Chapter 2

# Search: Formalization

Introduction to Artificial Intelligence

Chandra Gummaluru
University of Toronto

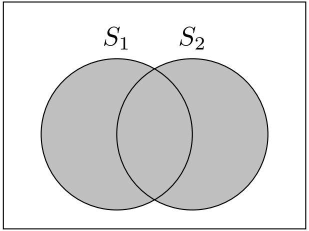
Version W22.1

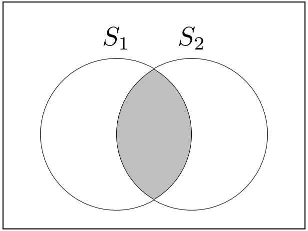
- The following is based on material developed by many individuals, including (but not limited to):
  - Sheila McIlraith
  - Bahar Aameri
  - Fahiem Bacchus
  - Sonya Allin

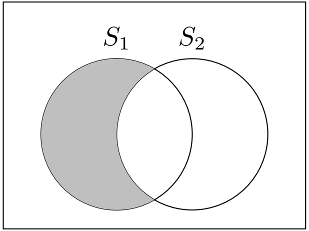
- A **set** can be thought of as a list of objects, called its **elements**:
  - We write $s \in \mathcal{S}$ to denote that $s$ is an element of the set $\mathcal{S}$.
  - The **empty set**, i.e., the set with no elements, is denoted $\emptyset$.
- A set, $\mathcal{S}'$ is a **subset** of another set, $\mathcal{S}$, denoted $\mathcal{S}' \subseteq \mathcal{S}$ if $\mathcal{S}$ contains all the elements of $\mathcal{S}'$.
  - We say that $\mathcal{S}'$ is a **proper subset** of $\mathcal{S}$, denoted $\mathcal{S}' \subset \mathcal{S}$, if $\mathcal{S}$ contains at least one element not contained in $\mathcal{S}'$.
- The **power-set** of a set, $\mathcal{S}$ denoted $\mathcal{P}(\mathcal{S})$ is the set of all of $\mathcal{S}$'s subsets.

- There are several operations we can perform on sets:
  - The **union** of two sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, denoted $\mathcal{S}_1 \cup \mathcal{S}_2$, is the set of elements contained in either $\mathcal{S}_1$ or $\mathcal{S}_2$.
  - The **intersection** of two sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, denoted $\mathcal{S}_1 \cap \mathcal{S}_2$, is the set of elements contained in both $\mathcal{S}_1$ and $\mathcal{S}_2$:
    - If $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$, we say that $\mathcal{S}_1$ and $\mathcal{S}_2$ are **disjoint**.
  - The **difference** beween two sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, denoted $\mathcal{S}_1 \setminus \mathcal{S}_2$, is the set of elements contained in $\mathcal{S}_1$ but not in $\mathcal{S}_2$:
    - Whenever $\mathcal{S}_1$ is obvious from context, we simply write $\neg\mathcal{S}_2$ to denote $\mathcal{S}_1 \setminus \mathcal{S}_2$, and call it the **compliment** of $\mathcal{S}_2$

- Below are pictorial representations of the union, intersection, and difference operators.



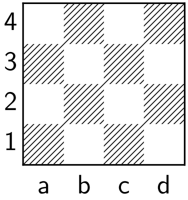



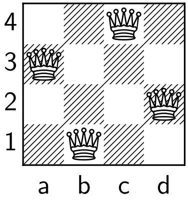
- We define a search problem as follows:

- **Definition**: Search Problem
  - Let $\mathcal{S}$ be a set of **states** that we want to search through.
  - From any given state, $s \in \mathcal{S}$, there exist a set of **actions**, $A(s)$.
  - When an action, $a \in A(s)$, is applied to $s$, the result is a new state, denoted $a(s)$.
  - A sequence of actions, $\langle a_1, \ldots, a_n \rangle$ defines a **path** between two states.
  - The **length** of the path is the number of actions that make it up.
  - Each action, $a$, may have an associated **cost**, $c(a) > 0$. In this case, the cost of the path is the cumulative cost of its actions.
  - Given some initial state, $s_0$, we seek a path (often the shortest/cheapest one) to some state in a subset, $\mathcal{G} \subseteq \mathcal{S}$, called the **goal space**.

**Example:** $N$-Queens Puzzle

- Given an $N \times N$ board with $N$ queens on it, move them using the rules of Chess so that none of the queens attack each other.
- The search is over the set of all board configurations.
- The element we seek is the specific configuration in which none of the queens attack each other.
- Finding such board configurations is non-trivial.
- However, given a board configuration, it is easy to check that it is a valid solution.

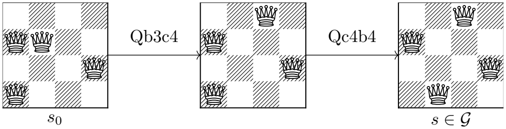


- One way to perform a search is as follows:
  1. Check if the current state, $s$ is the goal.
  2. If not, perform an action, $a \in A(s)$, resulting in a new state, $s' = a(s)$.
  3. Set the current state to $s'$ and repeat until a goal is found.

**Example:** Searching in $N$-Queens

- In the $N$-queens puzzle, we define $\mathcal{S}$ as the set of all possible board configurations, and $\mathcal{G}$ as the subset in which no two queens attack each other.
- We can search for a goal by starting with an arbitrary placement of the queens, and move them one at a time to achieve the desired configuration.



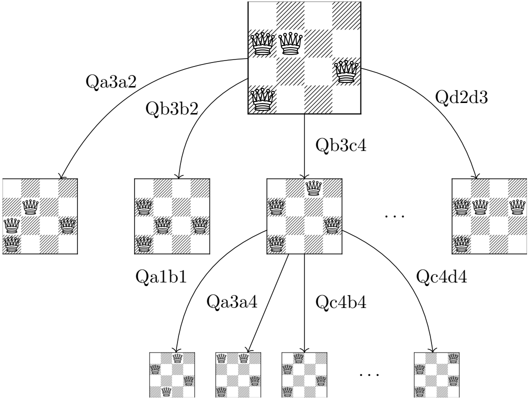
# General Search Algorithm: Pseudo-code

- We define a structure, $\mathcal{F}$, called the **frontier**, which stores discovered but unexplored states.
- To search, we remove a state, $s$, from $\mathcal{F}$. If $s$ is the goal, the search is complete. Otherwise, we add the successors of $s$ to $\mathcal{F}$ and search again.

$\mathcal{F} \leftarrow \{s_0\}$ ▷ initialize $\mathcal{F}$ with $s_0$<br>
**procedure** $\textsc{Search}(\mathcal{F})$<br>
    **if** $\mathcal{F} = \emptyset$ **then** ▷ the search failed to find a goal<br>
        **return** `NULL`<br>
    $s \leftarrow \textsc{Remove}(\mathcal{F})$<br>
    **if** $s \in \mathcal{G}$ **then**<br>
        **return** $s$<br>
    $\mathcal{F} \leftarrow \mathcal{F} \cup S(s)$ ▷ add $s$'s successors to $\mathcal{F}$<br>
    $\textsc{Search}(\mathcal{F})$

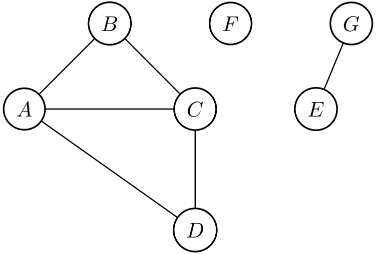
- It turns out that the search takes place on a tree.

**Example**: Search Tree for *N*-Queens

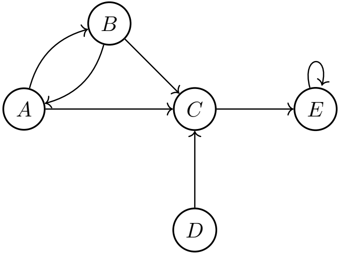


- A **tree** is a structure that is best described as a special case of another strcture called a "graph".
- A **graph** is used to model relationships between elements of a set, $\mathcal{V}$, called its **vertices** by connecting them through a set of **edges**, $\mathcal{E}$.
- An edge between two vertices, $u, v \in \mathcal{V}$ can be
  - **directed**, in which case, each edge is represented as a ordered-pair, $(u, v)$
  - **undirected**, in which case, each edge is represented as a pair, $\{u, v\}$, or two ordered pairs, $(u, v)$ and $(v, u)$.
- An ordered sequence of edges, $e_1, \ldots, e_n$, where $e_i = (v_i, v_{i+1})$ defines a **path** between $v_1$ and $v_{n+1}$.
- Two vertices are **related** if there exists at least one path between them.
  - If $v$ follows $u$ in a path, then $u$ is an **ancestor** of $v$ and $v$ is a **successor** of $u$.
  - If $v$ immediately follows $u$ in a path, then $u$ is a **parent** of $v$ and $v$ is a child of $u$.
- Each edge, $e$, may be associated with a weight $w(e)$.

- Below is a graph with:
  - $\mathcal{V} = \{A, B, C, D, E, F, G\}$
  - $\mathcal{E} = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{C, D\}, \{E, G\}\}$.



- Below is a graph with:
  - $\mathcal{V} = \{A, B, C, D, E\}$
  - $\mathcal{E} = \{(A, B), (A, C), (B, A), (B, C), (C, E), (D, C), (E, E)\}$.



- The order in which the successors are explored (i.e., the removal order) changes the search algorithm's properties.

- In the next chapter, we will consider three orderings:
  1. first-in-first-out (breadth-first search / BFS)
  2. first-in-last-out (depth-first search / DFS)
  3. smallest cumulative cost first (uniform cost search / UCS)

- There are four properties of particular interest:
  - **Time Complexity**: the order of the number of states that must be generated before the algorithm terminates.
  - **Space Complexity**: the order of the maximum number of states that exist on the frontier at any given iteration.
  - **Completeness**: the algorithm always find a goal if one exists.
  - **Optimality**: the algorithm always finds the cheapest solution if multiple exists.
- The ideal search algorithm is complete, optimal, and minimizes the time and space complexity, but we will see that this is difficult to achieve.

- We can reduce the time/space complexity if we make the following assumption:
  - actions have positive costs, i.e., $c(a) > 0$.

- In this case, we can prune paths as follows:
  - When a path $p = (s_0, \ldots, s_n)$ is being expanded to $p' = (s_0, \ldots, s_n, s_{n+1})$ if $s'_n = s_i$ for some $i \neq n+1$, do not add $p'$ to the frontier.
  - The rationale is that if actions have positive costs, then $p_{0:i}$ is a cheaper way to get to $s_i$ than $p'$, i.e., $c(p') \geq c(p_{0:i})$.

- This is called **path checking**.

# Reducing Time/Space Complexities: Multiple-Path Checking

- We can reduce the time/space complexity even further if we make the following assumption:
  - the first time the search algorithm finds a path to a given state, $s$, the path is the cheapest one to $s$.
- In this case, we can prune paths as follows:
  1. Define a list of already visited states, $\mathcal{S}_{\text{visited}}$.
  2. Initially, $\mathcal{S}_{\text{visited}} = \emptyset$.
  3. Whenever we ever discover a path, $p = (s_0, \ldots, s_n)$:
     - If $s_n \in \mathcal{S}_{\text{visited}}$, do not add $p$ to the frontier.
     - If $s_n \notin \mathcal{S}_{\text{visited}}$, then add $s_n$ to $\mathcal{S}_{\text{visited}}$.
- This is called **multiple-path checking**.