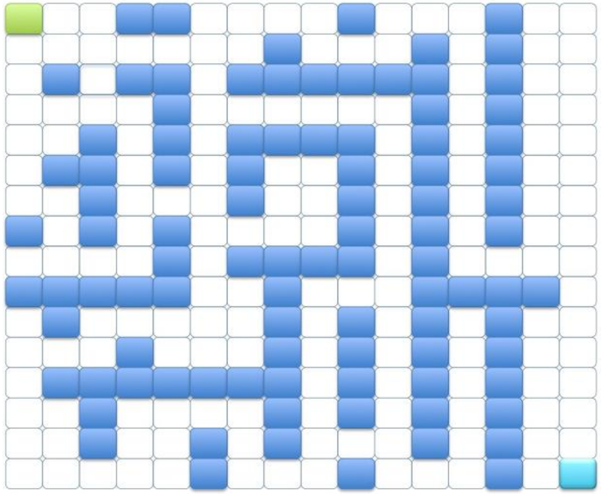
Grado en ingeniería informática
Artificial Intelligence 2021/2022

2. Problem solving using search

Lara Quijano Sánchez

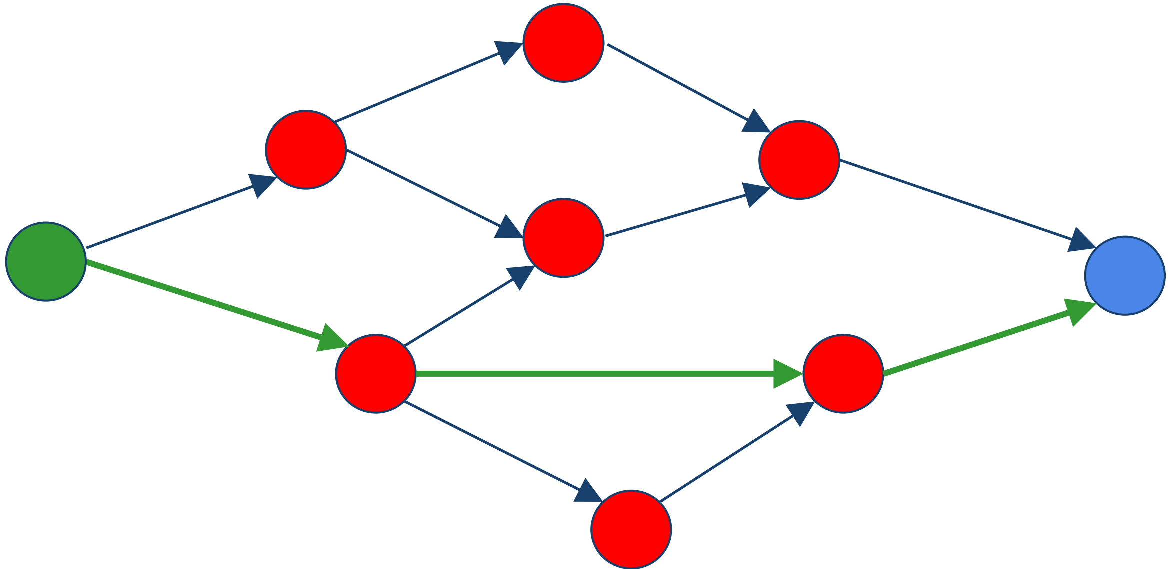


Universidad Autónoma
de Madrid

Grado en ingeniería informática
Artificial Intelligence 2021/2022

2.1. Uninformed search

Lara Quijano Sánchez



Universidad Autónoma de Madrid

Readings

- CHAPTER 3 of Russell & Norvig
- CHAPTERS 7 and 8 of Nilsson

Some figures from http://aima.cs.berkeley.edu/

Lesson 2: Problem solving using search

1. Representation of problems in a state space
2. Problem solving with search strategies
   - ❑Uninformed or blind methods
   - ❑Informed or heuristic methods
   - ❑Search between adversaries

State space search

Solve a problem by searching:

Formulate + search + execution

- ❑ **Problem formulation:**
  - ❑ Define **states**
  - ❑ Specify the **initial state**
  - ❑ Specify the **actions** that the agent can perform
    - ❑ Rules for allowed actions.
    - ❑ Successor function:

      Current state → List of directly accessible states.
  - ❑ Define the goal states
    - ❑ Extensive definition: List
    - ❑ Intensive definition: **Goal test**
  - ❑ Define **utility**: Cost function of the path.

Path: Sequence of states connected by actions.

State space: Set of accessible states from the initial state

Represented by a connected graph whose nodes ≡ states, arcs ≡ actions.

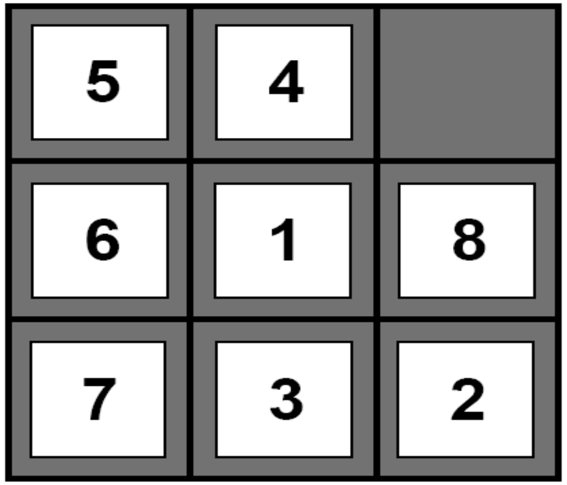
Solution: Path from the initial state to a goal state.

Optimal solution: Solution with lowest cost.

Simple problems I

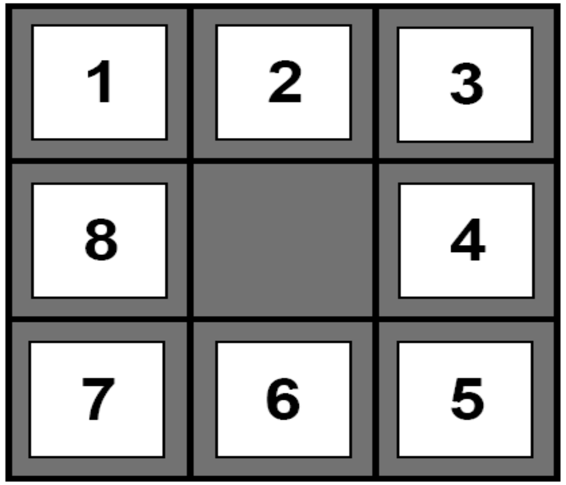
8-puzzle

- □ **States:** Arrangements of 8 tiles numbered from 1 to 8 + empty space on a 3 x 3 board.
- □ **Initial state:** A given arrangement (arbitrary).
- □ **Actions:** Move a tile adjacent to the empty space to that position.
  The empty space will be now where the moved tile was.
- □ **Goal state:** An ordered arrangement, with the empty box in the middle.
- □ **Utility:** Unit cost per movement



5	4	
6	1	8
7	3	2

Start State



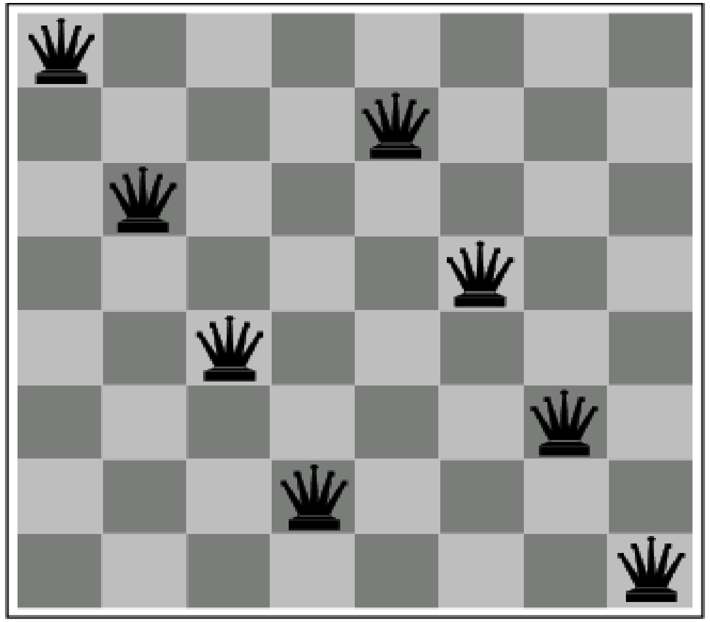
1	2	3
8		4
7	6	5

Goal State

Simple problems II

The N Queens problem (formalized using complete states)

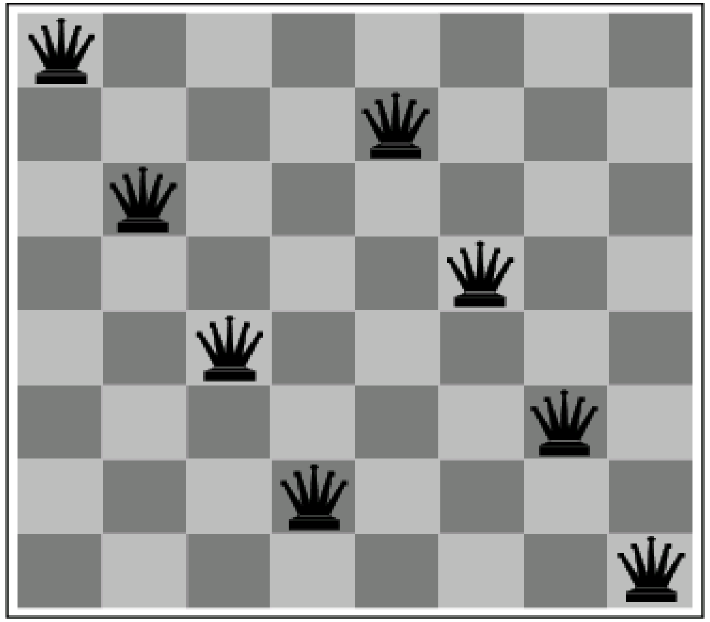
- ❑ **States:** Arrangements of N queens on a chessboard of N x N squares.
- ❑ **Initial state:** A given (arbitrary) arrangement of the N queens on the board.
- ❑ **Actions:** Move a queen to an empty square.
- ❑ **Final state:** N queens that do not attack each other.
- ❑ **Utility:** Only the final state is important.



Simple problems III

The N Queens problem (incremental formalization)

- ❑ **States:** Arrangements of n = 1,2, 3, … N queens on a N x N chessboard.
- ❑ **Initial state:** An empty board.
- ❑ **Actions:** From a state with m queens, place a new queen on an empty square of the empty column to the left which does not attack any current queen on the board.
- ❑ **Final state:** N queens that do not attack each other.
- ❑ **Utility:** All solutions have the same length (N steps) and the same cost.



Simple problems IV

Cryptoarithmetics

- ❑ **States:** letters + digits.
- ❑ **Initial state:** All letters.
- ❑ **Actions:** Replace a letter with a digit not used yet.
- ❑ **Final state:** All letters are replaced by digits in such a way that the arithmetic is correct.
- ❑ **Utility:** All solutions have the same length (10 steps) and the same cost.

```
   FORTY               29786
+    TEN           +     850
     TEN                 850
  ------              ------
    SIXTY              31486
```

Real-world problems I

- ❑ Path finder (e.g. plane trip)
    - ❑ **States:** Airport + arrival time.
    - ❑ **Initial state:** Departure airport + departure time
    - ❑ **Actions:** States resulting from taking one of the scheduled flights available from the current airport, leaving later than the current time + airport transit time.
    - ❑ **Final state:** Destination airport + desired arrival time.
    - ❑ **Utility:** Economic cost, total travel time, number of transfers, comfort, etc.

- ❑ Route planning problems:
    - ❑ **TSP** : Travelling Salesman Problem.

      Problem of optimal tour design: Find a route through N given cities in which each city is visited only once, minimizing the length of the road.
        - ❑ **States:** N cities.
        - ❑ **Initial state:** City of departure.
        - ❑ **Actions:** Travel to a city not visited yet that is directly accessible by road from the current city.
        - ❑ **Final state:** City of departure, having visited each of the other cities exactly once
        - ❑ **Utility:** Total traveled distance.

Real-world problems II

- ❑ Design of distribution of components in VLSI chips
  - Place components and connections on a chip to:
    - ❑ Minimize: area, latencies, unwanted capacitances, heat generation, costs.
    - ❑ Maximize: production level.
- ❑ Robot navigation
- ❑ Problems of sequential assembly in factories (spatial reasoning)
- ❑ Protein design (predict the tertiary structure from the primary structure).
- ❑ Search on the Internet

State Space: Basic Concepts

- ❑ Given a problem, I need to **represent** it on the computer using:
  - ❑ **Status**: current situation, set of all relevant characteristics
    - ❑ Chess: position of the pieces on the board
    - ❑ Traffic route: traffic density on each city street
- ❑ How to reach the solution of a problem?
  - ❑ Execution of valid actions (**movements** using **operators**)
    - ❑ Chess: I move a piece
    - ❑ Traffic route: I choose to go down a street
  - ❑ What has changed? another different situation **(another state**)
    - ❑ Chess: another piece situation
    - ❑ Traffic Route: I have advanced down that street, I am in another place
- ❑ **State Space**: The set of all possible situations
- ❑ **Solution**: a state or the path to reach the goal
- ❑ Reaching the solution depends on what movements the agent makes
  - ❑ What **search strategy** do I use?

How to define the State in a problem

- **Relevant characteristics.** Yes, but… which ones?

When we **modify**, through an **action**, the **state** of the system,

What **influence** does it make that is relevant to the **solution** of the problem?

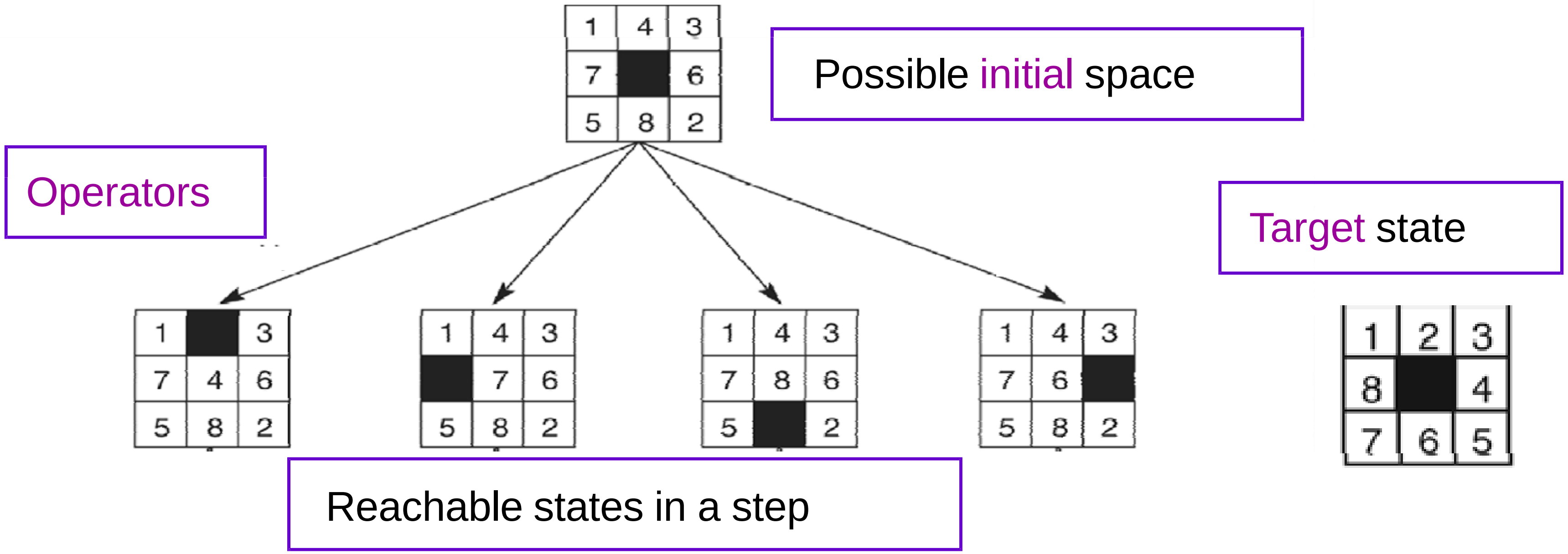
- Where does one state end and the next begins?
  - Eg. Path of a ship from Earth to Mars.
    - A state is a snapshot taken every certain time interval.
    - A rough estimation of the location is made.
- What happens if several things change at the same time?
  - Eg. tennis match: both players move at the same time!
    - Infinite states, continuous changes
    - Approximate location
    - continuous time becomes discrete (the process is monitored in discrete instants)
  - Eg. soccer game: much more complexity
- What happens if we do not know relevant parts of the problem?
  - Heuristics: Cost estimates from approximations of the problem using similar problems whose solution is known
  - Ignore the information: the search works worse. May not or will not find solution

Representation Agreement: Elements and Steps

1. **State**: define elements that characterize a state
2. **Initial state**
3. **Target states**: define the conditions a state must meet to determine if the target has been achieved.
4. **Operators** (actions or function that generates successors or movement):
   - ❑ Actions available to transform one state into the next
   - ❑ Pre and postconditions are defined for the initial (prior to the action) and final (after the action) states, respectively
   - ❑ Operator cost: represents the effort to apply said operator once
5. **Solution**: path from initial state to target state
   - ❑ Cost of the solution: Sum of the cost of the applied operators from the initial state to the target state
   - ❑ There may be solutions (paths) of different costs
   - ❑ Optimal Solution: Lower Cost Solution

8-puzzle example

- State
  - 3 * 3 board
  - 8 numbered tiles
  - 1 gap
- Definition of state space
  - Reachable states from initial state
  - Implicitly defined by initial state and operators
  - Directed graph: vertices (states), links (actions)



Possible initial space

Operators

Target state

Reachable states in a step

Represent State Space: Proper Abstraction

- ❑A problem is represented by abstraction
  - ❑Remove irrelevant details in representation
- ❑A representation can simplify or complicate the resolution of the problem
  - ❑Eg. Representation of the 8-puzzle problem → (is it usefull to solve the problem?)
    - ❑The state must not include information irrelevant to the solution of the problem: for example, the material or colour of the board.
    - ❑States: location of each tile(and gap) in each of the 9 squares on the board.
    - ❑Initial state: initial arrangement of the tiles on the board
    - ❑Operator cost: In this example, we assume that an operator has cost 1
    - ❑Cost of the path: sum of the costs of the applied operators = number of steps taken.

State space and operators: abstraction levels

- ❑States and operators must be defined in such a way that their definitions are compatible and complementary, in order to facilitate the resolution of the problem.
  - ❑Operators must be
    - ❑Deterministic: applied to the same state always give the same result
    - ❑As general as possible: parameterize them
    - ❑Reduce the number of different rules (operators).
- ❑8-puzzle example
  - ❑States: Layouts of the 8 numbered tiles and the gap: 9! different states
  - ❑There are different ways to define the operators

1. List of 9! successor states and states: maximum 9! * 4 *-- too specific*
2. move (<tile>, <direction>): 8 * 4 operators *-- preferable, but can be improved*

<tile> ∊ {1, 2,…, 8}; < direction> ∊ {up, down, left, right}

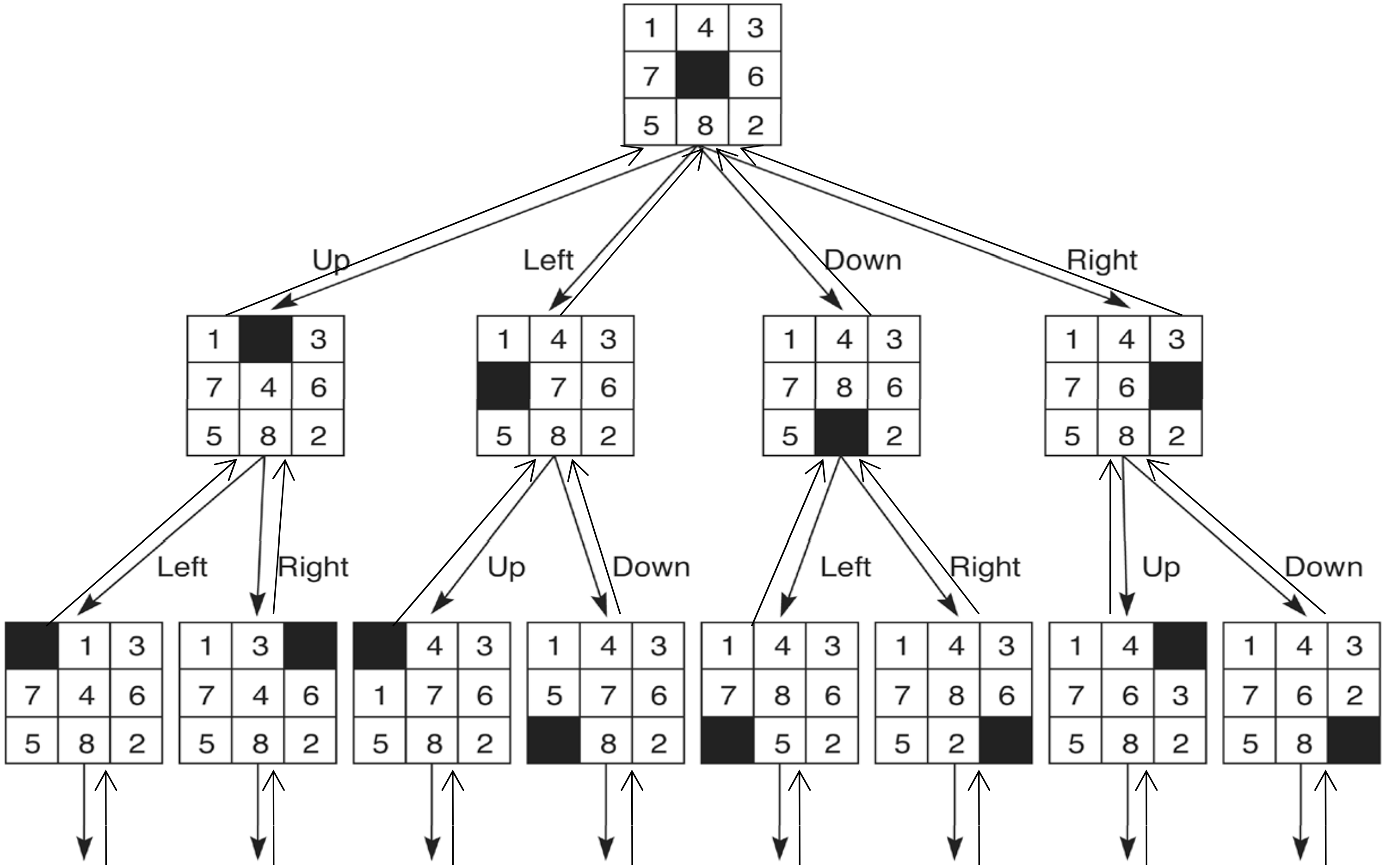
3. move_gap (< direction>): 4 operators *-- optimum*

< direction> ∊ {up, down, left, right}

i.e 4 operators to move the gap up, down, right or left

Search tree for the 8-puzzle

- Operators according to formalization 3



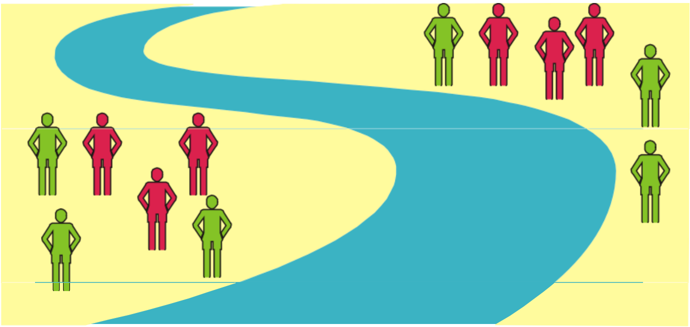
Problem representation: description and implementation

- Description of representation:
  - Specify states and operators
  - Through diagrams, pseudocode, etc.
- Representation implementation:
  - To solve the problem, states and operators are represented by a formal language (i.e. a programming language)
    - States are implemented through a data structure
    - Operators are implemented through operations (functions) in said formal language

- Example: Representation of the states of the 8-puzzle
  - Description
    - States: location of each tile and the gap in each of the 9 squares
  - Deployment: various options
    - 3 * 3 matrix,
    - Vector of length 9,
    - Fact set {(top_ left = 3), (top_center = 8), ...}

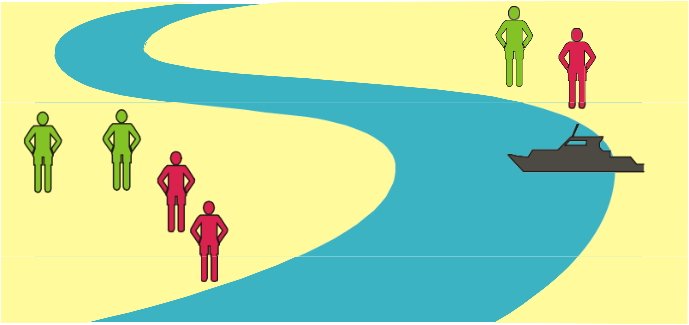
Example: missionaries and cannibals

- ❑ Statement 3 missionaries and 3 cannibals on a river bank along with 1 boat
- ❑ The goal is for everyone to pass to the other shore
- ❑ There are two restrictions
  - ❑ They must cross using the boat in which only 1 or 2 people can go
  - ❑ In case there are missionaries, there cannot be more cannibals than missionaries on any of the banks or the boat



- ❑ Represent the problem according to the state space paradigm and draw the state space

Example: missionaries and cannibals



- □ There are a total of 7 objects → do we save the position of each of them?

1. Identify the specific people: (m1, m2, m3, c1, c2, c3, b)

   State = (1, 1, 0, 1, 0, 1, 1) [0: right bank, 1: left bank]

2. It is preferable to abstract and leave out irrelevant characteristics such as the specific identity of each of the missionaries and cannibals: indicate only the number of missionaries, cannibals on each shore and position of the boat

   (nm_lb, nc_lb, nm_rb, nc_rb, b) [nm_lb + nm_rb = 3, nc_lb + nc_rb = 3]

   State = (2, 2, 1, 1, 0)

3. Taking into account that there are a total of 3 missionaries and 3 cannibals, we can only represent the number of missionaries and cannibals on the left bank.

   (nm_lb, nc_lb, b) [nm_rb = 3 - nm_lb; nc_rb = 3 - nc_lb]

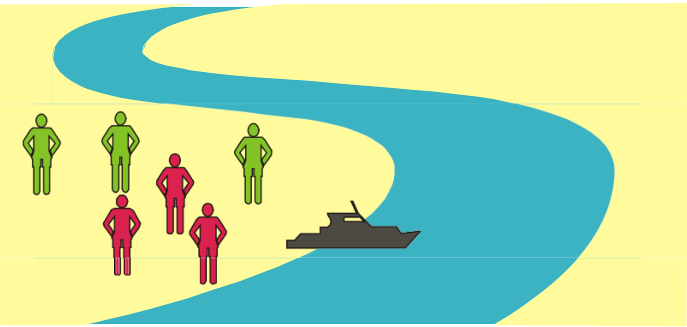
   State = (2, 2, 0)

Missionaries and cannibals: states

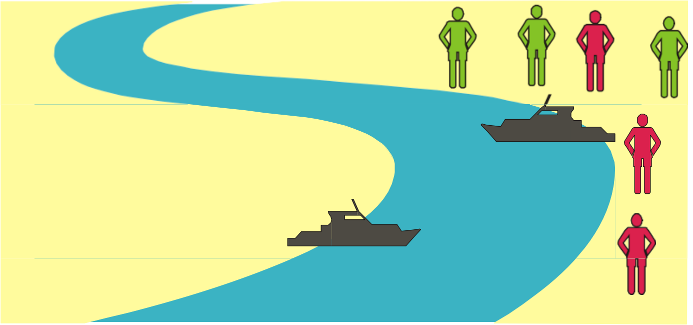
- Representation Description STEP 1: define the state
  - Status = number of missionaries, cannibals and boat on the shore of departure
  - We assume:
    - The starting bank is the left bank of the river
    - The river crossing occurs in one step (the time it takes to cross the river is irrelevant to solving the problem)
  - State = (nm, nc, b)
    - nm_lb: number of missionaries on the left bank (0, 1, 2 or 3)
    - nc_lb: number of cannibals on the left bank (0, 1, 2 or 3)
    - b: boat position (0 = right bank or 1 = left bank)
- The shore the boat is on is important. Specifically, it is used to determine whether or not the operators can be applied
  - $(2, 1, 0) \neq (2, 1, 1)$

Missionaries and cannibals: initial and target state

STEP 2: define initial and target states

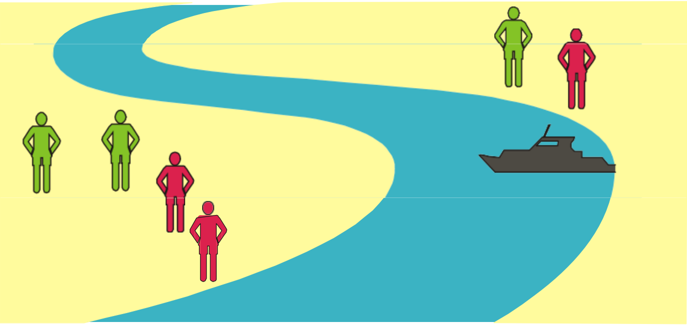


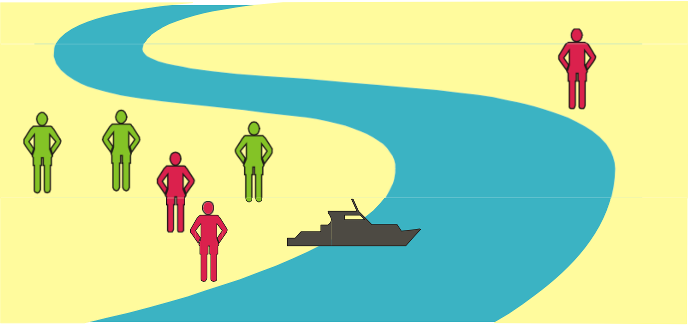
Initial state (3, 3, 1)



Target state (0, 0, 0)

(0, 0, 1) not possible





(2, 2, 0)

Possible intermediate states

(3, 2, 1)

Missionaries and cannibals: actions

Operators: What determines a state to change? (STEP 3)

- There are 5: the boat always crosses the river with 1 or 2 people
  - crossM: Cross with 1 missionary.
  - crossMM: Cross with 2 missionaries.
  - crossC: Cross with 1 cannibal.
  - crossCC: Cross with 2 cannibals.
  - crossMC: Cross with 1 missionary and 1 cannibal
- Cost: we assume that the operator cost = 1

Path cost = number of times the river is crossed

- Restrictions
  - Preconditions: The shore the boat is on is important
    - E.g. no missionaries could cross in states (3, _, 0) and (0, _, 1)
  - Postconditions: In case there are missionaries, the number of cannibals cannot exceed the number of missionaries on either of the two shores.

- crossM (nm_lb, nm_rb, B)
  - Preconditions: $\{(nm\_lb > 0 \wedge b = 1) \vee (nm\_lb < 3 \wedge b = 0)\}$
    - if $b = 1$ then $nm\_lb := nm\_lb - 1 \wedge b := 0 \rightarrow (nm\_lb - 1, nc\_lb, 0)$
    - if $b = 0$ then $nm\_lb := nm\_lb + 1 \wedge b := 1 \rightarrow (nm\_lb + 1, nc\_lb, 0)$
  - Postcondition: The final state must be an allowed state.

Missionaries and cannibals: restrictions

- **Situations not allowed**: In case there are missionaries, there may be more cannibals than missionaries on either shore or in the boat? On the banks?

We will define the operators in such a way that these situations cannot occur.

- **On the boat**: A maximum of 2 people travel, so there can be no more cannibals than missionaries on it.
  - If the maximum had been greater than 2, there would be situations not allowed (and it would be another problem)

- **On the banks**: Given the state (nm_lb, nc_lb, b)
  - (3, 3, 1) and (2, 2, 0) are allowed states
  - (1, 2, 0) and (2, 3, 0) are illegal states
  - Will nm_lb <nc_lb suffice as an illegal status condition?
    - Is (2, 1, 0) possible?
      - On the right bank there is 1 missionary with 2 cannibals!
    - Is (0, 2, 1) possible?
      - If there are no missionaries the state is not possible!

- **Condition for forbiden states**:

$$(nm\_lb < nc\_lb \wedge nm\_lb \neq 0) \vee (nm\_lb > nc\_lb \wedge nm\_lb \neq 3)$$

Missionaries and Cannibals: State Space

State space size

(nm_lb, nc_lb, b)

nm_lb ∈ {0,1,2,3}, nc_lb ∈ {0,1,2,3}; b ∈ {0,1}

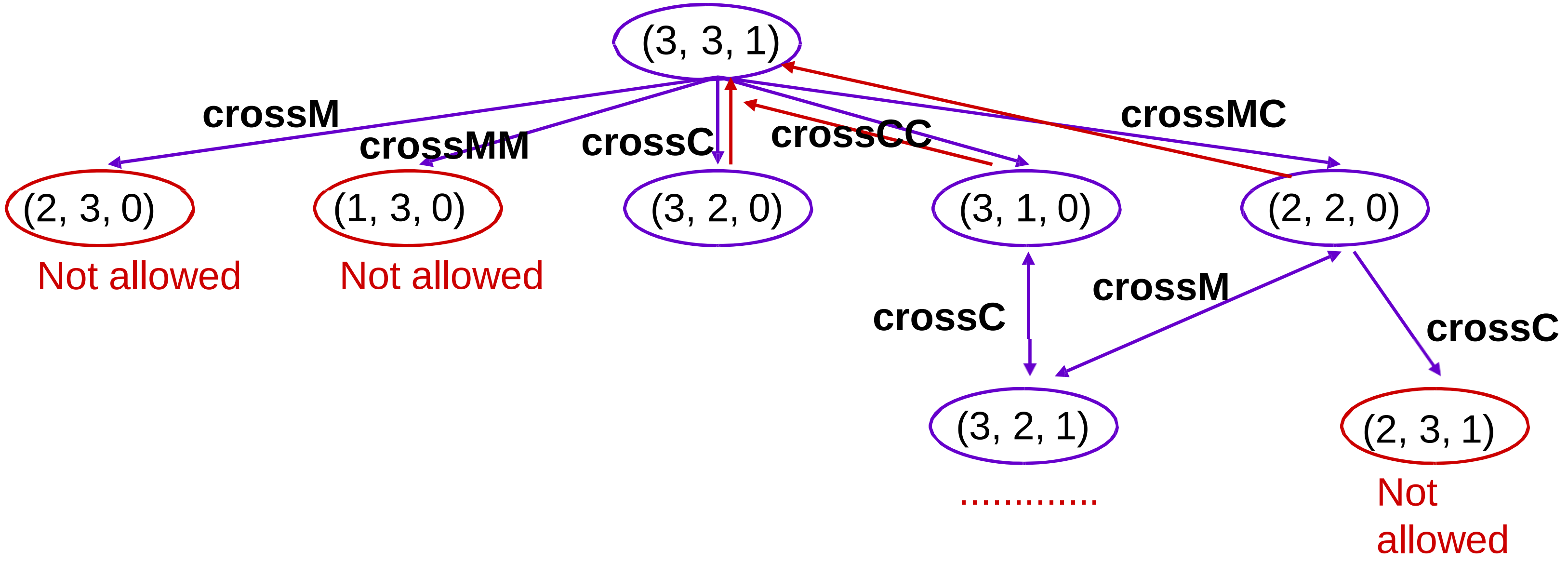
- Maximum: 4 * 4 * 2 = 32 possible states
  - There are 4 unreachable states (hence there are 28 reachable states)
    - Obvious like (0, 0, 1) and (3, 3, 0)
    - And not as obvious as (3, 0, 1) and (0, 3, 0)
  - There are 12 forbiden states
    - (1, 2, _), (2, 3, _), (1, 3, _), (2, 1, _), (2, 0, _), (1, 0, _)

State space: 16 reachable states allowed

- Sometimes the forbiden status condition is not included as a postcondition in the operator, but is checked later in the search algorithm.
  - In that case, the state space is made up of 12 + 16 = 28 states.

Missionaries and Cannibals: graph

- State space represented as a directed graph (there may be cycles)



Develop the rest of the state space from the state (3, 2, 1)

Missionaries and Cannibals: summary

- Formal representation of the problem
- Initial state: (3, 3, 1). Target state: (0, 0, 0)
- Restrictions: Forbiden States (Danger Condition (NM, NC, B))
  - (nm_lb <nc_lb ∧ nm_lb ≠ 0) ∨ (nm_lb> nc_lb ∧ nm_lb ≠ 3)
- Operators: crossM, crossMM, crossC, crossCC, crossMC
  - Specification example for crossM (NM, NC, B)
    - Preconditions
      - (b = 1 ∧ nm_lb> 0) ∨ (b = 0 ∧ nm_lb <3)
    - Post-conditions
      - (nm_lb, nc_lb, 1) → (nm_lb - 1, nc_lb, 0)
      - (nm_lb, nc_lb, 0) → (nm_lb + 1, nc_lb, 1)
      - Destination state is an allowed state
    - Operator cost: 1
  - Specification for crossMM, crossC, ...
- Cost of the solution = number of applied operators

Representation: think in states and operators

- ❑ The representation of states affects
  - ❑ the ease / difficulty of specifying operators
  - ❑ and the complexity to solve the problem
- ❑ Missionaries example
  1. Formalization: (m1, m2, m3, c1, c2, c3, B)
     - ❑ crossesM1 (m1, m2, m3, c1, c2, c3, b)
       - ❑ Preconditions: {m1 = b}
       - ❑ Post-conditions:

         if b = left then m1: = right $\wedge$ b: = right
         if B = right then M1: = left $\wedge$ b: = left
         target status not dangerous
     - ❑ 21 operators should be specified
       - ❑ 3 to cross a missionary, 3 to cross a cannibal
       - ❑ 3 to cross two missionaries, 3 to cross two cannibals
       - ❑ 9 to cross a cannibal and a missionary
  2. Formalization: (nm_lb, nc_lb, nm_rb, nc_rb, b)
     - ❑ Redundant information means making changes to more components

Representation: think in states and operators

- 8-puzzle example: possibilities for operators
  - Focusing on states (1 operator / 1 state)
    - 9! * 4 = 1,451,520 operators
  - Focusing on the tile to move it was more doable
    - 8 * 4 operators
    - The specification of the operators can be parameterized
      - Depending on how the representation of states is done
      - It should make it easier to locate and change the position of a specific piece
      - left (piece), right (piece), up (piece), down (piece)
    - Although it is easier that way, 32 operators would continue to appear
  - Focus on the gap (the best option)
    - 4 operators left

Lesson 2: Problem solving using search

1. Representation of problems in a state space
2. Problem solving with search strategies
    - ❑Uninformed or blind methods
    - ❑Informed or heuristic methods
    - ❑Search between adversaries

Uninformed/ blind search

When we do not have additional information about the states beyond the definition of the problem. We can only generate successor states and check if it is the solution or not.

- Exhaustive exploration of the search space
  - until a solution is found
- They do not incorporate knowledge to guide the search
  - It is decided a priori which way to go
    - e.g .: first in depth
- The search does not include domain information

Search strategies

- ❑ **Uninformed (blind) search:**
  - ❑ Only the **problem definition** is used in the search
  - ❑ The different search strategies differ in the **order** in which the nodes are expanded.
    - ❑ Breadth-first search.
    - ❑ Uniform cost search.
    - ❑ Depth-first search.
    - ❑ Depth-limited search.
    - ❑ Iterative deepening depth-limited search
    - ❑ Bidirectional search.

- ❑ **Informed (heuristic) search**
  - ❑ **Use heuristics** (estimates of how far a given state of the target state is) to guide the search.
  - ❑ Incorporates domain knowledge to guide the search in the form of "clues" to guide the search process and make it more efficient
  - ❑ **Heuristic function:** h(n)

  The value of the heuristic for state n provides an estimate of the cost of the optimal path from state n to a target state.

Search tree

- In almost all cases the search is in a **graph search** (there can be multiple paths from the initial node to a given node).
- Let's focus on searching in a **tree** (only one path from the root node to a given node)
  - The **nodes of a search tree** correspond to search states.
  - The **root node** of a search tree corresponds to the initial search state.
  - **Actions: Expand** the current search node: Generate child nodes (corresponding to new states) by applying the **successor** function to the current node.
  - **Goal state:** A node corresponding to a state that satisfies the **goal test**.
  - **Utility**: Cost of the path from root node to current one.

Terminology:

- **Parent node**: Node of the tree from which the current node was generated by applying the successor function only once.
- **Ancestor node**: A node in the tree from which the current node has been generated by applying the successor function one or multiple times.
- **Depth**: Length of the path from root node to current one.
- **Leaf node**: A generated node that has not been expanded yet.
- **Fringe set**: Set of leaf nodes.

Tree search: without removal of duplicate states

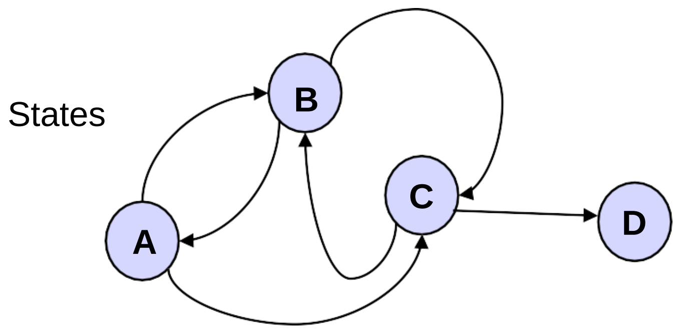
- ❑ Pseudocode for Tree-Search

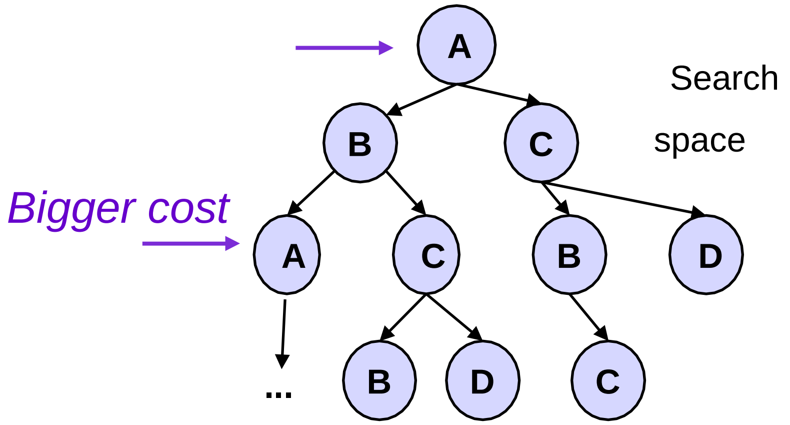
  problem* = {root-node, *expand*, *goal-test*}; *strategy

```
function Tree-Search (problem, strategy )
;;  returns solution or fail
;;  opened-list contains the nodes in the fringe of the search tree
Initialize search-tree  with root-node
Initialize opened-list  with root-node
Iterate
    If (opened-list  is empty) then return fail
   Choose from opened-list, according to strategy, a node to expand.
         If (node  satisfies goal-test )
            then return solution (path from root-node  to current node )
            else remove node from opened-list
                    expand node
                       add child nodes to opened-list
```

Search space via search tree

- Tree with **only** generated nodes in the search for the solution
  - It depends on the search algorithm used (the strategy)
  - Although the state space is finite, ……
  - The search space can be infinite
  - Different nodes in the search tree can correspond to the same state in the state space





  - A node in the tree represents a path from the root to a certain state

Search tree NODE data types :

(state, parent_node, operator, depth, cumulative_path_cost)

Lets Remember Terminology

- opened-list structure
  - Nodes are saved with generated states pending expansion
- Closed-list structure
  - Those already expanded or visited are saved (only in some algorithms)
- Node types
  - Generated
    - Those that appear or have appeared in ***opened-list*** nodes
  - Generated but not expanded
    - Those that appear in the ***opened-list***
  - Expanded:
    - Those that appear in the ***closed-list***
    - A state expands when
      1. A node representing it is removed from the ***opened-list***,
      2. All its descendants are generated and they are added to the opened-list
      3. It joins the ***closed-list***,
    - A state can be expanded multiple times (on different nodes)
      - A node is expanded only once

Deleting duplicate states

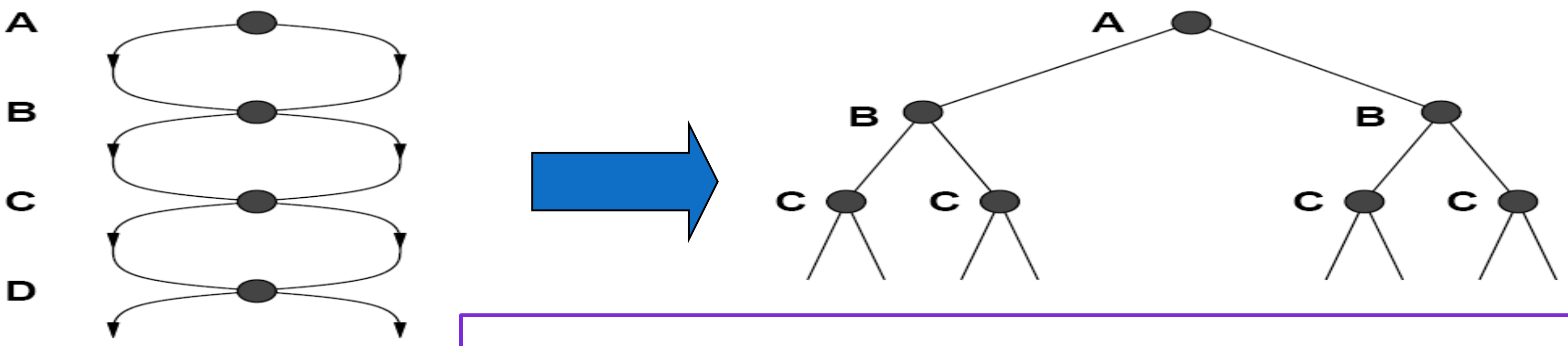
- ❑If the repetition of states is not detected, the complexity of the search problem can increase exponentially
  - ❑For example, in a search on a regular mesh
  - ❑Trivial example

Can fall in infinite loops

→ cycle control

Assuming a maximum depth m

The search tree has **2^m** leaf nodes



Changes properties/perfomance of search algorithms

- ❑Deleting duplicate states:
  - ❑Save expanded nodes in closed-list.
  - ❑Do not expand a candidate node if it is already in closed-list
    - ❑Extra memory requirements.
    - ❑**Optimal only** in Breadth-first search when the cost of all actions is identical and in search with uniform cost.
    - ❑Finding the optimal path with other strategies is not guaranteed.

Graph search: with elimination of duplicate states

- pseudocode for Graph-Search

    problem* = {*root-node*, *expand*, *goal-test*}; *strategy

```
function Graph-Search (problem, strategy )
;;  returns solution or fail
;;  opened-list contains all nodes of the fringe of search-tree
Initialize search-tree with root-node
Initialize opened-list  with root-node
Initialize closed-list  as an empty list
    Iterate
    If (opened-list  is empty)
        then return fail
    Choose from opened-list, according to strategy, a node  to expand.
        If (node satisfies goal-test )
            then return solution (path from root-node  to node)
    remove node from opened-list
    If (node  is not in closed-list )
        then  add node  to closed-list
         expand node
         add children node to opened-list
```

Performance evaluation of search strategies

- Performance of an algorithm in solving search problems
    - Completeness Does it guarantee to find a solution if it exists?
    - Optimality. Find the lowest cost solution?
    - Cost of the solution: given by the utility function.
    - Computational cost of the search
        - **Temporal complexity**. How long does it take to find a solution? Is it the worst case: the target node is the last in the tree (number of expanded nodes)
        - **Spatial complexity** (memory usage). How much memory is needed in the worst case? (maximum number of nodes that need to be stored simultaneously in memory)
- Total cost: 2 types of costs
    - Cost of the solution: sum of the cost of operators used on the road
        - Optimization problems: cost of the solution (independent of the path)
    - Computational cost of the search: complexity of the algorithm
- A compromise between both costs must be reached
    - Obtain the best possible solution with available resources
        - We do not want a good solution that costs (time, memory) to find

Performance evaluation of search strategies

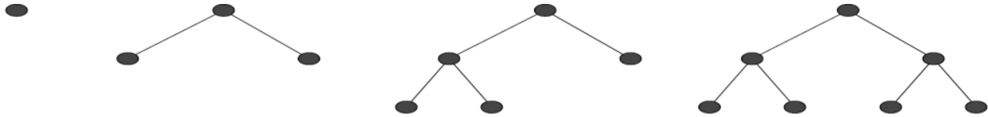
Cost of the search (complexity analysis):

- ❑Branching factor (*b*): maximum number of successors (children) of any node.
- ❑Depth of the most superficial target node (*d*)
- ❑Maximum depth of the search tree (*m*)
- ❑Minimum cost of each action (ε)

❑Hypothesis:

- ❑Finite branching factor (*b*).
- ❑From the point of view of cost analysis, all operations have the same cost
- ❑The maximum depth of the tree of search (*m*) can be infinite.
- ❑Cost of the path = sum of the costs of each step (not negative).

Breadth-first search



- Nodes are expanded in order of depth
    - Nodes of depth *p* expand before nodes of depth *p + 1*
    - For the opened-list, use a FIFO (first-in-first-out) queue for the list of candidates to be expanded.
    - New nodes generated are inserted at the end of the queue ⇒ the shallowest nodes expand before the deeper ones.
- Properties (evaluation criteria)
    - **Complete**: guarantees to find a solution (the shallowest of the possible ones)
    - **Optimal**: only if the path cost is a non-decreasing function of the node depth (the shortest path may not be optimal)
    - Time complexity is exponential
        - Assuming a maximum branching factor b (number of children of a node) and a path to the shallowest solution d, the number of nodes expanded in the worst case is

$$1+b+b^2+\ldots+b^d-1=\frac{b^{d+1}-b}{b-1}=O(b^d)$$

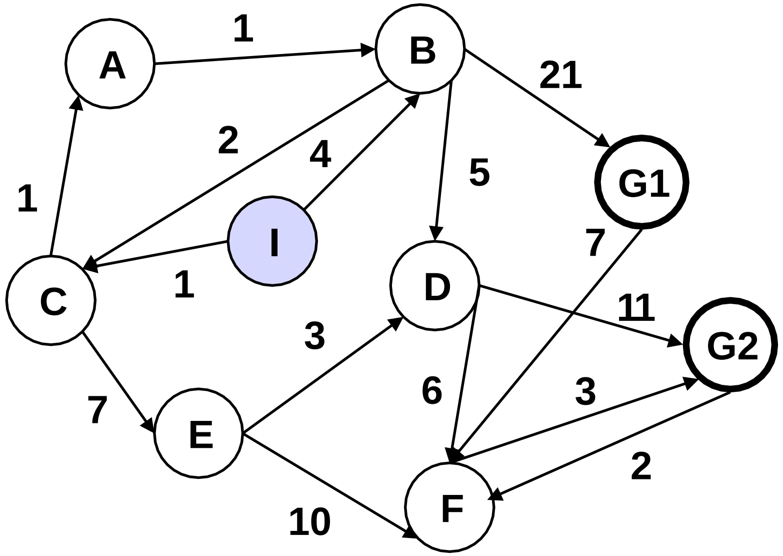
        - the number of nodes generated in the worst case is

$$b+b^2+\ldots+b^d+\left(b^{d+1}-b\right)=\frac{b^{d+2}-b^2}{b-1}=O(b^{d+1})$$

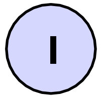
    - Spatial complexity is exponential *O(b^d)*
        - All nodes generated must be kept in memory
        - Major issue: memory (only workable for small cases)
- Cycles: With ***tree-search*** (without elimination of repeated states) solutions are not lost, although they suppose inefficiency.
- If there is no solution, the algorithm might not finish.

Example: Breadth-first search

State Space



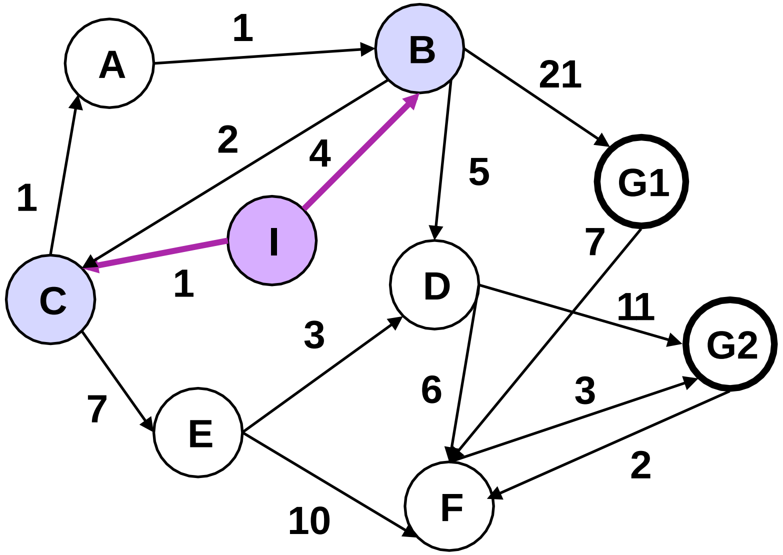
Search Tree



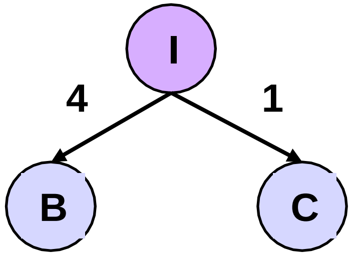
opened-list (queue): (I)

Example: Breadth-first search

State Space



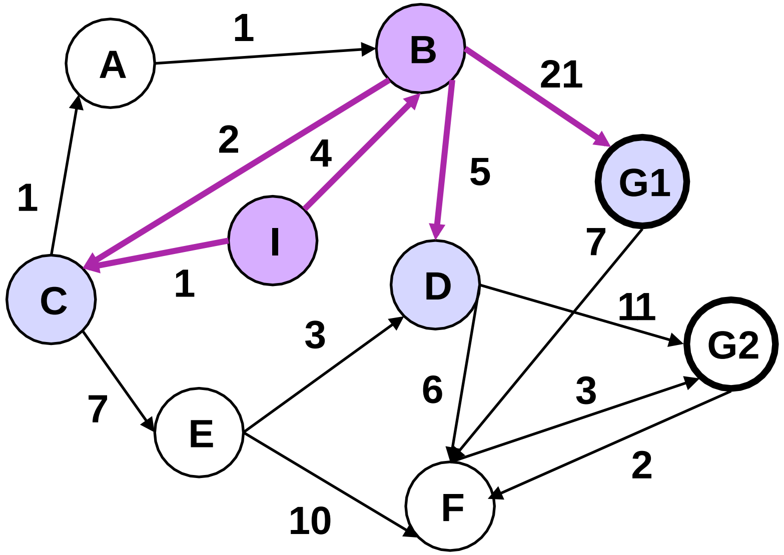
Search Tree



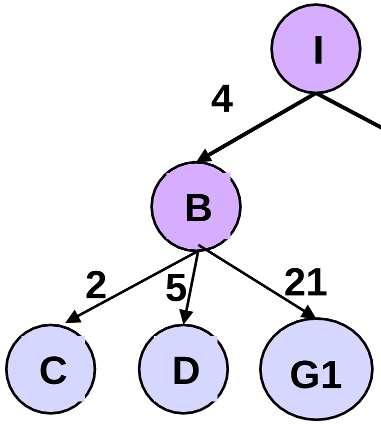
opened-list (queue): (B C)

Example: Breadth-first search

State Space



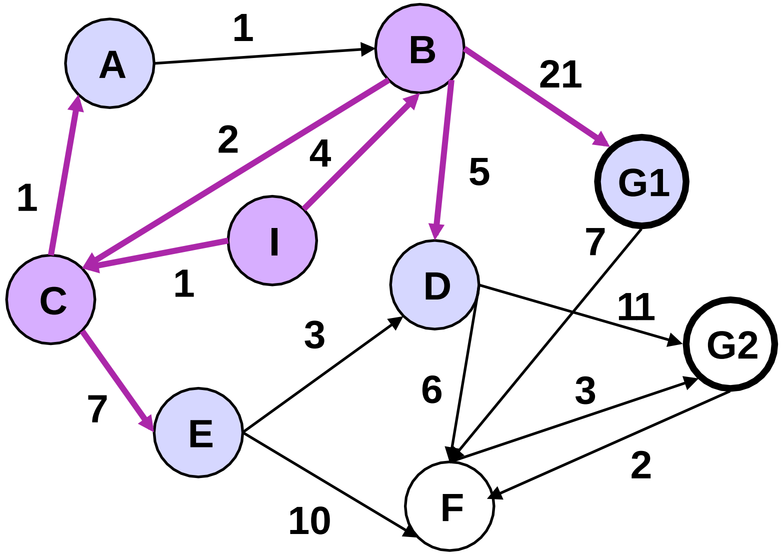
Search Tree



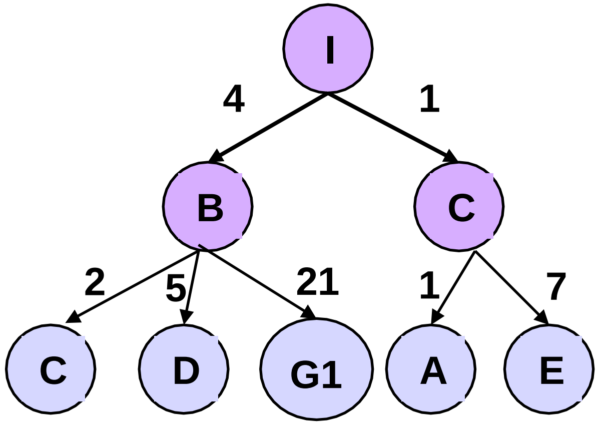
opened-list (queue): (C C D G1)

Example: Breadth-first search

State Space

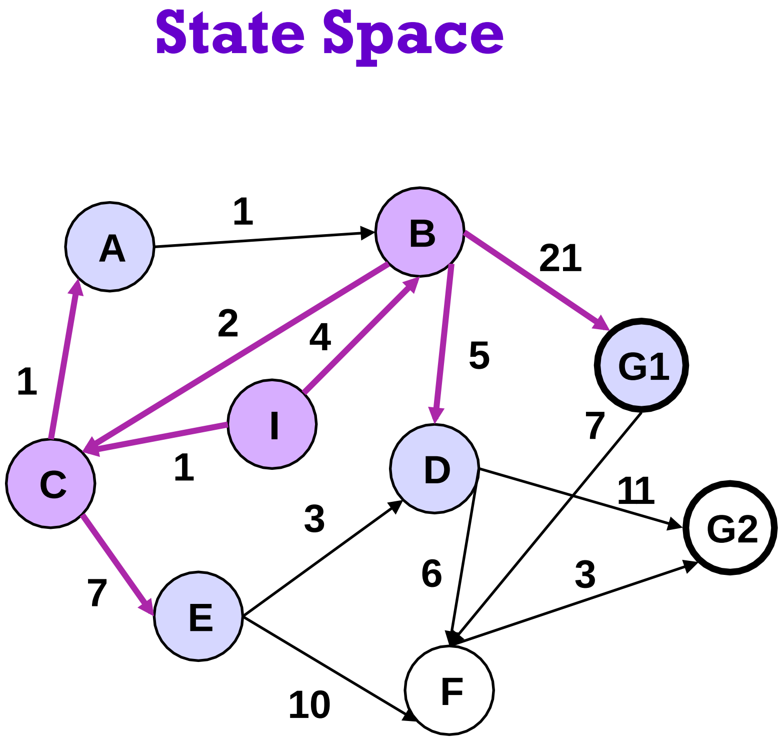


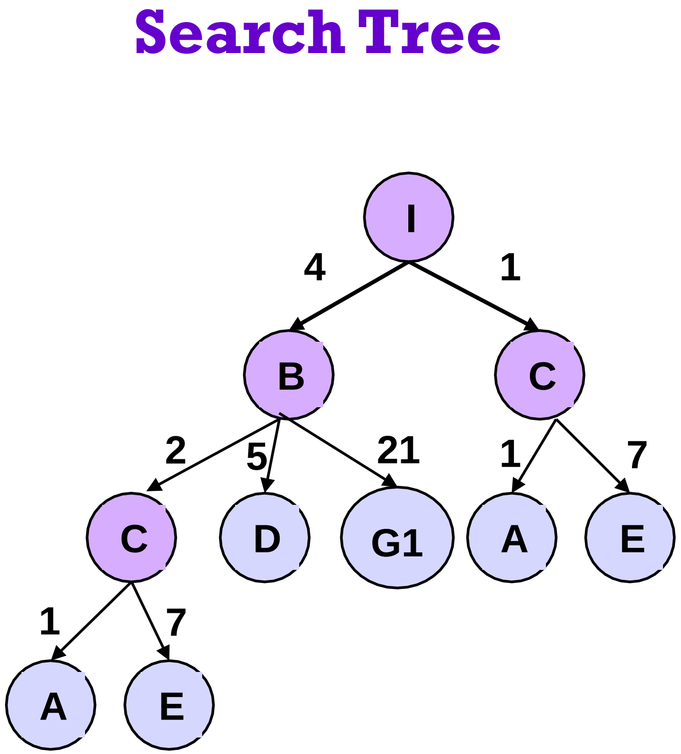
Search Tree



opened-list (queue): (C D G1 A E)

Example: Breadth-first search

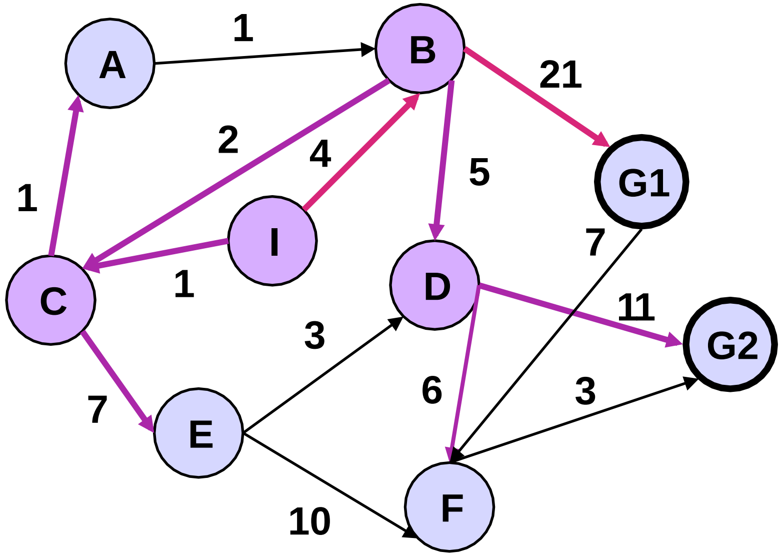




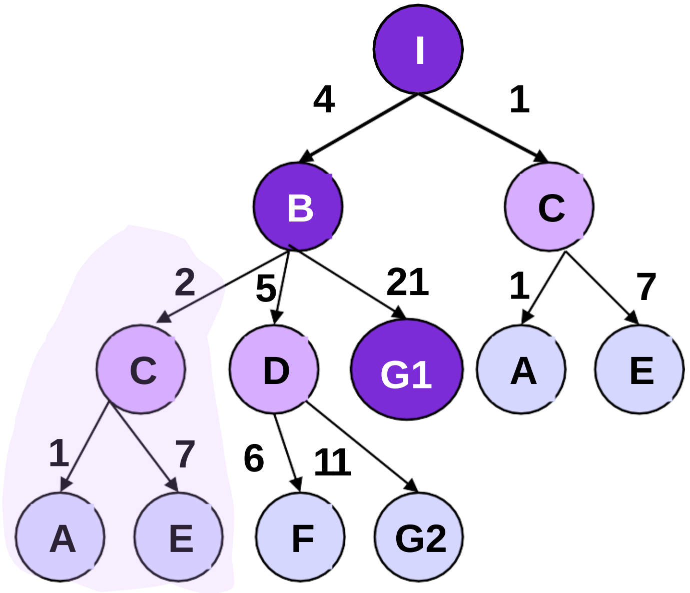
opened-list (queue): (D G1 A E A E)

Example: Breadth-first search

State Space



Search Tree



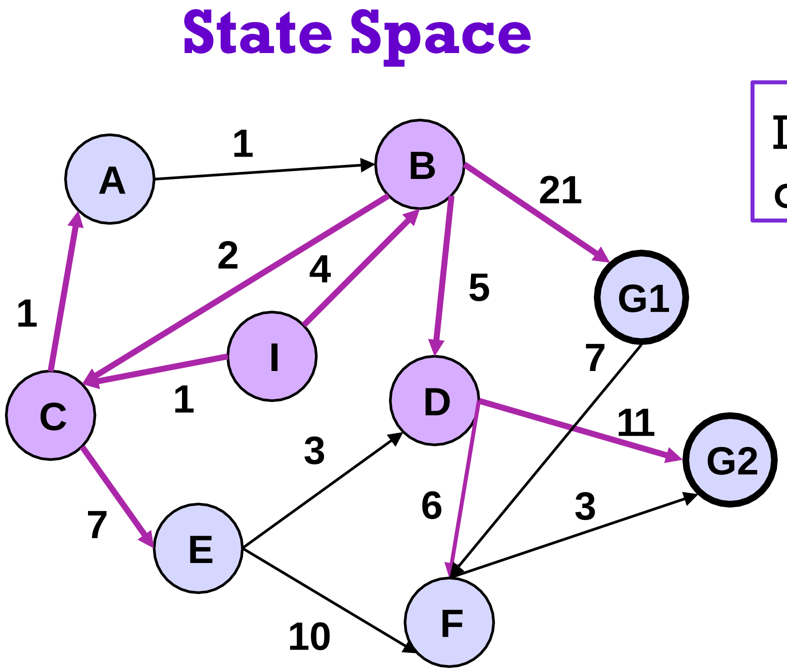
Expanded Nodes (in order): I B C C D G1

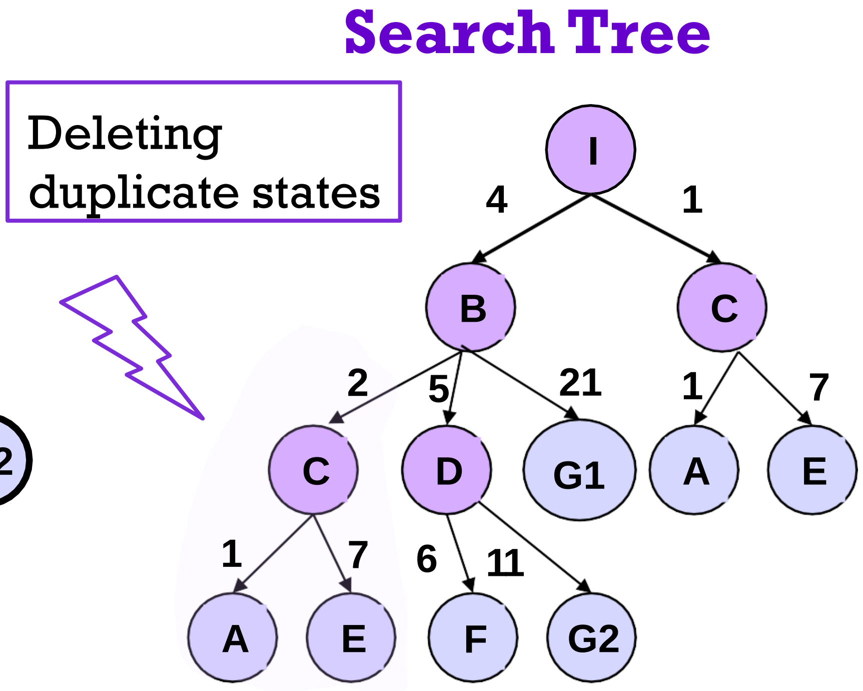
Generated Nodes (in order): I B C C D G1 A E A E F G2

Path to solution: I B G1 Cost: 4+21 = 25

IMPORTANT. Until the node is visited, the solution is not found. Although it is already generated

Example: Breadth-first search



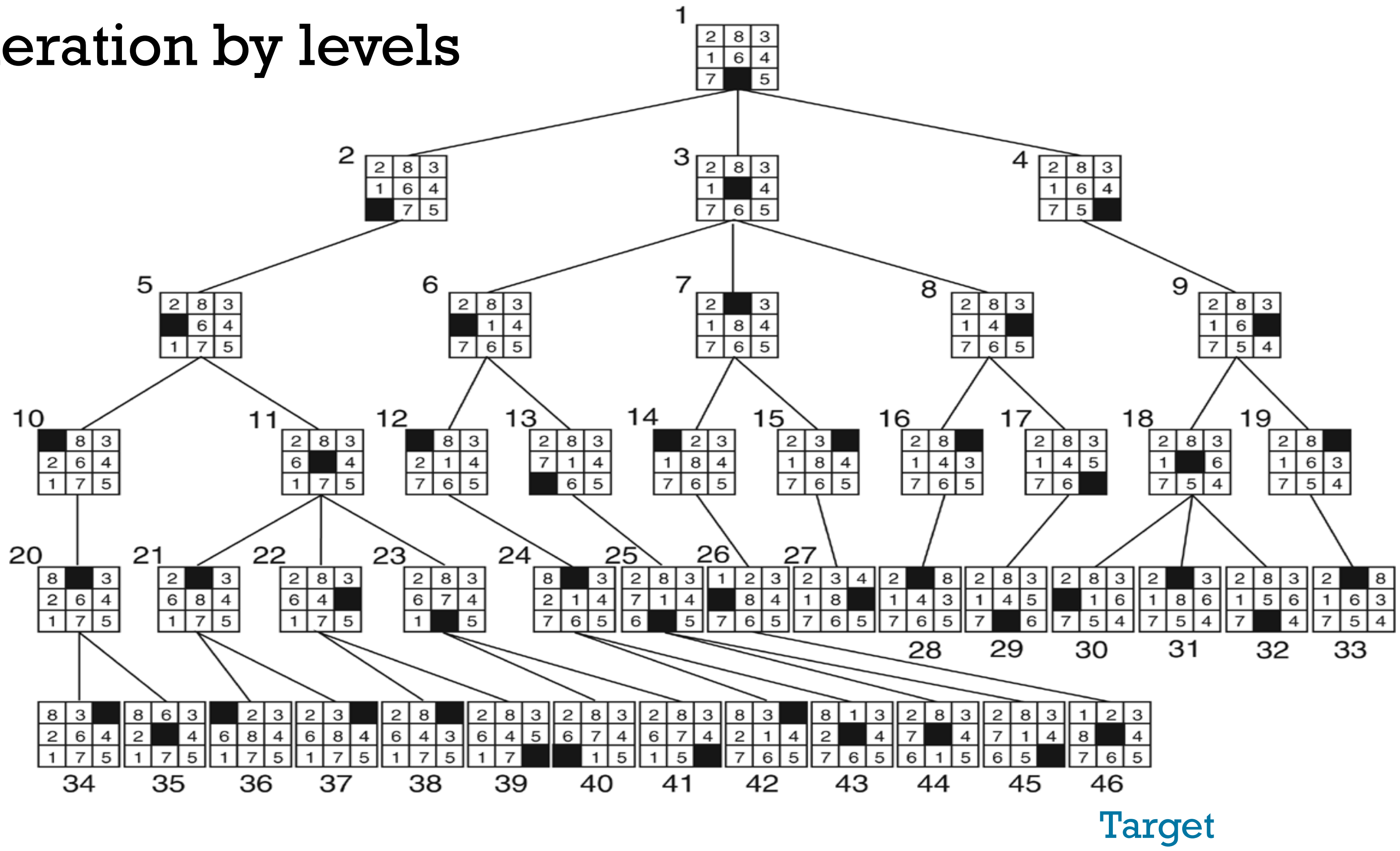


opened-list (queue): (G2 F E A E A G1)

- It's complete: Finds the shallowest (less deep) solution.
- It is not optimal: The solution found is **not** the one with the **lowest cost**, because (unless) the path cost increases monotonously with depth.

Example: Breadth-first search 8-puzzle

- ❑Tree generation by levels



Uniform cost search

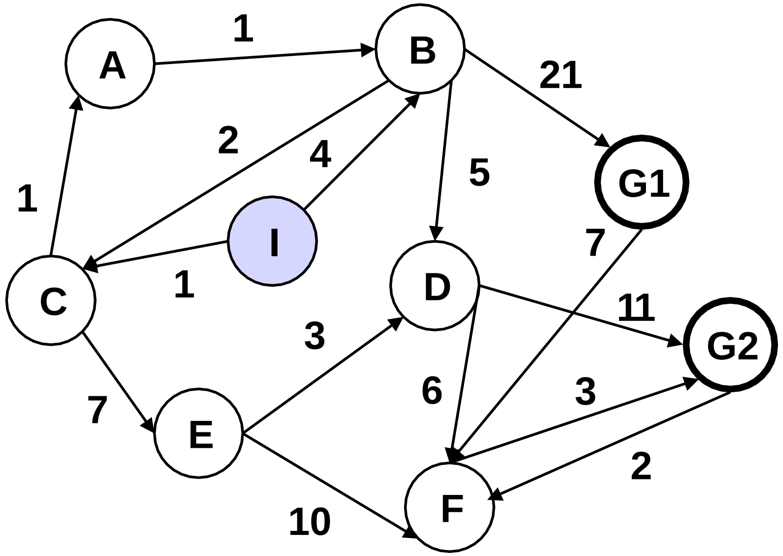
- Nodes are expanded in increasing order of (accumulated) path cost
    - At each step: From the nodes in the opened-list, the node with the lowest path cost is expanded until reaching the target
- opened-list is implemented with a priority queue (nodes ordered from lowest to highest path cost)
- Cost of the path versus the number of steps
- If node path cost increases monotonously with depth → uniform cost search equals breadth-first search
- Properties:
    - Complete if there are no paths of infinite length and finite cost.
    - Optimal if cost-path (successor (n)) $\geq$ cost-path (n)
    - It is satisfied when all operators have cost ≥ 0
    - Complexity in space and time equivalent to first in width if the cost of the path of the nodes increases monotonously with their depth: $O(b^d)$
    - If not, in the worst case: $O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$

$C^* \equiv$ Coste de camino de la solución óptima

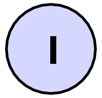
$\varepsilon \equiv$ Coste mínimo (>0) de una acción

Example: Uniform cost search

State Space



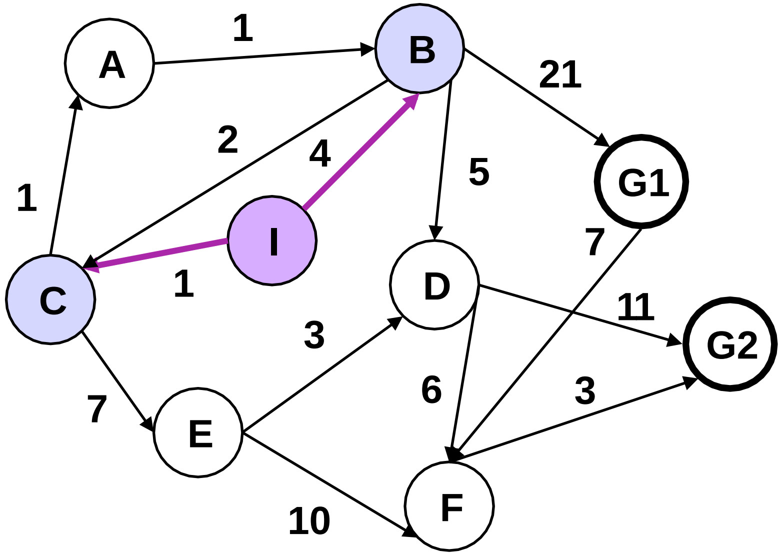
Search Tree



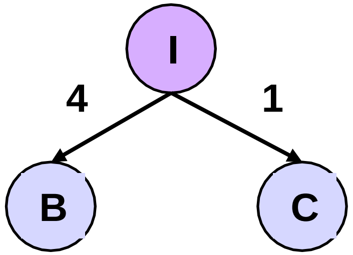
opened-list (priority queue): (I_0)

Example: Uniform cost search

State Space



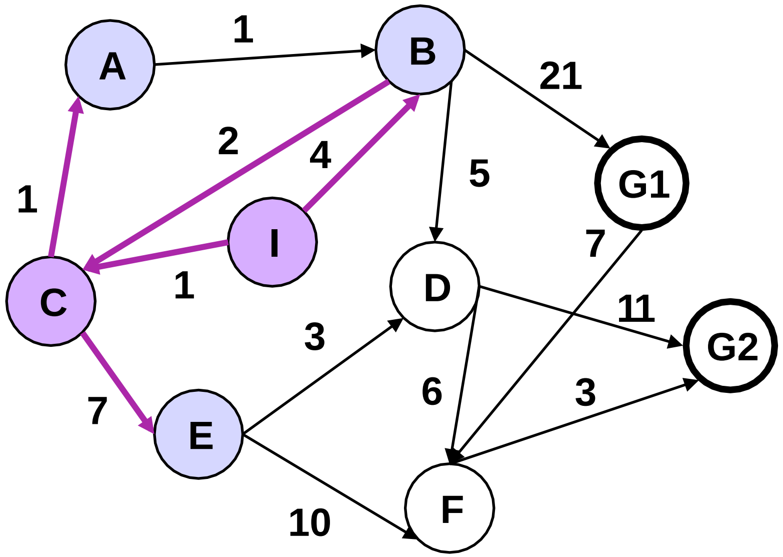
Search Tree



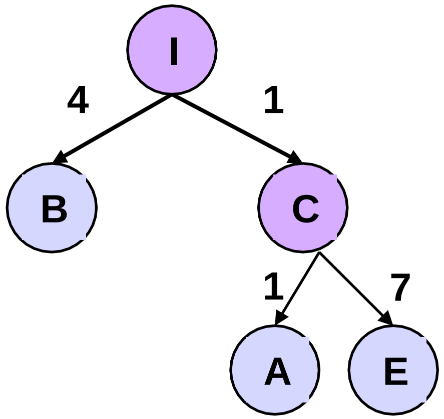
opened-list (priority queue): (C_1 B_4)

Example: Uniform cost search

State Space



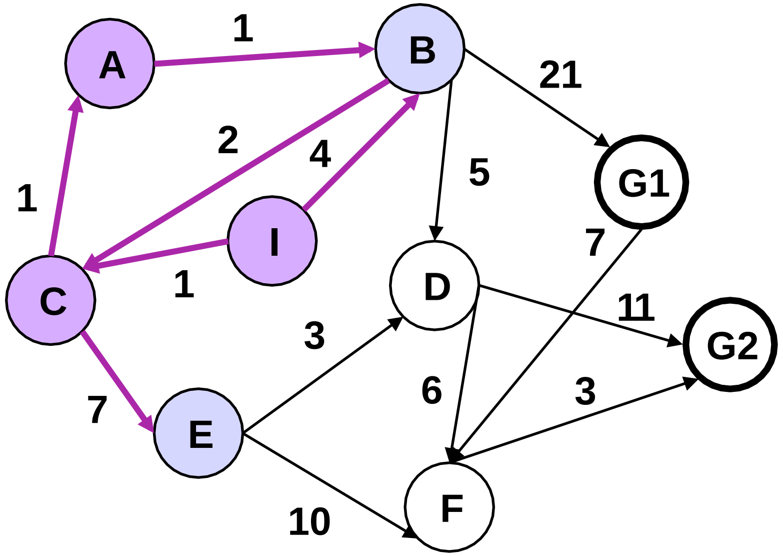
Search Tree



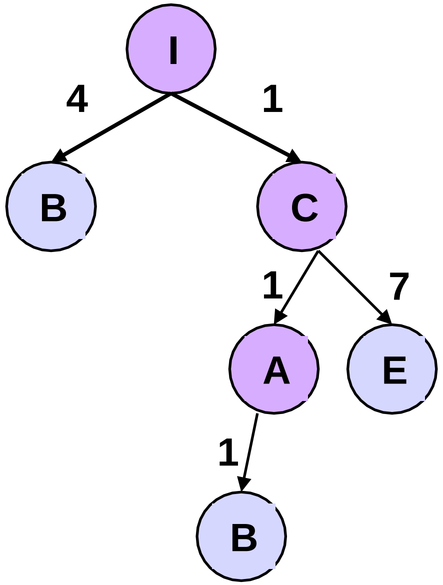
opened-list (priority queue): (A_2 B_4 E_8)

Example: Uniform cost search

State Space



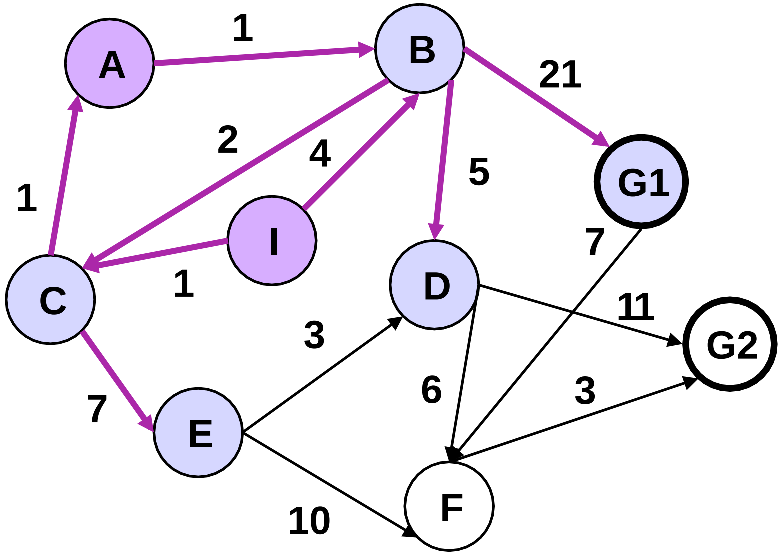
Search Tree



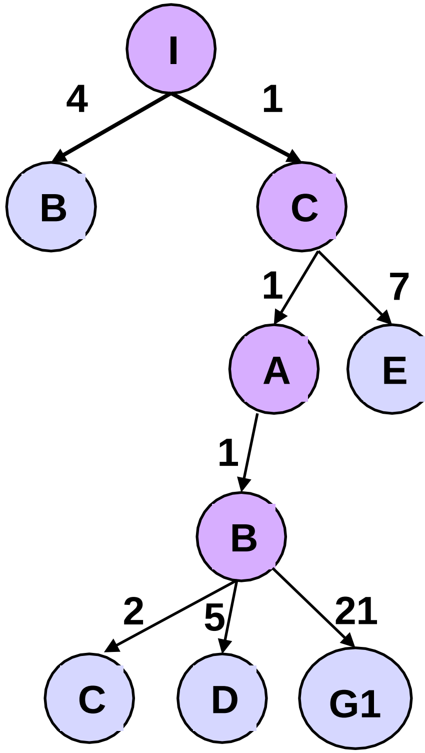
opened-list (priority queue): (B_3 B_4 E_8)

Example: Uniform cost search

State Space



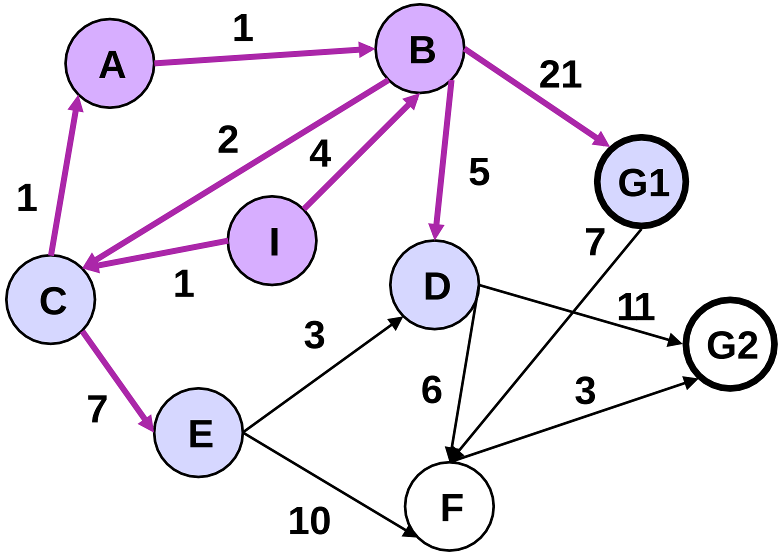
Search Tree



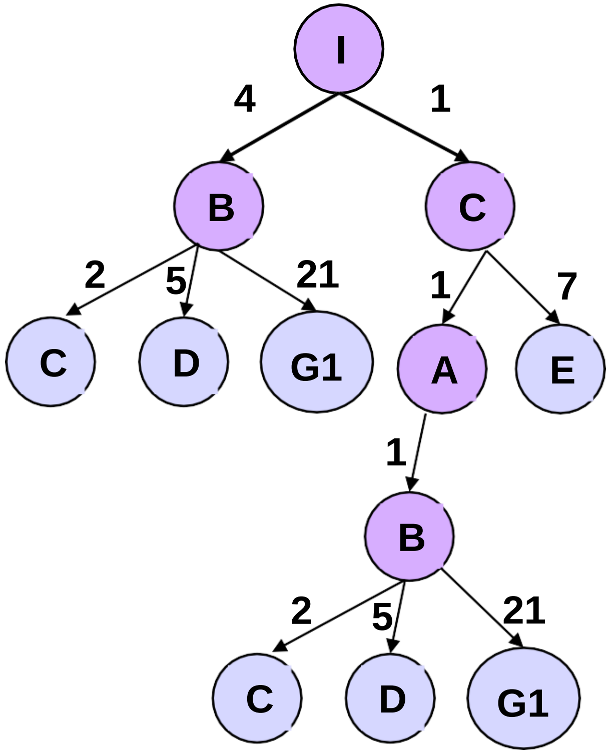
opened-list (priority queue): (B_4 C_5 D_8 E_8 $G1_{24}$)

Example: Uniform cost search

State Space



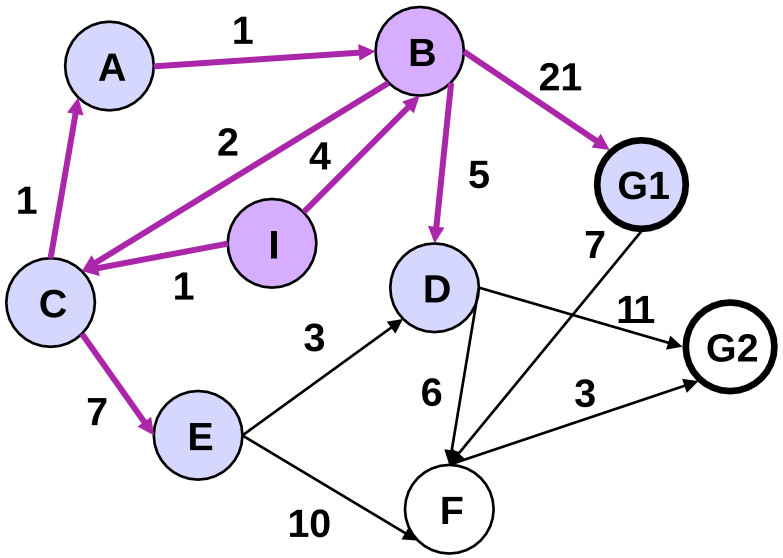
Search Tree



opened-list (priority queue): (C_5 C_6 D_8 E_8 D_9 $G1_{24}$ $G1_{25}$)

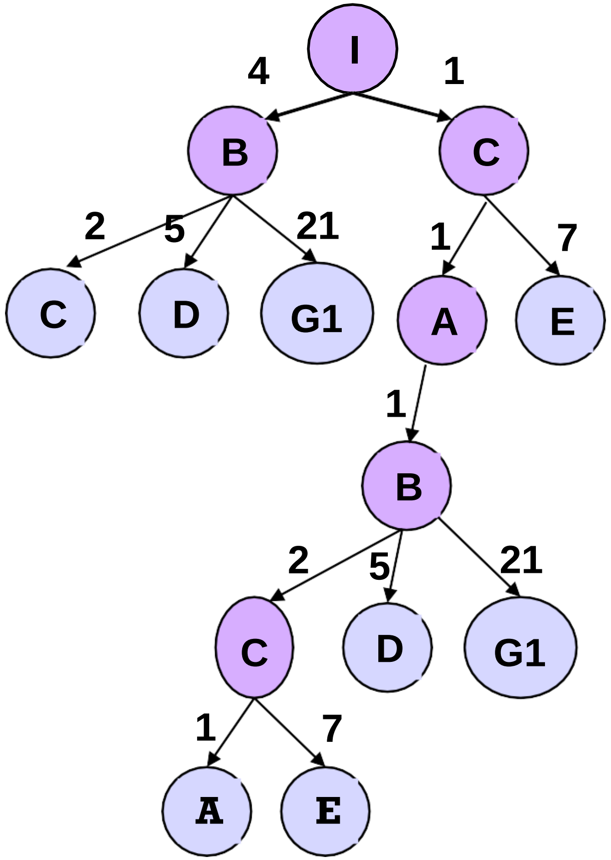
Example: Uniform cost search

State Space



Continues a couple more steps…

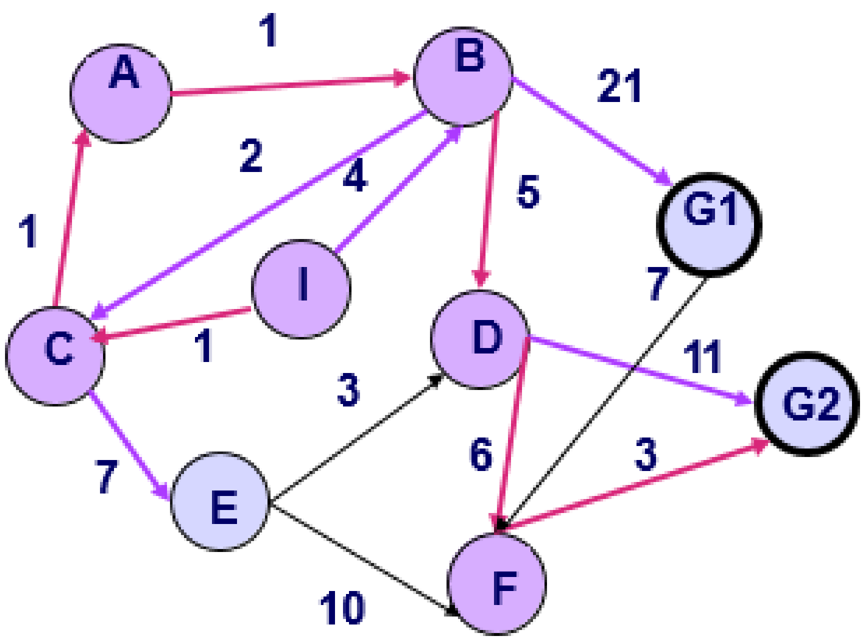
Search Tree



opened-list (priority queue): (A$_6$ C$_6$ D$_8$ E$_8$ D$_9$ E$_{12}$ G1$_{24}$ G1$_{25}$)

Example: Uniform cost search

State Space



Search Tree

- ❑Complete and optimal
  - ❑Expands a lot of nodes
- ❑Here the cycle/loop problem is more acute
  - ❑But a cycle control on opened-list would mean that the strategy would no longer be optimal…
    - ❑In the example: B son of A would not be generated and we wouldn't find an optimal solution
  - ❑cycle control only on expanded-nodes. Does work
  - ❑ Less cost Solution is found at depth 6

Path to solution: I C A B D F G2

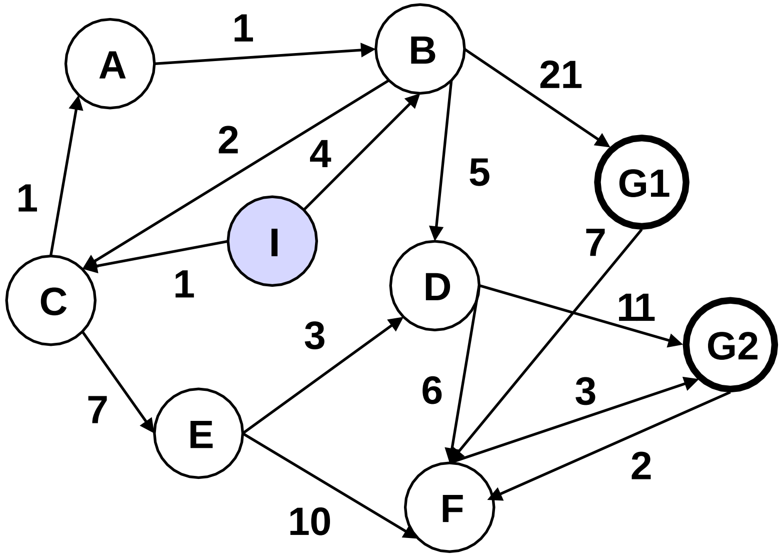
Cost: 1+1+1+5+6+3 = 17

Depth-first search

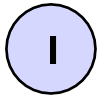
- The most recently generated nodes are the first to expand
  - The node that is expanded is the first in the *opened-list.*
    - Then is removed from opened-list.
    - If the considered node has no successors and is not a target, it is discarded
    - Otherwise it is expanded (and removed from opened-list)
  - Implementation:
    - Use for the *opened-list* a LIFO (last-in-first-out) **stack** to implement the list of candidates to be expanded. New generated nodes are introduced at the beginning of the stack=> deepest nodes are expanded first.
    - Alternative: Recursive function that calls itself by each of its children.
- The opened-list structure is implemented with a stack (or recursion is used)
- Properties:
  - **Not Complete**: The algorithm can explore paths of infinite length.
  - **Non-optimal**: There is no guarantee that if a solution is found, it will be the least costly. Not recommended if **m** (maximum tree depth) is a large value.
  - Space: : *$O(b*m)$,* where **b** is the branching factor-
    - The modest requirements: it is enough to simultaneously store in memory the current path and references to the siblings of the nodes expanded in that branch.
  - Time: *$O(b^m)$*(if there are many solutions, it can be faster than breath-first search; it depends on the application order of the operators).

Example: Depth-first search

State Space



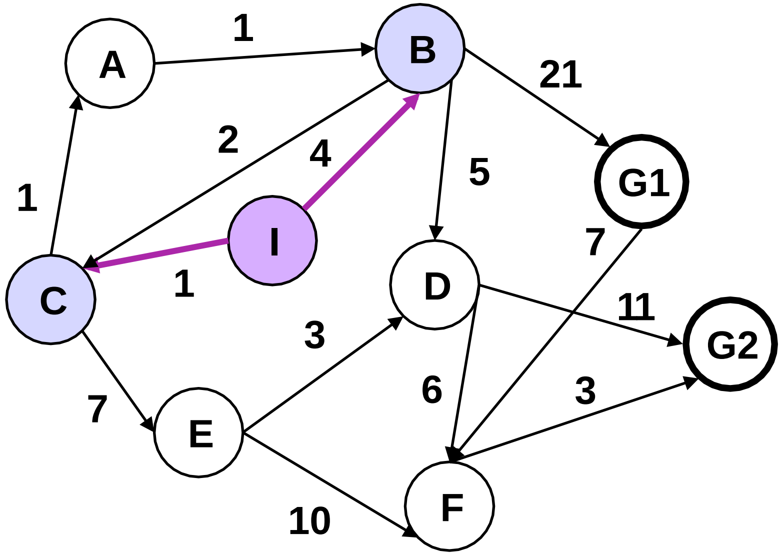
Search Tree



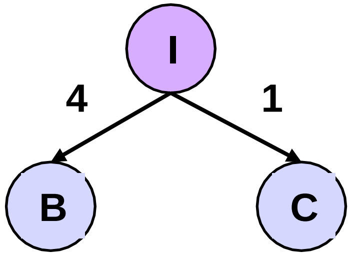
opened-list (stack): (I)

Example: Depth-first search

State Space



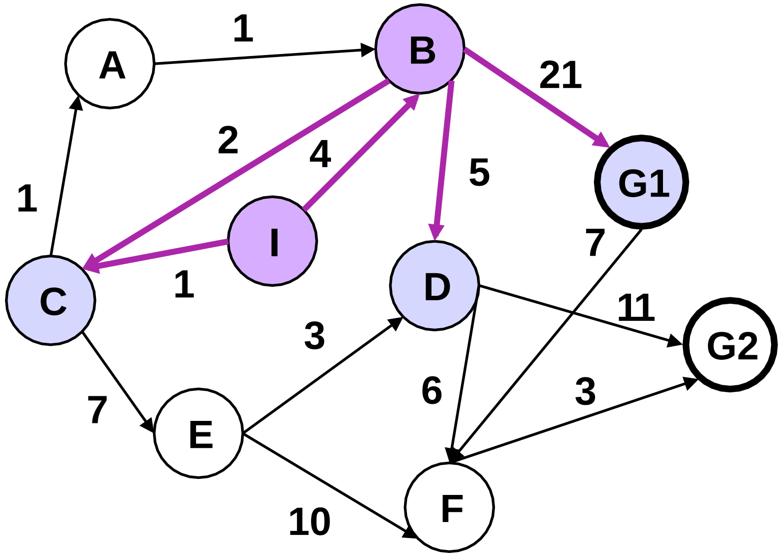
Search Tree



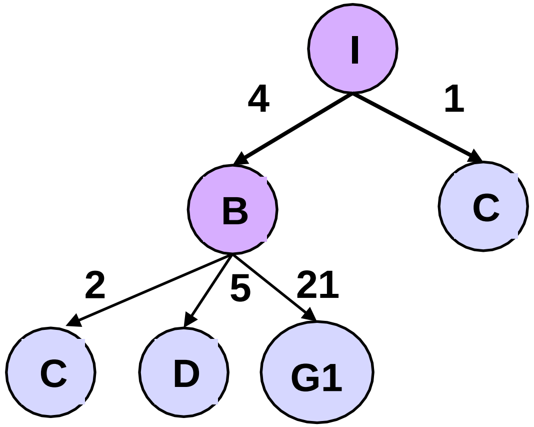
opened-list (stack): (B C)

Example: Depth-first search

State Space



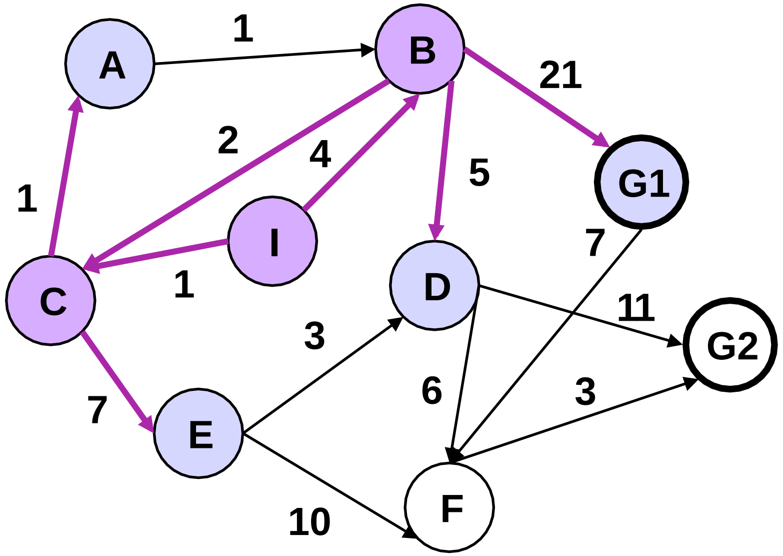
Search Tree



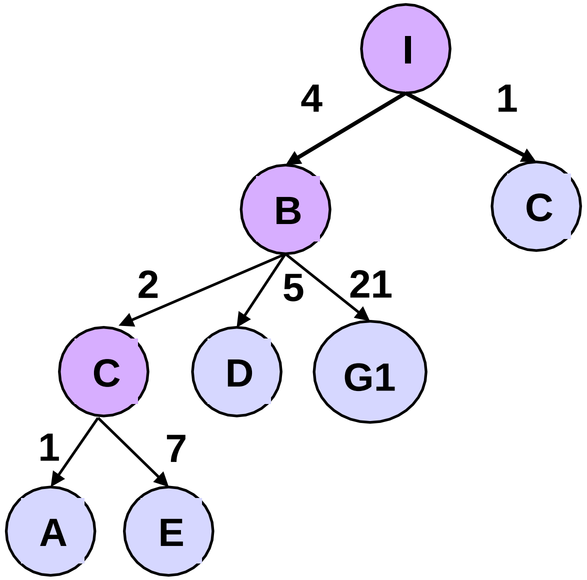
opened-list (stack): (C D G1 C)

Example: Depth-first search

State Space



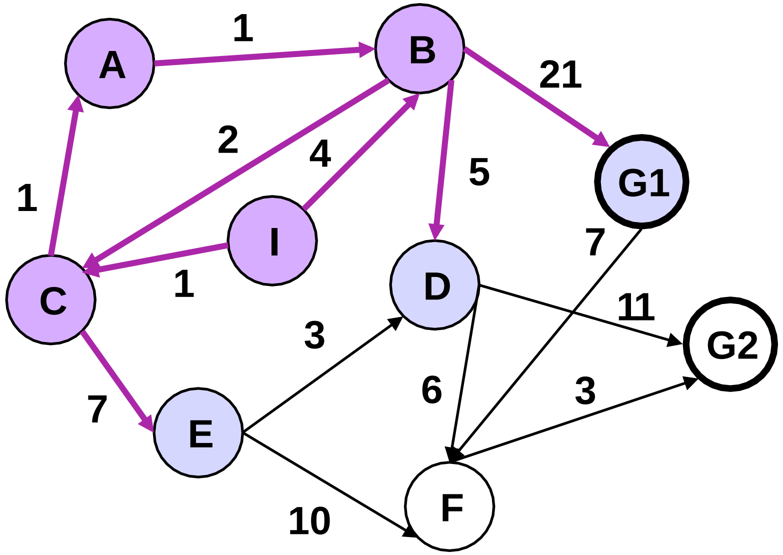
Search Tree



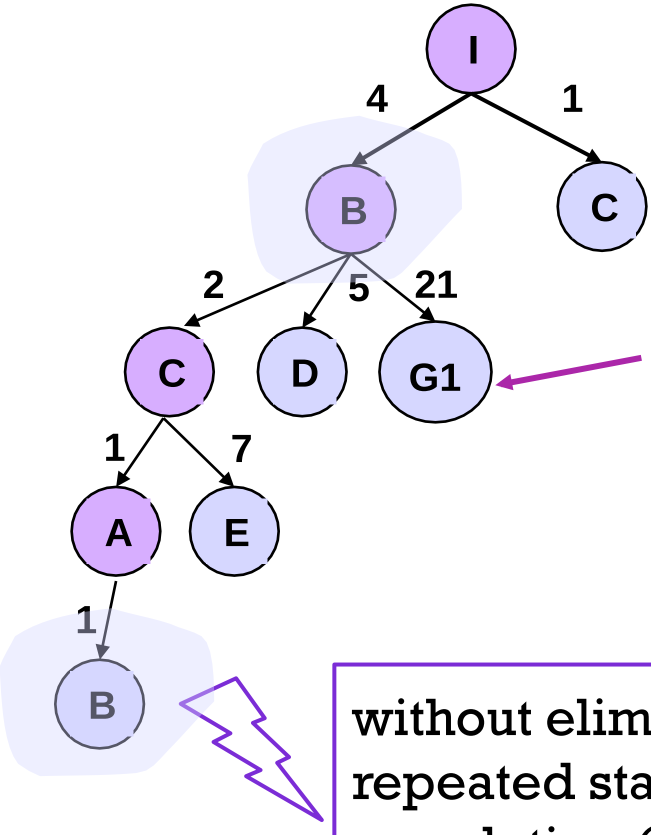
opened-list (stack): (A E D G1 C)

Example: Depth-first search

State Space



Search Tree



Does not get to check that it is a target

without elimination of repeated states it finds no solution (explores a branch of infinite length)

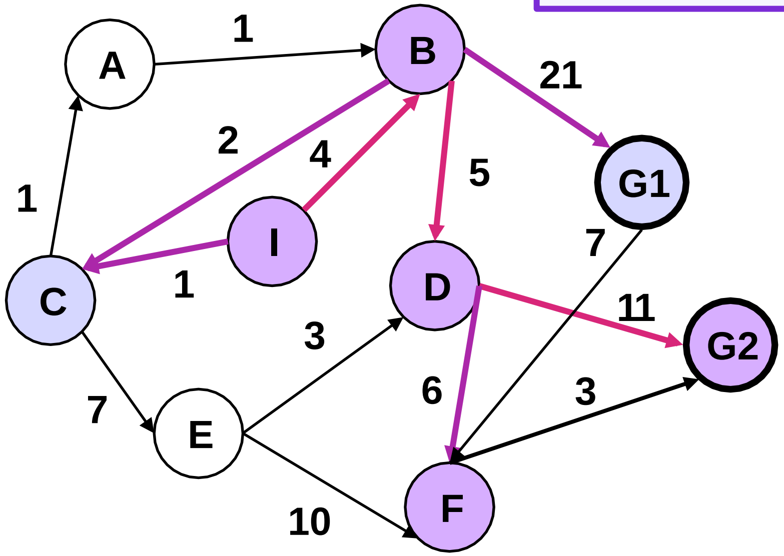
opened-list (stack): (B E D G1 C)

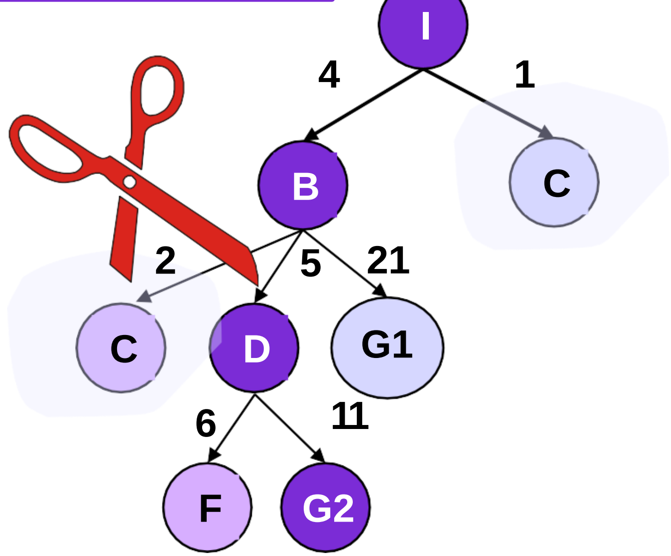
Expanded nodes: closed-list (stack): (A C B I)

Example: Depth-first search + elimination of repeated states with *opened-list*

State Space **Search Tree**

Avoid expanding nodes that are in the *opened-list*





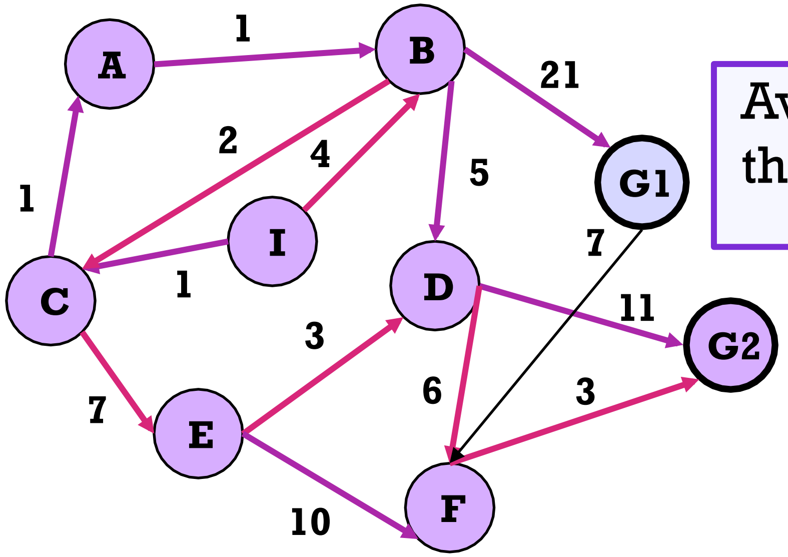
Expanded Nodes (by expansion order): I B D F G2

Generated Nodes (by generation order): I B C D G1 F G2

Path to solution: I B D G2 Cost: 4+5+11 = 20

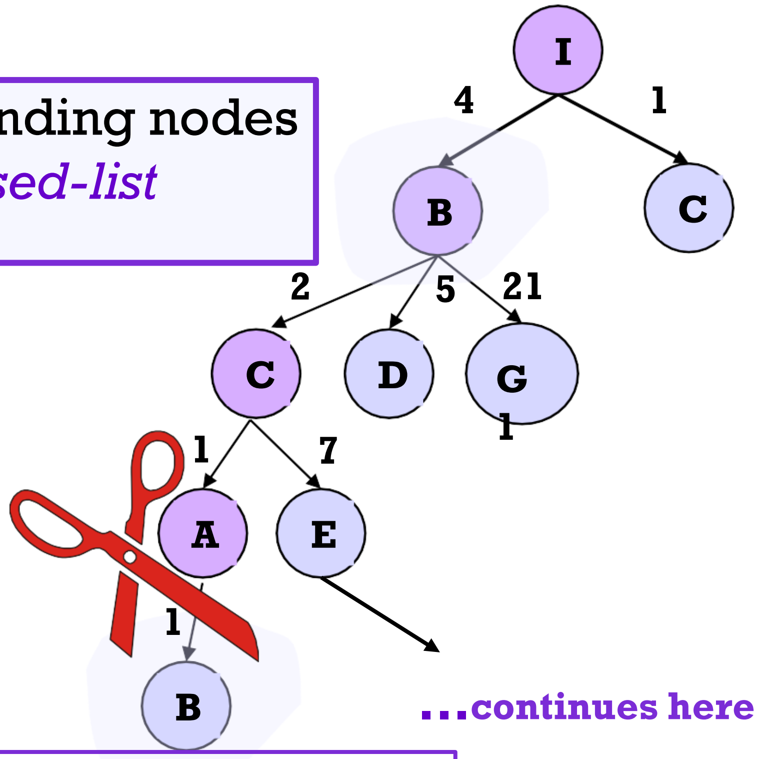
Example: Depth-first search + elimination of repeated states with *closed-list* [preferable]

State Space



Search Tree

Avoid expanding nodes that are *closed-list*



Expanded Nodes (by expansion order): I B C A E D F G2

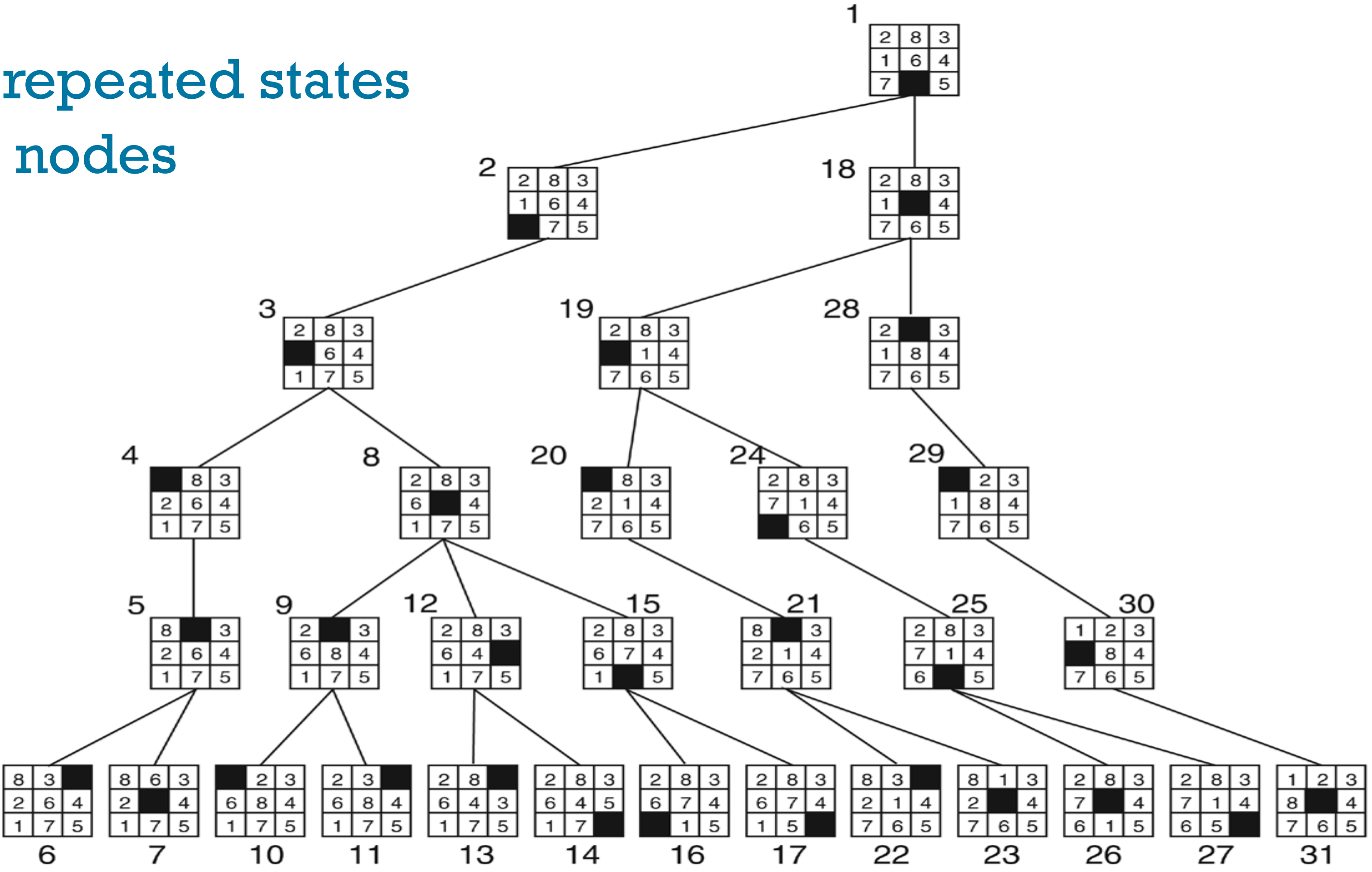
Path to solution: I B C E D F G2 Cost: 4+2+7+3+6+3 = 25

Depth-limited search

- ❑Search in depth with depth limited L
  - ❑Implementation: Same as depth-first search, assuming nodes with depth equal to *L* have no successors.
  - ❑Paths whose depth is greater than *L* are discarded.
  - ❑Avoid descending indefinitely in the search tree.
- ❑Properties:
  - ❑Complete only if L $\geq$ d (d depth of shallowest solution)
  - ❑If d is unknown, how to choose L?
    - ❑In general it is not known a priori
    - ❑Use domain-specific knowledge.
      - ❑Eg. There are problems in which the diameter of the state space is known. Said diameter is the maximum number of steps required to reach any state from any other state. Therefore, we can use L = diameter, which guarantees to find a solution, if exists
  - ❑Not optimal: There is no guarantee that if you find a solution, it will be the least costly.
  - ❑Temporal complexity: $O(b^L)$ [exponential in L]
  - ❑Spatial complexity: $O(b*L)$ [Linear in L] (good news)

Depth-limited search

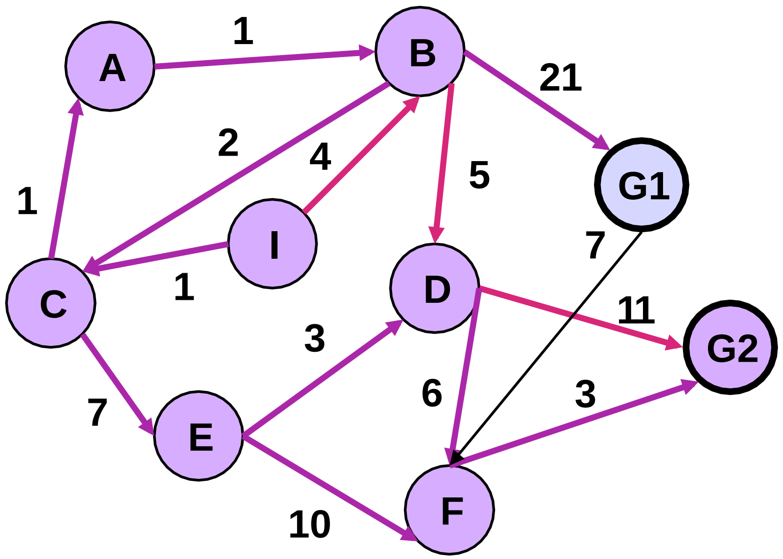
- ❑Limit L = 5
  - ❑Elimination of repeated states
  - ❑Just expanded nodes



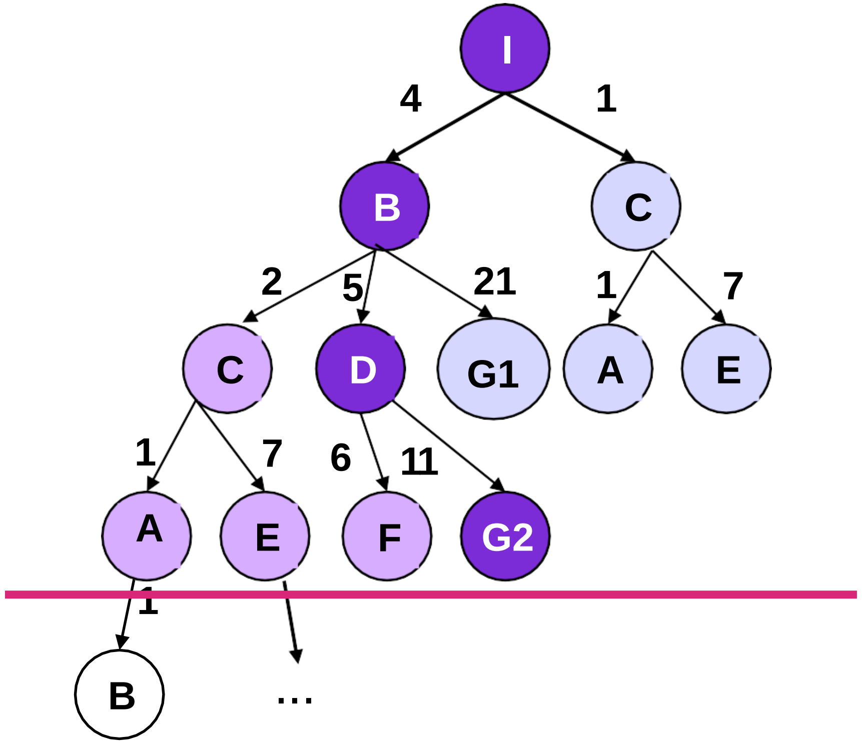
Target

Example: Depth-limited search with L =3

State Space



Search Tree



Expanded Nodes (by expansión order) : I B C A E D F G2

Path to solution : I B D G2 Cost: 4+5+11 = 20

Backtracking search

- ❑ Variant of limited depth search with **an efficient use of memory**

- ❑ Only one successor is generated in each expansion. In each partially expanded node it is memorized which is the next successor to be generated.

  - ❑ O(m) states

- ❑ Generate successor by modifying current state (instead of copy + modify). It must be able to undo modifications when the algorithm goes backwards to generate the following successors.

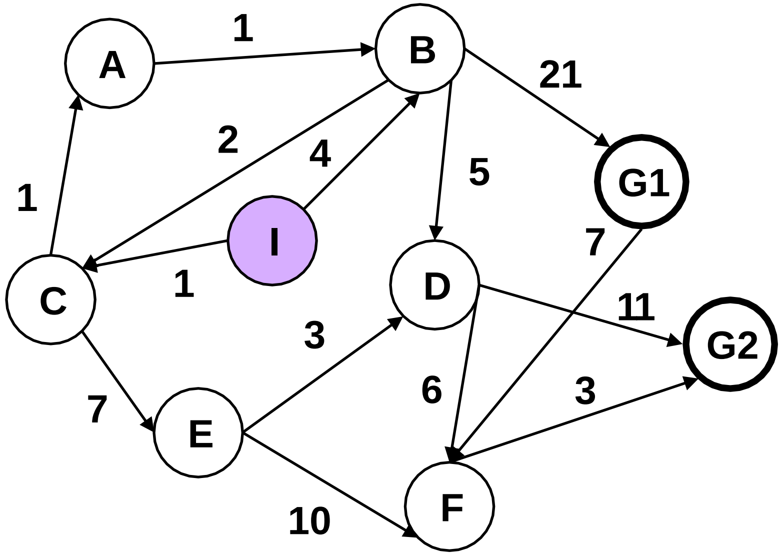
  - ❑ only one state + O(m) actions

Iterative deepening search

- Iterative application of the limited depth search algorithm, with increasing depth limit (L = 0, 1, 2, ...)
  - Avoid having to determine the depth limit
- Combines the advantages of depth-first and breath-first algorithms.
- This is often the preferred uninformed method when the state space is large and the depth of the solution is unknown.
- Properties:
  - **Complete**: guarantees to find a solution (the shallowest of the possible ones)
  - Optimal: only if the path cost is a non-decreasing function of the node depth (the shortest path may not be optimal)
  - Time: $O(b^d)$(May be better than breath-first)
    - Generated nodes $b^d * (1\ time) + \ldots + b^2 * (d-2) + b^1 * (d-1) + b^0$ (d times)
    - The nodes that are generated repeatedly are those that are at lower depth levels, but we avoid generating nodes at depth (d + 1), which, in the worst case, are b^{d+1} operations.
  - Space: $O(b*d)$ (as depth-first search algorithm)

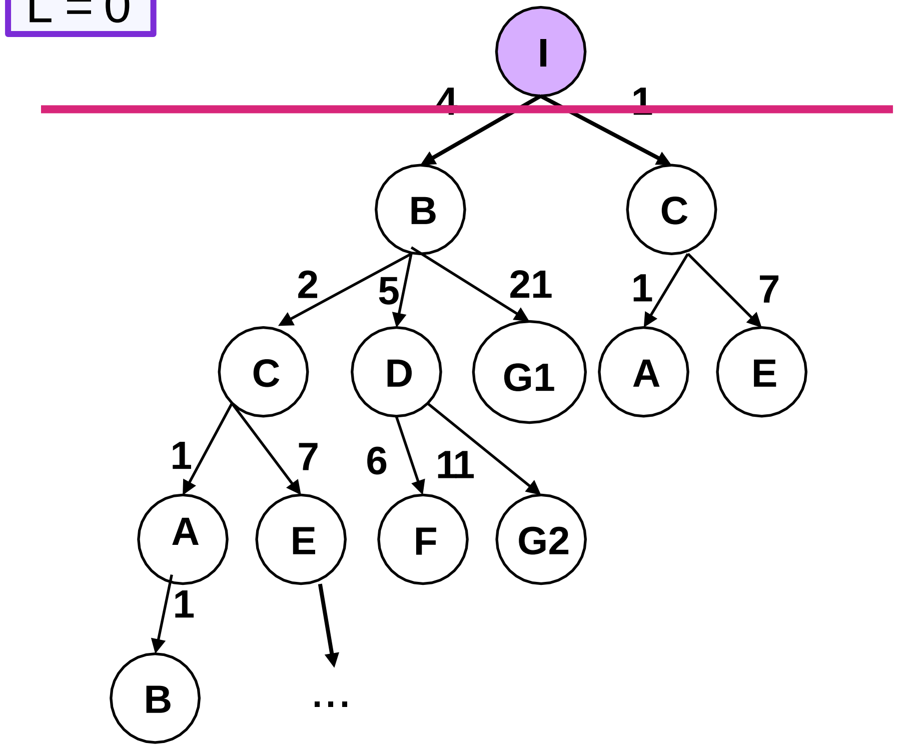
Example: Iterative deepening search

State Space



Search Tree

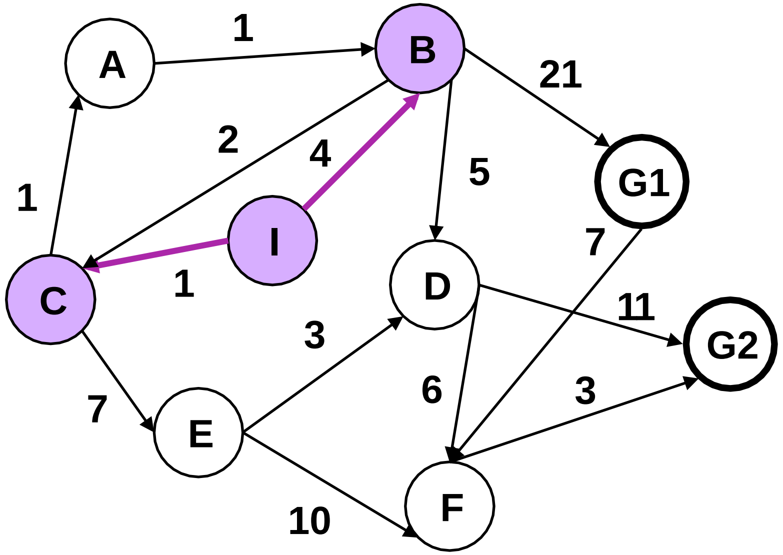
L = 0



Expanded nodes: I

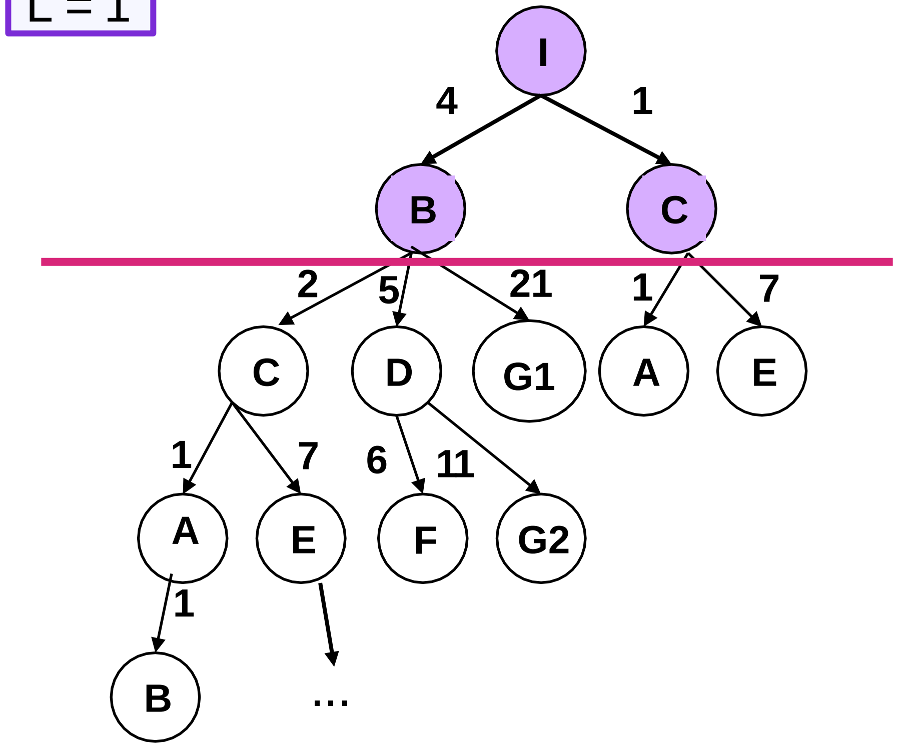
Example: Iterative deepening search

State Space



Search Tree

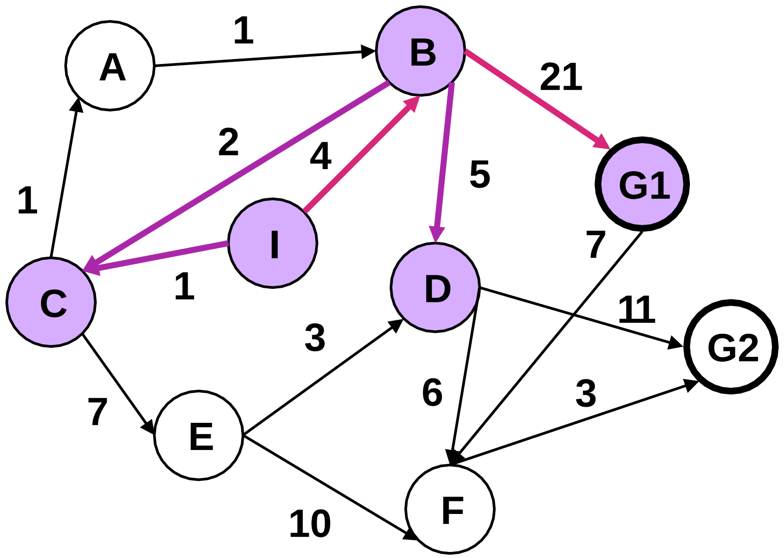
L = 1



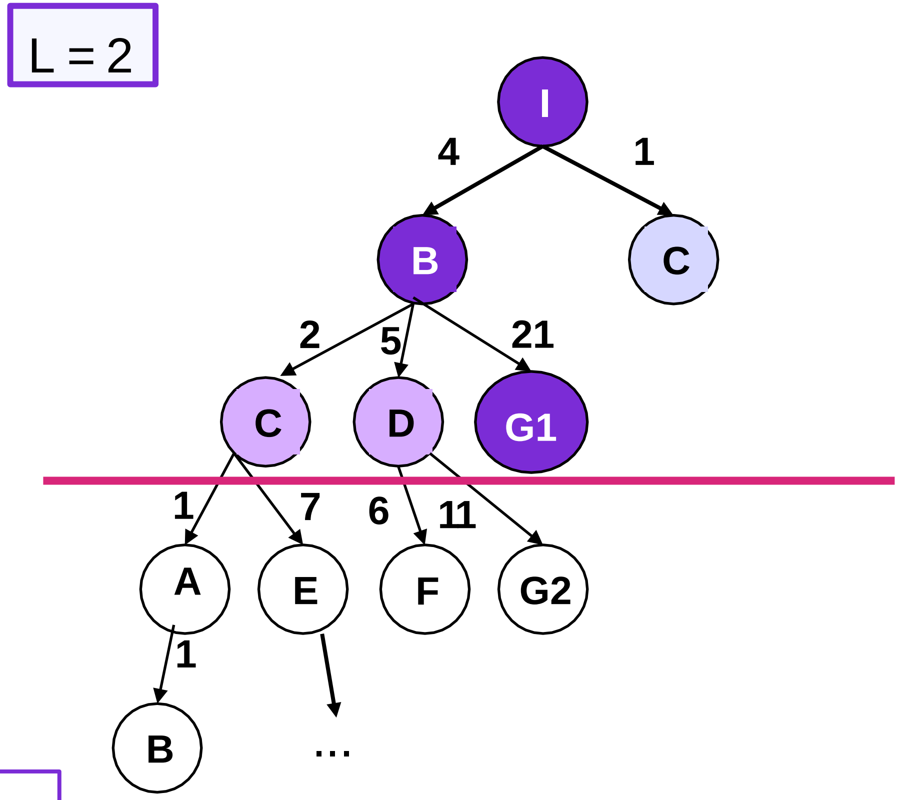
Expanded nodes: I B C

Example: Iterative deepening search

State Space



Search Tree



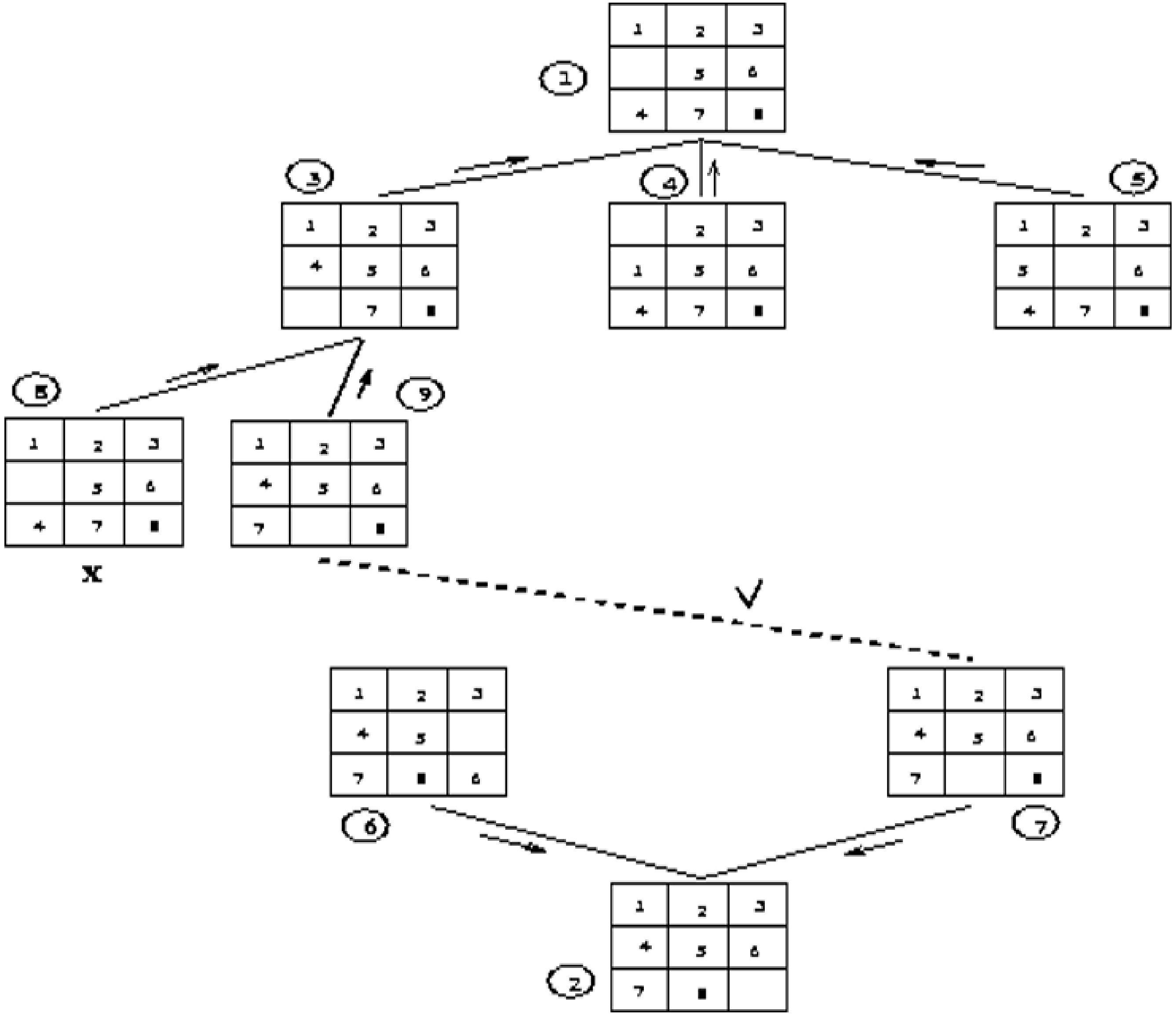
Expanded nodes: I B C I B C D G1

Path to solution: I B G1 Cost: 4+21 = 25

Bidirectional search

- Executes two simultaneous searches: one forward from the initial state and the other backward from the target state, stopping when the two searches are in a state
  - Motivation: $b^{d/2} + b^{d/2}$ is much less than b^d
  - Necessary
    - Know the target state. If there are multiple states, or these cannot be listed (for example in the N queens problem) this can be problematic
    - Being able to obtain the predecessors of a state in an efficient way (invertible operators.)
- Properties:
  - Optimal and complete if the searches are in width, the cost of the path is a non-decreasing function of the depth of the node.
  - Time: $O(2^*b^{d/2}) = O(b^{d/2})$
    - If the match check can be done in constant time (for example, using a hash table)
  - Space: $O(b^{d/2})$ (its greatest weakness)
    - At least the nodes of one of the two parts should be kept in memory for comparison (a hash table is usually used to store them)

Bidirectional search



Summary of uninformed methods

Blind Search	Complete	Optimal	Time Efficiency (worst case)	Space Efficiency (worst case)
Breadth-first	yes	yes cost ∝ depth	$O(b^{d})$	$O(b^{d})$
Uniform Cost	If there are no paths of finite cost and infinite length	yes operators cost > 0	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first	No	No	$O(b^{m})$	$O(b*m)$
Limited depth	yes $L \geq d$	No	$O(b^{L})$	$O(b*L)$
Iterative deepening	yes	yes cost ∝ depth	$O(b^{d})$	$O(b*d)$
Bidirectional Breadth-search	yes (width)	yes cost ∝ depth	$O(b^{d/2})$	$O(b^{d/2})$

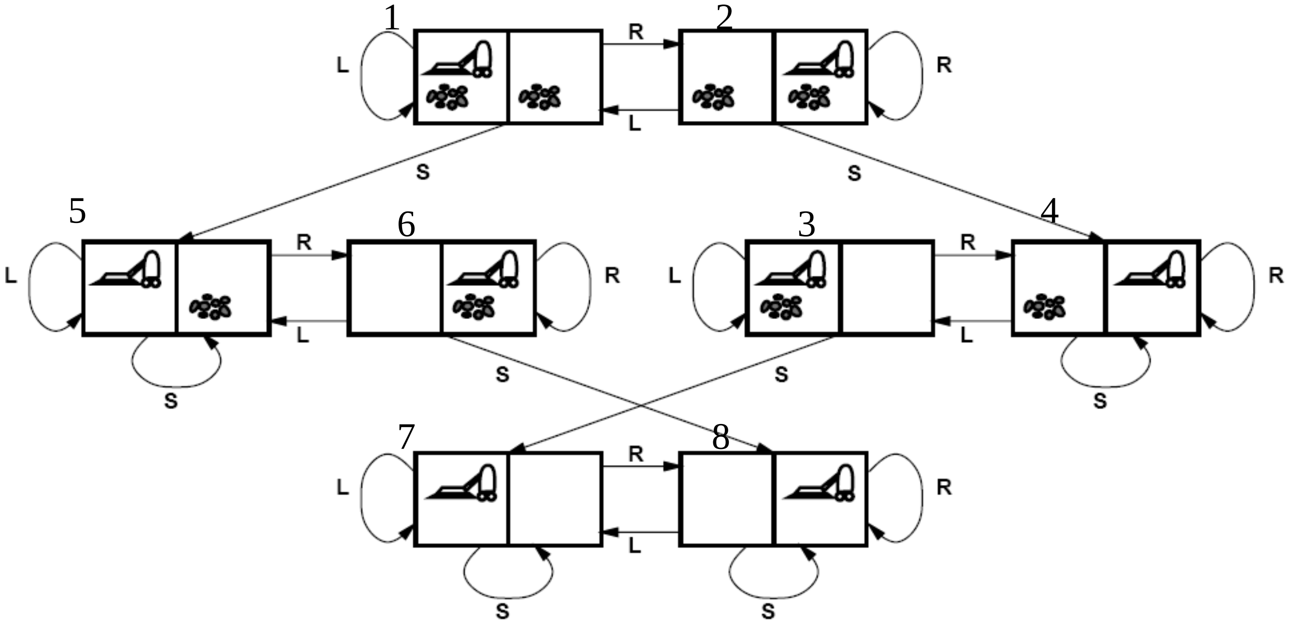
Branching factor (***b***): maximum number of successors of any node.
Shallowest target node depth(***d***)
Search tree maximum depth(***m***)
Depth limit(***L***)
Minimum cost of an action(ε)
Cost path optimal solution(C*)

Problem types: with partial information

- ❑ So far we have analyzed search problems where
  - ❑ The State space
    - ❑ Is **fully observable** (perceptions uniquely determine the state of the environment)
    - ❑ static (it only changes when we execute an action).
  - ❑ Deterministic actions: We know the result of any sequence of actions.
- ❑ What happens when we don't have complete information to uniquely characterize a state?
  - ❑ **Conformant search problems**: Not observable (sensorless problems). The agent does not have complete information about the state in which it is.
  - ❑ **Contingent problems**
    - ❑ Non-deterministic, and possibly partially observable. Agent insights provide new information after each action.
    - ❑ If the uncertainty is caused by the actions of another agent with objectives opposite to those of the agent conducting the search => search with adversaries
  - ❑ **Exploration problems**: Unknown state space. It is necessary to interact with the environment (sensors and effectors) to determine what state it is in
  - ❑ **Contingent search problems with exploration**
    - ❑ Standard search algorithms are generally not suitable for solving these types of problems.
    - ❑ The agent can collect information through his sensors after acting.
    - ❑ Proposition: alternate search and execution (actions + sensors).

Vacuum cleaner world

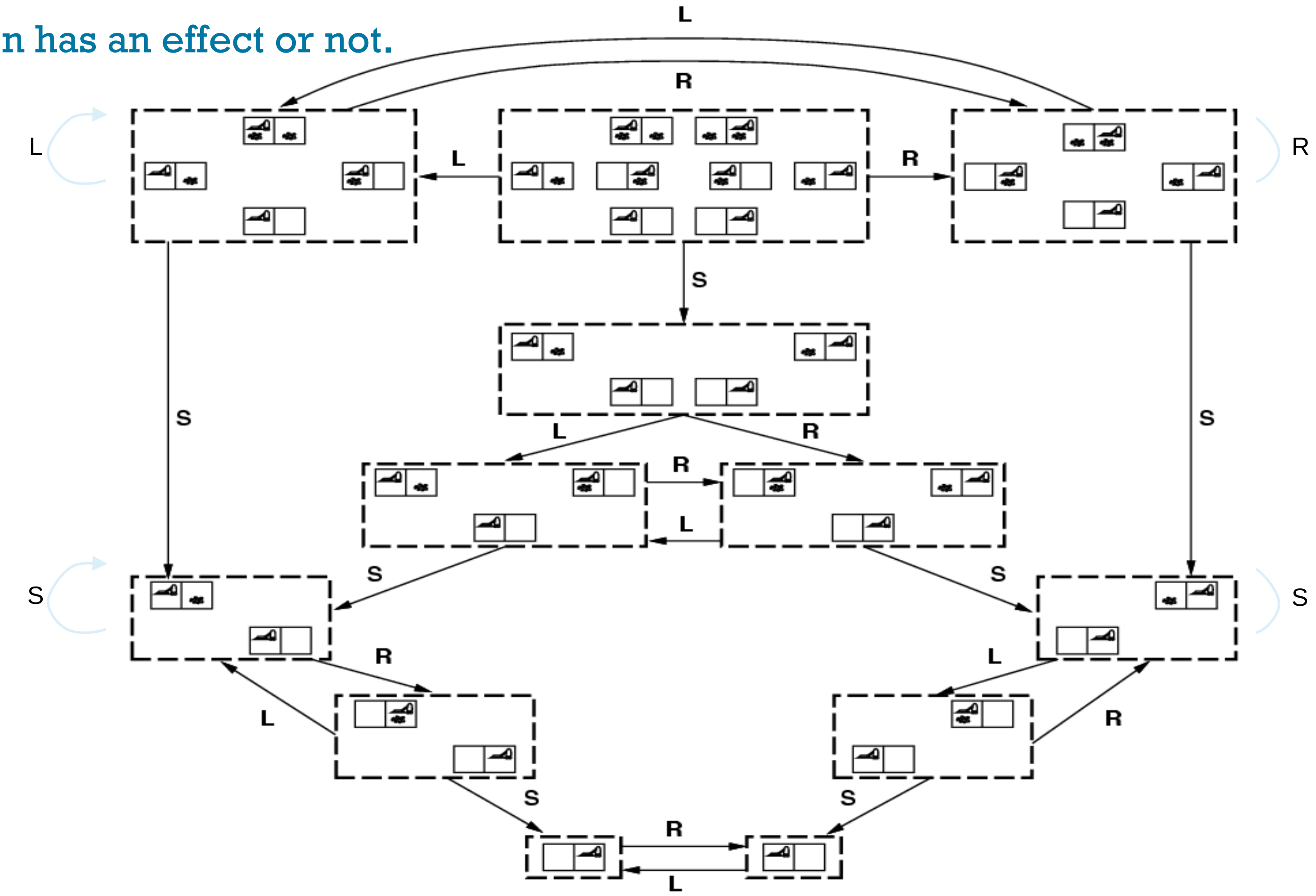
- Physical states:
  - Agent: Vacuum cleaner in one of the two adjacent rooms.
  - Each room can be dirty / clean
- Actions:
  - L / R: move to the left / right
  - S: Suck dirt
  - NoOp: Do nothing
- Target: Clean world (state 7 or 8)



Conformant problem (without sensors)

If the agent **does not have sensors** then

- ❑ The search states represent **belief states: subsets** of the possible physical states of the world.
- ❑ Through a **sequence of actions** one may be able to **force** the world into a goal state.
- ❑ Without sensors we do not know if the action has an effect or not.



Contingency search problem

- The solution can not be formulated in general as a fixed sequence of actions.
- **Contingency plan**: The solution can be given as a tree, where each branch is chosen depending on the perceptions

Example:

- Single state, we start at state #5. **Solution?**

  [*Right*, *Suck*]

- Conformant, start at {1, 2, 3, 4, 5, 6, 7, 8}

  For example, the *Right* action leads to the state {2, 4, 6, 8}. **Solution?**

  [*Right*, *Suck*, *Left*, *Suck*]

- Contingency, start at state #5.

  The vacuum cleaner has a defect and by using it, the floor can become dirty.

  Sensors: location, dirt. **Solution?**

  [*Right*, **if** *dirty-floor* **then** *Suck*]