Problem Solving

- To solve problems is an intelligent capability
- We are able to solve very different problems
  - To find a path in a labyrinth
  - To solve a crossword
  - To play a game
  - To diagnose an illness
  - To decide if invest in the stock market
  - ...
- The goal is to have programs also able to solve these problems

Problem solving

- We want to represent any problem so we can solve it automatically
- To do so we need:
  - A common representation for all the problems
  - Algorithms that use some strategy to solve the problems using the common representation

Representing a problem

- If we analyze the nature of a problem we can identify:
  - A starting point
  - A goal to achieve
  - Actions that we can use to solve the problem
  - Constraints for the goal
  - Elements relevant to the problem defined by the characteristics of the domain

Representing a problem

- General representations
  - **State space:** a problem is a sequence of steps from the starting point to the goal
  - **Problem decomposition:** a problem can be decomposed recursively in a hierarchy of subproblems
- Specific representations
  - **Games**
  - **Constraint satisfaction**

Representation of problems: States

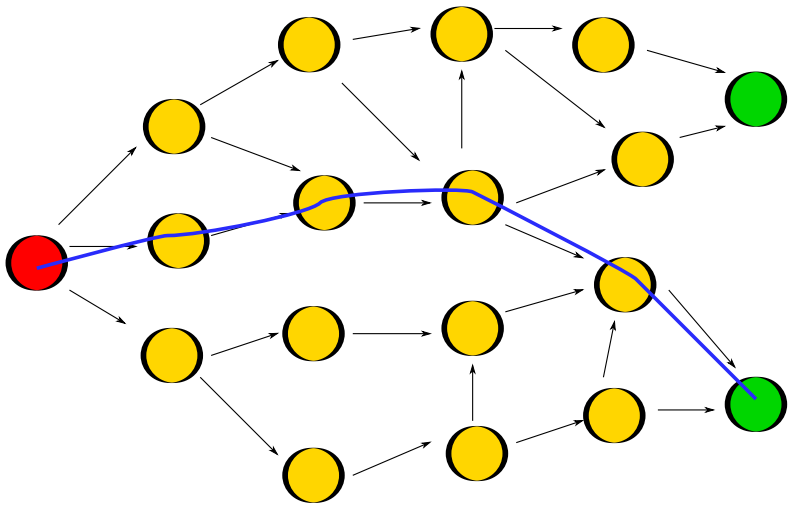
- A problem can be defined by the elements that it is composed of and the relationships among them
- At each step of a resolution these elements have specific characteristics and relationships
- We define a **state** as a set of elements that describe a problem in a specific moment of its resolution
- There a two special states the **Initial State** (starting point) and the **Goal State** (goal of the problem)
- What to include in the representation of a state?

Changing the state: actions

- In order to move from a state to another we need a successor function
- **Action:** function that applied to a state gives another state that follows in sequence in the resolution
- The actions define an accessibility relation among states
- Representation of an action:
  - Applicability conditions
  - Transformation function
- What actions? How many? What granularity?

State space

- The states and the accessibility relation define the **state space** of a problem
- The space state represents all the possible paths among all the states of the problem
- It could be assimilated to a road map of a problem
- Our solution is in this road map



Solving problems in state space

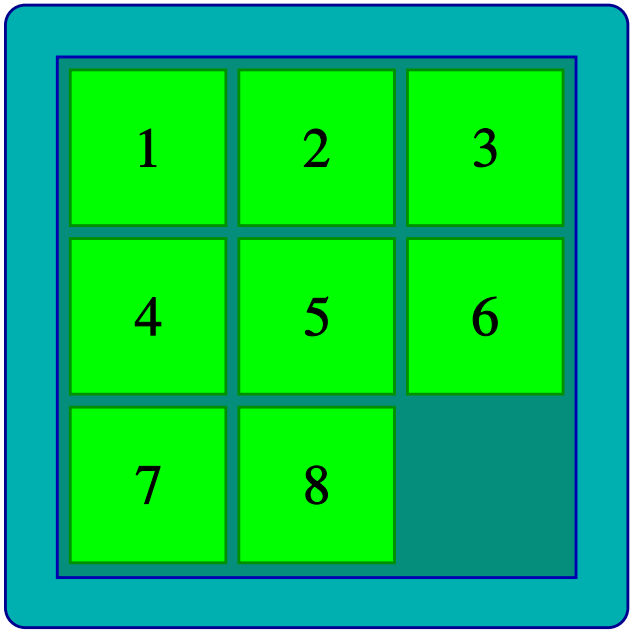
- **Solution:** Sequence of states that go from the initial state to the goal state (sequence of actions) or also the goal state
- **Types of solutions:** Any, the optimal, all
- **Cost of a solution:** Resources used by the actions from a solution. It could be important or not depending the type of the solution we are looking for.

Describing a problem for state space solving

- Define the set of states of the problem (explicitly or implicitly)
- Define the initial state
- Define the goal state or the conditions/constraints it has to hold
- Define the actions (applicability conditions and transformation function)
- Specify the type of the solution:
  - Sequence of actions or the goal state
  - Any solution, the optimal solution (define the cost of the actions), ...

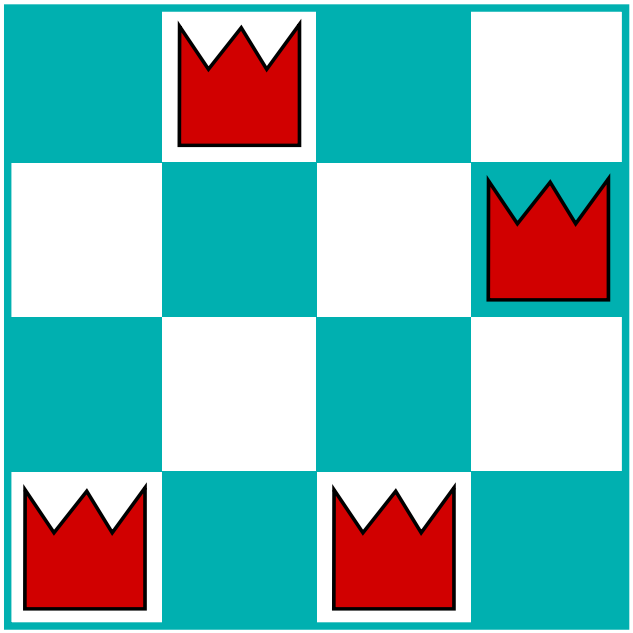
Example: 8 puzzle

- States: Configurations of 8 tiles on the board
- Initial state: Any configuration
- Goal state: Tiles in a specific configuration
- Actions: Move the blank tile
  - Conditions: The movement is inside the board
  - Transformation: Swap the blank tile with the tale in the position of the movement
- Solution: set of actions + optimal number of actions



Example: N queens

- States: Configurations from 0 to n chess queens in a $n \times n$ board with only one queen per column and row
- Initial state: Empty configuration
- Goal state: Configurations that hold that there are no queen that attacks another
- Actions: To add a queen to the board in a position
  - Conditions: No queen attacks the new queen
  - Transformation: There is a new queen on the board in a specific position
- Solution: Any solution, the actions does not matter



Search on state space

- To solve a problem in this representation an exploration of the state space is needed
- We have to start from the initial state, evaluating each possible action that leads to a new state until finding the goal state
- Worst case scenario: we will search all possible paths from the initial state to the goal state

Structure of the state space

- First we will define how to represent the state space in order to be able to implement search algorithms

- Data Structures: Trees and graphs
- States = Nodes
- Actions = Directed edges
- Trees: Only a path leads to a node
- Graphs: Many paths lead to a node

Basic algorithm

- The state space can be infinite
- A different approach to search and traverse trees and graphs is needed (we can not have the structure in memory)
- The data structure is constructed while the search is performed

Basic algorithm

Function: Basic Search Algorithm()

Data: The initial state
Result: A solution
Select the initial state as the current
while *current state* $\neq$ *goal state* **do**
    Generate and store the successors of the current state (expansion)
    Choose the next state among the pending states (selection)
end

- The selection of the next state will determine the strategy of search (order of expansion)
- It is necessary to define an order among the successors of a state (order of generation)

Basic algorithm

- **Open nodes:** States generated but not yet expanded
- **Closed nodes:** States already expanded
- We will need a data structure to store the open nodes
- The policy of insertion of the data structure will determine the search strategy
- If we are searching in a graph it could be necessary to detect repeated states (this means we also need a data structure to store the closed nodes). It is worth to have it if the number of different states is small compared to the number of paths

Characteristics of the algorithms

- Characteristics:
  - **Completeness:** Is it guaranteed to find a solution if there is one?
  - **Temporal Complexity:** How long does it take to find a solution?
  - **Spatial Complexity:** How much memory is needed?
  - **Optimality:** does it find the optimal solution?

General search algorithm

Algorithm: General search

```
St_open.add(initial state)
Current ← St_open.first()
while not is_goal?(current) and not St_open.empty?() do
    St_open.delete_first()
    St_closed.add(Current)
    Successors ← generate_successors(current)
    Successors ← treat_duplicated(Successors, St_closed, St_open)
    St_open.add(Successors)
    Current ← St_open.first()
end
```

- Changing the structure, the behaviour of the algorithm changes (order in which the states are visited)
- The function `generate_successors` will generate the siblings using the order determined by the problem
- The treatment of the repeated nodes will depend on the expansion strategy

Search strategies

- Uninformed search (blind search)
  - The cost of the solution is not used during the search
  - They are systematic, the order of visit and generation of the nodes are established by the structure of the state space
  - Breadth-first, Depth-first, Iterative deepening
- Heuristic search
  - An estimate of the cost of the solution is used to guide the search
  - Optimality is not always guaranteed, not even a solution
  - Hill-climbing, Branch and Bound, A*, IDA*

Breadth-first search

- Nodes are generated and visited level wise
- The structure to store the open nodes is a queue (FIFO)
- A node is visited when all the nodes from the previous level and its previous siblings in generation order have been visited
- Characteristics:
  - Completeness: The algorithm always find a solution
  - Temporal Complexity: Bounded by an exponential function of the branching factor over the depth of the solution $O(b^d)$
  - Spatial Complexity: Bounded by an exponential function of the branching factor over the depth of the solution $O(b^d)$
  - Optimality: The solution is optimal on the number of levels from the root of the search tree

Depth-first search

- The nodes are visited and generated always looking for the deepest node
- The structure to store the open nodes is a stack (LIFO)
- In order to assure that the algorithm stops, a maximum depth of exploration is used (to avoid infinite paths)
- Characteristics
  - Completeness: The algorithm finds a solution if a maximum depth of exploration is used and there is a solution above this limit
  - Temporal Complexity: Bounded by an exponential function of the branching factor over the maximum depth $O(b^m)$
  - Spatial Complexity: If duplicated nodes are not controlled, it is bounded by an linear function of the branching factor times the maximum depth $O(b \times m)$. If duplicated nodes are treated the space is the same than breadth-first search. If we use a recursive implementation of the algorithm (no stack needed), it is bounded by an linear function of the maximum depth $O(m)$
  - Optimality: There is no guarantee that the solution found is optimal

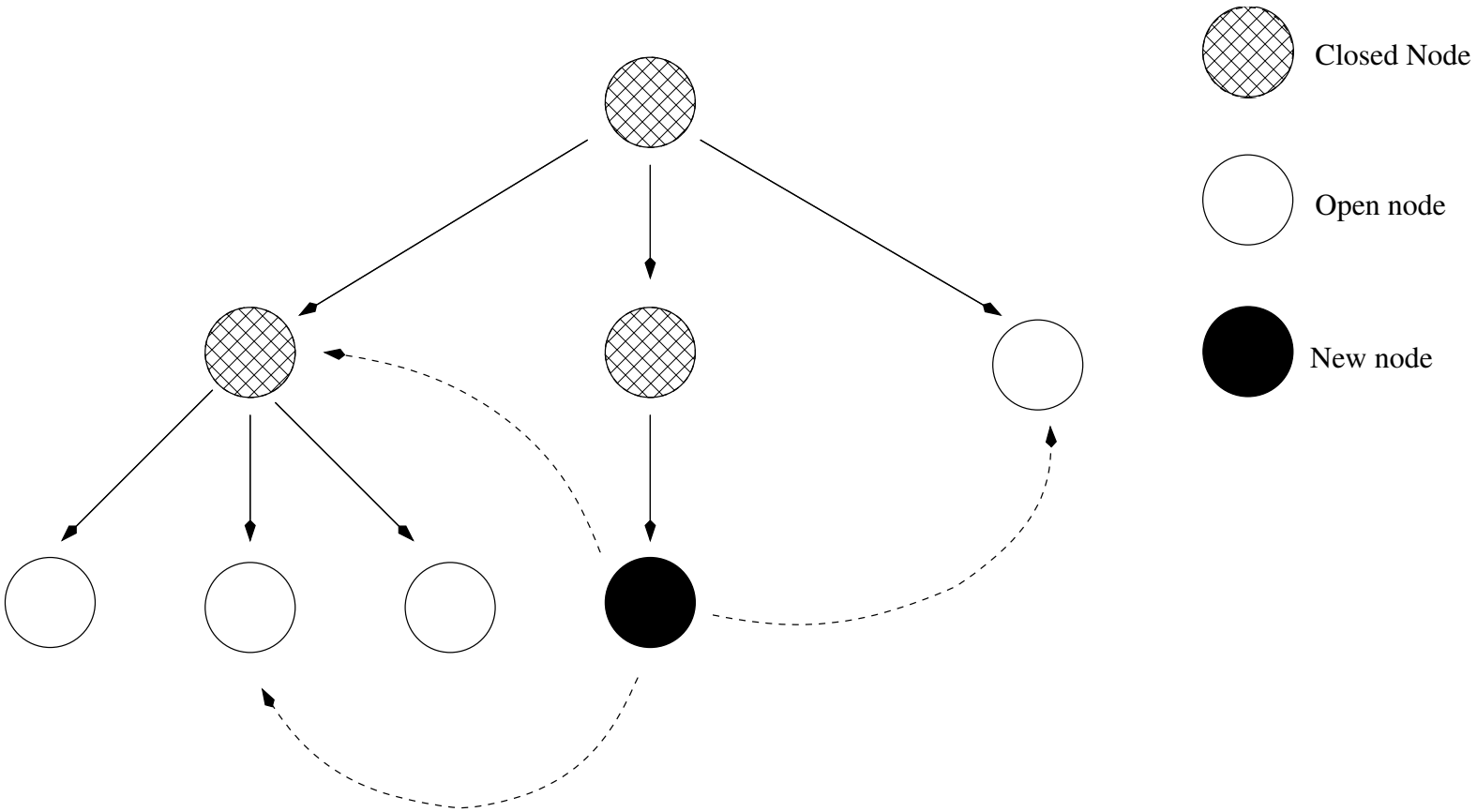
Depth-first limited search

Procedure: Depth-first limited search (limit: integer)

```
St_open.add(initial state)
Current ← St_open.first()
while not is_goal?(Current) and not St_open.empty?() do
    St_open.delete_first()
    St_closed.add(Current)
    if depth(current) ≤ limit then
        Successors ← generate_successors (Current)
        Successors ← treat_duplicated (Successors, St_closed, St_open)
        St_open.add(Successors)
    end
    Current ← St_open.first()
end
```

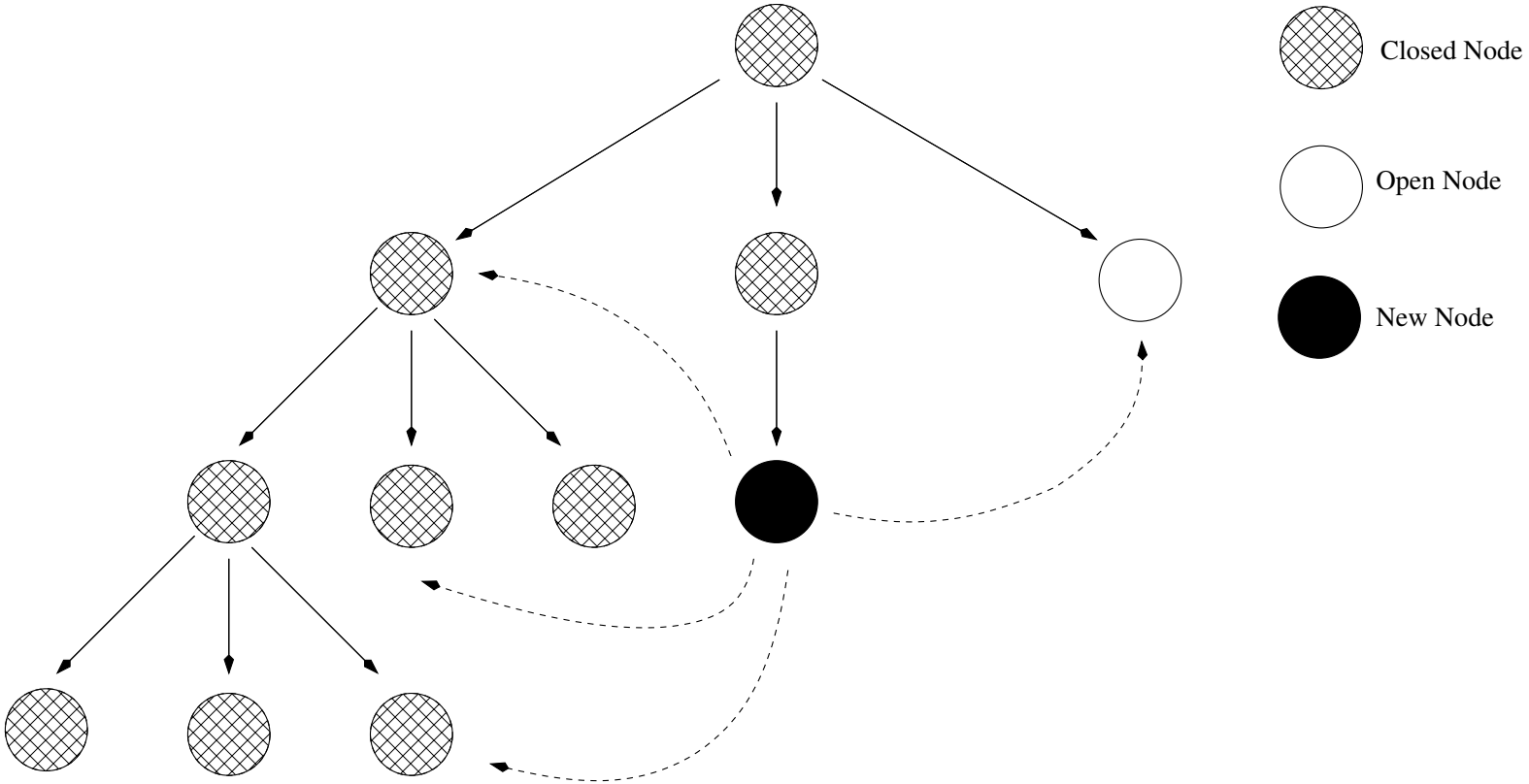
- The structure for the open nodes is a stack
- A node is not expanded when its depth is greater than the depth limit
- This guaranties that the algorithm stops
- If duplicated nodes are treated we lose the space complexity advantage

Treatment of duplicated nodes - Breadth-first



- If the duplicated is in the closed nodes structure it can be discarded. It has a depth that is greater or equal that the closed node.
- If the duplicated node is in the open nodes structure it can be discarded. It has a depth that is greater or equal that the open node.

Treatment of duplicated nodes - Depth-first

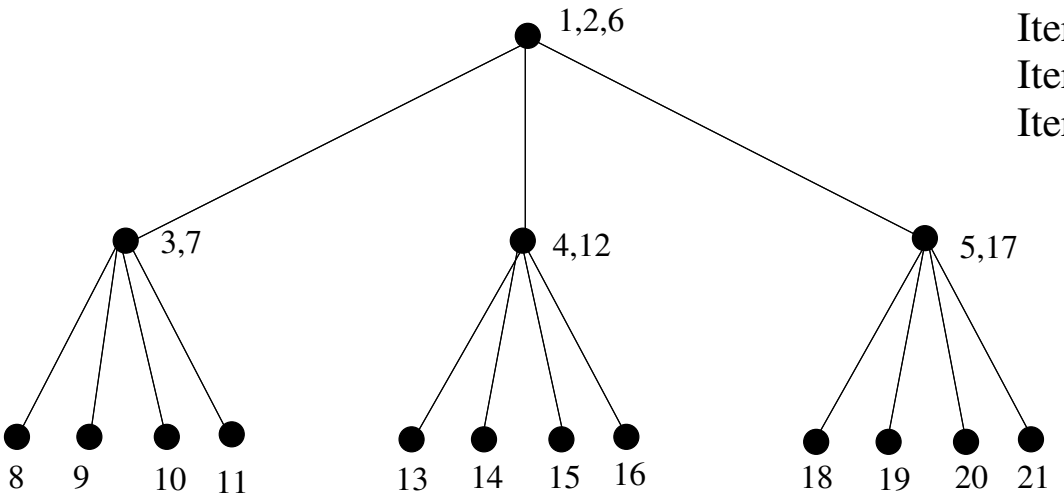


- If the node is in the closed nodes structure we keep it if has a depth that is smaller than the closed node
- If the node is in the open nodes structure it can be discarded. It has a depth that is greater or equal that the open node.

Iterative Deepening search (ID)

- Combines the space complexity of Depth-first search and the optimality of solutions of Breadth-first search
- The algorithm performs successive depth-first searches with limited depth that is increased each iteration
- This strategy gives a behaviour similar to breadth-first search but without its spatial complexity because each exploration is depth-first, although all the nodes are generated each iteration
- This strategy allows to avoid the cases when depth-first algorithm does not end (infinite paths)
- The first iteration the maximum depth is 1 and this value will be incremented until a solution is found
- In order to guarantee that the algorithm ends if there is no solution a depth limit can be imposed

Iterative Deepening (ID)



Iteration 1: 1
Iteration 2: 2,3,4,5
Iteration 3: 6,7,8,9,...21

Iterative Deepening Search

Procedure: Iterative Deepening Search (limit: integer)

```
depth ← 1
Current ← initial state
while not is_goal?(Current) and depth<limit do
    St_open.reset()
    St_open.add(initial state)
    Current ← St_open.first()
    while not is_goal?(Current) and not St_open.empty?() do
        St_open.delete_first()
        St_closed.add(Current)
        if depth(Current) ≤ depth then
            Successors ← generate_successors (Current)
            Successors ← treat_duplicated (Successors, St_closed, St_open)
            St_open.add(Successors)
        end
        Current ← St_open.first()
    end
    depth ← depth+1
end
```

Iterative Deepening

- Completeness: The algorithm always finds a solution (if it exists)
- Temporal Complexity: Same as Breadth-first search. To regenerate the nodes each iteration only adds a constant factor to the cost $O(b^d)$
- Spatial Complexity: Same as Depth-first search
- Optimality: Optimal as Breadth-first search