:** To accurately diagnose diseases based on symptoms, patient history, and test results.
    - **States:** Each state represents a set of features associated with a patient, including symptoms presented, medical 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.

2.3 Search Algorithms

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 Timioara 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 informationreferred to as heuristicsto 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.

2.3.1 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.

2.3.2 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).

2.3.3 Why do we use heuristics?

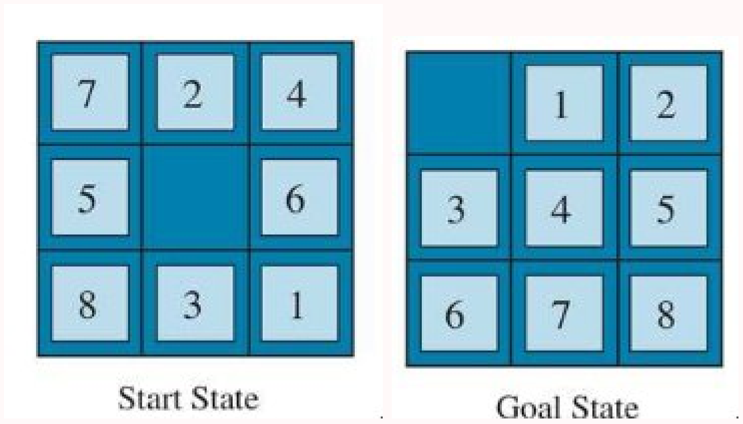
Guidance: The heuristic guides the search process, helping the algorithm prioritize which nodes to explore next based on which seem most promisingi.e., likely to lead to the goal with the least cost.

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.

2.4 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.

1. **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.

2. **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: $t_{adjusted} = t \times w$
.

3. **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.

4. **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 pages 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.

5. **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.

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

6. **Natural Language Processing (NLP):**

Word Probability: $P(w \mid \text{context})$ 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.

7. **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.

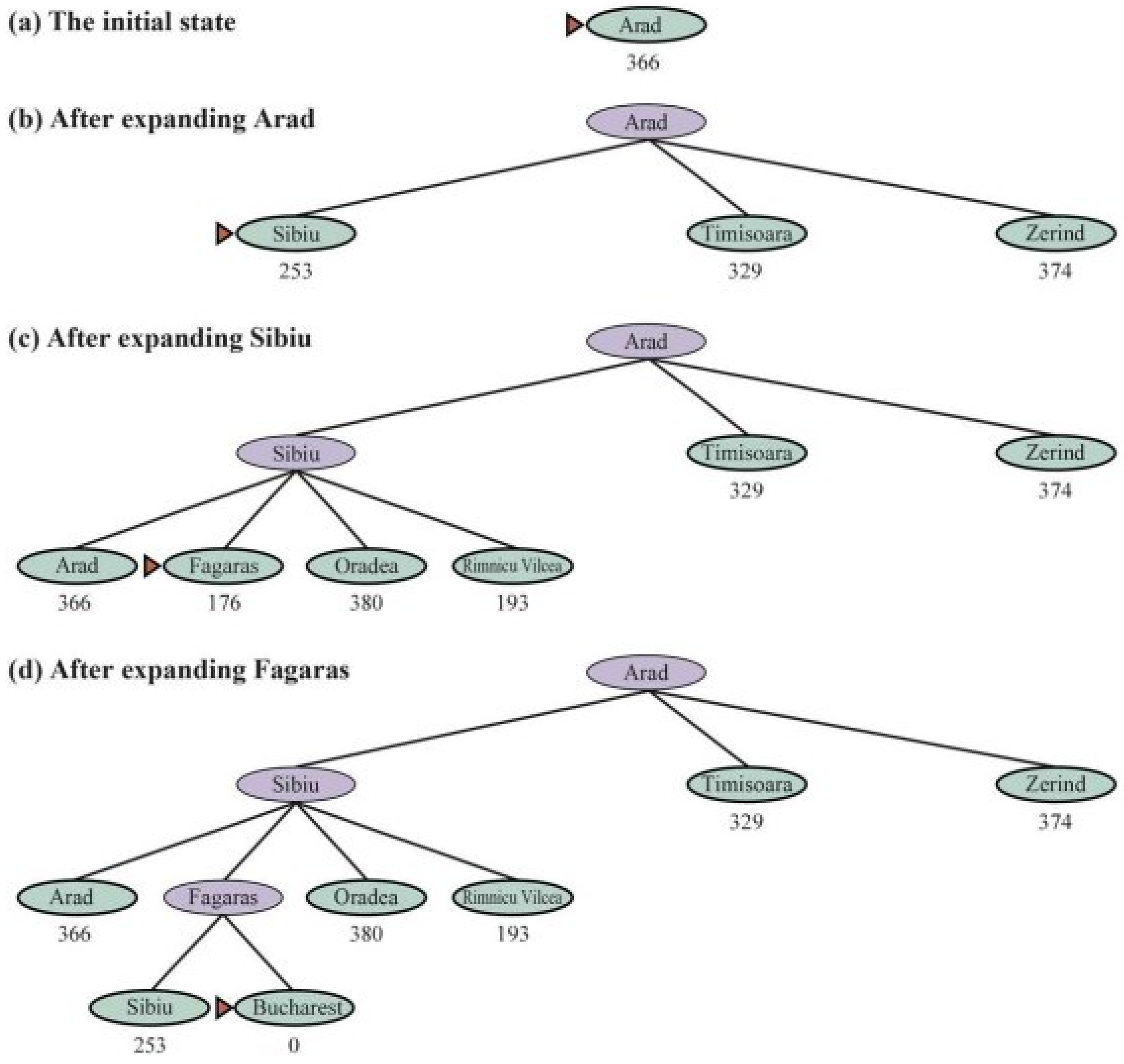
2.5 Greedy Best First Search- Finally an algorithm

Greedy best-first search is a form of best-first search that expands first the node with the lowest $h(n)$ valuethe node that appears to be closest to the goalon 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 greedyon 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 $O(bm)$.

2.5.1 Algorithm: Greedy Best-First Search

- `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`.

2.5.2 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 paths 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.**

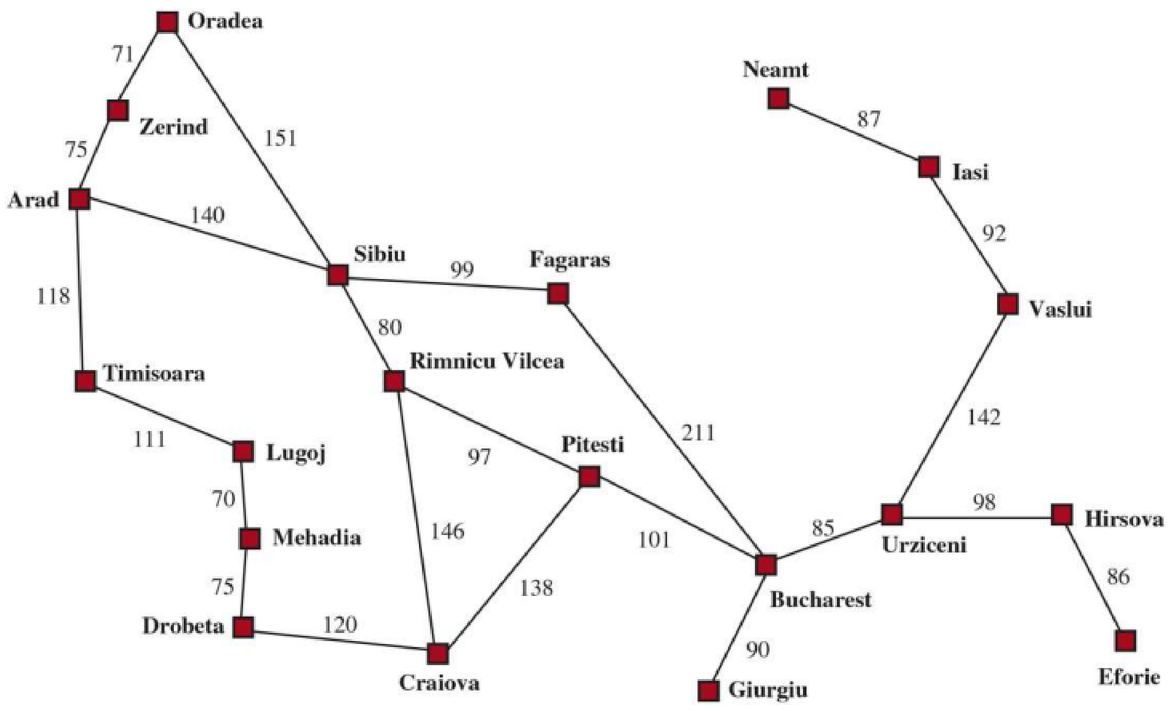
3 Lecture 3:

3.1 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. subsectionAlgorithm: A* Search

- `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 with a priority of `0 + heuristic(start)`.
    - ☐ Set `g_score[start]` to `0` (cost from start to `start`).
    - ☐ Set `f_score[start]` to `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 `f_score` 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:
        - ▪ Calculate `tentative_g_score` as `g_score[current] + cost(current, neighbor)`.
        - ▪ If `tentative_g_score` is less than `g_score[neighbor]` (or neighbor is not in `priority queue`):
          - – Update `cameFrom[neighbor]` to `current`.
          - – Update `g_score[neighbor]` to `tentative_g_score`.
          - – Update `f_score[neighbor]` to `tentative_g_score + 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`.

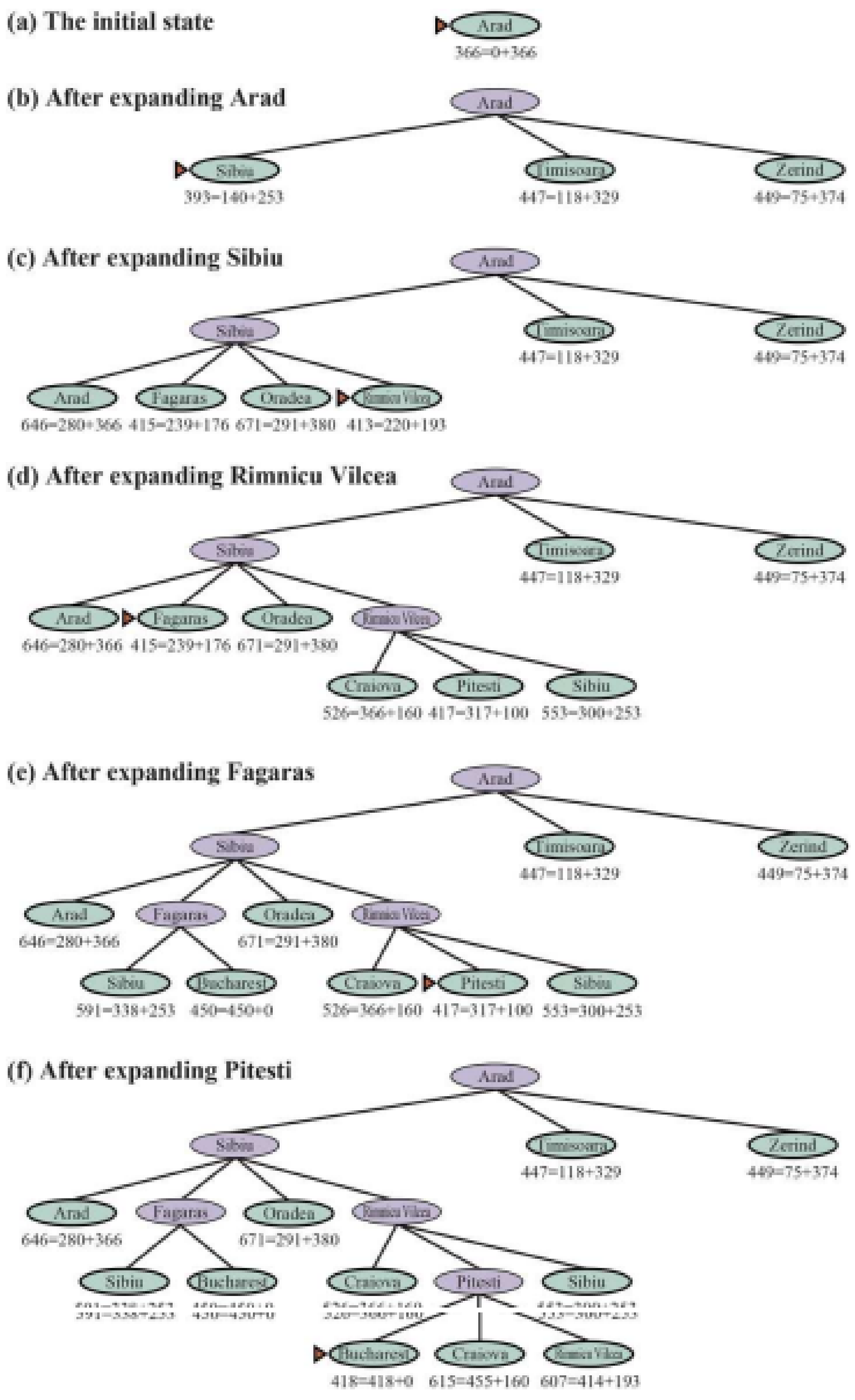
** **Path Reconstruction:** The path is reconstructed from the cameFrom map, which records where each node was reached from.



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.


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

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 Pitestis 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.

3.2 Condition on heuristics for A* to be optimal:

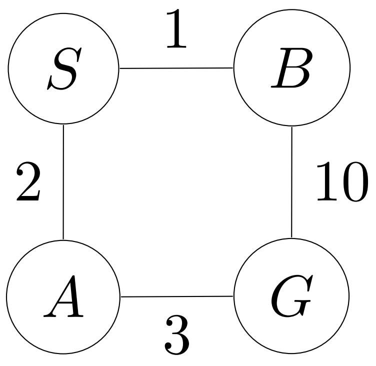
1. **Admissibility:** An admissible heuristic never overestimates the cost to reach the goal. This makes it optimistic about the path costs.

   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) 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:

$$g(A) = 2, f(A) = g(A) + h(A) = 2 + 10 = 12$$

For B:

$$g(B) = 1, f(B) = g(B) + h(B) = 1 + 2 = 3$$

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:

$$g(G) = 11, f(G) = g(G) + h(G) = 11 + 0 = 11$$

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.

2. **Consistency:** 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. Heres how it ensures that A* returns an optimal path:

   Definition of Consistency:

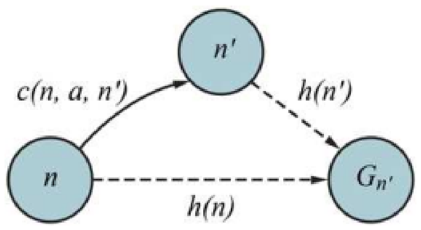
   A heuristic h is consistent if for every node n and every successor n' of n, the estimated cost from n to the goal, denoted $h(n)$, does not exceed the cost from n to n' plus the estimated cost from n' to the goal, $h(n')$. Mathematically, this is represented as: $h(n) \leq c(n, n') + h(n')$ where $c(n,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

   Ensuring 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.

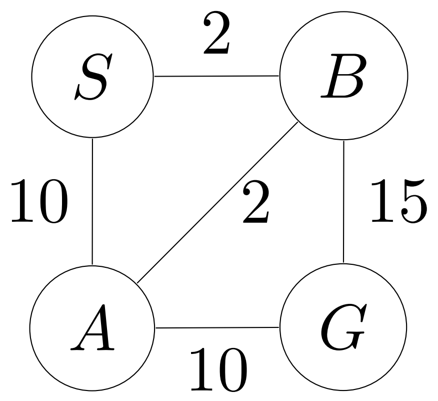
- This prevents the algorithm from revisiting nodes unnecessarily, thereby ensuring efficiency in path finding.

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.

3.3 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:

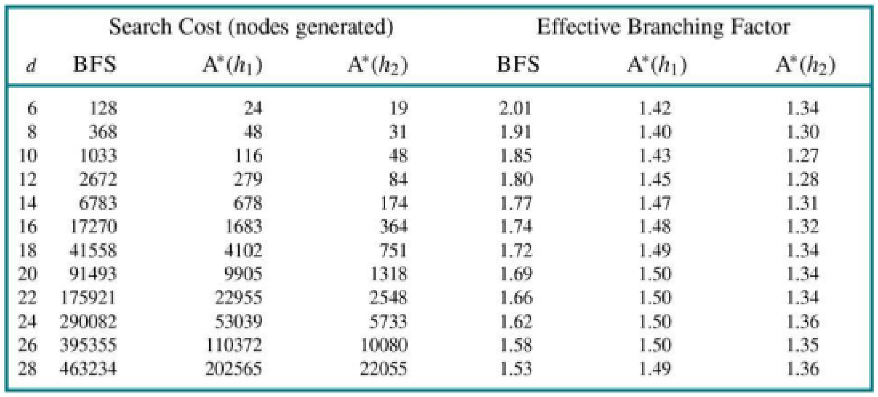
$$N + 1 = 1 + b + b^2 + b^3 + ... + b^d$$

   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 is better than 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.