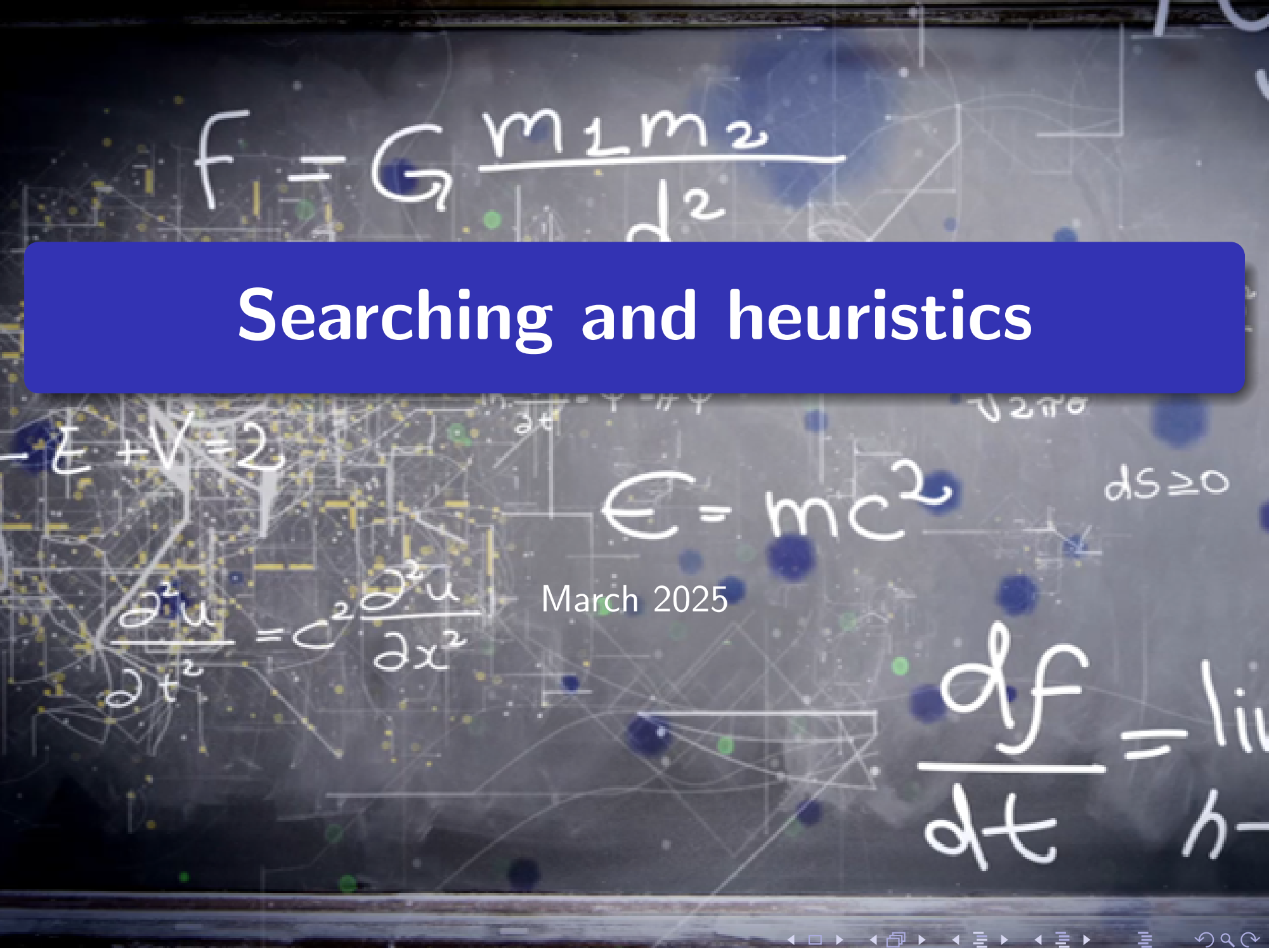


# Searching and heuristics

March 2025

# Topics to be discussed

**Part I. Introduction**

1. Introduction to Artificial intelligence

**Part II. Search and optimisation.**

2. Searching and heuristics
3. Search - optimisation
4. Two-player deterministic games
5. Evolutionary and genetic algorithms

**Part III. Machine learning and data analysis.**

6. Regression, classification and clustering (Part I & II)
8. Artificial neural networks
9. Bayesian models
10. Reinforcement Learning

**Part IV. Logic, Inference, Knowledge Representation**

11. Propositional logic and predicate logic
12. Knowledge Representation

**Part V. AI in Action: Language, Vision**

13. AI in Natural language processing
14. AI in Vision and Perception

**Part VI. Summary**

15. AI engineering, Explainable AI, Ethics

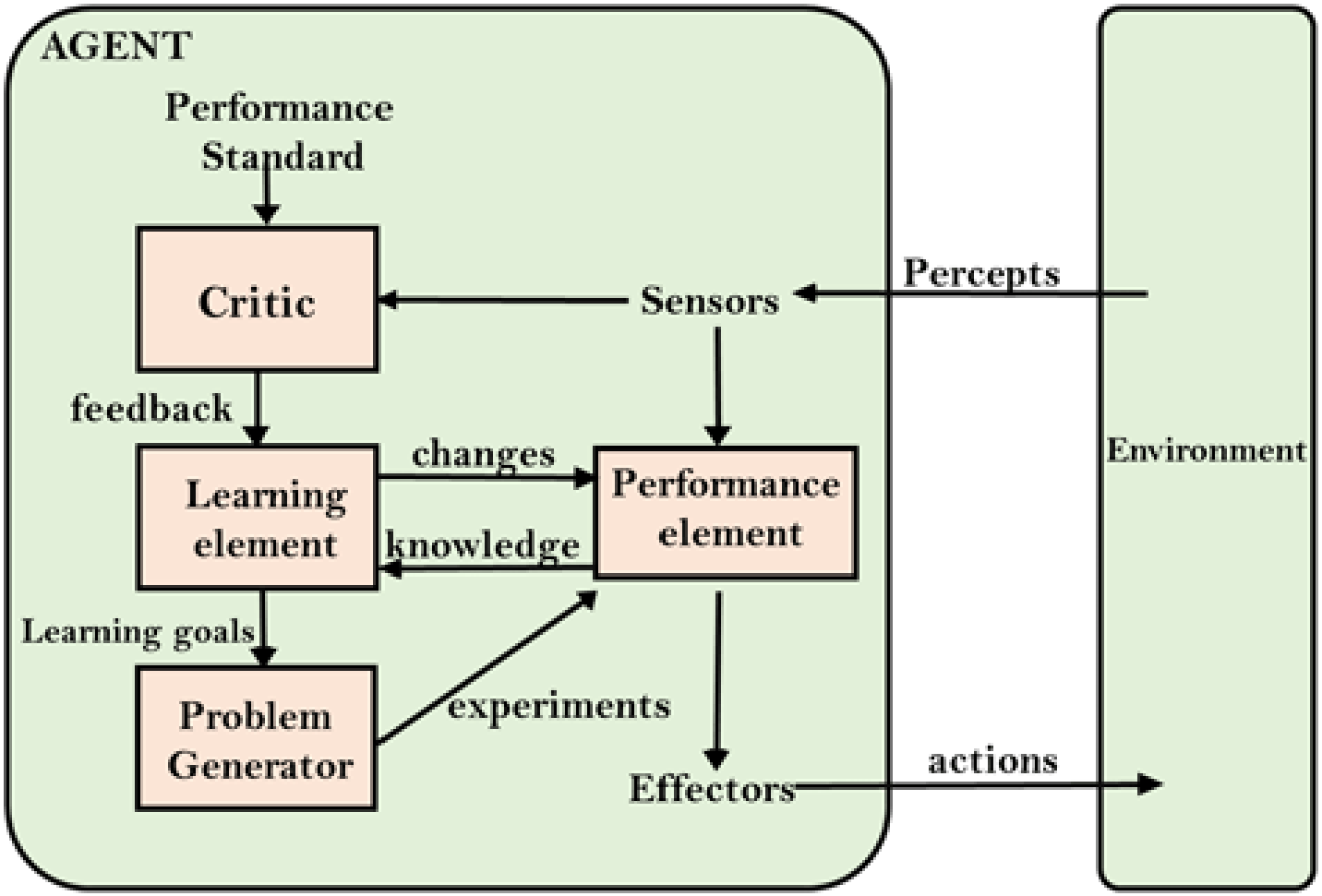
**Overview**

- Introduction
- Types of basic search methods in AI
  - Uninformed search methods
  - Informed search methods
- Comparing uninformed and informed search methods
- Applications of search methods in AI

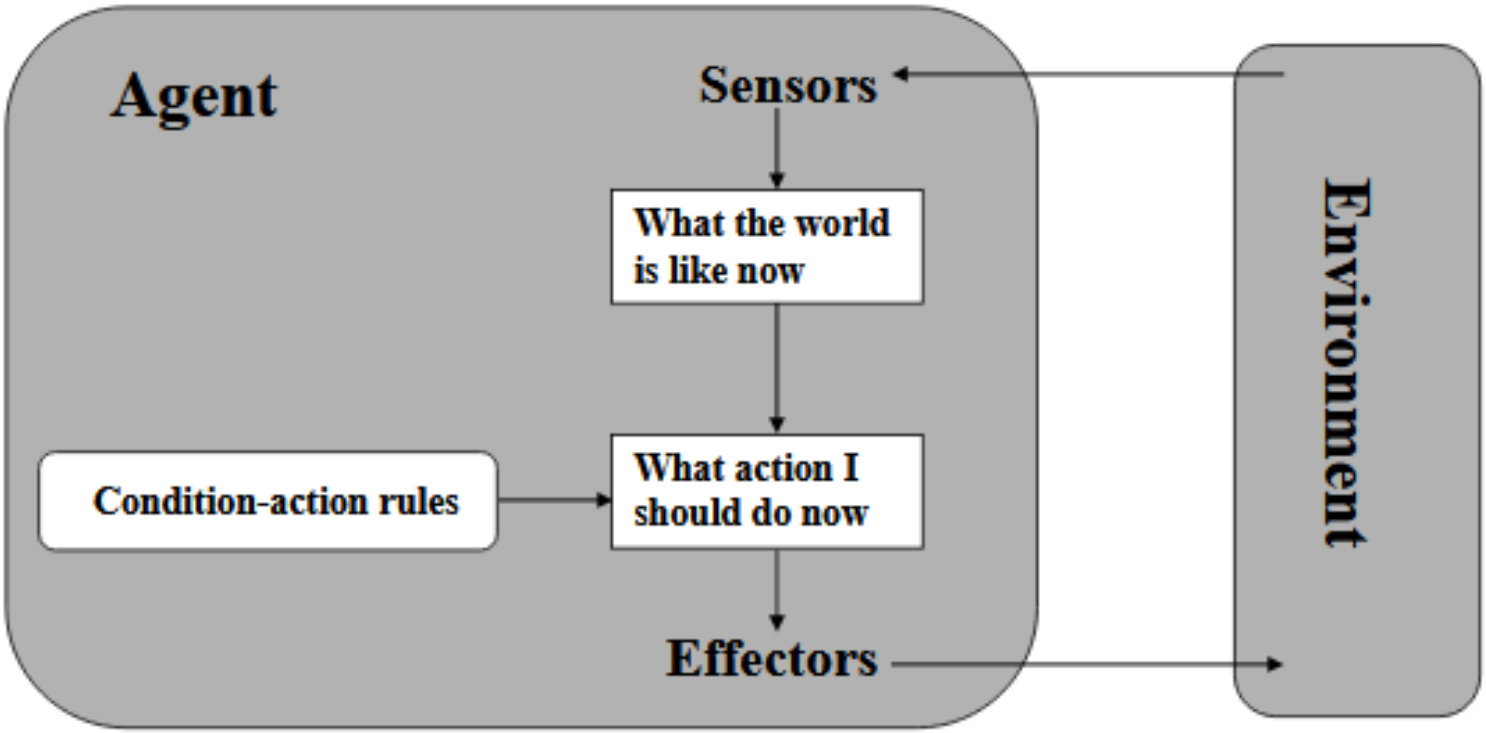
# Two views of AI

- **AI agents**: how can we create intelligence?
  - The inspiration comes from the types of capabilities that humans possess.
- **AI tools**: how can we benefit society?
  - Need comes from existing problems in the world, and techniques developed by the AI community happen to be useful for that.

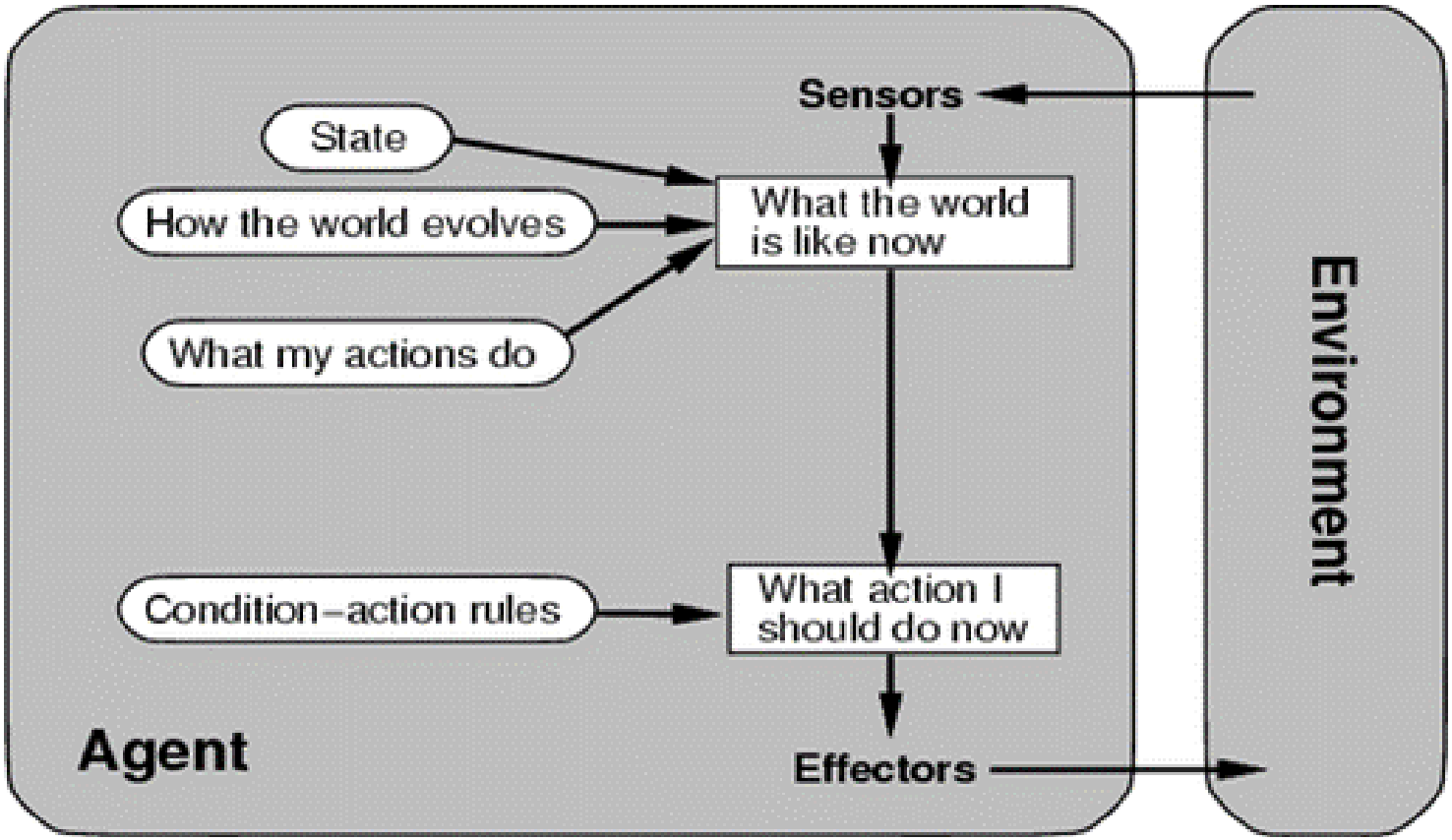
# Intelligent Agents in AI



# A simple Reflex Agent



# Agent that keeps track of the world



# What are search methods in AI?

- Search methods are techniques used by AI systems to find solutions to problems by searching through a set of possible solutions.
- These methods involve generating a set of candidate solutions and evaluating them based on some objective criteria.
- Different types of basic search methods exist, from uninformed methods like depth-first search and breadth-first search to informed methods like A* search and best-first search.

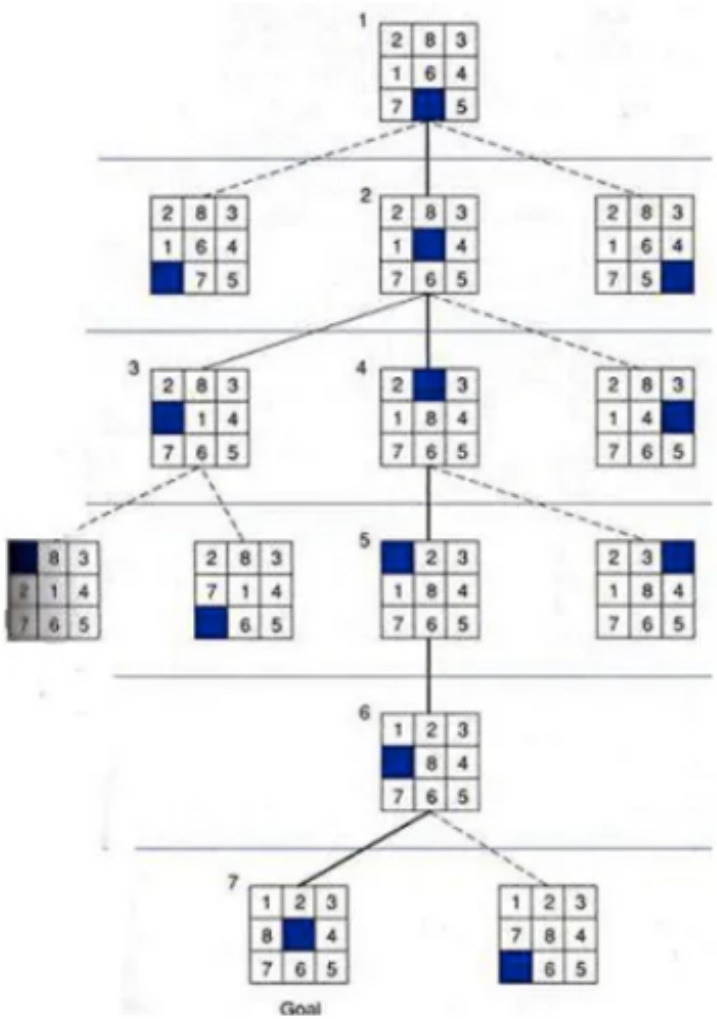
# Examples of classical" applications of search in AI

- In games such as chess or checkers, AI programs use search methods to explore different possible moves and their outcomes to find the best move.
- In natural language processing, search methods can be used to find the most likely interpretations of a given sentence by searching through possible syntax trees or semantic representations.
- In robotics, search methods can be used to plan the most efficient path for a robot to navigate through an environment by searching through a set of possible paths.
- In computer vision, search methods can be used in applications such as face recognition by searching through a database of images to find a match.

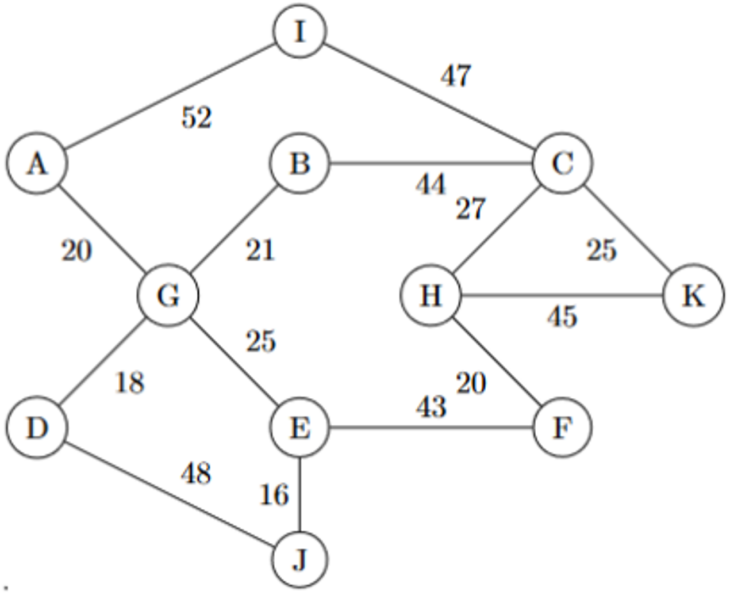
# Example: Toy problems

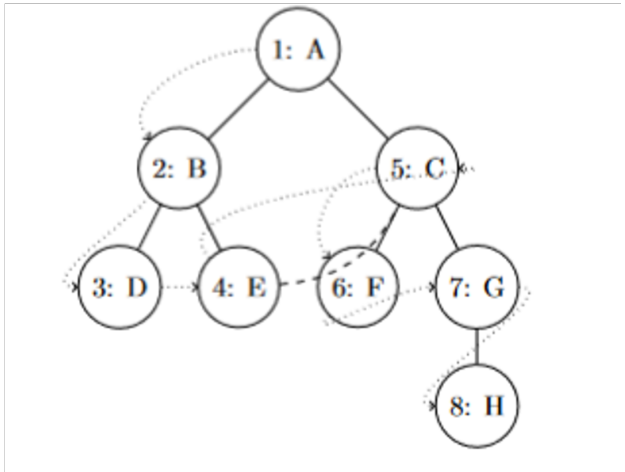
- N queens puzzle: how to place N queens on a N×N board so that none is attacked by any other
- Two containers have a capacity of 5 and 7 liters. How, by filling them to the full, emptying and pouring water between them, measure 4 liters?
- Traveling salesman problem

# 8 puzzle problem



# Graphs and networks



# State graph definitions (1)

- **Graph**: A set of vertices $V$ and a set of edges $E$, where each edge is a pair $(u, v)$ representing a connection between vertices $u$ and $v$.
- **Node/Vertex**: A point in the graph.
- **Edge**: A connection between two vertices $u$ and $v$.
- **Directed/Undirected Graph**: A directed graph has edges with a direction, while an undirected graph has edges without a direction.
- **Adjacency List**: A data structure used to represent a graph as a list of vertices, where each vertex has a list of its adjacent vertices.
- **Adjacency Matrix**: A two-dimensional matrix used to represent a graph, where the rows and columns correspond to the vertices, and the entries represent the existence of an edge between the corresponding vertices.
- **Traversal**: Visiting all the nodes/vertices of a graph in a systematic order.

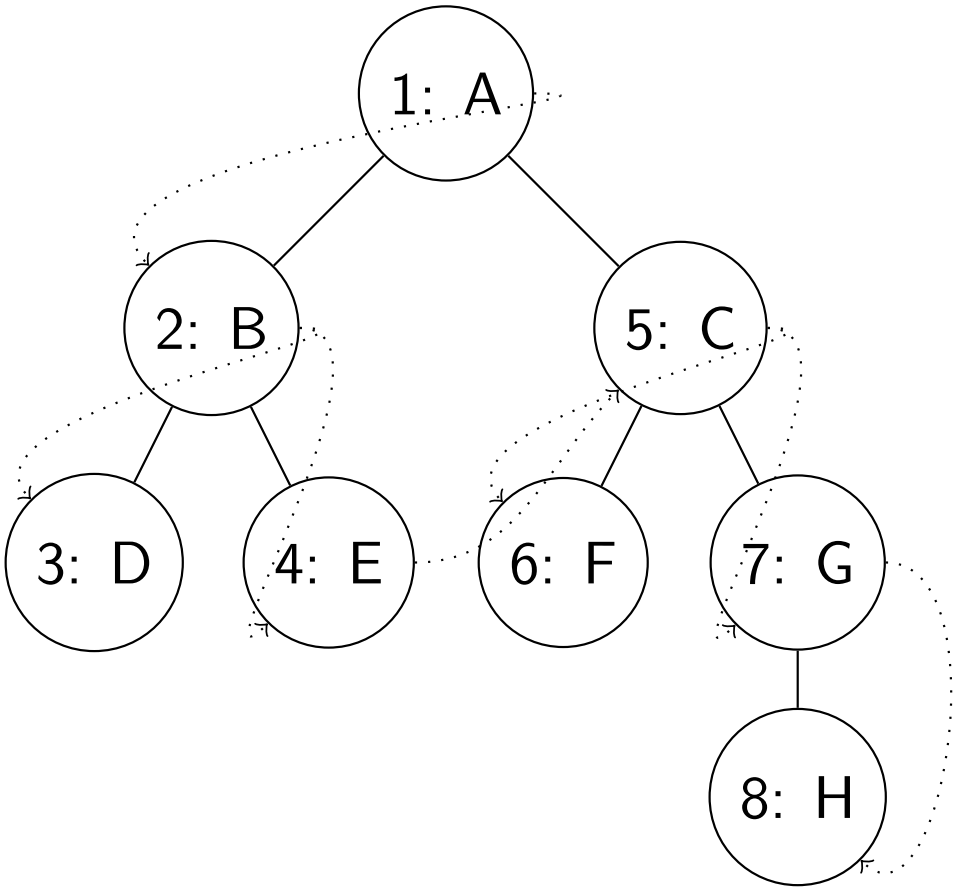
# State graph definitions (2)

- **Stack**: A data structure that follows the Last-In-First-Out (LIFO) principle, i.e., the last element added to the stack is the first one to be removed.
- **Queue**: A data structure that follows the First-In-First-Out (FIFO) principle, i.e., the first element added to the queue is the first one to be removed.
- **Visited Array/Set**: A data structure that keeps track of the nodes/vertices that have already been visited by an algorithm.
- **Path**: A sequence of nodes/vertices that represents a route between two nodes/vertices in a graph.
- **Cost/Distance**: A numerical value associated with an edge that represents the "cost" or "distance" of traversing that edge.
- **Search Space**: The set of all possible states/nodes that can be explored by a search algorithm.
- **Heuristic Function**: A function that estimates the "cost" or "distance" of reaching the goal node from a given node, and is used to guide a search algorithm towards the goal node.

# Two basic graph search algorithms

- **BFS (Breadth-First Search)**: A graph traversal algorithm that visits all the nodes/vertices of a graph in breadth-first order, i.e., it visits all the nodes at the current depth/level before moving on to the nodes at the next depth/level.
- **DFS (Depth-First Search)**: A graph traversal algorithm that visits all the nodes/vertices of a graph in depth-first order, i.e., it explores as far as possible along each branch before backtracking.

# Depth-First Search, FS

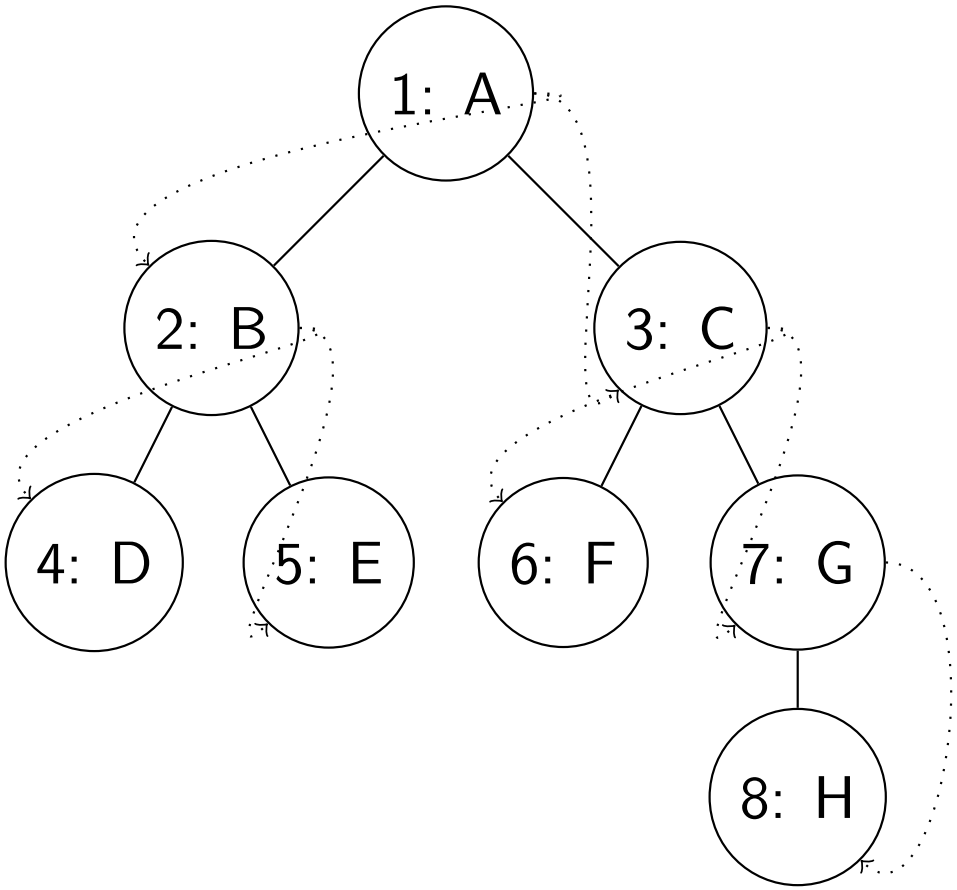


# Depth-First Search, DFS

**Algorithm 1:** DFS Algorithm

**Result:** Visited vertices in DFS order
initialize stack with starting vertex
initialize visited set as empty
initialize result list as empty
**while** *stack is not empty* **do**
    pop vertex $v$ from stack
    **if** *vertex $v$ has not been visited* **then**
        mark vertex $v$ as visited
        add vertex $v$ to result list
        **foreach** *neighbor $u$ of vertex $v$ in reverse order* **do**
            **if** *vertex $u$ has not been visited* **then**
                push vertex $u$ onto stack
                add dotted arrow from $v$ to $u$
            **end**
        **end**
    **end**
**end**

# Breadth-First Search, BFS



# Breadth-First Search, BFS

**Algorithm 2:** Breadth-First Search

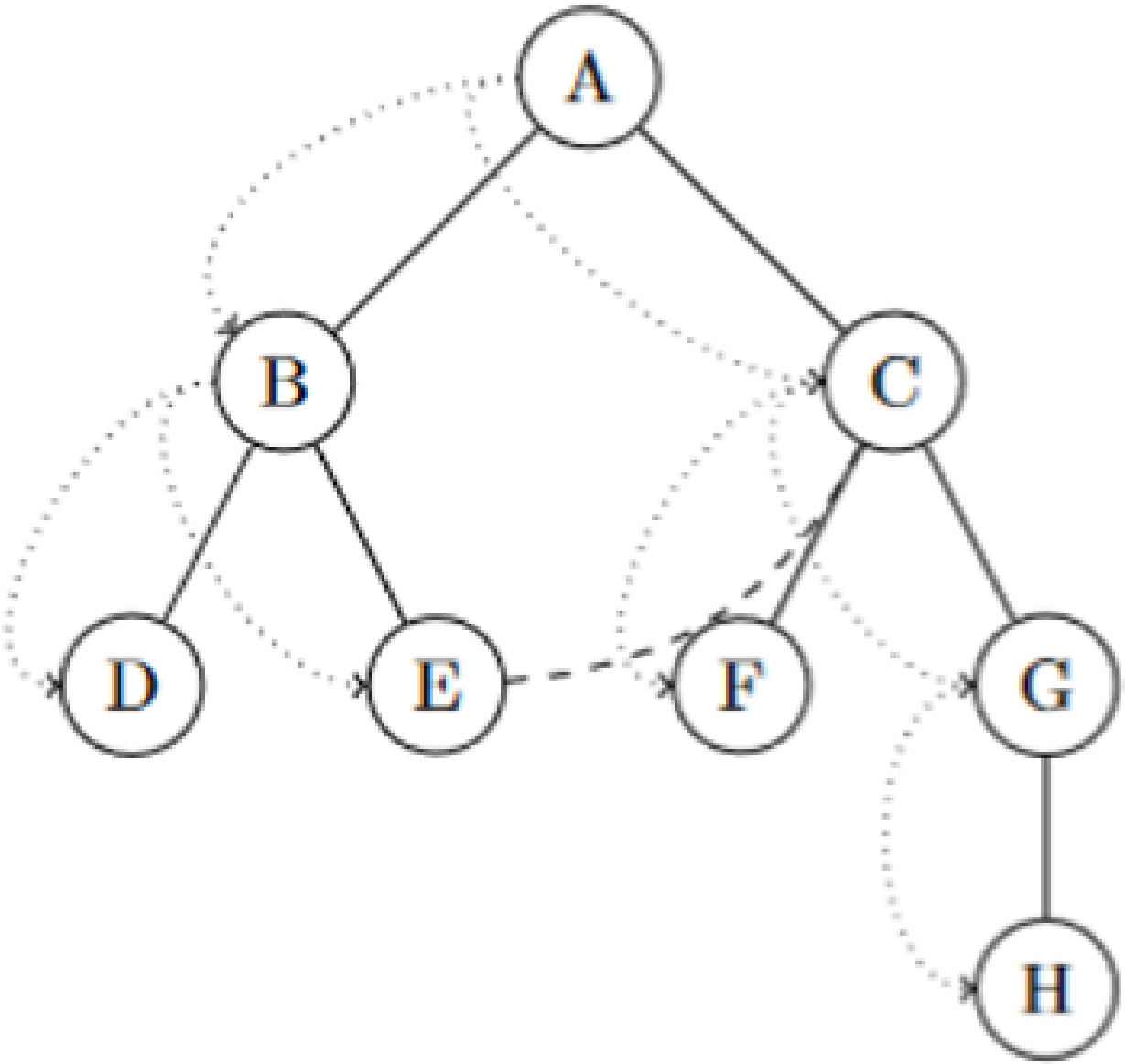
**Input:** Graph $G = (V, E)$ and starting vertex $s$
**Output:** Visited vertices

```
foreach vertex v ∈ V do
    visited[v] ← false;
end
queue ← [s];
visited[s] ← true;
while queue ≠ [] do
    u ← queue.pop(0);
    foreach vertex v adjacent to u do
        if visited[v] = false then
            visited[v] ← true;
            queue.append(v);
        end
    end
end
```

# BFS vs. DFS - properties summary

- BFS finds a shortest sequence, it needs less time but more memory
- DFS needs less memory, but is not guaranteed to find a shortest sequence

# Iterative Deepening in Depth-First Search



# Iterative Deepening Depth-First Search, IDDFS

**Algorithm 3:** Iterative Deepening Depth-First Search (IDDFS)

**Data:** Graph G, Start vertex s

**Result:** The sequence of visited vertices during the search

**Function** `DFS(`*G, s, depth*`)`**:**

- **if** *depth* $\leq 0$ **then**
  - **return**
- Mark s as visited;
- **forall** *neighbors n of s* **do**
  - **if** *n is not visited* **then**
    - `DFS(`*G, n, depth-1*`);`

**for** *depth* $= 1$ **to** $\infty$ **do**

- Run DFS on G with starting vertex s and maximum depth limit depth;
- **if** *all vertices are visited* **then**
  - **return** visited sequence;

# Bidirectional search

- **Bidirectional search** is a graph search algorithm that searches two ends of the graph simultaneously, starting from the initial and goal nodes.
- In each step of the search, it expands the node that has not been expanded before and has the lowest cost.
- Once both searches meet at a common node, the search is complete and returns the path from the start to the goal node.
- Bidirectional search is particularly useful when the search space is large, and the branching factor is high, as it can significantly reduce the number of nodes that need to be explored.
- It can also be more efficient when the graph is structured, such as in a tree or a grid, and the start and goal nodes are close to each other.
- However, bidirectional search requires the ability to efficiently check if a node has been visited, and it also requires a way to efficiently check if a node is a common node between the two searches.

## **Algorithm 4:** Bidirectional Search Algorithm

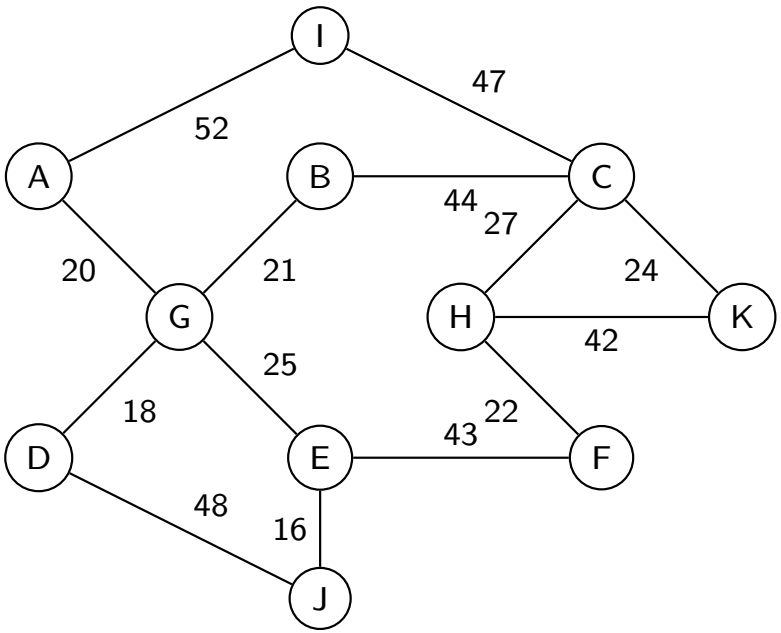
**Input:** start node $s$ and goal node $g$
**Output:** shortest path from $s$ to $g$

```
forwardFrontier ← {s};
backwardFrontier ← {g};
forwardExplored ← {};
backwardExplored ← {};
while forwardFrontier ≠ ∅ and backwardFrontier ≠ ∅ do
    if |forwardFrontier| ≤ |backwardFrontier| then
        current ← a node from forwardFrontier;
        remove current from forwardFrontier;
        if current ∈ backwardFrontier or current ∈ backwardExplored then
            return path from s to current and path from g to current;
        end
        add current to forwardExplored;
        for each neighbor n of current not in forwardExplored do
            add n to forwardFrontier;
        end
    end
    else
        current ← a node from backwardFrontier;
        remove current from backwardFrontier;
        if current ∈ forwardFrontier or current ∈ forwardExplored then
            return path from s to current and path from g to current;
        end
        add current to backwardExplored;
        for each neighbor n of current not in backwardExplored do
            add n to backwardFrontier;
        end
    end
end
return no path found;
```

# Finding the Shortest Path

- Suppose we have a network of cities and want to find the shortest path between two of them.
- We can represent the network as a graph, where each city is a node and the distance between them is the cost of the edge connecting them.
- To find the shortest path, we can use a search algorithm that explores the graph and keeps track of the current best path.
- One such algorithm is Dijkstra's algorithm, which starts at the source city and repeatedly selects the node with the lowest distance until the destination city is reached.
- Alternatively, we can use a heuristic search algorithm like A* that incorporates information about the estimated remaining distance to the goal city to guide the search.

# Uniform cost search – an example




| Node | Visited? | Total Cost |
|---|---|---|
| A | Yes | 0 |
| G | Yes | 20 |
| D | Yes | 38 |
| B | Yes | 41 |
| E | Yes | 45 |
| I | Yes | 52 |
| J | Yes | 61 |
| C | Yes | 85 |
| F | Yes | 88 |
| H | Yes | 100 |
| K | Yes | 123 |

# Uniform cost search

## **Algorithm 5:** Uniform Cost Algorithm

**Input:** start node s, goal node g
**Data:** edge cost function c, set of nodes V, set of edges E
**Output:** shortest path from s to g

```
foreach node v in V do
    d[v] ← ∞ ;                                   // initialize all distances as infinity
end
d[s] ← 0 ;                                       // distance from start to start is 0
frontier ← PriorityQueue() ;                     // initialize the priority queue
frontier.put(s, 0) ;                             // put the start node in the queue with priority 0
while frontier is not empty do
    u ← frontier.get() ;                         // get the node with the lowest cost so far
    if u = g then
        return construct_path(s, g) ;            // goal reached, return the path
    end
    foreach neighbor v of u do
        cost ← c(u, v) ;                         // get the cost of the edge from u to v
        new_distance ← d[u] + cost ;             // calculate the new distance to v via u
        if new_distance < d[v] then
            d[v] ← new_distance ;                // update the distance to v
            frontier.put(v, d[v]) ;              // add v to the queue with updated priority
            prev[v] ← u ;                        // update the parent of v to u
        end
    end
end
return None ;                                    // goal not reachable from start
```

# Greedy Best First Algorithm

- Greedy Best-First Search is an informed search algorithm that uses a heuristic function $h(n)$ to guide the search towards the goal state.
- At each step, the algorithm expands the node with the lowest heuristic value, i.e., the node that is closest to the goal according to the heuristic function.
- The algorithm maintains a priority queue called the frontier, which contains the nodes to be expanded in the search tree.
- The heuristic function must be admissible, i.e., it should never overestimate the true cost of reaching the goal state.
- Greedy Best-First Search is not guaranteed to find the optimal solution, but it can be very efficient if the heuristic function is well-designed.

# Greedy Best First Algorithm

**Algorithm 6:** Greedy Best-First Search

**Input:** initial state $s_0$, heuristic function $h$

**Output:** goal state, or failure

```
frontier ← {s_0}
explored ← ∅
while frontier ≠ ∅ do
    s ← state in frontier with lowest h(s)
    if s is goal state then
        return s
    Move s from frontier to explored
    for each successor s' of s do
        if s' ∉ frontier or h(s') < h(s) then
            Add s' to frontier
return failure
```

# A* algorithm

- A* pathfinding algorithm that finds the shortest path between two nodes in a weighted graph
- Works by combining the cost of the path from the start node to the current node (g-value) with the estimated cost of the path from the current node to the goal node (h-value)
- The f-value of a node is the sum of the g-value and the h-value: $f(n) = g(n) + h(n)$
- Keeps track of the f-values of all nodes in a priority queue, sorted by lowest f-value
- Expands the node with the lowest f-value, updating the f, g, and h values of its neighbors as needed
- Stops when the goal node is reached, or all paths have been explored

# A* algorithm

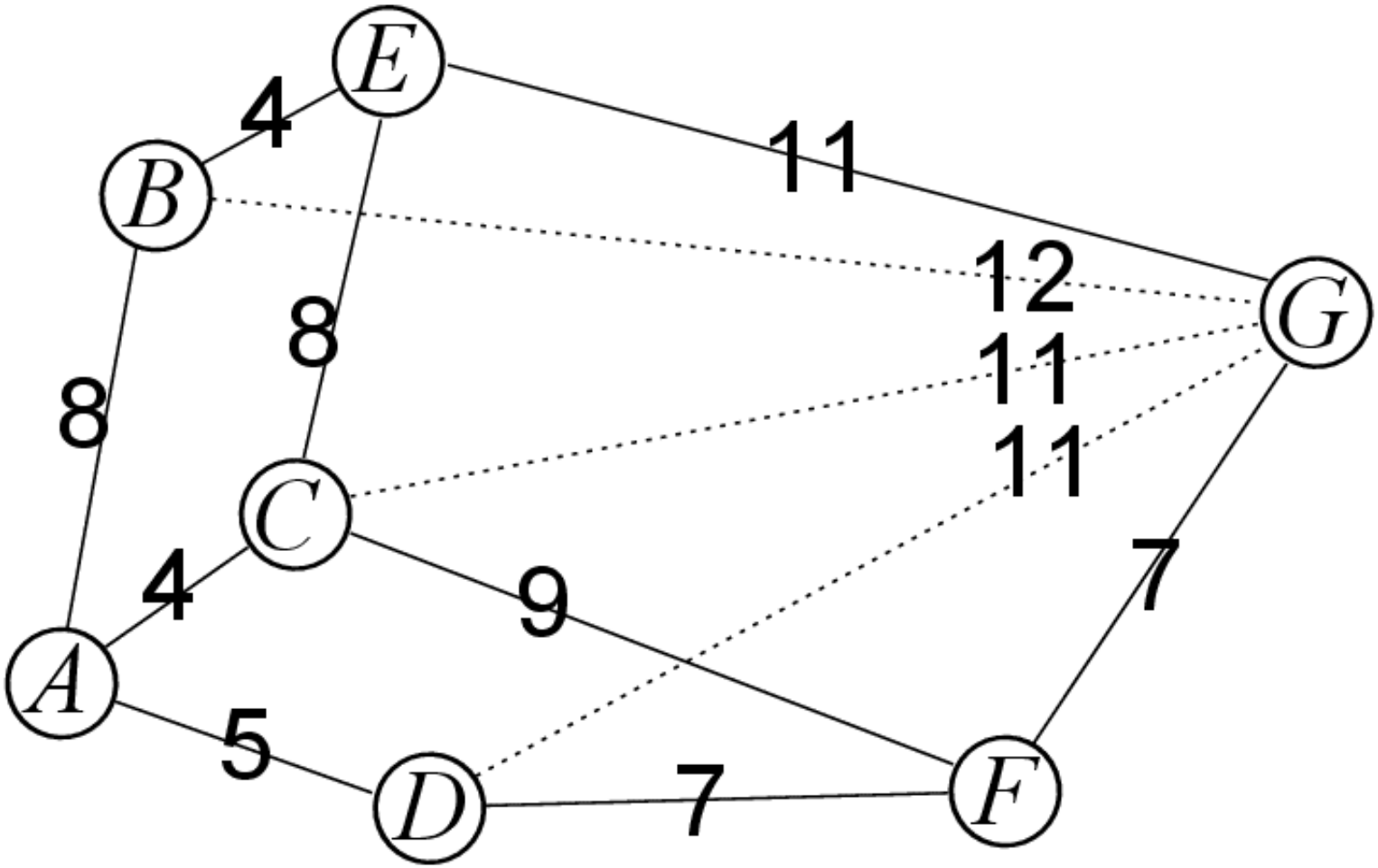
**Algorithm 7:** A* Algorithm

**Input:** Start node $s$, goal node $g$, heuristic function $h$, cost function $c$
**Output:** Shortest path from $s$ to $g$

```
frontier ← {s}
came_from ← {}
cost_so_far[s] ← 0
while frontier is not empty do
    current ← node in frontier with lowest f value
    if current = g then
        return reconstruct_path(came_from, s, g)

    remove current from frontier
    foreach neighbor next of current do
        new_cost ← cost_so_far[current] + c(current, next)
        if next not in cost_so_far or new_cost < cost_so_far[next] then
            cost_so_far[next] ← new_cost
            priority ← new_cost + h(next, g)
            add next to frontier with priority priority
            came_from[next] ← current
return null
```

# A* algorithm – an example



# The heuristic h function

- Relatively simple to evaluate
- Can be a solution to a simplified problem
  - By ignoring certain constraints
  - It is well if the simplified problem has an analytical solution
  - It can be evaluated through on-line solving of the simplified problem

# IDA* (Iterative Deepening A*)

- An informed search algorithm that finds the shortest path between two nodes in a weighted graph
- Combines the advantages of A* and iterative deepening depth-first search
- Works by searching the graph repeatedly with increasing depth limits until a solution is found
- Uses a heuristic function to estimate the remaining cost to reach the goal node, represented by $h(n)$
- The cost of the path from the start node to a node $n$ is represented by $g(n)$
- The total estimated cost of the path through node $n$ to the goal node is represented by $f(n) = g(n) + h(n)$
- At each iteration, nodes are expanded in increasing order of their $f$-values until the goal node is reached
- If the goal node is not reached within the depth limit, the search is repeated with a larger depth limit

# IDA* (Iterative Deepening A*)

## **Algorithm 8:** Iterative Deepening A* Algorithm

**Input :** start node $s$, goal node $t$
**Output:** shortest path from $s$ to $t$

```
foreach node n do
    g(n) ← ∞
    f(n) ← ∞
    closed(n) ← false
g(s) ← 0
f(s) ← h(s)
for depth ← 0 to ∞ do
    open ← priority queue with s as the only element
    while open is not empty do
        n ← node with minimum f(n) value in open
        if n is the goal node then
            return the path from s to n
        closed(n) ← true
        foreach successor s of n do
            if s is not in closed or g(s) > g(n) + w(n, s) then
                g(s) ← g(n) + w(n, s)
                f(s) ← g(s) + h(s)
                if depth > 0 then
                    add s to open
return failure
```

# Beam Search

- Beam search is a heuristic search algorithm that explores a graph by expanding the most promising nodes at each level of the search.
- **Strengths:**
  - Beam search is an improvement over simple best-first search, as it uses a beam width parameter to reduce the search space and avoid getting stuck in local optima.
  - It can be easily parallelized by processing different branches of the search space concurrently.
  - It can handle large search spaces and can be effective for finding approximate solutions to problems where the exact solution is not required.
- **Weaknesses:**
  - Beam search is not guaranteed to find the optimal solution, as it may discard the correct path early in the search process.
  - The performance of beam search depends heavily on the choice of the beam width parameter, which can be difficult to determine in advance.
  - Beam search may get stuck in local optima if the search space has many similar promising nodes, and the beam width is too narrow.

# Beam Search

**Algorithm 9:** Beam search algorithm

**Input:** Starting node $s$, beam width $k$, goal node $g$
**Output:** Shortest path from $s$ to $g$

$B \leftarrow \{s\}$;
**while** $g$ *not in* $B$ **do**
    $B' \leftarrow \emptyset$;
    **foreach** *node* $n$ *in* $B$ **do**
        **foreach** *successor* $s$ *of* $n$ **do**
            $B' \leftarrow B' \cup \{s\}$;
        **end**
    **end**
    Sort nodes in $B'$ based on cost from $s$ + heuristic estimate to $g$;
    $B \leftarrow$ the $k$ best nodes in $B'$;
**end**
**return** shortest path from $s$ to $g$ found in $B$;

# Algorithm Comparison on time, memory, completeness and optimality

| Algorithm | Time | Space | Complete | Optimal |
|---|---|---|---|---|
| Breadth First | $O(b^d)$ | $O(b^d)$ | Yes | Yes |
| Uniform Cost Search | $O(b^{1+floor(C^*/\epsilon)})$ | $O(b^{1+floor(C^*/\epsilon)})$ | Yes | Yes |
| Depth First | $O(b^m)$ | $O(bm)$ | No | No |
| Depth Limited | $O(b^l)$ | $O(bl)$ | No | No |
| Iterative Deepening | $O(b^d)$ | $O(bd)$ | Yes | Yes |
| Bidirectional | $O(b^{d/2})$ | $O(b^{d/2})$ | Yes | Yes |

# Comparison and recommendations

| Algorithm | Strengths | Weaknesses |
|---|---|---|
| Best First | • Can be implemented efficiently<br>• Useful when optimality is not critical<br>• Useful when heuristic function h is accurate | • Does not guarantee optimal solution<br>• May not be suitable for problems with large state spaces |
| A* | • Guaranteed to find optimal solution, provided heuristic function h is admissible and consistent<br>• Critical when optimality is important | • Can be memory-intensive<br>• May not be suitable for problems with large state spaces |
| IDA* | • Limits amount of memory used by expanding nodes incrementally<br>• Can be used even when memory requirements exceed available resources | • Can be less efficient than A*<br>• May require a large number of iterations to find optimal solution |

# Summary - Searching and heuristics methods

**Overview**

- Introduction
- Types of basic search methods in AI
  - Uninformed search methods
  - Informed search methods
- Comparing uninformed and informed search methods
- Applications of search methods in AI

# References


1. S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach", Financial Times Prentice Hall, 2019.
2. M. Flasiński, Introduction to Artificial Intelligence", Springer Verlang, 2016
3. M. Muraszkiewicz, R Nowak (ed.), Sztuczna Inteligencja dla inżynierów", Oficyna Wydawnicza PW, 2022
4. J. Prateek , Artificial Intelligence with Python", Packt 2017