Slide 05 - Intro
Slide 29 - Agents
Slide 53 - reAct

EE2213 Introduction to Artificial Intelligence

Lecture 1

Dr. FAN Shaojing
fanshaojing@nus.edu.sg

OVERVIEW OF COURSE CONTENTS

- **Introduction (Shaojing)**
  - What is AI
  - Applications of AI
  - AI agent
- **Search (Shaojing)**
  - Uninformed search algorithms: breadth-first, depth-first, uniform-cost(Dijkstra's algorithm)
  - Informed search algorithms: greedy best-first, A*
  - Applications
- **Optimisation (Shaojing)**
  - Linear programming
  - Convex problems
  - Applications
- **Machine learning (Wang Si)**
  - Supervised and unsupervised learning: regression, classification, clustering
  - Neural networks and deep learning
  - Applications
- **Knowledge representation (Wang Si)**
  - Knowledge Representation and Reasoning
  - Propositional Logic
  - Applications
- **Ethical considerations (Shaojing)**
  - Bias in AI
  - Privacy concerns
  - Societal impact

Course Design: From Foundational to Advanced Intelligence

mechanistic, rule-based forms of intelligence

adaptive, data-driven forms of intelligence

abstract, reasoning-based forms of intelligence

deductive

- **Search**
- **Optimisation**

Machine learning

inductive

Knowledge representation

lower-Level/foundational intelligence

mid-level/adaptive intelligence

higher-level/reasoning Intelligence

Final Exam (40%)

Goal: To assess your ability to apply knowledge to new problems, rather than just recall facts.

Mode: Digital exam conducted via Examplify

Open-book exam: You may bring any printed or digital materials, but internet access is not allowed.

Reference of Examplify:
https://nus.atlassian.net/wiki/spaces/DAstudent/pages/22511650/Getting+Started

Agenda

- A brief history of AI and its current landscape
- What is an AI agent?
- ReAct agents and the rise of agentic AI

What is Artificial Intelligence (AI)?

Artificial intelligence is the study of agents that perceive their environment and take actions that maximize their chances of achieving their goals.

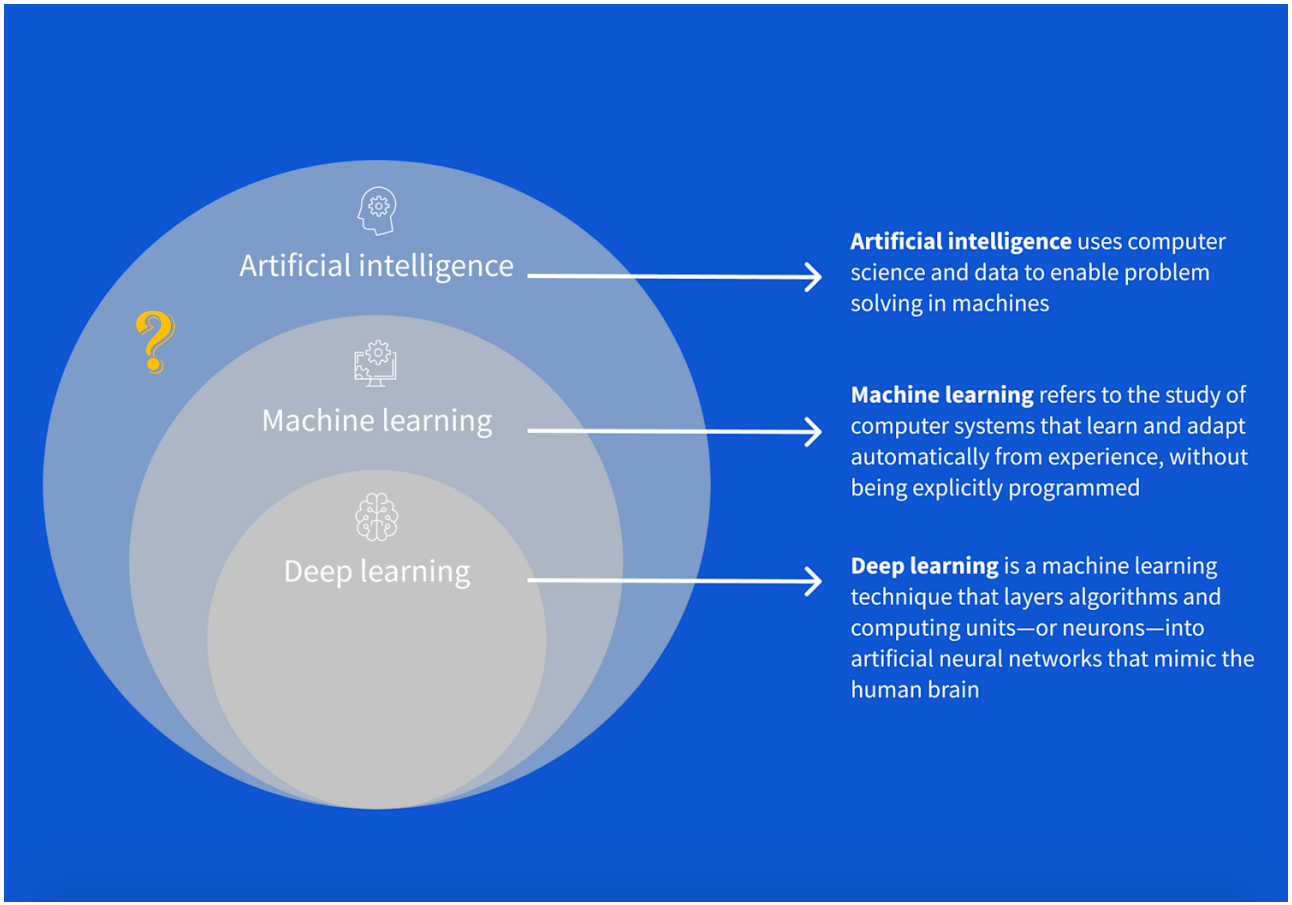
— Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.)*
*It is considered the "bible" of AI textbooks.

The science and engineering of making intelligent machines, especially intelligent computer programs.

— John McCarthy (1956), the “Father” of AI, who coined the term Artificial Intelligence

AI, Machine Learning, and Deep Learning

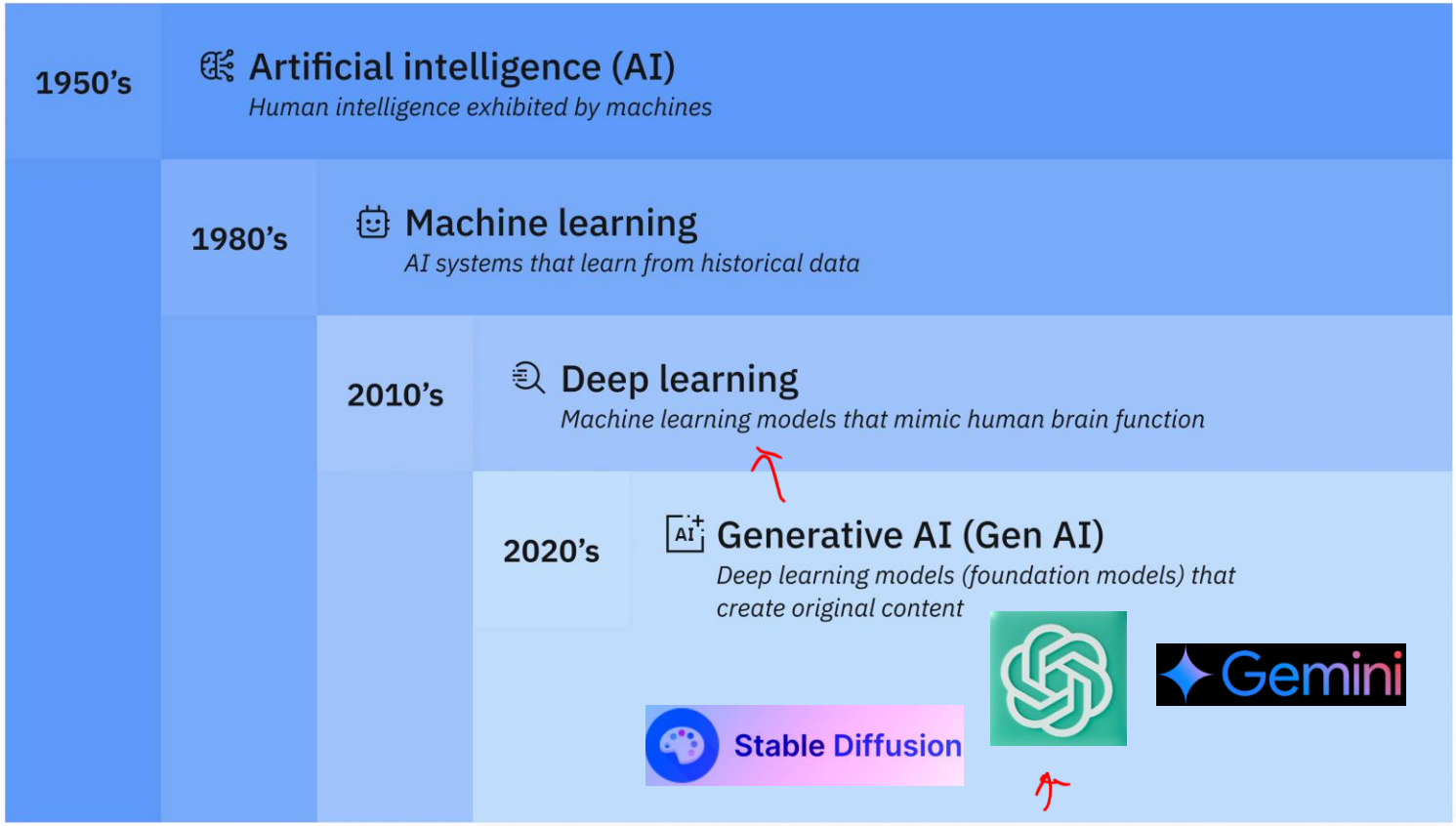
venn diagram
Deep Learning is a subset of Machine Learning, which is a subset of Artificial Intelligence.



Search algorithms (like BFS, DFS, A, etc.) are part of AI because they help an agent plan and make decisions, but not ML, as they follow rules rather than learn from data.*

Reference: https://www.coursera.org/articles/ai-vs-deep-learning-vs-machine-learning-beginners-guide

AI, Machine Learning, Deep Learning, and Generative AI

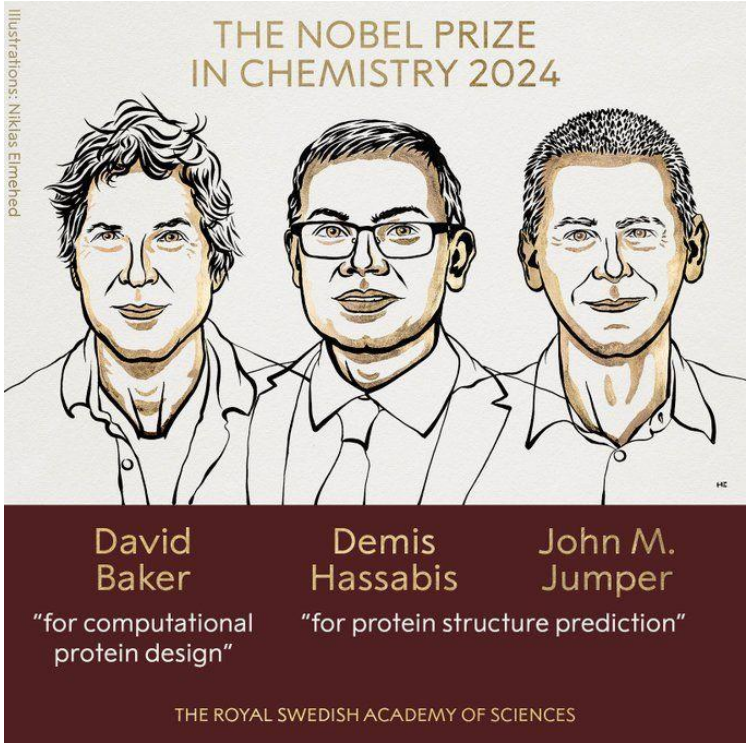


Credit: https://www.ibm.com/think/topics/artificial-intelligence

Why AI?

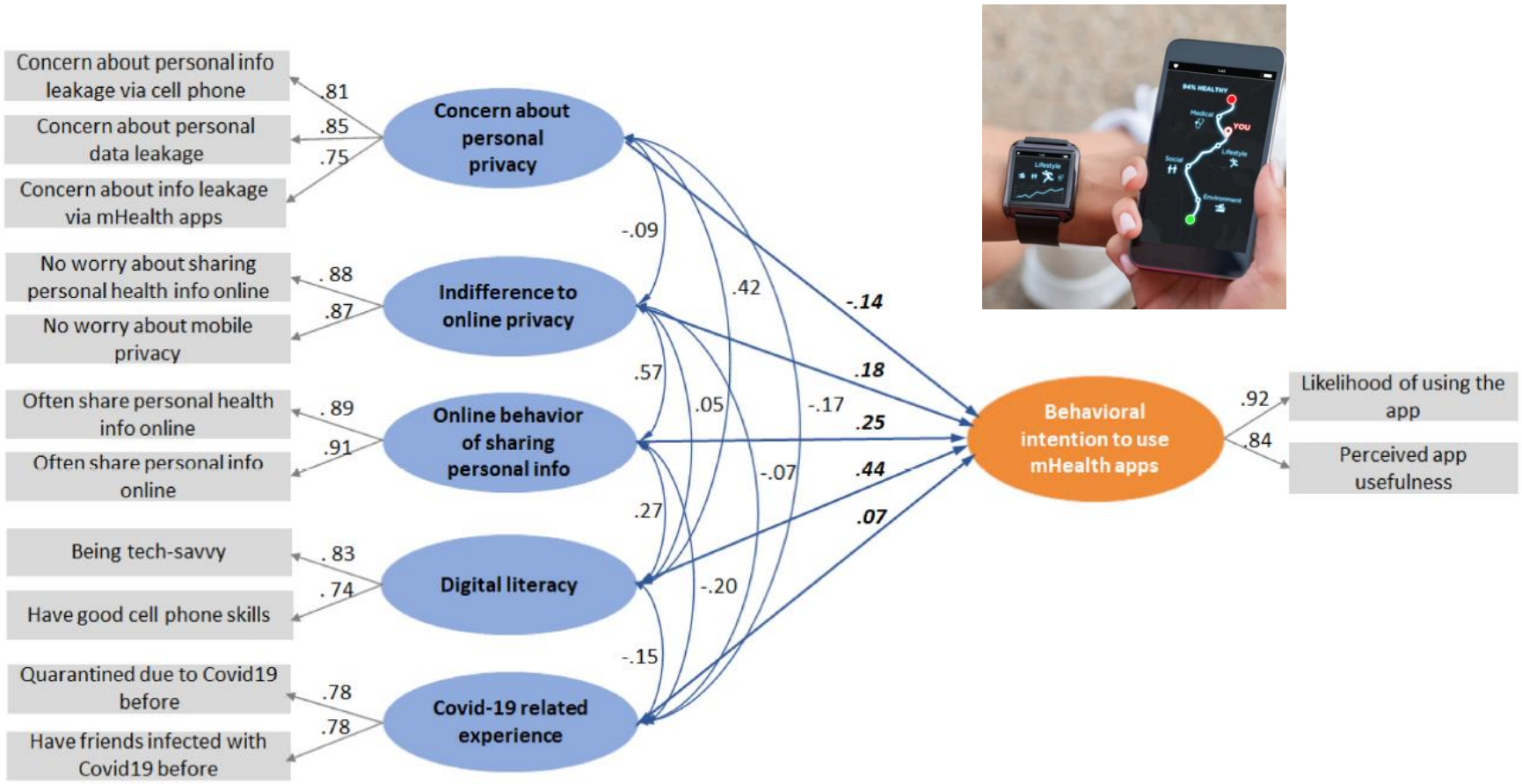
2024 Nobel prize in Chemistry for *Alphafold*.

A historic milestone: for the first time, an **AI-powered method** has been directly recognized by the Nobel Committee in Chemistry



News: https://www.theverge.com/2024/10/9/24265962/google-deepmind-ai-nobel-prize-winners-chemistry-protein?utm_source=chatgpt.com
News: DeepMind's new AlphaGenome AI tackles the 'dark matter' in our DNA, https://www.nature.com/articles/d41586-025-01998-w, June2025
Z.Avsec, et. al, "AlphaGenome: advancing regulatory variant effect prediction with a unified DNA sequence model", Google DeepMind

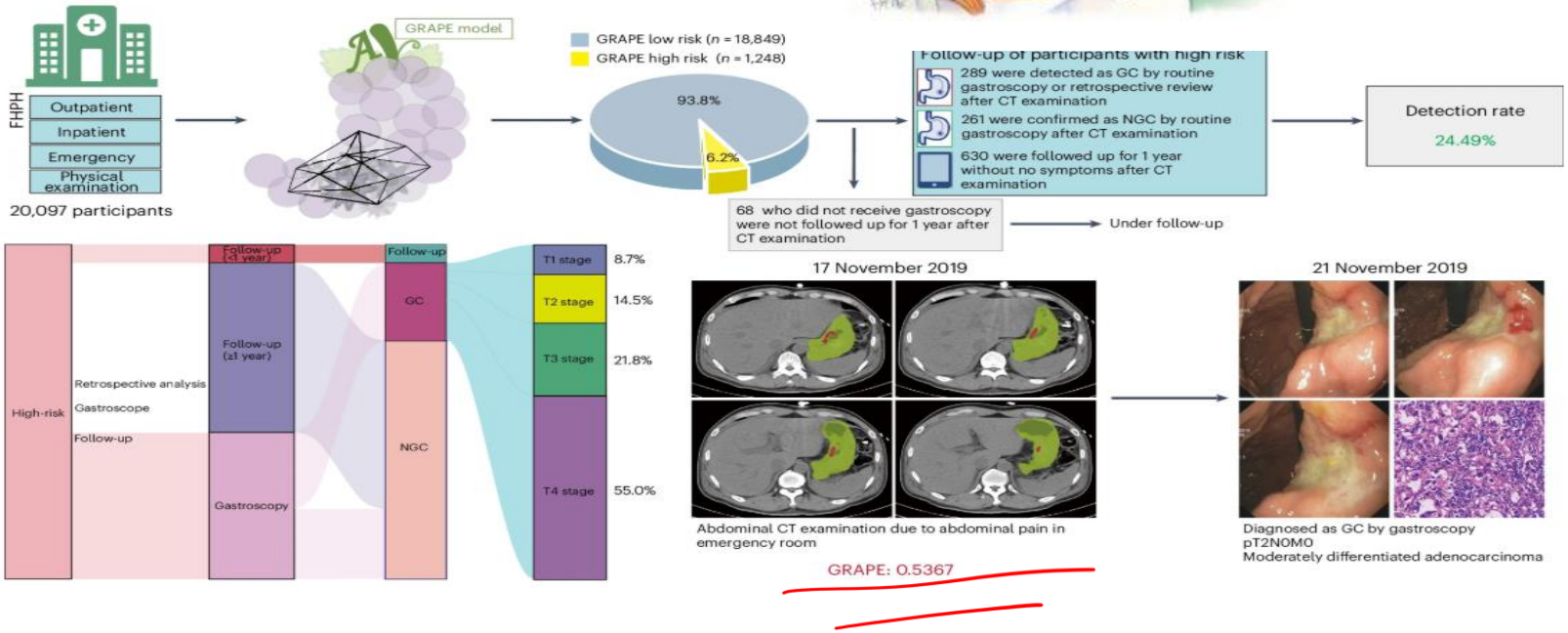
Why AI?



Fan, Shaojing, Ramesh C. Jain, and Mohan S. Kankanhalli. "A comprehensive picture of factors affecting user willingness to use mobile health applications." ACM Transactions on Computing for Healthcare 5.1 (2024): 1-31.

Why AI?

Illustration of gastroscopy



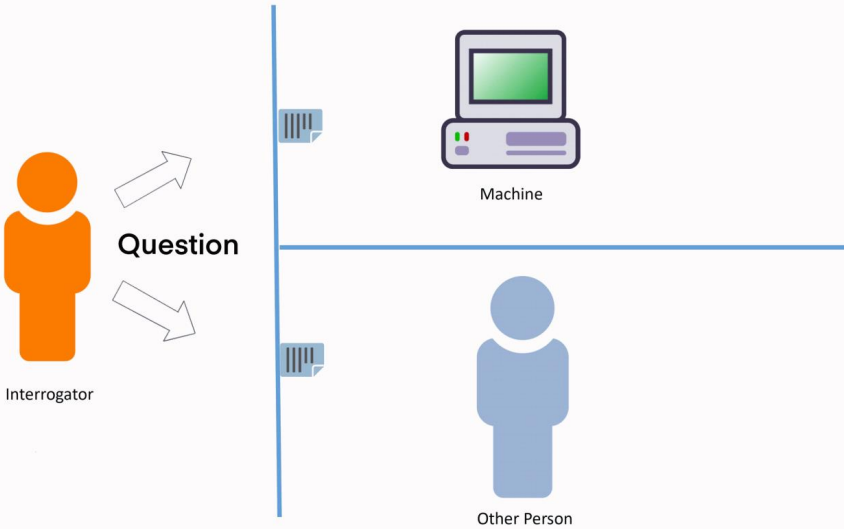
Hu, Can, et al. "AI-based large-scale screening of gastric cancer from noncontrast CT imaging." Nature Medicine (2025): 1-9.

Alan Turing and Turing Test (1950)

a test for intelligence in a computer, requiring that a human being should be unable to distinguish the machine from another human being by using the replies to questions put to both.

"A computer would deserve to be called intelligent if it could deceive a human into believing that it was a human."

— Alan Turing



Reference: Jones, Cameron R., and Benjamin K. Bergen. "Large language models pass the turing test." *arXiv preprint arXiv:2503.23674* (2025).

Turing Award

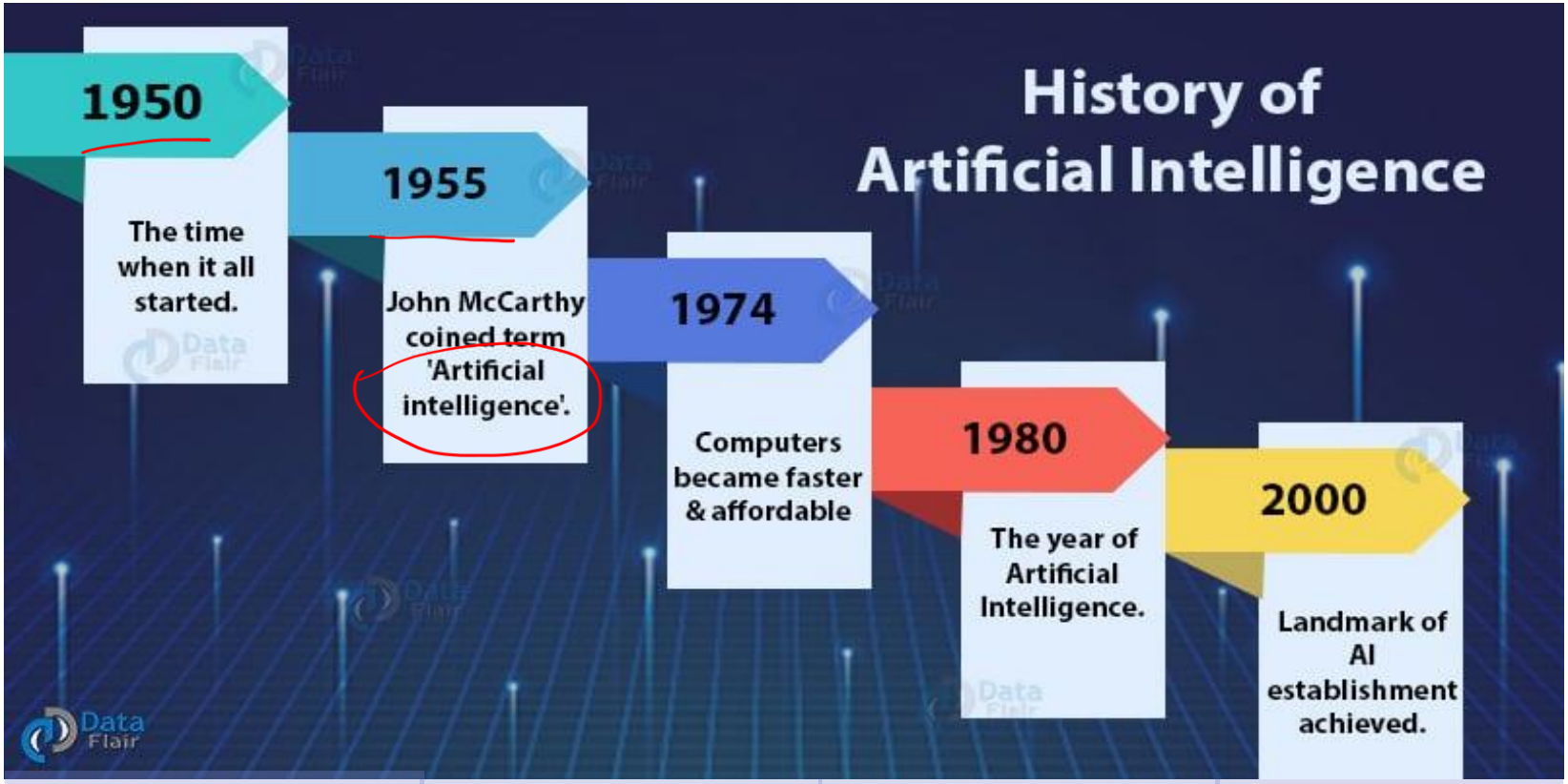
The top award in computer science, including the field of artificial intelligence (AI)

- (2024) Barto, Andrew; Sutton, Richard
- (2023) Wigderson, Avi
- (2022) Metcalfe, Robert Melancton
- (2021) Dongarra, Jack
- (2020) Aho, Alfred Vaino; Ullman, Jeffrey David
- (2019) Catmull, Edwin E. ; Hanrahan, Patrick M.
- (2018) Bengio, Yoshua; Hinton, Geoffrey E; LeCun, Yann

...

Reference (NUS120 Distinguished Speaker Series | Professor Yoshua Bengio):
https://www.youtube.com/watch?v=luJxOyryJ0o

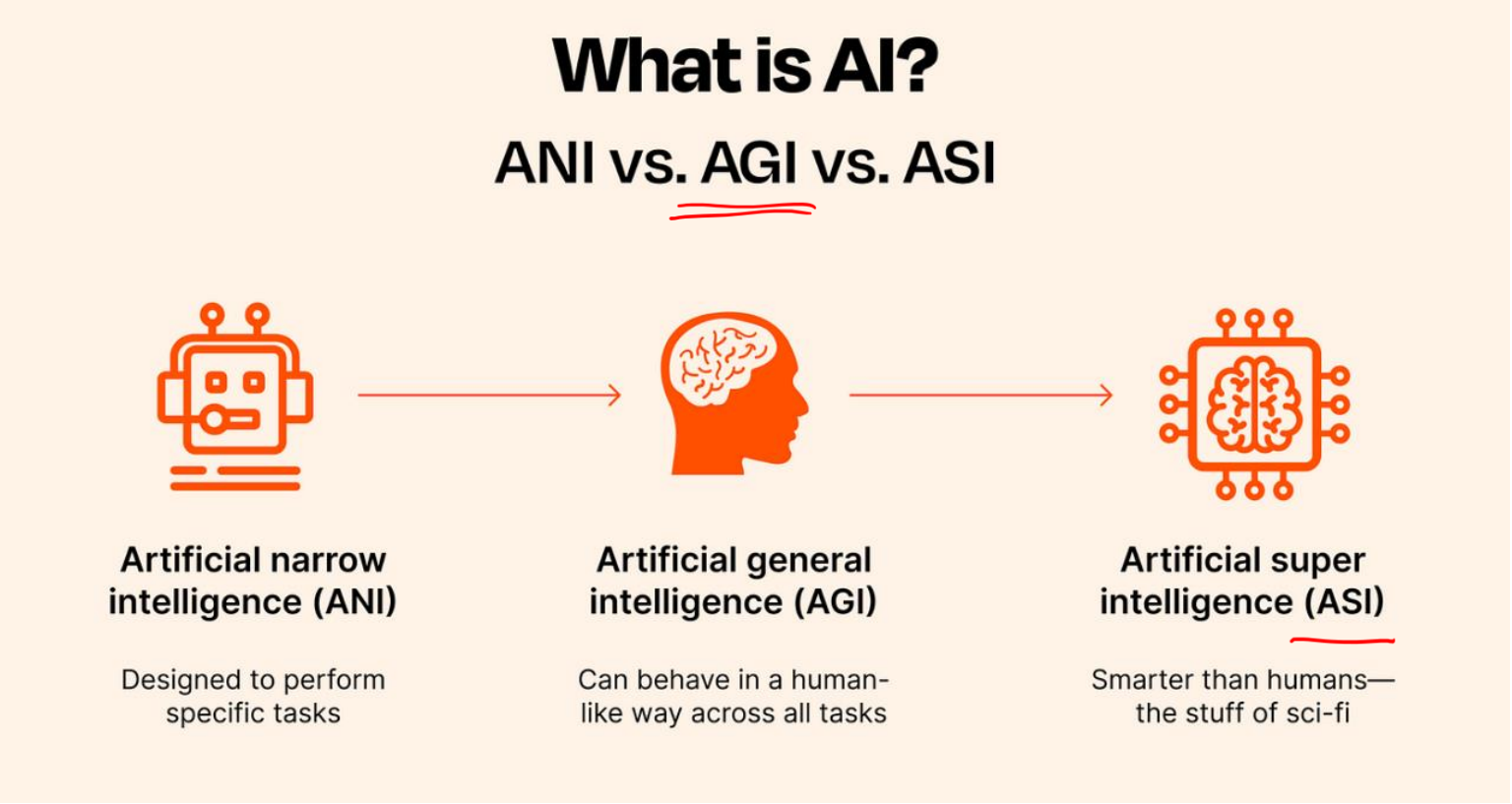
A Brief AI History



Birth of AI (1950s)	Early Development (1960s-70s)	Resurgence (1980s-2000s)	Modern AI (2010s-Present)
Turing Test (1950)	ELIZA chatbot	Rule-based systems	Deep learning
Dartmouth Conference (1956)	First AI Winter	Deep Blue (1997)	Generative AI

Reference: https://data-flair.training/blogs/history-of-artificial-intelligence/

AI in Future



Nick, Bostrom. "Superintelligence: Paths, dangers, strategies." 2014,

Agenda

- A brief history of AI and its current landscape
- What is an AI agent?
- ReAct agents and the rise of agentic AI

Artificial intelligence is the study of agents that perceive their environment and take actions that maximize their chances of achieving their goals.

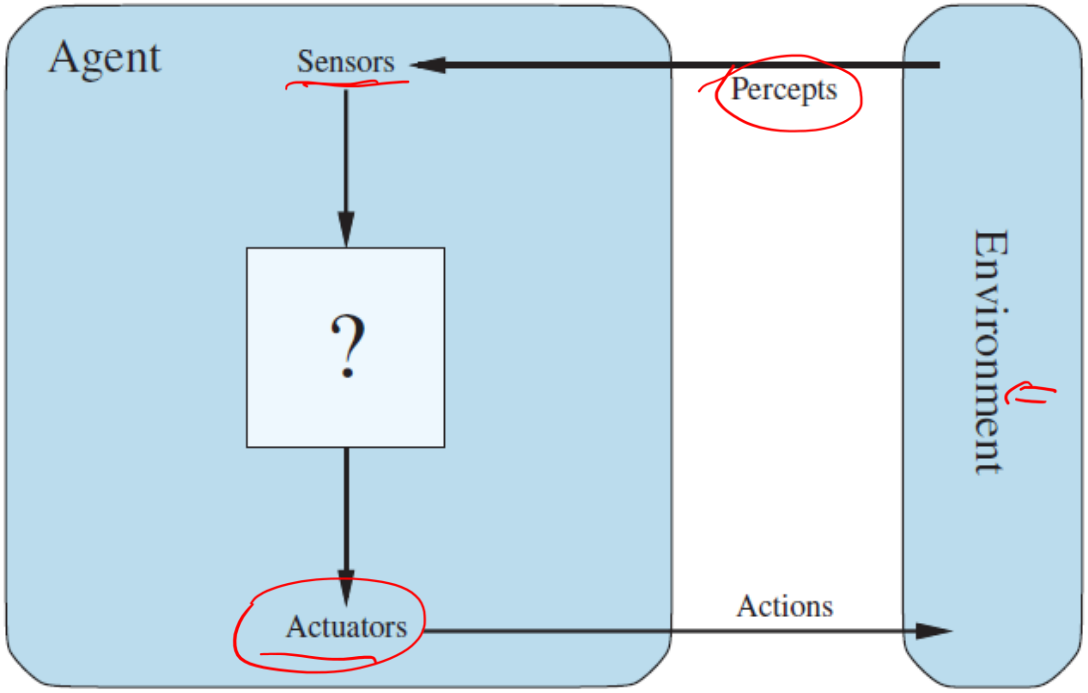
— Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.)

Definition of Agent

An **agent** is anything that **perceives** its environment through **sensors** and can **act** on its environment through **actuators**

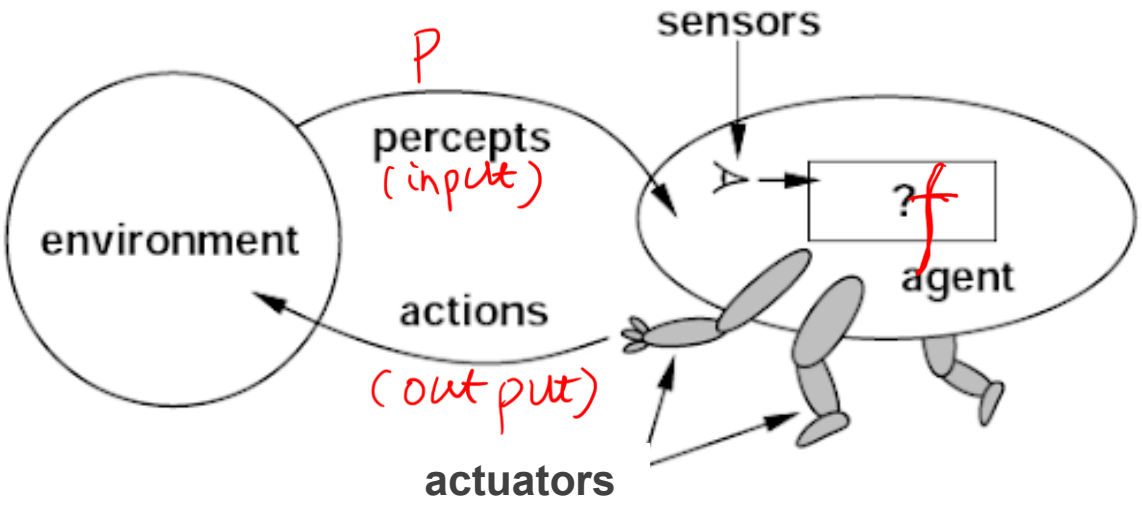
A **percept** is the agent's **perceptual inputs** at any given instance

An **actuator** is the part of an agent that allows it to take **action** in the environment based on its decisions.



Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P54 – P57.

Definition of Agent (cont.)



An agent is specified by an *agent function* $f: P \rightarrow a$ that maps a sequence of percept vectors P to an action a from a set A:

$$P = [p_0, p_1, \ldots, p_t]$$
$$A = [a_0, a_1, \ldots, a_t]$$

Q1 Which of the following statements about agents are true? Select all that apply.

a) Human and robot agents differ because the latter do not have actuators. In AI, all agents have actuators, which are necessary for action.

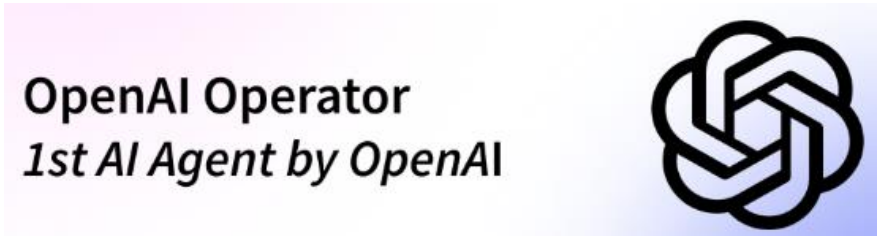
b) Eyes and ears are examples of a human agent's actuators. sensors

c) An agent senses its environment through perceptors. sensors

d) Percepts are the agent's perceptual inputs from the environment at any given time. ✓

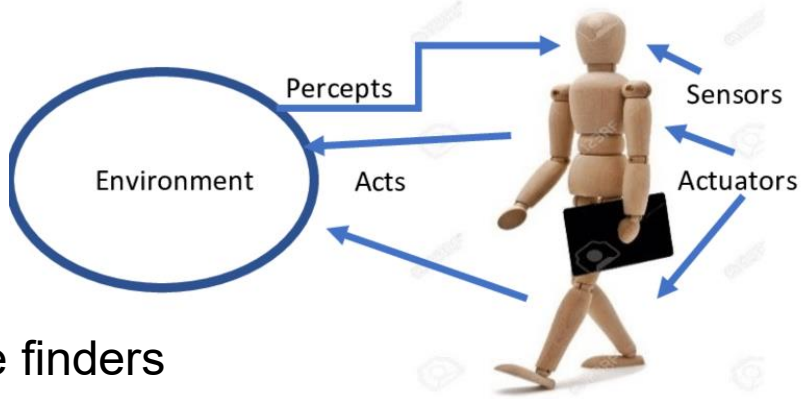
e) Touchless faucets can also be viewed as agents. ✓

Example of AI Agent



Examples of Agents

Human agent:
- Sensors: eyes, ears, …
- Actuators: hands, legs, mouth, …



Robotic agent:
- Sensors: cameras and infrared range finders
- Actuators: various motors

Agents include humans, robots, softbots, smart lighting systems…

An ***agent*** is anything that can be viewed as
- ***perceiving*** its ***environment*** through ***sensors*** and
- ***acting*** upon that environment through ***actuators***

Quick Question 1

True or false: A thermostat that adjusts room temperature based on sensor readings can be considered an agent in AI terms.

- True
- False

Ans: True. A thermostat is an AI agent because it senses the room temperature through sensors, and acts upon the environment by turning the heater/cooler on or off to adjust it.

Performance Measure of an AI Agent

How do we know if an agent is doing a good job?

We define a ***performance measure***:

- criterion to judge how successful the agent's behavior is

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P57 – P58.

Rule of Thumb for Performance Measure

It is better to design performance measures according to the goals you want to achieve in the environment, rather than prescribing how you think the agent should behave.

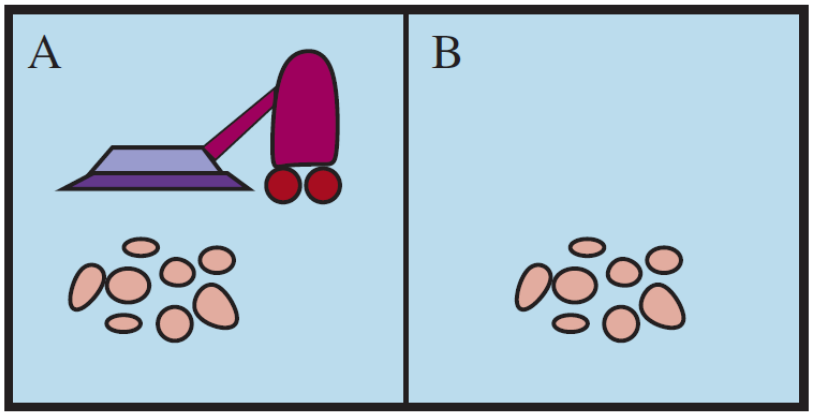
E.g., performance measure for a chatbot:

1. +1 point for every helpful message it sends. ✗
2. +1 point when it fully resolves the user's issue by the end of the conversation. ✓

Performance Measure: Example

Performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

- +1 point for each clean square at time T
- -1 point for each move
- -1000 for more than *k* dirty squares

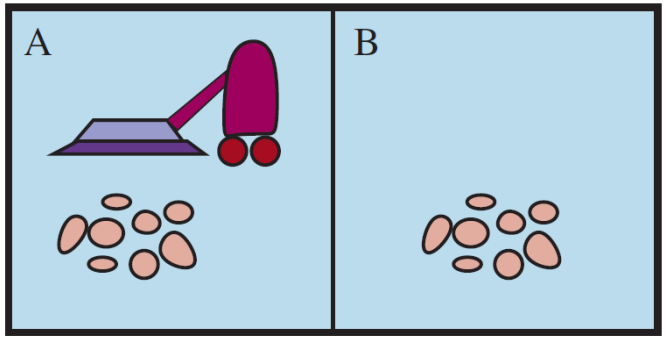


Quick Question 2

Consider two ways to reward a cleaning robot:

1. +1 point for every time the robot cleans a square
2. +1 point for each square that is clean at a specific time T

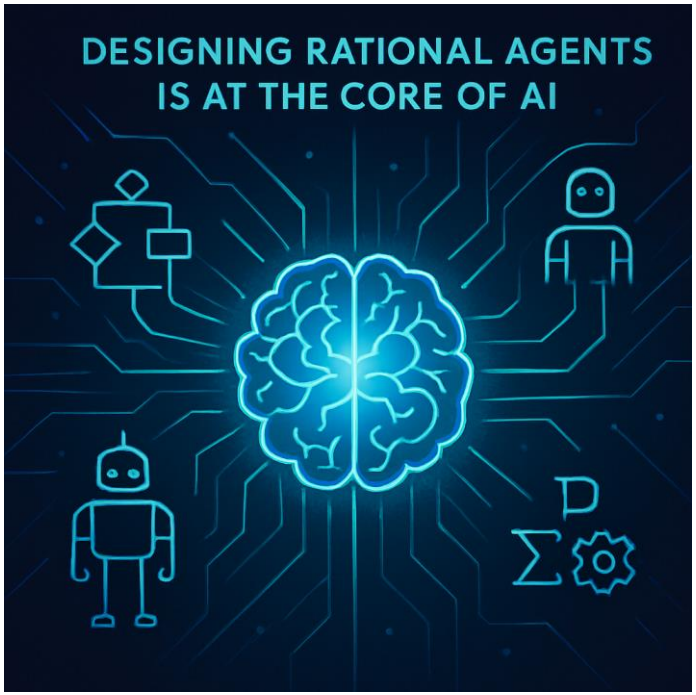
Which of these is a better performance measure? Why?



Ans: The second measure is better because it rewards the robot for maintaining cleanliness at a specific time, not just for cleaning repeatedly.

Introduction of Rational Agent

An agent should strive to "*do the right thing*", based on what it can perceive and the actions it can perform. The right action is the one that helps the agent *achieve its goals most effectively*.



Definition of Rational Agent (cont.)

A rational agent that acts rationally will NOT always achieve the true best outcome in every situation.

- Are rational agents omniscient?

No – they only know what their sensors tell them.

- Are rational agents clairvoyant?

No – they may lack knowledge of the environment dynamics. They don't magically know the future or hidden parts of the environment.

- Do rational agents explore and learn?

Yes – in unknown environments these are essential

- Are rational agents autonomous (i.e., transcend initial program)?

Yes – as they learn, their behavior depends more on their own experience

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P58 – P60.

Q2 Which of the following statements regarding rational agents are true? Select all that apply.

a) Rational agents always know the environment dynamics and can predict future updates of the environment. ✗ Agents are not clairvoyant

b) A rational agent is omniscient ✗

c) A rational agent selects an action that maximizes the expected value of its performance measure ✓

d) A rational agent is not guaranteed to find optimal solutions. ✓

Two levels of optimality:

- *Actual optimality:* what truly turns out to be best
- *Agent-optimality (rationality):* what is best according to the agent's knowledge at the time

a) Rational agents are assumed to know the complete environment dynamics and can predict how the environment will respond to their actions. (WRONG) b) A rational agent chooses actions that are expected to yield the highest performance according to a predefined measure. (YES) c) A rational agent that acts rationally will always achieve the true best outcome in every situation. (WRONG) d) Smart home devices, such as automatic lights, can be considered as rational agents (YES

Q2 Takeaway: Understanding rationality

A rational agent is *not guaranteed* to find optimal solutions

With *perfect* knowledge and *unlimited* computing power, the agent would find the true optimal solution.

However, the agent is not omniscient or clairvoyant: it does not have perfect knowledge of the environment.

In reality, agents have incomplete or noisy information and act optimally based on the current evidence available—this is called *bounded rationality.*

The key distinction:

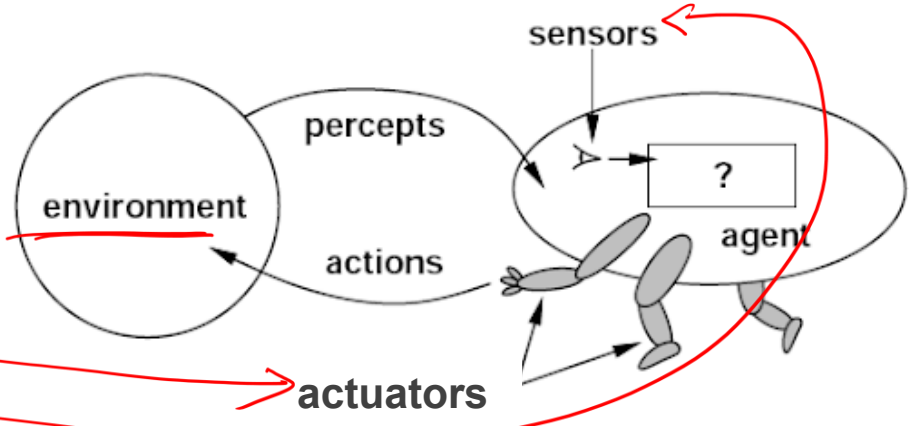
- Actual optimality: the best possible outcome in reality.
- Rationality: the best expected outcome based on the agent's current evidence and existing knowledge.

Design a Rational Agent: Task Environment

To design a rational agent, we need to specify a **task environment** --- the setting/context in which the agent will operate and make decisions, and the problem specification the agent is meant to solve

PEAS: to specify a task environment

- ***P***erformance measure
- ***E***nvironment
- ***A***ctuators
- ***S***ensors



Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P60 – P61.

PEAS: Example 1

Agent: autonomous vehicle

Performance measure *(evaluate how well it performs)*
- safety, speed, following traffic laws, comfort, maximize profits

Environment *(everything it interacts with)*
- roads, other traffic, pedestrians, customers

Actuators *(how it acts on the environment)*
- steering, accelerator, brake, signal, horn

Sensors *(where it gathers data)*
- cameras, LiDAR, speedometer, GPS

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P61.

PEAS: Example 2

Agent: Medical diagnosis system

Performance measure

- patient health outcomes, minimized treatment costs, reduced risk of legal issues

Environment

- patients, hospitals, medical staff, healthcare records

Actuators

- display screen (to show questions, diagnoses, treatment plans, referrals)

Sensors

- keyboard or interface input (for entering symptoms, test results, patient responses)

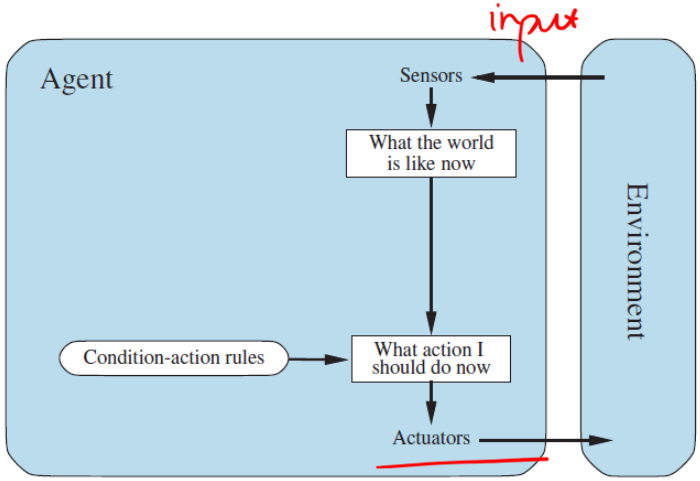
Simple Reflex Agents

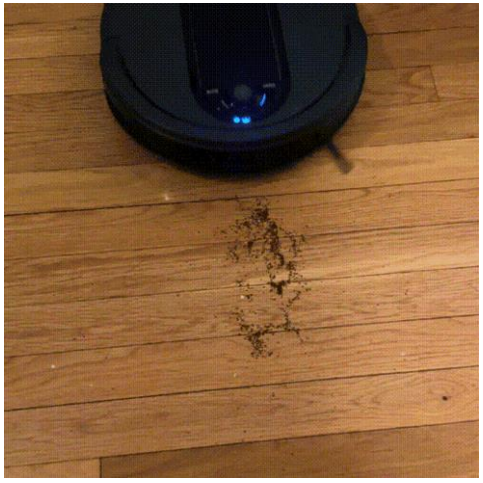
- **A Simple Reflex Agent**
  - Selects action based on the current percept.
  - Directly map states to actions. without any notion of goals.

- Operate using *condition-action* rules.
  - **If** (condition) **then** (action)

- Example:
  - A vacuum cleaner robot that reacts to dirt.
  - **If** *(current-location-is-dirty) then (suck-dirt)*

- Problem: no notion of goals
  - It doesn't plan where to clean next or remember cleaned areas.
  - *may repeatedly visit clean spots and miss others*.

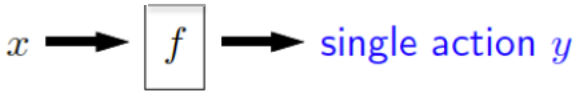




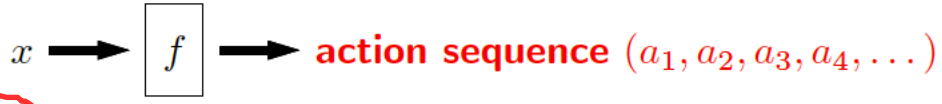
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P67 – P69.

From Simple Reflex to Problem Solving Agents

- **Simple Reflex Agents**
  - Directly map states to actions.
  - No notion of goals. Do not consider action consequences.



- **Problem Solving Agents**
  - Adopt a goal
  - Consider *future actions* and the desirability of their outcomes
  - Solution: a *sequence of actions* that the agent can execute leading to the goal.
  - Today's focus.



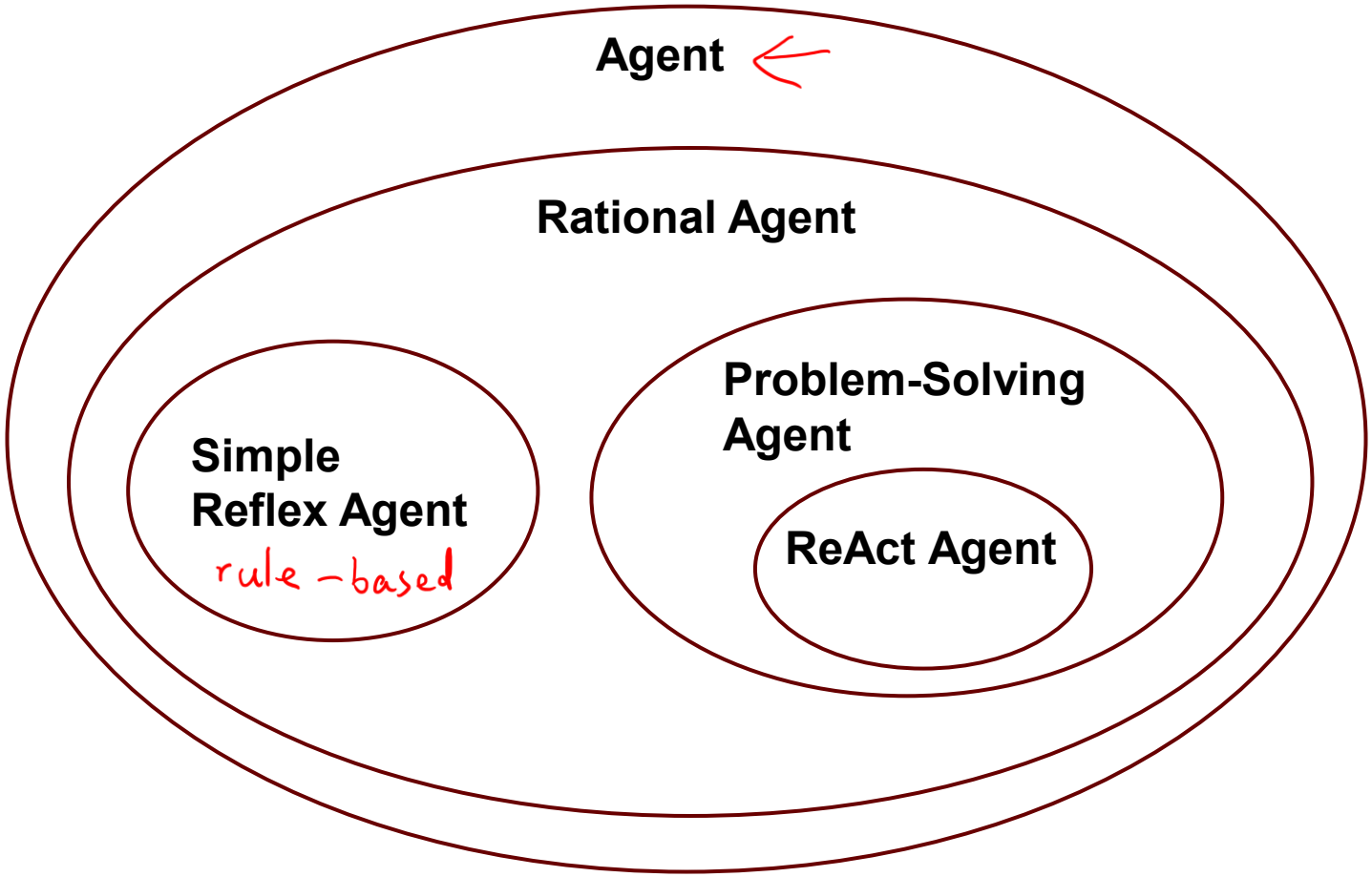
using search methods to achieve this

Definition of Problem Solving Agents

- **Adopt a Goal**: The agent has a specific objective to achieve.
- **Plan Ahead**: The agent evaluates possible *future actions* and their outcomes, typically by examining the outcomes of each available action in every state and assessing their desirability.
- **Find a Solution**: The agent determines *a sequence of actions* that will lead it from the *initial state* to the *goal*.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P81-P82

Summary



Q3. Which of the following are necessary to formally define a problem for a problem-solving agent? Select all that apply.

a) A clear goal condition that defines when the agent has successfully solved the problem ✓

b) Specifying the outcome of each action that the agent could take ✓

c) The initial state from which the agent begins its problem-solving process ✓

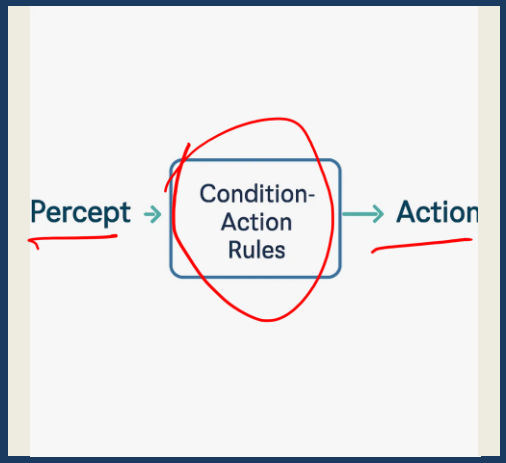
d) The set of possible actions the agent can perform in each state ✓

Agenda

- A brief history of AI and its current landscape
- What is an AI agent?
- ReAct agents and the rise of agentic AI

Simple Reflex vs. ReAct Agents

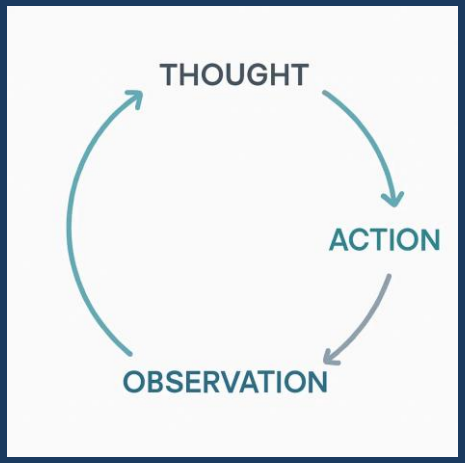
Simple Reflex Agent



- Direct percept-to-action mapping
- Uses predefined *condition-action* rules
- No memory of past actions

VS

ReAct Agent



- Iterative thought-action-observation loop
- Combines *reasoning with action*
- Can handle complex, multi-step tasks

* This slide was created by Manus, a “general AI agent.”, and verified by the instructor.

ReAct Agent Example

San Francisco Mother and Child Forced to Live in Car Due to Skyrocketing Housing Costs.

Please evaluate human reactions to this post: Do they believe it's true? Will they share it? Could it go viral?

Our Evaluator

Likelihood of being shared: 0.36
Trustworthiness: 0.45
Chance of being generated by AI: 0.54
Image matches text in content: 0.56
Image matches text in emotions: 0.49

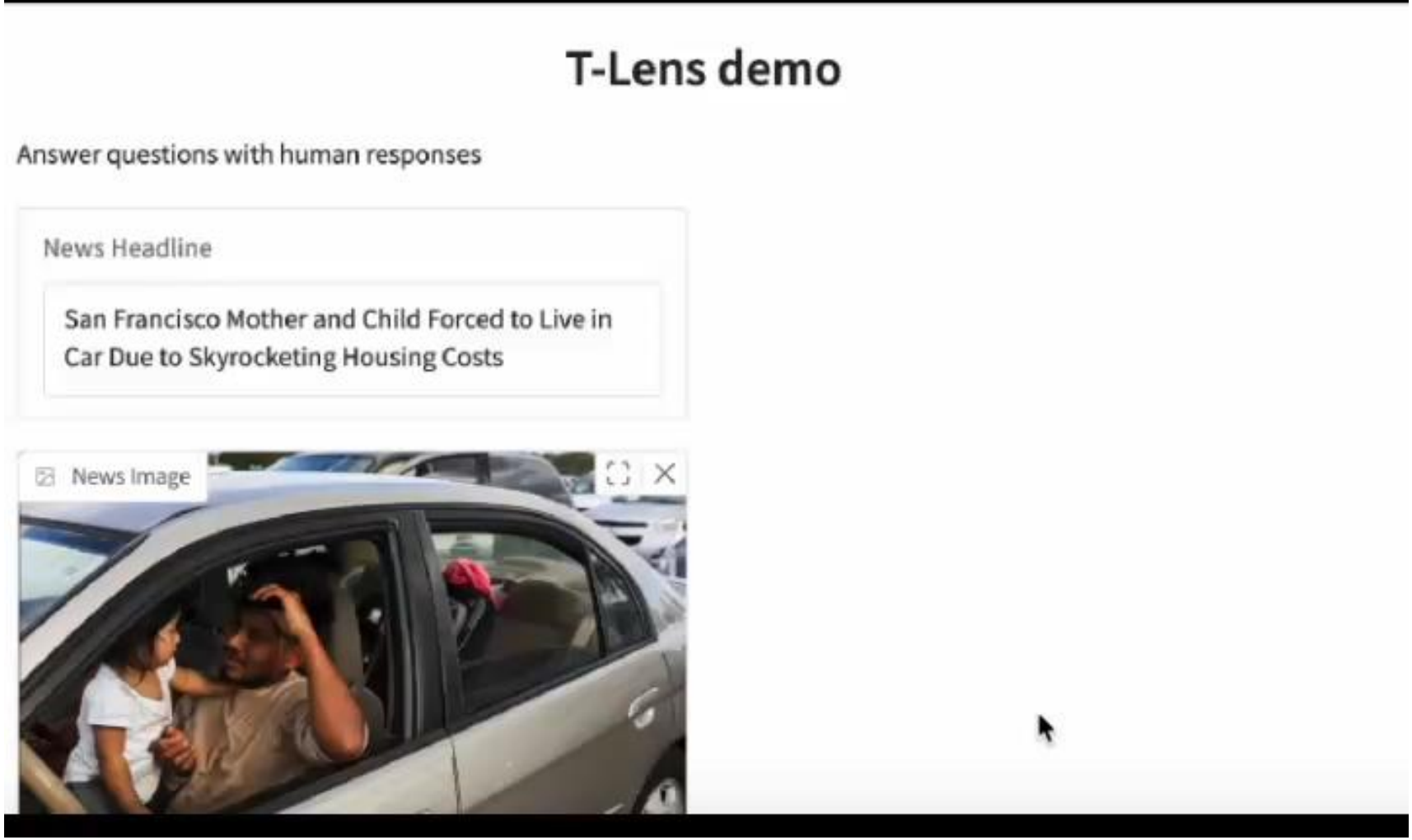
Model predictions

The title discusses a mother and child facing housing issue, while the image shows a man and a child in a car. Content consistency is moderate. The title conveys low happiness, and the image is likely to depict neutral emotions, with low sentiment consistency. Given these factors, there's *a moderate chance the post is fake or AI-generated*. This, along with the post's presentation and topic, implies a low level of trustworthiness and a low likelihood of being shared. Based on these observations, users are likely to show low acceptance and engagement with the post, suggesting a *low chance of it going viral*.

LLM-aided explanation

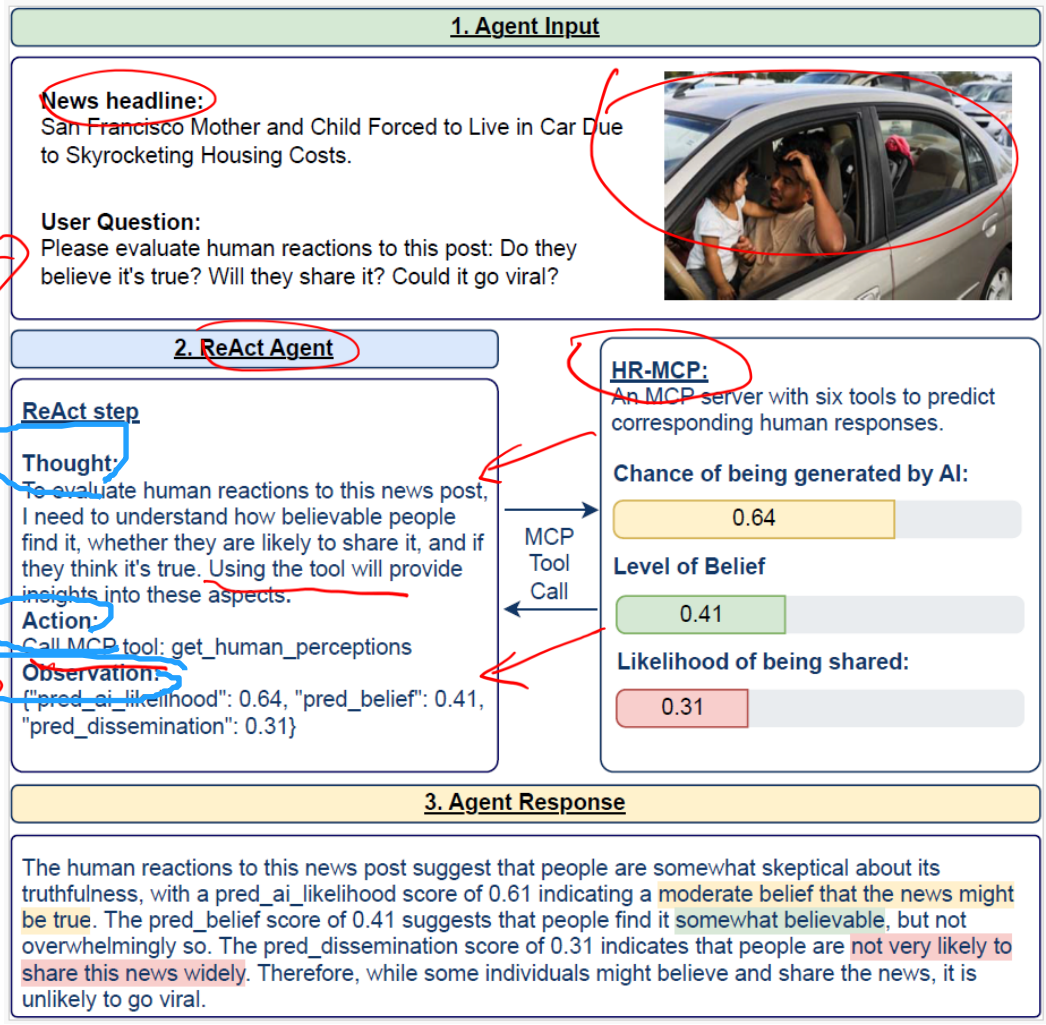
Zhiqi Shen*, Shaojing Fan*, Danni Xu, Terence Sim, and Mohan Kankanhalli. "Modeling Human Responses to Multimodal AI Content", Submitted to AAAI 2026.

ReAct Agent Example (Demo)



Zhiqi Shen*, Shaojing Fan*, Danni Xu, Terence Sim, and Mohan Kankanhalli. “Modeling Human Responses to Multimodal AI Content”, Submitted to AAAI 2026.

ReAct Agent Example (cont.)



- Understand the query (language understanding)
- Select appropriate tools needed to generate a response
- Gather predicted human responses.
- Understand human reactions about the news and produce an answer accordingly.

Zhiqi Shen*, Shaojing Fan*, Danni Xu, Terence Sim, and Mohan Kankanhalli. "Modeling Human Responses to Multimodal AI Content", Submitted to AAAI 2026.

Traditional AI Systems vs. Agentic AI

Features	Traditional AI	Agentic AI
Decision-Making	Follows rules, no autonomy	Independent goal-setting & action
Adaptability	Limited, needs reprogramming	Learns & adjusts continuously
Reactivity	Waits for input	Predicts and acts proactively
Execution	Requires step-by-step guidance	Self-directed execution

Reference: https://www.civo.com/blog/what-is-agentic-ai

EE2213 Introduction to Artificial Intelligence

Lecture 2

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

OVERVIEW OF COURSE CONTENTS

- **Introduction (Shaojing)**
  - What is AI
  - Applications of AI
  - AI agent
- **Search (Shaojing)**
  - Uninformed search algorithms: breadth-first, depth-first, uniform-cost (Dijkstra's algorithm)
  - Informed search algorithms: greedy best-first, A*
  - Applications
- **Optimisation (Shaojing)**
  - Linear programming
  - Convex problems
  - Applications
- **Machine learning (Wang Si)**
  - Supervised and unsupervised learning: regression, classification, clustering
  - Neural networks and deep learning
  - Applications
- **Knowledge representation (Wang Si)**
  - Knowledge Representation and Reasoning
  - Propositional Logic
  - Applications
- **Ethical considerations (Shaojing)**
  - Bias in AI
  - Privacy concerns
  - Societal impact

Problem solving as search

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P81-P122

Problem solving as search

Problem-Solving Agents

- Adopt a goal to achieve
- Plan ahead by considering future actions and the desirability of their outcomes
- The is a sequence of actions the agent can execute to reach the goal

Search

- *Generic definition*: The *computational process* that a *problem-solving agent* uses to find that optimal sequence of actions.
- *Narrow definition*: The specific algorithm used to find an optimal path to the goal.

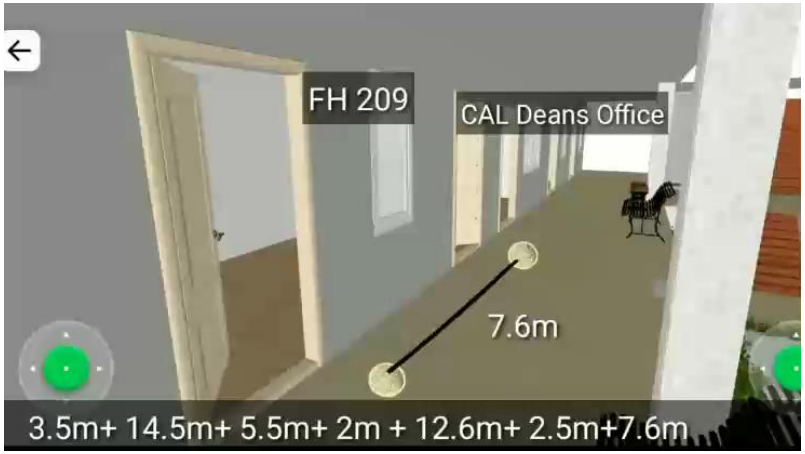
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P81-P82

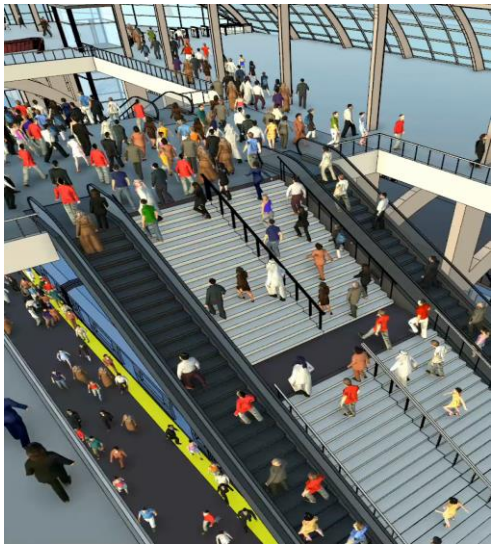
Agenda

- Background: Why search matters
- Search problem formulation
- Uninformed search (breadth-first search, depth-first search)

how can i find a way to save the princess
8-puzzle game
chess
fastest route from utown to changi
road trip from san francisco to new york
daily navigations

Civil engineering: Evacuation planning

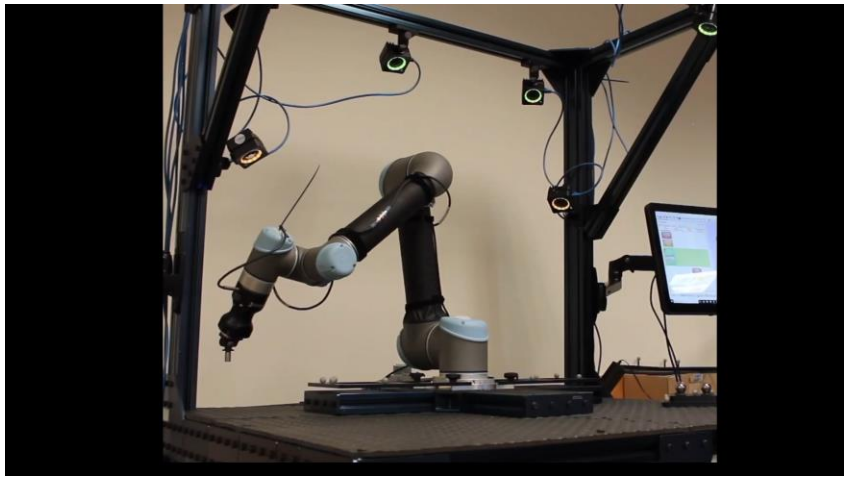


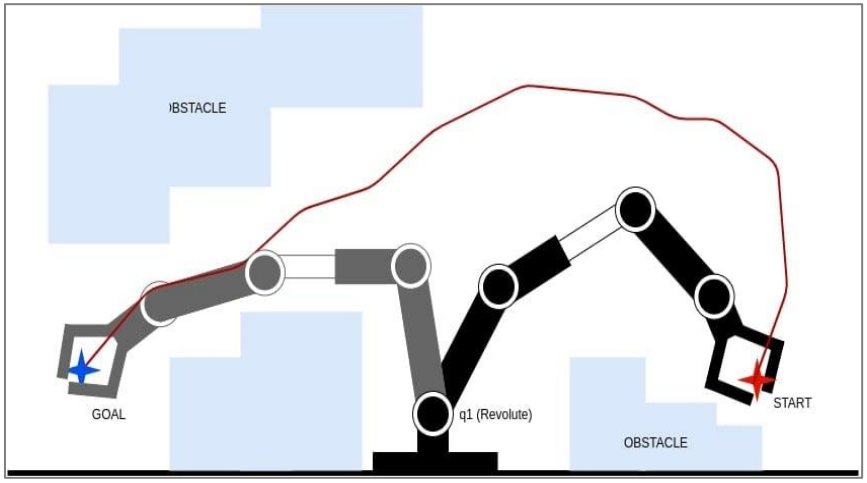


Khakzad, Nima. "A methodology based on Dijkstra's algorithm and mathematical programming for optimal evacuation in process plants in the event of major tank fires." *Reliability Engineering & System Safety* 236 (2023): 109291.

Mirahadi, Farid, and Brenda Y. McCabe. "EvacuSafe: A real-time model for building evacuation based on Dijkstra's algorithm." *Journal of Building Engineering* 34 (2021): 101687.

Robotics: Motion planning

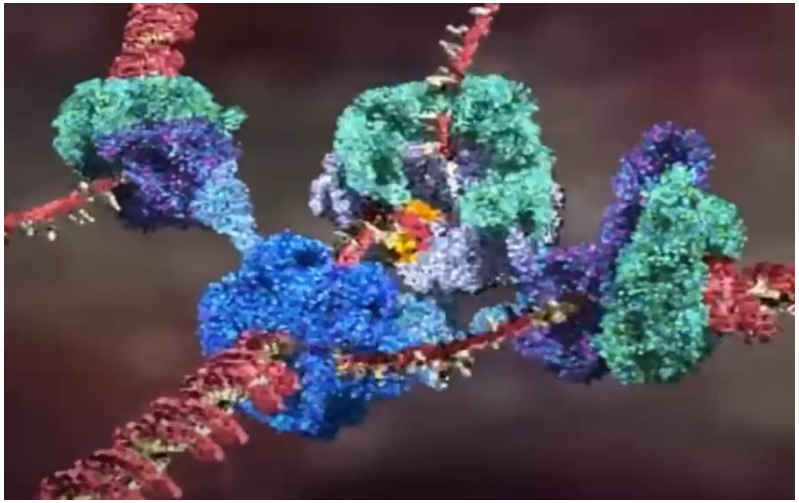


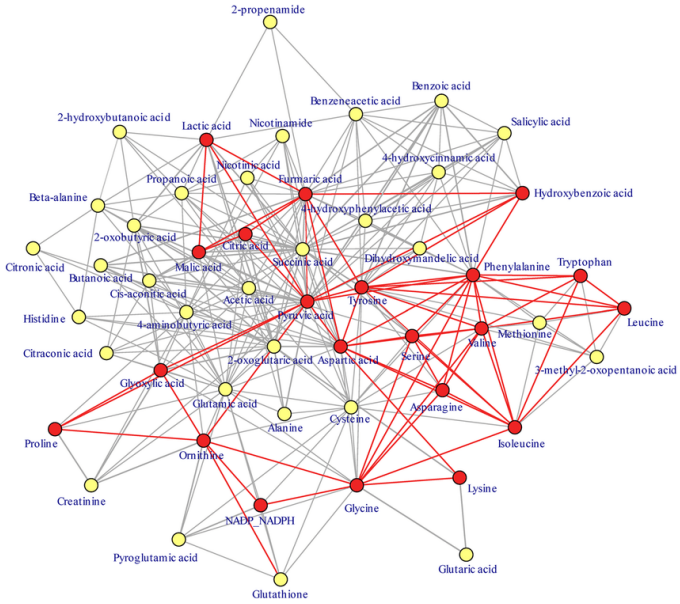


Sadiq, Ahmed T., Firas A. Raheem, and N. Abbas. "Ant colony algorithm improvement for robot arm path planning optimization based on D* strategy." *International Journal of Mechanical & Mechatronics Engineering* 21.1 (2021): 96-111.

Robotics computational motion planning, University of Pennsylvania, Coursera, HomeWork Project. https://www.youtube.com/watch?v=evJneD3WbyQ&t=8s

Bio-engineering: Optimising metabolic pathways for targeted biomolecule production





Krieger, Spencer, and John Kececioglu. "Shortest Hyperpaths in Directed Hypergraphs for Reaction Pathway Inference." *Journal of Computational Biology* 30.11 (2023): 1198-1225.

Rahman, Md Mostafizur, et al. "A novel graph mining approach to predict and evaluate food-drug interactions." *Scientific reports* 12.1 (2022): 1061.

Agenda

- Background: Why search matters
- Search problem formulation
- Uninformed search (breadth-first search, depth-first search)

Formulating search problem: 8-puzzle

Formulate *goal*

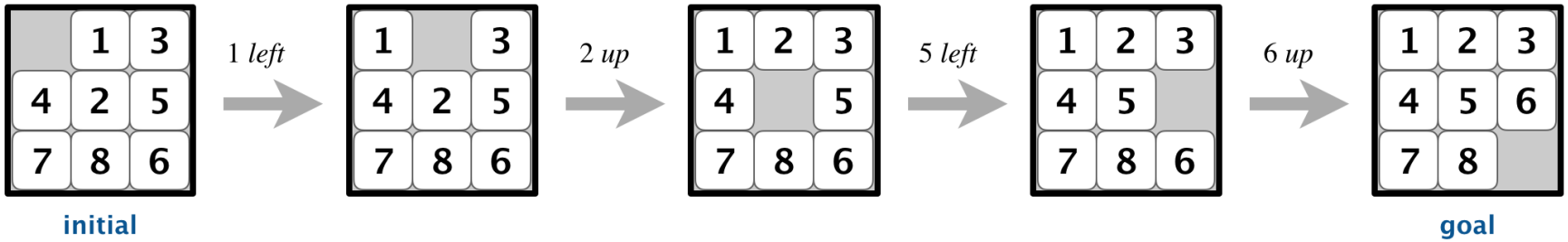
- Pieces to end up in order as shown

Formulate *search problem*

- ***States***: configurations of the puzzle
- ***Actions***: Move one of the movable pieces
- ***Performance*** measure: minimize total moves

Find *solution*

- Sequence of pieces moved: 1, 2, 5, 6



Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P83 – P88.

Formulating search problem: Route finding

Formulate *goal*

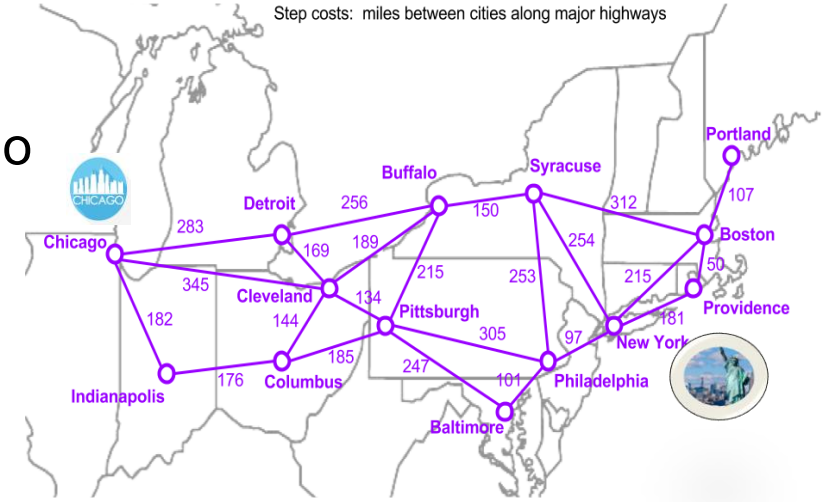
- Shorted driving route from Chicago to New York

Formulate *search problem*

- ***States***: various cities
- ***Actions***: drive between cities
- ***Performance measure***: minimize travel distance

Find *solution*

- Sequence of cities: e.g., Chicago, Indianapolis, Columbus, Pittsburgh, …



General search problems formulation

In general, a search problem is defined by:

can use dijkstra as an example to map each definition

1. ***State space:*** a set S containing all possible states of the environment.
2. ***Initial state:*** $s_i \in S$, indicating where the agent starts.
3. ***Actions:*** *a set* A containing possible moves from a state.

   $\forall\, s \in S$: $A = \text{ACTIONS}(s) =$ actions executable in s
4. ***Transition model :*** Result of applying an action.

   $\forall\, a \in \text{ACTIONS}(s)$, $\text{RESULT}(s, a) \rightarrow s'$, s' is called a successor of s
5. ***Action cost:*** Cost of applying an action, ACTION-COST(s, a, s'), aligned with the performance measure
6. ***Goal state(s):*** Desired state(s) with a desired property

   The agent uses a **goal test** to check if the current state is a goal state

   IS-GOAL(s): s is a goal state if IS-GOAL(s) is true

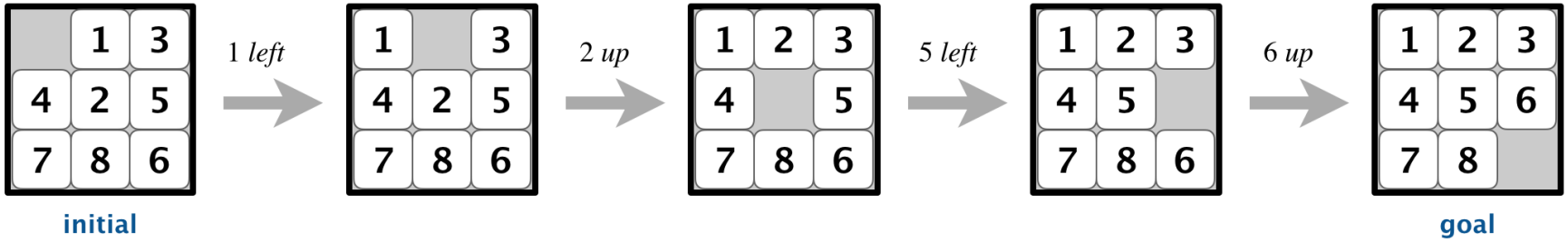
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P83

Key concepts: Agent and state

Agent (simplified definition):
entity that perceives its environment and acts upon that environment

State:
a configuration of the agent and its environment, typically represented by the letter s

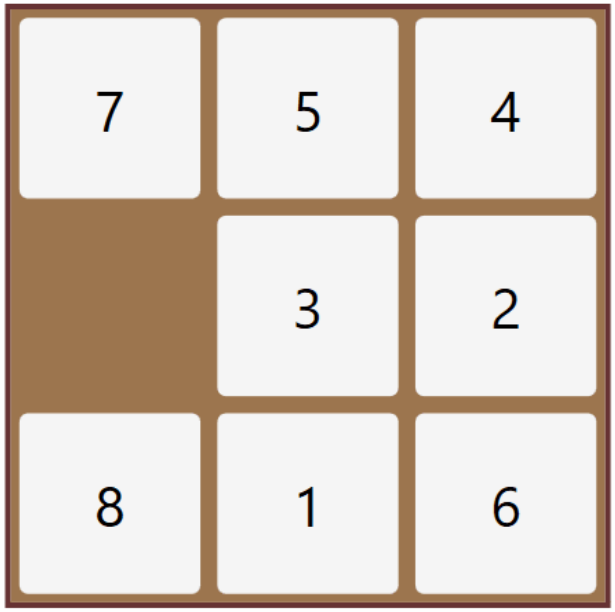
Initial state:
the state in which the agent begins, typically represented as s_i

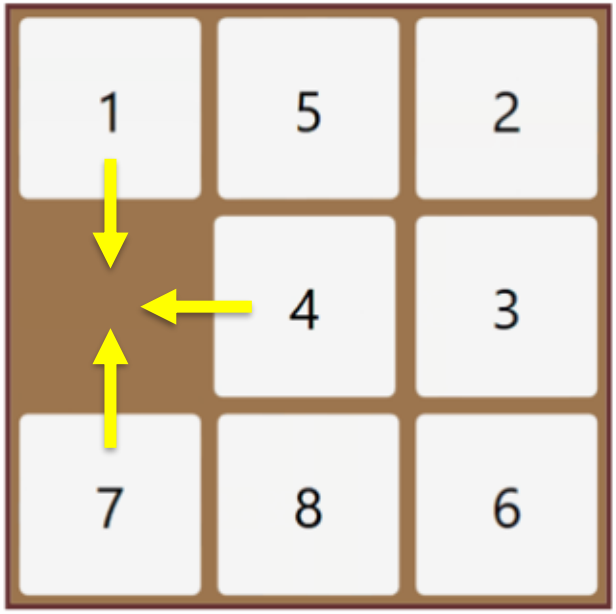


Key concepts: Actions

Actions

ACTIONS(s): the set of actions, that can be executed in state s.



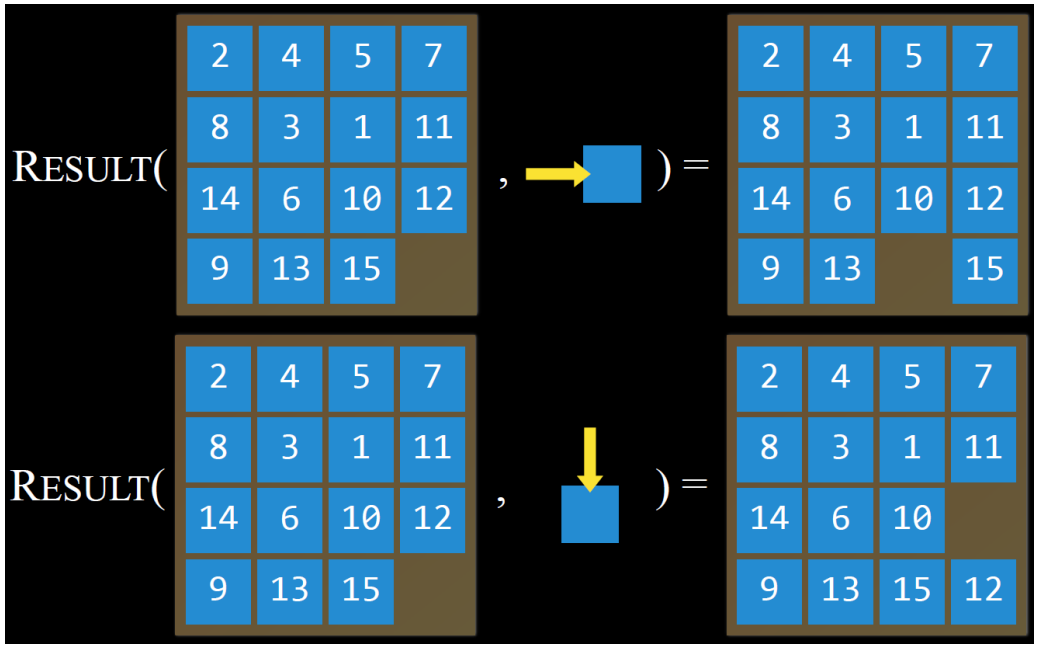


Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P83 – P88.

Key concepts: Transition model

Transition model:
maps a state and action to a resulting state

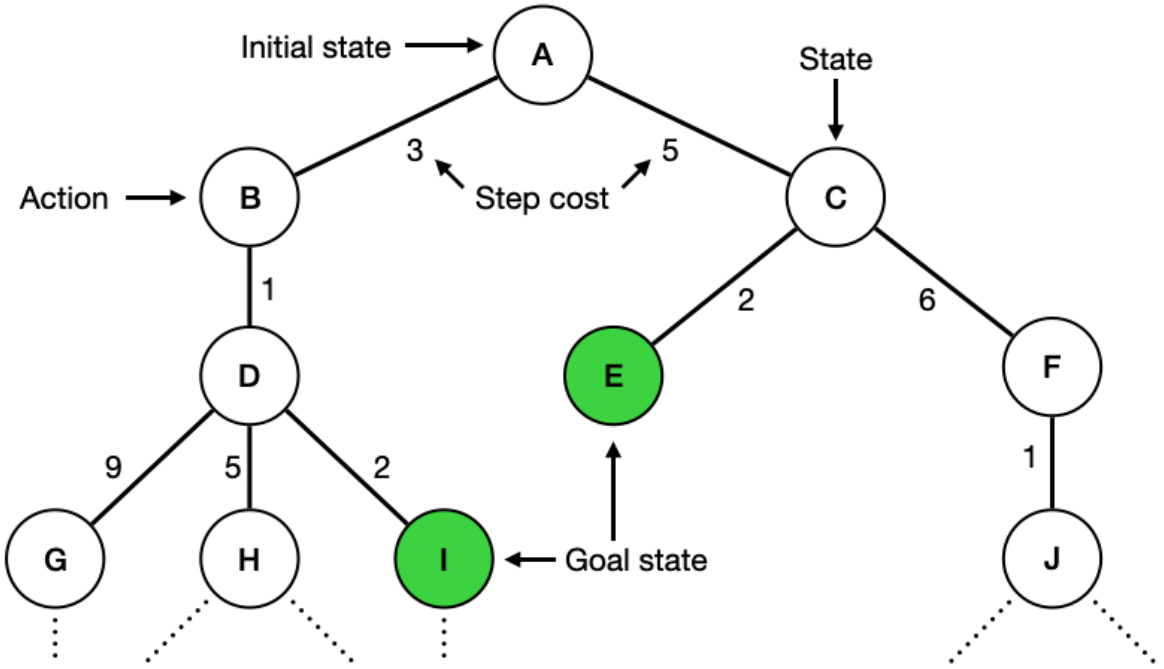
RESULTS(s, a) returns the state s', reached after performing action a in state s



Reference: https://pll.harvard.edu/course/cs50s-introduction-artificial-intelligence-python

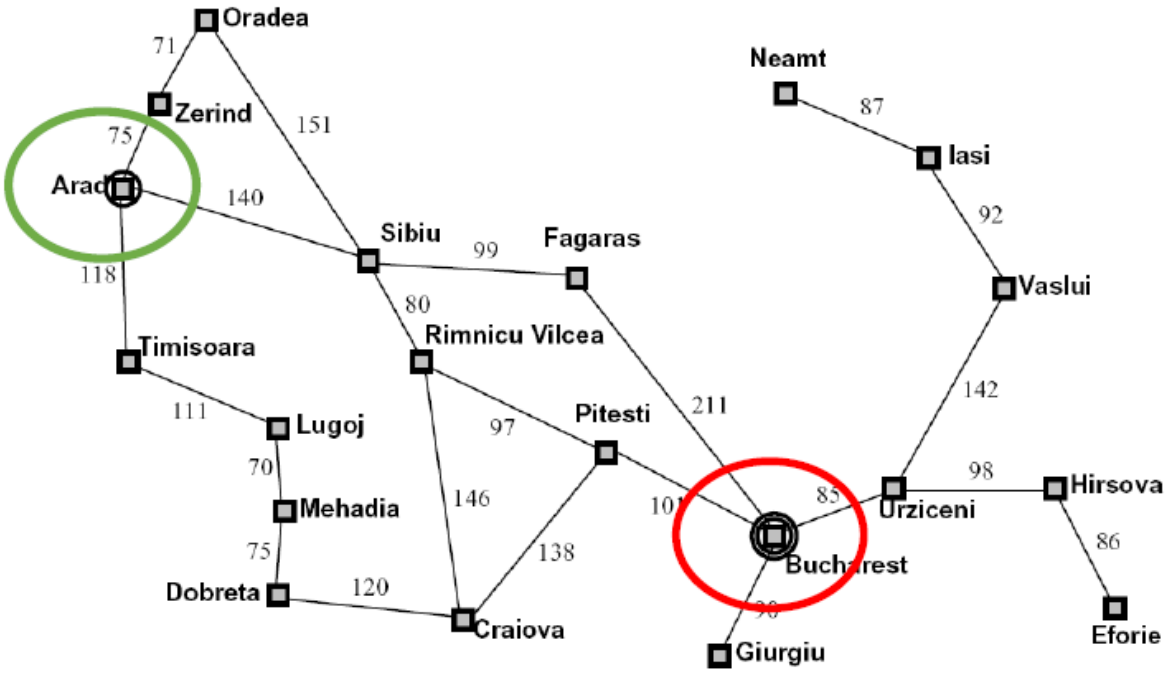
Key concepts: Action cost

Action cost/step cost: ACTION-COST(s, a, s') or $c(s, a, s')$
the numeric cost of applying action a in state s to reach state s'



Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P83 – P88

Example: Route finding as a search problem



State space:

- All the cities on the map

Actions:

- Traversing a road: Going to an adjacent city.

Action cost:

- Distance along each road

Start state:

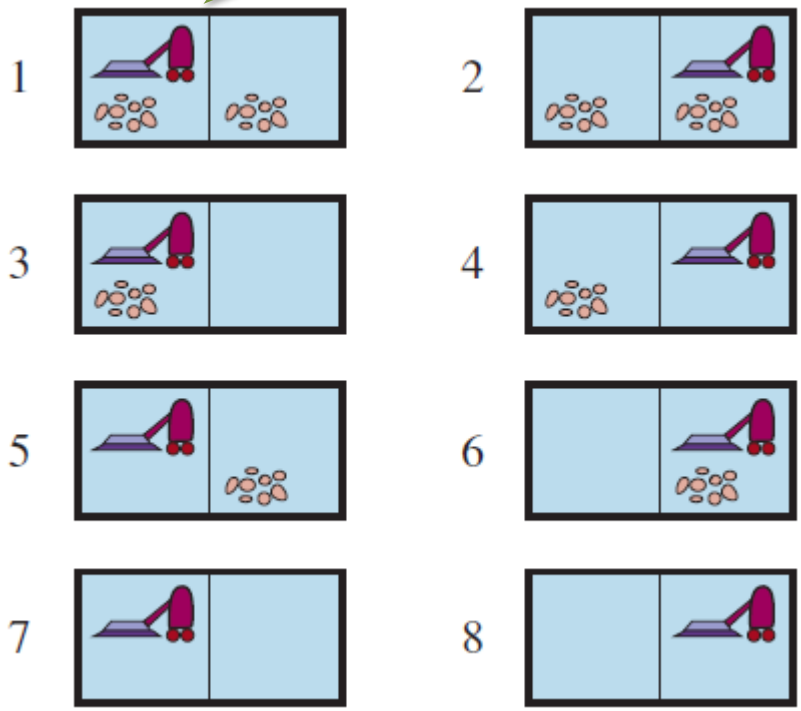
- Arad

Goal test:

- Is state == Bucharest?

Example: Vacuum world as a search problem

One state is designated as the initial state.



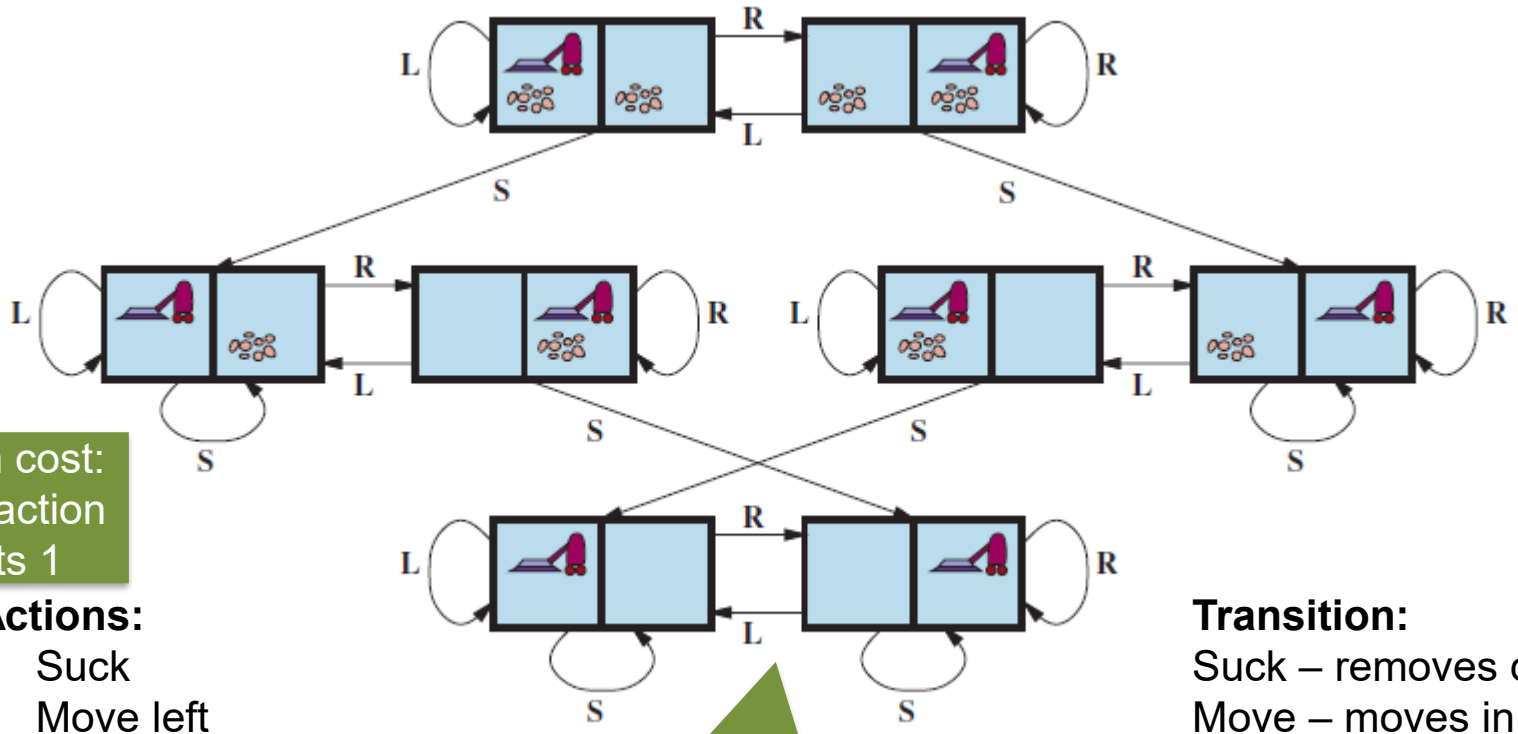
States: A state of the world says which objects are in which cells.

- the agent can be in either cell
- each cell can have dirt or not

2 cells * 2 positions for agent * 2 possibilities for dirt = 8 states.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P84 – P85.

Example: Vacuum world as a search problem



Action cost: Each action costs 1

Actions:

- Suck
- Move left
- Move right
- (Move up)
- (Move down)

Goal states: States where everything is clean

Transition:
Suck – removes dirt
Move – moves in that direction, unless agent hits a wall, in which case it stays put

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P84 – P85.

Quick question 1

The path cost depends only on the starting and ending states, AND the sequence of actions taken.

Which of the following statements about search problem formulation is true? Select all that apply.

a) The state space is the set of all states the agent plans to visit during the search.

b) The initial state may sometimes be the same as the goal state.

The transition model defines the successor state resulting from applying an action to a current state.

c) The transition model specifies, for each state and action, the resulting successor state(s).

d) The action cost only depends on the action taken and the resulting state.

e) A state is considered a goal state if the goal test for that state returns true.

Search algorithms: Search for a solution

Once the problem is formulated, we need to solve it.

A **solution** is a sequence of *actions* from the *initial state* to a *goal state*.

Optimal solution: A solution is *optimal* if no solution has a lower *path cost*.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P89 – P91.

Two fundamental search strategies

Tree Search

- Explores the state space as a tree
- Treats each occurrence of a state via a different path as a distinct node
- May generate redundant states

additional:graph can have cycles, trees are acyclic
graphs can be disconnected but trees nodes must be reachable from any other node
graph's edges may be directed or not, but tree follows heirarchical structure, from root down
graph can have any arrangements, but trees have exactly one unique path between any two nodes

Graph Search

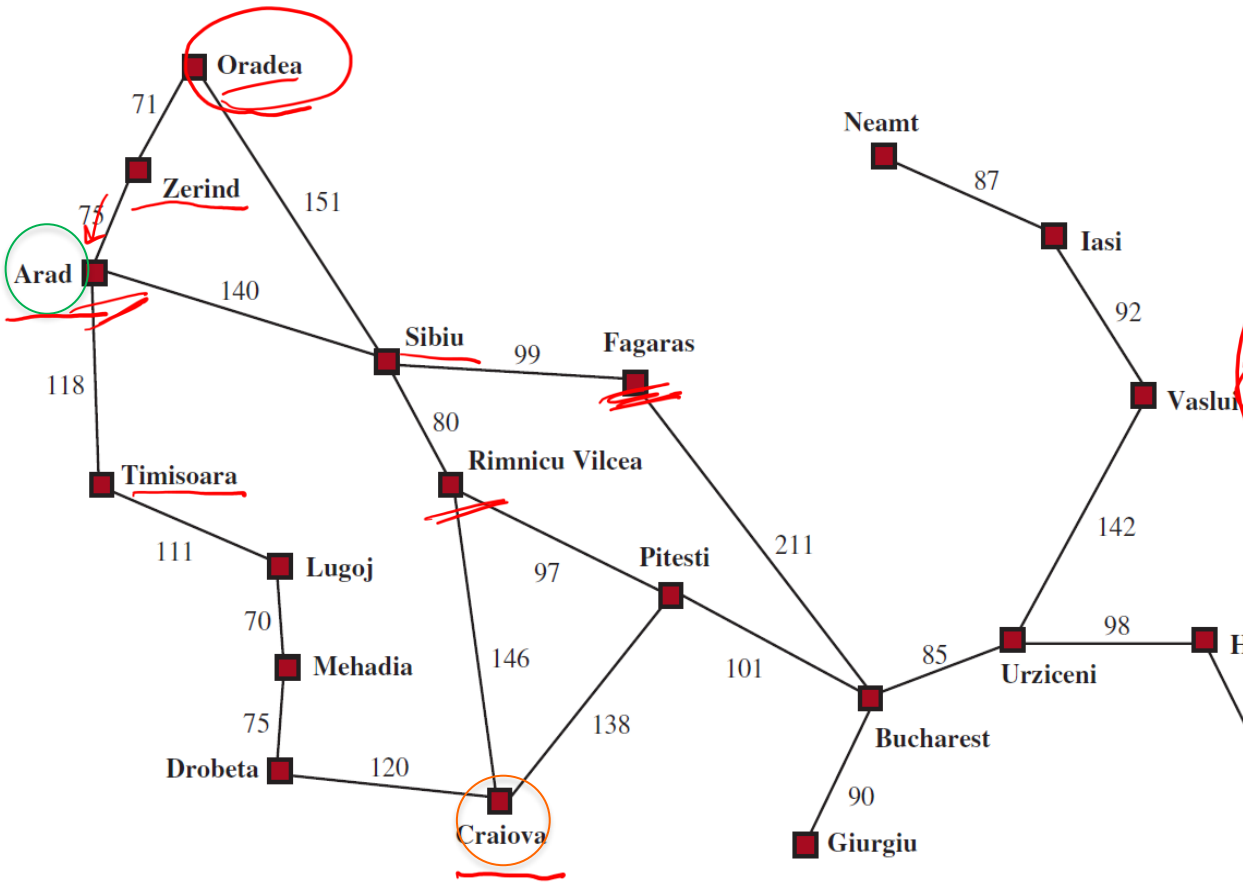
- Explores the state space as a graph
- Tracks visited states to avoid revisiting
- Prevents redundancy and loops

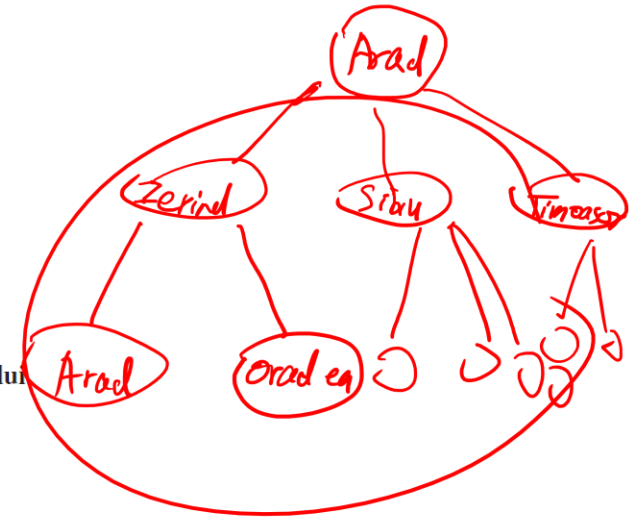
isVisited

Both are *search strategies*, and specific *search algorithms* are designed based on these strategies.

Reference: https://www.youtube.com/watch?v=Y1VywhuRFIs

Basic search strategy I: Tree search

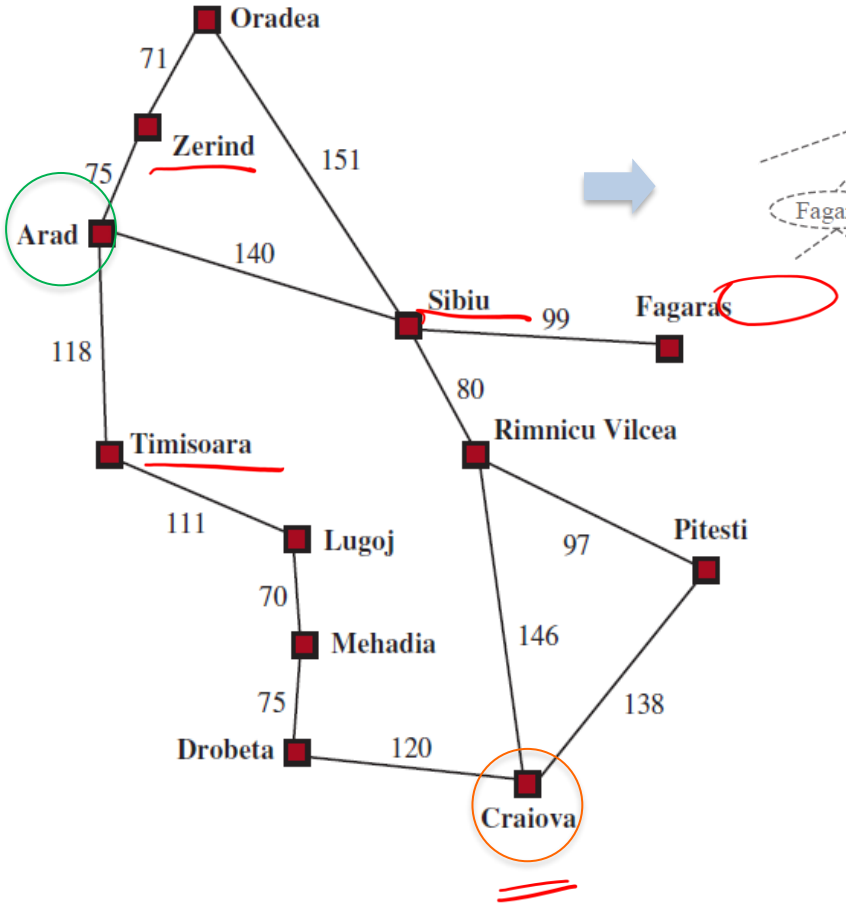


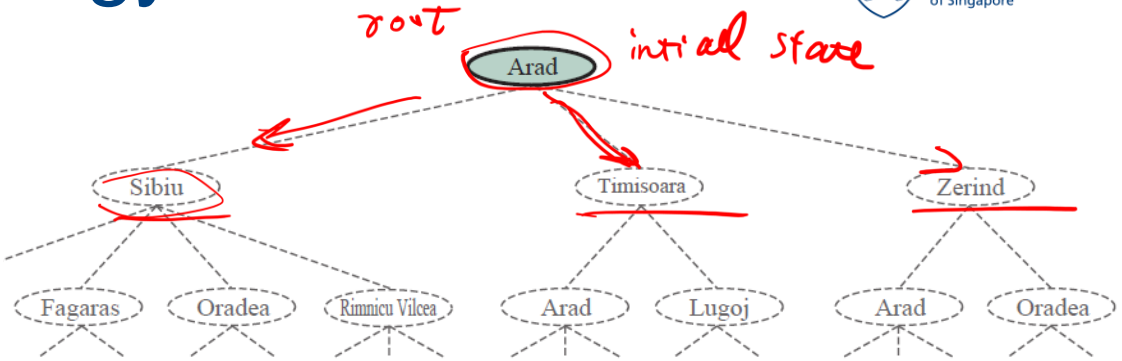


Arad is added again, a redundant state

Image source: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P82

Basic search strategy I: Tree search





A search tree: A conceptual tree representing possible plans and their outcomes---essentially a "what if" structure.

• The start state is the **root node**

At each step:

-- Check if the current node represents the goal state **(goal test)**.

-- If not, "expand" the node by applying all ***legal actions*** to the current state, generating new successor states (child nodes).

Basic search strategy I: Tree search

Tree Search:

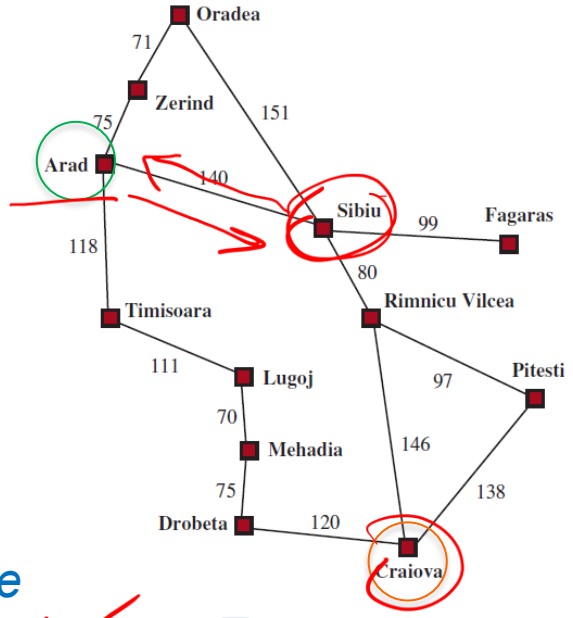
- Enumerates **all possible paths** from the initial state
- Explores states by generating an explicit search tree

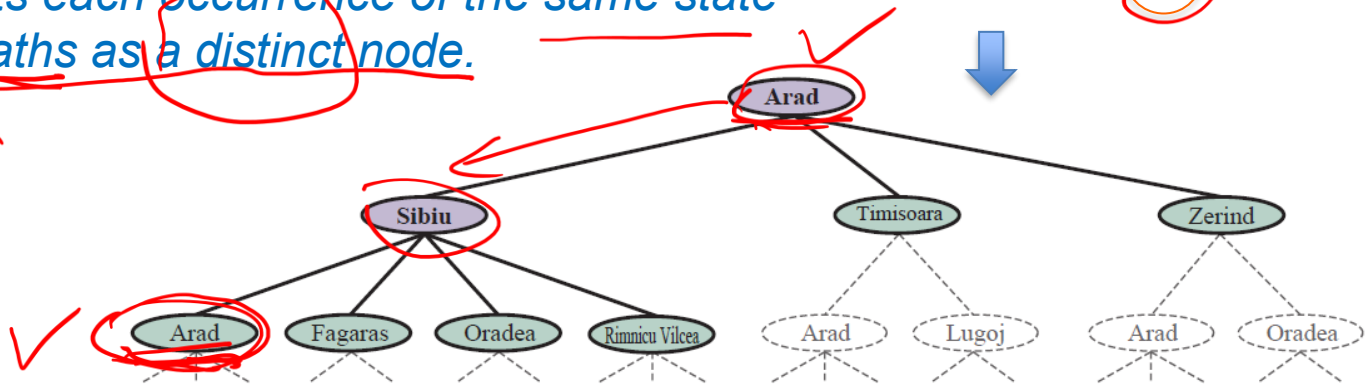
Search Tree Components:

- **Root**: Initial state
- **Nodes**: States generated via the transition model
- **Edges**: Actions connecting states

Key Point:

- *Tree search treats each occurrence of the same state along different paths as a distinct node.*





node can be the same content, but if they come from different paths, they are distinct instances (another node)

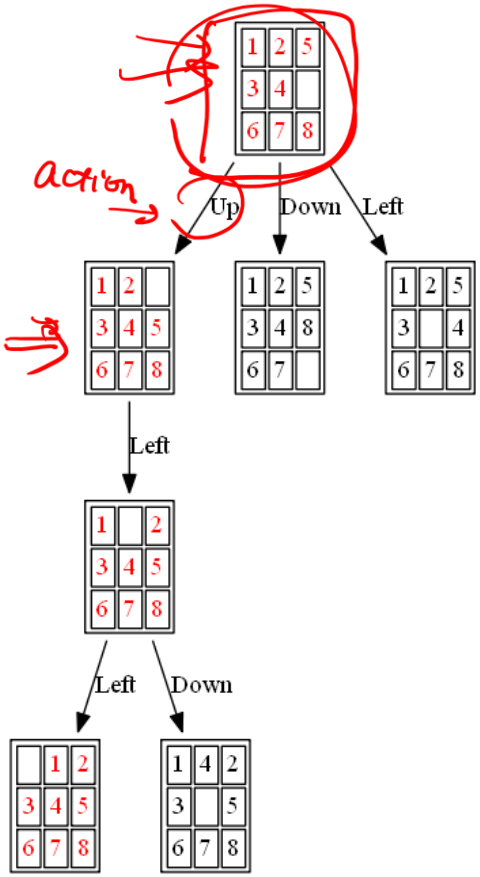
Tree search algorithm: Nodes and expansion

- ***Defining a search node:***

-- Each node contains a **state**

-- Node may also contains additional information:

  * the **parent node** (which generated this node)
  * an **action** (action applied to parent to get this node)
  * **path cost** (cost of the path from initial state to the node)



- ***Expanding a node:***

-- Applying all legal actions to the state contained in the node

-- Generating new nodes for all the corresponding successor states

Tree search algorithm: Node management

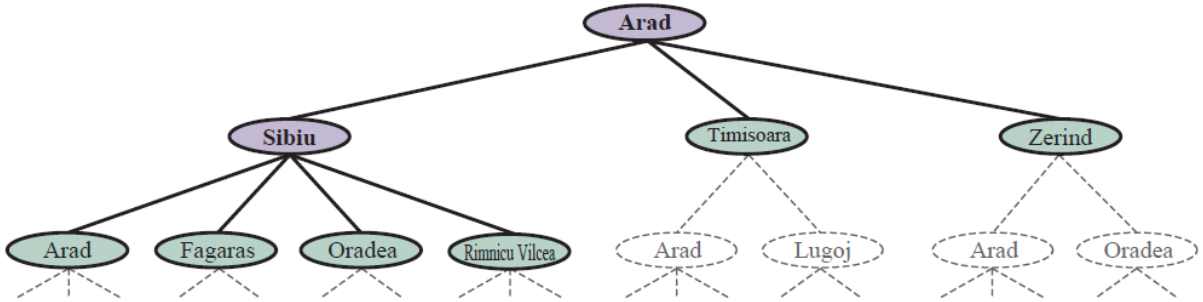
Managing generated nodes

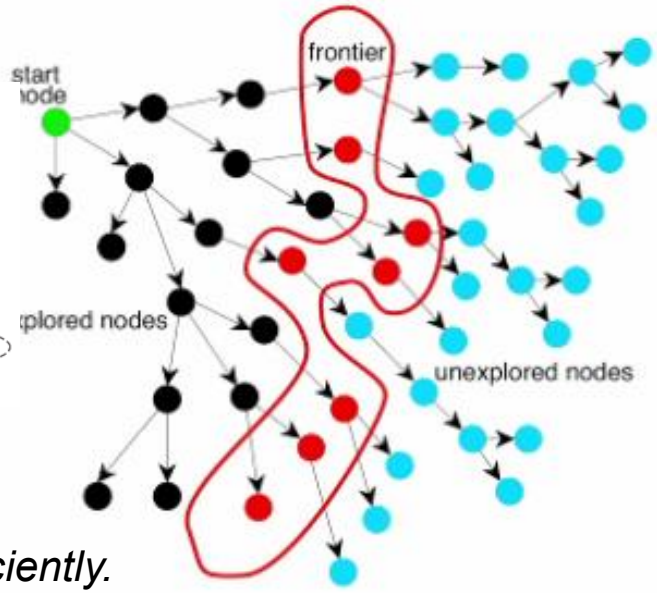
- Need: A data structure* to store nodes as they are generated

here is talking about priority queue or queue ADT, which is FIFO. If we talk about actual DS then it can be using Heap data structure (smallest weight as root, into a complete binary tree - every level is full, except the last level, and as far left as possible

Frontier

- Queue (or other structure) of nodes **waiting to be expanded**
- **Stores state and path information**, needed to reconstruct the solution once the goal is reached

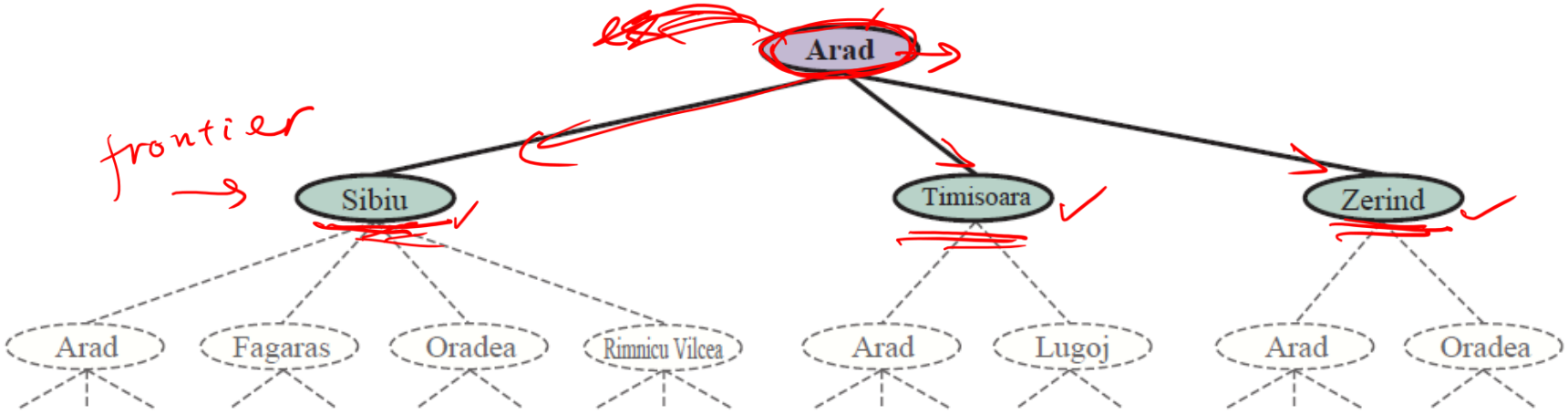




** A data structure is a way to organize and store data in a computer so it can be accessed, updated, or searched efficiently.*

Reference: https://www.youtube.com/watch?v=xoqcnWXTXcI

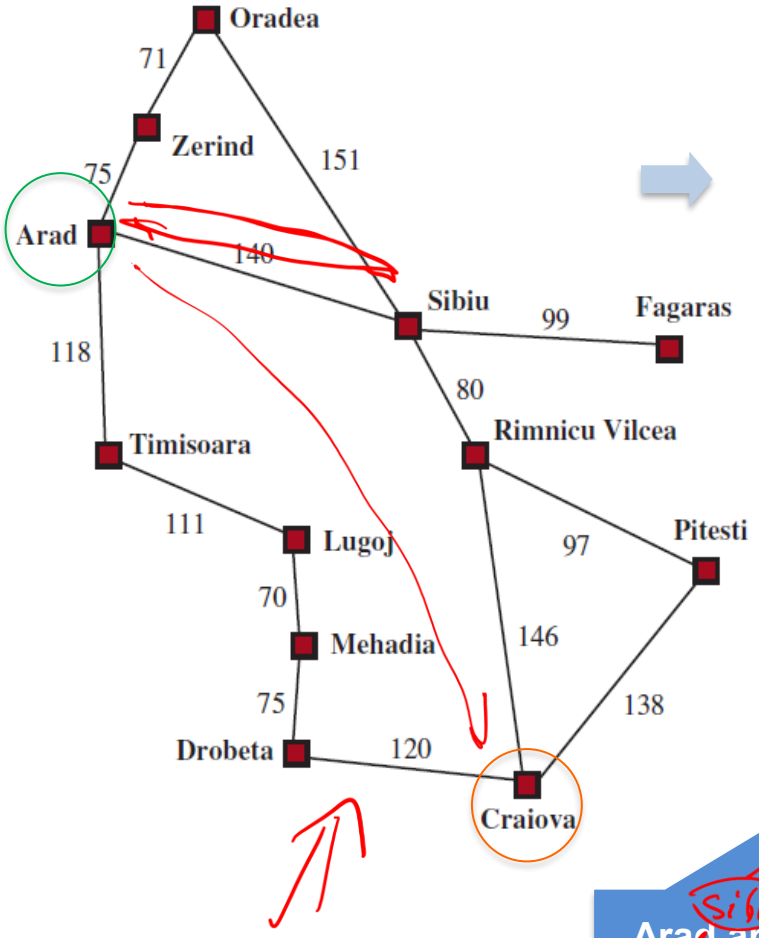
General tree search algorithm



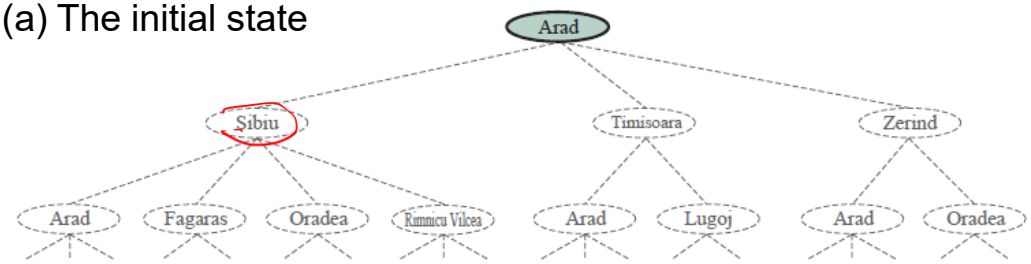
function TREE-SEARCH (*problem, strategy*) return a solution, or failure
    Initialize frontier to the initial state of the *problem*
    loop do
        if the frontier is empty then return failure
        choose a node from the frontier for expansion according to *strategy*, *and* remove it from frontier
        if the node contains goal state then return the corresponding solution
        else expand the node, and add the resulting child nodes to the frontier
    end

The strategy determines search process!

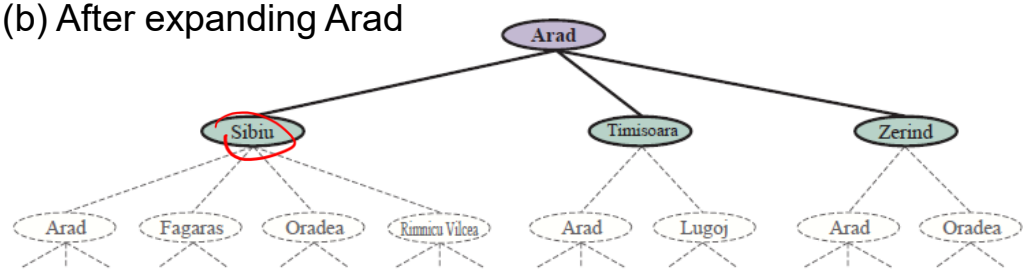
Problem of tree search: Repeated states



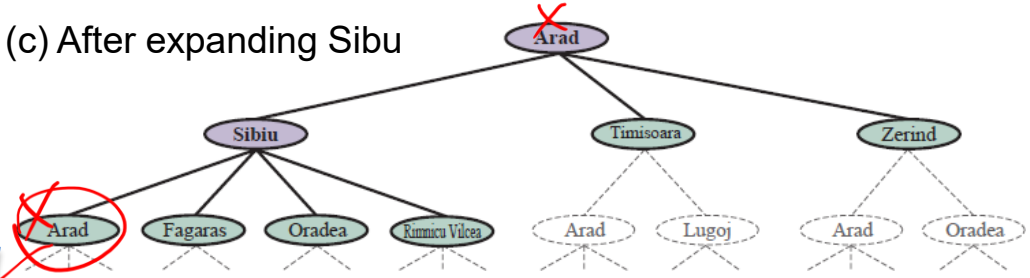
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibu



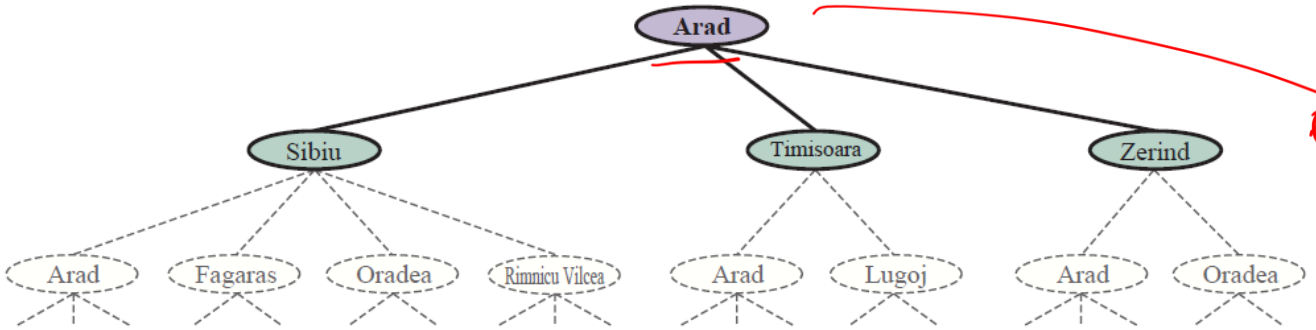
Arad appears again!

Loops cause redundancy because *path costs are additive*

Solution to repeated states

To avoid redundancy and inefficiency caused by revisiting the same states, we augment the basic tree search with one additional data structure:

Explored set: **A record of all previously visited nodes (states).**
Helps prevent revisiting the same state, reducing loops and redundancy.



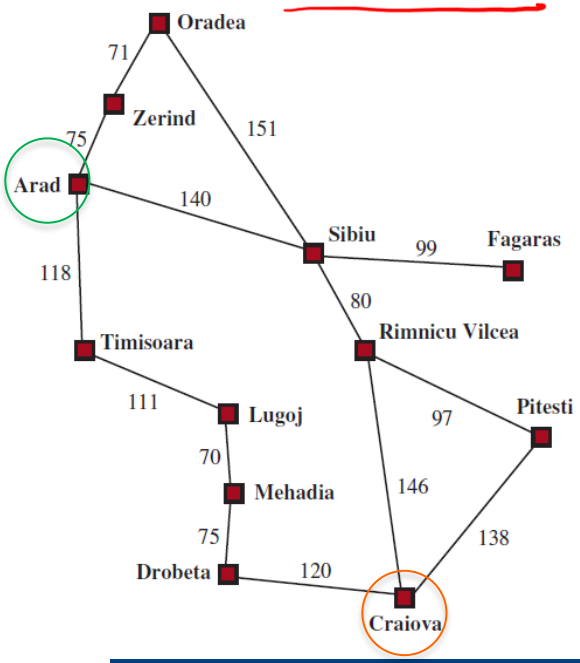
Lavender: Nodes that have been expanded (in the **explored set**)
Green: nodes currently on the **frontier**, waiting to be expanded.

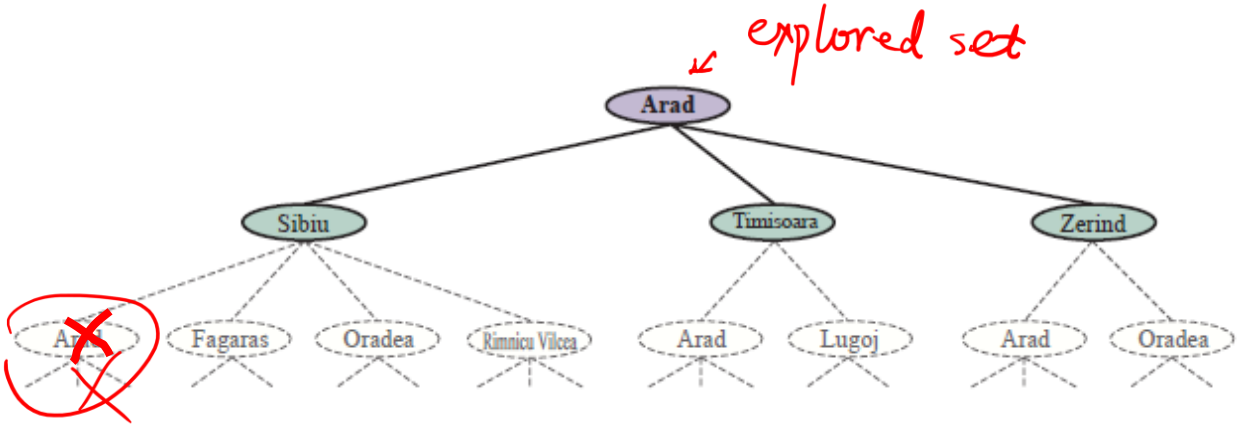
Image source: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P89

Basic search strategy II: Graph search

Graph Search: Simple modification of tree search

- Before adding a node to the frontier, **check if its state has been visited**
- Requires keeping track of all visited states (can use significant memory)
  e.g.: 8-puzzle has 9!/2 ≈ 182,000 possible states
- Although we are exploring a graph, the *expansion process* *still forms a tree-like structure*



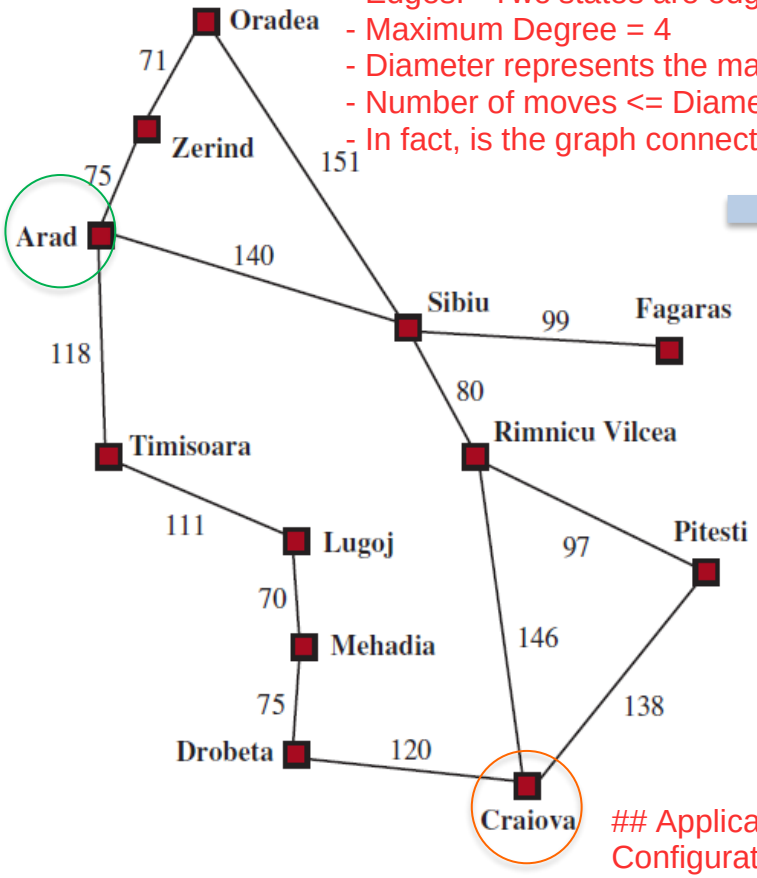


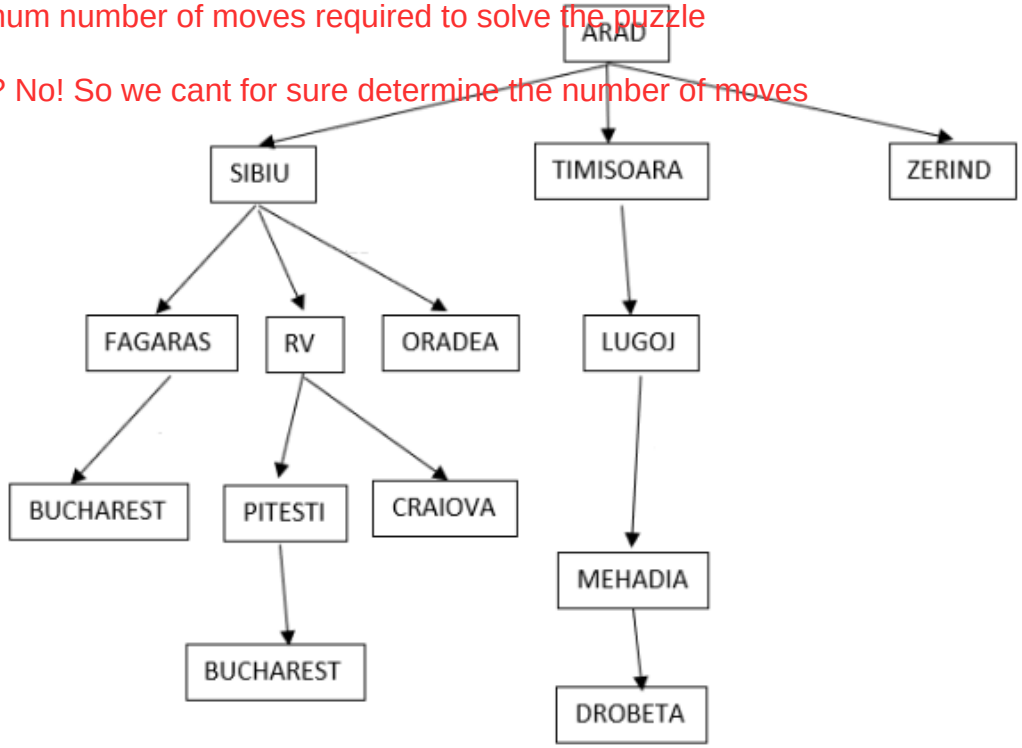
Reference: https://www.youtube.com/watch?v=Y1VywhuRFIs

Basic search strategy II: Graph search

Application: Sliding Puzzle is a Graph

- Each vertex is a state and each edge represents the possible connnection to another state after one action
  - Nodes:– State of the puzzle– Permutation of nine tiles = 9! = 362,880
  - Edges:– Two states are edges if they differ by only one move < 4 * 9! < 1,451,520
  - Maximum Degree = 4
  - Diameter represents the maximum number of moves required to solve the puzzle
  - Number of moves <= Diameter
  - In fact, is the graph connected? No! So we cant for sure determine the number of moves





Application: 2x2x2 Rubik's Cube

Configuration Graph

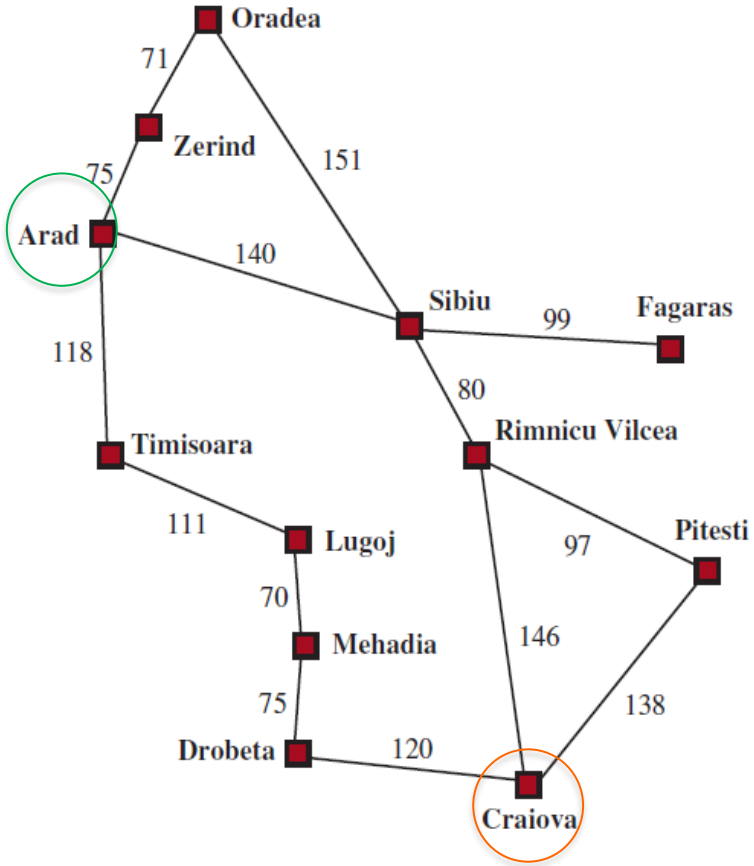
- Vertex for each possible state
- Edge for each basic move– 90 degree turn– 180 degree turn
- Puzzle: given initial state, find a path to the solved state.
- How many vertices? 8! · 3^8 = 264,539,520 (8 is # cubelets Each cubelet is in one of 8 positions. 3^8 is Each of the 8 cubelets can be in one of three orientations) but since it is symmetric, we can 7! · 3^7 = 11,022,480

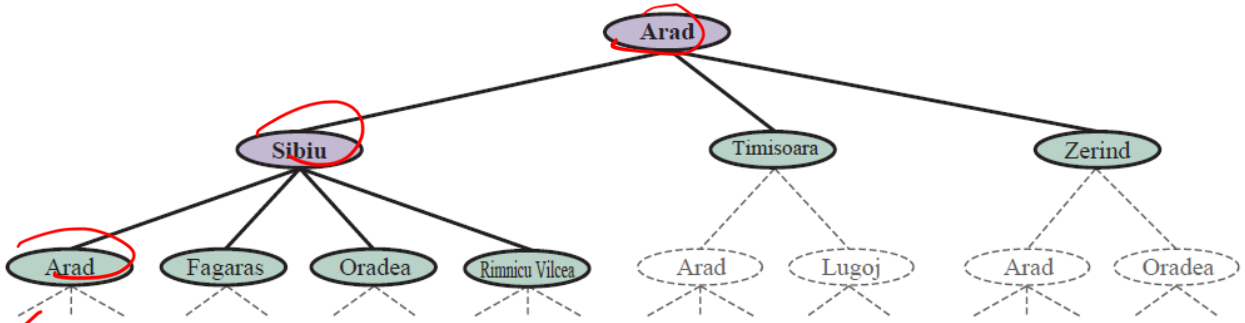
Quick question 2

True or false: In a finite state space, graph search with an explored set is guaranteed to terminate, whereas tree search may revisit the same states indefinitely.

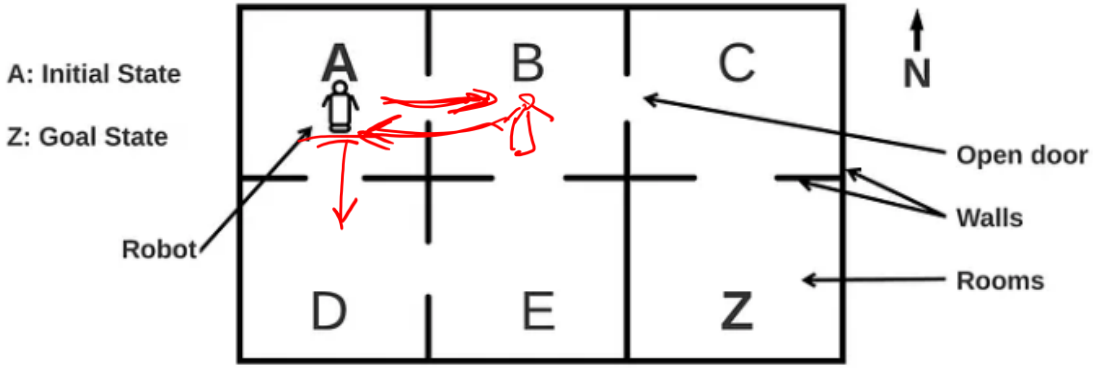
- True
- False

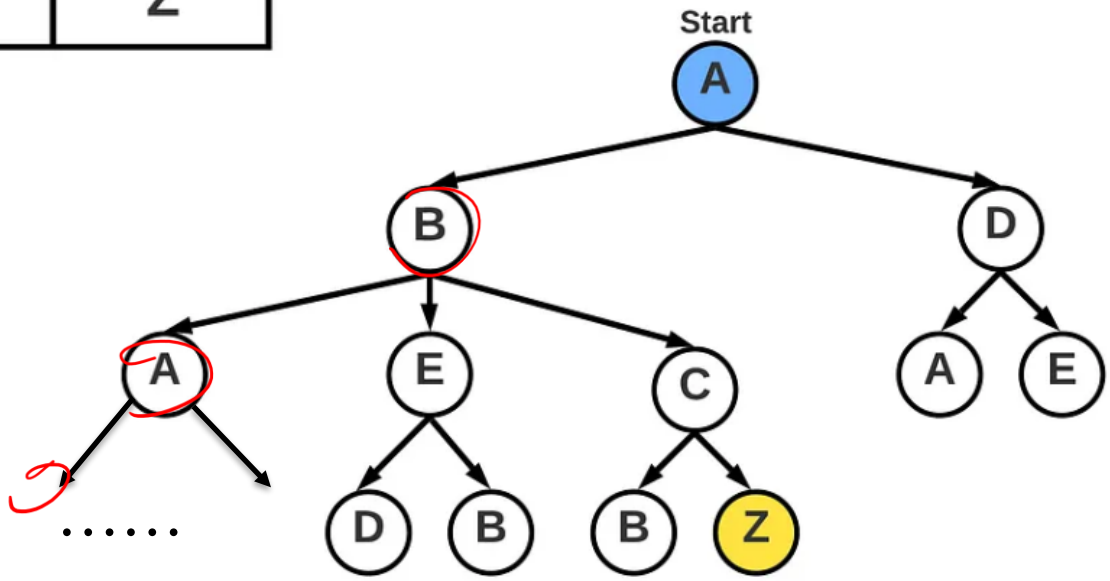
Quick question 2





Quick question 2





Credit: https://medium.com/nerd-for-tech/ai-search-algorithms-with-examples-54772c6d973a

Graph search vs Tree search

Tree search

```
function TREE-SEARCH(problem, strategy) return a solution, or failure
    Initialize frontier to the initial state of the problem
    loop do
        if the frontier is empty then return failure
        choose a node from the frontier for expansion according to strategy,
        and remove it from frontier
     if the node contains goal state then return the corresponding solution
        else expand the node, and add the resulting child nodes to the frontier
     end
```

Graph search

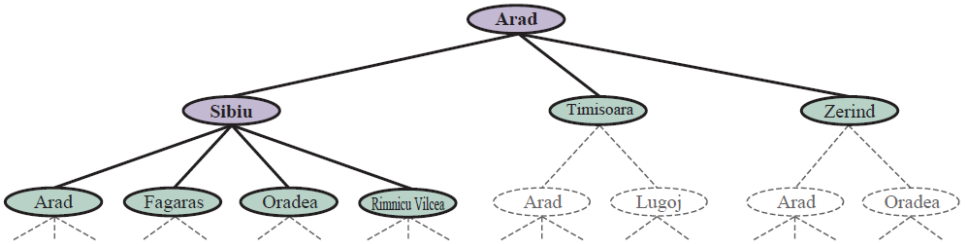
```
function GRAPH-SEARCH(problem, strategy) return a solution, or failure
    Initialize frontier to the initial state of the problem
    Initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a node from the frontier for expansion according to strategy,
        and remove it from frontier
     if the node contains goal state then return the corresponding solution
     add the node to the explored set
     expand the node, and add the resulting nodes to the frontier
        only if not in the frontier or explored set
     end
```

The strategy leads to a variety of search algorithms!

Newly generated nodes that match the frontier nodes or explored nodes are *discarded*.

Reference: https://www.youtube.com/watch?v=Y1VywhuRFIs

Q1. Which of the following statements about tree search and graph search are correct? Select all that apply.



A. If a state is reachable via multiple paths, tree search will create two separate nodes corresponding to each path ✓

B. Tree search checks if a node has been visited before adding it to the search queue. Graph search!

C. Tree search is more efficient than graph search. Tree search is less efficient as it allows revisiting the same nodes.

D. Graph search checks if a node has been visited before adding it to the search queue. ✓

Summary: Tree search vs graph search

graphs are not meant to be drawn!

Aspect	Tree search	Graph search
Explored set	No (no memory of visited states)	Yes (maintains a set of visited states)
Memory use	Low	High (stores all visited states) -ve
Time efficiency	Low (may revisit states many times)	High (avoids revisiting) +ve
Risk of infinite loops	High (esp. in graphs with cycles)	Low (prevents loops via state tracking)
Implementation complexity	Simpler	More complex
When to use	- Memory-constraint systems - Cycles are impossible or negligible.	- Cyclic or redundant state spaces - Sufficient memory is available
Typical use cases	Scenarios with huge state spaces and no cycles, memory-limited systems	Most real-world problems with state reuse (e.g., pathfinding, games and puzzles)

Node exploration strategies: Uninformed vs informed search

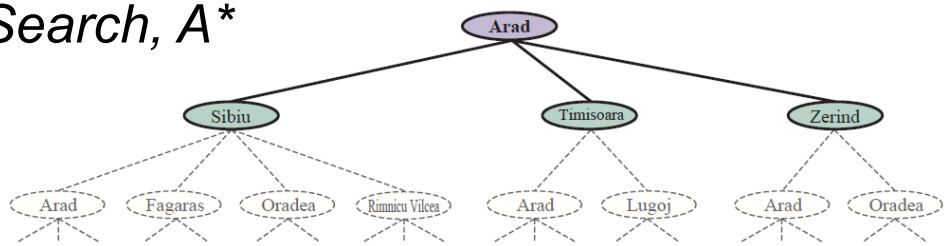
The **node exploration strategy** leads to a variety of search algorithms.

Uninformed search:

- Explores the search space **without additional knowledge** (e.g., the goal's location).
- Examples: *Breadth-First Search (BFS), Depth-First Search (DFS), Uniform-Cost Search/Dijkstra's algorithm*

Informed search:

- Uses **domain knowledge** (*heuristics*, e.g., an estimate of the distance to the goal) to guide the search towards the goal.
- Examples: *Greedy Best-First Search, A**



Evaluation of search algorithms

We can evaluate an algorithm's performance in four ways:

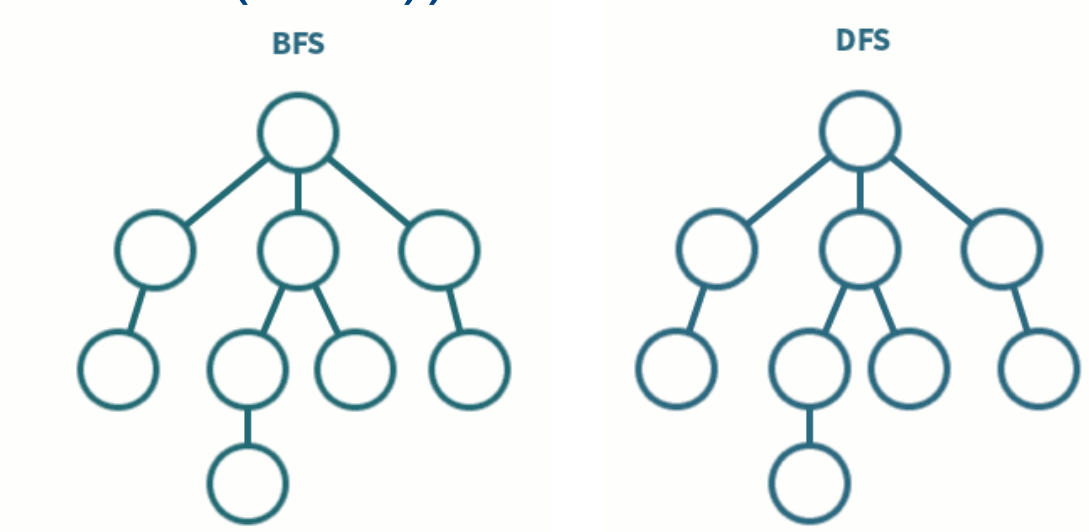
- **Completeness**: Does the algorithm always find a solution if one exists?
- **Optimality**: Does it find the lowest-cost (best) solution among all possible ones?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

These criteria apply to evaluating *general problem-solving algorithms.*
In our course, we will focus on *completeness* and *optimality*.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P93.

Agenda

- Background: Why search matters
- Search problem formulation
- Uninformed search (breadth-first search (BFS), depth-first search (DFS))

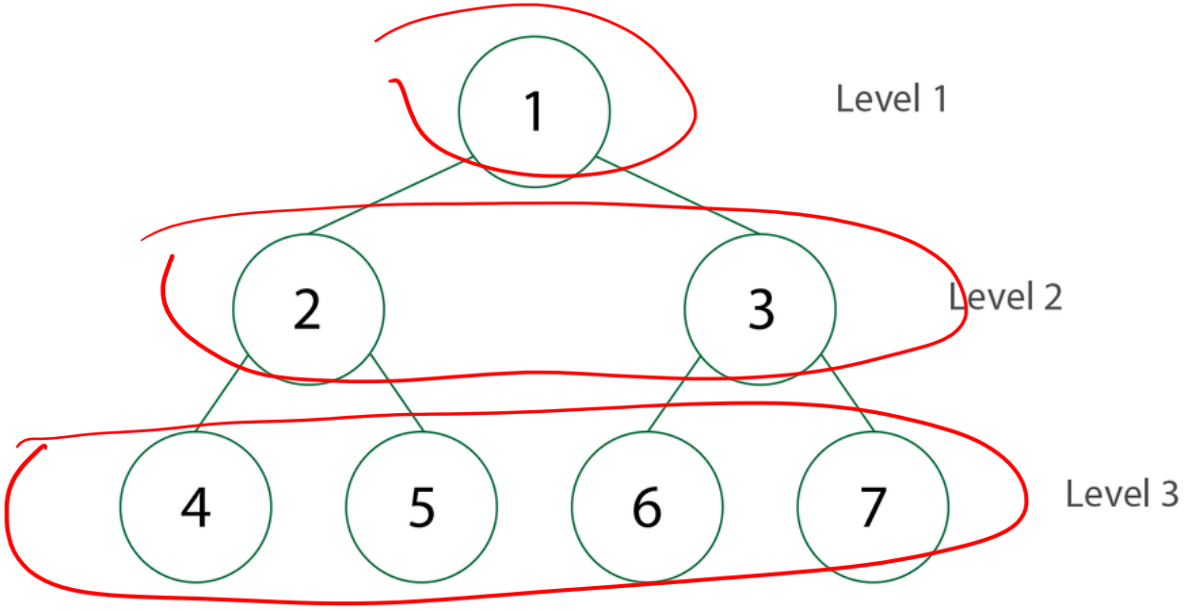


Breadth-first search (BFS)

If using graph search strategy

Strategy: Expand the shallowest unexplored node first.

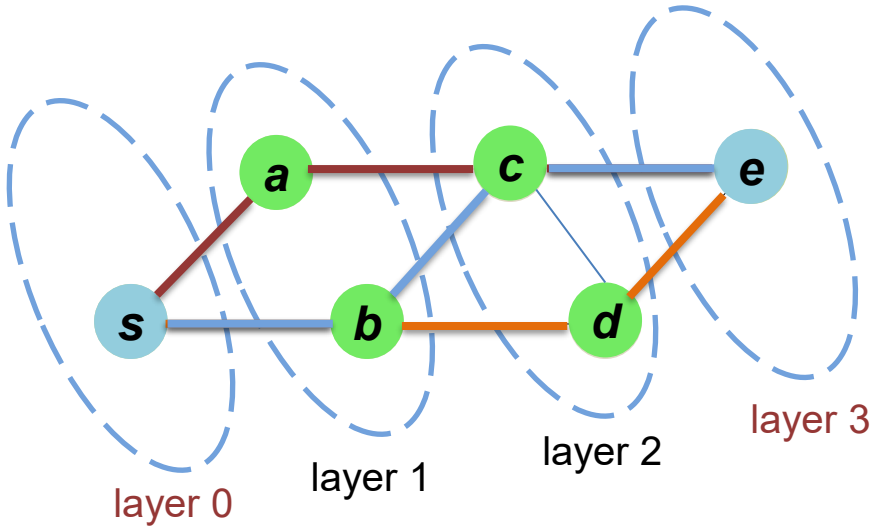
All the successors of a node are expanded, then their successors and so on.



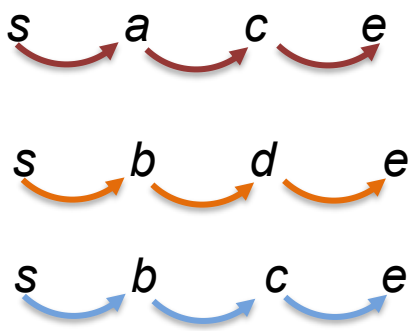
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P94 – P95.

Breadth-first search (BFS)

- Explore nodes in “layers” (level by level)
- Can find everything reachable from a given starting node
- Can compute minimum hops between two nodes (i.e., shortest path, if all steps cost the same)



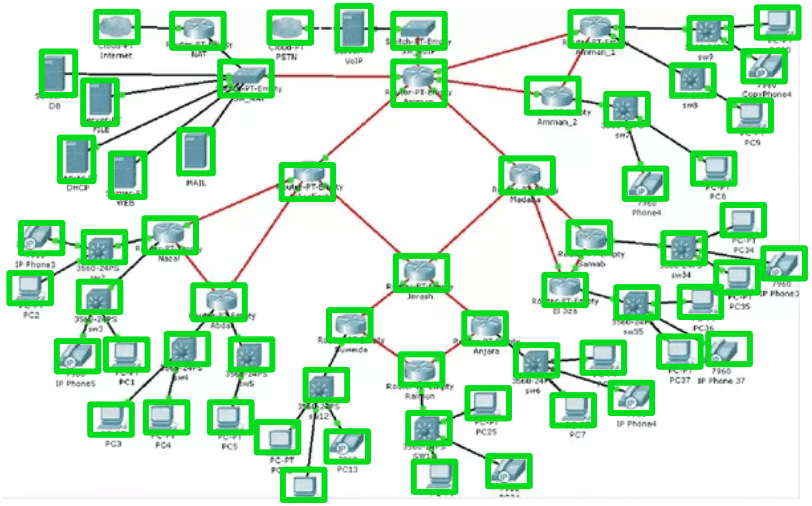
s → *e*: minimum 3 hops



Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P94 – P95.

BFS application: Ensure a thorough and orderly system recovery

(Major global IT outage hits airlines on Jul 19, 2024, photo credit: Lim Yaohui/The Straits Times)

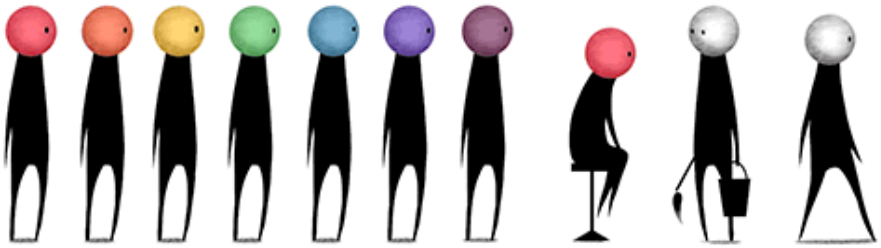


(Illustration of a computer network, picture credit: Sulav Paudel/AgResearch)

BFS: Frontier as a FIFO queue

- BFS always expands the *shallowest node* in the frontier
- In practice, the **frontier** in BFS is implemented as a **queue** (FIFO) to ensure nodes are expanded in the order they are discovered

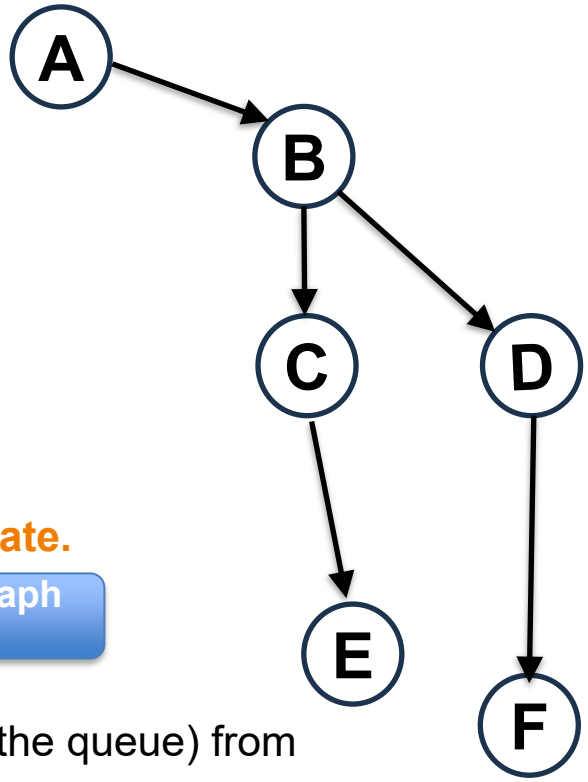
Frontier: A queue (or other data structure) of nodes waiting to be expanded.



Queue: A First-In-First-Out (FIFO) data structure, where the first element added is the first to be removed.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P94 – P95.

Finding a path from A to E using BFS

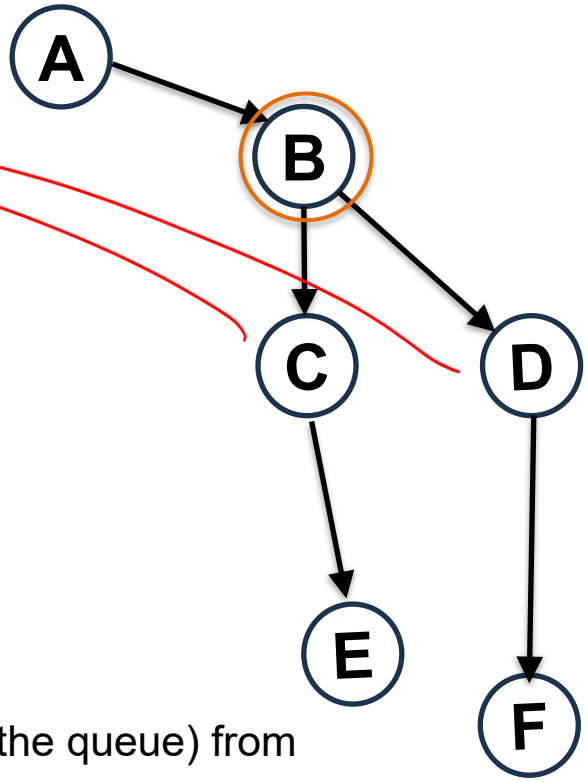


- **Start with a frontier (queue) that contains the initial state.**
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Dequeue the shallowest node (i.e., the first node in the queue) from the frontier.
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- Expand the node, for each resulting child node, if it's not in the frontier or explored set, enqueue it into the frontier

Implemented with graph search strategy

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P95.

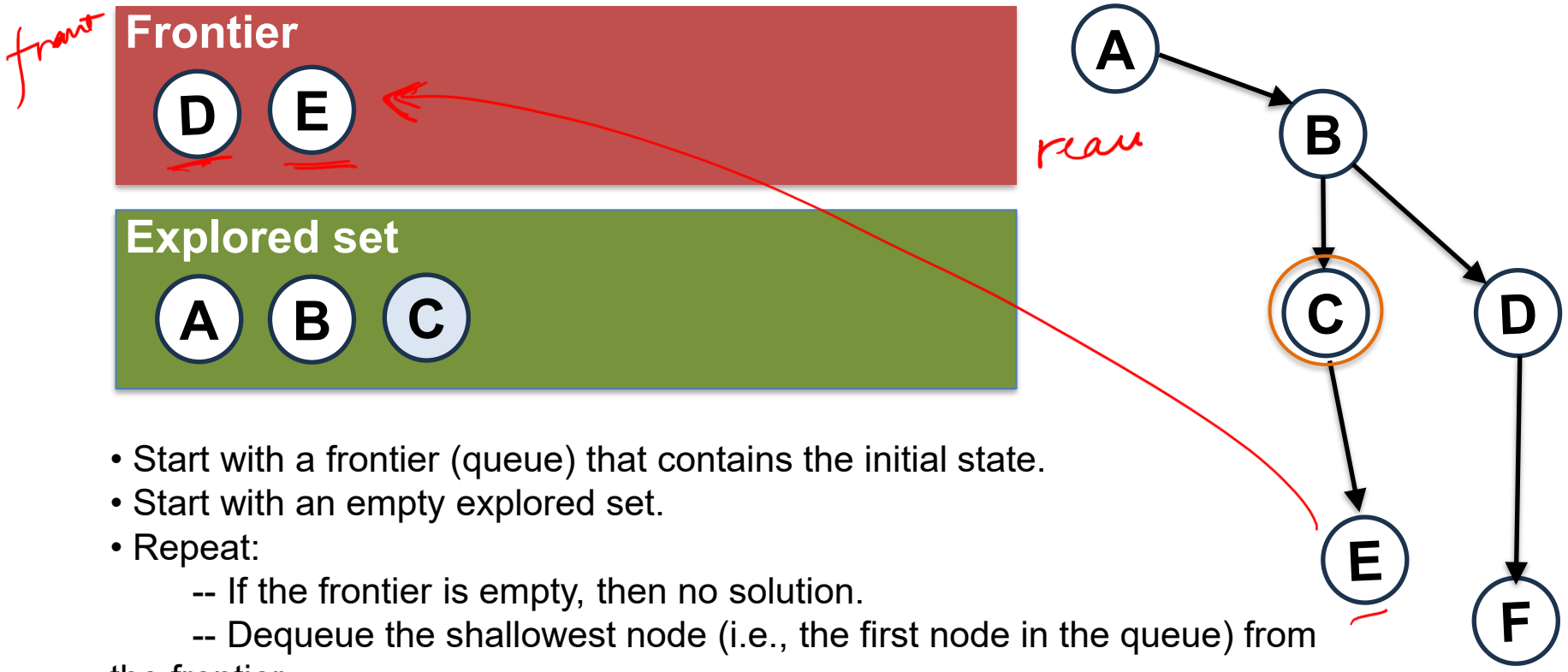
Finding a path from A to E using BFS



- Start with a frontier (queue) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Dequeue the shallowest node (i.e., the first node in the queue) from the frontier.
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- **Expand the node, for each resulting child node, if it's not in the frontier or explored set, enqueue it into the frontier**

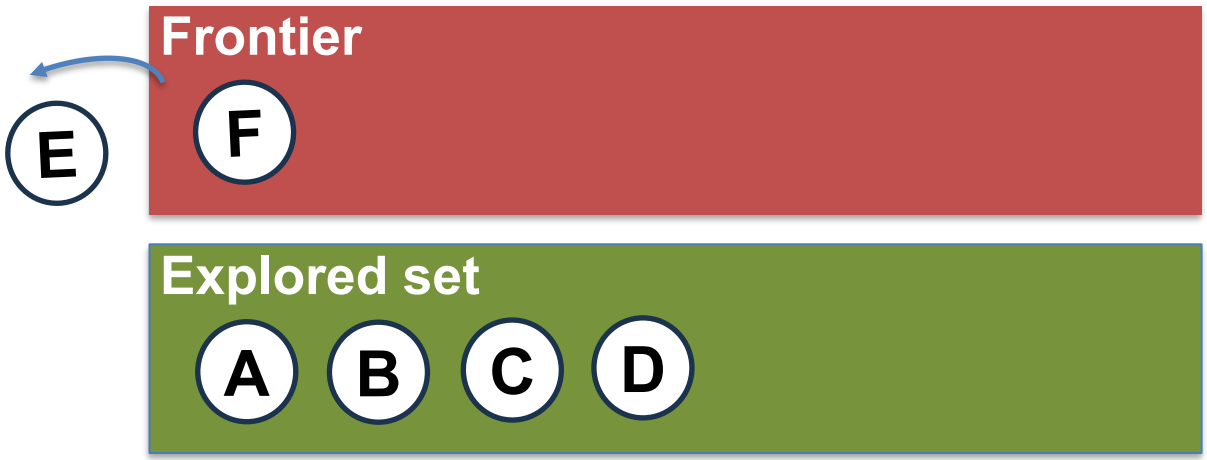
In this example, it doesn't matter whether we enqueued C or D first

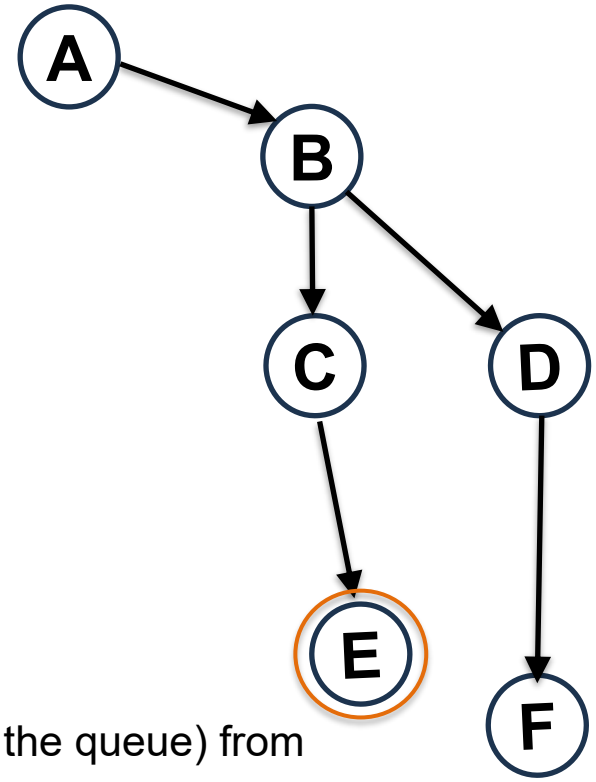
Finding a path from A to E using BFS



- Start with a frontier (queue) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Dequeue the shallowest node (i.e., the first node in the queue) from the frontier.
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- **Expand the node, for each resulting child node, if it's not in the frontier or explored set, enqueue it into the frontier**

Finding a path from A to E using BFS



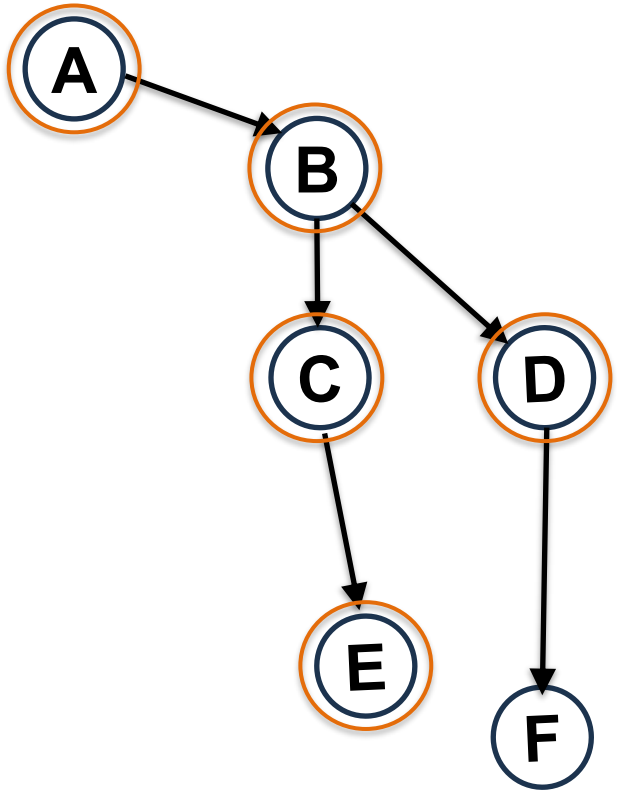


- Start with a frontier (queue) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Dequeue the shallowest node (i.e., the first node in the queue) from the frontier.
  - -- **If this node contains the goal state, return the solution.**
  - -- Add the node to the explored set.
  - -- Expand the node, for each resulting child node, if it's not in the frontier or explored set, enqueue it into the frontier

Each node stores information about its parent (i.e., the node that generated it.)

Finding a path from A to E using BFS

- Explore nodes in “layers” (level by level)
- Can compute minimum hops between two nodes (i.e., shortest path, if all steps cost the same) ONLY FOR UNWEIGHTED
- Can find everything reachable from a given starting node (without a goal test)



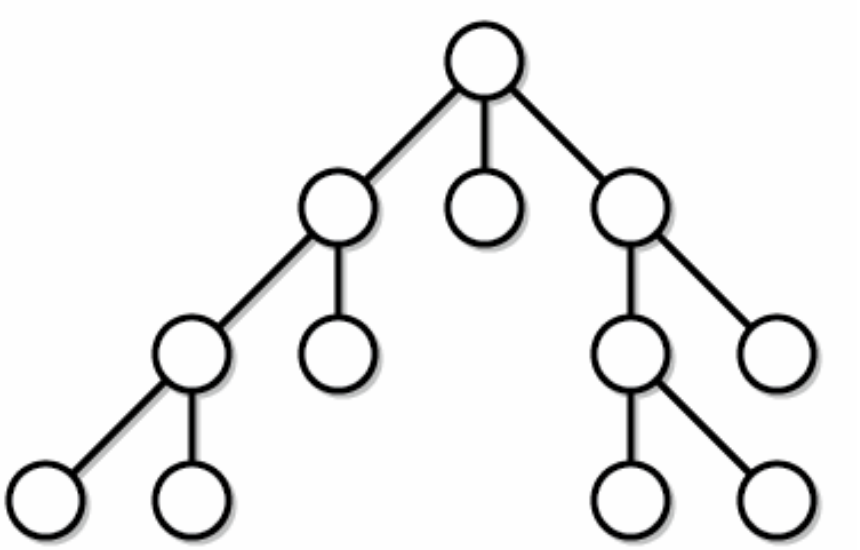
Depth-first search (DFS)

If using graph search strategy

Strategy: Expand the deepest unexplored node in the frontier first.

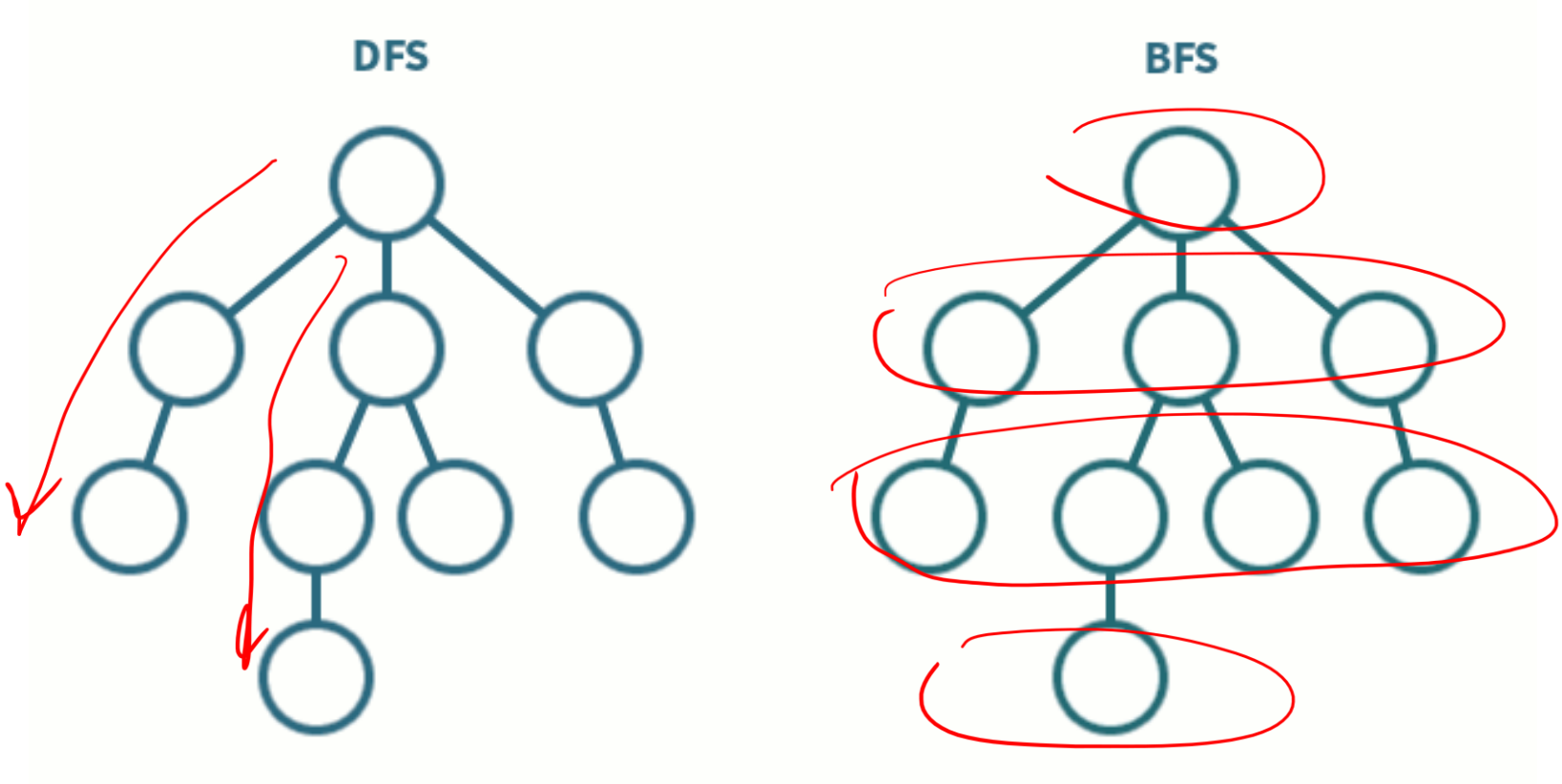
Explore one path as far as it can go before backtracking.

All the successors of a node are expanded **before** moving to other siblings.



Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P96 - P99.

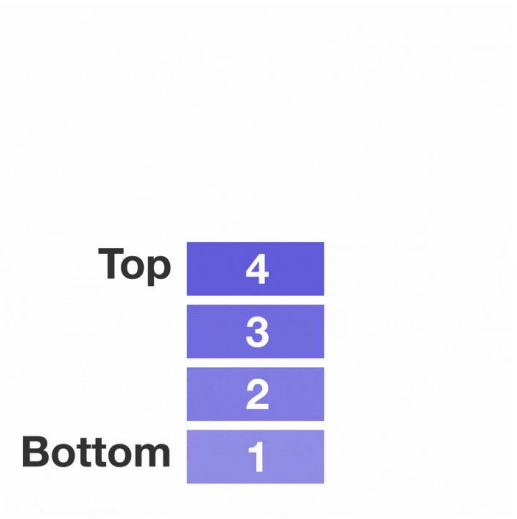
Traversal strategies: DFS vs BFS



Source: https://medium.com/analytics-vidhya/a-quick-explanation-of-dfs-bfs-depth-first-search-breadth-first-search-b9ef4caf952c

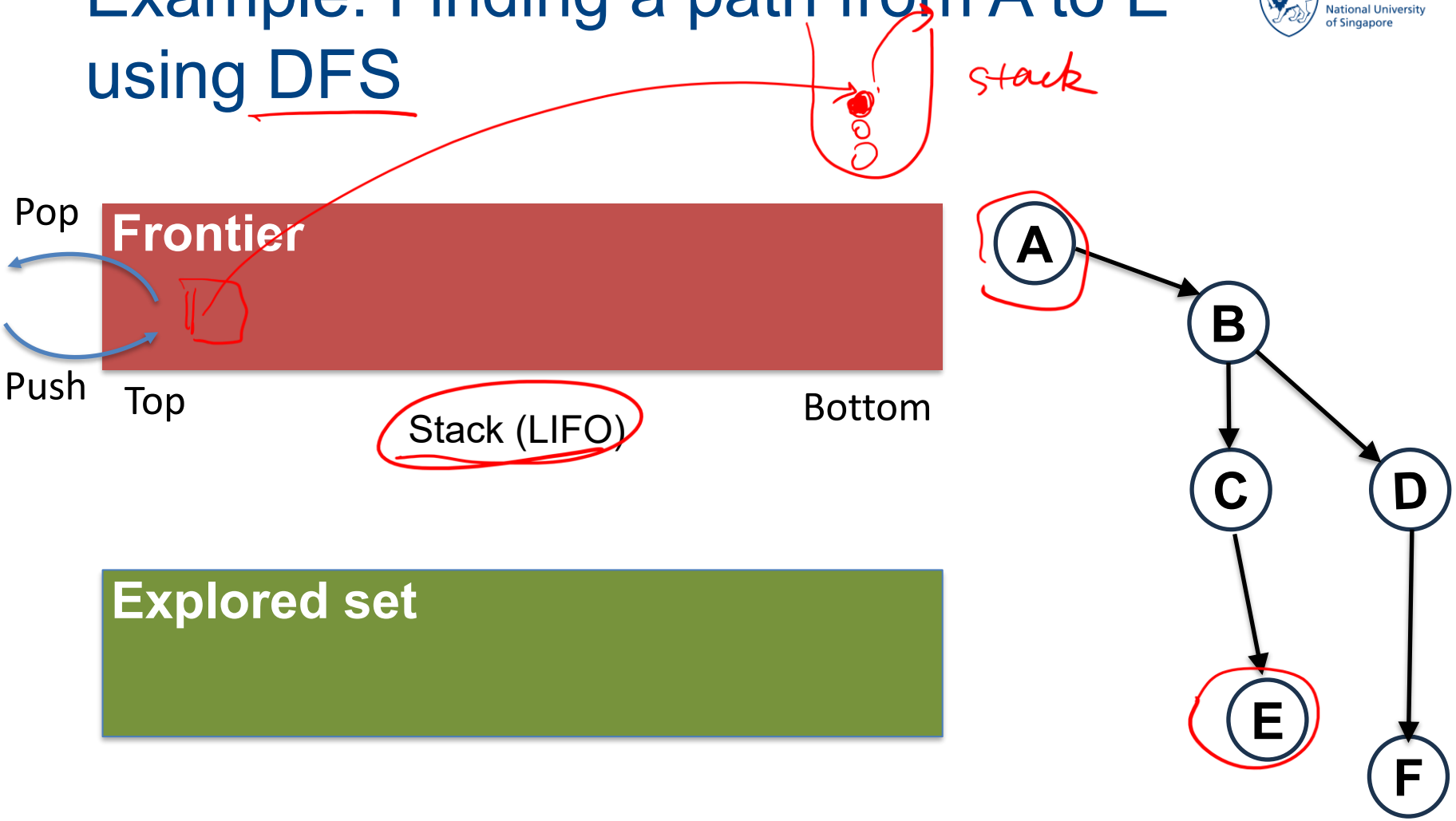
DFS: Using a LIFO stack for the Frontier

- DFS always expands the *deepest node* in the frontier
- In practice, the **frontier** is implemented as a **stack** (LIFO) to explore one branch fully before backtracking

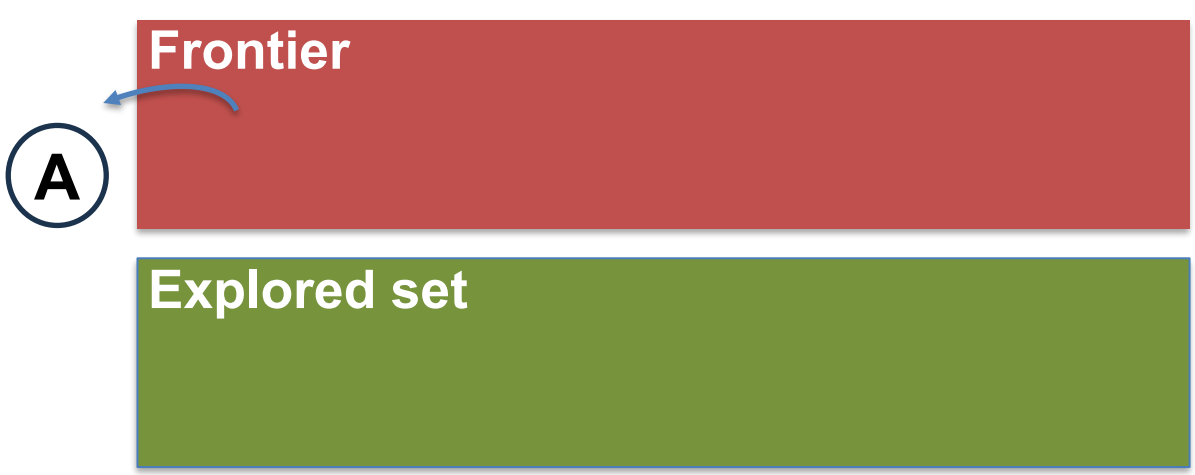


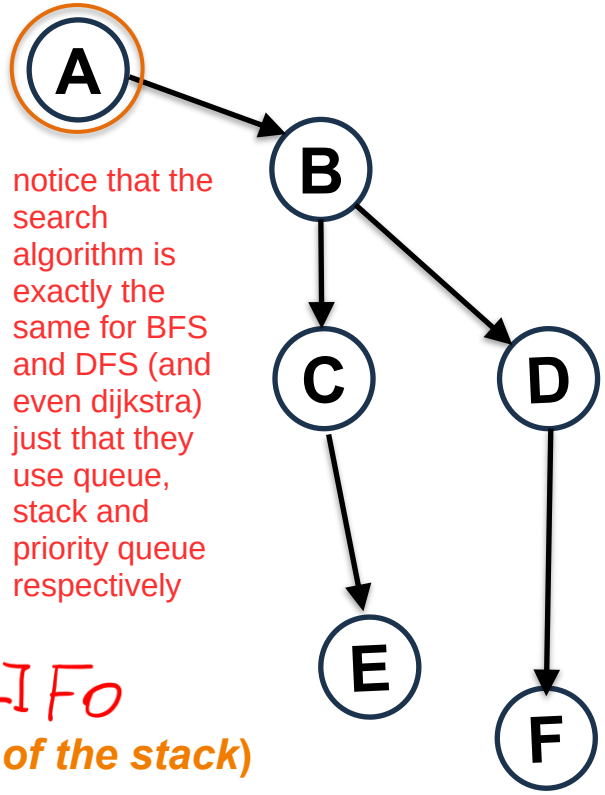
Stack: A Last-In-First-Out (LIFO) data structure, where the *last* element added is the *first* to be removed.

Example: Finding a path from A to E using DFS



Finding a path from A to E using DFS





notice that the search algorithm is exactly the same for BFS and DFS (and even dijkstra) just that they use queue, stack and priority queue respectively

- Start with a frontier (stack) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- **Pop the most recently added node (i.e., the *top of the stack*) from the frontier.**
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- Expand the node, for each resulting child node, if it's not in the frontier or explored set, push it onto the frontier (stack)

LIFO

Finding a path from A to E using DFS

Top

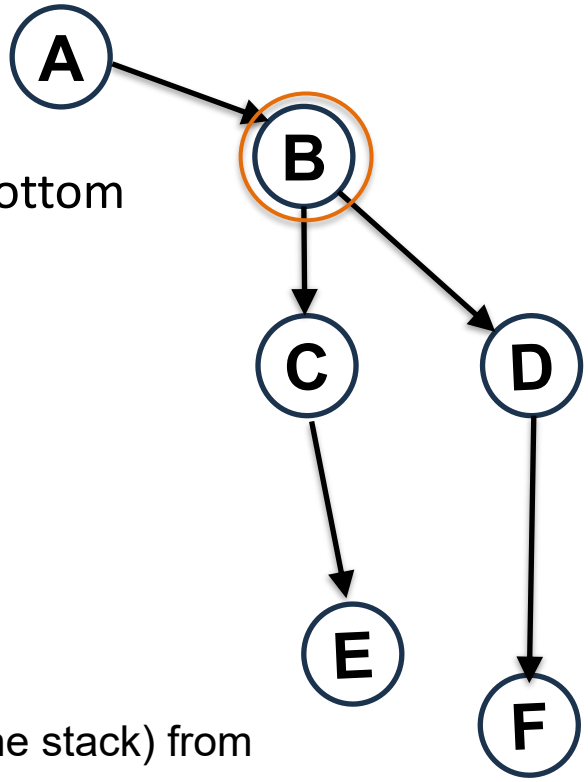
Frontier

D C

Bottom

Explored set

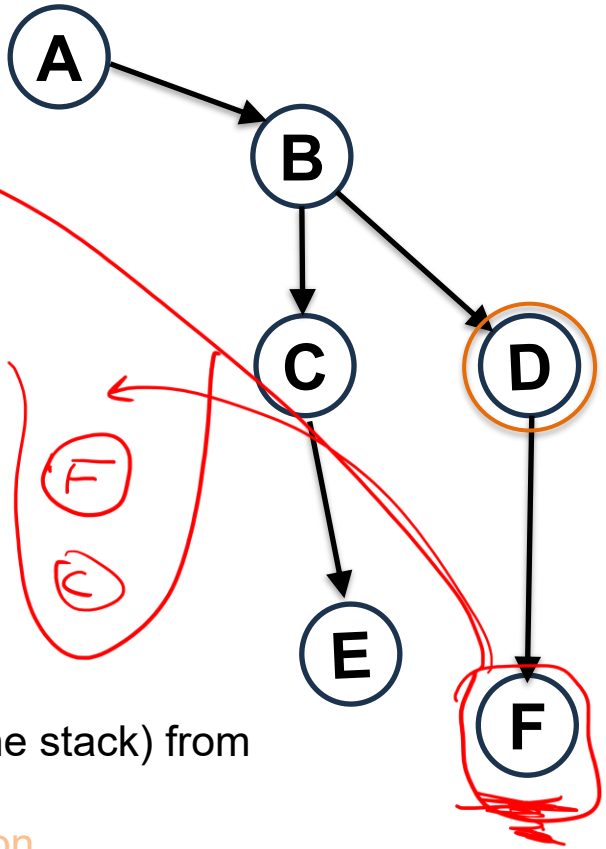
A B



- Start with a frontier (stack) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Pop the most recently added node (i.e., the top of the stack) from the frontier.
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- **Expand the node, for each resulting child node, if it's not in the frontier or explored set, push it onto the frontier (stack)**

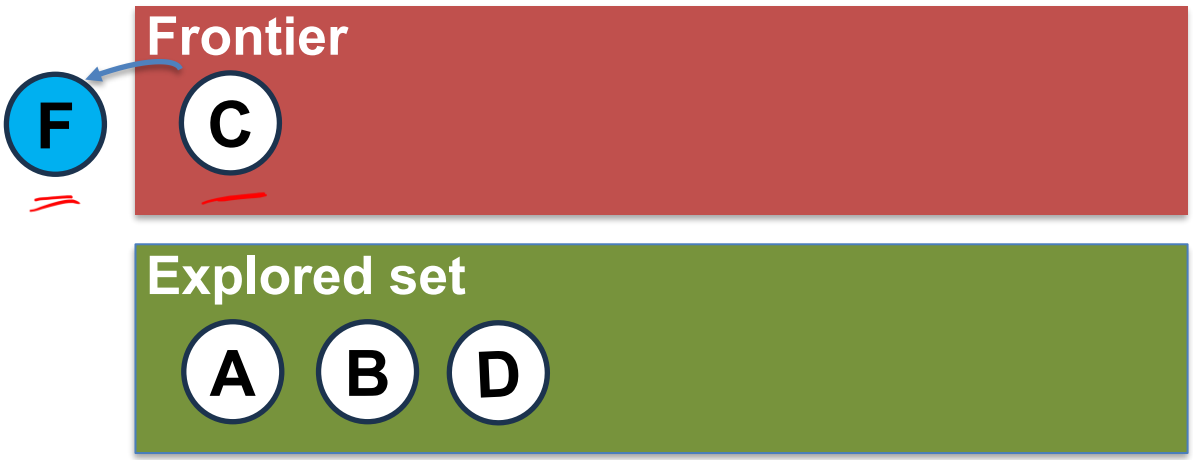
In this example, it doesn't matter whether we push C or D first

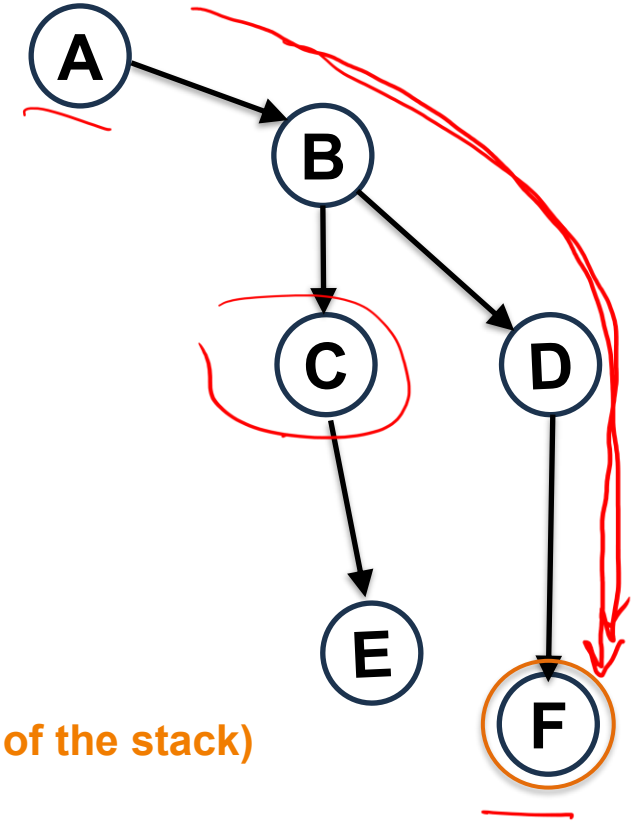
Finding a path from A to E using DFS



- Start with a frontier (stack) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Pop the most recently added node (i.e., the top of the stack) from the frontier.
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- **Expand the node, for each resulting child node, if it's not in the frontier or explored set, push it onto the frontier (stack)**

Finding a path from A to E using DFS

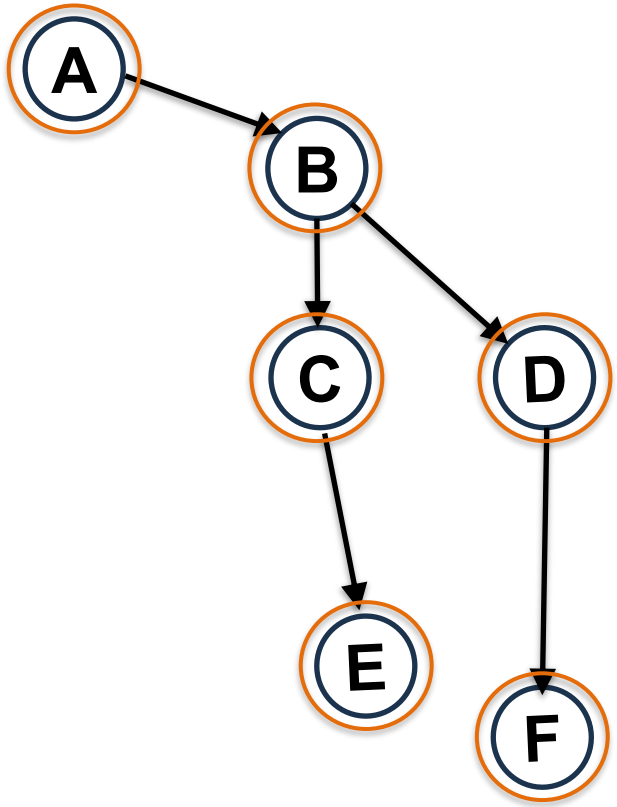




- Start with a frontier (stack) that contains the initial state.
- Start with an empty explored set.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- **Pop the most recently added node (i.e., the top of the stack) from the frontier.**
  - -- If this node contains the goal state, return the solution.
  - -- Add the node to the explored set.
  - -- Expand the node, for each resulting child node, if it's not in the frontier or explored set, push it onto the frontier (stack)

Finding a path from A to E using DFS

- Explore one path as far as it can go before backtracking.
- All the successors of a node are expanded before moving to other siblings.
- Often used for tasks like puzzle solving or analyzing maze-like structures where a solution path may be deep.



Summary: BFS vs DFS

(a)BFS always finds the shortest path in an unweighted gra
(b) BFS guarantees completeness in finite search spaces.
(d) BFS goes level by level, making it suitable for finding the shallowest solutio

NUS National University of Singapore

Feature	BFS (Breadth-First Search)	DFS (Depth-First Search)
Search style	Explores all nearby nodes first (level by level)	Goes as deep as possible before backtracking
Data structure used	Queue (FIFO - First in, first out)	Stack (LIFO - Last in, first out)
Will it always find a solution if one exists? (Completeness)	Yes, if a solution exists	Not guaranteed – may get stuck in deep or infinite paths
Will it find the shortest solution? (Optimality)	Yes, if all steps cost the same	No, may find a longer or less efficient path
Time needed	Slower if the solution is deep	Can be faster if the solution is deep
Memory needed	High – remembers all paths at one level	Low – tracks only one path at a time
Best for	Finding shortest paths (e.g., in maps) Small search space	Exploring puzzles or scenarios with deep solutions Large or deep search spaces
When to use	Need shortest route/minimum hops Equal step cost Enough memory	Memory is limited When solution may be far/deep Don't need shortest path

this is the only difference in implementation which results in all sorts of different performance and characteristics

2nd layer RHS onwards may not be explored, slide 100

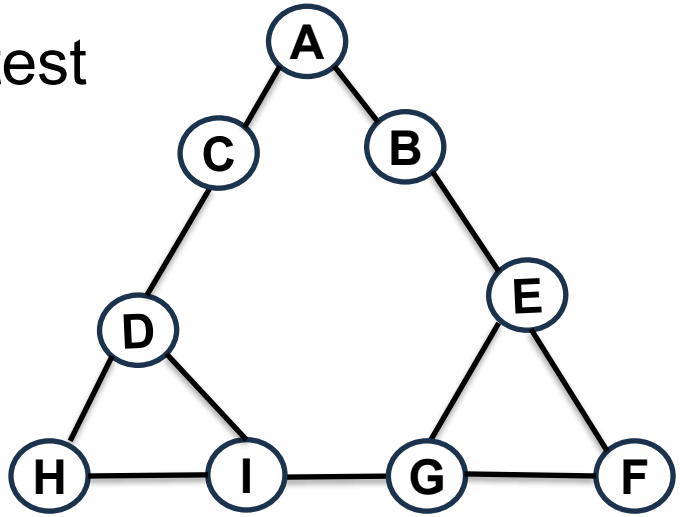
ONLY UNWEIGHTED

only focus on going deep to search for goal

the recursion stack will be destroyed every backtracking, slide 102

Examples: BFS vs DFS

Example 1: Finding the shortest path from A to C



Example 2: Finding the path in the maze

Lecture2_maze_BFS_vs_DFS.pdf (credit: Harvard CS50's Introduction to Artificial Intelligence with Python)

Reference1: https://www.youtube.com/watch?v=99oC5Kwxm5w
Reference2: https://www.youtube.com/watch?v=WbzNRTTrX0g

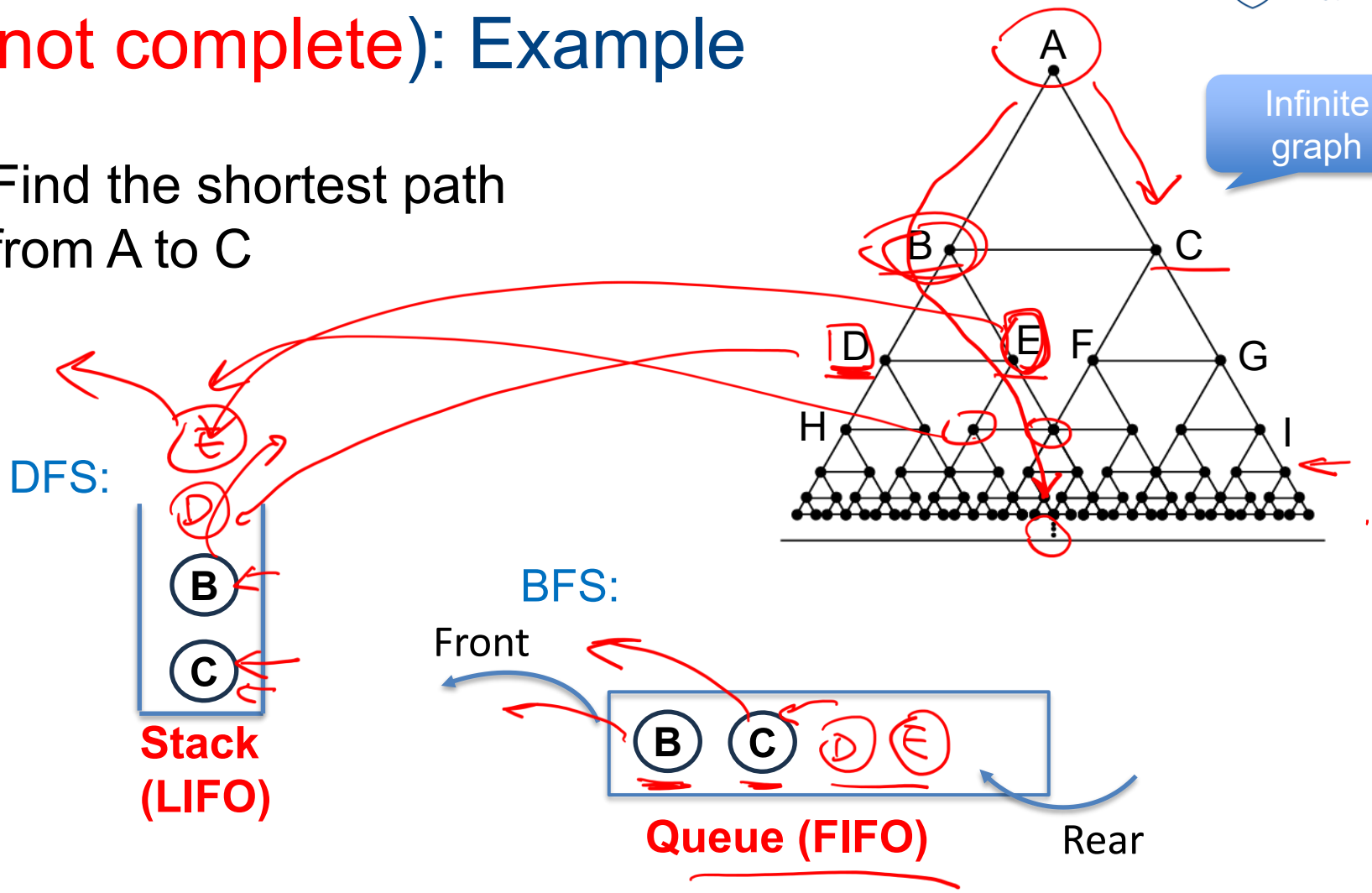
EE2213 Introduction to Artificial Intelligence

Lecture 3

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

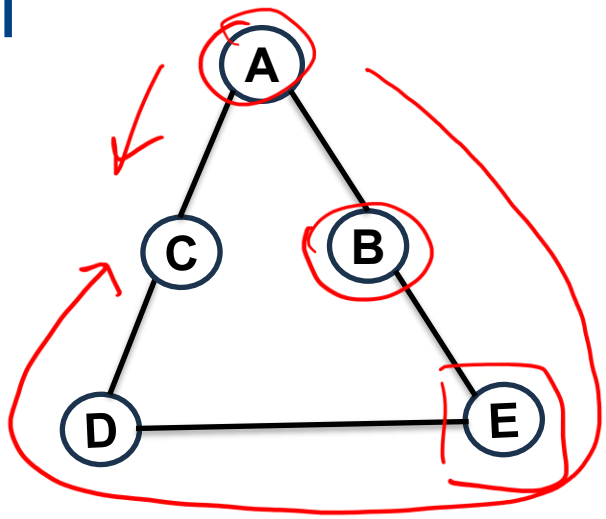
DFS may get stuck in a deep path (not complete): Example

Find the shortest path from A to C

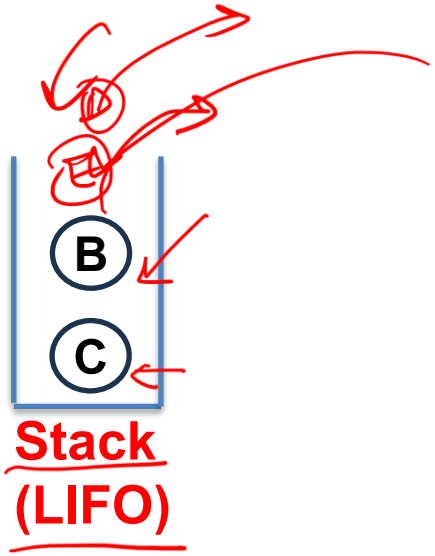


DFS may find a longer or less efficient path (not optimal): Example 1

Find the shortest path from A to C



DFS:



Stack (LIFO)

Ans from DFS: A -> B -> E -> D-> C

BFS:

Front

Queue (FIFO)

Rear

Ans from BFS: A -> C

Reference: https://www.youtube.com/watch?v=99oC5Kwxm5w

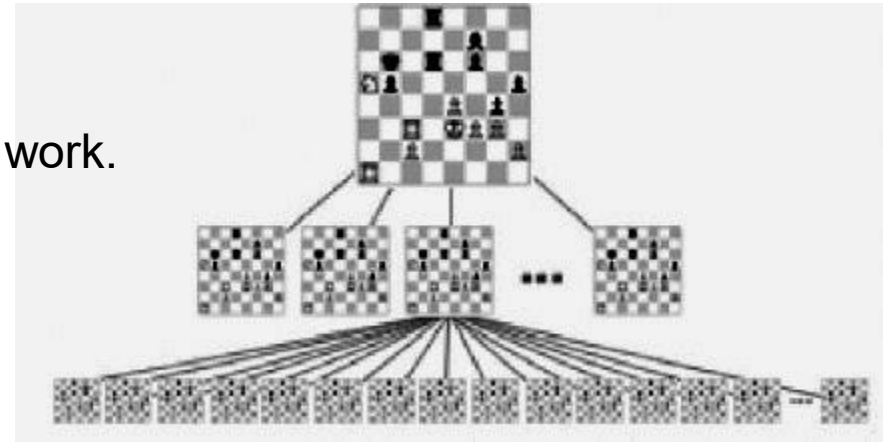
DFS: A natural fit for puzzles and games

- **Deep solutions:**
  DFS is good at finding solutions that are many steps away — common in puzzles and games.
- **Lower memory use:**
  DFS stores fewer nodes at a time, which helps in games with many possible moves like chess.
- **Early wins:**
  DFS can find a complete solution quickly, even if it's not the shortest.
- **Supports backtracking:**
  DFS supports trial-and-error strategies — try a path, and backtrack if it doesn't work.

solution is deep

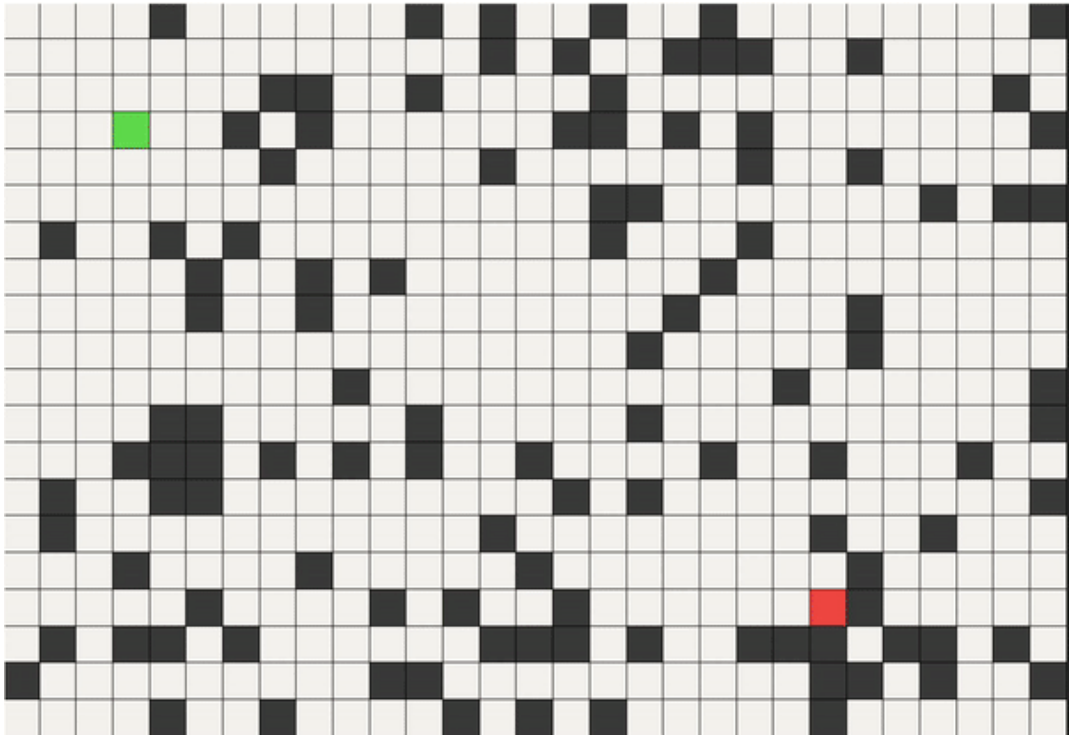
many possible states

using recursion we can add in backtracking or pruning



Depth-first search (DFS) in maze

- Expand the deepest unexplored node in the frontier first.
- *Explore one path as far as it can go before backtracking.*
- All the successors of a node are expanded before moving to other siblings.



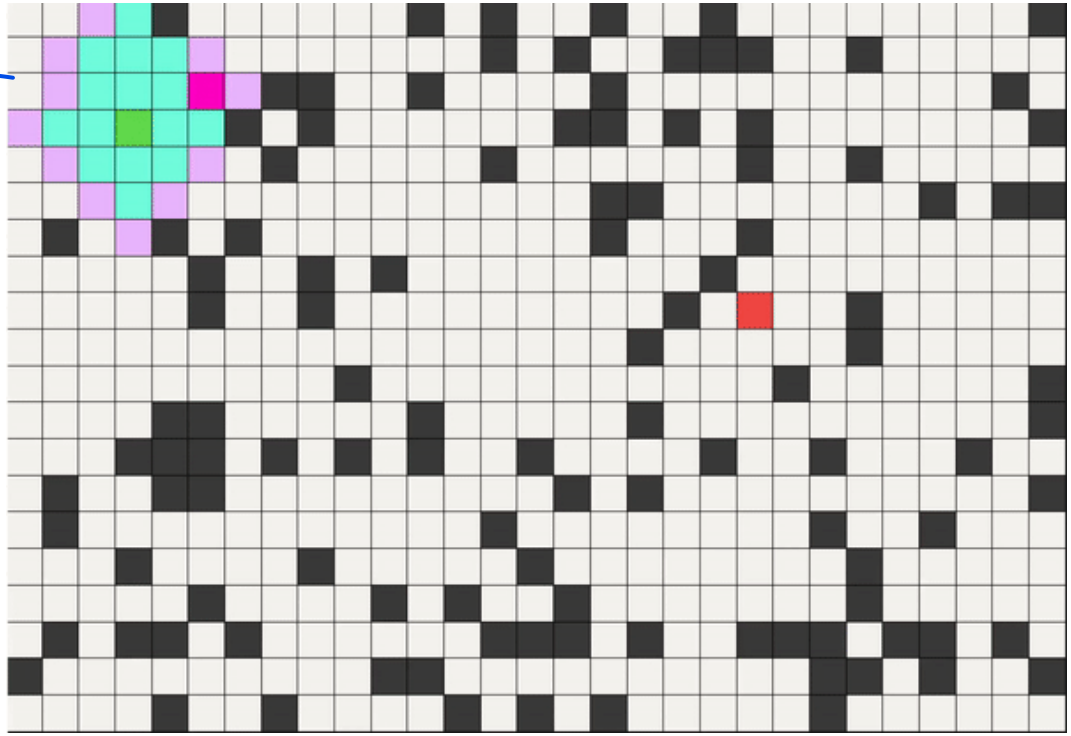
Source: https://medium.com/codex/python-project-idea-graph-traversal-and-pathfinding-algorithm-visualisations-99595c414293

Breadth-first search (BFS) in maze

- Explore nodes in "layers" (level by level)
- Can find everything findable from a given starting node
- *Can compute minimum hops between two nodes (i.e., shortest path, if all steps cost the same)*

not sssp

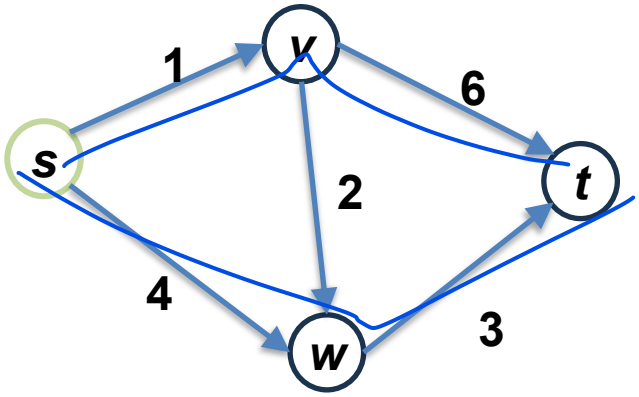
What if step costs are different?



Source: https://medium.com/codex/python-project-idea-graph-traversal-and-pathfinding-algorithm-visualisations-99595c414293

Quick question 1

Can we use BFS to find the shortest path from node *s* to node *t* ? The numbers on each edge represent the path distance of that edge.



BFS only finds minimum number of HOPS, it does not solve single source shortest path because it does not consider weights of edges (costs of path)

Limitation of BFS

BFS is effective for finding the shortest paths only in graphs where all the edges have the same (unit) length.

***Uniform-Cost Search* (UCS, also known as Dijkstra's algorithm)** extends this by handling graphs with any *non-negative* edge length.

for negative weights, need to use belman ford to detact

**Uniform-cost search (UCS) is the name used in the AI community, while the theoretical computer science community refers to it as Dijkstra's algorithm.*

Agenda

Uniform ≠ Uninformed

- Uniform-cost search (Dijkstra's algorithm)
- Informed search (greedy best-first, A*)

**The uniformity is in the way the algorithm fairly (uniformly) evaluates and expands paths based on cost, rather than by depth or number of steps.*

Application: Map navigation problems

VSLI circuit design too

BFS & DFS assume step-cost are equal.

This is only true for some problems.

If **step costs vary**, **uniform-cost search (Dijkstra's algorithm)** should be used instead.

Dijkstra's algorithm & BFS

- Dijkstra's algorithm extends BFS for varying edge lengths
- Have a shared principle
- Similarity with equal edge lengths

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P94 – P96

Dijkstra's algorithm: Overview

Notation:

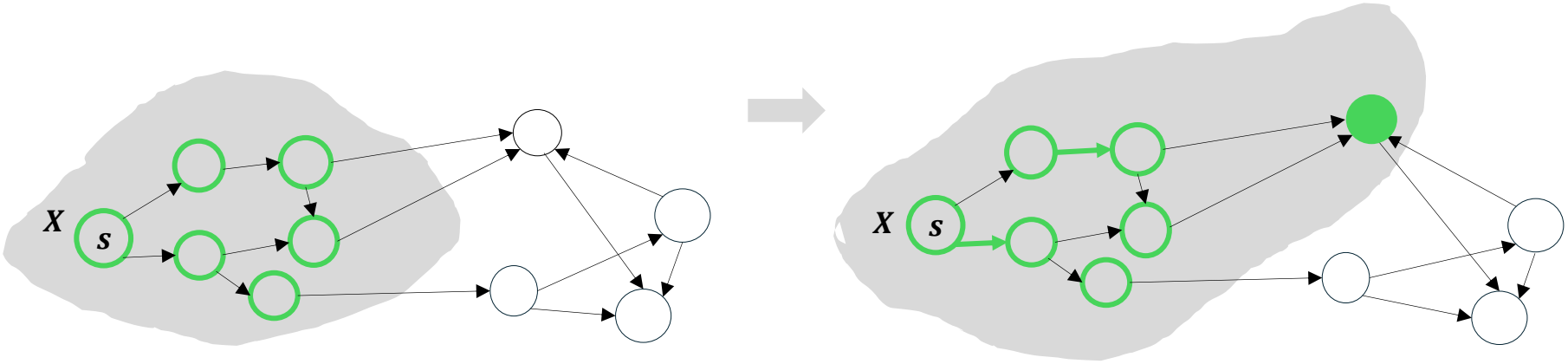
$X = \{\}$ [explored set: nodes processed so far]

$Q = \{\}$ [frontier: nodes waiting to be expanded]

$dist[v]$ [shortest path distances from the start s to v]

$e(v, w)$ [edge from v to w]

$l(v, w)$ [length/cost of $e(v, w)$]



Dijkstra's algorithm: Overview

Initialize:

$Q = \{s\}$ [start from initial sate s]

$dist[s] = 0$ [computed shortest path distances]

set all current estimated distance to 0 (should be inf based on 2040C)

Dijkstra's algorithm: Overview

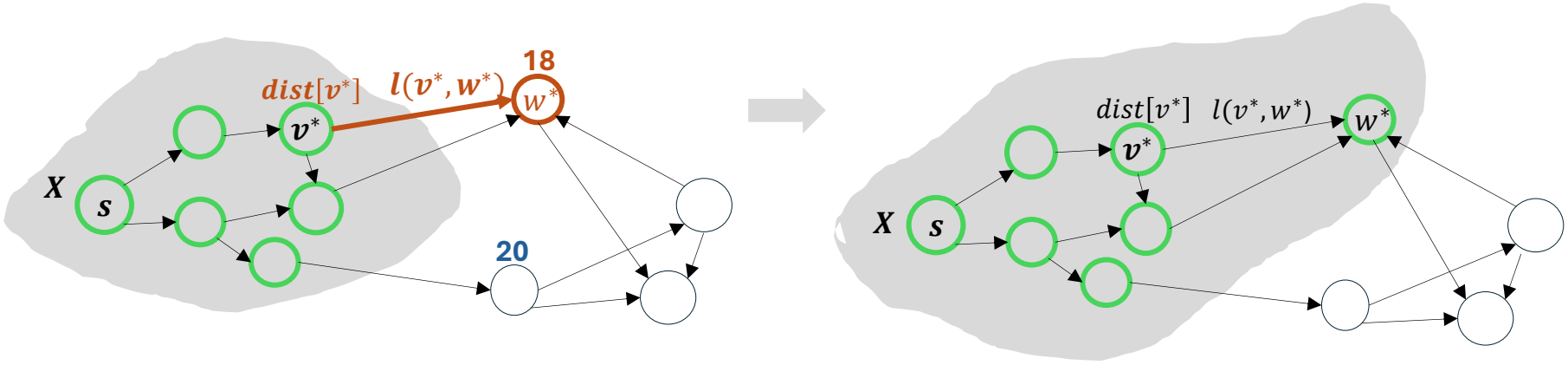
Loop:

Among all $e(v,w) \in E$, with $v \in X, w \notin X$,

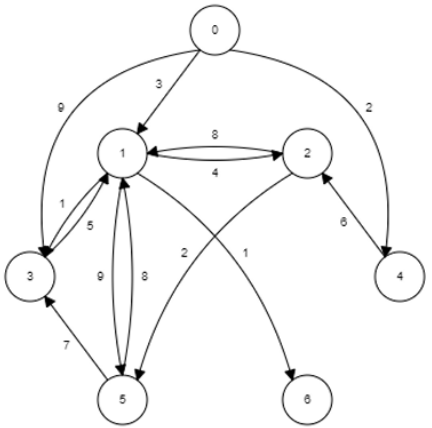
from frontier (pirority queue)
pick the one closes to source

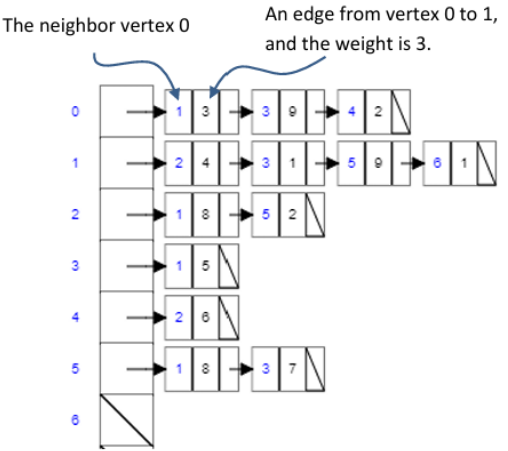
pick the one that minimizes $dist[v] + l(v,w)$ *(Dijkstra's greedy criterion)*, add w^* to $X, dist[w^*] = dist[v^*] + l(v^*,w^*)$

$\delta(u,v)=$ sum of weights of edges in the shortest path from u to v



Here is a graph (left) and its adjacency list (right). In the adjacency list, each node will point to another node with the left entry is the vertex index and the right value is the weight of the edge.





Perform Dijkstra Algorithm of this graph to find all the shortest distance from the node 0. Each of the following table is a priority queue sorted by the shortest estimated distance, δ(0,v).

Node v	δ(0,v)
0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞

Extract 0 →

Node v	δ(0,v)
1	3
4	2
3	9
2	∞
5	∞
6	∞

Extract 4 → +2

Node v	δ(0,v)
1	3
2	8
3	9
5	∞
6	∞

6+2

Extract 1 → +3

Node v	δ(0,v)
6	4
2	8
3	9
5	12
	9+3

Extract 6 → +4

Node v	δ(0,v)
2	8
3	9
5	12

Extract 2 → +8

Node v	δ(0,v)
5	10
3	9

Extract 3 → +9

Node v	δ(0,v)
5	12

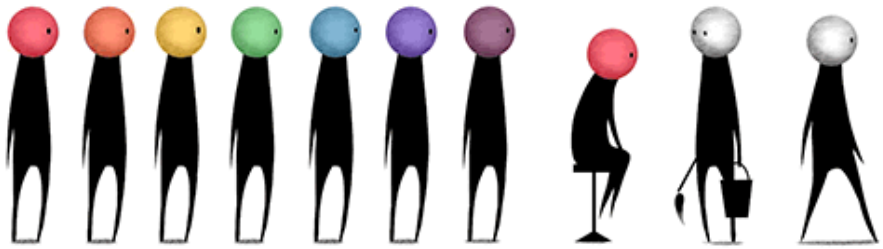
Extract 5 → +12

Node v	δ(0,v)

Recap: BFS—Frontier as a FIFO queue

- BFS always expands the *shallowest node* in the frontier
- In practice, the **frontier** in BFS is implemented as a **queue** (FIFO) to ensure nodes are expanded in the order they are discovered

Frontier: A queue (or other data structure) of nodes waiting to be expanded.

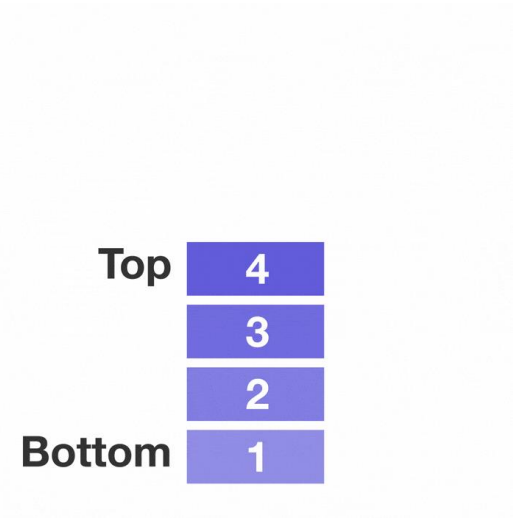


Queue: A First-In-First-Out (FIFO) data structure, where the first element added is the first to be removed.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P94 – P95.

Recap: DFS—Frontier as a LIFO stack

- DFS always expands the *deepest node* in the frontier
- In practice, the **frontier** is implemented as a **stack** (LIFO) to explore one branch fully before backtracking



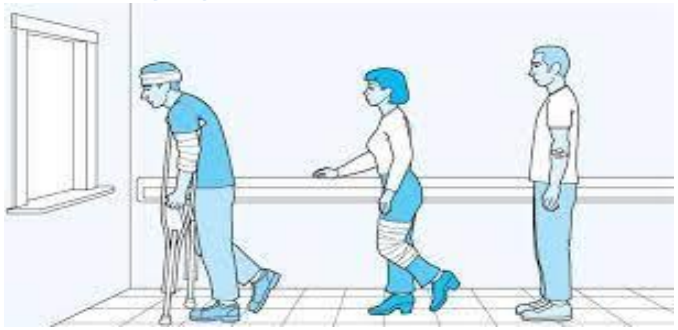
Stack: A Last-In-First-Out (LIFO) data structure, where the *last* element added is the *first* to be removed.

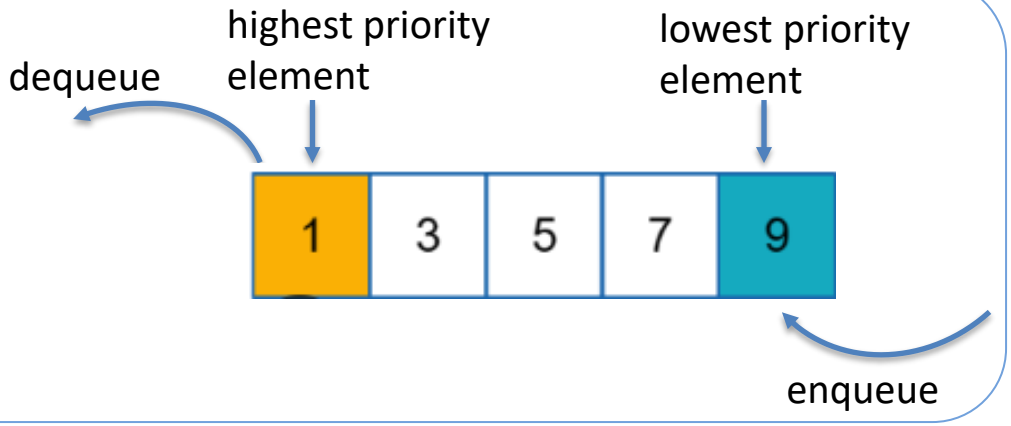
Dijkstra's algorithm: Frontier as a priority queue

- Dijkstra's algorithm always expands the frontier node with the *smallest distance from the source (i.e., initial state)*.
- In practice, the **frontier** is managed as a **priority queue,** so nodes in the frontier are expanded in order of their current *shortest* distance from the source.

Using Heap data structure

Priority queue:





Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P94 – P95.

Dijkstra's algorithm (uniform-cost search): Pseudocode

in the AI community language

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

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

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (3rd ed.) P84.

Dijkstra's algorithm (uniform-cost search): Pseudocode (v2)

- Start with a **frontier (priority queue)** containing the initial state (source), with path cost = 0.
- Start with an **empty explored set**.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Remove the frontier node with the lowest path cost from the source.
  - -- If this node is the goal, return the solution.
  - -- Add the node to the explored set.
  - -- Expand the node:
    - For each child, compute its total path cost from the source.
    - If the child is not in the frontier or explored set, **add it to the frontier** with its cost.
    - If the child is already in the frontier with a **higher cost**, update it with the **lower cost**.

If the new path is better than the old path, then discard the old one.

Dijkstra's algorithm (uniform-cost search): Pseudocode

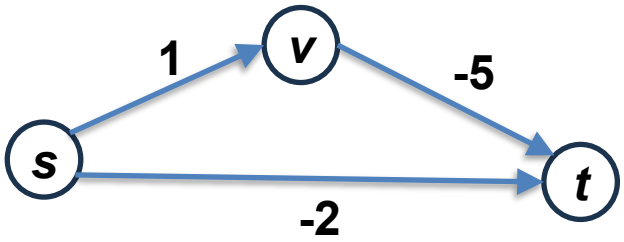
- Start with a **frontier (priority queue)** containing the initial state (source), with path cost = 0.
- Start with an **empty explored set**.
- Repeat:
  - -- If the frontier is empty, then no solution.
  - -- Remove the frontier node with the lowest path cost from the source.
  - -- If this node is the goal, return the solution.
  - -- Add the node to the explored set.
  - -- Expand the node:
    - For each child, compute its total path cost from the source.
    - If the child is not in the frontier or explored set, **add it to the frontier** with its cost.
    - If the child is already in the frontier with a **higher cost**, update it with the **lower cost**.

A node is *marked as visited* once it is removed from the frontier (if not goal)

Quick question 2

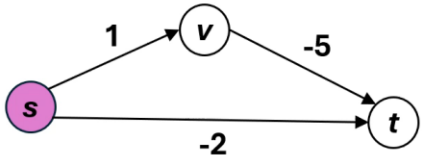
What is the computed shortest distance from *s* to *t*, when running Dijkstra's algorithm on this graph? Assuming it treats the graph normally despite negative edges.

(a) 1
(b) -4
(c) -2
(d) ∞



Quick question 2

Main loop: Iteration 1



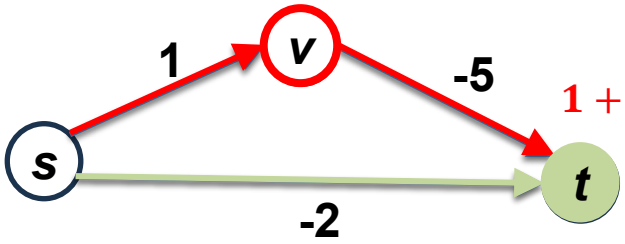
priority queue

vertex	dist
s	0

vertex	shortest distance from *a*	previous vertex
s	0	--
v	∞	--
t	∞	--

Processed Nodes = { }

Unprocessed Nodes = {s, v, t}



$1 + (-5) = -4$

Dijkstra's algorithm relies on *non-negative* edges!

Dijkstra's algorithm has a greedy nature

Greedy algorithms:

Make *locally* optimal choices at each step to achieve a *globally* optimal solution.

Dijkstra's algorithm is **greedy** because at each step, it selects the node with the **smallest known distance** from the source — assuming that this choice will lead to the overall shortest path.

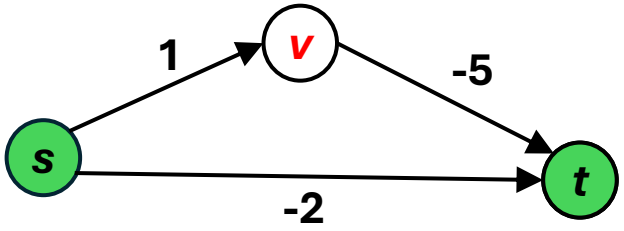
It **does not backtrack** or reconsider previous choices, which is a hallmark of greedy algorithms.

However, it's also **guaranteed to find the shortest path** in graphs with **non-negative edge** weights, so the greedy choice works correctly in that setting.

Why Dijkstra's algorithm fails in negative edges

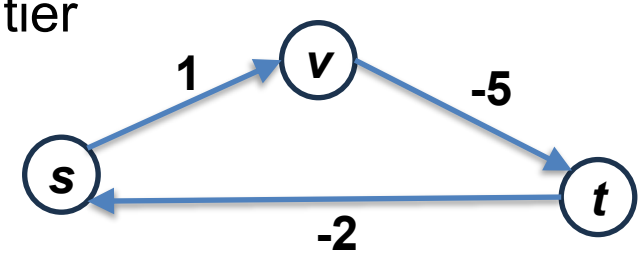
Assumes final path upon node removal:
Dijkstra assumes that once a node (like *t*) is removed from the frontier, the shortest path from the start node to it has been *finalized* (i.e., the node will not be added to the frontier again). This assumption breaks down when there are negative edge weights.

Fails to update better paths later:
Since *t* is first reached via $s \rightarrow t$ with cost –2, Dijkstra never enqueues t again, that is, it never reconsiders the better path $s \rightarrow v \rightarrow t$, which has a total cost of –4.



Can we modify it to handle negative edges?

If we revise Dijkstra's algorithm to allow revisiting nodes (i.e., allow nodes to be added to the frontier multiple times), it may lead to infinite loops in graphs with negative cycles.



Dijkstra's algorithm relies on non-negative edges.

For graph with negative edges or cycles, the Bellman-Ford algorithm is specifically designed to handle them properly (not covered in this course)

Further reading (beyond syllabus): https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

BFS vs DFS vs Dijkstra's algorithm

Feature	BFS (Breadth-First Search)	DFS (Depth-First Search)	Dijkstra's (Uniform-cost search)
Search style	Explores all nearby nodes first (level by level)	Goes as deep as possible before backtracking	Expands the node with the *lowest total cost* from the source (initial state)
Data structure used	Queue (FIFO - First in, first out)	Stack (LIFO - Last in, first out)	Priority queue (based on path cost)
Will it always find a solution if one exists? (Completeness)	Yes, if a solution exists	Not guaranteed – may get stuck in deep or infinite paths	Yes
Will it find the shortest solution? (Optimality)	Yes, if all steps cost the same	No, may find a longer or less efficient path	Yes, even when edge costs differ
Can it handle different edge costs?	No – assumes all edges have equal cost	No – ignores cost	Yes – works with any non-negative edge cost
Time efficiency	Slower if the solution is deep	Can be faster if the solution is deep	Slower in large graphs with varying costs
Memory usage	High – remembers all paths at one level	Low – tracks only one path at a time	High – must track cumulative costs for all paths
Best for	Finding shortest paths (e.g., in maps) Small search space	Exploring puzzles or scenarios with deep solutions Large or deep search spaces	Finding the cheapest/least-cost path in weighted graphs (e.g., shortest time, lowest expense)
When to use	Need shortest route Equal step cost Enough memory	Memory is limited When solution may be far/deep Don't need shortest path	Need shortest route with different edge costs; can afford higher memory usage

min hops

fastest search time for deep goal

SSSP

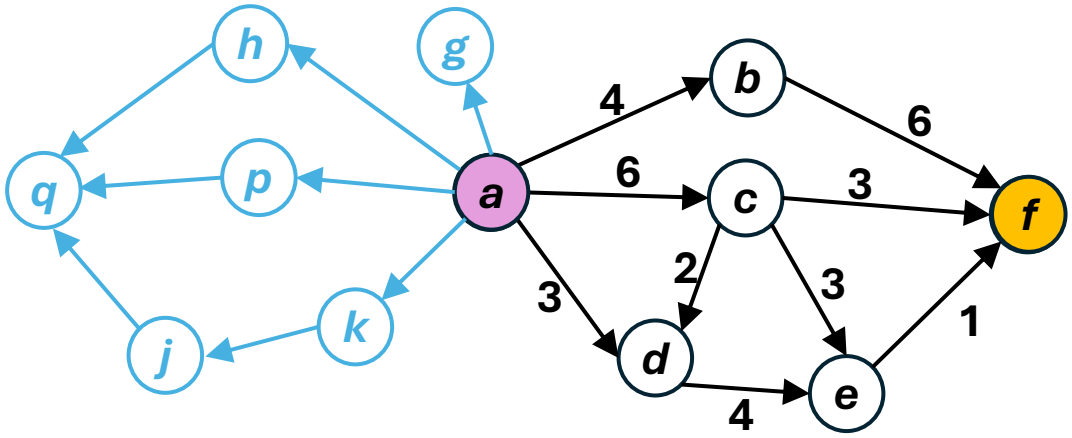
Quick question 3

If the starting node *a* has neighbors in the opposite direction of the target vertex *f*, will Dijkstra's algorithm explore them? If so, how can we improve its efficiency?

Yes

dijkstra's will explore all nodes

use heuristics



Agenda

- Uniform-cost search (Dijkstra's algorithm)
- Informed search (greedy best-first, A*)

Uninformed search & informed search

Uninformed search (BFS, DFS, Dijkstra's)

- Has no additional knowledge beyond what is given in the problem definition.
- Generate successor states and check whether each is a goal.
- As a result, the search proceeds blindly—systematically exploring the space until the goal is found.
- Search methods are distinguished by the order in which the nodes are expanded.

Informed search (greedy best-first, A*)

- Uses additional knowledge (heuristic) to estimate how close a state is to the goal.
- The heuristic guides the search, often reducing the number of nodes explored and improving efficiency.

Greedy best-first search

An informed search algorithm that expands the node that *appears to be closest to the goal*, as estimated by a heuristic function $h(n)$

rational, not omniscient*

$h(n)$: Estimated cost of the cheapest path from the current node n to a goal. (for goal node: $h(n) = 0$)

*In AI, *we don't assume perfect knowledge* — we just aim for agents that *behave rationally*, even if the result isn't always optimal.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P102 – P103

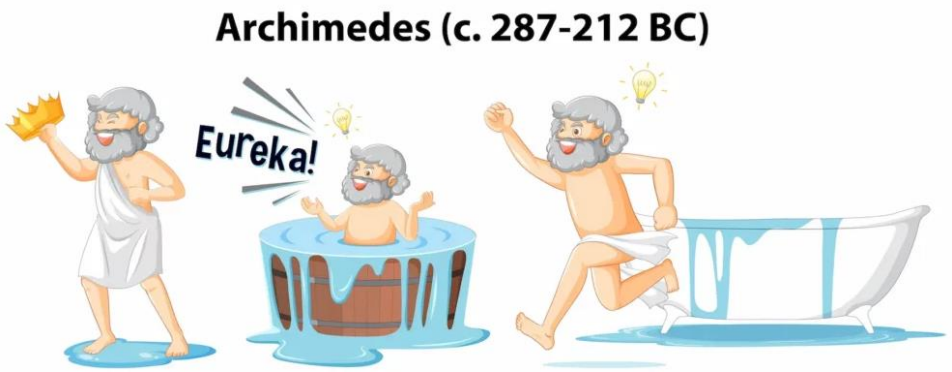
Heuristic function

Heuristic knowledge is useful, but NOT always accurate.

Heuristic algorithms use heuristic knowledge to guide the search process.

A **well-designed heuristic** can significantly reduce the number of nodes explored, making the search more efficient.

effectiveness depends on design



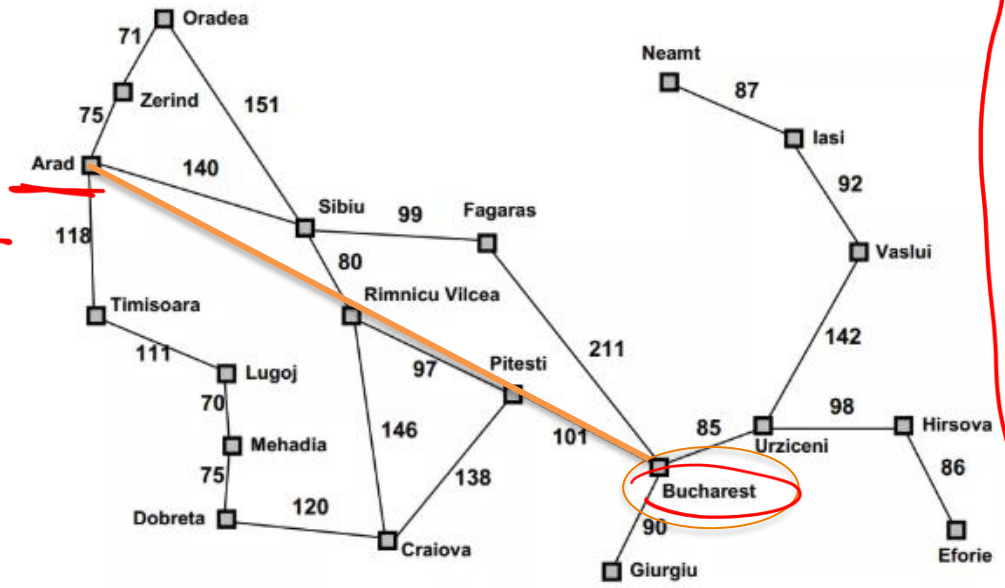
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P102 – P103

Example: Heuristic in path planning

when movement is allowed in any direction. but it is more computationally heavy than L1 distance (since need sqr + sqrt)

In map navigation, a common heuristic is the straight-line (Euclidean) distance to the goal.

This provides a reasonable estimate of how far a place is from the destination.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

For two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$, the Euclidean distance (also known as L2 distance) is:

$$d_{Euclidean}(p, q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The heuristic is NOT always accurate, as actual travel paths are rarely straight due to roads, obstacles, and terrain.

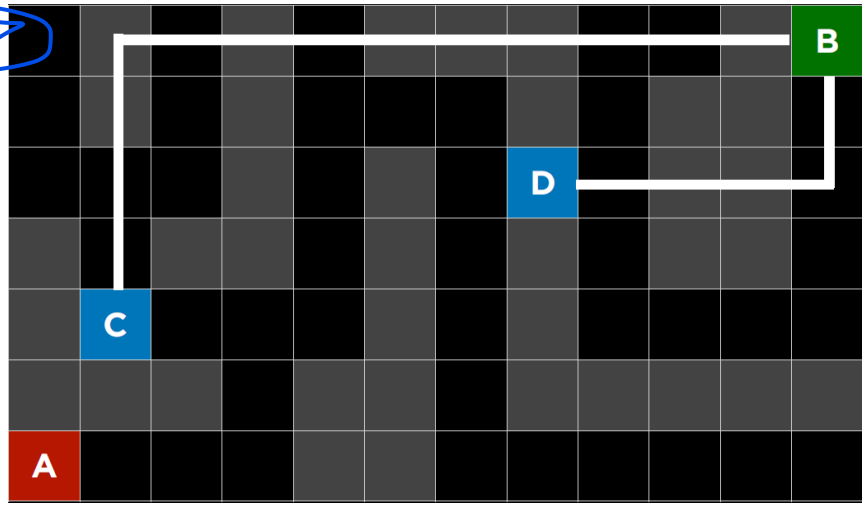
Example: Heuristic in grid maze

Heuristic used: Manhattan distance (also known as L1 distance), which sums horizontal and vertical steps to the goal.

when movement is restricted to horizontal + vertical only (no diagonal movements)

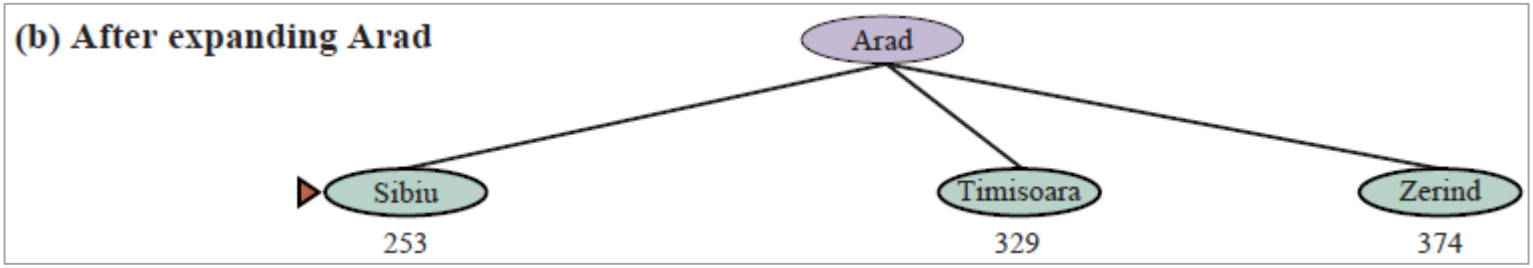
For two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$, the Manhattan distance is:

$$d_{Manhattan}(p, q) = |x_1 - x_2| + |y_1 - y_2|$$

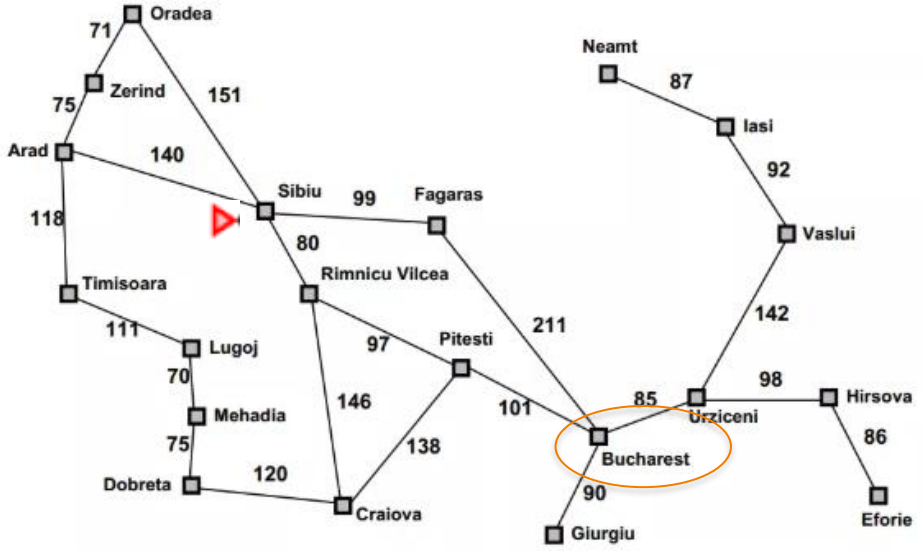


11		9		7				3	2		B
12		10		8	7	6		4			1
13	12	11		9		7	6	5			2
	13			10		8		6			3
	14	13	12	11		9		7	6	5	4
			13			10					
A	16	15	14			11	10	9	8	7	6

Path planning with greedy best-first



heuristic is
pick lowest

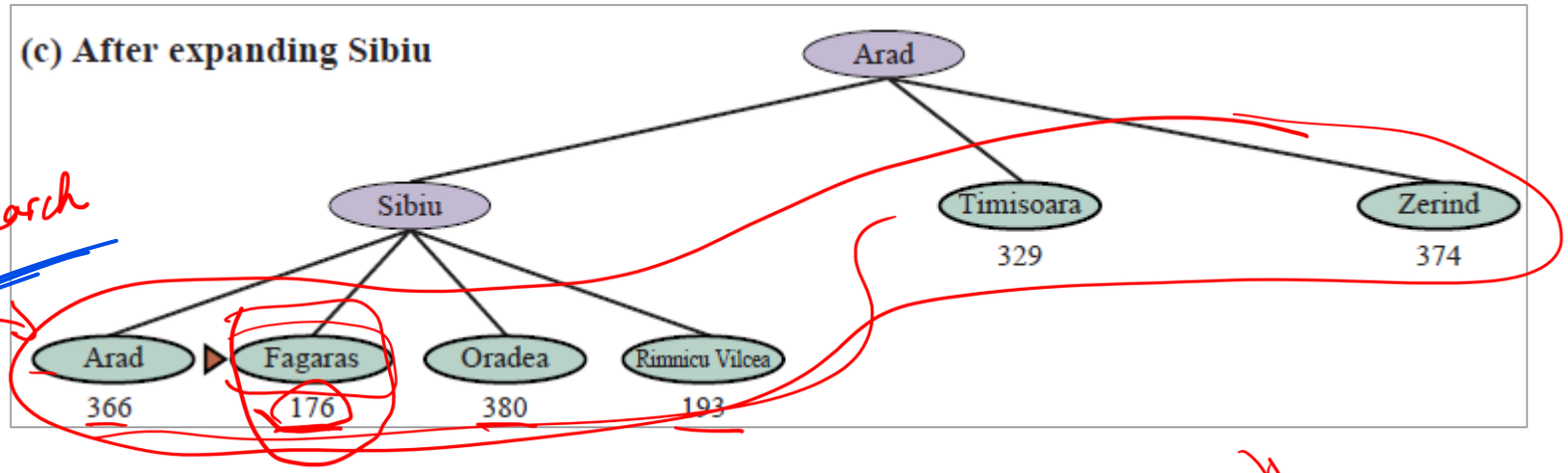


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P103 – P104

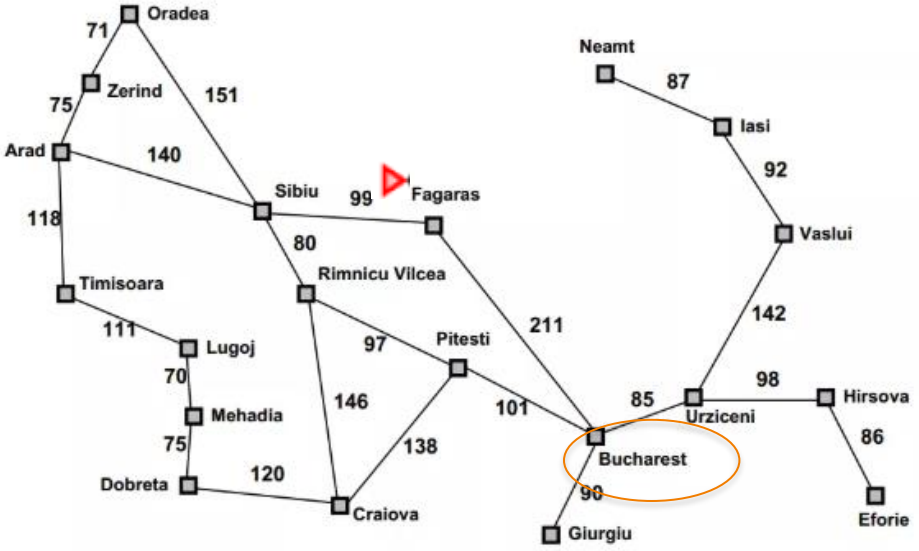
Path planning with greedy best-first



tree search

no need for graph search

since the duplicated node will not be visited anymore by heuristic

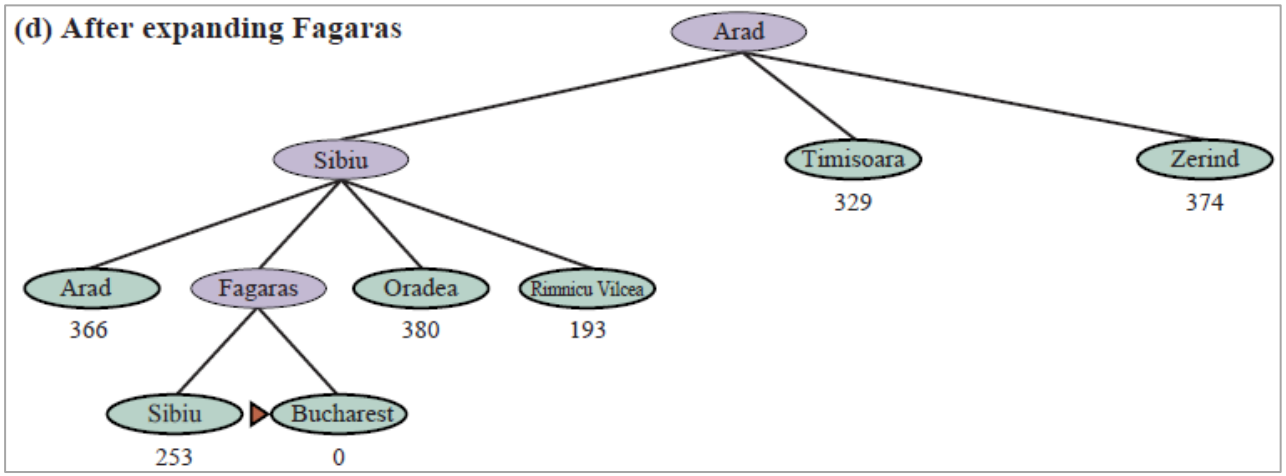


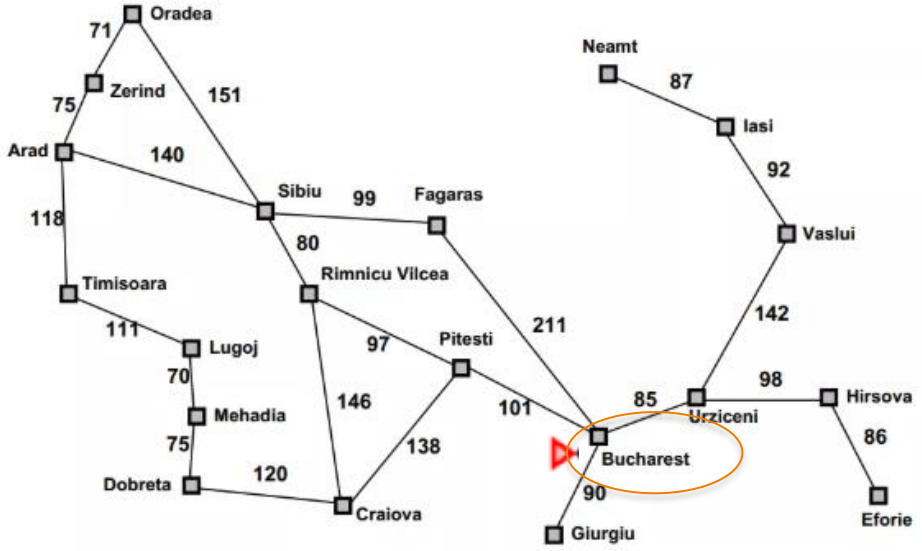
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P103 – P104

Path planning with greedy best-first





Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P103 – P104

Quick question 4

Which data structure is most suitable for managing the frontier in greedy best-first search?

A. Queue (FIFO)

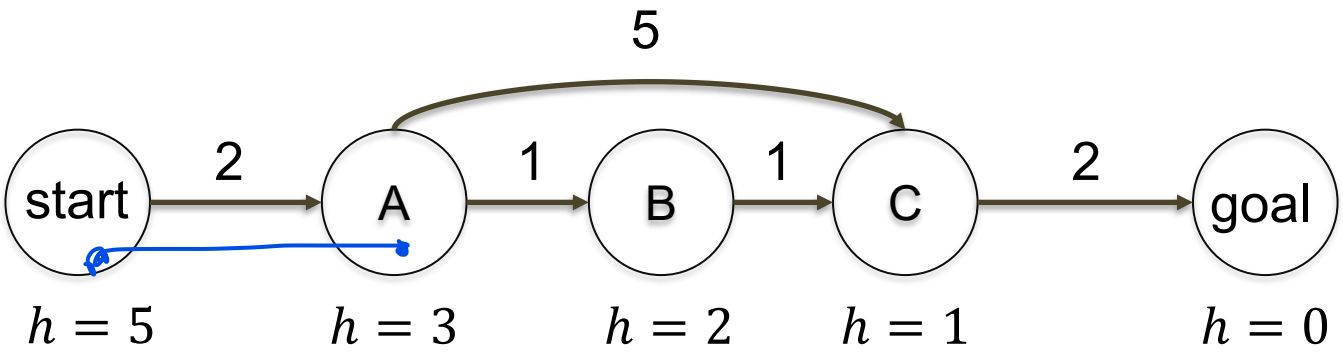
B. Stack (LIFO)

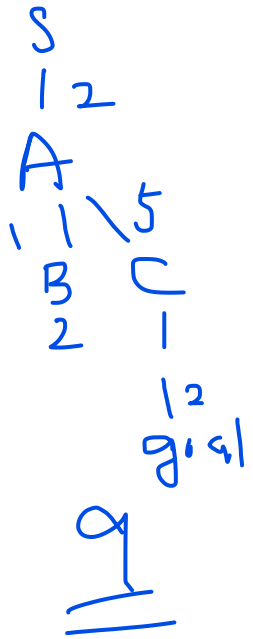
C. Priority queue

the heuristic function (instead of dijkstra's estimated distance) will be the greedy function for pq

Quick question 5

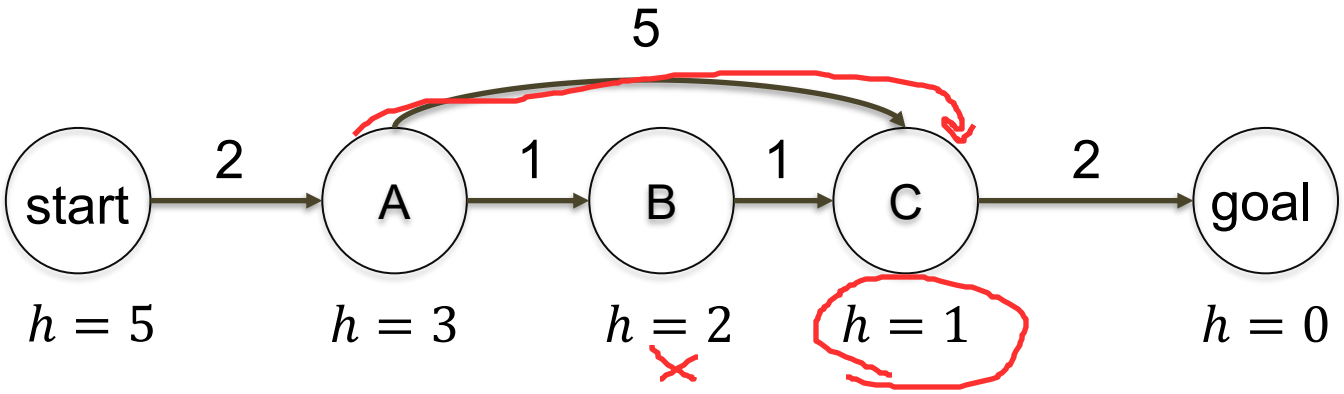
Which path does greedy best-first search return?





Each node is labeled with its heuristic estimate $h(n)$ (shown below the node), and each edge is labeled with its actual cost.

Quick question 4



Greedy best-first search: start → A → C → goal, path cost = 9

Actual optimal path: start → A → B → C → goal, path cost = 6

greedy BFS only cares about the heuristics, but not the path costs

Recap: Greedy best-first search

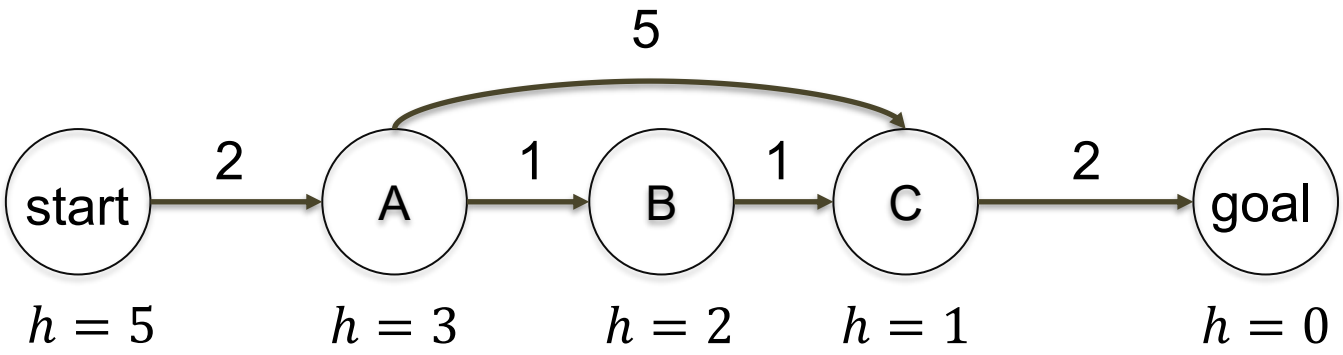
An informed search algorithm that expands the node that *appears to be closest to the goal*, as estimated by a heuristic function $h(n)$

$h(n)$: Estimated cost of the cheapest path from the current node n to a goal. (for goal node: $h(n) = 0$)

Greedy best-first search ignores the actual path cost

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P102 – P103

Greedy best-first search is not optimal



Greedy best-first search: start → A → C → goal, path cost = 9

Actual optimal path: start → A → B → C → goal, path cost = 6

Greedy best-first search uses *only the heuristic* to choose the node that seems closest to the goal, *ignoring the actual cost* from the start—so it can miss the cheapest path.

Greedy best-first search isn't optimal even with a well-defined heuristic

In this maze, each grid is labeled with its Manhattan distance to the goal as a heuristic. Use greedy best-first search to find the shortest path from A to B.

	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		10	9	8	7	6	5	4		2
	13		11						5		3
	14	13	12		10	9	8	7	6		4
			13		11						5
A	16	15	14		12	11	10	9	8	7	6

Greedy best-first search isn't optimal even with a well-defined heuristic

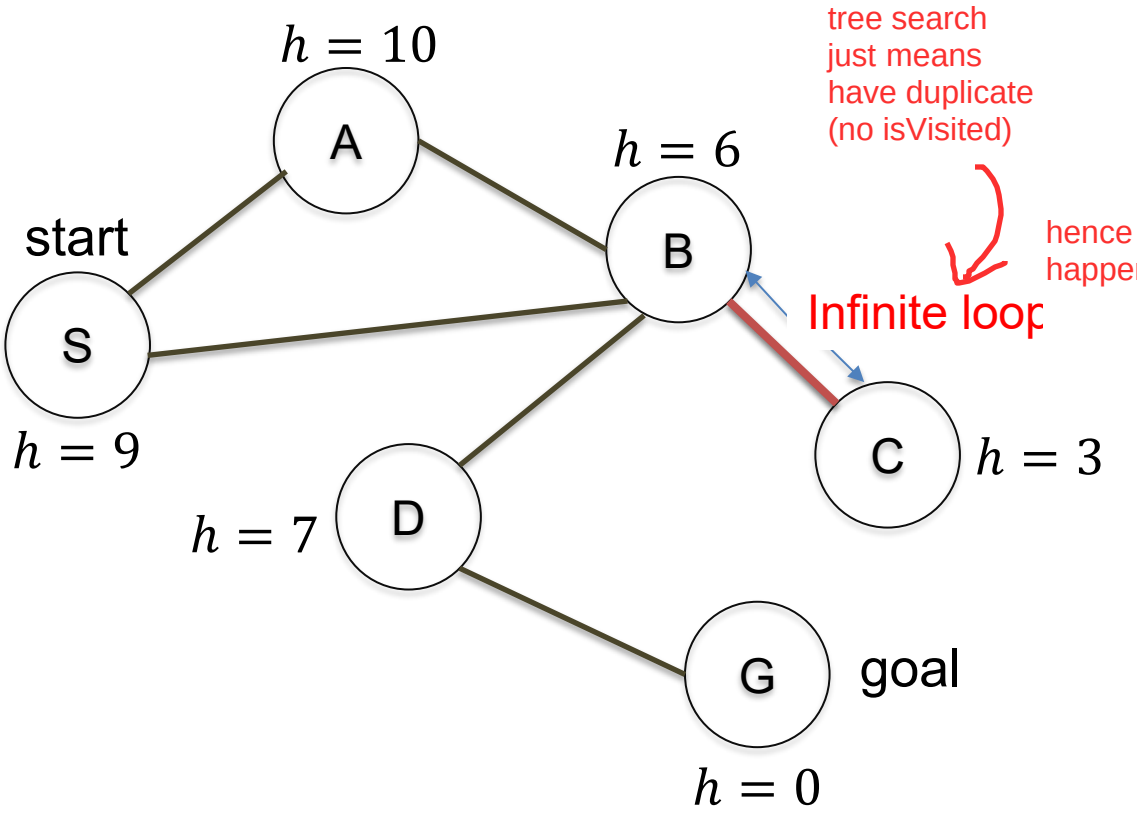
In this maze, each grid is labeled with its Manhattan distance to the goal as a heuristic. Use greedy best-first search to find the shortest path from A to B.

	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		10	9	8	7	6	5	4		2
	13		11						5		3
	14	13	12		10	9	8	7	6		4
			13		11						5
A	16	15	14		12	11	10	9	8	7	6

The problem isn't just the heuristic — it's the nature of greedy best-first search itself!

Greedy best-first search is not complete

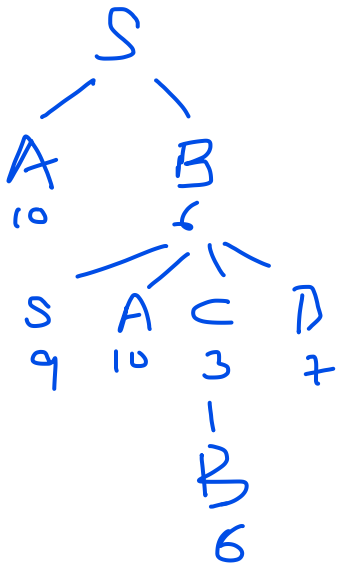
Use greedy best-first search to find a path from the start to the goal, assuming a tree search strategy.



tree search just means have duplicate (no isVisited)

hence this can happen

Infinite loop



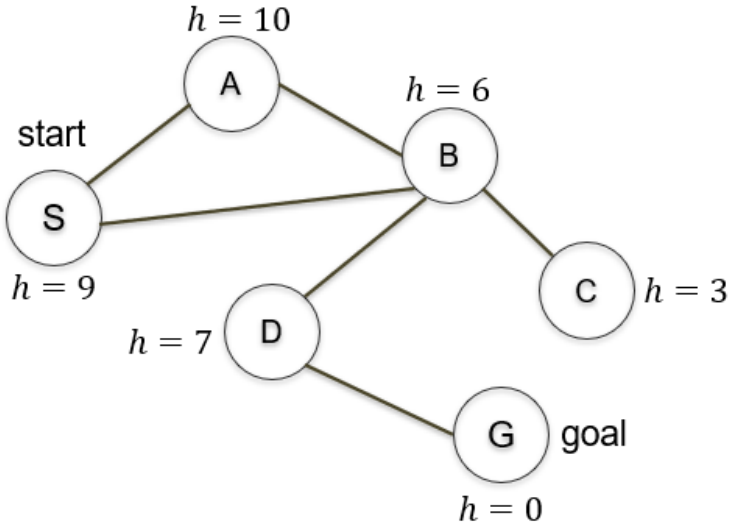
Each node is labeled with its heuristic estimate $h(n)$ (shown beside the node),

Start at S:

•The search begins at node **S** (h=9).

•Neighbors: A (h=10) and B (h=6), added to the frontier.

•Frontier: **[B(h=6), A(h=10)]**.



Expand B:

•**B** has the lowest heuristic value (6) in the frontier, so it's expanded next.

•Neighbors: A (h=10), S (h=9), C (h=3), and D (h=7), added to the frontier

•Frontier: **[C(h=3), D(h=7), S(h=9), A(h=10)]**.

Expand C:

•**C** has the lowest heuristic value (3) and is chosen next.

•Neighbor: B (h=6), added back to the frontier.

•Frontier: **[B(h=6), D(h=7), S(h=9), A(h=10)]**.

if this is smallest then will enter infinite

Expand B:

•**B** again has the lowest heuristic value (6) and is expanded. The path is currently **S -> B -> C -> B**.

•B's neighbors: (C, D, S, A) are again added to the frontier.

•Frontier: **[C(h=3), D(h=7), S(h=9), A(h=10)]**.

Expand C:

•**C** has the lowest heuristic value (3) and is chosen next.

•C's only neighbor is B (h=6).

•B is added back to the frontier.

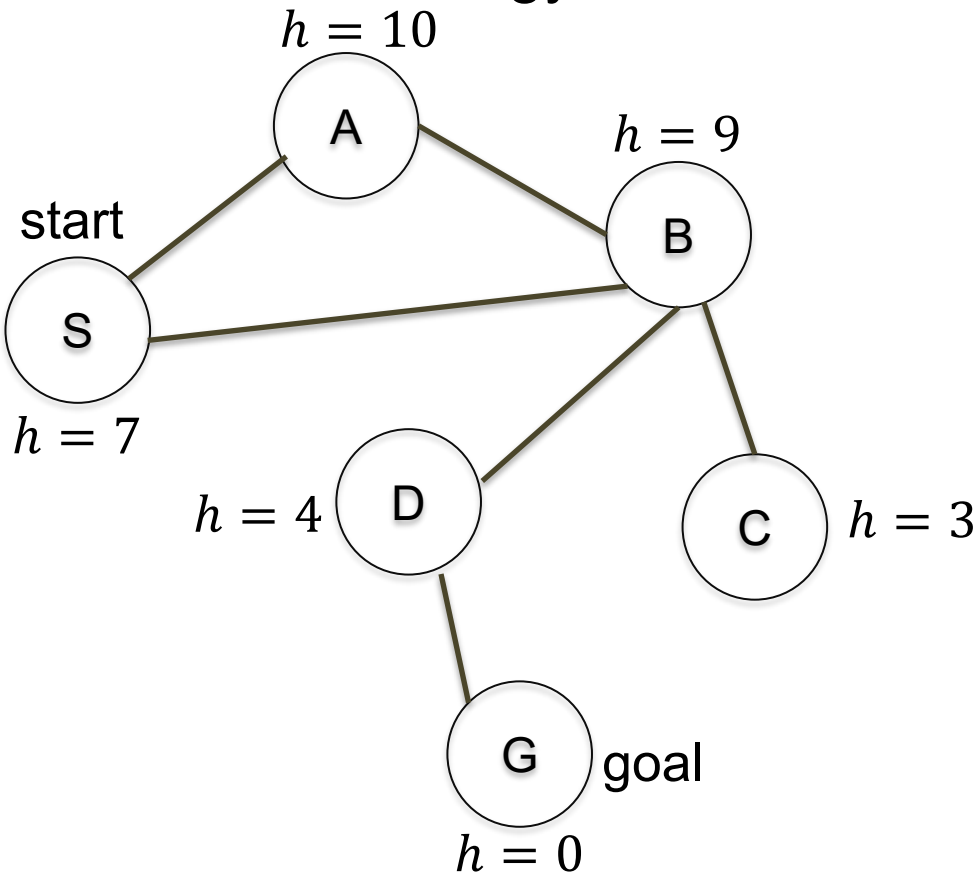
•Frontier: **[B(h=6), D(h=7), S(h=9), A(h=10)]**.

Infinite loop

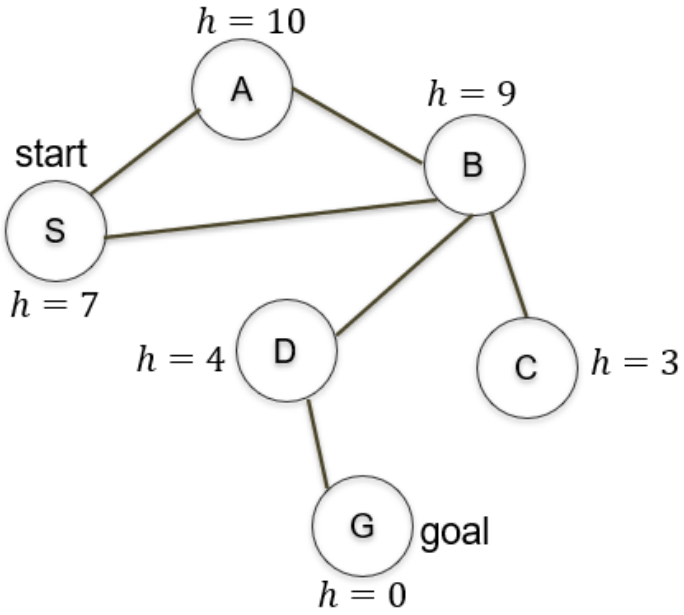
...

Earlier example:

Which path does greedy best-first search return, assuming tree search strategy?



Each node is labeled with its heuristic estimate $h(n)$ (shown beside the node),



Start at S (h=7):

•Neighbors: A (h=10), B (h=9).

•Frontier: **[B(h=9), A(h=10)]**.

Expand B (h=9):

•Neighbors: S (h=7), A (h=10), C (h=3), D (h=4).

•Frontier: **[C(h=3), D(h=4), S(h=7), A(h=10)]**.

Expand C (h=3):

•Neighbors: B (h=9).

•Frontier: **[D(h=4), S(h=7), B(h=9), A(h=10)]**.

•The path so far is **S -> B -> C**.

Expand D (h=4):

•The algorithm now chooses **D** because it has the lowest heuristic value (4) in the frontier.

•D's neighbors are B (h=9) and G (h=0).

•The goal state, **G**, has a heuristic of 0. The algorithm chooses G next.

•The path found is **S -> B -> D -> G** (this is because we previously get to D from B, that is, D's predecessor is B)

Properties of greedy best-first search

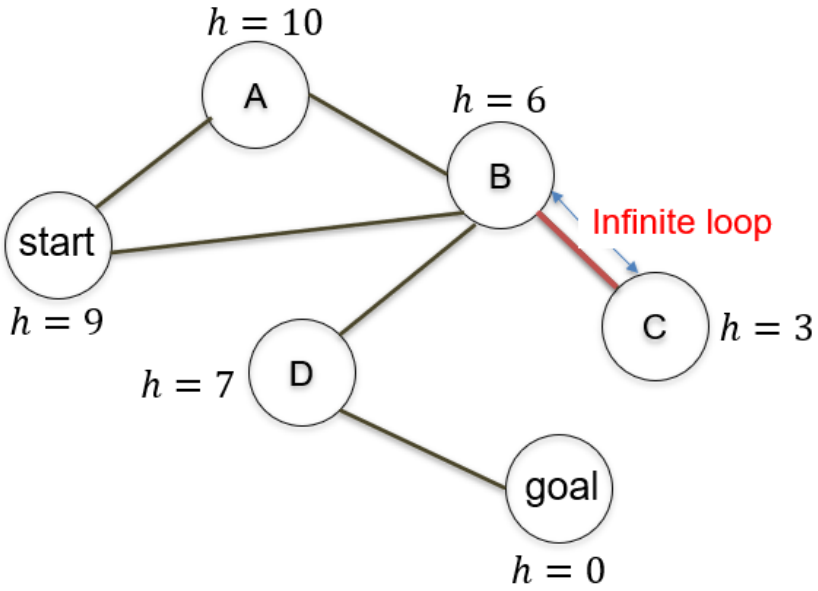
Complete? No–can get stuck in loops

Optimal? No–not guaranteed to render lowest cost solution.

Completeness: Does the algorithm always find a solution if one exists?
Optimality: Does it find the lowest-cost (best) solution among all possible ones?

Limitation of greedy best-first search

It *only considers* the estimated cost to the goal (heuristic) and *ignores* the actual cost already incurred to reach the current node. This can cause it to get trapped in local minima or follow paths that seem promising at first but stray from the optimal solution.



	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		10	9	8	7	6	5	4		2
	13		11						5		3
	14	13	12		10	9	8	7	6		4
			13		11						5
A	16	15	14		12	11	10	9	8	7	6

Q2. Which of the following is true about greedy best-first search? Select all that apply.

A. It guarantees both completeness and optimality when using a well-designed heuristic.

B. It expands nodes based solely on the lowest cumulative path cost from the start state.

C. It expands the node that appears closest to the goal, based only on the heuristic function. ✓

D. It behaves identically to uniform-cost search when edge costs are equal.

Q2 Takeaway

Greedy best-first search is:

- **Heuristic-guided**: At each step, it expands the node that **appears closest to the goal** based on the heuristic estimate (i.e., the smallest $h(n)$ value).
- **Not optimal**: It ignores the actual cost of the path taken so far ($g(n)$), so it may return a **suboptimal** solution.
- **Not always complete**: If the search space is infinite, contains cycles, or the heuristic is poor, it may get **stuck in a loop** or fail to find a solution, even if one exists.
- *Fast in practice* (sometimes): With a good heuristic, it can reach the goal quickly, but there are *no guarantees* of finding the best or even any solution.

EE2213 Introduction to Artificial Intelligence

Lecture 4

Dr Shaojing Fan
fanshaojing@nus.edu.sg

Agenda

- A* search
- Planning (STRIPS)

Properties of Dijkstra's algorithm (UCS)

Complete? Yes –when all edge costs are non-negative.

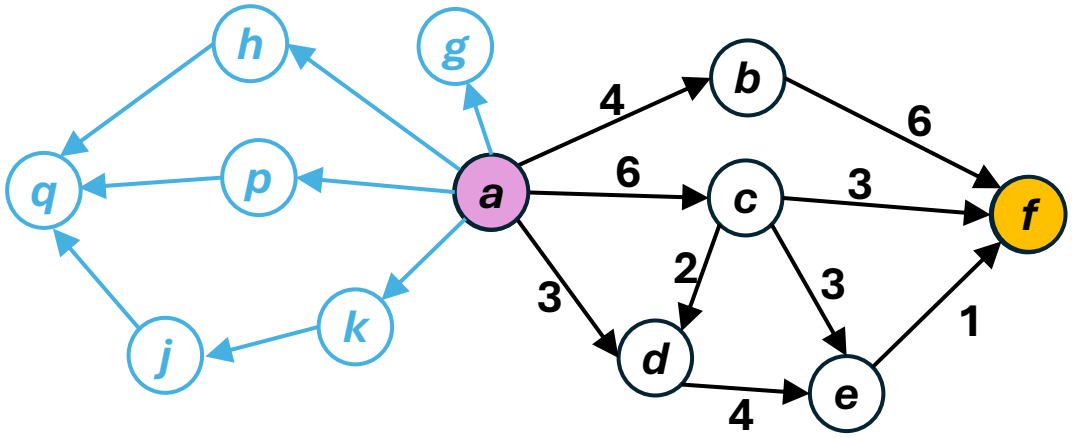
Optimal? Yes –when all edge costs are non-negative.

Completeness: Does the algorithm always find a solution if one exists?
Optimality: Does it find the lowest-cost (best) solution among all possible ones?

Dijkstra's algorithm may be inefficient

If the starting node *a* has neighbors in the opposite direction of the target vertex *f*, will Dijkstra's algorithm explore them? If so, how can we improve its efficiency?

Yes need heuristics



Dijkstra is complete and optimal, but inefficient if neighbours exists opp goal

GBFS use heuristics, but it is not complete (stuck in loop since using tree - no isVisited) and not optimal (only follow heuristics, not lowest cost)

A* search: A Preview

- A* search addresses the limitation of greedy best-first search by also considering the cost so far, not just the heuristic.

- Perhaps the most well-known search algorithm in AI and pathfinding

- Widely used in navigation systems like Google Maps, Apple Maps, and other routing applications

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P103 – P109

A*: Where *heuristics* meet *reality*

It evaluates nodes by combining $g(n)$, the actual cost to reach the node, and $h(n)$, the estimated cost to get from the node to the goal.

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

$f(n)$: Estimated total cost of the *cheapest path from the start to the goal via node* n

the greedy is now f(n) = g(n) + h(n)
dijkstra is only g(n)
GBFS is only h(n)

$g(n)$: Actual cost of the path from the start node to the current node n. (for start node: $g(n) = 0$)

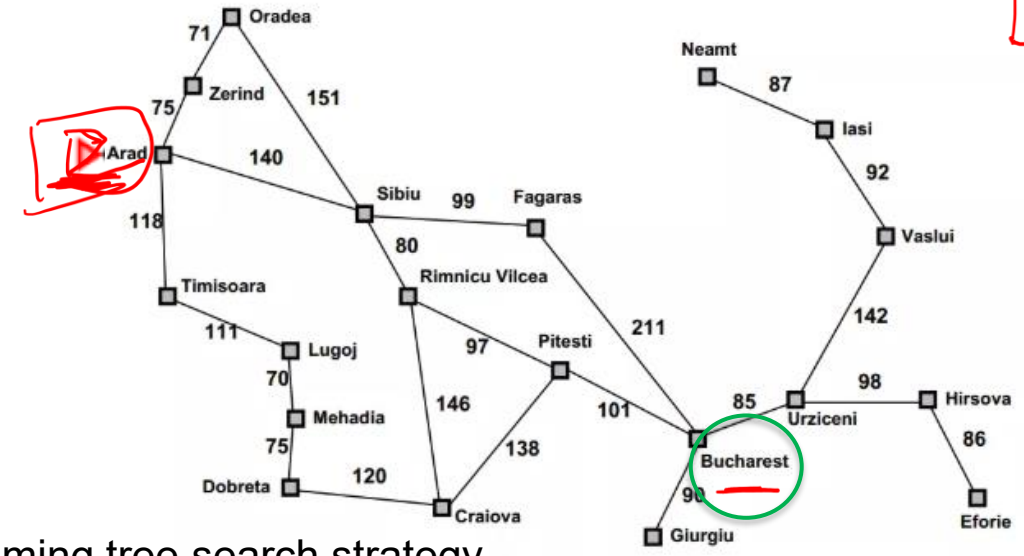
$f(n) = g(n) + h(n)$

$h(n)$: Estimated cost of the cheapest path from the current node n to the goal. (for goal node: $h(n) = 0$)

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P103 – P109

Example: Path planning with A*

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



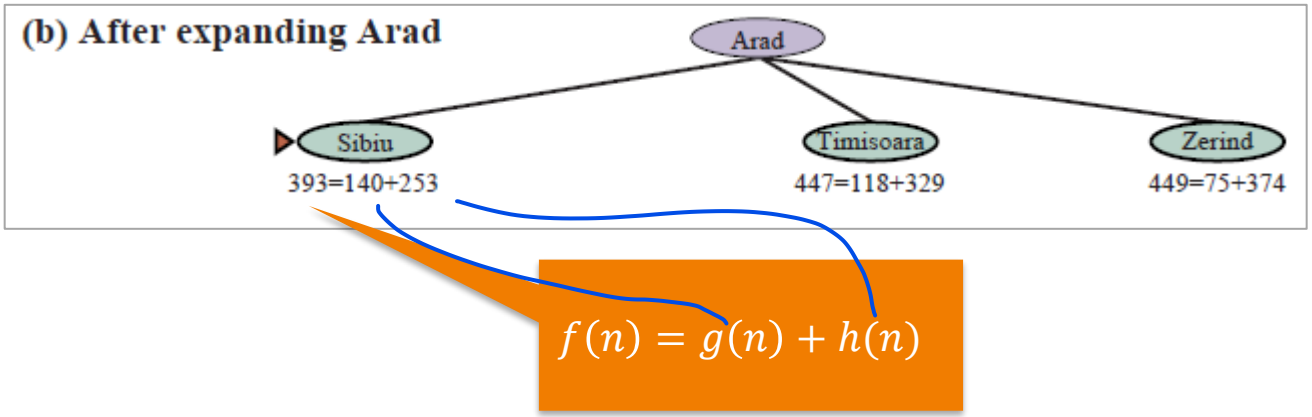
$h(n)$

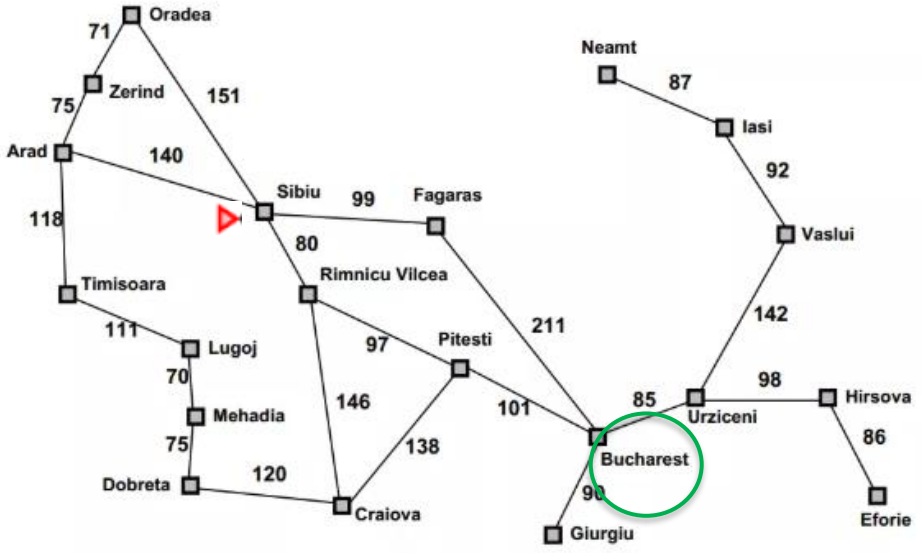
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

*Assuming tree search strategy

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Example: Path planning with A*



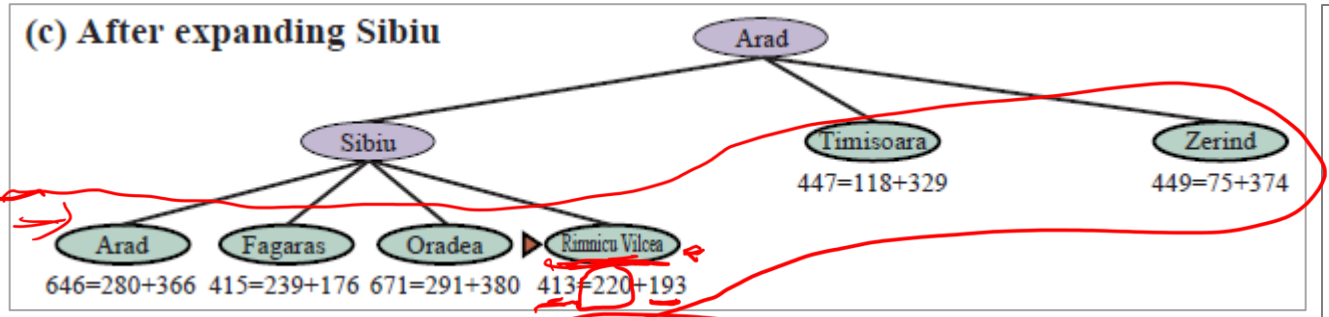


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

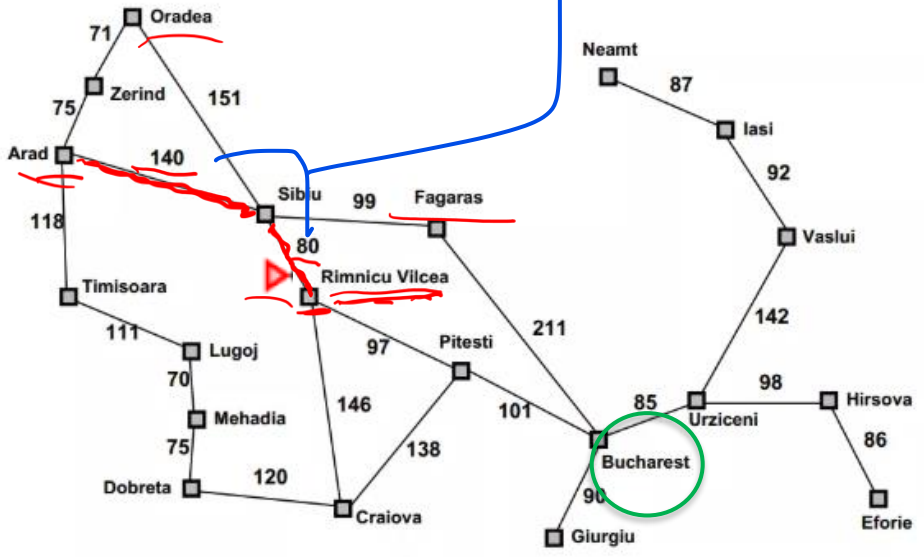
$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Example: Path planning with A*



Lavender: Nodes that have been expanded
Green: nodes currently on the **frontier**, waiting to be expanded.

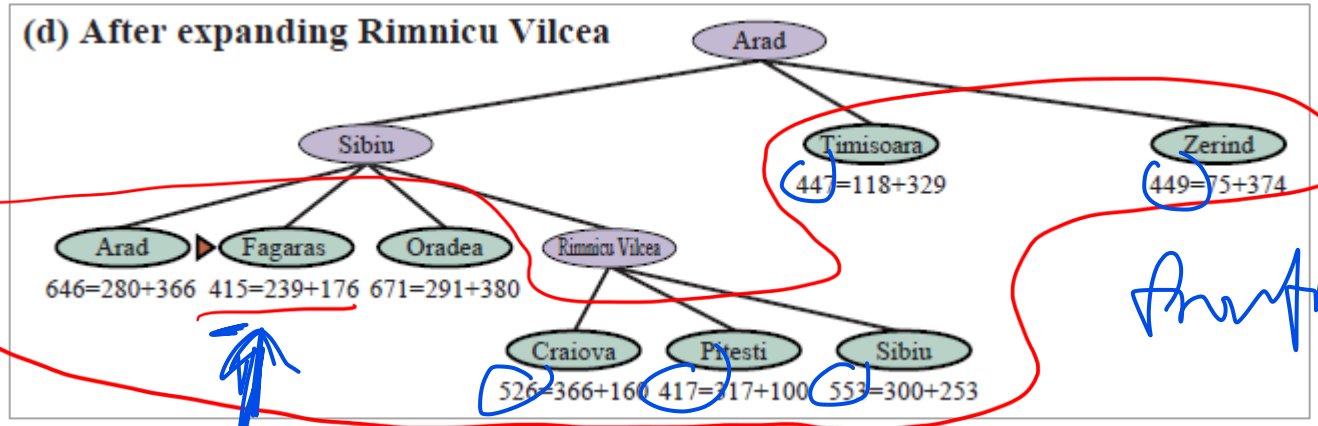


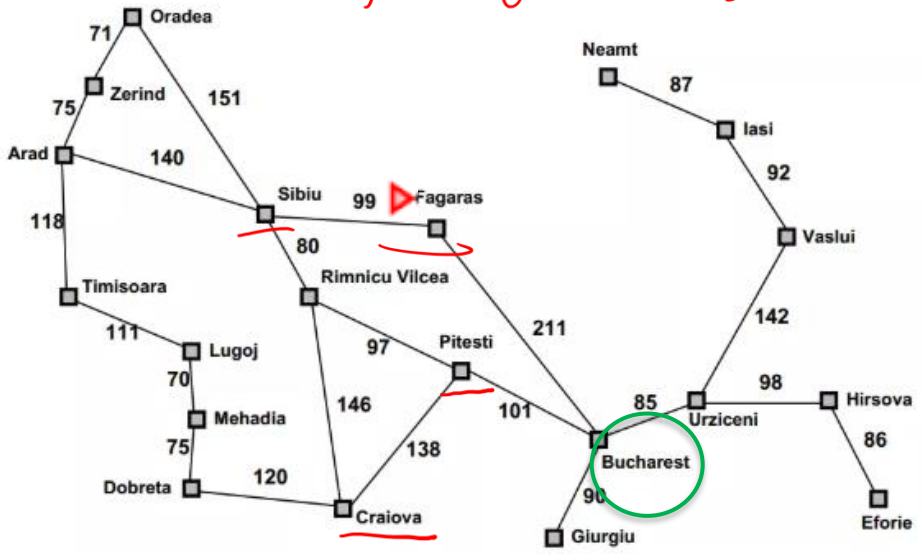
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Example: Path planning with A*



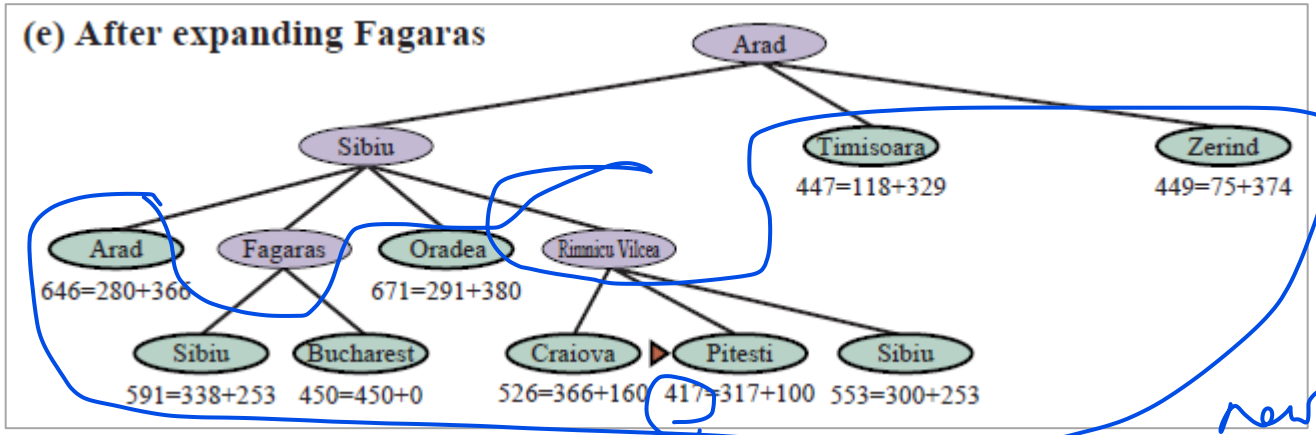


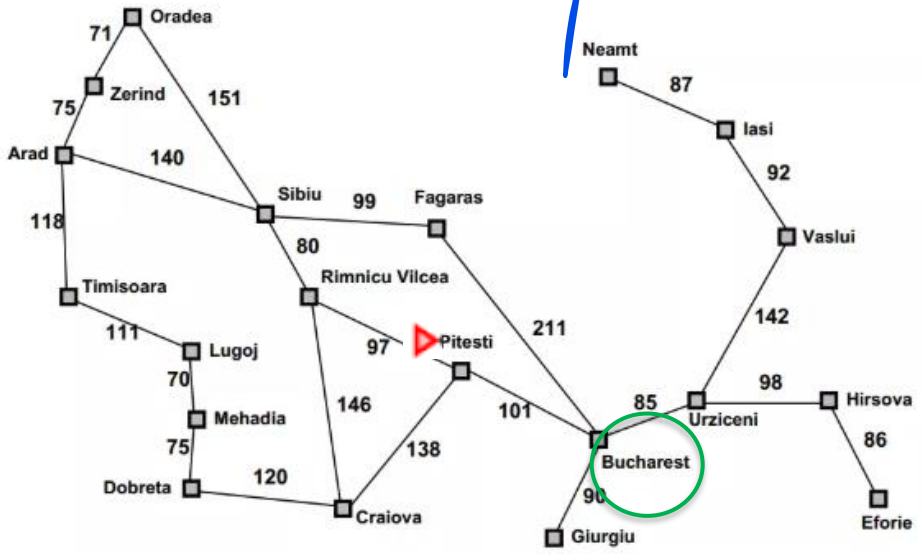
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Example: Path planning with A*



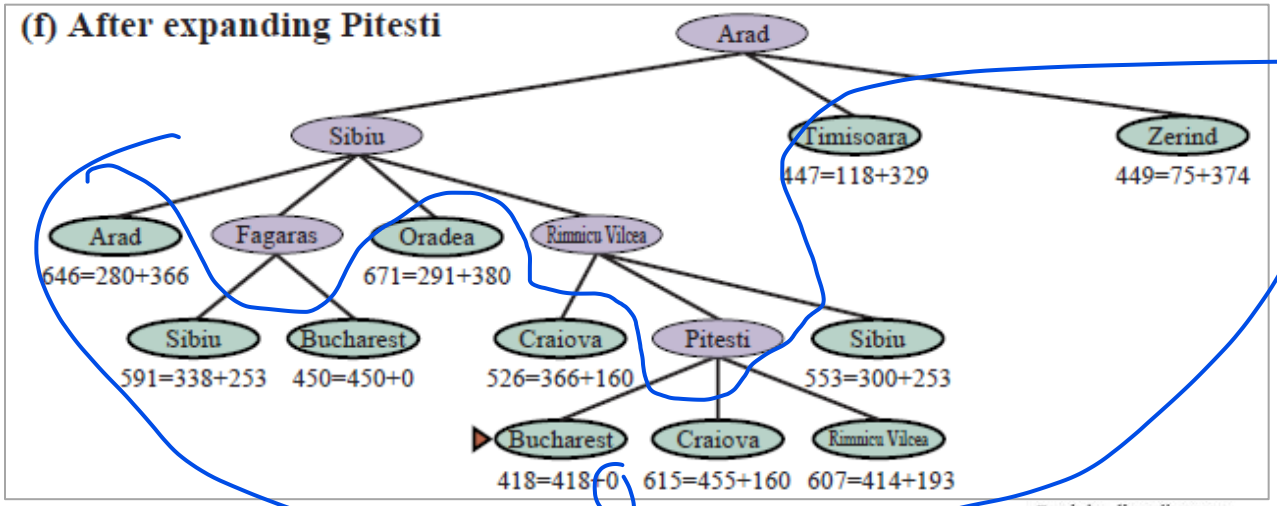


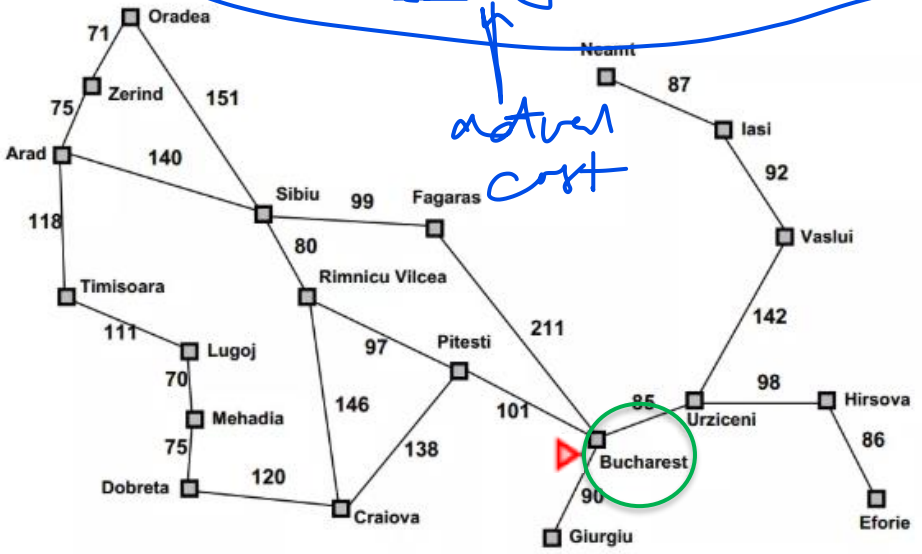
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Example: Path planning with A*





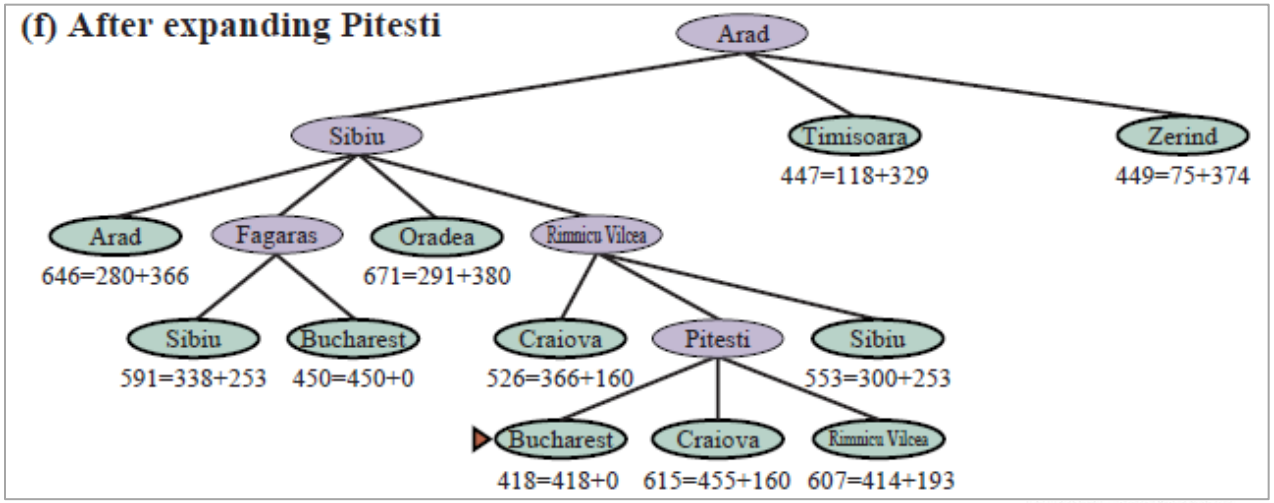
Straight-line distance to Bucharest

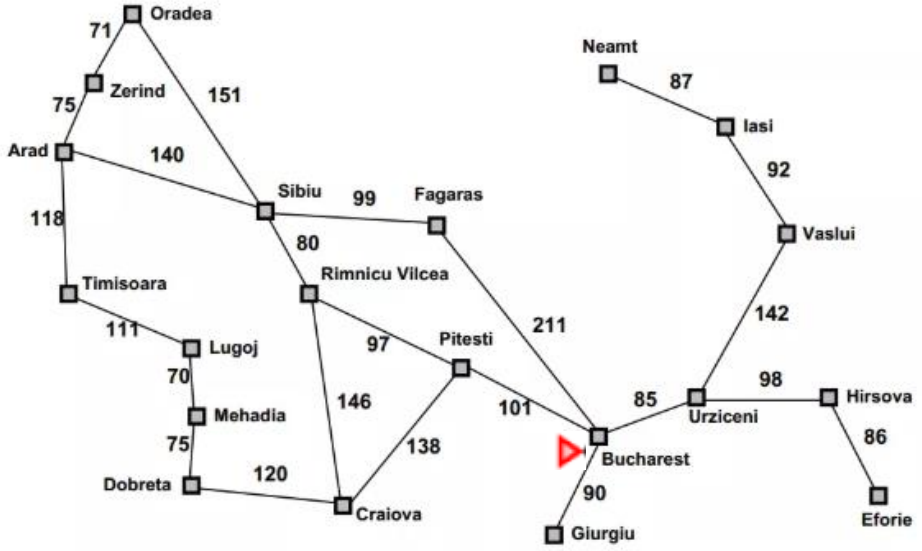
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Example: Path planning with A*





Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(n)$

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P105

Quick question 1

Which data structure is most suitable for managing the frontier in A* search?

A. Queue (FIFO)

B. Stack (LIFO)

C. Priority queue

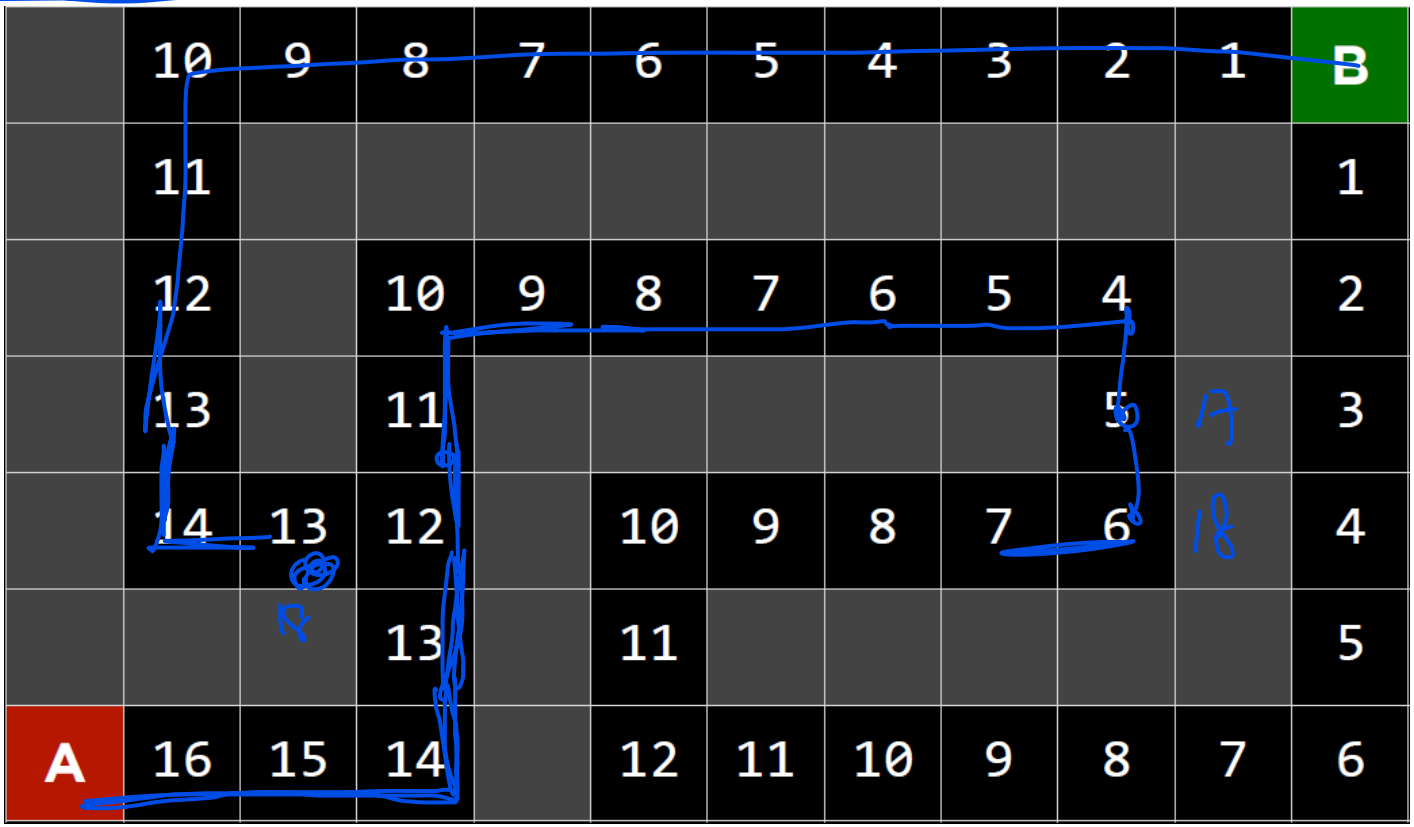
the greedy is now f(n) = g(n) + h(n)

dijkstra is only g(n)

GBFS is only h(n)

Quick question 2

In this maze, each grid is labeled with its Manhattan distance to the goal as a heuristic. Use A* search to find the shortest path from A to B, where $g(n)$ is the actual Manhattan distance from the start to node n.



A* Search

	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		10	9	8	7	6	5	4		2
	13		11	f(n)=6+11 = 17					5		3
	14	13	5+12		10	9	8	7	6		4
f(n)=6+13 = 19			4+13		11						5
A	1+16	2+15	3+14		12	11	10	9	8	7	6

A* Search

	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		7+10	8+9	9+8	10+7	11+6	12+5	13+4		2
	13		6+11						14+5		3
	14	13	5+12		10	9	8	7	6		4
f(n)=6+13 = 19			4+13		11				f(n)=15+6 = 21		5
A	1+16	2+15	3+14		12	11	10	9	8	7	6

A*
Sea
rch

A* Search

	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		7+10	8+9	9+8	10+7	11+6	12+5	13+4		2
	13		6+11						14+5		3
	14	6+13	5+12		10	9	8	7	6		4
			4+13		11						5
A	1+16	2+15	3+14		12	11	10	9	8	7	6

f(n)=7+14 = 21

f(n)=15+6 = 21

A* Search

	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		7+10	8+9	9+8	10+7	11+6	12+5	13+4		2
	13		6+11						14+5		3
f(n)=7+14 = 21	14	6+13	5+12		10	9	8	7	15+6		4
			4+13		11			f(n)=16+7 = 23			5
A	1+16	2+15	3+14		12	11	10	9	8	7	6

g(n) h(n)

A* Search

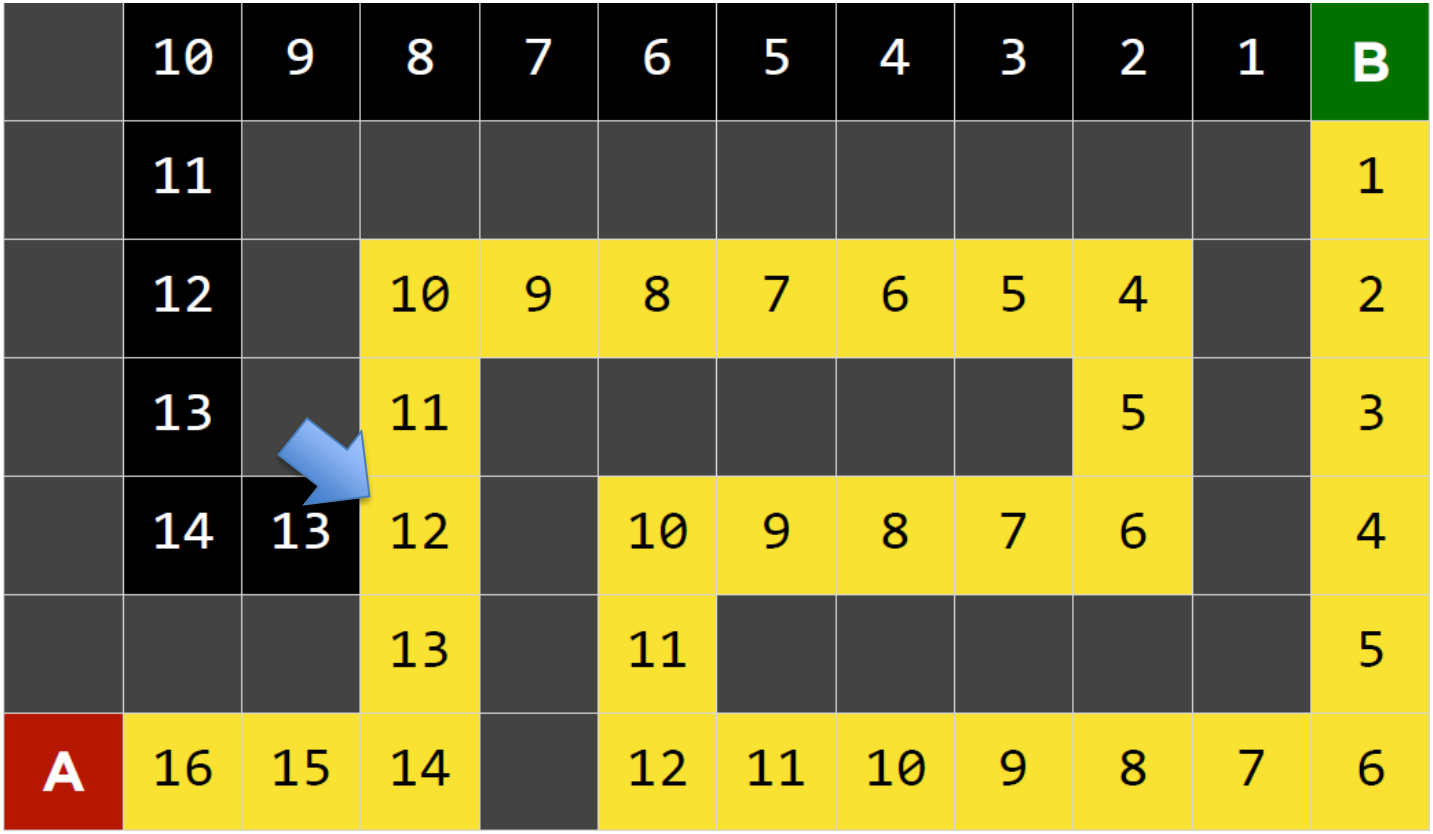
	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		7+10	8+9	9+8	10+7	11+6	12+5	13+4		2
	13		6+11						14+5		3
	7+14	6+13	5+12		10	9	8	7	15+6		4
			4+13		11			f(n)=16+7 = 23			5
A	1+16	2+15	3+14		12	11	10	9	8	7	6

Quick question 1

A* search found the correct shortest path:

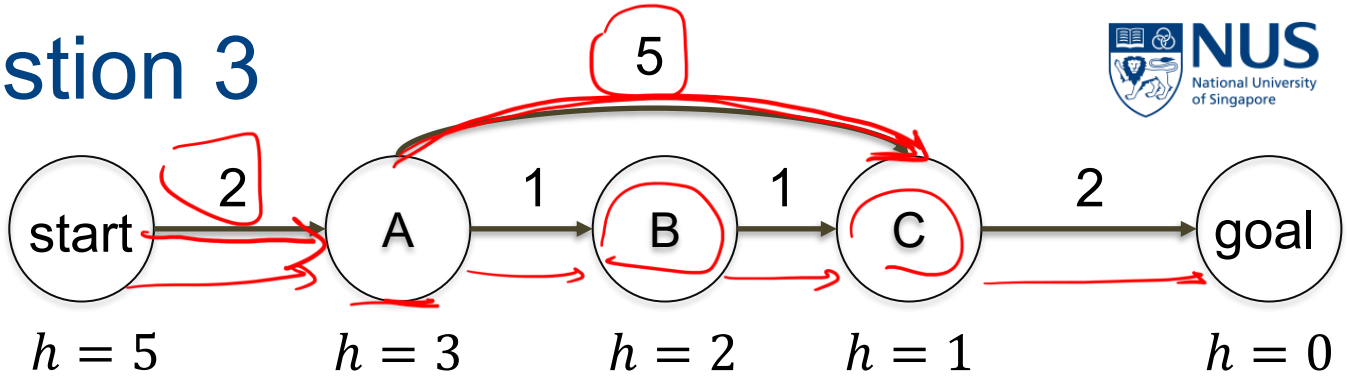
	10	9	8	7	6	5	4	3	2	1	B
	11										1
	12		10	9	8	7	6	5	4		2
	13		11						5		3
	14	13	12		10	9	8	7	6		4
			13		11						5
A	16	15	14		12	11	10	9	8	7	6

Greedy best-first search failed to find the correct shortest path.



The problem isn't just the heuristic — it's the nature of greedy best-first search itself!

Quick question 3



Start at start (f =5):

•Neighbors: A (f=5)

•Frontier: **[A(f=5)]**.

Expand A (f=5):

•Neighbors: B (f=5), C (f=8)

•Frontier: **[B(f=5), C(f=8)]**

Expand B (f=5):

•Neighbors: C (f=5).

•Frontier: **[C(f=5)]**

•The path so far is **start -> A -> B -> C**.

Expand C (f=5):

•C's only neighbor is **goal** (h=0).

•The goal state, **goal**, has a heuristic of 0. The algorithm chooses goal next.

•The path found is **start -> A -> B -> C -> goal**, which is the optimal cost

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

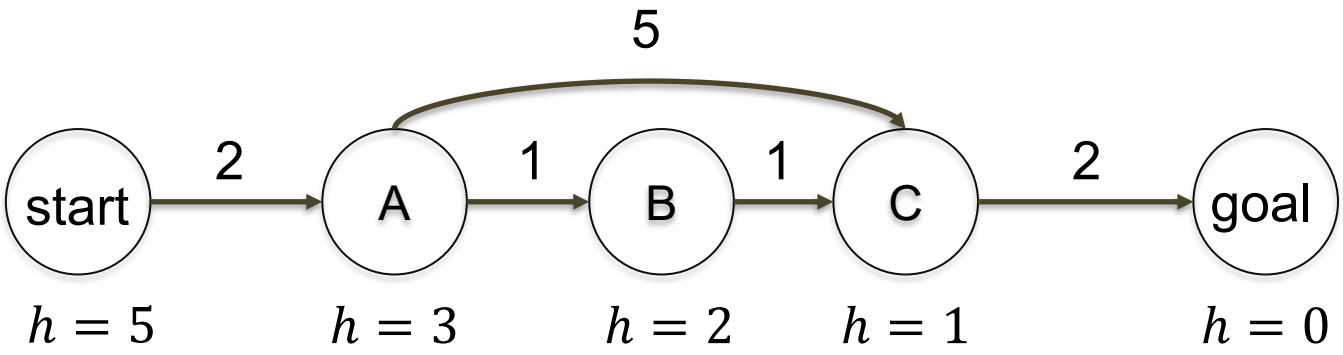
$2 + 1 + 2$

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

$= 2 + 5 + 1$

$= 8$

Compare with greedy best-first search:



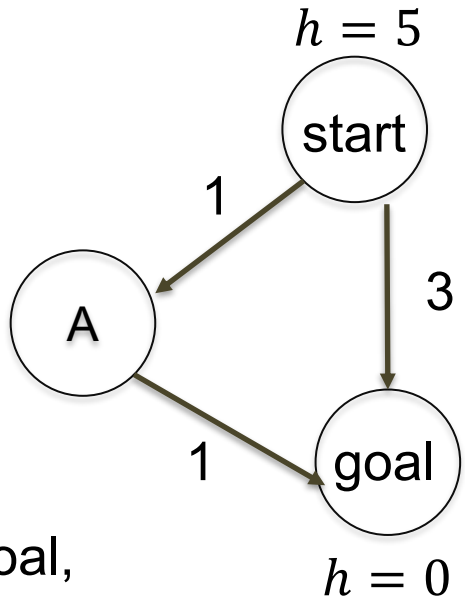
Greedy best-first search: start → A → C → goal, path cost = 9

A* search: start → A → B → C → goal, path cost = 6, which is the optimal cost

*Greedy best-first search uses only the heuristic to pick the node that seems closest to the goal, ignoring the actual cost from the start. In contrast, A** search considers both the actual path cost and the heuristic.

Quick question 4

Whether A* is optimal depends on how well the heuristics are designed, especially fulfilling the criteria on the next page



Given $h(A) = 1$, A* chooses the path start → A → goal, resulting in a total cost of 2

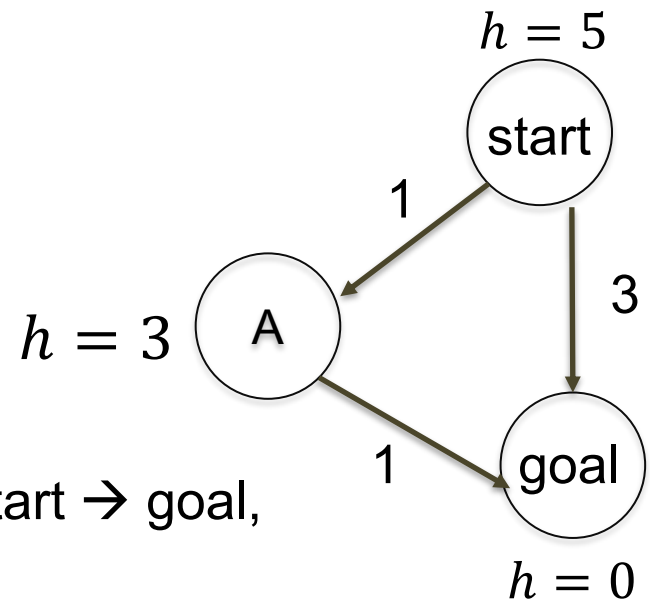
Given $h(A) = 2$, A* chooses either the path start → A → goal or start → goal

Given $h(A) = 3$, A* chooses the path start → goal, resulting in a total cost of 3

Optimal solution: start → A → goal, total cost = 2

Can we put conditions on the choice of heuristic to guarantee optimality?

Example when heuristic function can overestimates the true cost:



Given $h(A) = 3$, A* chooses the path start → goal, resulting in a total cost of 3

However, this is not the optimal path—A* fails due to the heuristic being inadmissible ($h(A) > h^*(A)$).

We should design heuristics to be optimistic—that is, they never overestimate the true cost to reach the goal. This property defines ***admissibility.***

Admissible heuristics

if heuristic are all admissible, A* is guaranteed to return optimal ONLY IF reexpansions are handled properly. (closed off nodes that not reopened, but those closed ones may return better path). Hence for ONLY FOR TREE SEARCH, admissible can guarantee, but if using graph search, nodes are not revisited, so need to see consistent (stricter)
Also note that non-admissible heuristic does not guarantee that A* will return a non-optimal solution

A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

since dijkstra is guaranteed to return an optimal solution, if non-admissible is used, run dijkstra to check if A Star Cost = Optimal Cost

Admissible heuristics are optimistic: they never overestimate the cost to reach the goal, always estimating it as less than or equal to the actual cost.

Example:
Driving from City A to B
•Actual road distance: 100 km
•Straight-line (Euclidean) distance as heuristic: 80 km
The heuristic *never overestimates* → **Admissible**

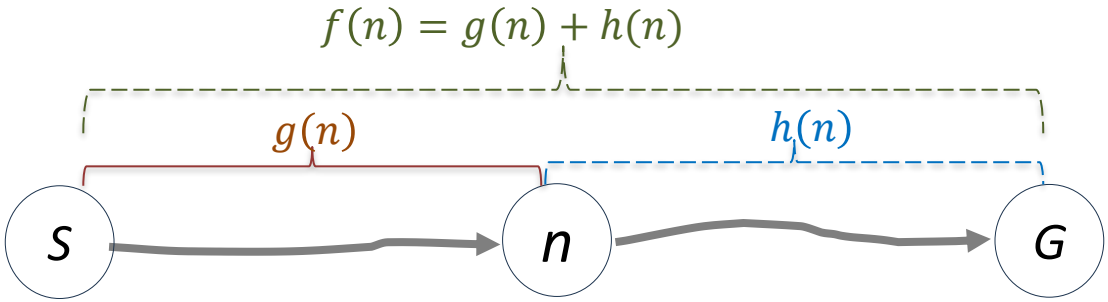
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P104 – P106

A*: Where *heuristics* meet *reality*

It evaluates nodes by combining $g(n)$, the actual cost to reach the node, and $h(n)$, the estimated cost to get from the node to the goal.

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

$f(n)$: *Estimated* total cost of the *cheapest path from the start to the goal via node* n



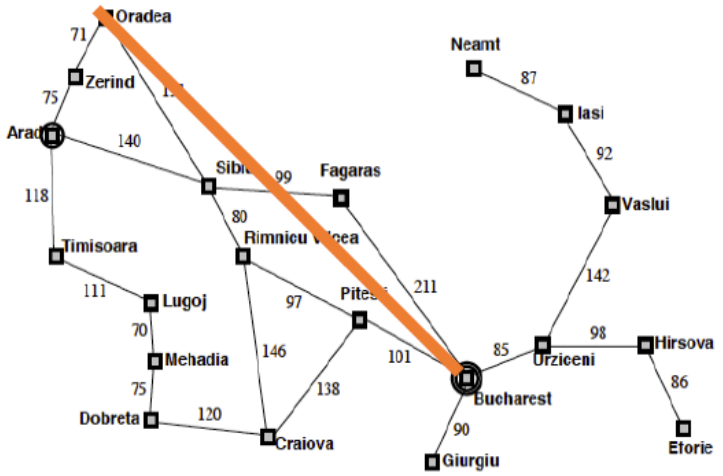
$g(n)$: Actual cost of the path from the start node to the current node n. (for start node: $g(n) = 0$)

$h(n)$: Estimated cost of the cheapest path from the current node n to the goal. (for goal node: $h(n) = 0$)

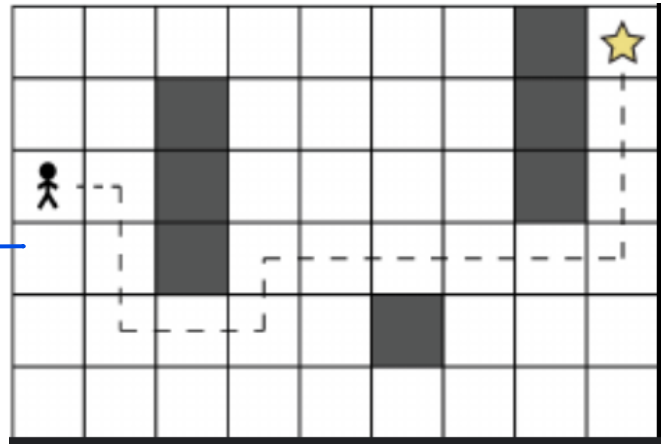
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P103 – P109

Examples of admissible heuristics

In navigation, straight line (Euclidean) distance -- $h_{ED}(n)$, never overestimates the actual road distance



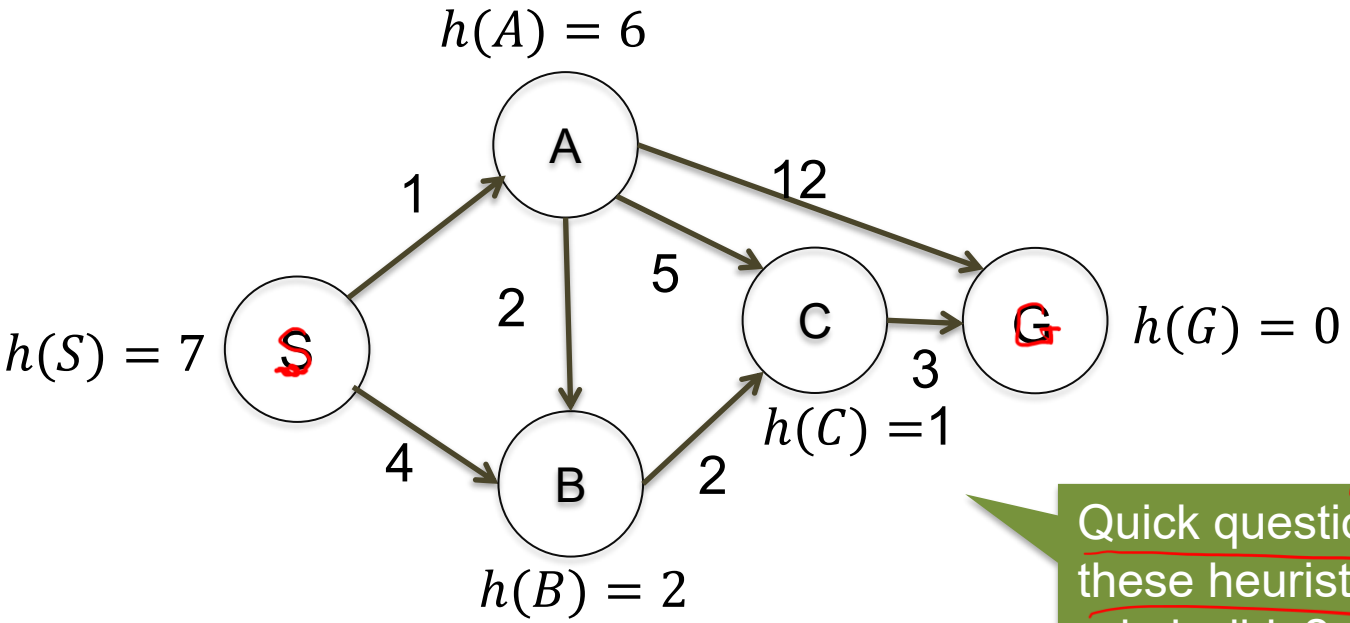
In grid worlds with 4-directional moves, Manhattan distance -- $h_{MD}(n)$, never overestimates the actual cost (actual path ≥ sum of horizontal + vertical steps)



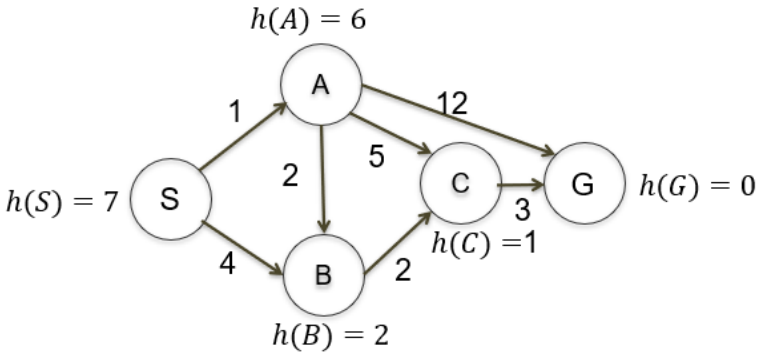
Class practice 1

Which path does A* search return using tree search? The starting node is S and goal node is G.

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

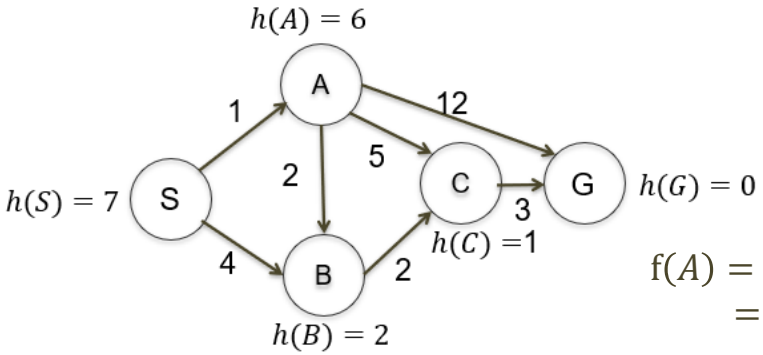


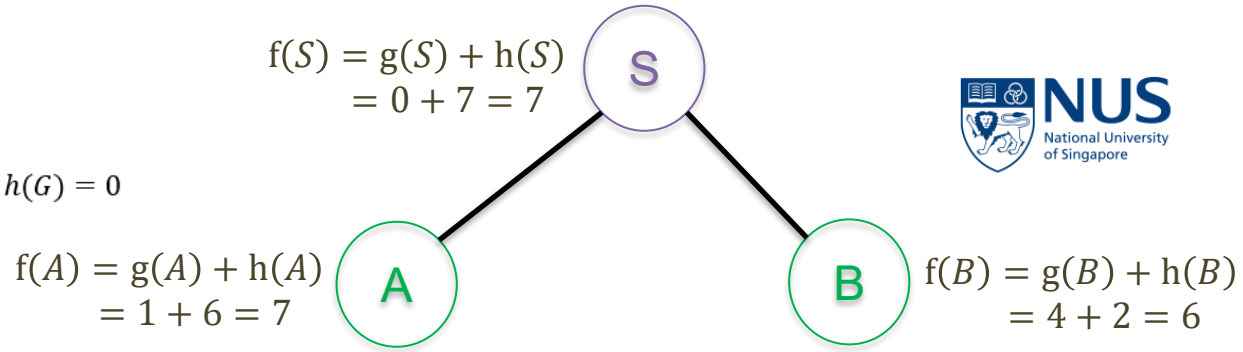
Quick question: Are these heuristics admissible?



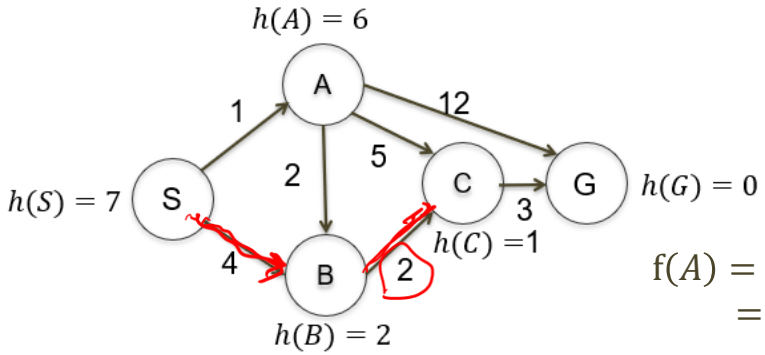
$$f(S) = g(S) + h(S)$$
$$= 0 + 7 = 7$$

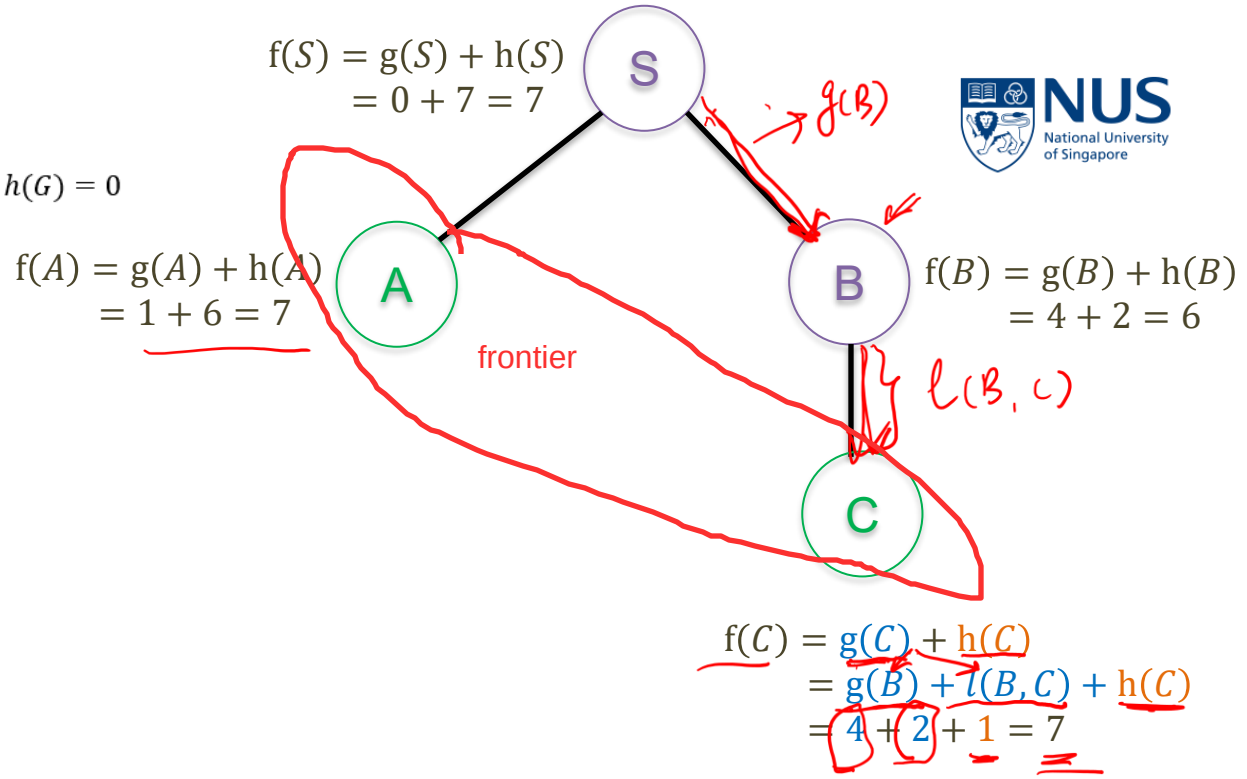
S



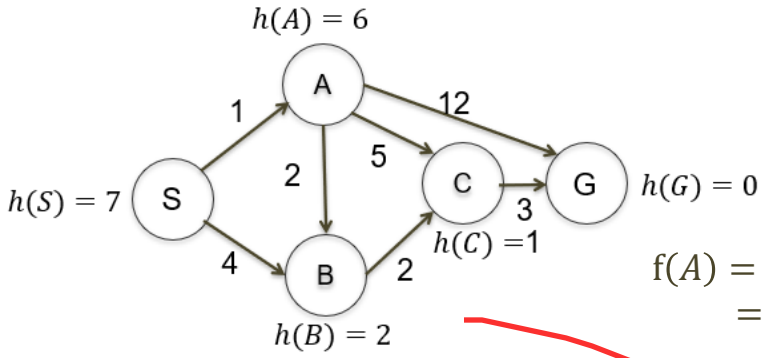


Lavender: Nodes that have been expanded
Green: nodes currently on the **frontier**, waiting to be expanded.

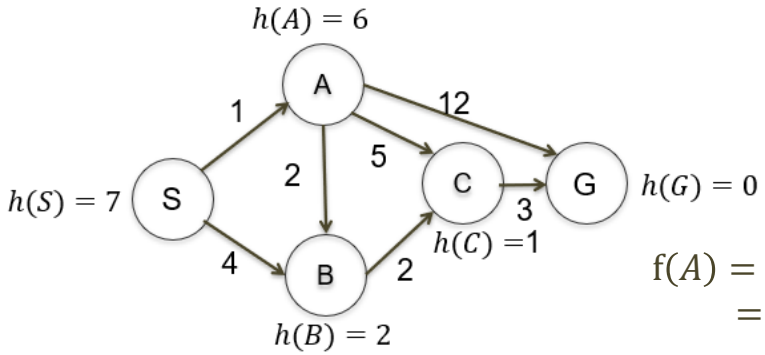


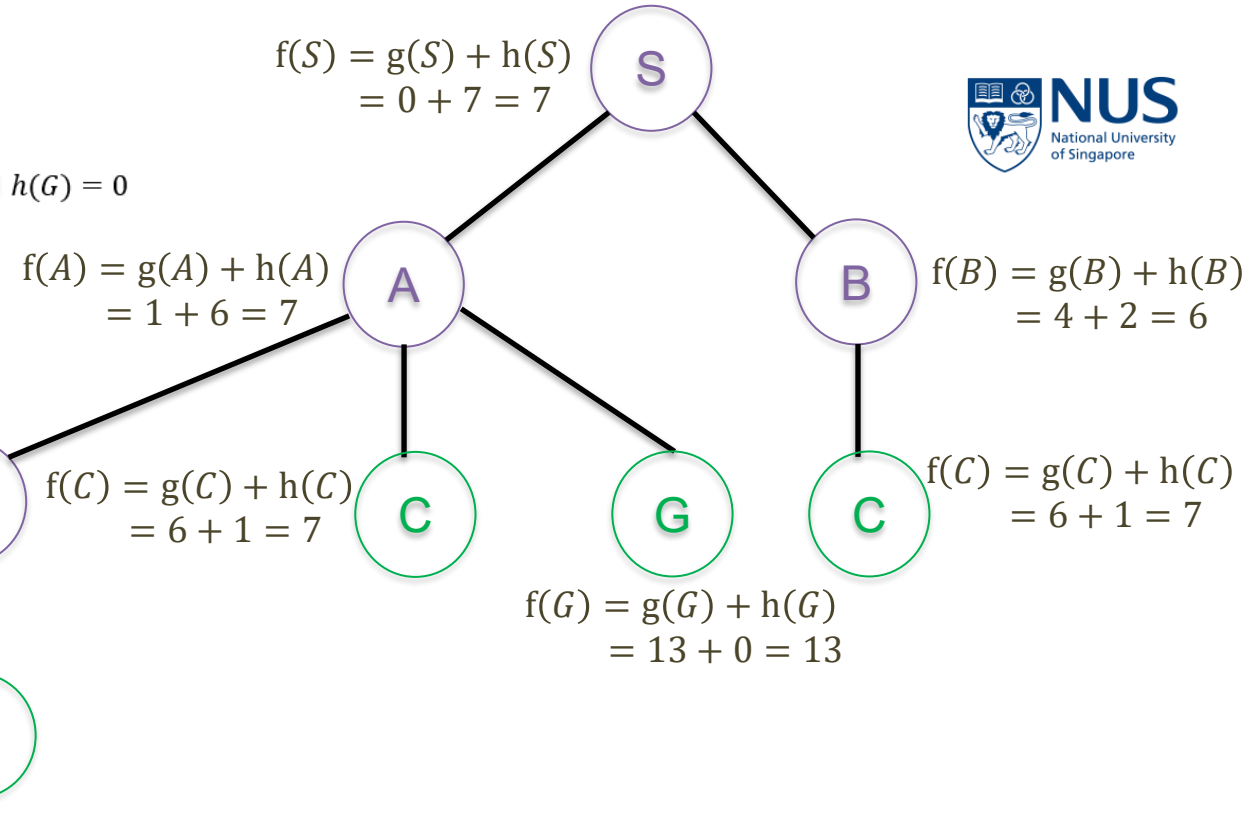


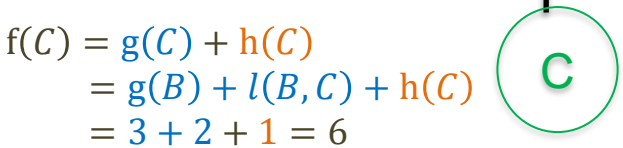
Lavender: Nodes that have been expanded
Green: nodes currently on the **frontier**, waiting to be expanded.

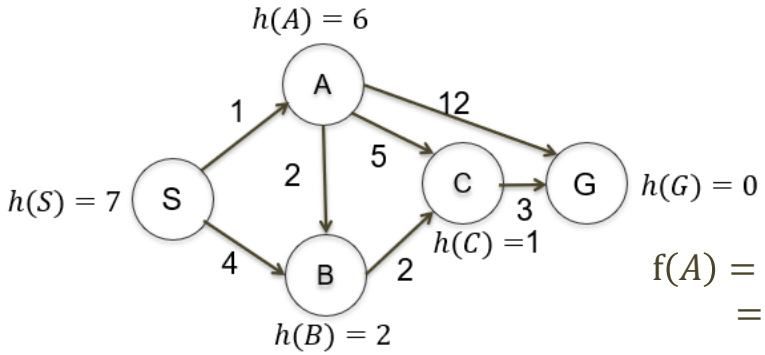


this is revisited, so admissible (via tree search) guarantees optimal



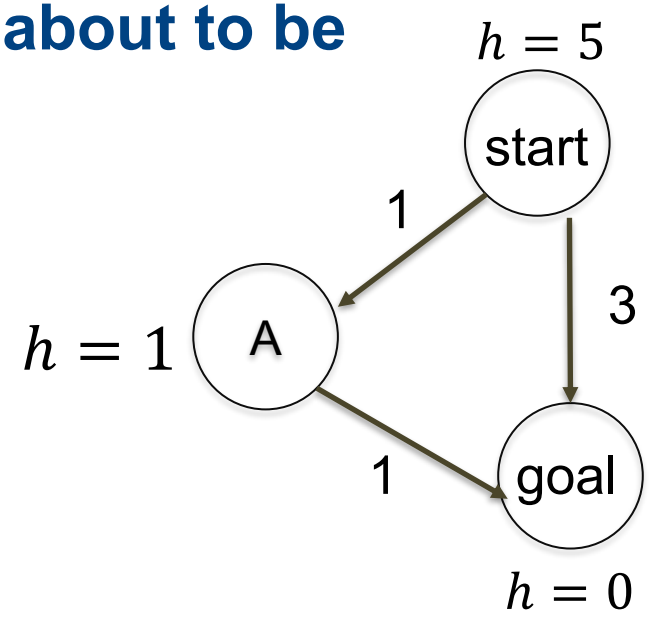


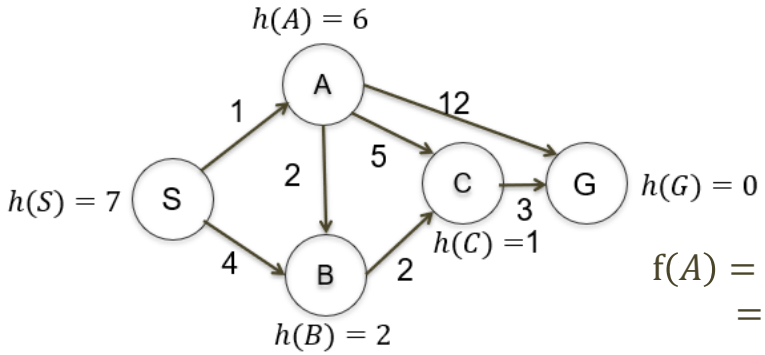


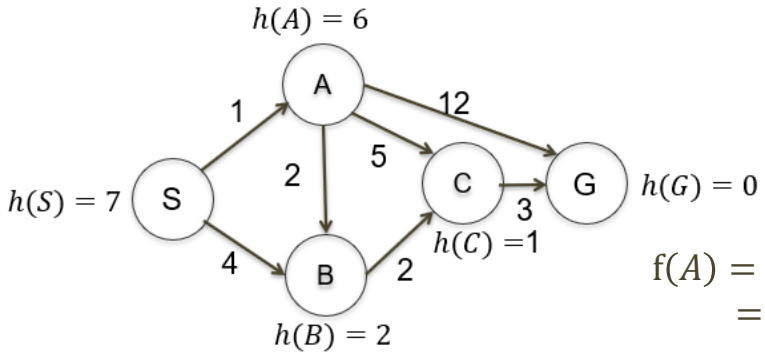


We only perform the goal test *when expanding a node*, and not *when adding it to the frontier*.

Because **we can't be sure that we've found the *shortest path* to the goal until it's about to be expanded.**







$f(S) = g(S) + h(S)$
$= 0 + 7 = 7$

S

$f(A) = g(A) + h(A)$
$= 1 + 6 = 7$

A

B

$f(B) = g(B) + h(B)$
$= 4 + 2 = 6$

$f(B) = g(B) + h(B)$
$= 3 + 2 = 5$

B

$f(C) = g(C) + h(C)$
$= 6 + 1 = 7$

C

G

C

$f(C) = g(C) + h(C)$
$= 6 + 1 = 7$

$f(G) = g(G) + h(G)$
$= 13 + 0 = 13$

$f(C) = g(C) + h(C)$
$= 5 + 1 = 6$

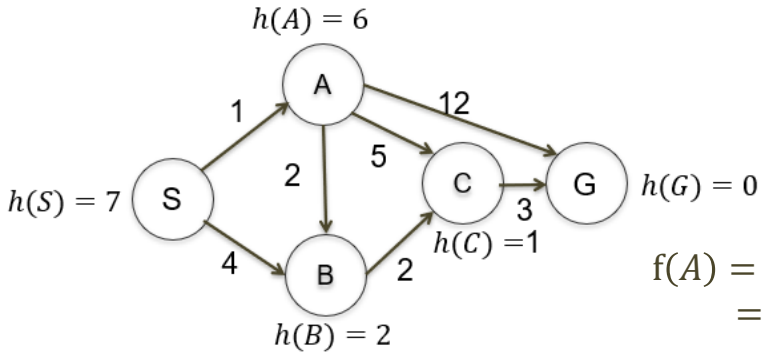
C

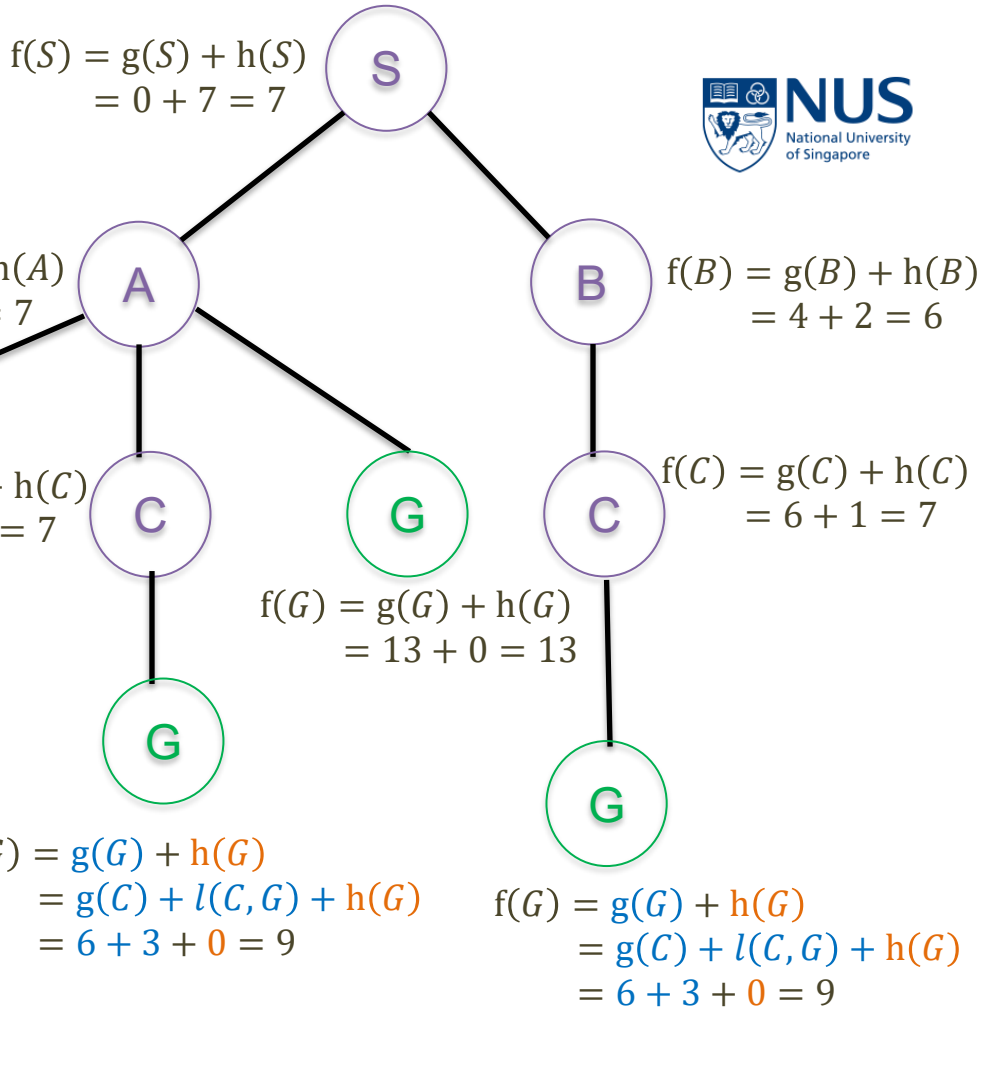
G

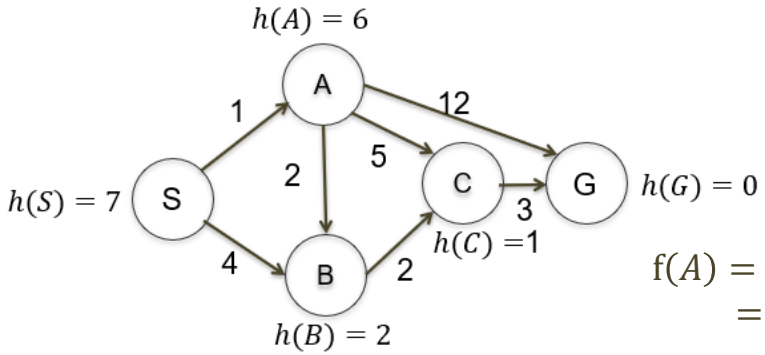
$f(G) = g(G) + h(G)$
$= g(C) + l(C, G) + h(G)$
$= 6 + 3 + 0 = 9$

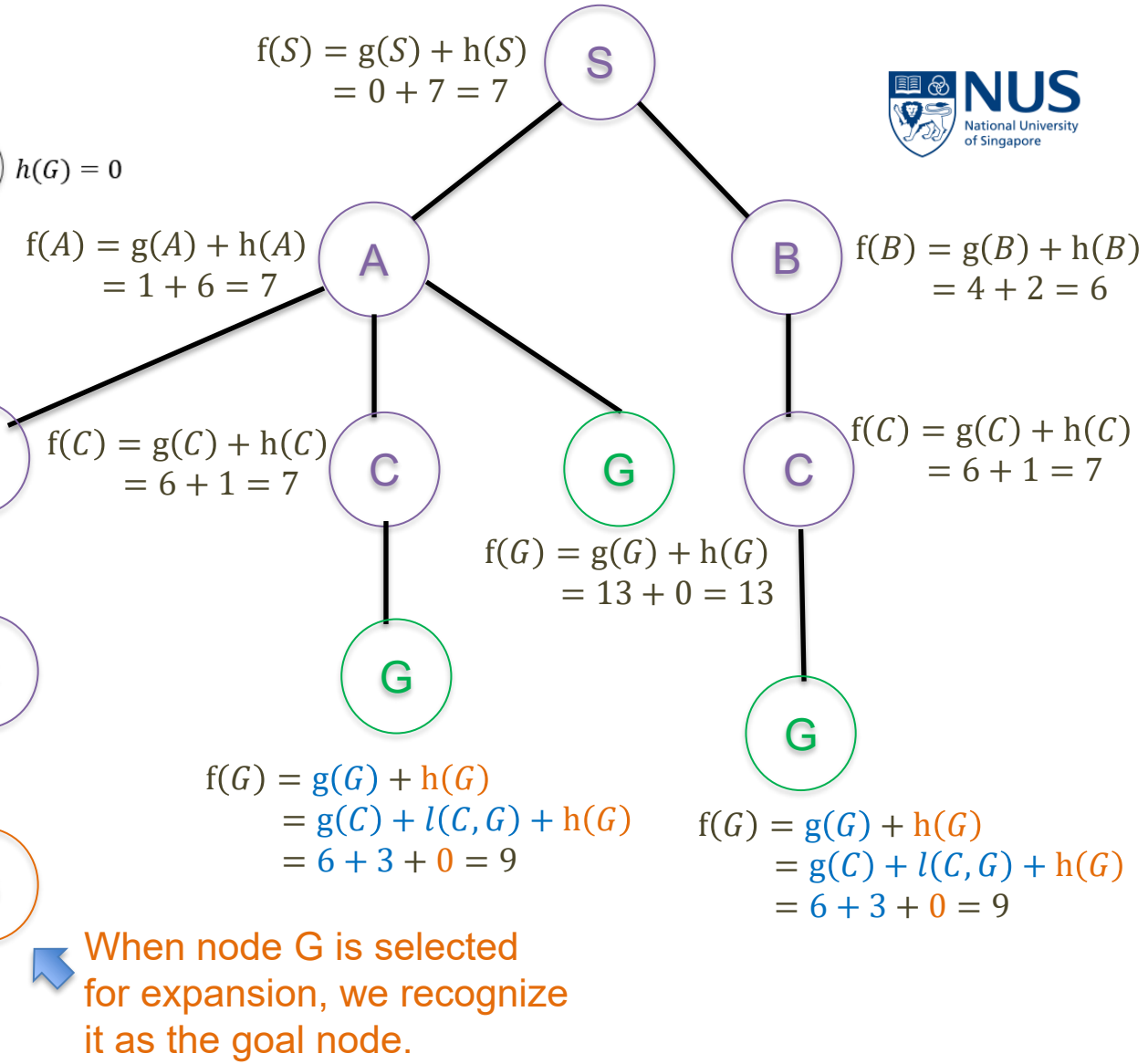
$f(G) = g(G) + h(G)$
$= g(C) + l(C, G) + h(G)$
$= 5 + 3 + 0 = 8$

G







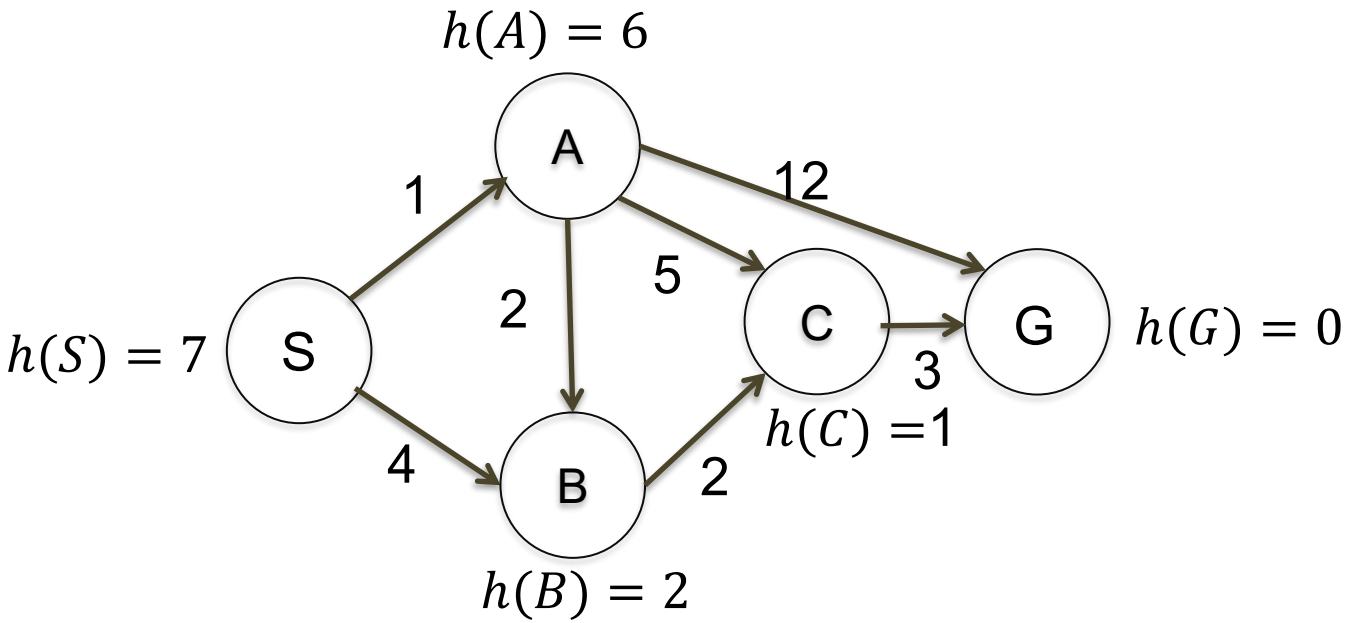


Optimal path: S → A → B → C → G with a total cost of 8

Class practice 2

Which path does A* search return using graph search? The starting node is S and goal node is G.

NO REVISIT!!!



Tree search expands nodes without checking for duplicates, potentially revisiting the same state multiple times, whereas graph search keeps track of explored states to avoid revisiting them.

Consistent heuristics

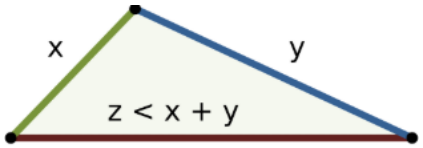
this is needed for graph search using A* to guarantee optimal

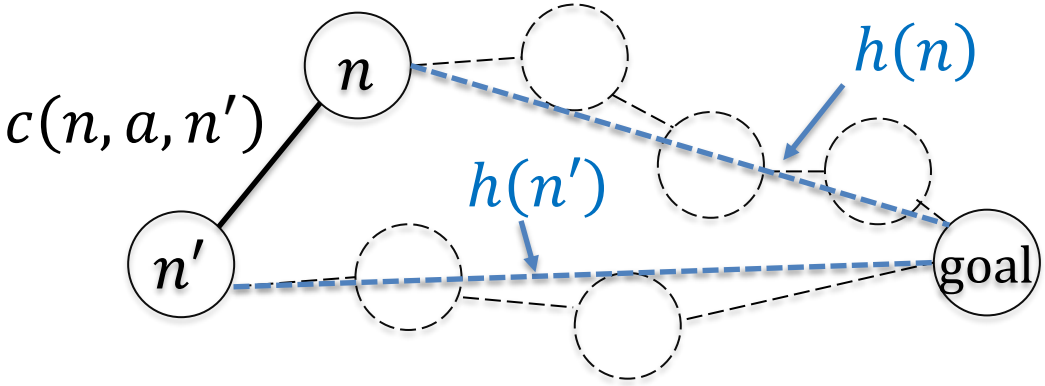
A heuristic $h(n)$ is consistent (or monotonic) if for every node n, every successor n' of n generated by any action a:

$$h(n) \leq c(n, a, n') + h(n')$$

$c(n, a, n')$ is the actual cost of applying action a to go from state n to n'

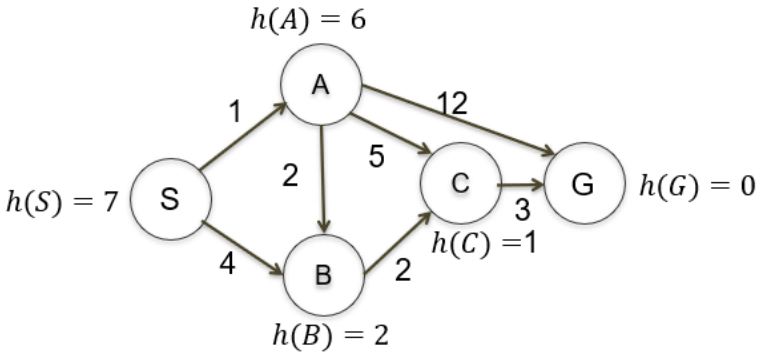
Consistency is a version of triangle inequality.





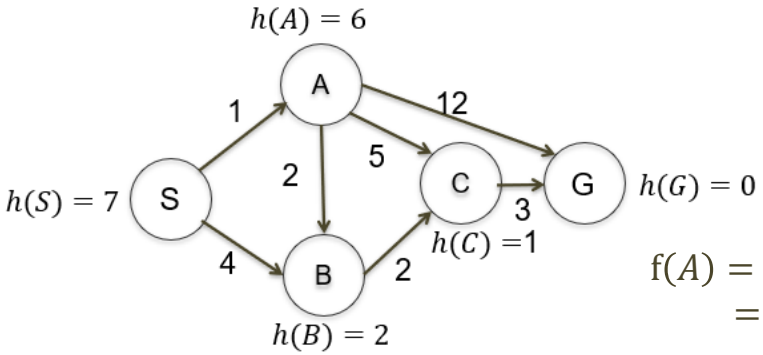
Consistency is a **stricter requirement** than admissibility.

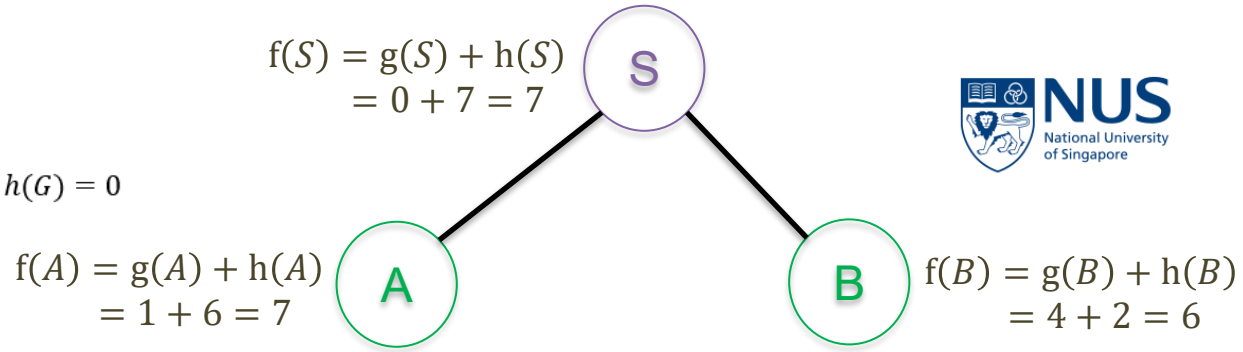
Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P106



$$f(S) = g(S) + h(S)$$
$$= 0 + 7 = 7$$

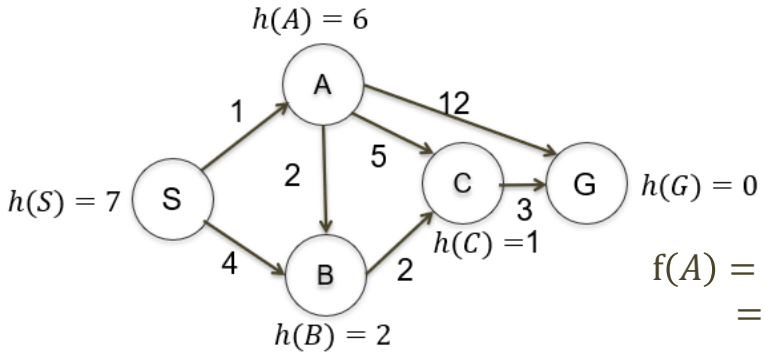
S

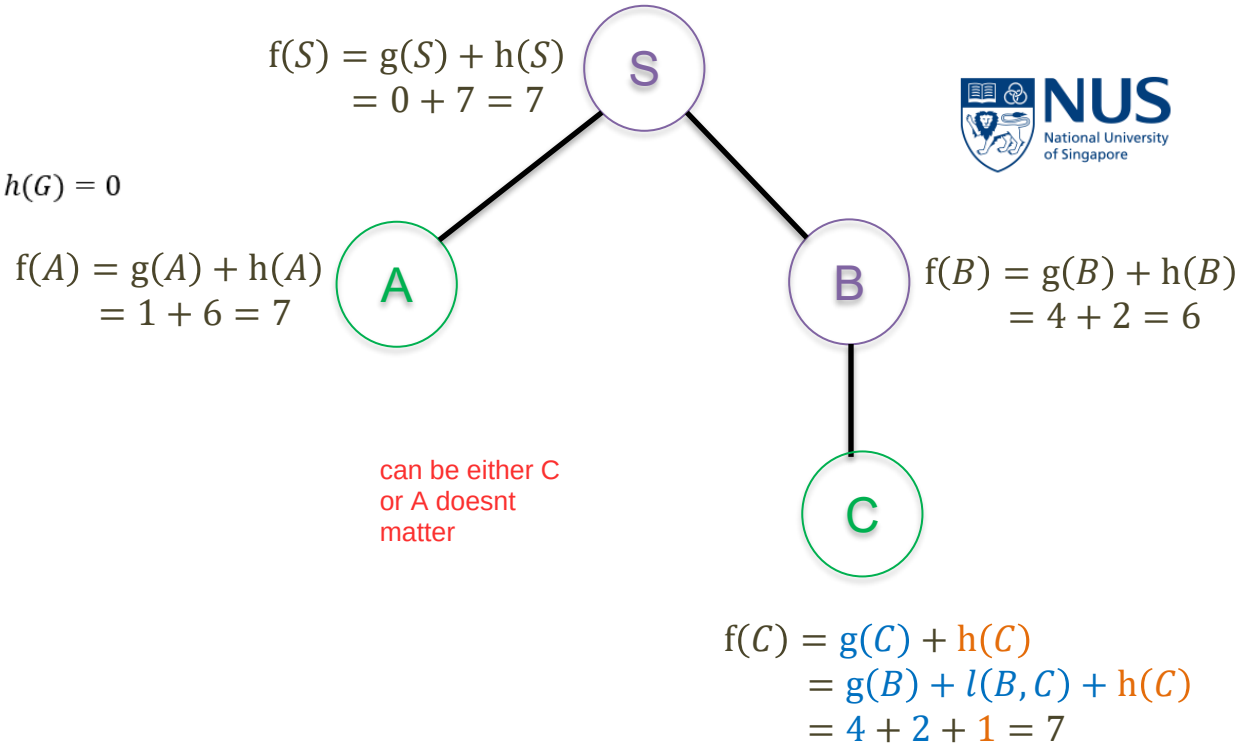




Lavender: Nodes that have been expanded (in the **explored set**)
Green: nodes currently on the **frontier**, waiting to be expanded.

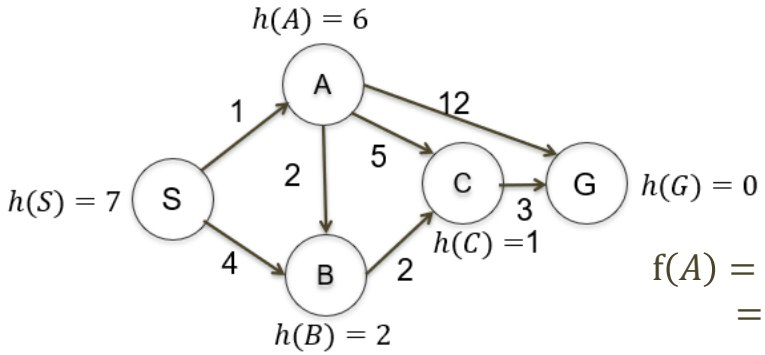
⇒ **Explored set = { S }**

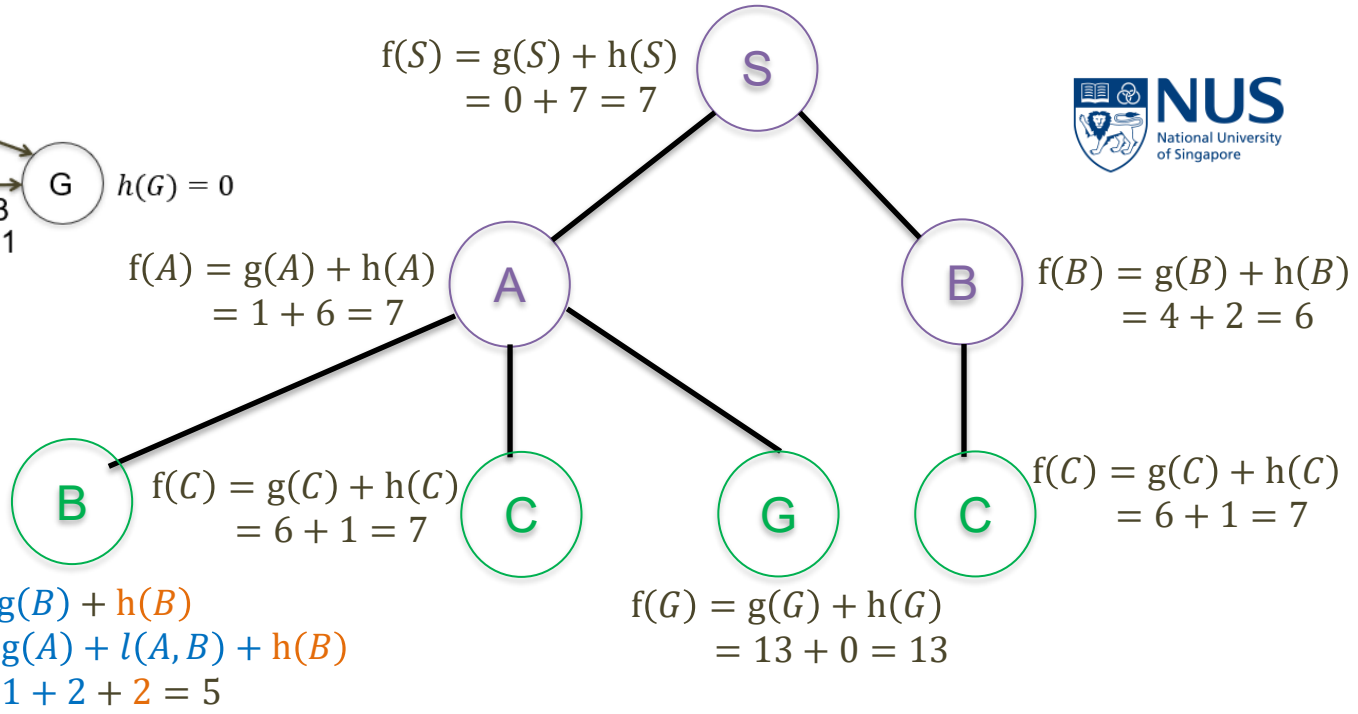




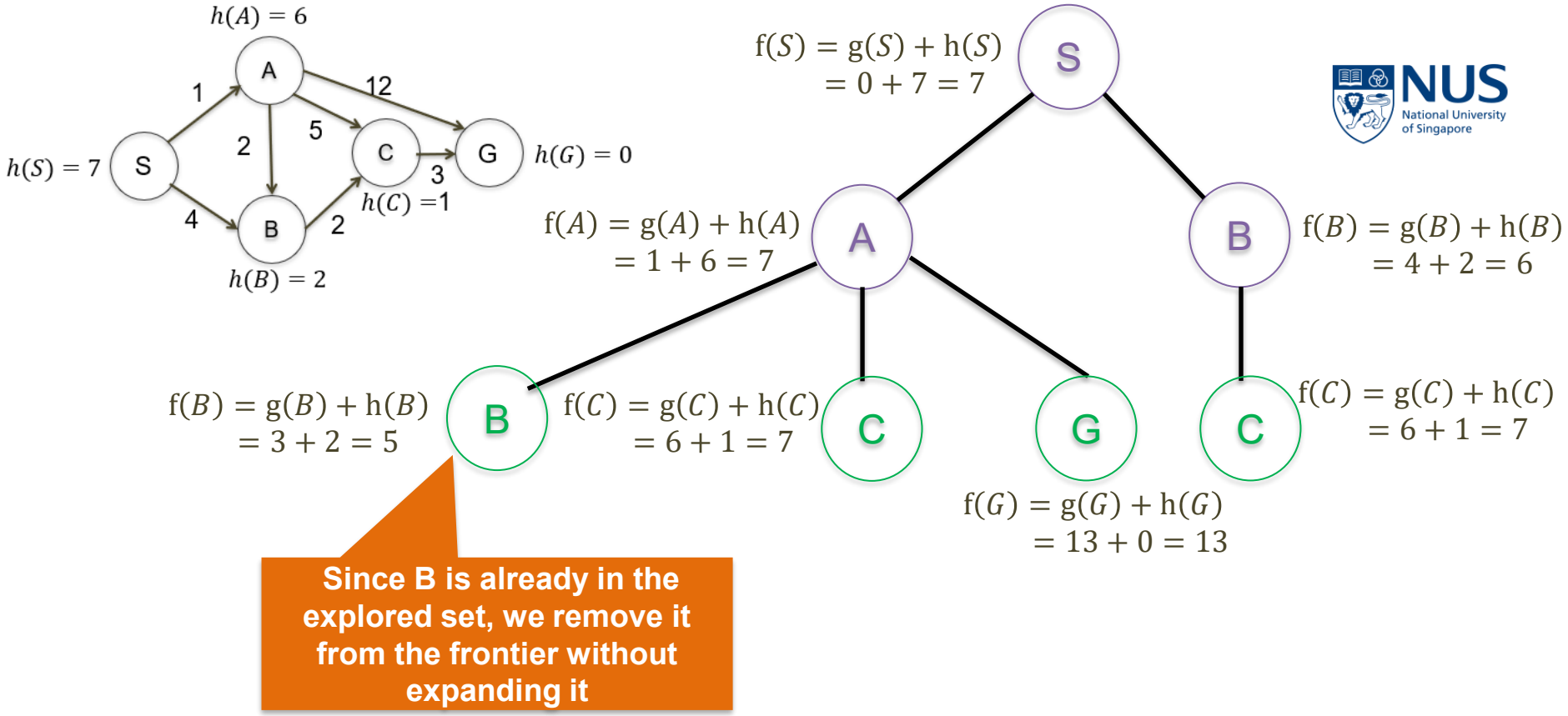
Lavender: Nodes that have been expanded (in the **explored set**)
Green: nodes currently on the **frontier**, waiting to be expanded.

Explored set = { S, B }



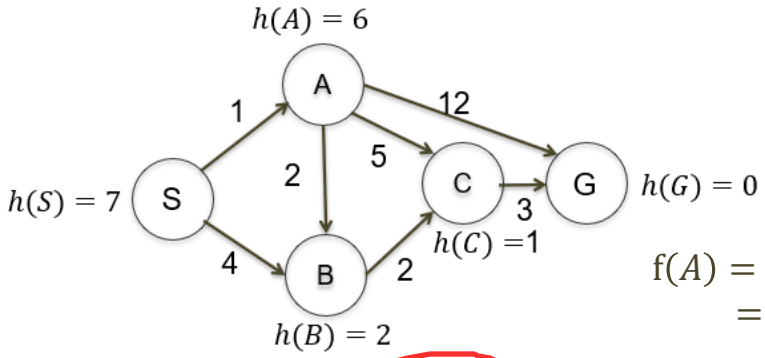


Explored set = { S, B, A }



Tree search expands nodes without checking for duplicates, potentially revisiting the same state multiple times, whereas graph search keeps track of explored states to avoid revisiting them.

Explored set = { S, B, A }

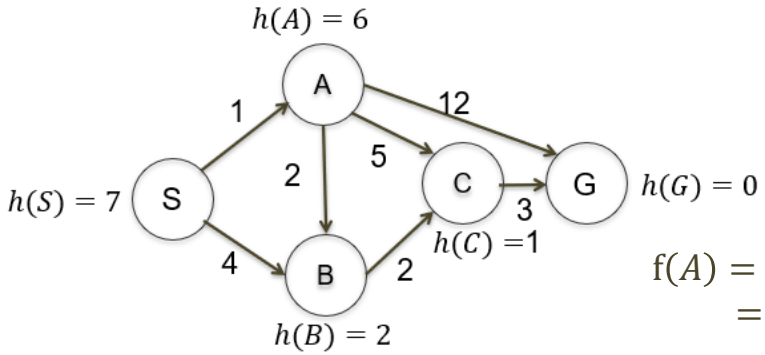


Since B is already in the explored set, we remove it from the frontier without expanding it

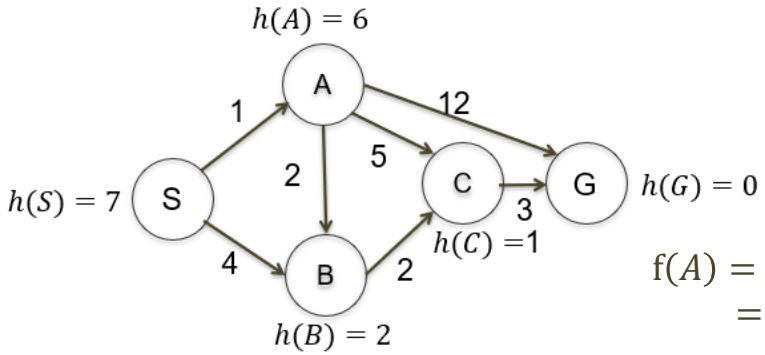
this is the problem for graph. No revisit so admissible is not enough to guarantee optimal, need consistent too.

Explored set = { S, B, A }

Lavender: Nodes that have been expanded (in the **explored set**)
Green: nodes currently on the **frontier**, waiting to be expanded.
Brown: nodes removed from the frontier without expansion because they had already been expanded earlier and are in the explored set.



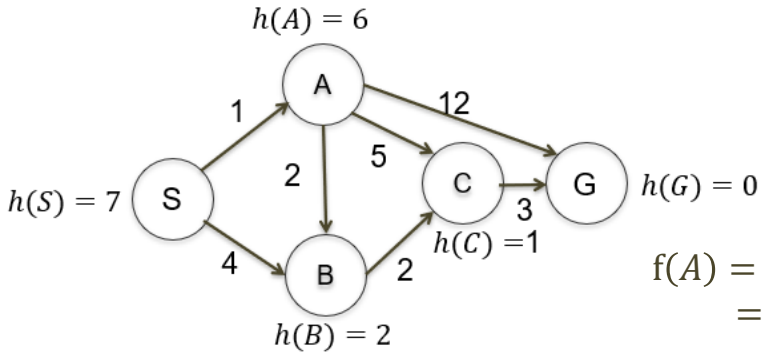
Explored set = { S, B, A, C }



Since C is already in the explored set, we remove it from the frontier without expanding it

Lavender: Nodes that have been expanded (in the **explored set**)
Green: nodes currently on the **frontier**, waiting to be expanded.
Brown: nodes removed from the frontier without expansion because they had already been expanded earlier and are in the explored set.

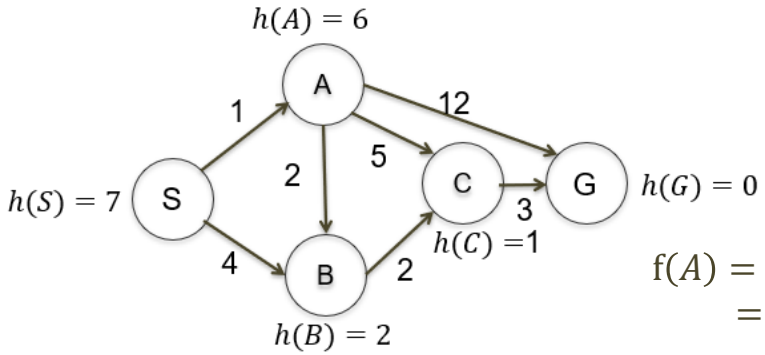
Explored set = { S, B, A, C }



Since C is already in the explored set, we remove it from the frontier without expanding it

Lavender: Nodes that have been expanded (in the **explored set**)
Green: nodes currently on the **frontier**, waiting to be expanded.
Brown: nodes removed from the frontier without expansion because they had already been expanded earlier and are in the explored set.

Explored set = { S, B, A, C }



$f(G) = g(G) + h(G)$
$= g(C) + l(C, G) + h(G)$
$= 6 + 3 + 0 = 9$

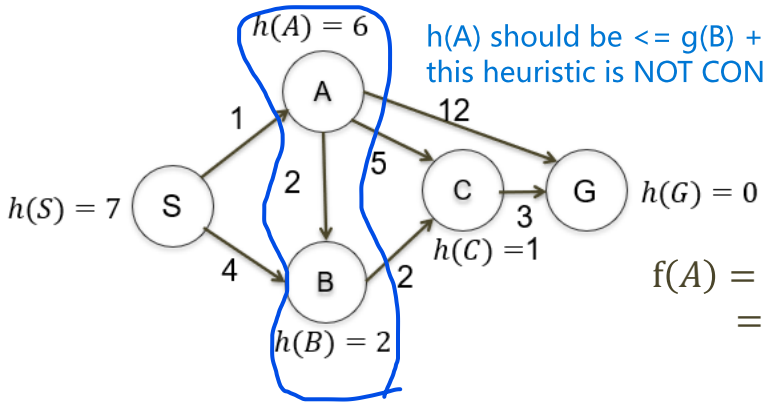
When node G is selected for expansion, we recognize it as the goal node.

Explored set = { S, B, A, C }

Not optimal!

Resulting path: S → A → C → G with a total cost of 9

Optimal path: S → A → B → C → G with a total cost of 8 can check using dijkstra



These heuristics are *admissible*, yet not *consistent*!

$$f(G) = g(G) + h(G)$$
$$= g(C) + l(C, G) + h(G)$$
$$= 6 + 3 + 0 = 9$$

When node G is selected for expansion, we recognize it as the goal node.

Explored set = { S, B, A, C }

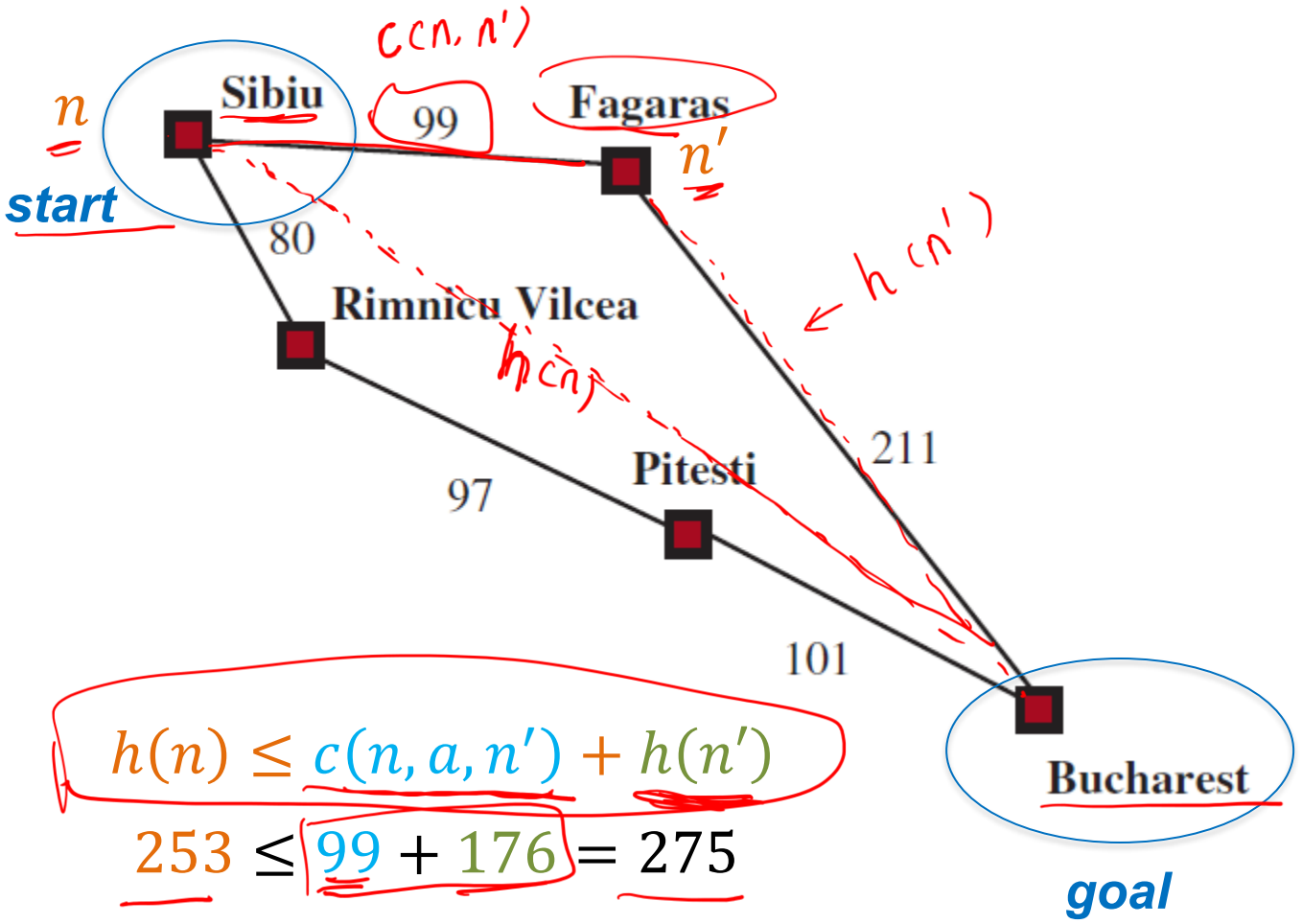
Not optimal!

Resulting path: S → A → C → G with a total cost of 9

Optimal path: S → A → B → C → G with a total cost of 8

Consistent heuristics: Example 1

Straight line (Euclidean) distance



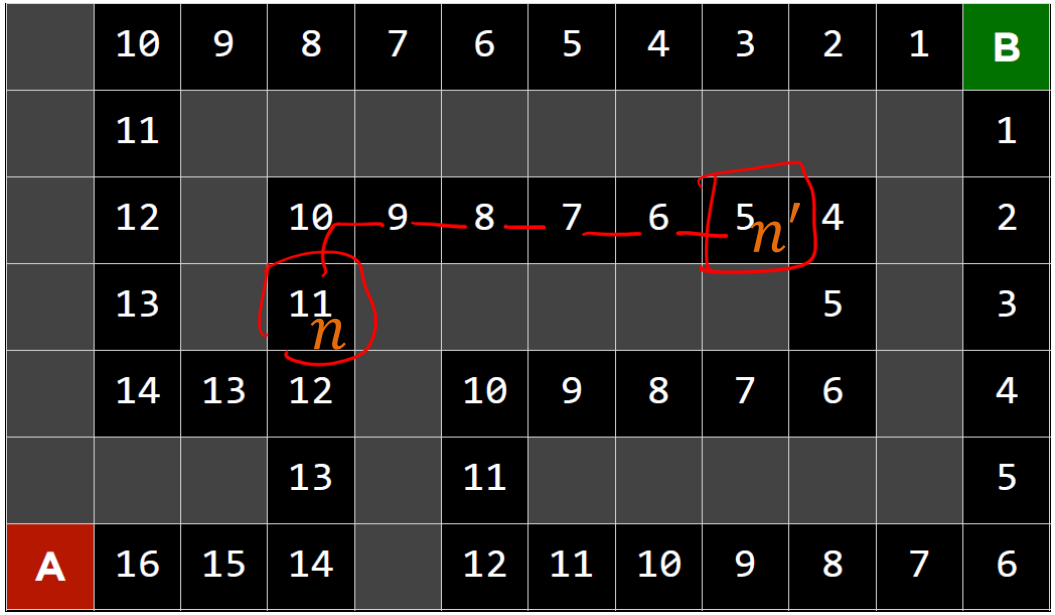
$$h(n) \leq c(n, a, n') + h(n')$$

$$253 \leq 99 + 176 = 275$$

h(n) = Straight Line Distances to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P106

Consistent heuristics: Example 2



goal

start

$$h(n) \leq c(n, a, n') + h(n')$$

$$11 \leq 6 + 5$$

Property of being consistent

If h is a consistent heuristic and all costs are non-zero, then $f(n)$ is **non-decreasing along any path**. (As you move toward the goal, the estimated total cost to reach it never decreases)

f = g + h, and h should not be changing

FOR GRAPH SEARCH, HEURISTICS MUST BE CONSISTENT
so that we can be sure of

With **graph search**, if the heuristic is consistent, A* never needs to **re-expand** a node. Once a node is removed from the frontier, the shortest path to that node has been *finalized* and will remain optimal.

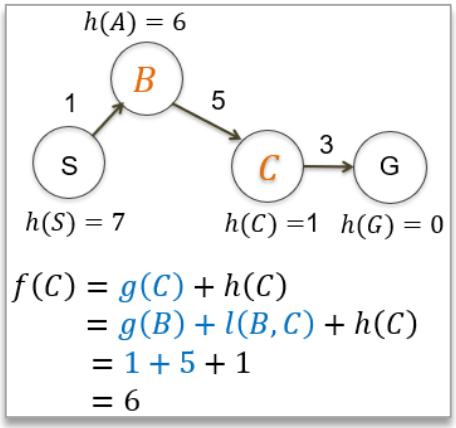
consistency (which includes admissible) guarantees optimal for graph search, since graph search dont reexpand a node

tree search will re expand nodes, thus it only needs admissible to guarantee optimal

Property of being consistent

Claim:

If h is a consistent heuristic and all costs are non-zero, then $f(n)$ is **non-decreasing along any path**. (As you move toward the goal, the estimated total cost to reach it never decreases)



Proof:

Let n' be a successor of n via action a. Then

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

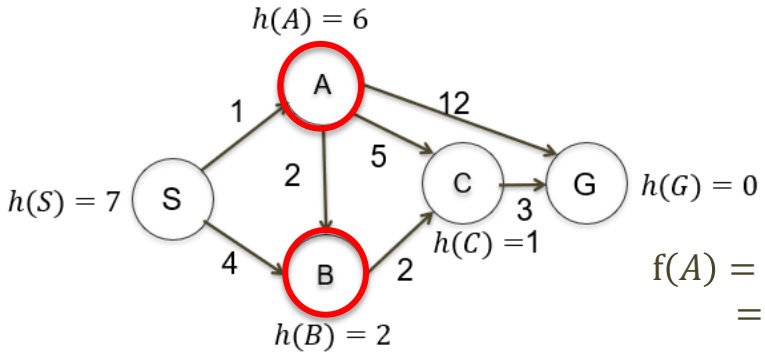
Since h is consistent, we have:

$$h(n) \leq c(n, a, n') + h(n')$$

So:

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

This implies that when a node is explored, the shortest path to it has already been found, and any future path to that node will not be shorter.



A heuristic $h(n)$ is consistent (or monotonic) if for every node n, every successor n' of n generated by any action a:

$$h(n) \leq c(n, a, n') + h(n')$$

Explored set = { S, B, A, C }

$$h(A) \geq c(A, a, B) + h(B)$$

$$6 \geq 2 + 2$$

Not consistent!

Resulting path: S → A → C → G with a total cost of 9

Optimal path: S → A → B → C → G with a total cost of 8

A* search properties

but not admissible/consistent
doesnt mean not optimal

Optimality

- Tree search version of A* is optimal if the heuristic is *admissible*.
- Graph search version of A* is optimal if the heuristic is *consistent*.

it ensures non-decreasing f(n) values along a path

Completeness

- A* search is generally complete (provided finite branching factor).

Consistency is a **stricter requirement** than admissibility.

Note: *These properties of A* — optimality and completeness —* ***hold only when all edge costs are non-negative****. If negative edge costs exist, A* may fail to find the optimal path or even a solution at all.*

Summary

- A* combines the cost so far (from the start) and an estimated cost to the goal.
- A* is optimal with admissible (tree search) / consistent (graph search) heuristics
- Heuristic design is key
- When the heuristic values are all zero, A* search behaves exactly like Dijkstra's algorithm.

Q2. Which of the following is true about A* search? Select all that apply.

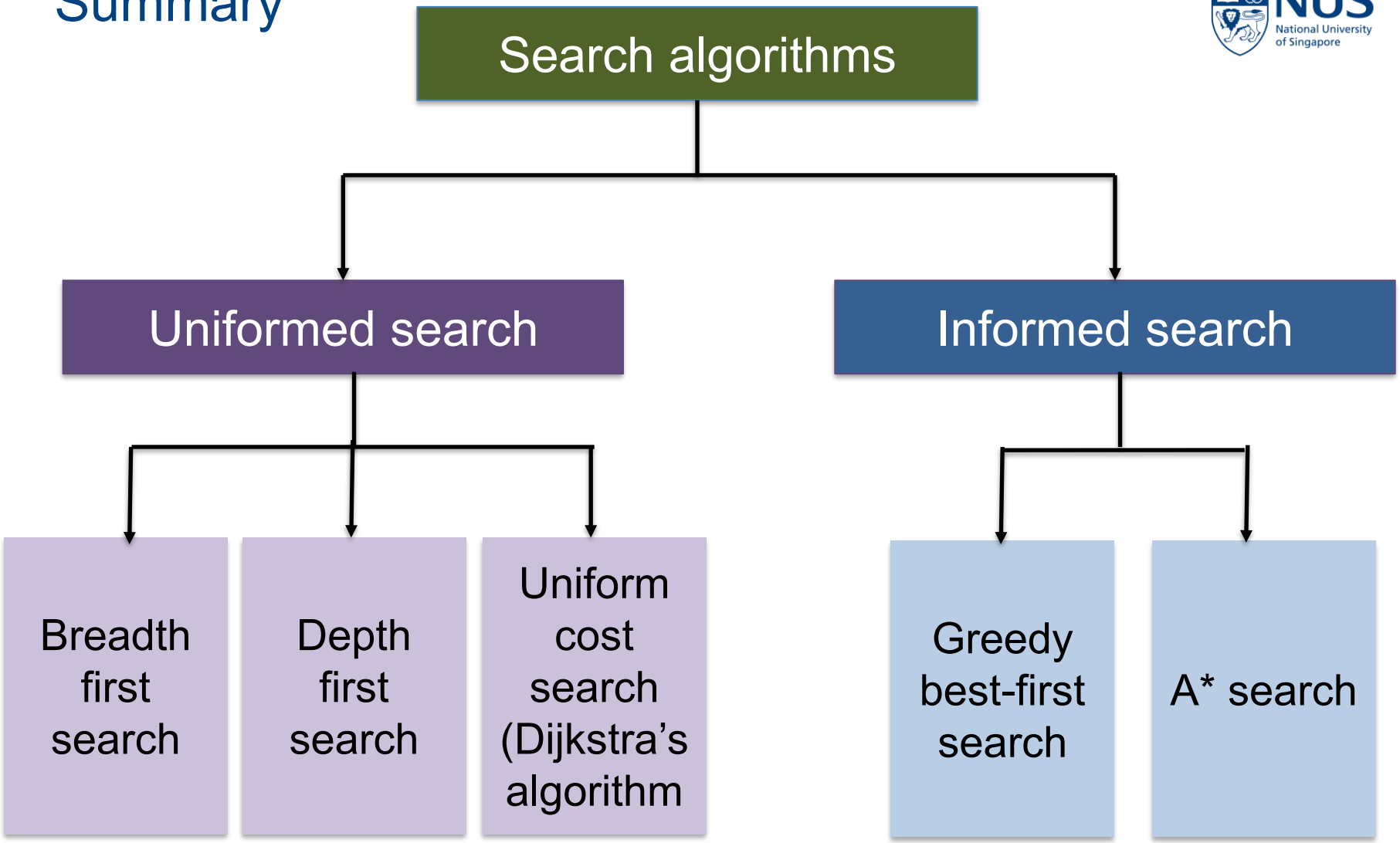
A. The heuristic function can overestimate the true cost to the goal without affecting the optimality of A* search. it MAY affect

B. A* search is generally complete and returns a solution if one exists, provided the search space is finite and all step costs are positive. ✓

C. Every consistent heuristic is also admissible. ✓

D. The graph-search of A* is optimal as long as the heuristic is admissible.

Summary



Q1. Which of the following search algorithms typically use a priority queue to implement the frontier? Select all that apply.

A. Breadth-first search

B. Uniform cost search (i.e., Dijkstra's algorithm) ✓ prioritize based on $g(n)$

C. Greedy best-first search ✓ prioritize based on $h(n)$

D. A* search ✓ prioritize based on $f(n) = g(n) + h(n)$

Summary: Uninformed search & informed search

Uninformed search (BFS, DFS, Dijkstra's)

- Has no additional knowledge beyond what is given in the problem definition.
- Generate successor states and check whether each is a goal.
- As a result, the search proceeds blindly—systematically exploring the space until the goal is found.
- Search methods are distinguished by the order in which the nodes are expanded.

Informed search (greedy best-first, A*)

- Uses additional knowledge (a heuristic) to estimate how close a state is to the goal.
- The heuristic guides the search, often reducing the number of nodes explored and improving efficiency.

Q3. Prove the following statement:

If a heuristic $h(n)$ is consistent, then it is also admissible.

Hint: Proof by induction

Q3. Prove if a heuristic $h(n)$ is consistent, then it is also admissible

Admissible heuristic:

For every node n,

$$h(n) \leq h^*(n),$$

where $h^*(n)$ is the actual cost to from n to goal.

To be admissible, $h(\text{goal}) = 0$

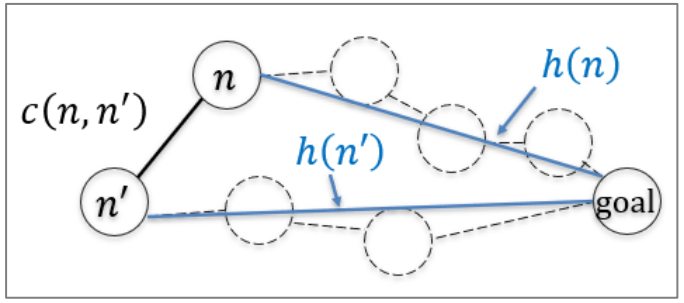
Consistent heuristic:

For every node n and successor n' with step cost $c(n, a, n')$ by any action a :

$$h(n) \leq c(n, n') + h(n')$$

and

$$h(\text{goal}) = 0$$



The notation $c(n, n')$ is a simplified version of $c(n, a, n')$

Q3. Prove if a heuristic $h(n)$ is consistent, then it is also admissible

Proof by induction

1. Show it is true for a **base case** ($n = \text{goal}$)
2. Assume it is true for all nodes at most k steps away from the goal
3. Use step 2 to show for $k + 1$: Prove it holds for nodes at most $k + 1$ steps away from the goal

Induction is a key technique for proving the correctness of search algorithms, especially when reasoning about properties of paths or heuristics across multiple steps in the search process.

Q3. Prove if a heuristic $h(n)$ is consistent, then it is also admissible

Proof by induction

Base case:

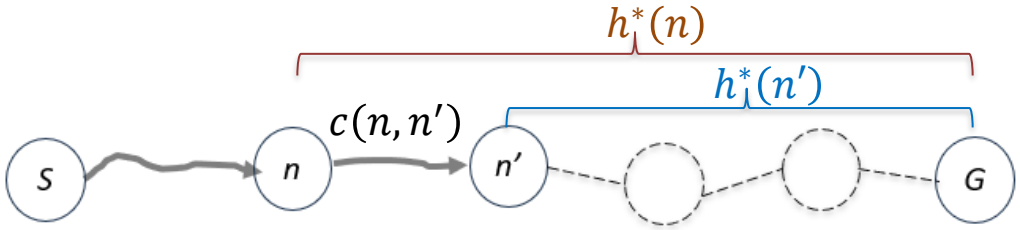
For the goal node $n = \text{goal}$,

By definition:

$$h(\text{goal}) = 0$$

$h(\text{goal}) \leq h^*(\text{goal})$ holds

So the heuristic is admissible (doesn't overestimate).



Inductive Hypothesis:

Assume that any node n' that is at most k steps away from the goal along the shortest path satisfies the admissible property:

$$h(n') \leq h^*(n') \quad \text{(i.e., } h \text{ is admissible for those nodes)}$$

Inductive step:

Let n be the predecessor of n' on that shortest path (so $n \rightarrow n' \rightarrow \cdots \rightarrow$goal), i.e., n is $k+1$ steps away from the goal along a shortest path.

By consistency: $h(n) \leq c(n,n') + h(n')$

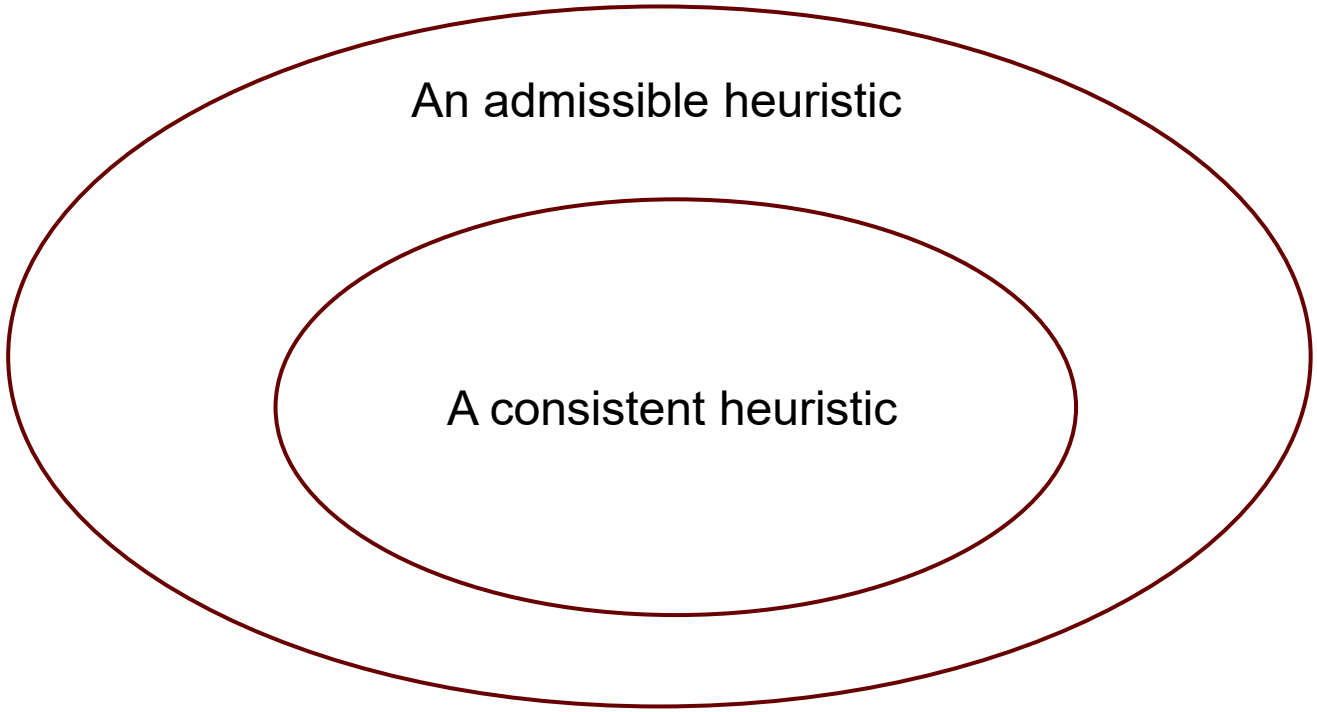
By the *inductive hypothesis*: $h(n') \leq h^*(n')$

So: $h(n) \leq c(n,n') + h^*(n') = h^*(n)$

Therefore, $h(n) \leq h^*(n)$: admissibility holds for n

$h^*(n)$: the actual cost from n to the goal

Q3. Takeaway



A consistent heuristic is always admissible.

However, an admissible heuristic is not always consistent.

Summary

Feature	Uninformed search	Informed search
Search Algorithms	BFS, DFS, UCS (Dijkstra's algorithm)	Greedy best-first search, A* search
Additional knowledge available?	No (explores blindly, only uses problem definition, i.e., start, actions, goal)	Yes (uses domain-specific knowledge/heuristics to guide search)
Complete?	BFS & UCS (Dijkstra's algorithm): Yes DFS: No (can get stuck in infinite paths)	Greedy best-first: No (can get stuck in local optima) A*: Generally yes (finite search space and non-negative edge cost)
Optimal?	BFS: Yes (if all step costs are equal/unweighted graph) DFS: No (finds any path, not necessarily shortest) UCS (Dijkstra's algorithm): Yes (finds optimal path in weighted graphs with non-negative costs) always guaranteed	Greedy best-first: No (may find suboptimal path instead) A*: Yes (with admissible & consistent heuristic) tree search graph search
Efficiency	Generally less efficient for large search spaces, as it explores broadly/deeply without guidance. May explore many irrelevant paths such as reverse	Generally more efficient for large search spaces, as heuristics guide it towards the goal.
When to use	Small or simple problems No heuristic available	Larger or more complex problems When good heuristic is available

Agenda

- A* search
- Planning (STRIPS)

STRIPS: A classic model for planning

STRIPS = STanford Research Institute Problem Solver

Introduced in the early 1970s for robot planning

A STRIPS planning problem specifies:

- An initial state S
- A goal G
- A set of STRIPS actions

STRIPS laid the foundation for task planning in robotics and still influences modern AI systems.

Reference: https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/strips.pdf
https://www.geeksforgeeks.org/artificial-intelligence/strips-in-ai/

STRIPS: Example

Search Concept (What they know)	STRIPS Concept (What you're introducing)	Example (Coffee Robot)
State	A state is a collection of **logical propositions** that are true.	{Has(Coffee), On(Mug, Table)}
Goal Test	The goal is a set of conditions that must be true.	{Has(Coffee)}
Actions	An action is an **operator** with: - **Preconditions:** Logical facts that must be true before you can perform the action. - **Effects:** How the action changes the state. It adds and removes facts.	**Action**: Pour(CoffeePot, Mug) **Pre**: Has(CoffeePot), On(Mug, Table) **Effect**: ~Has(CoffeePot), Has(Coffee)
Path Cost	Often just the number of actions (plan length).	Cost of [Grasp(Mug), Pour(...)] = 2
Successor Function	Applying an action. For a given state, check an action's preconditions. If they are all true, the new state is: (Old State - Effects.Remove) + Effects.Add	

With STRIPS modeling, we can represent the planning problem as a graph, where nodes are states and edges are actions, and then apply the search algorithms we've learned to find a solution.

STRIPS: Example

How this creates a search problem

Use BFS to find a plan:

State: {Has(CoffeePot), On(Mug, Table)} (*Initial State*)
Action: Apply Pour(CoffeePot, Mug)

- Check Preconditions: True.

Calculate New State:

- Remove: Has(CoffeePot)
- Add: Has(Coffee)

New State: {Has(Coffee), On(Mug, Table)} *GOAL*!

The Plan (Path): [Pour(CoffeePot, Mug)]

Note: BFS systematically explores all possible action sequences level by level, guaranteeing it will find the shortest plan (fewest actions) if one exists.

Demo: Refer to Lecture4_STRIPS_Example.py

Reference: https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/strips.pdf
https://www.geeksforgeeks.org/artificial-intelligence/strips-in-ai/

EE2213: Introduction to Artificial Intelligence

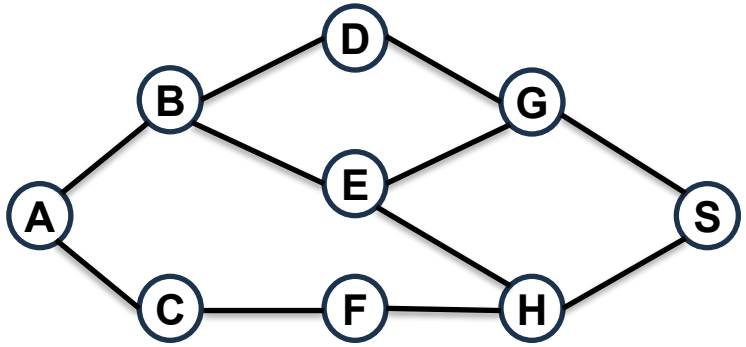
Tutorial 2 (Lectures 2 & 3)

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

Q3. You are given the following undirected graph.

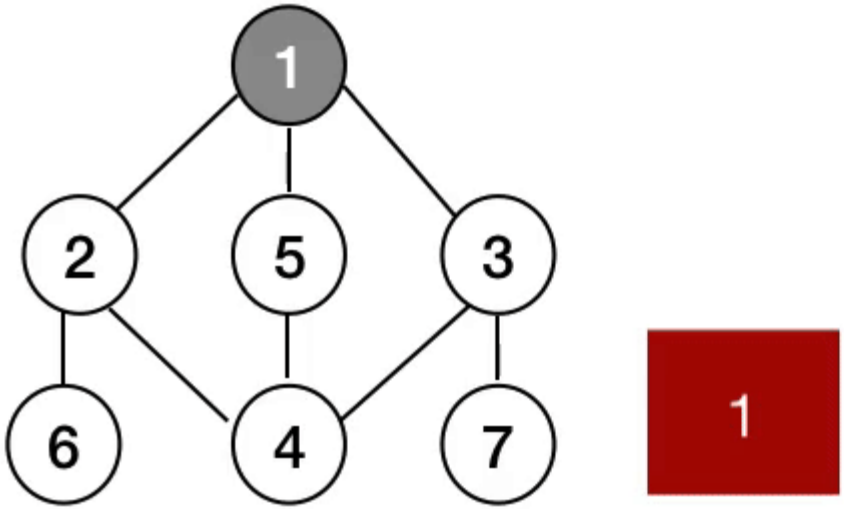
(i) Write a Python program to perform a Breadth-First Search (BFS) traversal starting from node ‘A’. Print the order in which nodes are visited.

(ii) Write a Python program to perform a Depth-First Search (DFS) traversal starting from node ‘A’. Print the order in which nodes are visited.



Note: The order of nodes visited may differ depending on the order neighbors are processed.

Recap: BFS



Credit: https://medium.com/@dillihangrae/searching-in-ai-dfs-bfs-7417d34e8113

Q3. (i) BFS

```python
def bfs(graph, start):
    visited = []    # List to keep track of visited nodes
    queue = []      # Queue to hold nodes to explore
    sequence = []  # List to store the visiting sequence

    visited.append(start)
    queue.append(start)

    while queue:
        node = queue.pop(0)
        print(node)
        sequence.append(node)

        for neighbour in graph[node]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)
    return sequence
```

Create an empty list to keep track of explored (visited) nodes
Create an empty list serve as the frontier, which acts as a **queue**
Create an empty list to store the visited nodes *in the order they are explored*

Add the starting node to the list of visited nodes.
Add the starting node to the frontier (queue) to begin the search

While frontier is not empty, repeat:

-- Pop the earliest added node (i.e., the first node in the queue) from the frontier (**FIFO order**).

-- Add it to the sequence, and start to expand the node

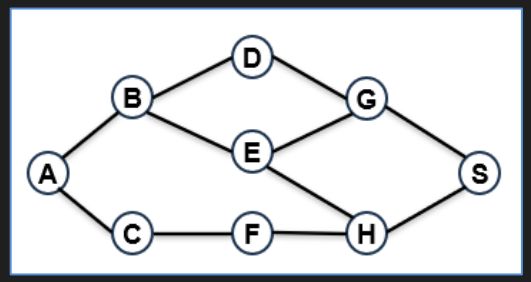
-- For each neighbor of the node:

- If the neighbor has not been visited,
- Mark it as visited and add it to the end of the queue.

Q3. (i) BFS (cont.)

```python
# Define the graph using a dictionary.
# Each node maps to a list of its connected neighbors.
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B', 'G'],
    'E': ['B', 'G', 'H'],
    'F': ['C', 'H'],
    'G': ['D', 'E', 'S'],
    'H': ['E', 'F', 'S'],
    'S': ['G', 'H']
}

# Run BFS starting from node 'A'
visit_sequence = bfs(graph, 'A')
print("Visit sequence:", visit_sequence)
```



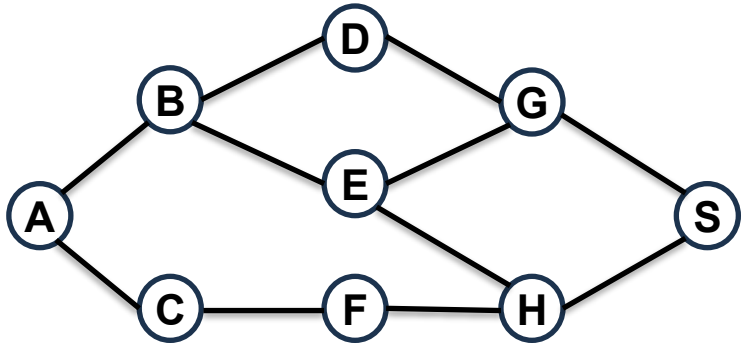

```
Visit_sequence: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'S']
```

(The order of nodes visited may differ depending on the order neighbors are processed.)

Q3. You are given the following undirected graph.

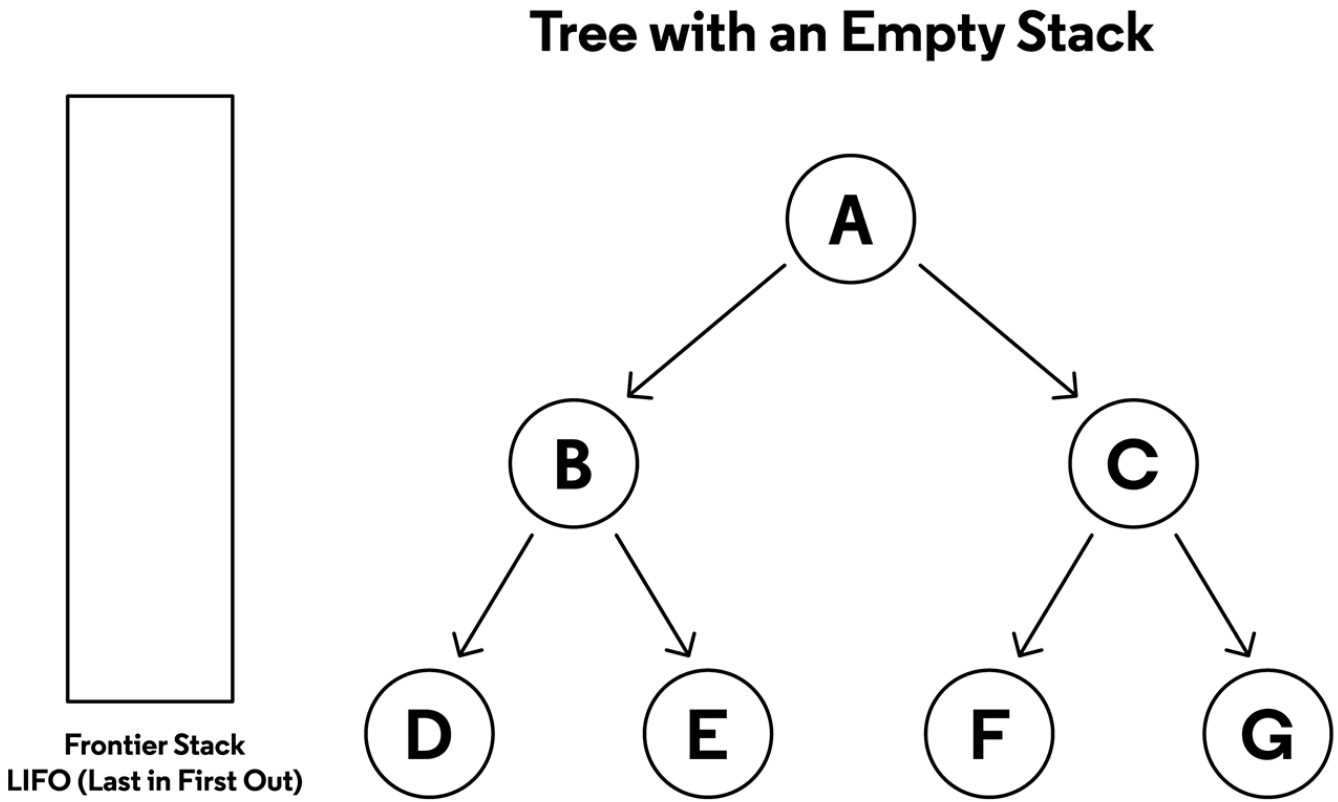
(i) Write a Python program to perform a Breadth-First Search (BFS) traversal starting from node ‘A’. Print the order in which nodes are visited.

(ii) Write a Python program to perform a Depth-First Search (DFS) traversal starting from node ‘A’. Print the order in which nodes are visited.



Note: The order of nodes visited may differ depending on the order neighbors are processed.

Recap: DFS



Credit: https://www.codecademy.com/article/depth-first-search-conceptual

Q3. (ii) DFS

```python
# Define DFS function using stack
def dfs(graph, start):
    visited = []   # List to keep track of visited nodes
    stack = []     # Stack to hold nodes to explore
    sequence = []  # List to store the visiting sequence

    visited.append(start)
    stack.append(start)

    while stack:
        node = stack.pop()
        print(node)
        sequence.append(node)

        for neighbour in graph[node]:
            if neighbour not in visited:
                visited.append(neighbour)
                stack.append(neighbour)

    return sequence
```

Create an empty list to keep track of explored (visited) nodes
Create an empty list serve as the frontier, which acts as a **stack**
Create an empty list to store the visited nodes *in the order they are explored*

Add the starting node to the frontier (stack) and mark it as visited

While frontier (stack) is not empty, repeat:

-- Pop the most recently added node from the stack (**LIFO order**).

-- add it to the sequence and start to expand the node

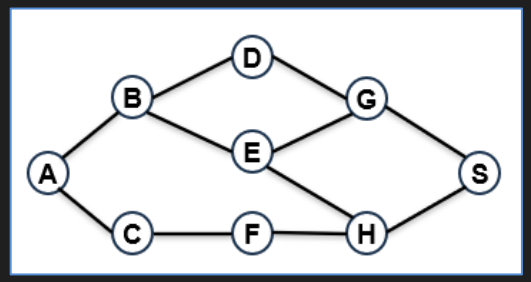
-- **For each neighbor of the node**:

- If the neighbor has not been visited,
- Mark it as visited and push it onto the stack.

Q3. (ii) DFS (cont.)

```python
# Define the graph using a dictionary.
# Each node maps to a list of its connected neighbors.
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B', 'G'],
    'E': ['B', 'G', 'H'],
    'F': ['C', 'H'],
    'G': ['D', 'E', 'S'],
    'H': ['E', 'F', 'S'],
    'S': ['G', 'H']
}

# Call DFS starting from node 'A'
visit_sequence = dfs(graph, 'A')
print("Visit_sequence:", visit_sequence)
```




```
Visit_sequence: ['A', 'C', 'F', 'H', 'S', 'G', 'D', 'E', 'B']
```

(DFS doesn't guarantee a unique order - it depends on the order neighbors that are processed)

Q3. (ii) DFS (recursive method)

```python
# Recursive DFS function
def dfs(graph, node, visited):
    print(node)
    visited.append(node)

    for neighbour in graph[node]:
        if neighbour not in visited:
            dfs(graph, neighbour, visited)
```

Recursion: call the same dfs function **on the unvisited neighbor**. This means we go deeper before we come back up, **a core idea of depth-first search.**

- Visits nodes as deep as possible before backtracking.
- The visited list is passed along to remember which nodes we've already explored.
- Here, **recursion** handles the **stack-like behavior** of DFS automatically.

Q3 Code highlights: List operations in Python

```
visited = []
queue = []
sequence = []
```
Create empty lists

append()- Add an item to a list

- **Adds an element to the end of a list.**
- **Commonly used to add nodes to the visited list or search frontier (queue/stack).**

```
visited.append(start)
```
Adding the start node to the visited list

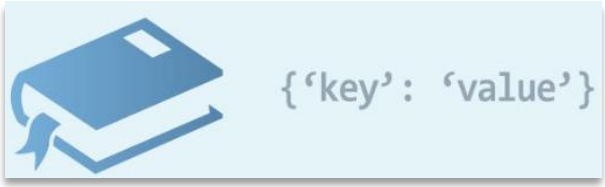
pop()- Remove and return an item from a list

- **pop(0) removes the first item → used in queues (FIFO)**
- **pop() (without index) removes the last item → used in stacks (LIFO)**

```
node = queue.pop(0)
```
Removes the earliest added node from the queue

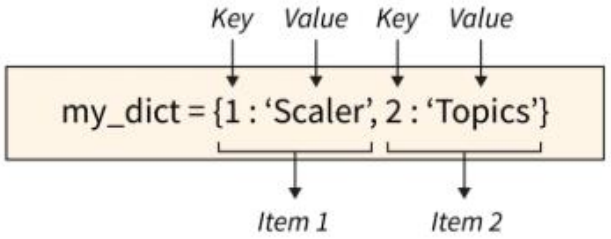
Reference: https://docs.python.org/3/tutorial/datastructures.html

Q3 Code highlights: Python dictionary



A Python dictionary is a collection of key-value pairs used to store and quickly access data by unique keys.

```
my_dict = {}           # create an empty dictionary
my_dict[1] = 'Scaler'  # assign the value 'scalar' to key 1
my_dict[2] = 'Topics'  # get the value 'Topics' to key 2
```




```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B', 'G'],
    'E': ['B', 'G', 'H'],
    'F': ['C', 'H'],
    'G': ['D', 'E', 'S'],
    'H': ['E', 'F', 'S'],
    'S': ['G', 'H']
}
```

- In search algorithms, we often use a dictionary to represent a graph.
- Each key represents a node (e.g., 'A', 'B'), and
- each value is a list of that node's direct neighbors (i.e., the nodes it connects to directly).
- We also commonly use dictionary to record how we reach each node during traversal. E.g., each key is a node, and each value is the predecessor node from which that node was reached.

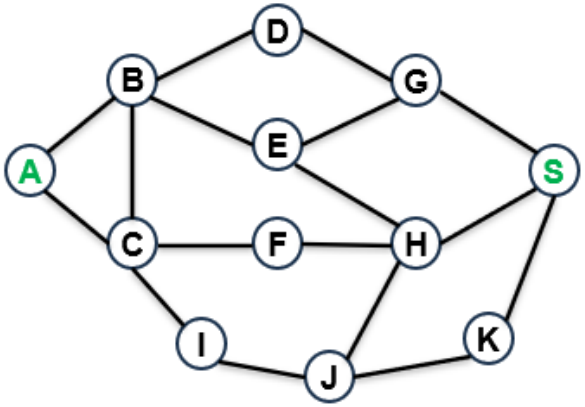
Reference: https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Q4. Your friend is flying from Arequipa (A), a historic city and the second-largest in Peru, to Singapore (S). However, there is no direct flight between the two cities.

You decide to plan a route with the minimum number of flight segments needed for your friend. To do this, you model the available flight connections as an undirected graph as shown below, where nodes represent cities, and edges represent direct flights between cities. The modeled graph is shown below.

(i) Which strategy is better for finding the fewest number of flight segments: Breadth-First Search (BFS) or Depth-First Search (DFS)?

(ii) Write a Python programme to compute the minimum number of flight segments from Arequipa to Singapore.

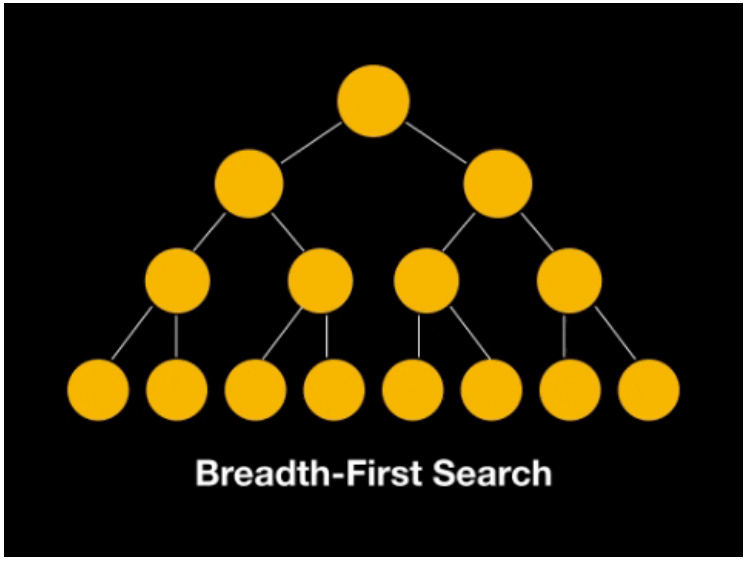


Reference: A similar but tougher question is LeetCode #127: https://leetcode.com/problems/word-ladder/description/

Q4. (i) Which strategy is better for finding the fewest number of flight segments: Breadth-First Search (BFS) or Depth-First Search (DFS)?

Why BFS?

- BFS explores all cities that are 1 flight away, then 2 flights away, and so on.
- It guarantees that the first time we reach the destination, it's through the **shortest path (least number of flights)**.
- Think of it as searching **level by level.**



Why not DFS?

DFS explores deeply first, possibly following long or inefficient paths.
It might reach the destination later, not guaranteeing the shortest route.

Source: https://medium.com/@basabjha/dsa-graph-secrets-dfs-and-bfs-made-simple-for-beginners-4532f90d7611

(ii) Write a Python function to compute the minimum number of flight segments from Arequipa to Singapore.

```python
def bfs_shortest_path(graph, start, goal):
    visited = []  # List to keep track of cities we have already visited
    queue = []    # Queue for BFS: stores cities to explore next
    path = {}

    queue.append(start)
    visited.append(start)
    path[start] = None

    while queue:
        city = queue.pop(0)

        if city == goal:
            result = []  # List to store the path from start to goal
            while city is not None:
                result.append(city)
                city = path[city]
            result.reverse()  # reverse to get path from start to goal
            return result

        for neighbor in graph[city]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
                path[neighbor] = city

    return None  # if no path found
```

Create an empty dictionary to remember how we got to each city
Each **key** is a **city**, and its **value** is the **city we came from** (its **predecessor**).
path = {city: predecessor}

Start node has no predecessor

Loop until there are no more cities to explore

Remove (pop) the **earliest** city added to the queue to explore (**FIFO order**)

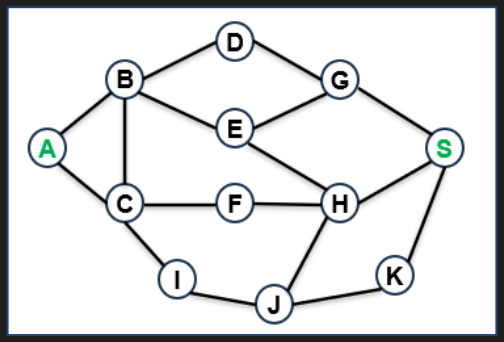
- If the current city == goal, **reconstruct the path** taken to reach it.
- We do this by following each city's **predecessor** in the path dictionary.

Check all neighboring cities directly connected to the current city.
If a neighbor has not been visited:

- Mark it as visited
- Add it to the queue
- Record its **predecessor** (i.e. the **current city**) in the path dictionary

(ii) Write a Python function to compute the minimum number of flight segments from Arequipa to Singapore (cont.).

```
# Define the graph using a dictionary.
# Each city (node) maps to a list of cities it has direct flights to (neighbors).
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D', 'E'],
    'C': ['A', 'B', 'F', 'I'],
    'D': ['B', 'G'],
    'E': ['B', 'G', 'H'],
    'F': ['C', 'H'],
    'G': ['D', 'E', 'S'],
    'H': ['E', 'F', 'J', 'S'],
    'I': ['C', 'J'],
    'J': ['I', 'K'],
    'K': ['J', 'S'],
    'S': ['G', 'H']
}
```

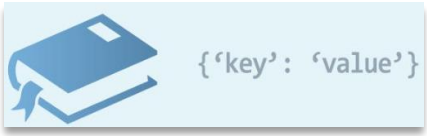


Note: There can be more than one path with the same minimum number of flight segments, and our program returns just one of them

```
# Find the shortest path using bfs
route = bfs_shortest_path(graph, 'A', 'S')
if route:
    print("Path with the fewest flight transfers from Arequipa to Singapore:", route)
    print("Minimum number of flight segments:", len(route) - 1)
else:
    print("No route found.")
```

```
Path with the fewest flight transfers from Arequipa to Singapore: ['A', 'B', 'D', 'G', 'S']
Minimum number of flight segments: 4
```

Q4 Code highlights: Dictionary operations



`path = {city: predecessor}`

Dictionary structure to track how we reach each city
It allows us to **reconstruct the entire route** from the start city to the goal by backtracking from the goal using this stored information.

```
path[neighbor] = city
```

Dictionary Assignment
Records the **current city** as the **predecessor** of the **neighbor.**
Helping us remember how we reached each node during the search.

```
city = path[city]
```

Dictionary Lookup
Retrieves the **predecessor** of the current city from the path dictionary, and assigns it back to city.
Used to trace the path backward from goal to start.

Reference: https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Q4 Code highlights: List operations

```
result = reverse()
```

Reversing a list

`.reverse()` reverse the order of elements in the list.
Modifies the original list (does not create a new list)

In pathfinding algorithms like BFS, we often build the path backward (from goal to start).

We use .reverse() to flip it and get the correct order: from start to goal.

Reference: https://docs.python.org/3/tutorial/datastructures.html

Q4 Takeaway

Breadth-First Search (BFS) is the most suitable strategy for finding the fewest number of flight segments, as it explores all paths layer by layer and guarantees the shortest path in terms of edge count in unweighted graphs.

Multiple equally optimal solutions may exist: There can be more than one path with the same minimum number of flight segments. Since in our program, BFS stops when it finds the first shortest path, it returns just one of these equally valid solutions.

Reference: https://docs.python.org/3/tutorial/datastructures.html

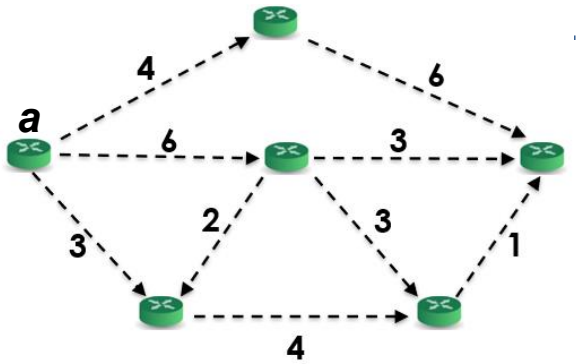
Q5. Signal Propagation in a Network (Adapted from LeetCode #743)

You are given a communication network consisting of several nodes connected by directed links, each with an associated transmission delay (indicated by the numbers beside each edge), as illustrated in the graph below. Given a starting node a (the leftmost node), determine the minimum time required for a signal to reach all nodes in the network.

i. Which search algorithm that you have learned so far would be most suitable for this problem? Briefly justify your choice.

dijkstra since weighted

ii. (b) Implement your solution in Python using the selected search algorithm. Your program should:
   - Model the network as a graph.
   - Compute the minimum time it takes for the



Q5 (i) Which search algorithm that you have learned so far would be most suitable for this problem?

Algorithm	Handles weights	Finds optimal Path	Uses heuristic	Complete	Best use case
BFS	No	Yes (unweighted)	No	Yes	Shortest path in **unweighted** graphs
DFS	No	No	No	No	Explores deeply, not optimal paths
Dijkstra	Yes	Yes	No	Yes	**Best for shortest paths with weights**
Greedy Best-First	Yes	No	Yes	No	Fast guesses, not always correct
A*	Yes	Yes (if heuristic is good)	Yes	Yes	Combines cost + heuristic

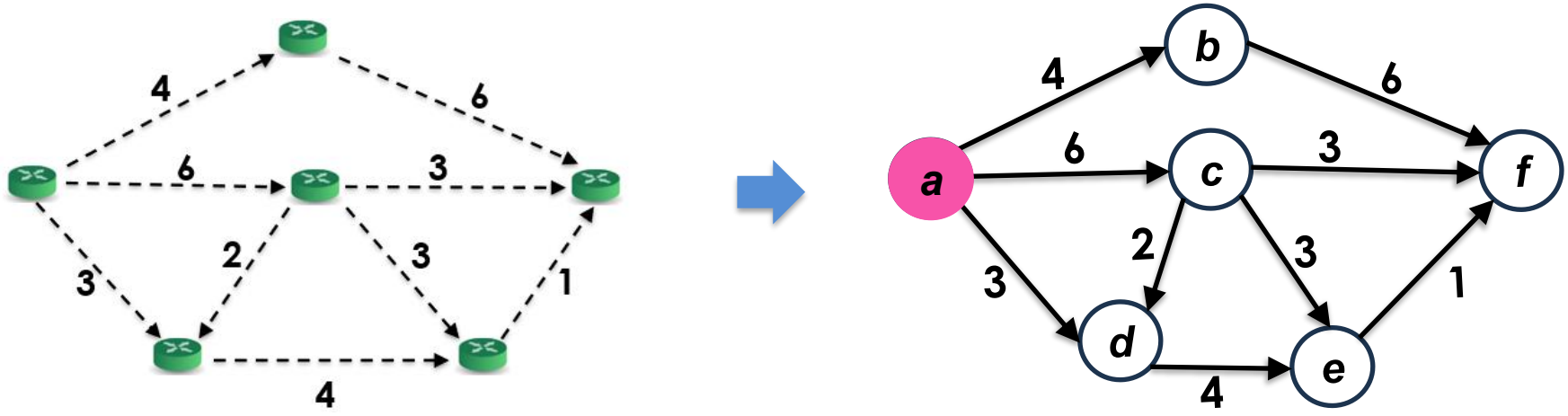
Q5 (i) Which search algorithm that you have learned so far would be most suitable for this problem? (cont.)

The most suitable algorithm is **Dijkstra's Algorithm**.

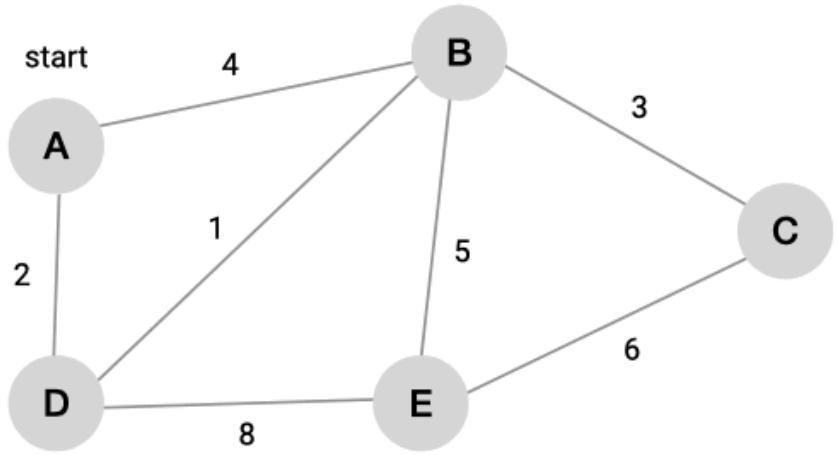
Justification: Dijkstra's algorithm is specifically designed to find the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights. It works by greedily selecting the unvisited node with the smallest known distance from the source, ensuring that the first time it reaches a node, it has found the shortest path to it.

Q5 (ii) Implement your solution in Python using the selected search algorithm.

Model the communication network as a graph and specify the starting node.



Recap: Dijkstra's algorithm



Vertex	Shortest distance from A	Previous vertex
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Credit: https://memgraph.com/docs/advanced-algorithms/deep-path-traversal

Dijkstra's algorithm: Pseudocode

Dijkstra(G, l, s)

for all $u \in V$:

$dist[u] \leftarrow \infty$

$prev[u] \leftarrow nil$

$X \leftarrow \emptyset$ // X: processed vertices

$Q \leftarrow \emptyset$ // Q: frontier, implemented as priority queue

$dist[s] \leftarrow 0$

updateQueue $(Q, s, 0)$ // enqueue starting node with distance 0

while Q is not empty:

$u \leftarrow$ extractMin (Q) // remove and return node with min distance

$X \leftarrow X \cup \{u\}$ // mark u as processed

for all edges $(u, v) \in E$:

if $v \notin X$ and $dist[u] + l(u, v) < dist[v]$:

$dist[v] \leftarrow dist[u] + l(u, v)$ // update shortest distance

$prev[v] \leftarrow u$ // update previous node

updateQueue $(Q, v, dist[v])$ // update priority queue

Case1: a node isn't in the queue, it's inserted with its distance.
Case2: a node is in the queue and a shorter path is found, update the distance

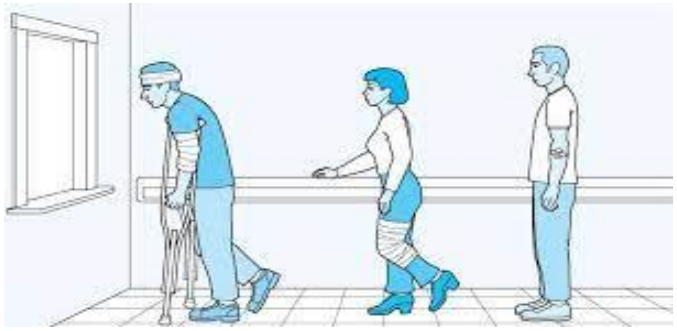
Q5 (ii) Implement your solution in Python using the selected search algorithm.

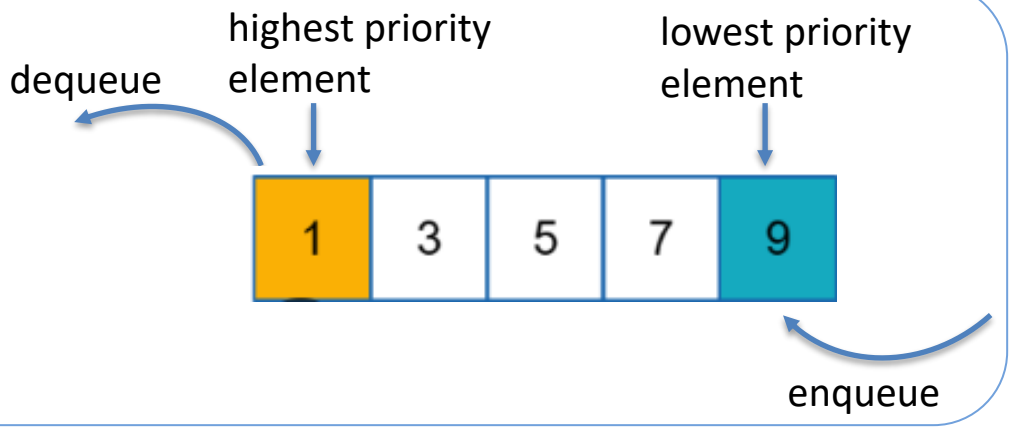
Key implementation detail:

•Dijkstra's algorithm uses a **priority queue** to select the nearest unvisited node

•In Python, a **heap-based priority queue** (like heapq) is the most common and efficient way to implement Dijkstra's algorithm, though other structures are also possible.

Priority queue:





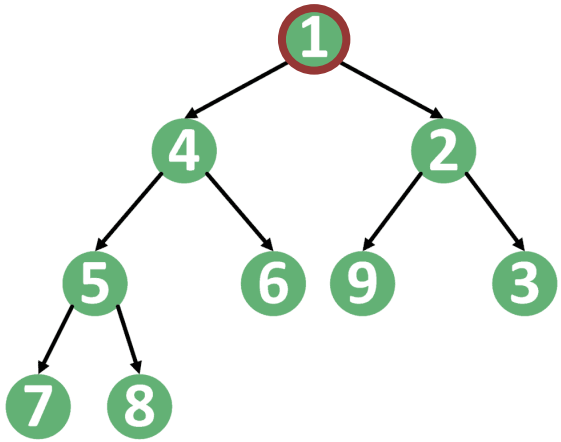
Overview: Heap data structure

- A heap is a data structure designed to quickly find and update the minimum/maximum element

- Binary min-heap:

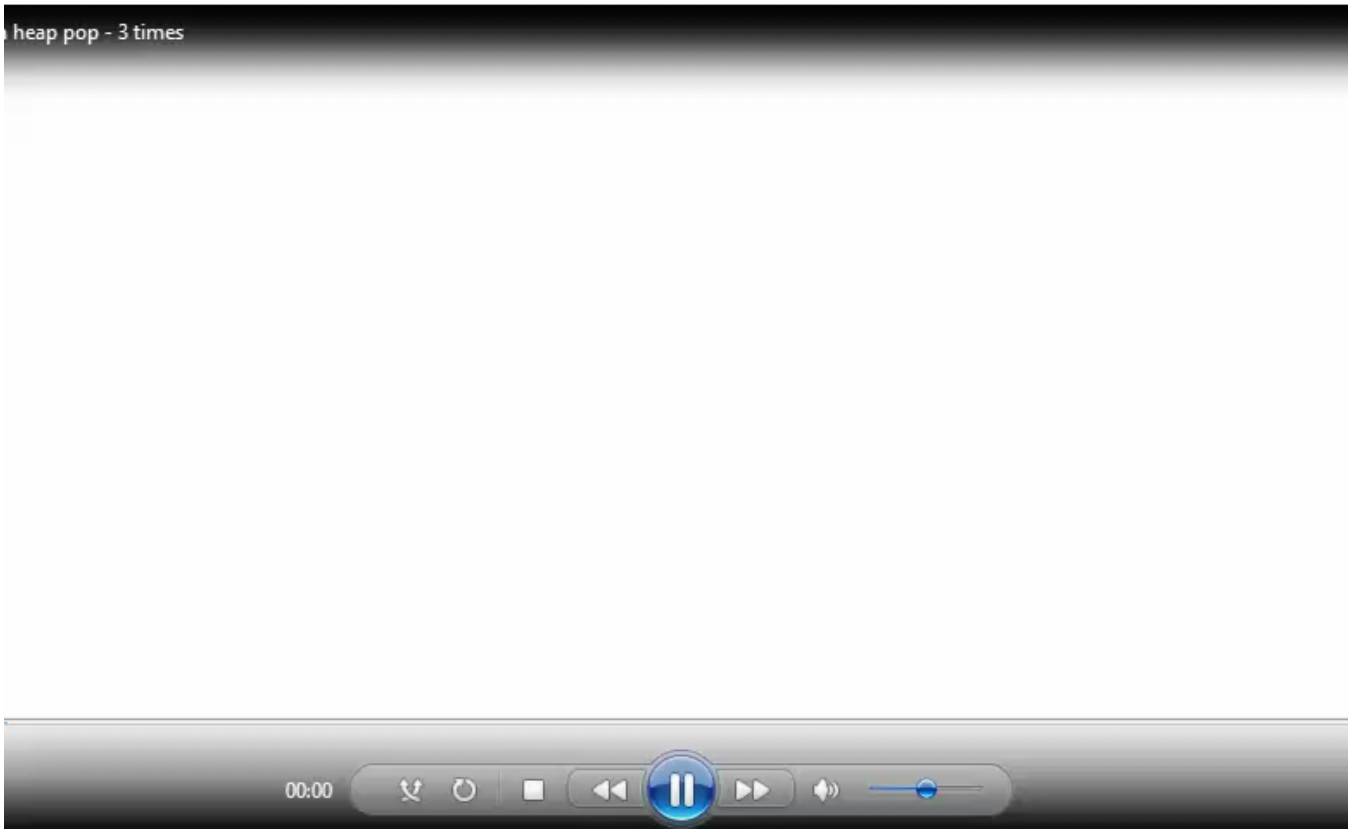
  -- a complete binary tree.

  -- at every node, key ≤ children's keys



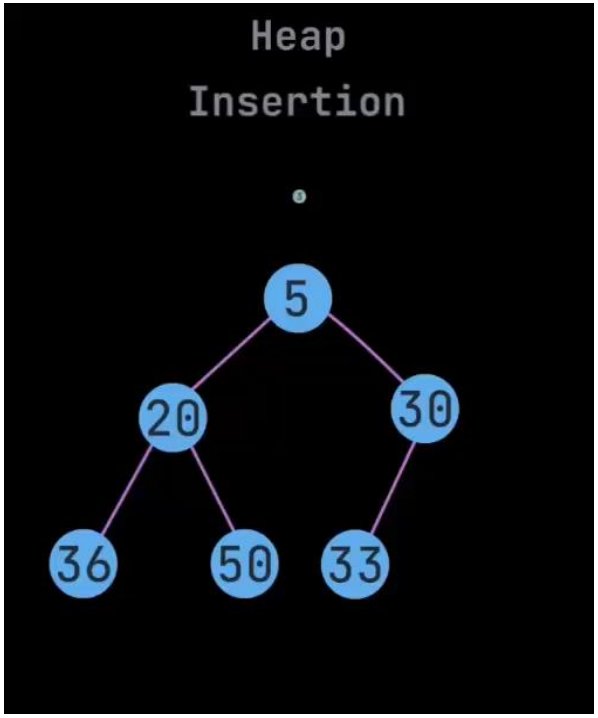
Overview: Heap operation

Extract the minimum from a min-heap by swapping up last leaf, bubbling down:

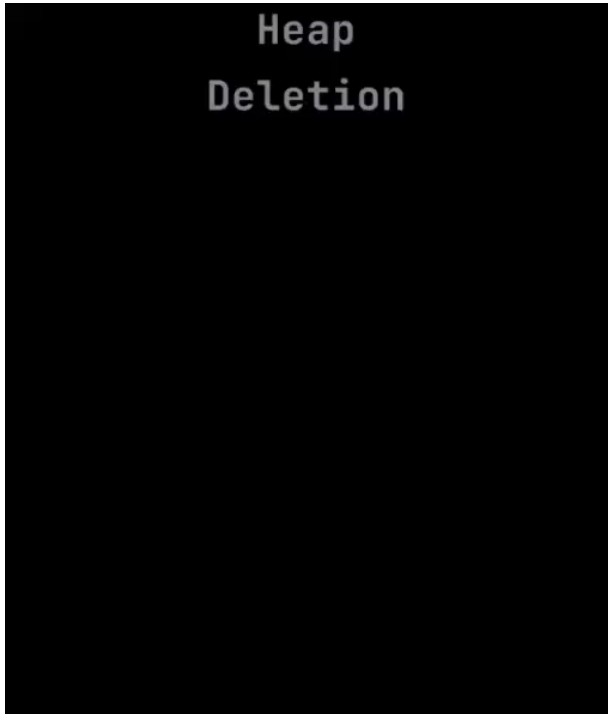


Reference: https://www.youtube.com/shorts/3Q4oQk4pEks

Overview: Heap operation (cont.)



Insertion in min-heap



Deletion in min-heap

- In our class, you don't need to understand how a heap works internally.
- Python provides a built-in library (heapq) that makes it easy to use.
- Just remember: a **heap** is a data structure used for a **priority queue**, helping Dijkstra's algorithm efficiently *select the node with the smallest distance*.

Source: https://www.youtube.com/shorts/QAaxJmdneS4

Q5 (ii) Implement your solution in Python

```python
from heapq import heappush, heappop
import math

def dijkstra(graph, s):
    node_data = {node: {'dist': math.inf, 'prev': []} for node in graph}
    X = set()  # processed/visited nodes
    Q = []     # frontier, implemented as a priority queue

    heappush(Q, (0, s))  # enqueue starting node (source) s with distance 0
    node_data[s]['dist'] = 0

    while Q:
        u_dist, u = heappop(Q)  # extractMin(Q)
        if u in X:
            continue
        X.add(u)  # mark u as processed

        for v in graph[u]:
            if v not in X:
                dist = node_data[u]['dist'] + graph[u][v]
                if dist < node_data[v]['dist']:
                    node_data[v]['dist'] = dist  # update shortest distance
                    node_data[v]['prev'] = node_data[u]['prev'] + [u]
                    heappush(Q, (node_data[v]['dist'], v))  # updateQueue

    return node_data
```

for all $u \in V$:
  $dist[u] \leftarrow \infty$
  $prev[u] \leftarrow nil$
$X \leftarrow \emptyset$
$Q \leftarrow \emptyset$
$dist[s] \leftarrow 0$
updateQueue $(Q, s, 0)$

while Q is not empty:
  $u \leftarrow$ extractMin (Q)

$X \leftarrow X \cup \{u\}$

for all edges $(u, v) \in E$:
  if $v \notin X$ and $dist[u] + l(u, v) < dist[v]$:
    $dist[v] \leftarrow dist[u] + l(u, v)$
    $prev[v] \leftarrow u$
    updateQueue $(Q, v, dist[v])$

Q5 (ii) Implement your

```
node_data = {
   'a': {'dist': 0, 'prev': []} ,
   'b': {'dist': 4, 'prev': ['a']} ,
   'c': {'dist': 6, 'prev': ['a']} ,
   'd': {'dist': 3, 'prev': ['a']} ,
   'e': {'dist': 7, 'prev': ['a', 'd']} ,
   'f': {'dist': 8, 'prev': ['a', 'd', 'e']} ,
}
```

node	shortest distance from *a*	previous node
a	0	--
b	4	*a*
c	6	*a*
d	3	*a*
e	7	*d*
f	8	*e*

```
from heapq import heappush, heappop
import math

def dijkstra(graph, s):
    node_data = {node: {'dist': math.inf, 'prev': []} for node in graph}
    X = set()  # processed/visited nodes
    Q = []     # frontier

    heappush(Q, (0, s))  # enqueue starting node (source) s with distance 0
    node_data[s]['dist'] = 0

    while Q:
        u_dist, u = heappop(Q)  # extractMin(Q)
        if u in X:
            continue
        X.add(u)  # mark u as processed

        for v in graph[u]:
            if v not in X:
                dist = node_data[u]['dist'] + graph[u][v]
                if dist < node_data[v]['dist']:
                    node_data[v]['dist'] = dist  # update shortest distance
                    node_data[v]['prev'] = node_data[u]['prev'] + [u]
                    heappush(Q, (node_data[v]['dist'], v))  # updateQueue

    return node_data
```

Create a **dictionary** to store:
- dist – current shortest distance from the source (initial value set to $+\infty$)
- prev – previous nodes on the shortest path

Q helps us always pick the unprocessed node that is currently closest to the source.

- Create a priority queue implemented using a min heap
- Push the source node s into the priority queue with distance 0

Distance from source to itself is 0.

- Get the node with the smallest distance from the source
- Q is a priority queue (min heap), so heappop() always returns the node with the smallest distance

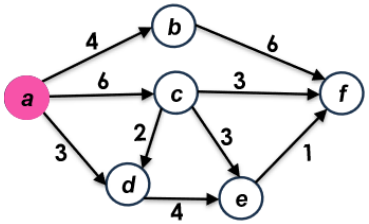
For all neighbouring nodes that are not processed:

If going through u gives a **shorter distance** to v:

- Update the shortest distance and remember the path taken
- Push node v into the priority queue with its new distance

node_data contains the shortest path and distance from s to every node

Q5 (ii) Implement your solution in Python (cont.)



Inside result dictionary:

```
result = {
  'a': {'dist': 0, 'prev': []} ,
  'b': {'dist': 4, 'prev': ['a']} ,
  'c': {'dist': 6, 'prev': ['a']} ,
  'd': {'dist': 3, 'prev': ['a']} ,
  'e': {'dist': 7, 'prev': ['a', 'd']} ,
  'f': {'dist': 8, 'prev': ['a', 'd', 'e']} ,
}
```

Output:

```
Shortest Distance to a : 0
Shortest Path to a : ['a']
Shortest Distance to b : 4
Shortest Path to b : ['a', 'b']
Shortest Distance to c : 6
Shortest Path to c : ['a', 'c']
Shortest Distance to d : 3
Shortest Path to d : ['a', 'd']
Shortest Distance to e : 7
Shortest Path to e : ['a', 'd', 'e']
Shortest Distance to f : 8
Shortest Path to f : ['a', 'd', 'e', 'f']
```

```
# Graph represented as an adjacency list (dictionary of dictionaries)
graph = {
    'a': {'b': 4, 'c': 6, 'd': 3},
    'b': {'f': 6},
    'c': {'d': 2, 'e': 3, 'f': 3},
    'd': {'e': 4},
    'e': {'f': 1},
    'f': {},
}

source = 'a'
result = dijkstra(graph, source)

# Print shortest distances from the source to all other nodes
for node in result:
    dist = result[node]['dist']
    path = result[node]['prev'] + [node]   # build the final path
    print("Shortest Distance to", node, ":", dist)
    print("Shortest Path to", node, ":", path)
```

- The graph is stored as a **dictionary of dictionaries**
- Each key (like 'a') is a node, and its value is another dictionary listing neighbors and their edge weights
- For example, 'a': {'b': 4, 'c': 6, 'd': 3} means node 'a' connects to 'b' with cost 4, 'c' with cost 6, and 'd' with cost 3

There may be more than one shortest path with the same total distance — Dijkstra's algorithm returns just one of them.

For every node in result:

- Get the shortest distance from the source
- Get the full path from the source to that node
- Print both the distance and the path clearly

Q5 Code highlights – Python libraries

```
from heapq import heappush, heappop
import math
```

•**heapq**: A library for **priority queue** operations using a **heap** data structure

•**math**: A fundamental Python package for **mathematical functions**, supporting basic and advanced math operations

```
node_data = {node: {'dist': math.inf, 'prev': []} for node in
graph}
```

Set the initial shortest distances from the source to every node as positive infinity (math.inf).

vertex	shortest distance from *a*	previous vertex
a	∞	--
b	∞	--
c	∞	--
d	∞	--
e	∞	--
f	∞	--

Q5 Code highlights – heapq library

```
from heapq import heappush, heappop
```

heappush(heap, item) is a function that adds an element (item) to the heap (heap) while maintaining the heap property.

The heap is a list that stores elements arranged so that the smallest element is always at the front (index 0).

Parameters:
heap: A list representing the heap (priority queue).
item: The element you want to add to the heap.

For example:
heappush(Q, (0, s)) push the tuple (0, s) into the heap named Q.

For more information, check the heapq documentation: https://docs.python.org/3/library/heapq.html

Q5 Code highlights – heapq library (cont.)

```
from heapq import heappush, heappop
```

heappop(heap) is a function that **removes and returns the smallest element** from the heap while maintaining the heap property.

The heap is a list that stores elements arranged so that the smallest element is always at the front (index 0).

Parameter:
heap: A list representing the heap (priority queue).

For example:
heappop(Q) removes and returns the smallest element from the heap named Q.

For more information, check the heapq documentation: https://docs.python.org/3/library/heapq.html

Q5 Code highlights – heapq library (cont.)

```
from heapq import heappush, heappop
```

heappop(heap) is a function that **removes and returns the smallest element** from the heap while maintaining the heap property.

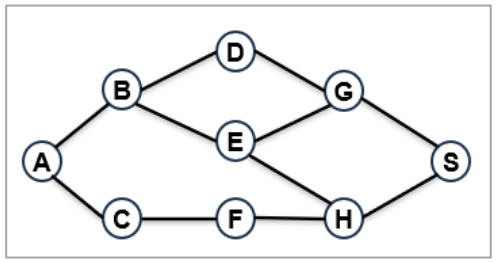
The heap is a list that stores elements arranged so that the smallest element is always at the front (index 0).

Parameter:
heap: A list representing the heap (priority queue).

For example:
heappush(Q) removes and returns the smallest element from the heap named Q.

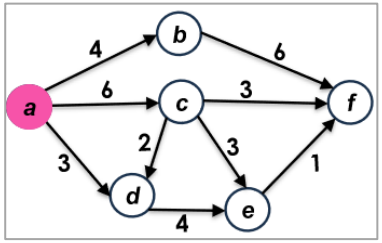
For more information, check the heapq documentation: https://docs.python.org/3/library/heapq.html

Q5 Code highlights – Graph representations in Python




```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B', 'G'],
    'E': ['B', 'G', 'H'],
    'F': ['C', 'H'],
    'G': ['D', 'E', 'S'],
    'H': ['E', 'F', 'S'],
    'S': ['G', 'H']
}
```

Unweighted (directed) graph:
Represented as a **dictionary** where each *key* is a node, and the *value* is a **list** of directly connected neighbors.



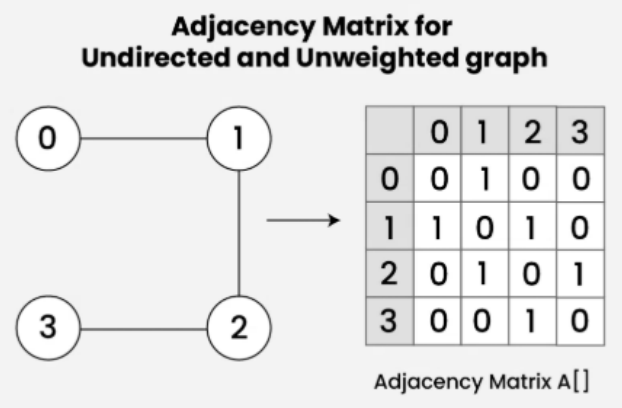

```
graph = {
    'a': {'b': 4, 'c': 6, 'd': 3},
    'b': {'f': 6},
    'c': {'d': 2, 'e': 3, 'f': 3},
    'd': {'e': 4},
    'e': {'f': 1},
    'f': {},
}
```

Weighted (directed) graph:
Represented as a **dictionary of dictionaries** (adjacency list)**:**

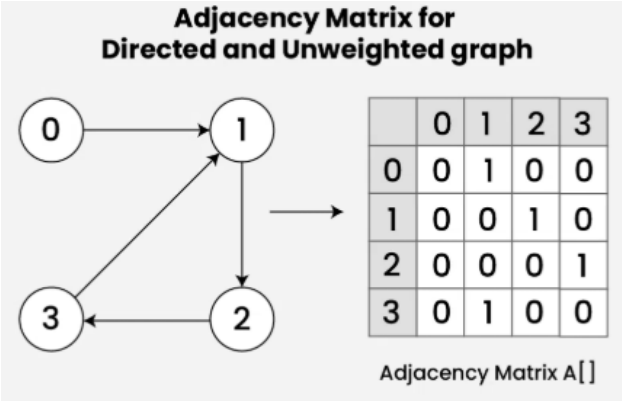
- Each *key* is a node.
- The *value* is another dictionary, where:
  - Each *key* is a **neighboring node**, and
  - Each *value* is the **weight** of the edge to that neighbor.

Q5 Code highlights – Graph representations in Python (cont.)

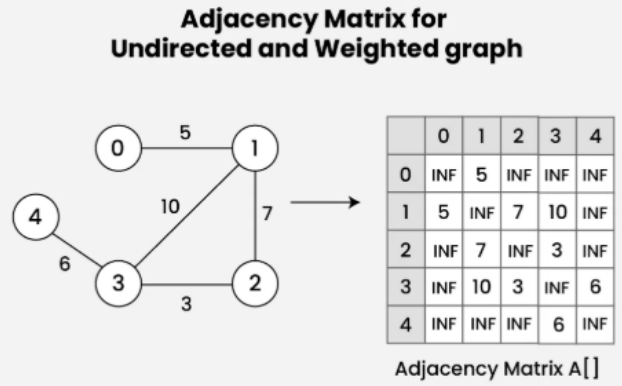
Representing graph with adjacency matrix



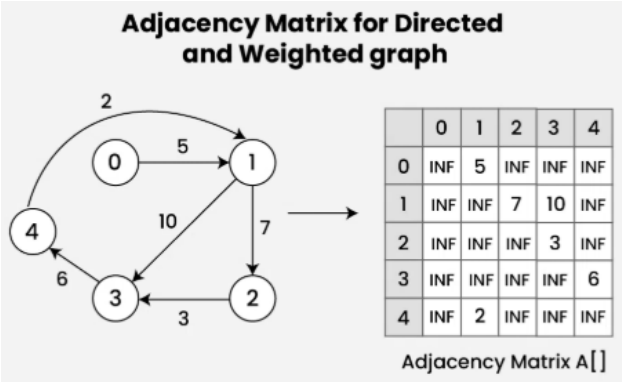
	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0



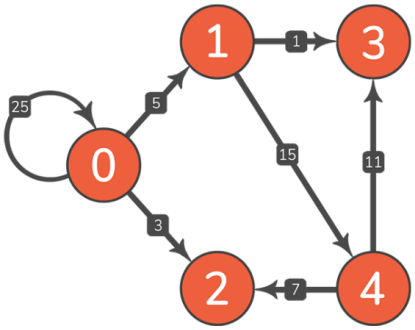
	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	INF	INF	7	10	INF
2	INF	INF	INF	3	INF
3	INF	INF	INF	INF	6
4	INF	2	INF	INF	INF

Reference: https://www.geeksforgeeks.org/dsa/adjacency-matrix/

Q5 Code highlights – Graph representations in Python (cont.)

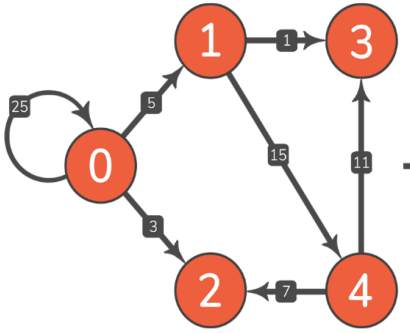


Dictionary with values assigned

weight

```
{  0: {(0, 25), (1, 5), (2, 3)},
   1: {(3, 2), (4, 15)},
   2: {},
   3: {},
   4: {(2, 7), (3, 11)},
}
```

Adjacency list/dictionary



	0	1	2	3	4
0	25	5	3	0	0
1	0	0	0	1	15
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	7	11	0

Adjacency matrix

Representation	Adjacency list/Dictionary	Adjacency matrix
Best for	Sparse graphs	Dense graphs
Traversal	Efficient (only actual neighbors)	Slower (scan all possible neighbors)
Space efficiency	Efficient: Uses less memory for sparse graphs	Consumes more space
Graph updates	Easy to add/remove nodes and edges	Less flexible (matrix resizing needed)
Typical use cases	Preferred for most real-world problems, like road maps or networks.	Dense networks (e.g., social network of close-knit friends), fixed-size graphs

Reference: https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/representing-graphs-in-code/

Q5 Code highlights: Dictionary operations

```
node_data = {node: {'dist': math.inf, 'prev': []} for node in graph}
```

Initializes a dictionary called node_data to store information about each node in the graph. It's used to track:

- The shortest distance from the source node to each node ('dist')
- The previous nodes in the shortest path ('prev')

Code break down:

`for node in graph`: **Loops** through all nodes in the graph. For each node, create an entry in the `node_data` dictionary:

-- **key:** the node itself.

-- **value:** another **dictionary** `{'dist': math.inf, 'prev': []}`:

- `'dist': math.inf` –initializes the distance to infinity, meaning the node is initially unreachable.
- `'prev': []` – starts with an empty path. This list will later be used to reconstruct the shortest path.

node	shortest distance from *a*	previous node
a	0	--
b	4	*a*
c	6	*a*
d	3	*a*
e	7	*d*
f	8	*e*

`node_data` after the search completes contains the shortest distance and path from the source to every node:

```
node_data = {
  'a': {'dist': 0, 'prev': []} ,
  'b': {'dist': 4, 'prev': ['a']} ,
  'c': {'dist': 6, 'prev': ['a']} ,
  'd': {'dist': 3, 'prev': ['a']} ,
  'e': {'dist': 7, 'prev': ['a', 'd']} ,
  'f': {'dist': 8, 'prev': ['a', 'd', 'e']} ,
}
```

Q5 Code highlights: Dictionary operations

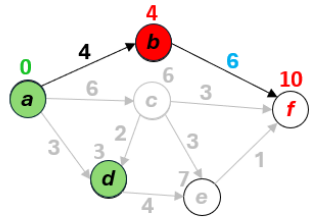
`dist = node_data[u]['dist'] + graph[u][v]`

Calculates the total distance from the source to node v, by going through node u.

`node_data[u]['dist']`: known shortest distance from the source to u

`graph[u][v]`: distance of the edge from u to v

Their sum gives the distance from the source to v via u



$$dist[f] = dist[b] + 6$$
$$= 4 + 6$$
$$= 10$$

Nested dictionary:

`node_data[u]['dist']`: u refers to the current node, and 'dist' accesses the shortest distance from the source to that node, stored in the *inner dictionary*.

`graph[u][v]`: u is the current node, and v is one of its neighbors in the *inner dictionary* (the value of u).

node_data after the search completes contains the shortest distance and path from the source to every node:

```
node_data = {
  'a': {'dist': 0, 'prev': []} ,
  'b': {'dist': 4, 'prev': ['a']} ,
  'c': {'dist': 6, 'prev': ['a']} ,
  'd': {'dist': 3, 'prev': ['a']} ,
  'e': {'dist': 7, 'prev': ['a', 'd']} ,
  'f': {'dist': 8, 'prev': ['a', 'd', 'e']} ,
}
```

```
graph = {
    'a': {'b': 4, 'c': 6, 'd': 3},
    'b': {'f': 6},
    'c': {'d': 2, 'e': 3, 'f': 3},
    'd': {'e': 4},
    'e': {'f': 1},
    'f': {},
}
```

Q5 Takeaway

Graph Representation:
The choice between an *adjacency list* and an *adjacency matrix* depends on the graph's density.

- Use an adjacency list for sparse graphs (with fewer edges) — it's more memory-efficient.
- Use an adjacency matrix for dense graphs (with many edges) — it allows faster edge lookups.

Implementation Shortcuts:
In practice, we sometimes deviate slightly from the textbook version of an algorithm to prioritize efficiency and speed.
For example, in Dijkstra's algorithm, we may allow the same node to enter the frontier multiple times, instead of updating its position in the queue — this avoids the overhead of updating.
These trade-offs prioritize efficiency while still *guaranteeing* the algorithm's *correctness*.

Want to explore more?

A similar but more challenging problem:

LeetCode #787: Cheapest Flights Within K Stops
https://leetcode.com/problems/cheapest-flights-within-k-stops/description/

- Model cities as nodes and flights (with cost) as directed weighted edges
- Find the cheapest path from source to destination within at most K stops
- Similar to minimizing signal delay across a network with constraints
- Can use a modified Dijkstra's algorithm or other graph traversal methods considering cost and stop limits.

Try it in your spare time if you're comfortable with today's material and want an extra challenge!

Reference of LeetCode #787:
https://www.youtube.com/watch?v=vWgoPTvQ3Rw
https://medium.com/@hongjje.dev/leetcode-solution-787-cheapest-flights-within-k-stops-321ca7929fa9

EE2213 Introduction to Artificial Intelligence

Tutorial 3 (Lectures 3 & 4)

Dr Shaojing Fan
fanshaojing@nus.edu.sg

Q4.

You are working on implementing A* search in a navigation app that finds the shortest path between two cities on a map. The graph is represented as an adjacency list, edge weights represent driving time, and each city has a precomputed heuristic value $h(n)$, which estimates the remaining travel time to the goal.

Below is a partially written pseudocode for the A* search algorithm. Some parts are missing.

i. Fill in the blanks (1) – (4) with the correct logic in the pseudocode.
ii. Explain how the f-score is used to prioritize nodes in the open list.
iii. The performance of A* heavily depends on the quality of the heuristic h(n). What are the consequences if the heuristic overestimates the actual cost to the goal?

Q4.

```
function A_Star(Graph, start, goal, h):
    for each node n in Graph:
        g[n] ← ________  both inf          // (1) cost from start to n
        f[n] ← ________                    // (2) estimated total cost (f = g + h)
        prev[n] ← null

    g[start] ← 0
    f[start] ← h[start]
    open ← priority queue ordered by f-value
    Insert(open, start)

    closed ← empty set

    while open is not empty:
        current ← Extract-Min(open)

        if current == goal:
            return Reconstruct-Path(prev, goal)

        Add(closed, current)

        for each neighbor in neighbors of current:
            if neighbor in closed:
                continue

            tentative_g ← g[current] + cost(current, neighbor)

            if tentative_g < g[neighbor]:
                prev[neighbor] ← ____________  curr        // (3)
                g[neighbor] ← tentative_g
                f[neighbor] ← ____________                 // (4)
                                  g[neighbor] + h [neighbor]
                if neighbor in open:
                    Update-Priority(open, neighbor)
                else:
                    Insert(open, neighbor)

    return failure
```

Q4.

Fill in the blanks (1) – (4)

```
function A_Star(Graph, start, goal, h):
    for each node n in Graph:
        g[n] ← ∞
        f[n] ← ∞
        prev[n] ← null

    g[start] ← 0
    f[start] ← h[start]
    open ← priority queue ordered by f-value
    Insert(open, start)

    closed ← empty set  Explored set

    while open is not empty:
        current ← Extract-Min(open)

        if current == goal:
            return Reconstruct-Path(prev, goal)

        Add(closed, current)

        for each neighbor in neighbors of current:
            if neighbor in closed:
                continue

            tentative_g ← g[current] + cost(current, neighbor)

            if tentative_g < g[neighbor]:
                prev[neighbor] ← current                    // (3)
                g[neighbor] ← tentative_g
                f[neighbor] ← g[neighbor] + h[neighbor]     // (4)

                if neighbor in open:
                    Update-Priority(open, neighbor)
                else:
                    Insert(open, neighbor)

    return failure
```

- g(n) – the current shortest distance from the start to node n (initial value set to $+\infty$ for all nodes)
- f(n) – the estimated total cost from the start node to the goal through node n, computed as f(n) = g(n) + h(n) (initially set to $+\infty$ for all nodes)
  - The distance from the start to itself is 0.
  - The estimated total cost from the start node to the goal is the heuristic value of the start node.
- Create a priority queue (frontier) ordered by f value.
- Push the start node into the priority queue with its f value.
- Get the node with the smallest f value from the frontier used here
- if goal, return the path
- Otherwise, mark it as visited, and start to expand the node.

For all neighbouring nodes that are not processed:

If going through current node gives a **shorter distance** to neighbour:

- update the shortest distance and f-value
- push neighbour into the priority queue or update with its new f-value

- If no path is found, return failure.

(ii) Explain how the f-score is used to prioritize nodes in the open list.

The f-score is used to estimate the total cost of a path going through a node to the goal.

- A* search always selects the node with the lowest f-score from the open list (via priority queue).
- This ensures that paths that appear promising (i.e. with low estimated total cost) are explored first.

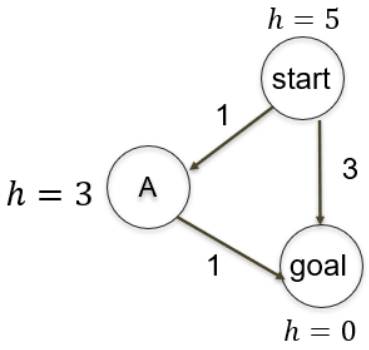
- Mathematically:

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

where $g(n)$ is the known cost from the start to n, and $h(n)$ is the heuristic estimate from n to the goal.

(iii) The performance of A* heavily depends on the quality of the heuristic $h(n)$. What are the consequences if the heuristic overestimates the actual cost to the goal?

If the heuristic overestimates the actual cost to the goal, A* is no longer guaranteed to find the optimal path. The algorithm may incorrectly dismiss the true shortest path because the overestimated heuristic makes it seem more expensive than it actually is. As a result, A* might explore a different path that looks better under the faulty estimate, leading to a valid solution — but not necessarily the shortest one.



Q5. (Optional – For Exploration)

Extend the STRIPS planner from Lecture 4 to model a robot performing a multi-step coffee-making process.

Please refer to TUT3-Q5.py for sample implementation.

Additional references

Python networkx library:
https://networkx.org/documentation/networkx-0.37/

Planning: STRIPS:
https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/strips.pdf

https://www.geeksforgeeks.org/artificial-intelligence/strips-in-ai/