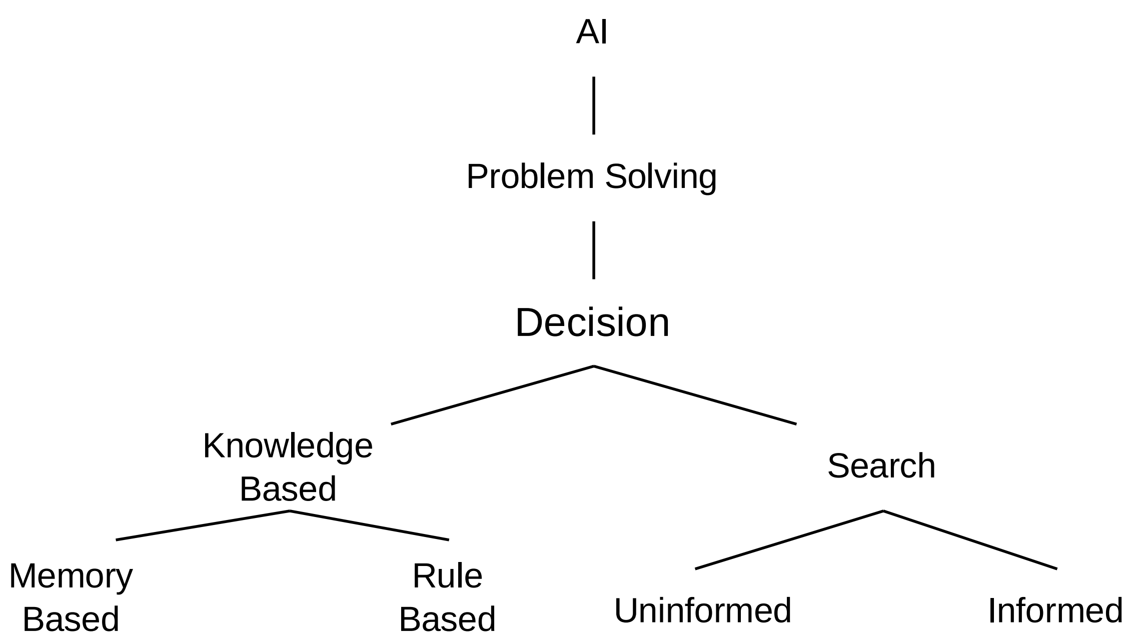
Introduction

1 Overview

Overview

We present a problem to the agent by specifying the initial situation and defining a goal. The agent then makes a series of decisions to achieve the given goal. The approaches to problem-solving can be categorized as follows:

- **Search**: Does not rely on prior experience.
  - **Uninformed Search**: Conducted without any prior information (blind search).
  - **Informed Search**: Utilizes available information to guide the search.
- **Knowledge-Based Approaches**: Relies on prior experience.
  - **Memory-Based**: Uses a database of problems and their corresponding solutions.
  - **Rule-Based**: Converts experience into rules with human intervention.



Chapter 1: State Space Search

1 State Space Search

Definition

State space search does not rely on prior experience. It involves solving a problem by exploring possible states and actions. The process includes:

1. **Modeling the Problem:** Represent the problem using simple or complex data structures.
2. **Defining the Search Space:**
   - **Set of States:** Includes the initial state (S), the goal state (G), and all intermediate states.
   - **Set of Legal Moves:** Represents the actions that allow transitions from one state to another.
3. **Defining Functions:**
   - **Movegen(S: state):** Returns a set of states representing the results of all legal moves from the given state.
   - **GoalTest(S: state):** Takes a state as input and returns a boolean value, verifying whether the given state is a goal state.

Types of Problems

- **Configuration Problems:** The goal state is not explicitly defined but is identified by its properties. The concept of a path solution is irrelevant; only the goal state is retrieved.
- **Planning Problems:** The goal state is explicitly defined, and the solution involves finding the path to reach the goal state.

Note

Search-based agents are not highly efficient due to the issue of combinatorial explosion. As the search tree grows deeper, the number of nodes increases exponentially, resulting in an unmanageable number of possibilities to explore.

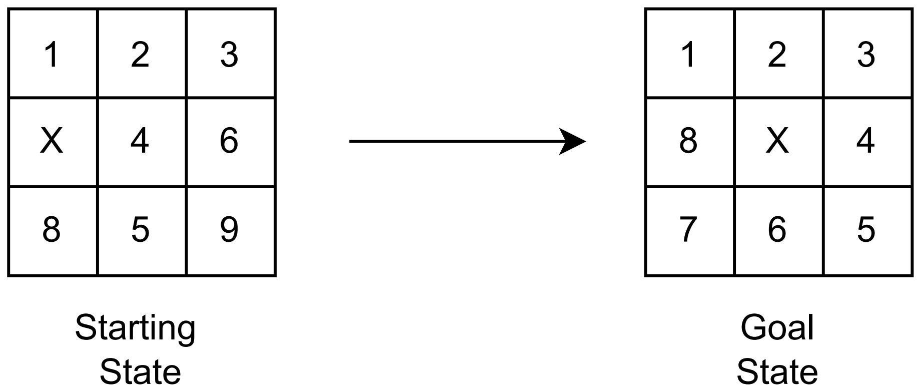
Search Categories

- **Uninformed Search:** Conducted without any guidance or additional data (blind search).
- **Informed Search:** Guided by additional information or heuristics.

Example

8-Puzzle Problem

The 8-puzzle problem consists of a 3×3 matrix with integers from 1 to 8 placed randomly, along with an empty cell represented by X. The goal is to rearrange the matrix to achieve the following configuration:



Legal Moves

- Slide Up
- Slide Down
- Slide Right
- Slide Left

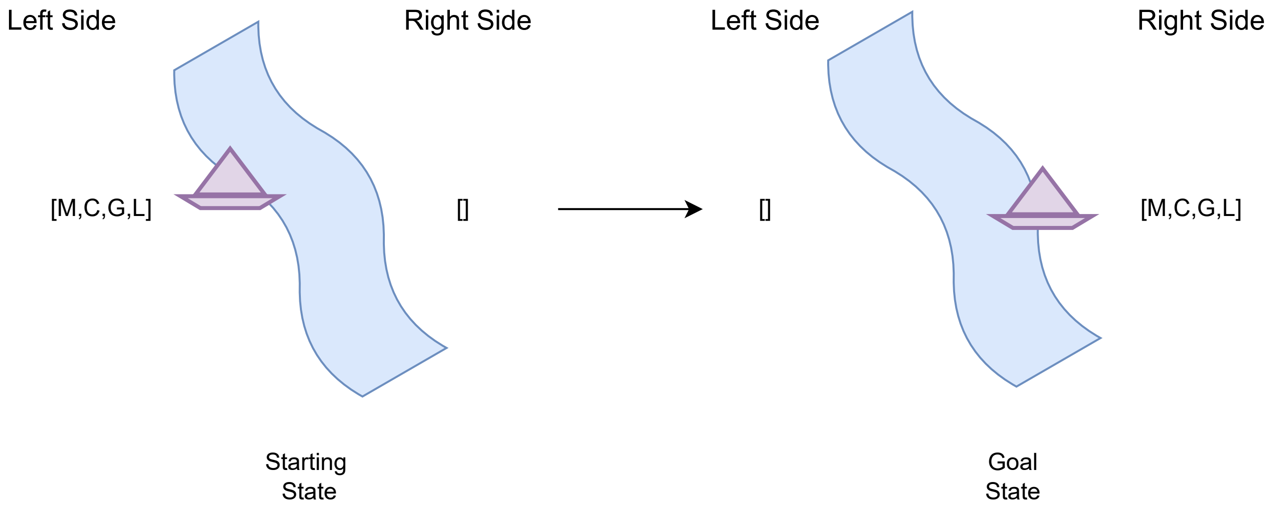
Representation

The matrix can be represented as a 2D array of integers, with the empty cell (X) optionally represented as `nil`.

River Crossing Problem

This problem involves a man, a lion, a goat, a cabbage, a boat, and a river. Initially, all entities are on the left side of the river. The goal is to transport all of them to the right side without any entity being harmed:

- If the lion and goat are left alone, the lion will kill the goat.
- If the goat and cabbage are left alone, the goat will eat the cabbage.



Legal Moves

- Man takes nothing to the right side.
- Man takes the cabbage.
- Man takes the lion.
- Man takes the goat.

Representation

A list of structures, where each structure has the following fields:

- `char name`: The name of the entity (`'M'` Man , `'C'` Cabbage, `'L'` Lion ,`'G'` Goat).
- `char position`: The position of the entity (`'L'` Left side, `'R'` Right side).

If any entity is harmed (eaten or killed), it is removed from the list, signifying its destruction.

Name: 'M' Position: 'L'	Name: 'C' Position: 'L'	Name: 'G' Position: 'L'	Name: 'L' Position: 'L'

Starting State

Name: 'M' Position: 'R'	Name: 'C' Position: 'R'	Name: 'G' Position: 'R'	Name: 'L' Position: 'R'

Goal State

Chapter 1.1: Uninformed

1 Searching Algorithms

1.1 Simple Search 1

Algorithm 1 SS1

```
Open ← {S};
while Open ≠ nil do
    N ← Remove node from Open;
    if (GoalTest(N)) then
        return N;
    else
        Open ← Open ⋃ MoveGen(N);
    end if
end while
return nil;
```

Issues With SS1

It can infinitly loop because we aren't keeping track of node we already seen to fix that we will use another list to mark seen nodes

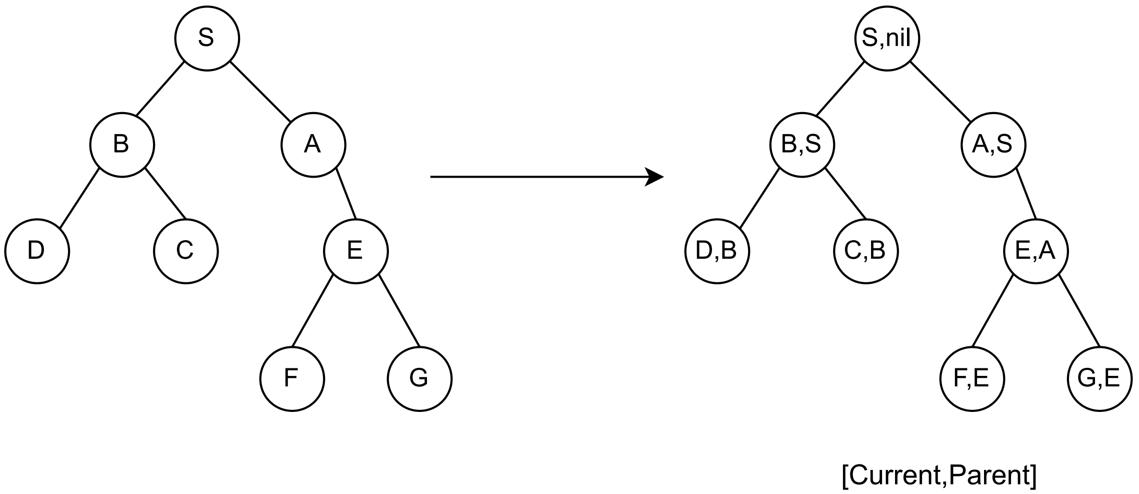
1.2 Simple Search 2

Algorithm 2 SS2

```
Open ← {S};
Closed ← {};                                          ▷ Seen Node List
while Open ≠ nil do
    N ← Remove node from Open;
    if (GoalTest(N)) then
        return N;
    else
        Closed ← Closed ⋃ {N};
        Open ← Open ⋃ ( MoveGen(N) \ (Open ⋃ Closed));   ▷ Append With No Duplicates
    end if
end while
return nil;
```

Issues with SS2

Issue is the current algo only return the goal state and not the path, this would be enough if we are in the context of configuration problem but in case of planning problem we must have the path to fix that we will use current parent node pair representation



1.3 Simple Search 3

Algorithm 3 SS3

Open ← {{S,nil}};
Closed ← {};
while Open ≠ nil **do**
  N ← Remove node from Open;
  if (GoalTest(N.current)) **then**
    return reconstructPath(N, Closed);
  else
    Closed ← Closed $\bigcup$ {N};
    Open ← Open $\bigcup$ (MoveGen(N) \ (Open $\bigcup$ Closed));
  end if
end while
return nil;

Algorithm 4 reconstructPath

```
function RECONSTRUCTPATH((I/N:(current,parent),Closed : List(current,parent)): List)
    path ← {N.Current};
    N ← find node in Closed where N.parent = Closed.current
    while N.parent ≠ nil do
        path ← path ⋃ {N.Current};
        N ← find node in Closed where N.parent = Closed.current
    end while
    return reverse(Path);
end function
```

How To Choose N

We have two choices for selecting N: either the head or the tail. The behavior of each choice differs as follows:

- **Head:** If we choose the head, it is treated as a queue. In this case, we *reappend* the state to the end of the structure, effectively following a breadth-first search approach.
- **Tail:** If we choose the tail, it is treated as a stack. We simply *append* the state to the end of the structure, following a depth-first search approach.

1.4 BFS

Algorithm 5 BFS

```
Open ← {{S,nil}};
Closed ← {};
while Open ≠ nil do
    N ← Head(Open);
    if (GoalTest(N.current)) then
        return reconstructPath(N, Closed);
    else
        Closed ← Closed ⋃ {N};
        for each new in MoveGen(N.current) do
            if new ∉ Open ⋃ Closed then
                Open ← Preappend (new,N);
            end if
        end for
    end if
end while
return nil;
```

1.5 DFS

Algorithm 6 DFS

```
Open ← {{S,nil}};
Closed ← {};
while  Open ≠ nil do
    N ← Tail(Open);
    if (GoalTest(N.current)) then
        return reconstructPath(N, Closed);
    else
        Closed ← Closed ⋃ {N};
        for  each new in MoveGen(N.current)  do
            if new ∉ Open ⋃ Closed then
                Open ← Append (new,N);
            end if
        end for
    end if
end while
return nil;
```

1.5.1 Bounded DFS

Algorithm 7 Bounded DFS

```
Open ← {{S,nil, 0}};                              ▷ Add depth information
Closed ← {};
Bound ← max depth;
while  Open ≠ nil do
    N ← Tail(Open);
    Depth ← N.depth;                              ▷ Get current depth
    if (GoalTest(N.current)) then
        return reconstructPath(N, Closed);
    else if Depth ≤ Bound then                    ▷ Check if depth is within the bound
        Closed ← Closed ⋃ {N};
        for  each new in MoveGen(N.current)  do
            if new ∉ Open ⋃ Closed then
                Open ← Append (new, N, Depth+1);
            end if
        end for
    end if
end while
return nil;
```

1.5.2 DFID

Algorithm 8 DFID

```
db ← 0;
while BDFS(db) = nil do
    ++db;
end while
```

Algorithm	Space Complexity	Time Complexity	Completeness	Optimality
BFS	$O(b^d)$	$O(b^d)$	Yes	Yes
DFS	$O(bm)$	$O(b^m)$	No	No
Bounded DFS	$O(b \cdot d)$	$O(b^d)$	No	No
DFID	$O(b \cdot d)$	$O(b^d)$	Yes	Yes

Note

- b : branch factor, max number of children node has
- d : deepth ,max edges between nodes
- Completness : Find The Solution if it exist
- Optimal : if it result in shortest path

Chapter 1.2: Informed

1 Heuristic Function $h(n)$

Definition

The Heuristic Function takes a state as input and returns a heuristic value, which indicates how close the state is to the solution.

Better Heuristic $h(n)$

A single problem can have many heuristic functions. These heuristics can be compared on two main points:

- **Cost:** We aim for a low cost since each node we traverse will involve applying the heuristic function.
- **Effectiveness:** How efficient the heuristic is at guiding the search. If the ratio is equal to 1, it is perfect.

$$\text{Effectiveness} = \frac{\text{Nb}_{\text{Seen Nodes}}}{|\text{Path}|}$$

2 Types of Heuristic Functions

Types

Heuristic functions can be divided into the following types:

- **Static (Dependent on the Domain):** A static rule derived from the goal state is applied to a given state to compute its heuristic value.
- **Dynamic (Independent of the Domain):** Uses a relaxed problem, typically simplifying the problem constraints to guide the search.

3 Searching Algorithms

3.1 Best-First Search

Algorithm 9 Best-First Search

```
Open ← {{S, nil, h(S)}};
Closed ← {};
while  Open ≠ nil)  do
    N ← Head(Open);
    if GoalTest(N.current) then
        return reconstructPath(N, Closed);
    else
        Closed ← Closed ⋃ {N};
        for each new state in MoveGen(N.current) do
            if new ∉ Open ⋃ Closed then
                Open ← append({new, N, h(new)});
            end if
        end for
        sort_h(Open);
    end if
end while
return nil;
```

3.2 Hill Climbing

Algorithm 10 Best-First Search

```
next ← {{S, nil, h}};
value ← Next.h;
b ← true;
path ← {next};
while  b  do
    for each new in MoveGen(next)  do
        if new better than next then
            next ← new;
            path ← append(new);
        end if
    end for
    if value = next.h then
        b ← false;
    else
        value ← next.h;
    end if
end while
return path;
```

Algorithm	**Space Complexity**	**Time Complexity**	**Completeness**	**Optimality**	**Scale**
Best-First Search	Dependent Of $h(n)$	Dependent Of $h(n)$	Yes	No	Global Search
Hill Climbing	$O(1)$	linear	No	No	Local Search