Artificial Intelligence

For naturally intelligent beings

CSE422 Lecture Notes

First Edition

Ipshita Bonhi Upoma

BRAC UNIVERSITY

Inspiring Excellence

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**

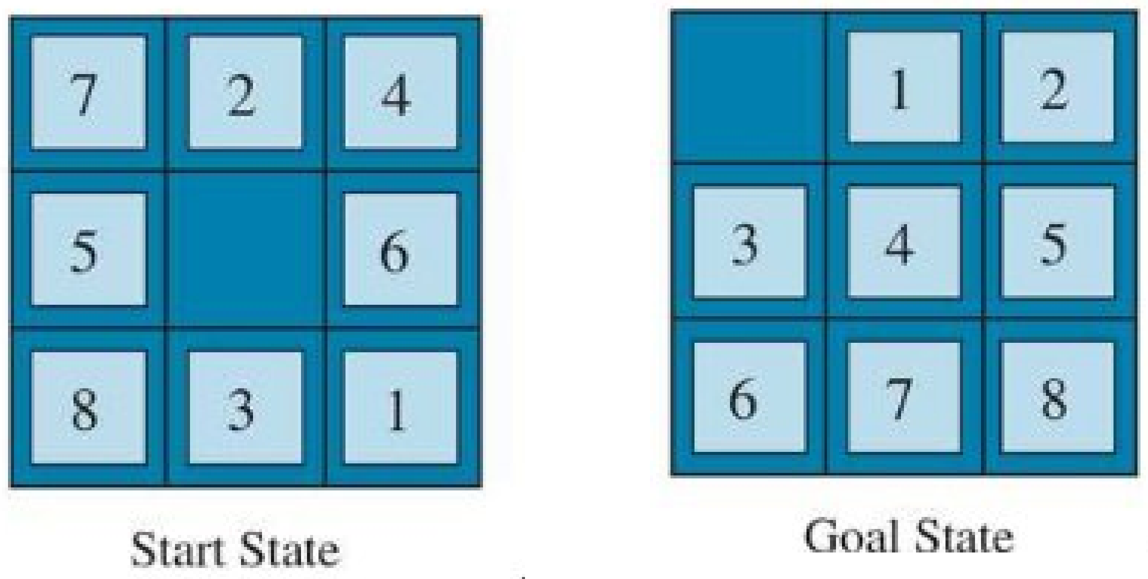


Figure 2.1: Initial and Goal State of a 8-Puzzle Game (Russel and Norvig, *Artificial Intelligence: A Modern Approach*)

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.

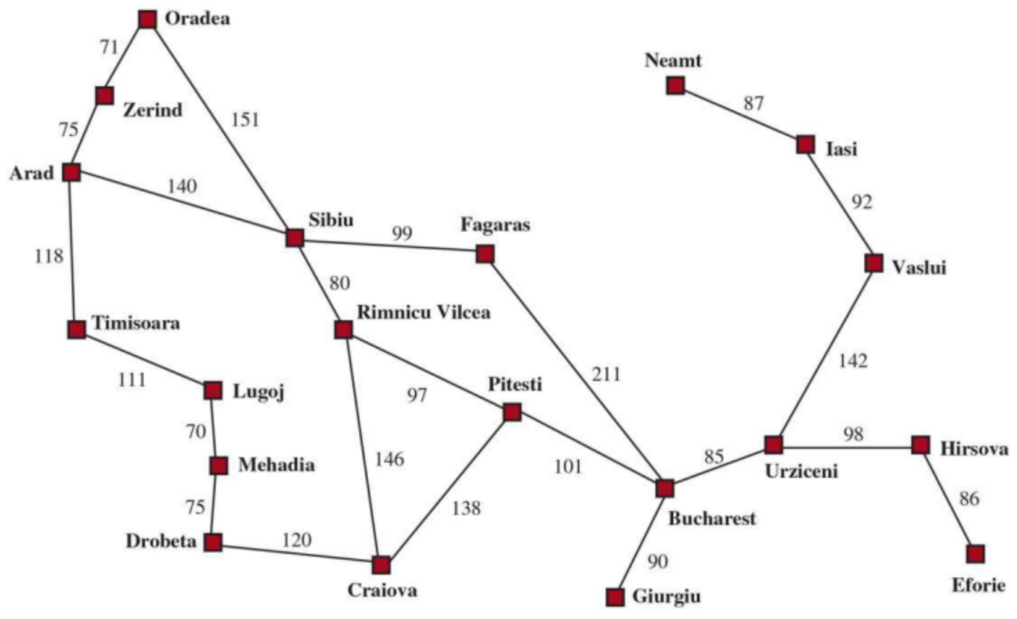
- **Demographic Data:** Age, sex, genetic information, lifestyle factors.
- **Medical History:** Previous diagnoses, treatments, family medical history.

CHAPTER 3

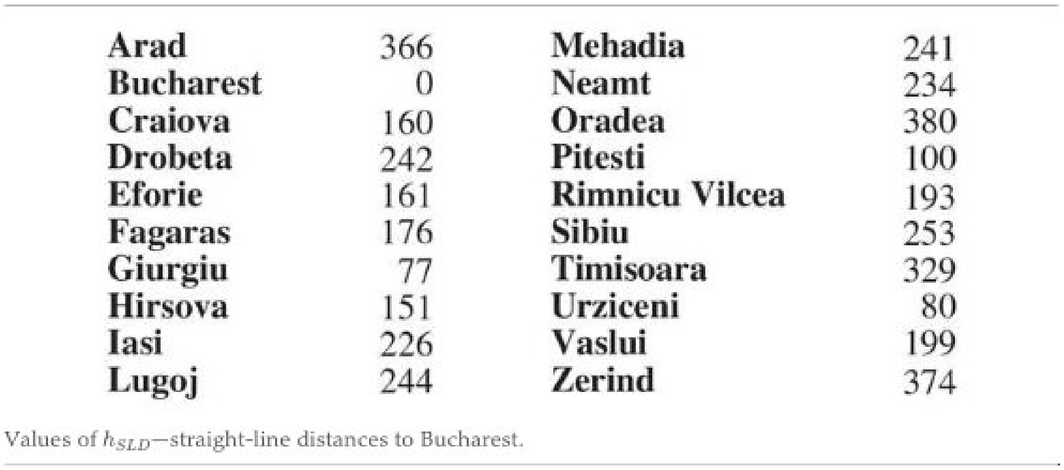
INFORMED SEARCH ALGORITHMS

Note: This lecture closely follows Chapter 3.6 (Heuristic Functions), 3.5 (Informed Search) to 3.5.1 (Greedy Best First Search), 3.5.2 to 3.5.4 (A* Search), 3.6.1 (Effect of Heuristic on accuracy and performance) 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.

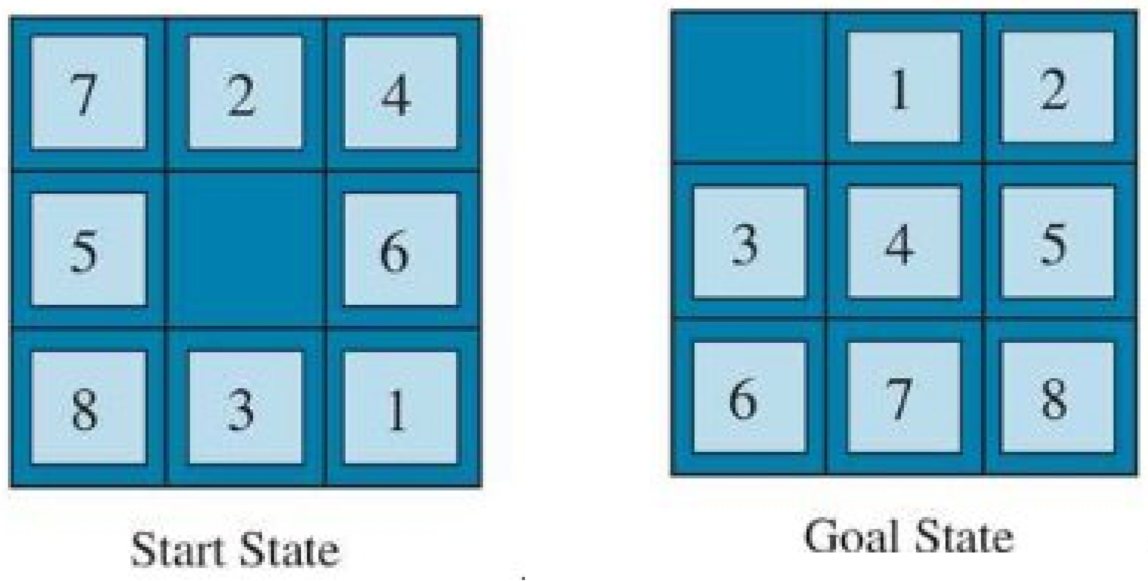


Figure 3.1: Initial and Goal State of a 8-Puzzle Game (Russel and Norvig, *Artificial Intelligence: A Modern Approach*)

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.

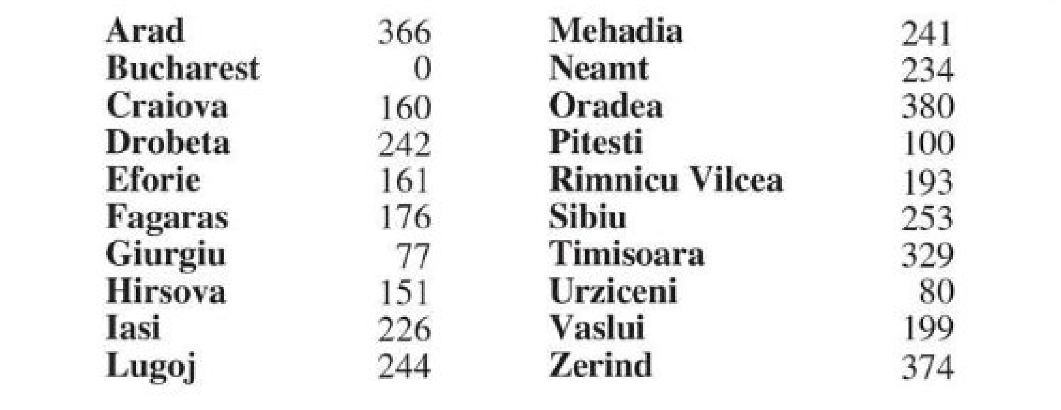
Collaborative Filtering: Using preferences of similar users to recommend items. Matrix factorization techniques or neighbor-based algorithms to predict user preferences.

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 $f(n) = h(n)$.

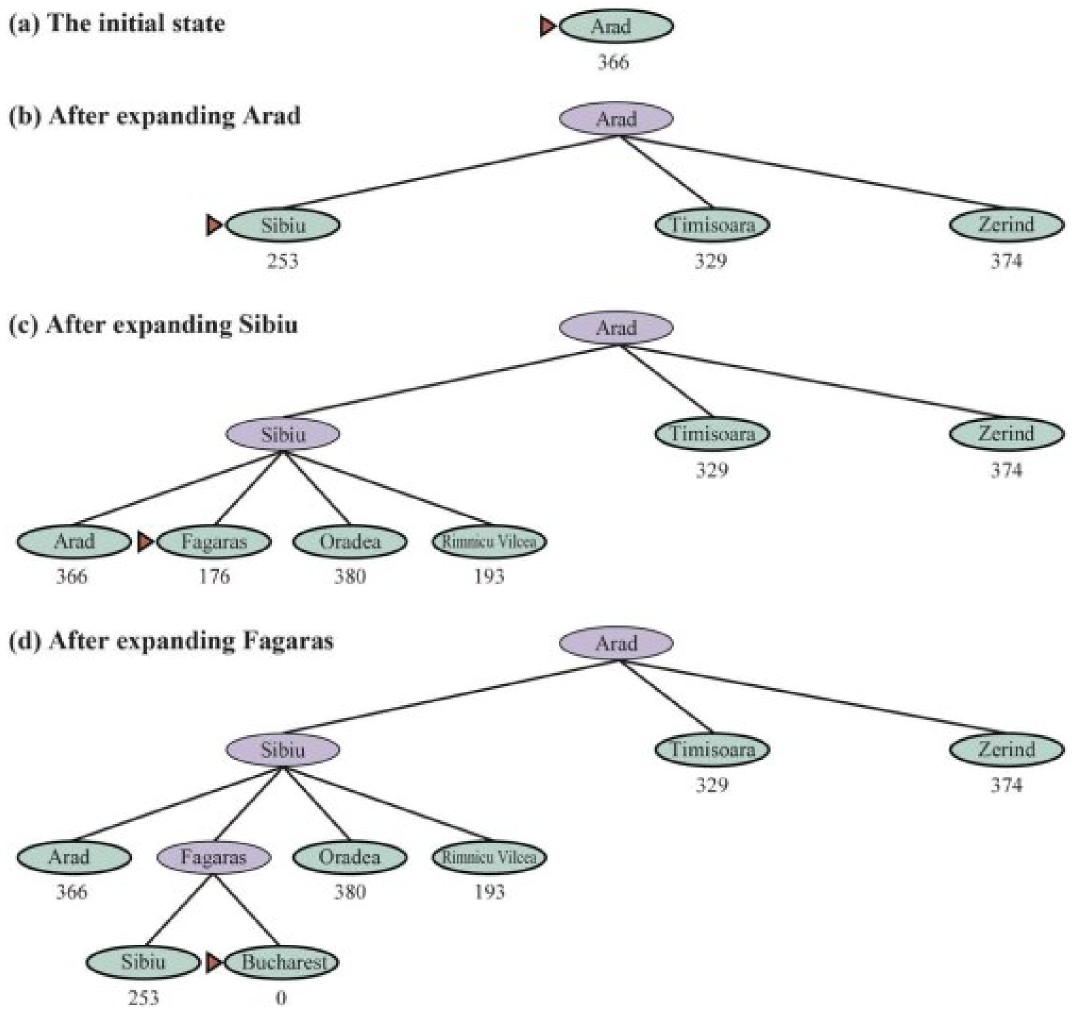
Let us see how this works for route-finding problems in Romania; we use the **straight-line-distance** heuristic, which we will call h_{SLD} . If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in the figure below. For example, $h_{SLD}(Arad) = 366$. Notice 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 pri-
              ority 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.

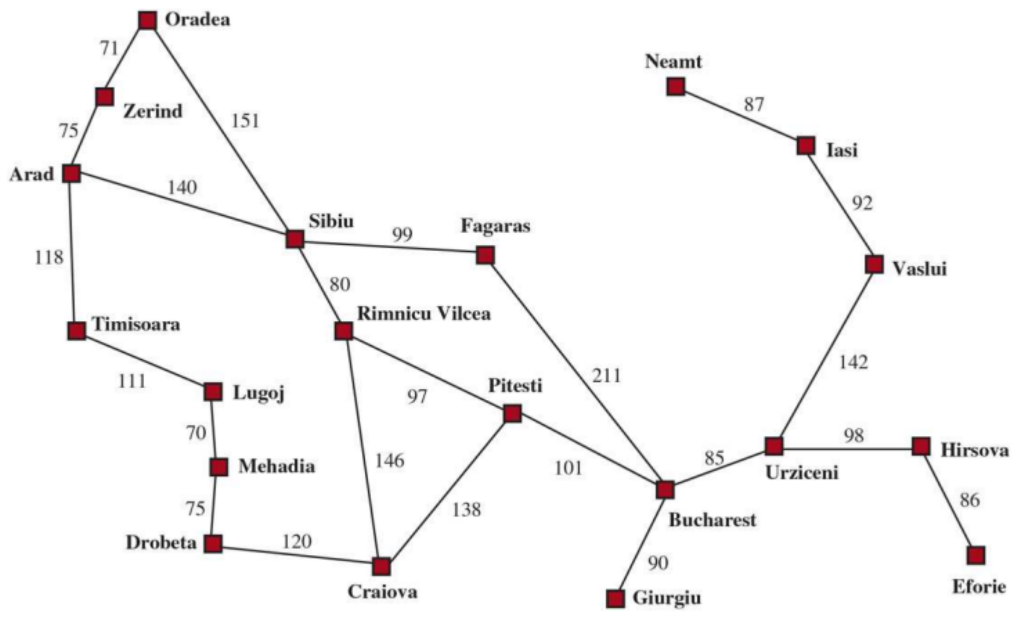
- ☐ Set `gScore[start]` to `0` (cost from start to `start`).
- ☐ Set `fScore[start]` $= 0 +$ `heuristic(start)` (total estimated cost from start to goal through `start`).
- ☐ Define `cameFrom` to store the path reconstruction data.

– 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.

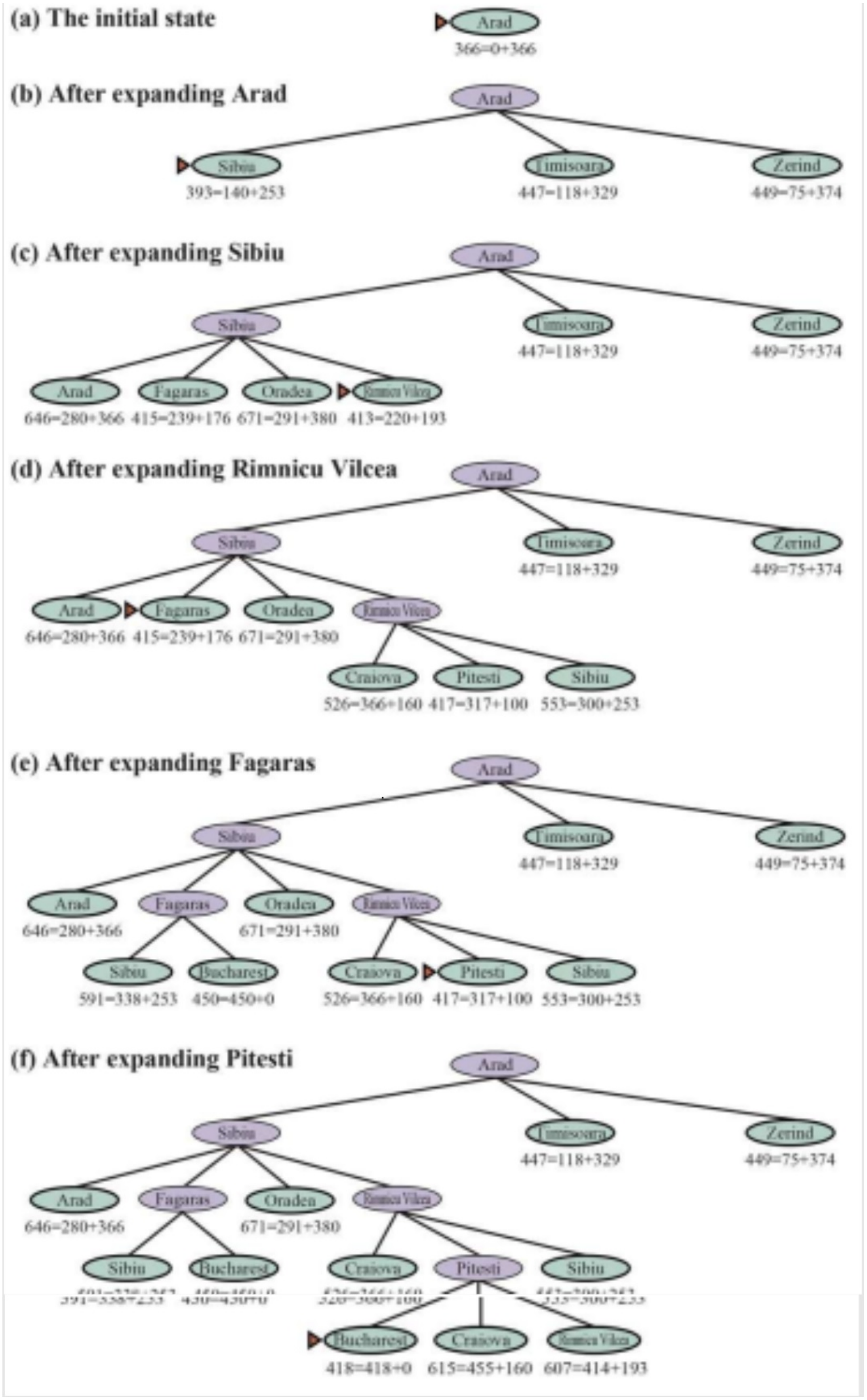


Figure 3.2: Simulation of A* Algorithm to find a path from Arad to Bucharest.

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.

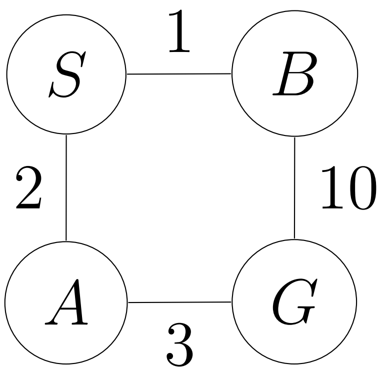
Proof of Cost-Optimality (via Contradiction)

- 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.

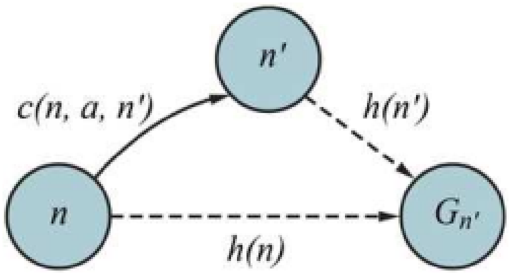
A consistent heuristic, also known as a monotonic heuristic, is a stronger condition than admissibility and plays a crucial role in ensuring that A* search finds the optimal path. Here's how it ensures that A* returns an optimal path:

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.

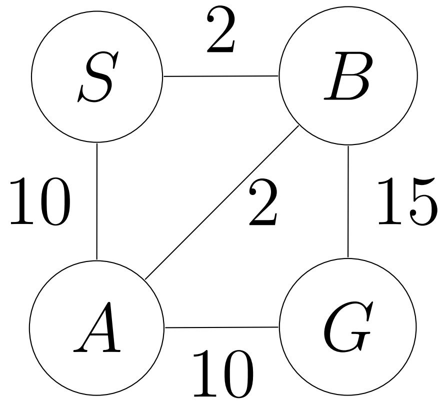
4. **Optimality Guarantee:**

   - 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:

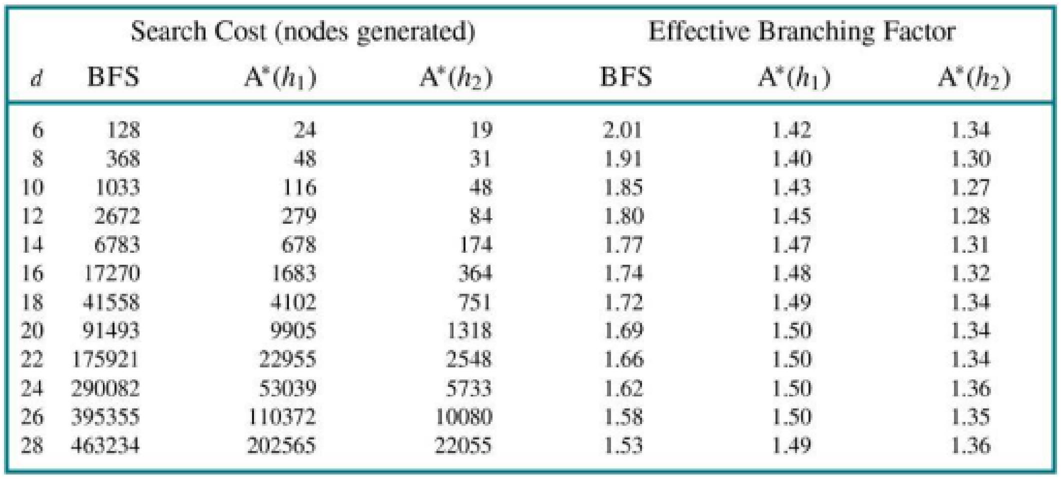
   $$N + 1 = 1 + b + b^2 + b^3 + ... + b^d$$

   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.

CHAPTER 4

LOCAL SEARCH

Note: This lecture closely follows Chapter 4 to 4.1.1 (Local Search and Hill Climbing), 4.1.2 (Simulated Annealing) and 4.1.4 (Evolutionary algorithm) of Russel and Norvig, *Artificial Intelligence: A Modern Approach*.

4.1 Local Search

So far, we have utilized search strategies like A* search and the Greedy Best Search algorithm to navigate vast solution spaces and find sequences of actions that lead to optimal solutions. These strategies use heuristics to estimate costs from any node to the goal, improving efficiency by focusing on promising areas of the state space. However, the effectiveness of these methods depends heavily on the quality of the heuristic used. Informed search strategies are particularly useful for finding paths to the goal state. For example, solving the 8-puzzle requires a series of consecutive actions (such as moving the empty space up, down, left, or right) to reach the solution; similarly, traveling from Arad to Bucharest involves a sequence of actions moving from one connected city to another.

On the other hand, some problems focus solely on generating a goal state or a good state, regardless of the specific actions taken. For instance, in a simplified knapsack problem, the solution involves selecting a set of items that maximizes reward without exceeding the weight limit. The process does not concern itself with the order in which items are checked or selected to achieve the maximum reward. Local search strategies can be used to solve such problems. It is especially useful in problems with very large search spaces.

Local search algorithms offer a practical solution for tackling large and complex problems. Unlike global search methods that attempt to cover the entire state space, local search begins with an initial setup and gradually refines it. It makes incremental adjustments to the solution, exploring nearby possibilities through methods like Hill Climbing, Simulated Annealing, and Genetic Algorithms. These techniques adjust parts of the current solution to systematically explore the immediate area, known as the "neighborhood," for better solutions. This approach is particularly effective in environments with numerous potential solutions (local optima), helping to find an optimal solution more efficiently.

4.1.1 The Need for Local Search

Local search is particularly useful in:

- **Large or poorly understood search spaces, where exhaustive global search is impractical.** For example, in large scheduling problems where the number of potential schedules increases exponentially with the number of tasks and resources.
- **Optimization tasks, focusing on improving a measure rather than finding a path.** This is seen in machine learning hyperparameter tuning, where algorithms like Gradient Descent and its variants iteratively adjust parameters to minimize a loss function.
- **Dynamic problems, where solutions must adapt to changes without restarting the search.** A common example is in real-time strategy games, where AI opponents must continuously adjust their strategies based on the player's actions.
- Local search also complements hybrid algorithms, **combining its strengths with other strategies for enhanced performance** across various problems. For instance, Genetic Algorithms use local search within their crossover and mutation phases to fine-tune solutions.

In summary, local search provides a flexible and efficient approach for refining solutions incrementally, making it a critical tool in modern computational problem-solving. These strategies bridge the gap between the theoretical optimality of informed search and the practical needs of real-world applications.

4.1.2 Examples of problems that can be solved by local search:

Local search algorithms are versatile tools for tackling various optimization problems across many fields. Here are some example problems where local search algorithms are particularly effective:

Traveling Salesman Problem (TSP): In the TSP, the goal is to find the shortest possible route that visits a list of cities and returns to the origin city. A local search algorithm like Simulated Annealing or 2-opt (a simple local search used in TSP) can iteratively improve a given route by swapping the order of visits to reduce the overall travel distance.

Knapsack Problem: As previously mentioned, the goal here is to maximize the value of items placed in a knapsack without exceeding its weight capacity. Local search techniques can be used to iteratively add or remove items from the knapsack to find a combination that offers the highest value without breaching the weight limit.

Max Cut Problem: This is a problem in which the vertices of a graph need to be divided into two disjoint subsets to maximize the number of edges between the subsets. Local search can adjust the placement of vertices in subsets to try and maximize the number of edges that cross between them.

Protein Folding: In computational biology, protein folding simulations involve finding low-energy configurations of a chain of amino acids. Local search can be used to tweak the configuration of the protein to find the structure with the lowest possible energy state.

Layout Design: In manufacturing and architectural design, layout problems involve the placement of equipment or rooms to minimize the cost of moving materials or to maximize accessibility. Local search can rearrange the placements iteratively to improve the overall layout efficiency.

Parameter Tuning in Machine Learning: Tuning hyperparameters of a machine learning model to minimize prediction error or maximize model performance can also be approached as an optimization problem. Techniques like Grid Search, Random Search, or more sophisticated local search methods can be applied to find optimal parameter settings.

4.1.3 Some Constraint Satisfaction Problems can also be solved using local search strategies:

Graph Coloring: This problem requires assigning colors to the vertices of a graph so that no two adjacent vertices share the same color, using the minimum number of colors. Local search can explore solutions by changing the colors of certain vertices to reduce conflicts or the number of colors used.

Job Scheduling Problems: In job scheduling, the task is to assign jobs to resources (like machines or workstations) in a way that minimizes the total time to complete all jobs or maximizes throughput. Job scheduling can be viewed as a CSP when the task is to assign start times to various jobs subject to constraints such as job dependencies (certain jobs must be completed before others can start), resource availability (jobs requiring the same resource cannot overlap), and deadlines. Local search can be used to iteratively shift jobs between resources or reorder jobs to find a more efficient schedule.

Vehicle Routing Problem Similar to TSP but more complex, this problem involves multiple vehicles and aims to optimize the delivery routes from a central depot to various customers. This problem can also be modeled as a CSP, where constraints might include vehicle capacity limits, delivery time windows, and the requirement that each route must start and end at a depot. Local search can adjust routes by reassigning customers to different vehicles or changing the order of stops to minimize total distance or cost.

Local search algorithms are ideal for these and many other problems because they can provide high-quality solutions efficiently, even when the search space is extremely large and complex. They are particularly valuable when exact methods are computationally infeasible, and an approximate solution is acceptable. Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are **not systematic—they might never explore a portion of the search space where a solution actually resides.** However, they have **two key advantages:**

1. **They use very little memory; and**
2. **They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.**

4.2 Local Search Algorithms:

In this course we are going to learn about:

1. Hill-Climbing Algorithm / Gradient Descent
2. Problems of Hill Climbing Algorithms and Remedies
3. Simulated Annealing
4. Genetic Algorithm

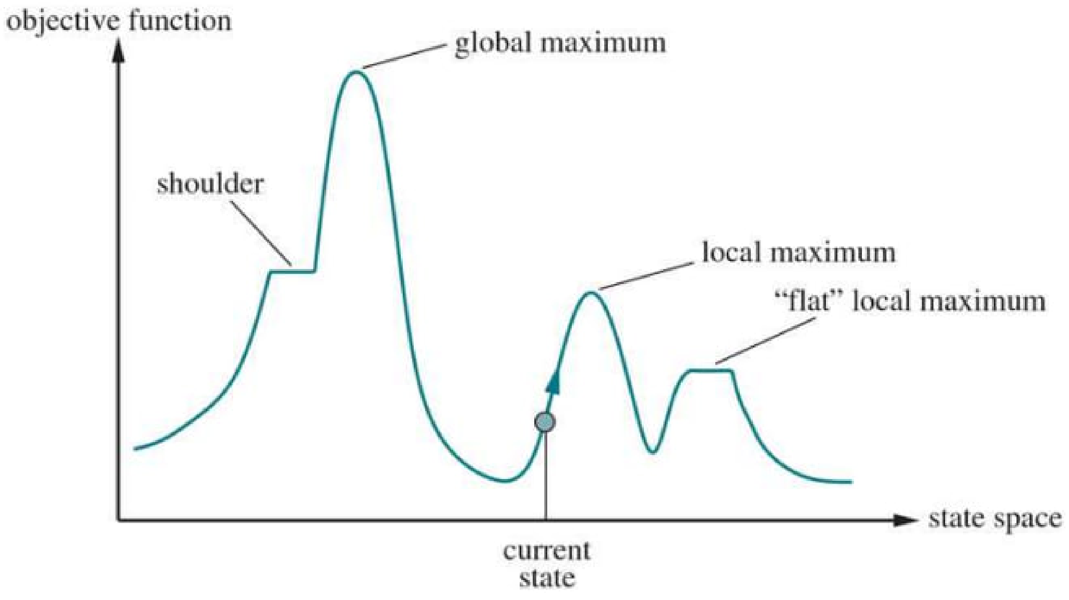
4.2.1 State-Space and Objective Function:

Local search algorithms are powerful tools for addressing optimization problems, where the objective is to find an optimal state that maximizes or minimizes a given objective function. These algorithms navigate through a metaphorical "state-space landscape," where each point or state in this landscape represents a possible solution with a specific "elevation" defined by the objective function. The concept can be more clearly understood through the following points, illustrated with examples:

1. **Understanding the State-Space Landscape**

Concept: Imagine the state-space of a problem as a geographical landscape where every point represents a possible solution. The elevation at each point is determined by the value of the objective function at that state.

Elevation as Objective Function: In optimization terminology, elevation could represent a value or a cost associated with each state. High elevations indicate better values in maximization problems, and lower elevations indicate lower costs in minimization problems.



A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

2. **Objective of Local Search**

   Maximization (Hill Climbing): Here, the goal is to find the highest point in the landscape, akin to climbing to the peak of a hill. The algorithm iteratively moves to higher elevations, seeking to locate the highest peak, which represents the optimal solution.

   Example: Maximizing sales in a retail chain by adjusting variables such as pricing, marketing spend, and store layout. Each adjustment is a "step" in the landscape, seeking higher profits (higher elevations).

   Minimization (Gradient Descent): In contrast, this approach aims to find the lowest point or valley in the landscape. This is suitable for cost reduction problems where each step attempts to move to lower elevations, minimizing the objective function.

Example: Minimizing production costs in a manufacturing process by altering materials, labor, and energy usage. Each configuration change is a step toward lower costs.

3. **Examples of Local Search Algorithms in Action**

   i. **Traveling Salesman Problem (TSP):**

      Objective function: Minimize the total travel distance for a salesman needing to visit multiple cities and return to the starting point.

      Local Search Strategy: Start with a random route and iteratively improve it by swapping the order of cities if it results in a shorter route (seeking lower valleys in terms of distance).

   ii. **Knapsack Problem**

      Objective function: Maximize the value of items placed in a knapsack without exceeding its weight capacity.

      Local search Strategy: Begin with a random combination of items. Iteratively add or remove items from the knapsack to find a combination that offers the highest value without breaching the weight limit.

   iii. **8-queen problem**

      Objective function: Minimize the number of pairs of queens that are attacking each other either horizontally, vertically, or diagonally. The ideal goal is to reduce this number to zero, indicating that no queens are threatening each other.

      Local search strategy: Start with a random instance of the board and iteratively improving the placement of queens on the board.

4.3 Hill-Climbing Search:

Definition 4.3.1: Hill-Climbing Algorithm

The Hill Climbing search algorithm maintains one current state and iteratively moves to the neighboring state with the highest value, effectively following the steepest path upward. It stops when it reaches a "peak," meaning no adjacent state offers a better value. This algorithm only considers the immediate neighbors and does not plan beyond them.

Algorithm 3 Local Maximum Search

```
1: Input: initial_state, objective_function()
2: Output: state that is a local maximum
3: begin
4:   current_state ← initial_state
5:   loop do
6:     neighbor ← a state that is a neighbor of current_state
7:     If objective_function(neighbor) ≤ objective_function(current_state)
8:       Return current_state
9:     current_state ← neighbor
```

Definition 4.3.2: Gradient descent algorithm:

When the goal of the hill-climbing algorithm is to find the state with the minimum value rather than maximizing the objective, it is known as gradient descent.

4.3.1 Examples:

Simplified Knapsack Problem Setup:

Items (value, weight):
Item 1: Value = \$10, Weight = 2 kg
Item 2: Value = \$15, Weight = 3 kg
Item 3: Value = \$7, Weight = 1 kg
Item 4: Value = \$20, Weight = 4 kg
Item 5: Value = \$8, Weight = 1 kg

Knapsack Capacity: 7 kg

Objective: Maximize the total value of the items in the knapsack such that the total weight does not exceed 7 kg.

Hill Climbing Algorithm:

Initial Solution Start with a randomly selected set of items that do not exceed the capacity. For this example, let's start with the 1st, 2nd and 3rd item in the knapsack.

The knapsack can be represented as a string of 1s and 0s, where a '1' at i'th position indicates that the corresponding (i'th) item has been included, and a '0' means the item has not been taken.

Initial Solution: 11100, weight: 6kg, value: 32

Neighbor Generation: Generate neighboring solutions by toggling the inclusion of each item. For instance, if an item is not in the knapsack, consider adding it; if it is in the knapsack, consider removing it or replace an item with another item not in the solution.

Evaluate and Select: Calculate the total value and weight for each neighbor. If a neighbor exceeds the knapsack's capacity, discard it.

Choose the neighbor with the highest value that does not exceed the weight capacity.

Iteration: Repeat the process of generating and evaluating neighbors from the current solution.

If no neighbors have a higher value than the current solution, terminate the algorithm.

Termination: The algorithm stops when it finds a solution where no neighboring configurations offer an improvement.

Demonstration (for two iterations):
Step 1: Initial Solution
Knapsack: 11100
Total Value: $32
Total Weight: 6 kg

Step 2: Generate Neighbors Add Item 4: 11110, Total Value = $52, Total Weight = 10 kg
Add Item 5: 11101, Total Value = $40, Total Weight = 7 kg
Replace Item 2 with Item 4: 10110, Total Value = $37, Total weight: 7kg/
. . . . There can be other possible neighbors.

Step 3: Evaluate and Select
Discard Neighbor: 11110 as it exceeds knapsack weight limit.
Best neighbor: 11101 Total Value = 40, Total Weight = 7 kg

Step 4: Iteration (next steps)
Current configuration: 11101
Add Item 4: 11111, Total Value = $60, Total Weight = 11 kg Replace items with Item 4: All will exceed the weight

Evaluation:
All neighbors are discarded. No better valued neighbor.
Terminates as no better valued neighbor is found.

4.3.2 Key Characteristic:

Hill-Climbing Search is essentially a greedy approach, meaning it always looks for the best immediate improvement at each step, without considering long-term consequences. This can sometimes lead to suboptimal solutions.

4.3.3 Drawbacks:

Hill climbing algorithms may encounter several challenges that can cause them to get stuck before reaching the global maximum. These challenges include:

1. **Local Maxima:** A local maximum is a peak that is higher than each of its neighboring states but is lower than the global maximum. Hill climbing algorithms that reach a local maximum are drawn upward toward the peak but then have nowhere else to go because all nearby moves lead to lower values.

   Example:

   8-queen problem:

   Background: In a 8-queens problem, our goal is to maximize the number of non-attacking pairs of queens. The maximum number of pairs possible from 8 queens is 8*7/2 =28. So, we want all 28 pairs to be in non-attacking positions.

   Let us represent the instances as an array of 8 numbers, ranging from 1 to 8 denoting the columns and the index, ranging from 1 to 8 (sorry, programmers!) denoting the rows.

   Scenario: For an instance of the 8-puzzle, 13528647, there are 5 attacking pairs or in other words, 23 non-attacking pairs. This configuration is a local maximum because it's better than all immediate neighboring configurations, but it is not the global solution since it's not conflict-free.

   Problem: The Hill Climbing algorithm would stop here because all single-move alternatives lead to worse configurations, increasing the number of threats. Despite the presence of better configurations (global maxima with zero threats), the algorithm gets stuck.

2. **Plateaus:** A plateau is an area of the state-space landscape where the elevation (or the value of the objective function) remains constant. Hill climbing can become stuck on a plateau because there is no upward direction to follow. Plateaus can be particularly challenging when they are flat local maxima, with no higher neighboring states to escape to, or when they are shoulders, which are flat but eventually lead to higher areas. On a plateau, the algorithm may wander aimlessly without making any progress. **Sce-**

   nario: Imagine a large part of the chessboard setup where several queens are placed in such a manner that moving any one of them doesn't change the number of conflicts—it remains constant. This flat area in the search landscape is a plateau.

   For Example, the instance, 13572864 has 3 attacking pairs. Swapping the last two queens might not immediately lead to an increase in non-attacking pairs, resulting in a plateau where many configurations have an equal number of non-attacking pairs.

Problem: The Hill Climbing algorithm would find it difficult to detect any better move since all look equally non-promising.

On a plateau, every move neither improves nor worsens the situation, causing Hill Climbing to wander aimlessly without clear direction toward improvement. This lack of gradient (change in the number of conflicts) can trap the algorithm in non-productive cycles, preventing it from reaching configurations that might lead to the global maximum.

3. **Ridges:** Ridges are sequences of local maxima that make it very difficult for greedy algorithms like hill climbing to navigate effectively. Because the algorithm typically makes decisions based on immediate local gains, it struggles to cross over ridges that require a temporary decrease in value to ultimately reach a higher peak.

   These challenges highlight the limitations of hill climbing algorithms in exploring complex landscapes with multiple peaks and flat areas, making them susceptible to getting stuck without reaching the best possible solution.

   Scenario: Consider a situation where moving from one configuration to another better configuration involves moving through a worse one. For example, the instance 13131313 creates a ridge because small moves from this configuration typically result in fewer non-attacking pairs, requiring several coordinated moves to escape this pattern, which hill climbing does not facilitate well.

   Problem: Hill Climbing may fail to navigate such transitions because it does not allow for a temporary increase in the objective function (number of conflicts in this case). Ridges in the landscape can make it very challenging for the algorithm to find a path to the optimal solution since each step must immediately provide an improvement.

4.3.4 Remedies to problems of Hill-Climbing Search Algorithm:

Hill climbing has several variations that address its basic version's limitations, such as getting stuck at local maxima, navigating ridges ineffectively, or wandering on plateaus.

4.3.5 Variants of Hill Climbing:

- **Stochastic Hill Climbing:**

  How it works: This method randomly chooses among uphill moves rather than always selecting the steepest ascent. The probability of choosing a move can depend on the steepness of the ascent.

  Performance: It typically converges more slowly than the standard hill climbing because it might not take the steepest path. However, it can sometimes find better solutions in complex landscapes where the steepest ascent might lead straight to a local maximum.

Example in 8-Queens: In a landscape where queens are close to forming a solution but are stuck due to subtle needed adjustments, stochastic hill climbing can randomly explore different moves that gradually lead out of a local maximum.

- **First-Choice Hill Climbing:**

  How it works: A variant of stochastic hill climbing, this strategy generates successors randomly and moves to the first one that is better than the current state.

  Advantages: This is particularly effective when a state has a vast number of successors, as it reduces the computational overhead of generating and evaluating all possible moves.

  Example in Knapsack Problem: With thousands of possible item combinations, generating all potential successors to evaluate the best move is impractical. First-choice hill climbing can quickly find a better solution without exhaustive comparisons.

- **Random-Restart Hill Climbing:**

  How it works: This approach involves performing multiple hill climbing searches from different randomly generated initial states. This process repeats until a satisfactory solution is found.

  Completeness: It is "complete with probability 1" because it is guaranteed to eventually find a goal state, assuming there is a non-zero probability of any single run succeeding. If each hill-climbing search has a probability of success, p then the expected number of restarts required is $\frac{1}{p}$.

  Example in Knapsack Problem: If the algorithm gets stuck in a sub-optimal configuration due to local maxima, restarting with a new random set of items can lead to discovering better solutions.

  Example in 8-Queens: Due to the complex landscape of the 8-Queens problem, random restarts can help escape from sub-optimal arrangements by exploring new configurations that might be closer to the global maximum (i.e., a solution with zero attacking pairs).

4.4 Introduction to Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. It is particularly useful for solving large optimization problems where other methods might be too slow or get stuck in local optima.

The technique is inspired by the physical process of annealing in metallurgy, where metals are heated to a high temperature and then cooled according to a controlled schedule to achieve a more stable crystal structure.

The method was developed by S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi in 1983, based on principles of statistical mechanics.

Simulated Annealing is an optimization technique that simulates the heating and gradual cooling of materials to minimize defects and achieve a stable state with minimum energy.

It starts with a randomly generated initial solution and a high initial temperature, which allows the **acceptance of suboptimal solutions** to explore the solution space widely.

The algorithm iterates by generating small modifications to the current solution, evaluating the cost changes, and **probabilistically deciding** whether to accept the new solution based on the current temperature.

The temperature is gradually reduced according to a predetermined cooling schedule, which **decreases the likelihood of accepting worse solutions** and helps fine-tune towards an optimal solution.

The process concludes once a stopping criterion, like a specified number of iterations, a minimal temperature, or a quality threshold of the solution, is met.

Algorithm 4 Algorithm Simulated Annealing

```
1: Input: initial_solution, initial_temperature, cooling_rate,
   stopping_temperature
2: Output: best_solution_found
   Procedure:
3: current_solution ← initial_solution
4: current_temperature ← initial_temperature
5: best_solution ← current_solution
6: while: current_temperature > stopping_temperature
7:   new_solution ← generate_neighbor(current_solution)
8:   cost_difference ← cost(new_solution) - cost(current_solution)
9:   If cost_difference < 0 or exp(-cost_difference / current_temperature) >
   random(0, 1) current_solution ← new_solution
10:      current_solution ← new_solution
11:  If cost(new_solution) < cost(best_solution)
12:      best_solution ← new_solution
13:  current_temperature ← current_temperature × cooling_rate
14: return best_solution
```

Key Parameters:

- **Initial Temperature:** High enough to allow exploration.
- **Cooling Rate:** Determines how quickly the temperature decreases.
- **Stopping Temperature:** Low enough to stop the process once the system is presumed to have stabilized.

4.5 How does simulated annealing navigate the solution space:

4.5.1 Probability of Accepting a New State

The key mathematical concept in simulated annealing is the probability of accepting a new state S' from a current state S . This probability is determined using the Metropolis-Hastings algorithm, which is defined as follows:

$$P(\text{accept } S') = \min\left(1, e^{-\frac{\Delta E}{T}}\right)$$

where:

- $\Delta E = E(S') - E(S)$ is the change in the objective function (cost or energy) from the current state S to the new state S'.
- T is the current temperature.
- e is the base of the natural logarithm.

  The equation $e^{-\frac{\Delta E}{T}}$ is crucial as it controls the acceptance of new solutions:
- If $\Delta E < 0$ (meaning S' is a better solution than S), then $e^{-\frac{\Delta E}{T}} > 1$, and the new solution is always accepted ($\min(1, \text{value}) = 1$).
- If $\Delta E > 0$ (meaning S' is worse), the new solution is accepted with a probability less than 1. This probability decreases as ΔE increases or as T decreases.

4.5.2 Cooling Schedule

The cooling schedule is a rule or function that determines how the temperature T decreases over time. It is typically a function of the iteration number k. A common choice is the exponential decay given by:

$$T(k) = T_0 \cdot \alpha^k$$

where:
- T_0 is the initial temperature.
- α is a constant such that $0 < \alpha < 1$, often close to 1.
- k is the iteration index.

The choice of α and T_0 influences the convergence of the algorithm. A slower cooling (higher α) allows more thorough exploration of the solution space but takes longer to converge.

4.5.3 Random Selection of Neighbors

The random selection of a neighbor S' is typically governed by a neighborhood function, which defines possible transitions from any given state S. The randomness allows the algorithm to explore the solution space non-deterministically, which is essential for escaping local optima.

4.5.4 Mathematical Convergence

Theoretically, given an infinitely slow cooling (i.e., $\alpha \to 1$ and infinitely many iterations), simulated annealing can converge to a global optimum. This stems from the ability to continue exploring new states with a non-zero probability, provided the cooling schedule allows sufficient time at each temperature level for the system to equilibrate.

Simulated annealing integrates concepts from statistical mechanics with optimization through a controlled random walk. This walk is guided by the probabilistic acceptance of new solutions, which balances between exploiting better solutions and exploring the solution space broadly, moderated by a temperature parameter that systematically decreases over time.

4.6 Example Problem:

4.6.1 Traveling Salesman Problem (TSP)

Let's consider an example of solving a small Traveling Salesman Problem (TSP) using simulated annealing. The TSP is a classic optimization problem where the goal is to find the shortest possible route that visits a set of cities and returns to the origin city, visiting each city exactly once.

Problem Setup

Suppose we have a set of five cities, and the distances between each pair of cities are given by the following symmetric distance matrix:

	A	B	C	D	E
A	0	12	10	19	8
B	12	0	3	5	6
C	10	3	0	6	7
D	19	5	6	0	4
E	8	6	7	4	0

Objective

Find the shortest path that visits each city exactly once and returns to the starting city.

Simulated Annealing Steps

1. **Initialization:** Randomly generate an initial route, say, $A \to B \to C \to D \to E \to A$.

2. **Heating:** Set an initial high temperature to allow significant exploration. For instance, start with a temperature of 100.

3. **Iteration: - Generate a Neighbor:** Create a new route by making a small change to the current route, such as swapping two cities. For instance, swap cities B and D to form a new route $A \rightarrow D \rightarrow C \rightarrow B \rightarrow E \rightarrow A$.

   - **Calculate the Change in Cost:** Determine the total distance of the new route and compare it with the current route.

   - **Acceptance Decision:** Use the Metropolis criterion to decide probabilistically whether to accept the new route based on the change in cost and the current temperature.

4. **Cooling:** Reduce the temperature based on a cooling schedule, e.g., multiply the temperature by 0.95 after each iteration.

5. **Termination:** Repeat the iteration process until the temperature is low enough or a fixed number of iterations is reached. Assume the stopping condition is when the temperature drops below 1 or after 1000 iterations.

Example Calculation

Assuming the first iteration starts with the initial route and the randomly generated neighbor as described:

- Current Route: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$ (Distance $= 12 + 3 + 6 + 4 + 8 = 33$).

- New Route: $A \rightarrow D \rightarrow C \rightarrow B \rightarrow E \rightarrow A$ (Distance $= 19 + 6 + 3 + 6 + 8 = 42$).

The change in cost ΔE is 42 - 33 = 9. The probability of accepting this worse solution at temperature 100 is $e^{-9/100} \approx 0.91$. A random number is generated between 0 and 1; if it is less than 0.91, the new route is accepted, otherwise, the algorithm retains the current route.

This process is repeated, with the temperature decreasing each time, until the termination conditions are met. The result should be a route that approaches the shortest possible loop connecting all five cities.

4.7 Genetic algorithm

Genetic algorithms (GAs) are a class of optimization algorithms inspired by the principles of natural selection and genetics. These algorithms are used to solve search and optimization problems and are particularly effective for complex issues that are difficult to solve using traditional methods.

Genetic algorithms mimic the process of natural evolution, embodying the survival of the fittest among possible solutions. The core idea is derived from the biological mechanisms of reproduction, mutation, recombination, and selection. These biological concepts are translated into computational steps that help in finding optimal or near-optimal solutions to problems across a wide range of disciplines including engineering, economics, and artificial intelligence.

4.7.1 Algorithm

Algorithm 5 Genetic Algorithm

```
 1: Input: Population size, fitness function, mutation rate, crossover rate, maximum generations
 2: Output: Best solution found
 3: begin
 4:    Initialize population with random candidates
 5:    Evaluate the fitness of each candidate
 6:    while termination condition not met do
 7:       Select parents from the current population
 8:       Perform crossover on parents to create new offspring
 9:       Perform mutation on offspring
10:       Evaluate the fitness of new offspring
11:       Select individuals for the next generation
12:       best solution in new generation > best solution so far
13:          Update best solution found
14:    end while
15:    return best solution found
16: end
```

4.7.2 Explanation of the Pseudocode

Initialize Population:

A genetic algorithm begins with a population of randomly generated individuals. Each individual, or chromosome, represents a possible solution to the problem. Depending on the problem the chromosomes are encoded.

4.7.3 Evaluate Fitness:

Each solution in the population is assessed using the fitness function to determine how well it solves the problem. Depending on the problem the fitness function will be measured.

Selection:

This step involves choosing the fitter individuals to reproduce. Selection can be done in various ways, such as Truncation Selection, tournament selection, roulette wheel selection, or rank selection. In this course we are using truncation selection, where we select the fittest 3/4th of the population.

Truncation Selection

Description: Only the top-performing fraction of the population is selected to reproduce.

Procedure: Rank individuals by fitness, then select the top $x\%$ to become parents of the next generation.

Pros and Cons: Very straightforward and ensures high-quality genetic material is passed on, but can quickly reduce genetic diversity.

Crossover (Recombination):

Pairs of individuals are crossed over at a randomly chosen point to produce offspring. The crossover rate determines how often crossover will occur. Two common types of crossover techniques are single-point crossover and two-point (or double-point) crossover.

- **Single-Point Crossover:**

  In single-point crossover, a single crossover point is randomly selected on the parent chromosomes. The genetic material (bits, characters, numbers, depending on the encoding of the solution) beyond that point in the chromosome is swapped between the two parents. This results in two new offspring, each carrying some genetic material from both parents.

  Procedure:

  Select a random point on the chromosome. The segments of the chromosomes after this point are swapped between the two parents.

  Example:

  Suppose we have two binary strings:

  Parent 1: 110011

  Parent 2: 101010

  Assuming the crossover point is after the third bit, the offspring would be:

  Offspring 1: 110010 (first three bits from Parent 1, last three bits from Parent 2)

  Offspring 2: 101011 (first three bits from Parent 2, last three bits from Parent 1)

- **Two-Point Crossover:**

  Two-point crossover involves two points on the parent chromosomes, and the genetic material located between these two points is swapped between the parents. This can introduce more diversity compared to single-point crossover because it allows the central segment of the chromosome to be exchanged, potentially combining more varied genetic information from both parents.

Procedure:

Select two random points on the chromosome, ensuring that the first point is less than the second point.

Swap the segments between these two points from one parent to the other.

Example:

Continuing with the same parent strings:

Parent 1: 110011

Parent 2: 101010

Let's choose two crossover points, between the second and fifth bits. The offspring produced would be:

Offspring 1: 100010 (first two bits from Parent 1, middle segment from Parent 2, last bit from Parent 1)

Offspring 2: 111011 (first two bits from Parent 2, middle segment from Parent 1, last bit from Parent 2)

4.8 Mutation

With a certain probability (mutation rate), mutations are introduced to the offspring to maintain genetic diversity within the population.

The purpose of mutation is to maintain and introduce genetic diversity into the population of candidate solutions, helping to prevent the algorithm from becoming too homogeneous and getting stuck in local optima.

4.8.1 Purpose of Mutation

1. **Introduce Variation:** Mutation introduces new genetic variations into the population by altering one or more components of genetic sequences, ensuring a diversity of genes.
2. **Prevent Local Optima:** By altering the genetic makeup of individuals, mutation prevents the population from converging too early on a suboptimal solution.
3. **Explore New Areas:** It enables the algorithm to explore new areas of the solution space that may not be reachable through crossover alone.

4.8.2 How Mutation Works

Mutation operates by making small random changes to the genes of individuals in the population. In the context of genetic algorithms, an individual's genome might be represented as a string of bits, characters, numbers, or other data structures, depending on the problem being solved.

4.8.3 Common Types of Mutation

1. **Bit Flip Mutation (for binary encoding):**
   - **Procedure:** Each bit in a binary string has a small probability of being flipped (0 changes to 1, and vice versa).
   - **Example:** A binary string '110010' might mutate to '110011' if the last bit is flipped.
2. **Random Resetting (for integer or real values):**
   - **Procedure:** A selected gene is reset to a new value within its range.
   - **Example:** In a string of integers $[4, 12, 7, 1]$, the third element 7 might mutate to 9.
3. **Swap Mutation:**
   - **Procedure:** Two genes are selected and their positions are swapped. This is often used in permutation-based encodings.
   - **Example:** In an array $[3, 7, 5, 8]$, swapping the second and fourth elements results in $[3, 8, 5, 7]$.
4. **Scramble Mutation:**
   - **Procedure:** A subset of genes is chosen and their values are scrambled or shuffled randomly.
   - **Example:** In an array $[3, 7, 5, 8, 2]$, scrambling the middle three elements might result in $[3, 5, 8, 7, 2]$.
5. **Uniform Mutation (for real-valued encoding):**
   - **Procedure:** Each gene has a fixed probability of being replaced with a uniformly chosen value within a predefined range.
   - **Example:** In an array of real numbers $[0.5, 1.3, 0.9]$, the second element 1.3 might mutate to 1.1.

4.8.4 Mutation Rate

The mutation rate is a critical parameter in genetic algorithms. It defines the probability with which a mutation will occur in an individual gene. A higher mutation rate increases diversity but may also disrupt highly fit solutions, whereas a lower mutation rate might not provide enough diversity, leading to premature convergence. The optimal mutation rate often depends on the specific problem and the characteristics of the population.

Evaluate New Offspring: The fitness of each new offspring is calculated.

Generation Update: The algorithm decides which individuals to keep for the next generation. This can be a mix of old individuals (elitism) and new offspring.

Termination Condition: The algorithm repeats until a maximum number of generations is reached, or if another stopping criterion is satisfied (like a satisfactory fitness level).

Return Best Solution: The best solution found during the evolution is returned.

Note: For exam problems if you are asked to simulate, unless otherwise instructed, start with 4 chromosomes in the population, select best 3 at each step, crossover between the best and the other two selected

4.8.5 Diversity in Genetic Algorithm

It is often the case that the population is diverse early on in the process, so crossover frequently takes large steps in the state space early in the search process (as in simulated annealing). After many generations of selection towards higher fitness, the population becomes less diverse, and smaller steps are typical.

It is important for the initial population to be diversed. Otherwise, similar type of chromosomes will crossover to and produce offsprings with little change in their fitness. This will lead to quick convergence of the algorithm and the chances of finding a solution with maximum fitness will be very low.

4.8.6 Advantages and Applications

Genetic algorithms are particularly useful when:
- The search space is large, complex, or poorly understood.
- Traditional optimization and search techniques fail to find acceptable solutions.
- The problem is dynamic and changes over time, requiring adaptive solutions.

4.9 Examples

4.9.1 8-Queen Problem

The 8-queens problem involves placing eight queens on an 8x8 chessboard so that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal.

Chromosome Representation:

Each chromosome can be represented as a string or array of 8 integers, each between 1 and 8, representing the row position of the queen in each column represented by the index of the string or array.

Fitness Function:

Fitness is calculated based on the number of non-attacking pairs of queens. The maximum score for 8 queens is 28 (i.e., no two queens attack each other).

Calculating the number of attacking pairs: The 8-queen problem has the following conditions.

- No pairs share the same column.
- No pairs share the same row.
- No pairs share the same diagonal.

Column-wise conflict Now, due to the representation of the configuration where we have ensured that no pairs can share the same column as the indices of the array are unique.

Row-wise conflict Now, the values in each index represent the row where the queen is placed. Now, if we find the same value in multiple indices that means there are queens sharing that row.

Take for example, the configuration: [8, 7, 7, 3, 7, 1, 4, 4]. Here, the value 7 is repeated thrice and the value 4 is repeated twice. This means, there are 3 queens on the 7th row and 2 queens in the 4th row.

Counting the conflicts in 7th row: 3 queens will form ${}^{3}C_{2} = 3(3-1)/2 = 3$ pairs.

Counting the conflicts in 4th row: 2 queens will form ${}^{2}C2 = 2(2-1)/2 = 1$ pairs.

Total 4 row-wise conflicting pairs.

Diagonal Conflicts Two types of diagonals are formed in a square board we call them Major ("/" shaped) diagonal, and Minor ("\" shaped) diagonal. **Major diagonal conflict**

If two queens, $Q1$ and $Q2$ share the same major diagonal conflict abs($Q1$[column]$-Q1$[row]) = abs($Q2$[column] $-$ $Q2$[row]).

Going back to the configuration: [8, 7, 7, 3, 7, 1, 4, 4], the queen in 7th row, 2nd column is in the same diagonal as the queen in 1st row, 6th column and the queen in 7th row, 3rd column is in conflict with the queen in 4th row, 8th column. Or in other words 2 queens share the 5th (|7-2| = |1-6|) Major diagonal making ($^2C2 = 2(2-1)/2 = 1$) conflicting pair and 2 queens share the 4th (|7-3| = |4-8|) Major diagonal making another (1) conflicting pair.

So, there are two attacking pairs along the major diagonal.

Minor diagonal conflict If two queens, $Q1$ and $Q2$ share the same major diagonal conflict $Q1$[column] + $Q1$[row] = $Q2$[column] + $Q2$[row].

Going back to the configuration: [8, 7, 7, 3, 7, 1, 4, 4], the queen in 8th row, 1st column is in the same diagonal as the queen in 7th row, 2nd column, the queen in 3rd row, 4th column is in conflict with the queen in 1st row, 6th column and the queen in 7th row, 5th column is in conflict with the queen in 4th row, 8th column. Or in other words 2 queens share the 9th (|8+1|=|7+2|) minor diagonal making 1 ($^2C2 = 2(2-1)/2$) conflicting pair, 2 queens share the 7th (|3+4| = |1+6|) minor diagonal making another ($^2C2 = 2(2-1)/2$) conflicting pair and 2 queens share the 12th (|7+5| = |4+8|) minor diagonal making another ($^2C2 = 2(2-1)/2$) conflicting pair.

So, there are three attacking pairs along the minor diagonal.

Summing up the number of attacking pairs:

- 4 row-wise attacking pairs.
- 2 attacking pairs on the major diagonal.
- 3 attacking pairs on the minor diagonal.
- In total ($4 + 2 + 3 = 9$) attacking pairs.

Calculating the number of non-attacking pairs: The maximum number of pairs formed from n number of queens is $^nC2 = \frac{n(n-1)}{2}$. The maximum number of pairs that can be formed by 8 queens is $\frac{8(8-1)}{2} = 28$. In the ideal configuration with zero attacking pairs, we can have all 28 pairs in non-attacking mode. Thus, in any configuration,
No. of non-attacking pairs = No. of max possible non-attacking pairs - No. of attacking pairs.

For the example of [8, 7, 7, 3, 7, 1, 4, 4] configuration of the 8-queen problem, the max possible non-attacking pairs is 28 and the total no. of attack we calculated is 9. So the number of non-attacking pairs in this configuration is $28 - 9 = 19$.

So, the Fitness value of this configuration is 19.

A better way to calculate fitness is to converting the value within a range of (0 to 1) or in a $x\%$.

$$\text{fitness} = \frac{\text{no. of non-attacking pairs}}{28}$$

However, for simplicity, we are going to take the no. of non-attacking pairs as our fitness value.

Iteration 1

Initialization of Population:

We randomly generate a small population of 4 individuals for simplicity. To keep track of the individual with the best fitness so far we initially set best-so-far = null and best-fitness-so-far = 0 and update when we find better individuals.

```
best-so-far = [];
best-fitness-so-far = 0;
max-fitness = 28;
```

Step 1: Population

Initial Population

Chromosome 1:	4 2 7 3 6 8 5 1
Chromosome 2:	2 7 4 1 8 5 3 6
Chromosome 3:	5 3 1 7 2 8 6 4
Chromosome 4:	7 1 4 2 8 5 3 6

Step 2: Calculate Fitness

	Chromosomes	**Fitness**
Chromosome 1:	4 2 7 3 6 8 5 1	26
Chromosome 2:	2 7 4 1 8 5 3 6	24
Chromosome 3:	5 3 1 7 2 8 6 4	23
Chromosome 4:	7 1 4 2 8 5 3 6	24

As we have found a chromosome with better fitness value we have saved before, we update,

```
best-so-far = [4 2 7 3 6 8 5 1];
best-fitness-so-far = 26;
```

However, the fitness is not the highest possible value of 28, so we continue to the next step.

Step 3: Selection

Select the top 3/4th of chromosomes based on their fitness. This results in selecting Chromosome 1, 2 and 3.

	Chromosomes	**Fitness**	**Selection**
Chromosome 1:	4 2 7 3 6 8 5 1	26	selected
Chromosome 2:	2 7 4 1 8 5 3 6	24	selected
Chromosome 3:	5 3 1 7 2 8 6 4	23	not selected
Chromosome 4:	7 1 4 2 8 5 3 6	24	selected

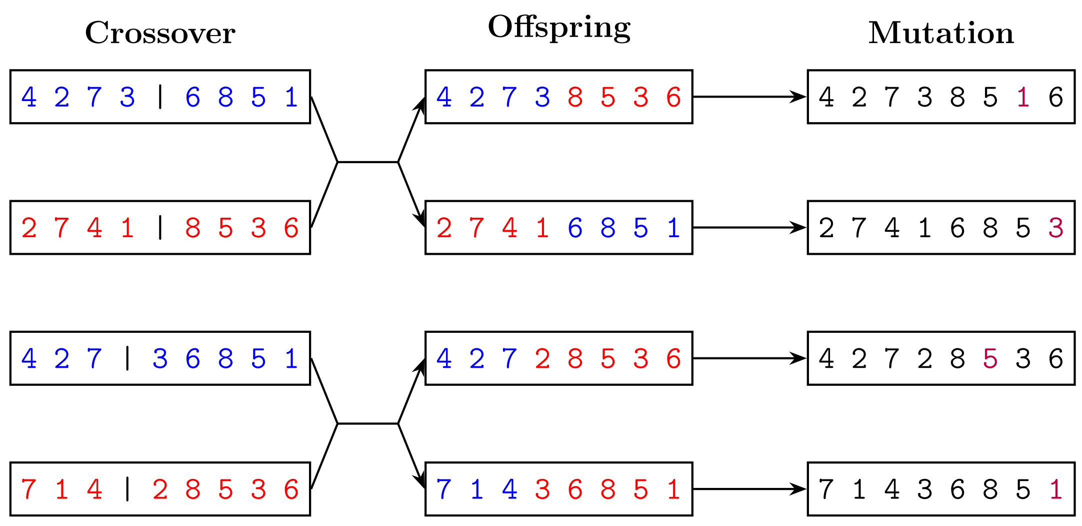
	Selected Chromosomes	**Fitness**
Chromosome 1:	4 2 7 3 6 8 5 1	26
Chromosome 2:	2 7 4 1 8 5 3 6	24
Chromosome 3:	7 1 4 2 8 5 3 6	24

Step 4: Crossover (Single Point)

We pair the best chromosome [4, 2, 7, 3, 8, 5, 1, 6] with the other two selected chromosomes to create new offspring.

Step 5: Mutation

For each offspring, We randomly choose an index and change the value of the row to a number chosen randomly between 1 to 8. As the number is chosen randomly it may happen that the value remains same as we see for offspring 3 and 4.



Step 6: Create new Population replacing the old population

After the mutation is complete we replace the previous population with the set of new chromosomes and repeat from step 2 with the new population until the configuration with highest fitness (non-attacking pair) is found.

New Population

Chromosome 1:	4 2 7 3 8 5 1 6
Chromosome 2:	2 7 4 1 6 8 5 3
Chromosome 3:	4 2 7 1 8 5 3 6
Chromosome 4:	2 7 4 3 6 8 5 1

4.9.2 Traveling Salesman Problem (TSP)

Problem Setup:

We have five cities: A, B, C, D and E. The distance matrix given:

	A	B	C	D	E
A	0	2	9	10	7
B	2	0	6	5	8
C	9	6	0	12	10
D	10	5	12	0	15
E	7	8	10	15	0

Genetic Algorithm Setup

Chromosome Representation:

A permutation of the cities $[A, B, C, D, E]$.

Fitness Function:

The fitness of each chromosome is calculated as the inverse of the total route distance. The shorter the route, the higher the fitness.

$$\boxed{\text{Fitness} = \frac{1}{\text{Route Distance}}}$$

Example Calculation: For the chromosome $[A, B, C, D, E]$:
Distance $(A--B) : 2$
Distance $(B--C) : 6$
Distance $(C--D) : 12$
Distance $(D--E) : 15$
Distance ($E--A$ to complete the loop): 7
Total Distance: $2 + 6 + 12 + 15 + 7 = 42$

$$\text{Fitness} = \frac{1}{\text{Total Distance}} = \frac{1}{42}$$

Iteration 1

Initialization of Population:

We start with a population of four randomly generated routes as combination of the cities. We keep track of the best found route and its fitness.

```
best-so-far = [];
best-fitness-so-far = 0;
```

Initial Population

Chromosome 1: $\boxed{\text{A B C D E}}$

Chromosome 2: $\boxed{\text{B E D C A}}$

Chromosome 3: $\boxed{\text{E D B A C}}$

Chromosome 4: $\boxed{\text{B D C E A}}$

Step 2: Fitness Calculation

Lower distance means higher fitness in this problem. The fitness for each chromosome in the population is calculated.

	Chromosomes	**Fitness**	**Selection**
Chromosome 1:	$\boxed{\text{A B C D E}}$	$\frac{1}{42}$	selected
Chromosome 2:	$\boxed{\text{B E D C A}}$	$\frac{1}{45}$	selected
Chromosome 3:	$\boxed{\text{E D B A C}}$	$\frac{1}{49}$	not selected
Chromosome 4:	$\boxed{\text{B D C E A}}$	$\frac{1}{39}$	selected

Step 3: Selection

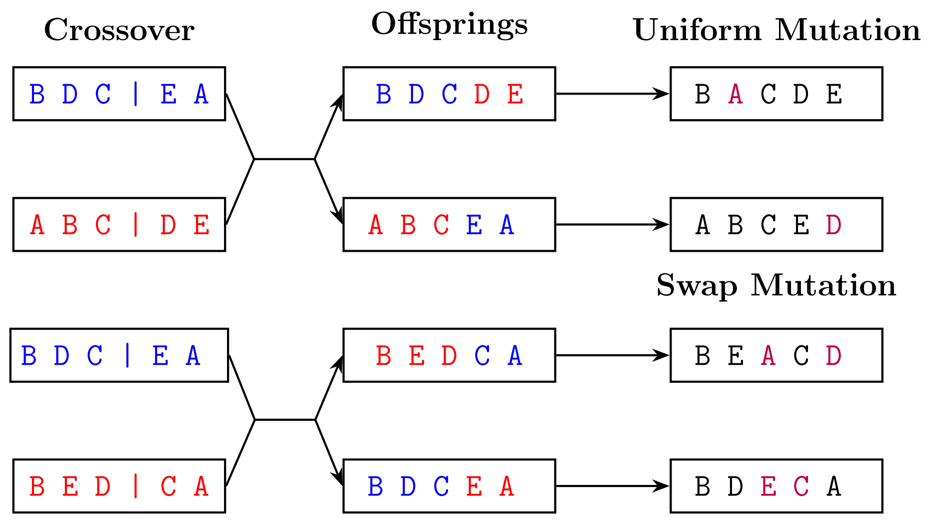
Select top 3/4th based on fitness: Chromosome 4, Chromosome 1 and Chromosome 2. Chromosome 4 has a higher fitness value than the `best-fitness-so-far` we have. So, we make an update.

```
best-so-far = [B D C E A];
best-fitness-so-far = 1/39;
```

	Selected Chromosomes	**Fitness**
Chromosome 1:	$\boxed{\text{A B C D E}}$	$\frac{1}{42}$
Chromosome 2:	$\boxed{\text{B E D C A}}$	$\frac{1}{45}$
Chromosome 3:	$\boxed{\text{B D C E A}}$	$\frac{1}{39}$

Step 4: Crossover (One-Point)

We make pairs between chromosome with the highest fitness, [B D C E A] with the other two chromosomes that are selected. The crossover point for the pairs are randomly generated. The crossover point in this case for both the pairs were randomly chosen to be 3.



Step 5: Mutation

Offspring 1 is mutated by randomly changing the second city from D to A. Offspring 2 is mutated by changing the fifth city from A to D.
On the other hand, Offspring 3 and Offspring 4 are mutated by swapping cities. In Offspring 3, the third and fifth cities are swapped. In Offspring 4, the third and fourth cities are swapped.
Note: It is better to use one specific type of mutation in your simulation. Always mention what type of mutation you are using.

Step 6: New Population

Replace previous population with the newly generated chromosomes.

New Population

Chromosome 1:	B A C D E
Chromosome 2:	A B C E D
Chromosome 3:	B E A C D
Chromosome 4:	B D E C A

Step 7: Repeat

This process is repeated from step 2 over several generations to find the chromosome with the highest fitness, representing the shortest possible route that visits each city exactly once and returns to the starting point.

4.9.3 0/1 Knapsack Problem

Problem Setup

Objective

Maximize the value of items packed in a knapsack without exceeding the weight limit.

Constraints:

Maximum weight the knapsack can hold: 15 kg Items:

Item	Weight (Kg)	Value ($)
1	6	30
2	3	14
3	4	16
4	2	9

Genetic Algorithm Setup

Chromosome Representation:

Each chromosome is a string of four bits, where each bit represents whether an item is included (1) or not (0). For example, ‘1010’ means Items 1 and 3 are included, while Items 2 and 4 are not.

Step 1: Initialization

Generate four random chromosomes (combination of the items) making the initial population. We keep track of the best found route and its fitness.

```
best-so-far = [];
best-fitness-so-far = 0;
```

Initial Population

Chromosome 1: $\boxed{1\ 1\ 0\ 1}$

Chromosome 2: $\boxed{1\ 0\ 1\ 0}$

Chromosome 3: $\boxed{0\ 1\ 1\ 0}$

Chromosome 4: $\boxed{1\ 0\ 0\ 1}$

Step 2: Fitness Evaluation

Calculate the total value of each chromosome, ensuring the weight does not exceed the capacity.

	Chromosomes	Fitness
Chromosome 1:	1 1 0 1	53
Chromosome 2:	1 0 1 0	46
Chromosome 3:	0 1 1 0	30
Chromosome 4:	1 0 0 1	39

Step 3: Selection

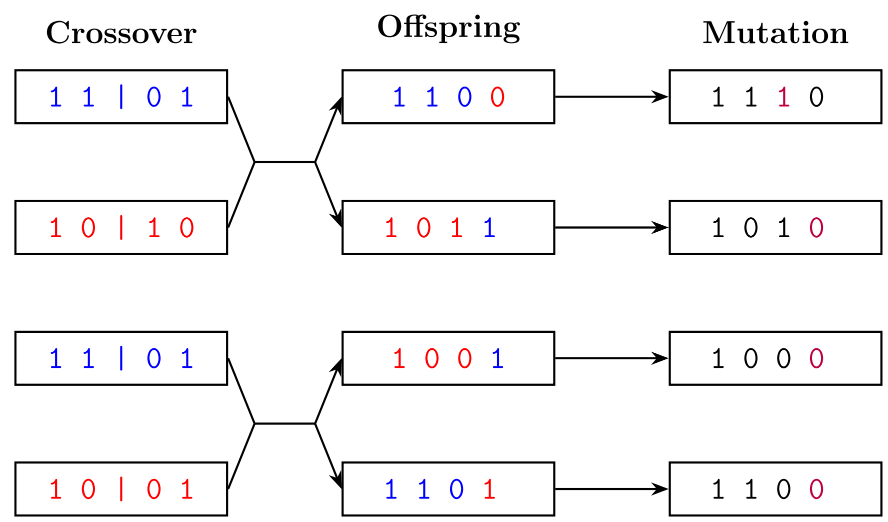
Select the top 3/4th of chromosomes based on their value.

```
best-so-far = [1101];
best-fitness-so-far = 53;
```

	Chromosomes	Fitness
Chromosome 1:	1 1 0 1	53
Chromosome 2:	1 0 1 0	46
Chromosome 4:	1 0 0 1	39

Step 4: Crossover

Perform crossover between the chromosome with the highest fitness value with the two other chromosomes from the selected chromosomes at the third bit.



Step 5: Mutation

Apply a mutation by flipping a random bit in each offspring.

Step 6: New Population

Replace the previous population with the set of new chromosomes.

New Population

Chromosome 1:	1 1 1 0
Chromosome 2:	1 0 1 0
Chromosome 3:	1 0 0 0
Chromosome 4:	1 1 0 0

Step 7: Repeat

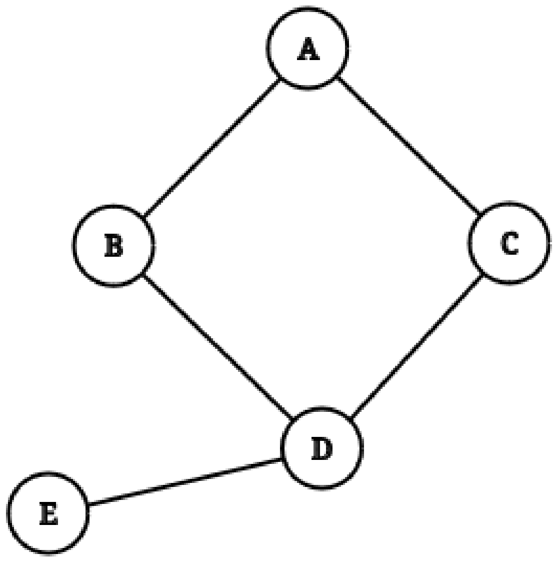
This process is repeated over several generations to find the chromosome with the highest fitness, representing the maximum value found.

4.9.4 Graph Coloring Problem:

The graph coloring problem involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color, and the goal is to minimize the number of colors used.

Graph Description

Consider a simple graph with 5 vertices (A, B, C, D, E) and the following edges: AB, AC, BD, CD, DE.



Genetic Algorithm Setup for Graph Coloring

Chromosome Representation

Each chromosome is an array where each position represents a vertex and the value at that position represents the color of that vertex. For simplicity, we'll use numerical values to represent different colors.

Initial Population

We randomly generate a small population of 4 solutions. We keep track of the best found route and its fitness.

```
best-so-far = [];
best-fitness-so-far = 0;
```

Initial Population

Chromosome 1: $\boxed{1\ 2\ 3\ 1\ 2}$

Chromosome 2: $\boxed{2\ 3\ 1\ 2\ 3}$

Chromosome 3: $\boxed{1\ 2\ 1\ 3\ 2}$

Chromosome 4: $\boxed{3\ 1\ 2\ 3\ 3}$

Fitness Function

Fitness is determined by the number of properly colored edges (i.e., edges connecting vertices of different colors). The maximum fitness for this graph is 5 (one for each edge).

Calculate the fitness based on the coloring rules.
Chromosome 1: Fitness = 5 (all edges correctly colored).
Chromosome 2: Fitness = 5.
Chromosome 3: Fitness = 4 (CD edge is incorrectly colored).
Chromosome 4: Fitness = 4 (DE edge is incorrectly colored).

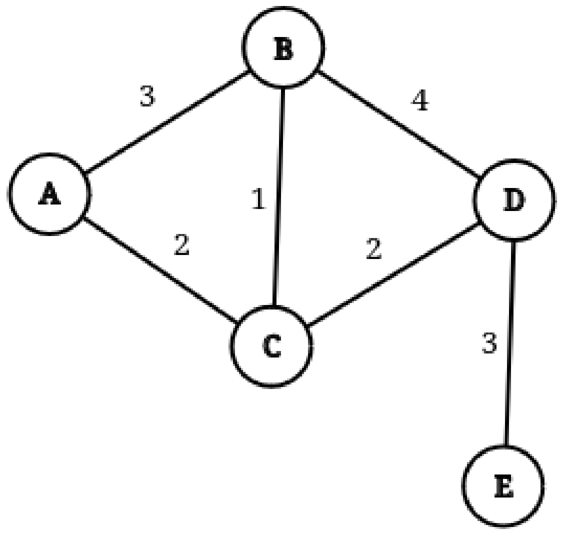
Carry on selection, crossover, mutation and population replacement as before.

4.9.5 Max-cut Problem

The Max-Cut problem is a classic problem in computer science and optimization in which the goal is to divide the vertices of a graph into two disjoint subsets such that the number of edges between the two subsets is maximized. Here, I will outline how to solve this problem using a genetic algorithm (GA).

Problem Setup

Consider a graph with 5 vertices (A, B, C, D, E) and the following edges with given weights:
AB = 3, AC = 2, BC = 1, BD = 4, CD = 2, DE = 3



The goal is to find a division of these vertices into two sets that maximizes the sum of the weights of the edges that have endpoints in each set.

Genetic Algorithm Setup for Max-Cut

Chromosome Representation:

Each chromosome is a string of bits where each bit represents a vertex. A bit of '0' might represent the vertex being in set X and '1' in set Y.

Fitness Function

Fitness is determined by the sum of the weights of the edges between the two sets. For a chromosome, calculate the sum of weights for edges where one endpoint is '0' and the other is '1'.

For example, consider a configuration 10101 where the Vertices A, C, E in set Y; B, D in set X.

Fitness = Sum of Edges (AB, BC, CD, DE) − Sum of Edges(AC) = $3 + 1 + 2 + 3 - 2 = 7$.

Carry on selection, crossover, mutation, and replacement as before.

4.10 Application of Genetic Algorithm in Machine Learning

4.10.1 Problem Setup: Feature Selection for Predictive Modeling

Suppose you're working with a medical dataset aimed at predicting the likelihood of patients developing a certain disease. The dataset contains hundreds of features, including patient demographics, laboratory results, and clinical parameters. Not all of these features are relevant for the prediction task, and some may introduce noise or redundancy.

Genetic Algorithm Setup for Feature Selection

Chromosome Representation:

Each chromosome in the GA represents a possible solution to the feature selection problem. Specifically, a chromosome can be encoded as a binary string where each bit represents the presence (1) or absence (0) of a corresponding feature in the dataset.

Initial Population

Generate an initial population of chromosomes randomly, where each chromosome has a different combination of features selected (1s) and not selected (0s).

Fitness Function:

The fitness of each chromosome (feature subset) is determined by the performance of a predictive model trained using only the selected features. Common performance metrics include accuracy, area under the ROC curve, or F1-score, depending on the problem specifics. Optionally, the fitness function can also penalize the number of features to maintain model simplicity.

4.10.2 Genetic Algorithm Process for Feature Selection

Step 1: Initialization

- Generate an initial population of feature subsets encoded as binary strings.

Step 2: Evaluation

- For each chromosome, train a model using only the features selected by that chromosome. Evaluate the model's performance on a validation set.

Step 3: Selection

- Select chromosomes for reproduction. Techniques like tournament selection or roulette wheel selection can be used, where chromosomes with higher fitness have a higher probability of being selected.

Step 4: Crossover

- Perform crossover between pairs of selected chromosomes to create offspring. A common method is one-point or two-point crossover, where segments of parent chromosomes are swapped to produce new feature subsets.

Step 5: Mutation

- Apply mutation to the offspring with a small probability. This could involve flipping some bits from 0 to 1 or vice versa, thus adding or removing features from the subset.

Step 6: Replacement

- Form a new generation by replacing some of the less fit chromosomes in the population with the new offspring. This could be a generational replacement or a steady-state replacement (where only the worst are replaced).

Iteration

- Repeat the process for a number of generations or until a stopping criterion is met (such as no improvement in fitness for a certain number of generations).

Example Use Case: Feature Selection in Medical Diagnosis

You have a dataset with 200 features from various medical tests. You apply a genetic algorithm with an initial population of 50 chromosomes, evolving over 100 generations. Each chromosome dictates which features are used to train a logistic regression model to predict disease occurrence.

The process might reveal that only 30 out of the 200 features significantly contribute to the prediction, eliminating redundant and irrelevant features and thus simplifying the model without sacrificing (or possibly even improving) its performance.

4.10.3 Problem Setup: Hyperparameter Optimization for a Neural Network

Suppose we are developing a neural network to classify images into categories (e.g., for a fashion item classification task). The performance of the neural network can depend heavily on the choice of various hyperparameters such as the number of layers, number of neurons in each layer, learning rate, dropout rate, and activation function.

Genetic Algorithm Setup for Hyperparameter Optimization

Chromosome Representation:

Each chromosome represents a set of hyperparameters for the neural network. For instance:

- Number of layers (e.g., 2-5)

- Neurons in each layer (e.g., 64, 128, 256, 512)
- Learning rate (e.g., 0.001, 0.01, 0.1)
- Dropout rate (e.g., 0.1, 0.2, 0.3)
- Activation function (e.g., relu, sigmoid, tanh)

Initial Population:

Generate an initial population of chromosomes, each encoding a different combination of these hyperparameters.

Fitness Function:

The fitness of each chromosome is evaluated based on the validation accuracy of the neural network configured with the hyperparameters encoded by the chromosome. Optionally, the fitness function can also include terms to penalize overfitting or excessively complex models.

Genetic Algorithm Process for Hyperparameter Optimization

Step 1: Initialization

- Generate an initial population of random but valid hyperparameter sets.

Step 2: Evaluation

- For each chromosome, construct a neural network with the specified hyperparameters, train it on the training data, and then evaluate it on a validation set. The validation set accuracy serves as the fitness score.

Step 3: Selection

- Select chromosomes for reproduction based on their fitness. High-fitness chromosomes have a higher probability of being selected. Techniques like tournament selection or rank-based selection are commonly used.

Step 4: Crossover

- Perform crossover operations between selected pairs of chromosomes to create offspring. Crossover can be one-point, two-point, or uniform (where each gene has an independent probability of coming from either parent).

Step 5: Mutation

- Apply mutation to the offspring chromosomes at a low probability. Mutation might involve changing one of the hyperparameters to another value within its range (e.g., changing the learning rate from 0.01 to 0.001).

Step 6: Replacement

- Replace the least fit chromosomes in the population with the new offspring, or use other replacement strategies like elitism where some of the best individuals from the old population are carried over to the new population.

Iteration

- Repeat the evaluation, selection, crossover, and mutation steps for several generations until the performance converges or a maximum number of generations is reached.

Example Use Case: Image Classification

You are using a dataset of fashion items where the task is to classify images into categories like shirts, shoes, pants, etc. You apply a genetic algorithm to optimize the hyperparameters of a convolutional neural network (CNN). After several generations, the GA might converge to an optimal set of hyperparameters that gives the highest accuracy on the validation dataset.

For instance, the best solution found by the GA could be:

- Number of layers: 3
- Neurons per layer: [512, 256, 128]
- Learning rate: 0.01
- Dropout rate: 0.2
- Activation function: relu

4.10.4 Genetic Algorithm in AI Games:

Example Use Case: Developing a Chess AI

Imagine you're developing an AI for a chess game. You start with 50 different strategies encoded as chromosomes. Each strategy is evaluated based on its performance in 100 games against diverse opponents. The strategies are then evolved over 100 generations, with each generation involving selection, crossover, and mutation to develop more refined and successful game strategies.

By the end of these iterations, the genetic algorithm might produce a strategy that effectively balances aggressive and defensive play, adapts to different opponent moves, and optimizes piece positioning throughout the game.

4.10.5 Genetic algorithms in Finance

Example Use Case: Diversified Investment Portfolio

Assume you manage an investment fund that considers a diverse set of assets, including stocks from various sectors, government and corporate bonds, and commodities like gold and oil. The task is to determine how much to invest in each asset class.

Assets: Stocks (technology, healthcare, finance), government bonds, corporate bonds, gold, oil.

Objective: Maximize the Sharpe ratio, considering historical returns and volatility data for each asset class.

Problem Setup:

Portfolio Optimization for Investment Management.

Suppose you are an investment manager looking to create a diversified investment portfolio. You want to determine the optimal allocation of funds across a set of available assets (e.g., stocks, bonds, commodities) to maximize returns while controlling risk, subject to various constraints like budget limits or maximum exposure to certain asset types.

Genetic Algorithm Setup for Portfolio Optimization

Chromosome Representation:

Each chromosome in the GA represents a potential portfolio, where each gene corresponds to the proportion of the total investment allocated to a specific asset.

Initial Population:

Generate an initial population of chromosomes, each encoding a different allocation strategy, ensuring that each portfolio adheres to the budget constraint (i.e., the total allocation sums to 100%).

Fitness Function:

The fitness of each chromosome (portfolio) is typically evaluated based on its expected return and risk (often quantified as variance or standard deviation). A common approach to measure fitness is to use the Sharpe ratio, which is the ratio of the excess expected return of the portfolio over the risk-free rate, divided by the standard deviation of the portfolio returns.

After running the genetic algorithm for several generations, the GA might find an optimal portfolio that, for example, allocates 20% to technology stocks, 15% to healthcare stocks, 10% to finance stocks, 20% to government bonds, 15% to corporate bonds, 10% to gold, and

10% to oil. This portfolio would have the highest Sharpe ratio found within the constraints set by the algorithm.

CHAPTER 5

LECTURE 6: ADVERSARIAL SEARCH/GAMES

Note: This lecture closely follows (sometimes, directly borrowed from) Chapter 5 to 5.2.4 of Russel and Norvig, *Artificial Intelligence: A Modern Approach*.

5.1 Introduction

Now we would like to focus on competitive environments, in which two or more agents have conflicting goals. The agents in these environments work against each other, minimizing the evaluation value of each other. To tackle such environments, we use Adversarial search. We will study Minimax search, using which we can find the optimal move for an agent in the restricted game environment. We will also study alpha-beta pruning where including pruning we can make the search more efficient by ignoring portions that do not help us find the optimal solution.

Instead of handling real life problems which have a lot of uncertainty and are difficult to handle we focus on games. Examples of such games are chess, Go, poker etc. We will use much easier games than these to demonstrate the adversarial nature.

5.1.1 Two player zero-sum games

- **Deterministic:** We can determine the outcome of an action.
- **Two players:** Only two agents working against each other. Example: Player A and Player B.
- **Turn taking:** Agents take turns alternatively. Example: After Agent (Player) A plays a move (chooses their action), Agent (Player) B will get to play their move (choose their action) and so on.
- **Perfect Information:** Fully observable environment, as in there are no parts of the game unknown to the agents.

- **Zero-Sum Games:** The total sum of points of the players in the game is zero. If one player wins (+1 points) that would mean the opponent player loses (-1 points). The total sum of the points in the game will be zero. When there is a draw, the point for both players is zero, meaning the sum will be zero.

We will use the term **move** as a synonym for **action**, and **position** as a synonym for **state**.

5.1.2 Maximizer and Minimizer

Continuing from the concept of a zero-sum game, let us refer to the two players (agents) of the game as MAX and MIN.

MAX aims to maximize its own game points. In a zero-sum game, the total payoff for all players is constant, meaning one player's gain is exactly equal to another's loss. Therefore, MAX seeks to achieve the highest possible outcome for itself, knowing that this will occur at the expense of the other player.

MAX evaluates all possible moves by considering what its opponent might choose, countering their moves. It chooses the option that leads to the highest evaluated payoff under optimal play.

On the other hand, MIN aims to win by minimizing MAX's points. While choosing its move, MIN selects the move that will reduce the good options for MAX, limiting MAX's game points and forcing it towards an outcome where MAX's game points are minimal. This strategy minimizes MIN's loss as well.

For example, in a simple game like tic-tac-toe, MAX aims to align three of its marks in a row, thereby maximizing its chance of winning and ensuring the other player's loss. Conversely, MIN attempts to block MAX's efforts to align three marks, while also seeking opportunities to create a threat that MAX must respond to, thereby steering the game towards a draw or a win for MIN.

MAX and MIN alternatively choose moves, each striving to reach their respective goals. The minimax algorithm provides a way to determine the best possible move for a player (assuming both players play optimally) by minimizing the possible loss for a worst-case (maximum loss) scenario.

These concepts are crucial in designing agents capable of thinking several steps ahead, anticipating and countering the opponent's strategies effectively.

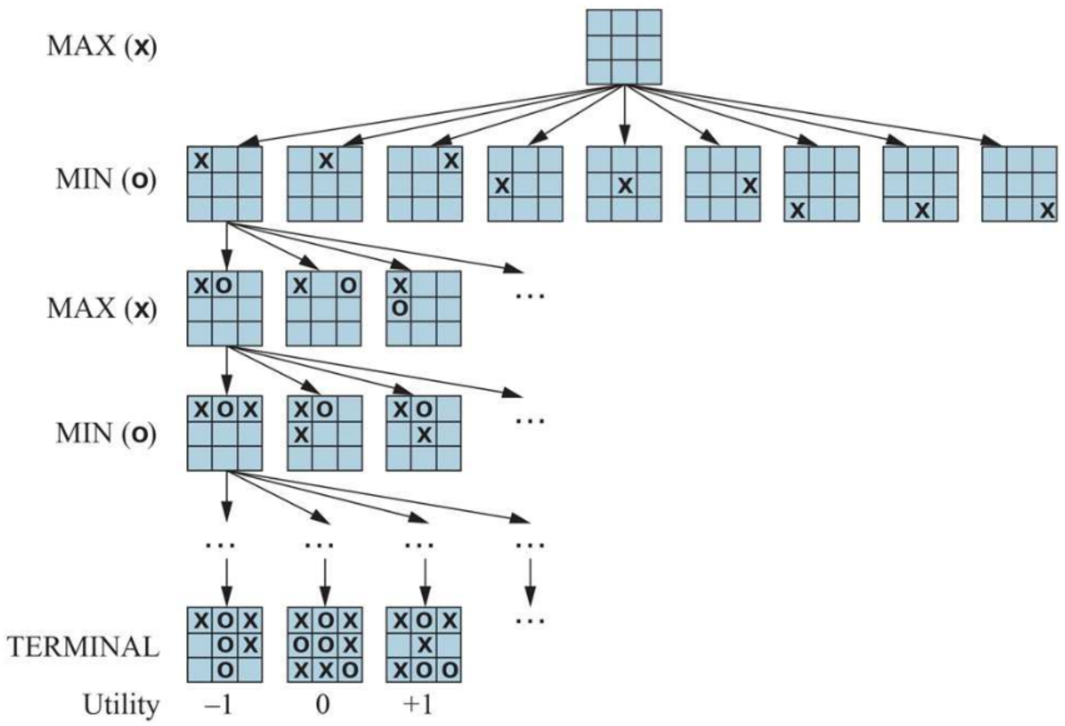
The following elements formally define the game:

- $\mathbf{S_0}$: The initial state, which specifies how the game is set up at the start.
- `TO-MOVE(s)`: The player whose turn it is to move in state `s`.
- `ACTIONS(s)`: The set of legal moves in state `s`.

- `RESULT(s, a)`: The transition model, specifying the state that results from executing action `a` in state `s`.
- `IS-TERMINAL`: A test to determine if state `s` is terminal, returning true if the game has concluded and false otherwise. States where the game has ended are called terminal states. Example, Win, Lose, Draw.
- `UTILITY`: A utility function (also known as an objective function or payoff function), which provides the final numerical value to player `p` when the game concludes in a terminal state `s`. For instance, in chess, the result can be a win, loss, or draw, with values 1, 0, or $\frac{1}{2}$ respectively. In some games, the payoff range can be broader, such as in backgammon, where it ranges from 0 to 192.

The initial state, the action function, and the result function together define the state space graph. In this graph, the vertices represent states and the edges represent moves. A state can be reached by multiple paths. From the search tree, we can construct the game tree that traces every sequence of moves all the way to the terminal state.

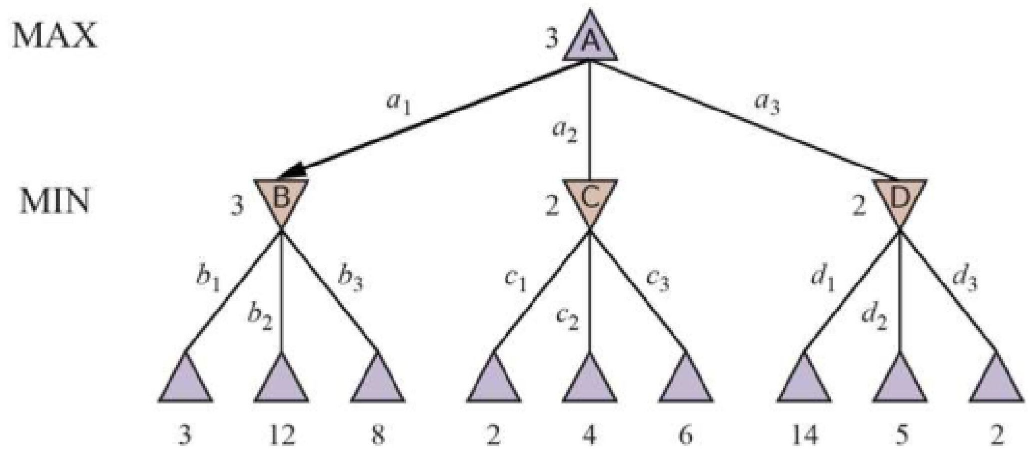
The figure below illustrates part of the game tree for tic-tac-toe (noughts and crosses). Starting from the initial state, MAX has nine possible moves. Players alternate turns, with MAX placing an X and MIN placing an O, until reaching terminal states where either one player has three in a row or the board is completely filled. The number on each leaf node represents the utility value of the terminal state from MAX's perspective; higher values benefit MAX and disadvantage MIN (hence the players' names).



A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

For tic-tac-toe, the game tree is relatively small—fewer than $9! = 362,880$ terminal nodes (with only $5,478$ distinct states). But for chess, there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.

5.2 Making optimal decisions using Minimax Search



A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Consider the trivial game in Figure. The possible moves for MAX at the root node are labeled a_1, a_2, a_3, and so on. The possible replies to for MIN are b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN.

NOTE: In some games, the word "move" means that both players have taken an action; therefore the word *ply* is used to unambiguously mean one move by one player, bringing us one level deeper in the game tree. The utilities of the terminal states in this game range from 2 to 14.

Given a game tree, the optimal strategy can be determined by working out the **minimax value** of each state in the tree, which we write as MINIMAX(S). The minimax value is the utility (for MAX) of being in that state, *assuming that both players play optimally* from there to the end of the game. The minimax value of a terminal state is just its utility. In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX'S turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN). So we have:

$$\textsc{Minimax}(\textsc{s}) = \begin{cases} \textsc{Utility}(s,\ \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

5.3 Minimax algorithm

The minimax algorithm explores the game tree using a depth-first search. If the tree's maximum depth is m and there are b legal moves at each point, the time complexity is $\mathcal{O}(b^m)$

and the space complexity is $\mathcal{O}(bm)$.

The exponential complexity makes minimax impractical for complex games. For instance, chess has a branching factor of about 35, and the average game has a depth of about 80 plies. This means we would need to search almost 10^{123} states, which is not feasible.

Algorithm 6 MINIMAX-SEARCH

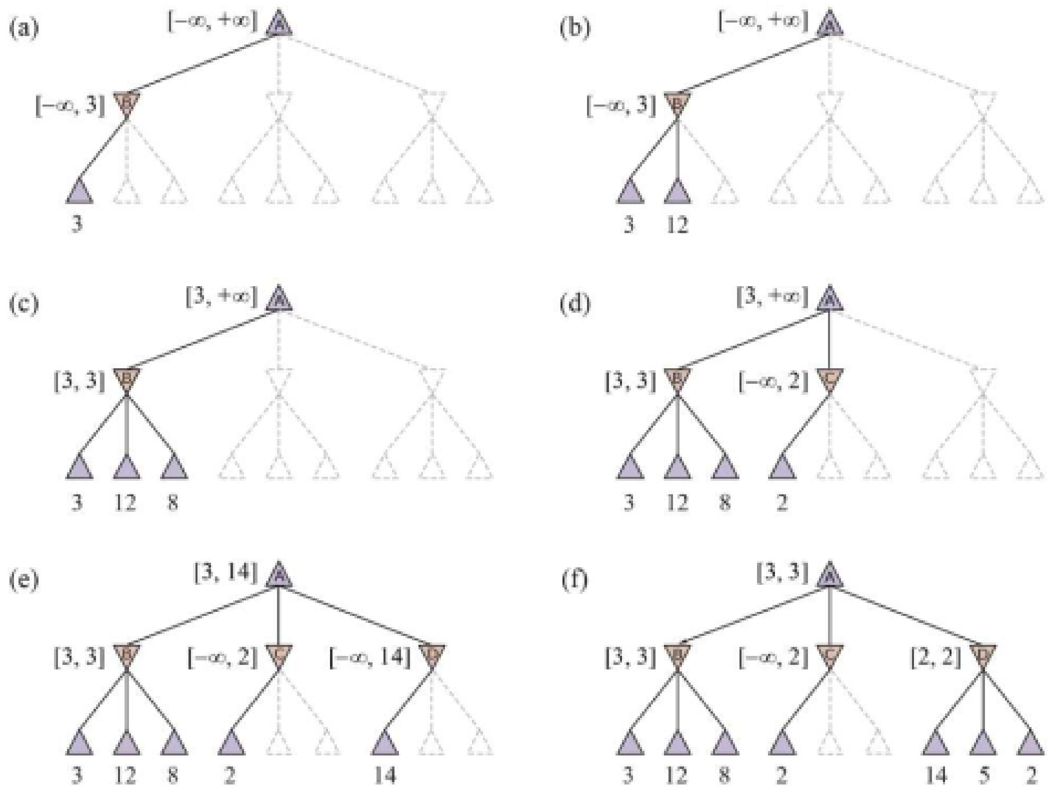
```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility,move) pair
    if game.IS-TERMINAL(state) then return game. UTILITY(state,player),null
    $v \leftarrow -\infty$
    for each a in game. ACTIONS(state) do
        $v2$, $a2$ ← MIN-VALUE(game, game.RESULT(state,a))
        if $v2 > v$ then
            $v$, move ← $v2, a$
    $v$, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state,player),null
    $v \leftarrow +\infty$
    for each a in game.ACTIONS(state) do
        $v2, a2$ ← MAX-VALUE(game, gmae.RESULT(state,a))
        if $v2 < v$ then
            $v$, move ← $v2, a$
    return $v$, move
```

5.4 Alpha—Beta Pruning

Alpha-beta pruning is an optimization technique for the minimax algorithm, which is used in decision-making processes for two-player games like chess or tic-tac-toe. The primary goal of alpha-beta pruning is to reduce the number of nodes evaluated in the game tree, thereby improving the efficiency of the minimax algorithm.

In the minimax algorithm, every possible move and counter-move is explored to determine the optimal strategy. However, this exhaustive search can become computationally expensive, especially for complex games with large search spaces. Alpha-beta pruning addresses this issue by eliminating branches in the game tree that do not influence the final decision.

Let's revisit the two-ply game tree from the previous figure. This time, we will carefully track what we know at each step. The steps are detailed in the figure below. The result is that we can determine the minimax decision without needing to evaluate two of the leaf nodes.



Stages in calculating the optimal decision for the game tree in the previous figure are shown below. At each point, we display the range of possible values for each node.

(a) The first leaf under B has a value of 3. Thus, B, a MIN node, has a value of at most 3.

(b) The second leaf under B has a value of 12. Since MIN would avoid this move, B's value remains at most 3.

(c) The third leaf under B has a value of 8. After evaluating all of B's successor states, B's value is exactly 3. This means the root value is at least 3, because MAX can choose a move worth 3 at the root.

(d) The first leaf under C has a value of 2. Therefore, C, a MIN node, has a value of at most 2. Since B is worth 3, MAX would never choose C. Hence, there is no need to examine the other successors of C. This demonstrates alpha-beta pruning.

(e) The first leaf under D has a value of 14, making D worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to continue exploring D's successors. Now, we also know the root's value is at most 14.

(f) The second successor of D has a value of 5, so we continue exploring. The third successor is worth 2, so D's exact value is 2. MAX's decision at the root is to move to B, yielding a value of 3.

Another way is to see this as a simplification of the MINIMAX formula. Consider the two unevaluated successors of node C in the figure above. Let these successors have values x and y. Then the value of the root node is given by

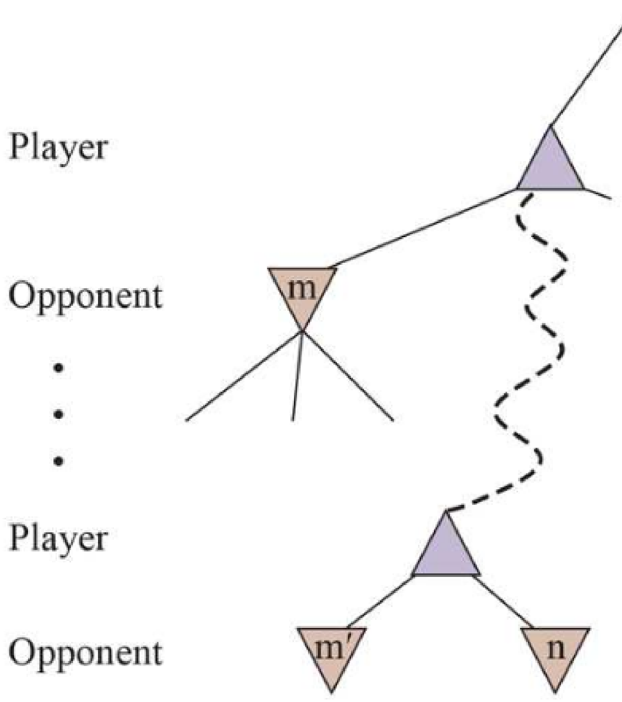
$$\begin{aligned}\textsc{Minimax}(root) &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2))\\ &= \max(3,\min(2,x,y),2)\\ &= \max(3,z,2) \quad (where) z = \min(2,x,y) \leq 2\\ &= 3\end{aligned}$$

In other words, the value of the root and hence the minimax decision are *independent* of the values of the leaves x and y and therefore they can be pruned.

Algorithm 7 Alpha-beta Search Algorithm

```
function ALPHA-BETA-SEARCH(game,state) returns an action
  player ←TO-MOVE(state)
  $value, move$ ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
  return $move$

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a ($utility, move$) pair
  if game.IS-TERMINAL(state)then return game.UTILITY($state, player$), null
  $v \leftarrow -\infty$
  for each a in $game$.ACTIONS(state) do
    $v2, a2$ ← MIN-VALUE($game$, $game$.RESULT($state$, $a$), $\alpha$, $\beta$)
    if $v2 > v$ then
      $v, move \leftarrow v2, a$
      $\alpha \leftarrow$ MAX($\alpha, v$)
    if $v \geq \beta$ then return $v, move$
  return $v, move$

function MIN-VALUE($game$, $state$, $\alpha$, $\beta$)returns a ($utility$, $move$) pair
  if game.IS-TERMINAL(state)then return game.UTILITY($state, player$), null
  $v \leftarrow +\infty$
  for each a in $game$.ACTIONS(state) do
    $v2, a2$ ← MAX-VALUE($game$, $game$.RESULT($state$, $a$), $\alpha$, $\beta$)
    if $v2 < v$ then
      $v, move \leftarrow v2, a$
      $\beta \leftarrow$ MIN($\beta, v$)
    if $v \leq \alpha$ then return $v, move$
  return $v, move$
```

Alpha–beta pruning can be used on trees of any depth, and it often allows for pruning entire subtrees, not just leaves. The basic principle is this: consider a node n in the tree (see Figure below) where Player can choose to move to n. If Player has a better option either at the same level (e.g., m' in the figure below) or higher up in the tree (e.g., m in the figure below), Player will not move to n. Once we learn enough about n by examining some of its successor state to make this decision, we can prune n.



The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

Minimax search uses a depth-first approach, so we only consider the nodes along a single path in the tree at any given time. Alpha-beta pruning uses two additional parameters in MAX-VALUE($state, \alpha, \beta$), which set bounds on the backed-up values along the path.

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: α = " at least."

β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: β = "at most".

As the search progresses, these values are updated to reflect the highest and lowest scores that MAX and MIN can achieve, respectively. When a move is found that makes a branch less favorable than previously examined branches, that branch is pruned, meaning it is not explored further.

This pruning does not affect the final result of the minimax algorithm but significantly reduces the number of nodes evaluated, making it more efficient. In the best case, alpha-beta pruning can reduce the time complexity from $\mathcal{O}(b^m)$ to $\mathcal{O}(b^{\frac{m}{2}}$ where b is the branching factor and m is the maximum depth of the tree.

Alpha-beta pruning enables more effective decision-making in complex games, allowing for deeper searches and better strategic planning within practical time limits.

5.4.1 Worst Case Scenario for Alpha-Beta Pruning

In the worst-case scenario of alpha-beta pruning, the algorithm does not effectively prune any branches, and it ends up exploring the entire game tree, similar to a standard minimax search. This situation can arise when the nodes are ordered in the least optimal way for pruning.

Key Points

- **Branch Ordering:** The worst case occurs when the best moves are always considered last. Alpha-beta pruning relies on evaluating the most promising moves first to maximize pruning. If the least promising moves are evaluated first, fewer branches are pruned.
- **No Pruning:** When the algorithm does not prune any branches, it evaluates every possible move at each depth level, leading to the maximum number of nodes being explored.
- **Time Complexity:** In the worst case, the time complexity of alpha-beta pruning is the same as that of the minimax algorithm without pruning, which is $\mathcal{O}(b^m)$ Here, b is the branching factor (the number of legal moves at each point), and m is the maximum depth of the tree.

Improving alpha-beta pruning

In complex games with large search spaces, to improve the efficiency and effectiveness of the alpha-beta pruning these following techniques can be used.

- **Move-Ordering:** Move-ordering prioritizes evaluating the most promising moves first, increasing the chances of effective pruning in alpha-beta search. This is usually achieved by sorting moves based on heuristic evaluations or previous search results, aiming to maximize the number of branches pruned.
- **Killer Moves:** Killer moves are specific moves that have previously caused significant pruning in similar positions and are tried early in the search to maximize pruning opportunities. These moves are stored and reused, assuming that moves which were effective in the past are likely to be effective again.
- **Iterative Deepening:** Iterative deepening involves repeatedly running depth-limited searches with increasing depth limits, combining the benefits of depth-first search (using

less memory) and breadth-first search (finding the optimal solution). It provides a way to use the best move from shallow searches to improve move-ordering in deeper searches.

- **Heuristic Alpha-Beta:** Heuristic alpha-beta uses evaluation functions to estimate the value of non-terminal nodes, guiding the search and pruning process more effectively based on heuristics. These evaluation functions consider factors like material balance, positional advantages, and other domain-specific knowledge to approximate the true minimax value of a position.