Problem Formulation

Problem solving

- We want:
  - To automatically solve a problem
- We need:
  - A representation of the problem
  - Algorithms that use some strategy to solve the problem defined in that representation

Problem representation

- General:
  - **State space**: a problem is divided into a set of resolution steps from the initial state to the goal state
  - **Reduction to sub-problems**: a problem is arranged into a hierarchy of sub-problems
- Specific:
  - Game resolution
  - Constraints satisfaction

States

- A problem is defined by its elements and their relations.
- In each instant of the resolution of a problem, those elements have specific descriptors (How to select them?) and relations.
- A **state** is a representation of those elements in a given moment.
- Two special states are defined:
  - **Initial state** (starting point)
  - **Final state** (goal state)

State modification: successor function

- A successor function is needed to move between different states.
- A **successor function** is a description of possible actions, a set of operators. It is a transformation function on a state representation, which convert it into another state.
- The successor function defines a relation of accessibility among states.
- Representation of the successor function:
  - Conditions of applicability
  - Transformation function

State space

- The **state space** is the set of all states reachable from the initial state.
- It forms a graph (or map) in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The solution of the problem is part of the map formed by the state space.

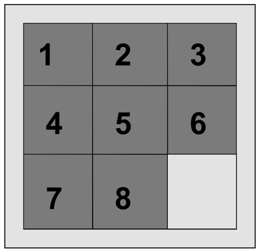
Problem solution

- A **solution** in the state space is a path from the initial state to a goal state or, sometimes, just a goal state.
- **Path/solution cost**: function that assigns a numeric cost to each path, the cost of applying the operators to the states
- Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.
- Solutions: any, an optimal one, all. Cost is important depending on the problem and the type of solution sought.

Problem description

- Components:
  - State space (explicitly or implicitly defined)
  - Initial state
  - Goal state (or the conditions it has to fulfill)
  - Available actions (operators to change state)
  - Restrictions (e.g., cost)
  - Elements of the domain which are relevant to the problem (e.g., incomplete knowledge of the starting point)
  - Type of solution:
    - Sequence of operators or goal state
    - Any, an optimal one (cost definition needed), all

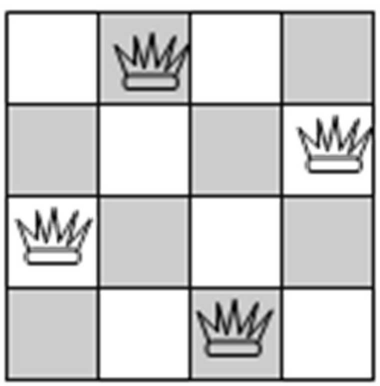
Example: 8-puzzle

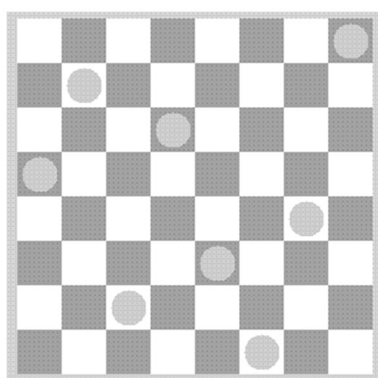


Example: 8-puzzle

- **State space**: configuration of the eight tiles on the board
- **Initial state**: any configuration
- **Goal state**: tiles in a specific order
- **Operators or actions**: “blank moves”
  - Condition: the move is within the board
  - Transformation: blank moves *Left*, *Right*, *Up*, or *Down*
- **Solution**: optimal sequence of operators

Example: *n* queens (n = 4, n = 8)





Example: *n* queens (n = 4, n = 8)

- **State space**: configurations from 0 to n queens on the board with only one queen per row and column
- **Initial state**: configuration without queens on the board
- **Goal state**: configuration with n queens such that no queen attacks any other
- **Operators or actions**: place a queen on the board
  - Condition: the new queen is not attacked by any other already placed
  - Transformation: place a new queen in a particular square of the board
- **Solution**: one solution (cost is not considered)

Structure of the state space

- Data structures:
  - Trees: only one path to a given node
  - Graphs: several paths to a given node
- Operators: directed arcs between nodes
- The search process explores the state space.
- In the worst case all possible paths between the initial state and the goal state are explored.

Search as goal satisfaction

- Satisfying a goal
  - Agent knows what the goal is
  - Agent cannot evaluate intermediate solutions (uninformed)
  - The environment is:
    - Static
    - Observable
    - Deterministic

Example: holiday in Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest at 13:00
- Let's configure this to be an AI problem

Romania

- What's the problem?
  - Accomplish a *goal*
    - Reach Bucharest by 13:00
- So this is a goal-based problem

Romania

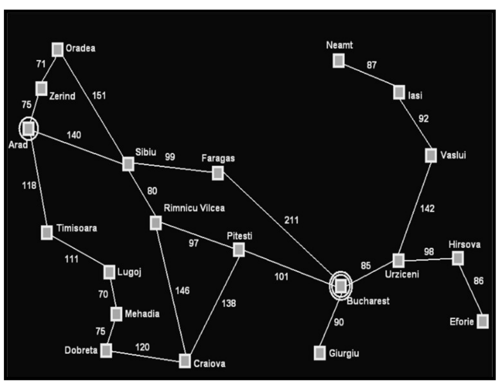
- What's an example of a non-goal-based problem?
  - Live long and prosper
  - Maximize the happiness of your trip to Romania
  - Don't get hurt too much

Romania

- What qualifies as a solution?
  - You can/cannot reach Bucharest by 13:00
  - The actions one takes to travel from Arad to Bucharest along the shortest (in time) path

Romania

- What additional information does one need?
  - A map



More concrete problem definition

A state space	***Which cities could you be in?***
An initial state	***Which city do you start from?***
A goal state	***Which city do you aim to reach?***
A function defining state transitions	***When in city foo, the following cities can be reached***
A function defining the "cost" of a state sequence	***How long does it take to travel through a city sequence?***

More concrete problem definition

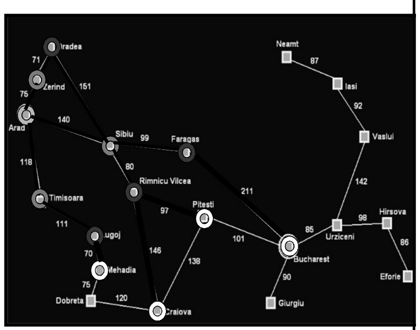
A state space	***Choose a representation***
An initial state	***Choose an element from the representation***
A goal state	***Create*** `goal_function(state)` ***such that TRUE is returned upon reaching goal***
A function defining state transitions	`successor_function(state`$_i$`) =` ***{<action$_a$, state$_a$>, <action$_b$, state$_b$>, …}***
A function defining the "cost" of a state sequence	`cost (sequence)` ***= number***

Important notes about this example

- Static environment (available states, successor function, and cost functions don't change)
- Observable (the agent knows where it is)
- Discrete (the actions are discrete)
- Deterministic (successor function is always the same)

Tree search algorithms

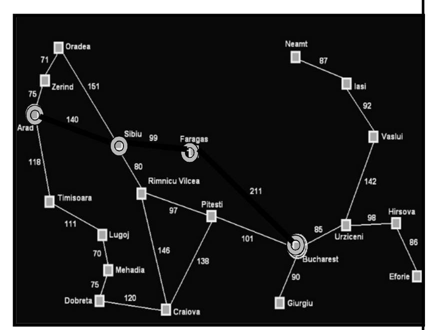
- Basic idea:
  - Simulated exploration of state space by generating successors of already explored states (AKA expanding states)



Sweep out from start (breadth)

Tree search algorithms

- Basic idea:
  - Simulated exploration of state space by generating successors of already explored states (AKA expanding states)



(depth)

Implementation: general search algorithm

```
Algorithm General Search
 Open_states.insert (Initial_state)
 Current= Open_states.first()
  while not is_final?(Current) and not Open_states.empty?() do
   Open_states.delete_first()
   Closed_states.insert(Current)
   Successors= generate_successors(Current)
   Successors= process_repeated(Successors, Closed_states,
        Open_states)
   Open_states.insert(Successors)
   Current= Open_states.first()
  eWhile
eAlgorithm
```