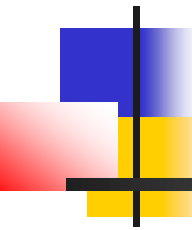
Part 2.1: Problem solving - Searching



- Part 2.1 General introduction to searching
- Part 2.2 Uninformed (Blind)/ Informed (Heuristic) search
- Part 2.3 Game playing search

Workers are always Searching

Brief info on Ants, http://www.winnipeg-bugline.com/ants.html,
http://www.boston.com/globe/magazine/2002/0623/frontiers.htm

Problem solving techniques in A.I

- *Broad* Approaches
  - using search techniques
    - e.g. in Games
  - modeling
    - using Knowledge Base Systems (KBS)
    - using Machine Learning techniques e.g. Artificial Neural Networks, Decision Trees, Case-base reasoning, Genetic algorithms, ..

Searching as a problem solving technique

- **Searching** is the process of looking for the solution of a problem through a set of possibilities (state space).
- **Search** conditions include:
  - Current state -where one is;
  - Goal state – the solution reached; check whether it has been reached;
  - Cost of obtaining the solution.
- The **solution** is a path from the current state to the goal state.

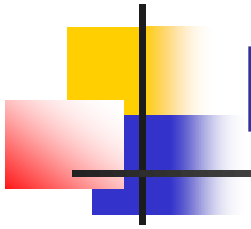
Searching as a problem solving technique

Process of Searching

- Searching proceeds as follows:
  - Check the current state;
  - Execute allowable actions to move to the next state;
  - Check if the new state is the solution state; if it is not, then the new state becomes the current state and the process is repeated until a solution is found or the state space is exhausted.

Search problem

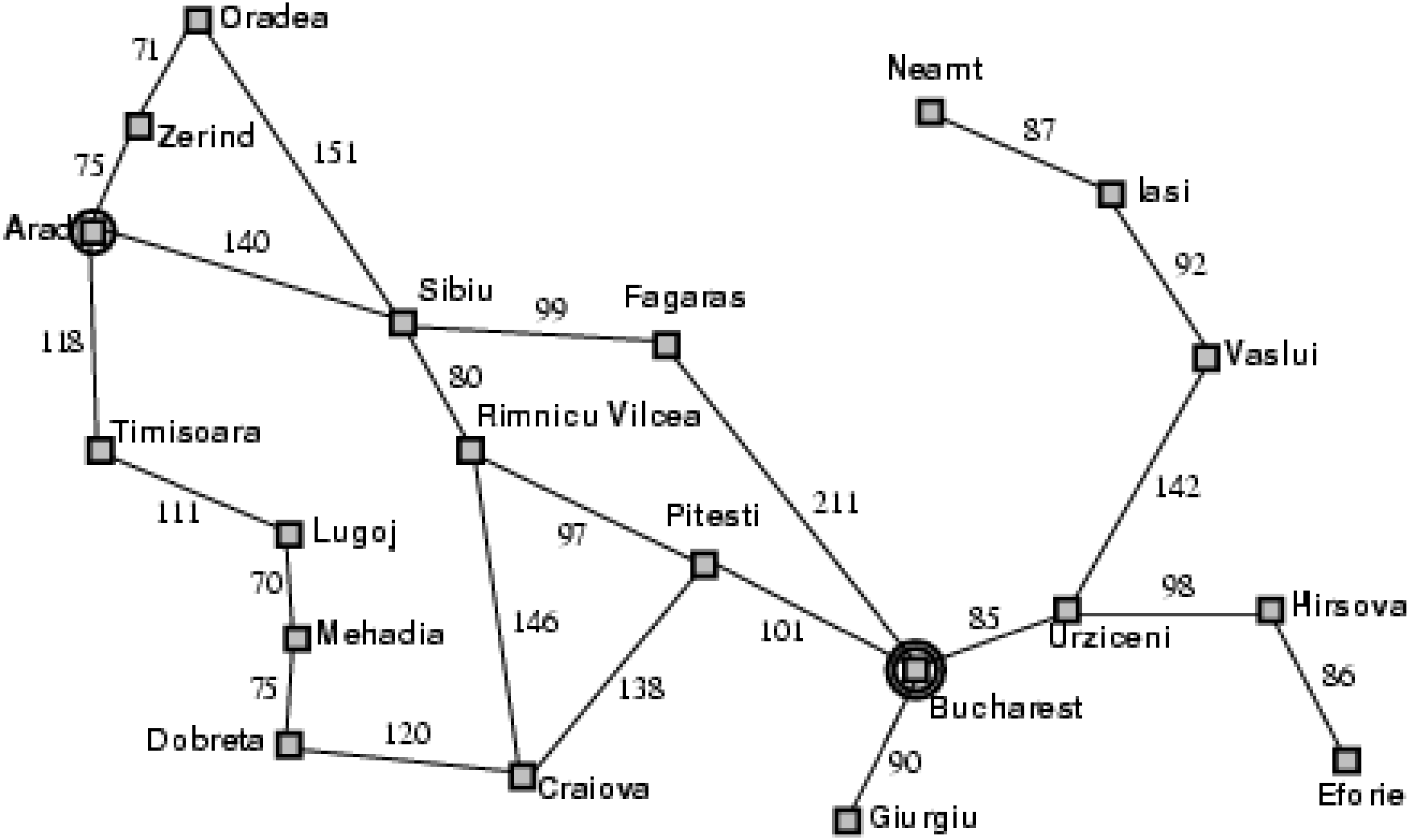
- The **search problem** consists of finding a solution **plan**, which is a path from the current state to the goal state.
- **Representing search problems**
  - A search problem is represented using a directed graph. The states are represented as nodes while the allowed steps or actions are represented as arcs.
- **A search problem is defined by specifying:**
  - State space;
  - Start node;
  - Goal condition, and a test to check whether the goal condition is met;
  - Rules giving how to change states.
  - Path cost

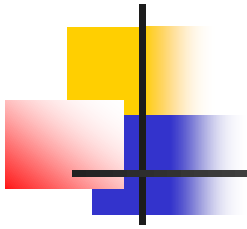


Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



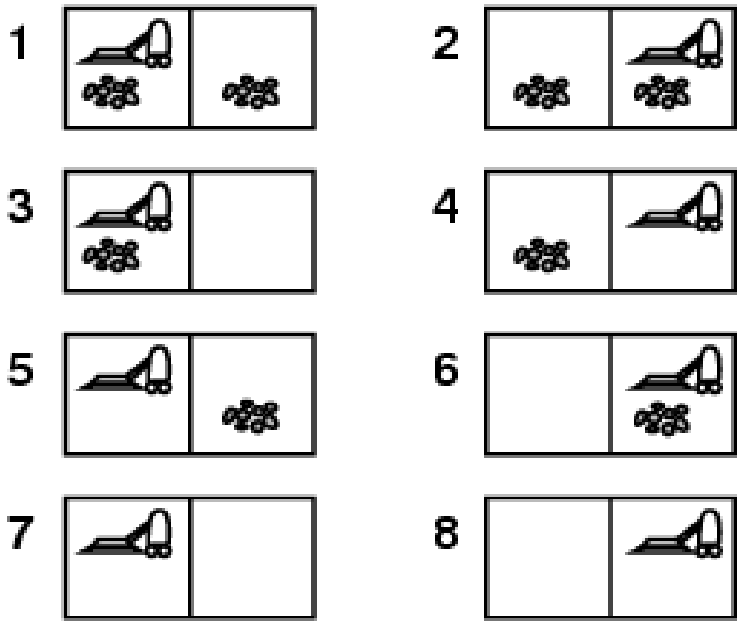


Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave} search, execution
- Unknown state space → exploration problem

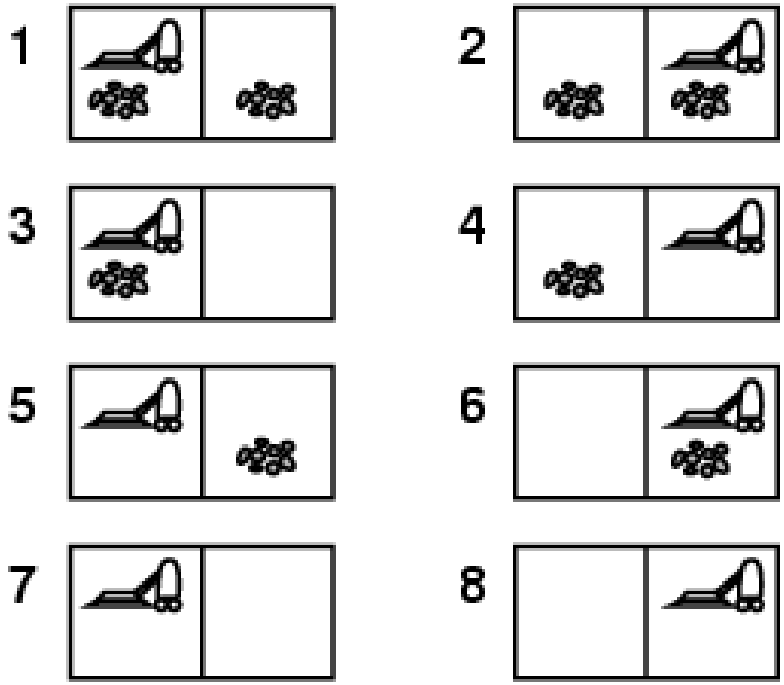
Example: vacuum world

- Single-state, start in #5.
  Solution?



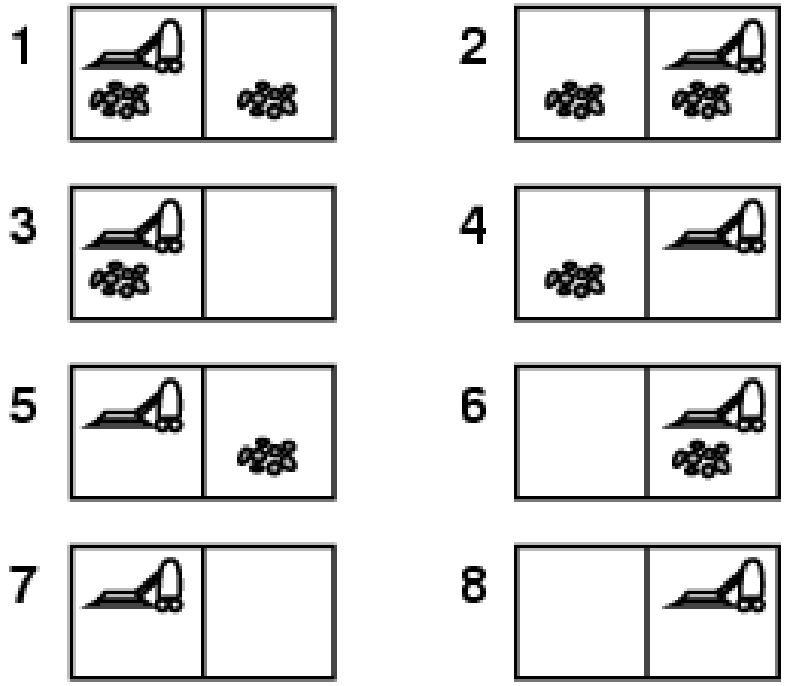
Example: vacuum world

- Single-state, start in #5. Solution? *[Right, Suck]*
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution?



Example: vacuum world

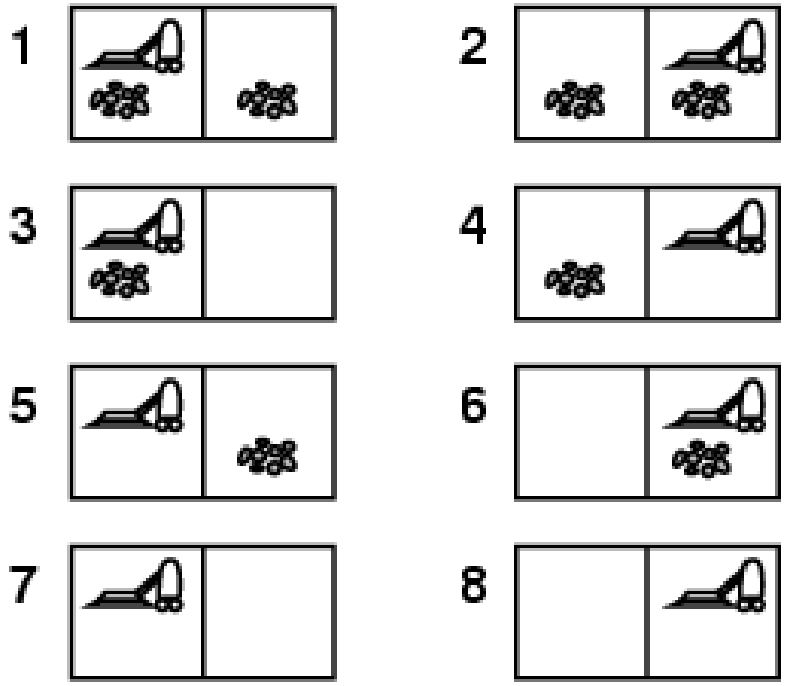
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution? *[Right,Suck,Left,Suck]*



- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7 Solution?

Example: vacuum world

- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution? *[Right,Suck,Left,Suck]*



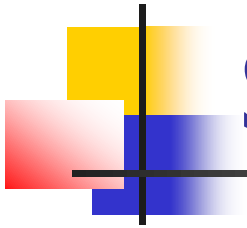
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7 Solution? *[Right, **if** dirt **then** Suck]*

Single-state problem formulation

A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state



Selecting a state space

- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Problem Definition - Example, 8 puzzle

5	4	
6	1	8
7	3	2

Initial State

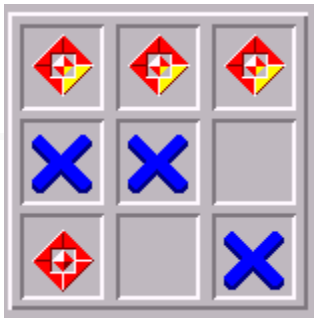
1	4	7
2	5	8
3	6	

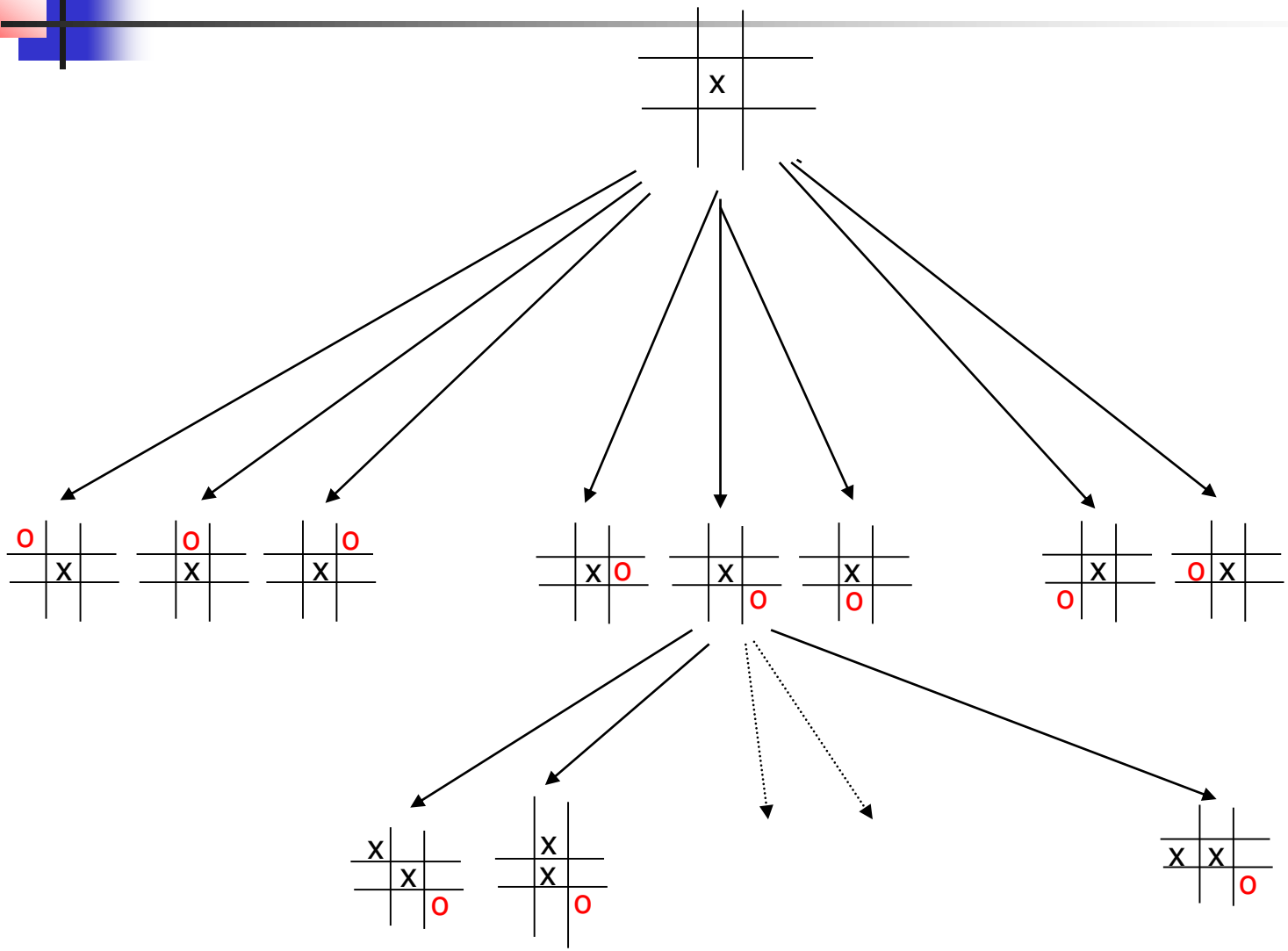
Goal State

Problem Definition - Example, 8 puzzle

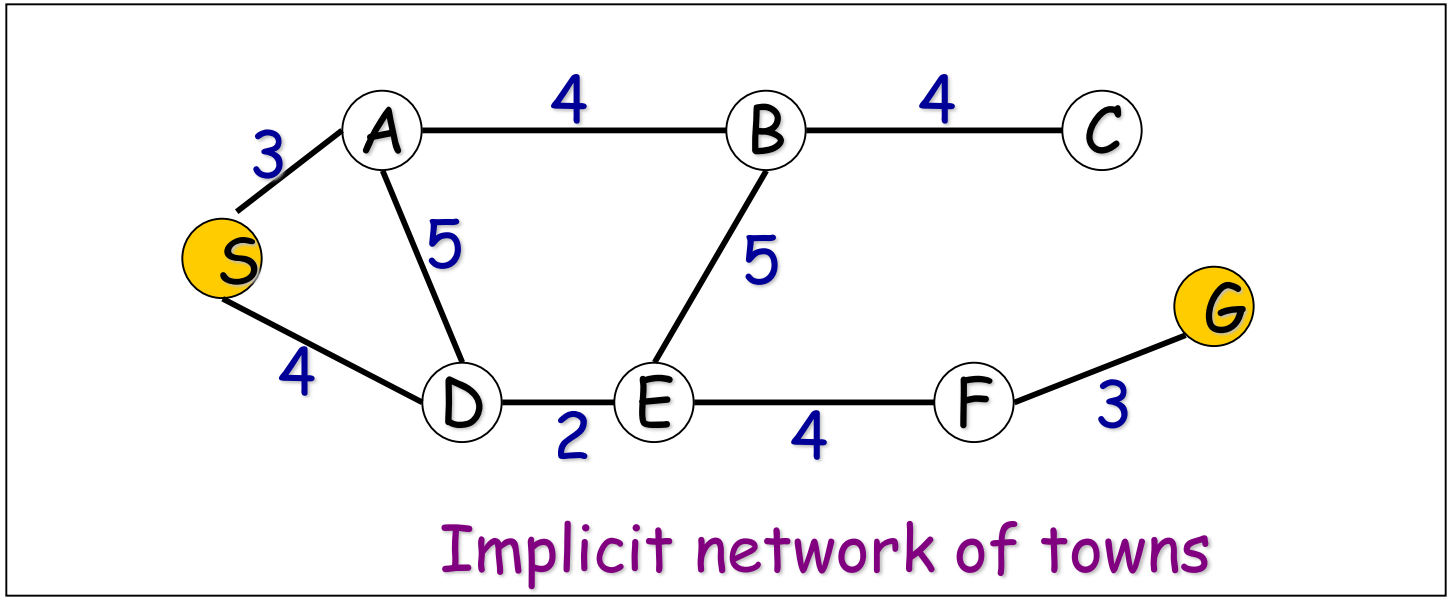
- States
  - A description of each of the eight tiles in each location that it can occupy. It is also useful to include the blank
- Operators/Action
  - The blank moves left, right, up or down
- Goal Test
  - The current state matches a certain state (e.g. one of the ones shown on previous slide)
- Path Cost
  - Each move of the blank costs 1

Problem Definition - Example, tic-tac-toe



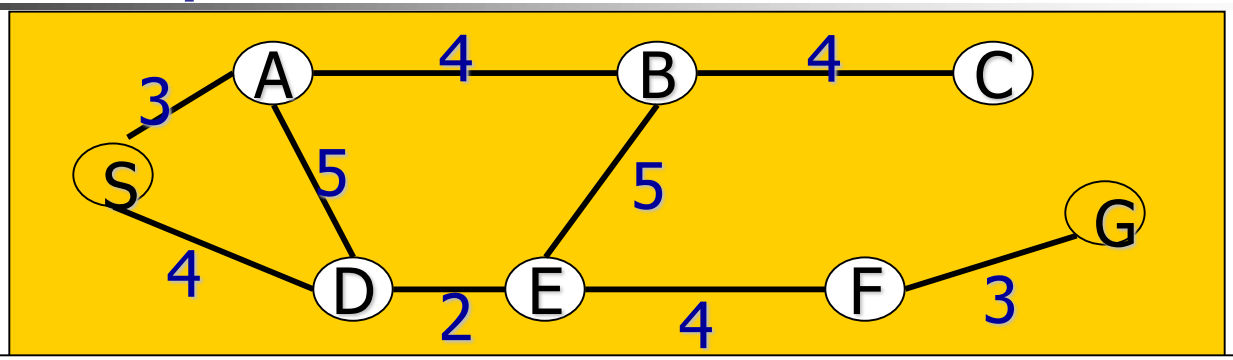


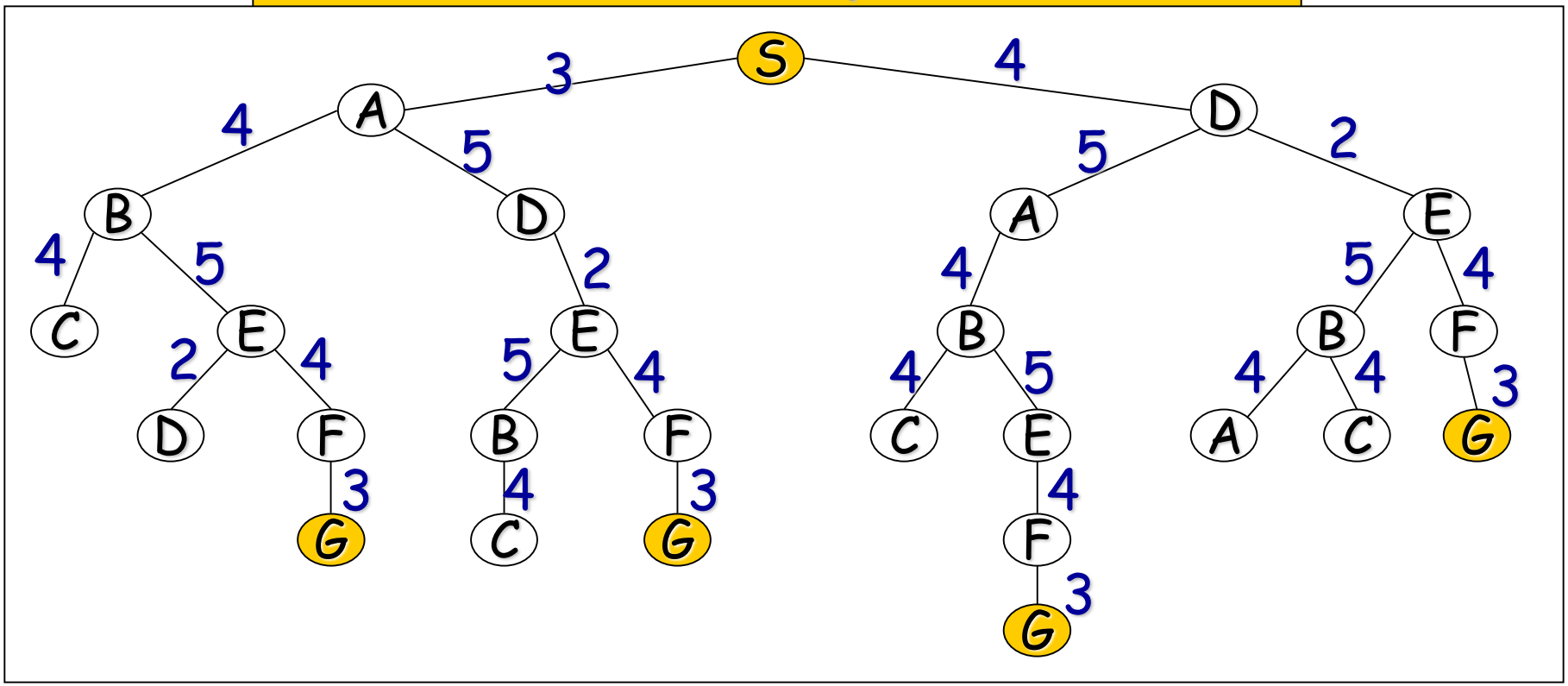
Tree/Path example:



- Two possible tasks:
  - 1. FIND a (the) path. = computational cost
  - 2. TRAVERSE the path. = travel cost
- 2. relates to finding optimal paths

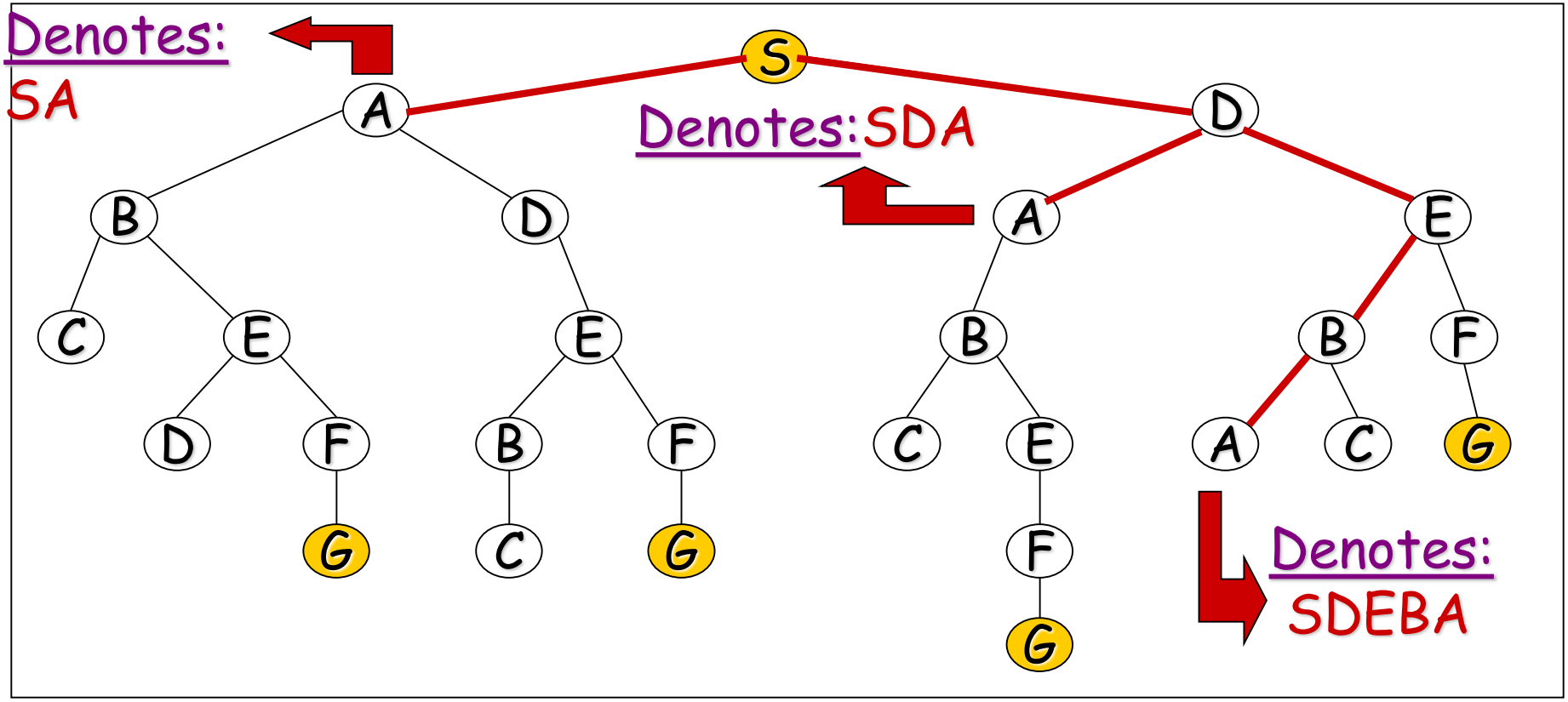
The associated loop-free tree of partial paths





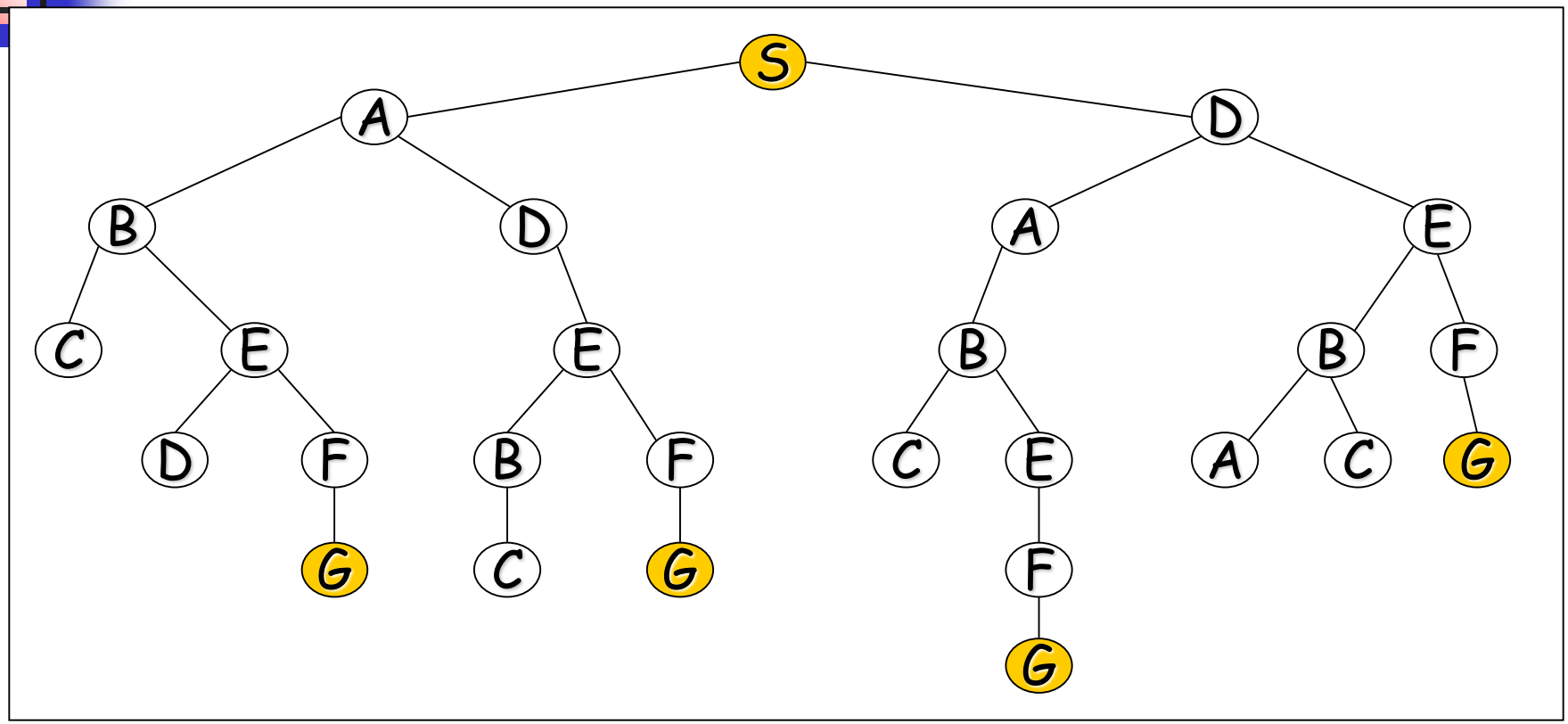
Paths:

- We are not interested in optimal paths here, so we can drop the costs.



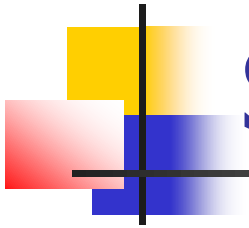
- Nodes do not denote themselves, but denote the partial path from the root to themselves!!

Terminology:

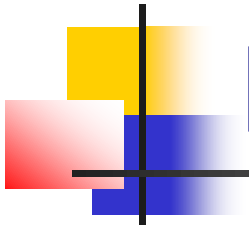


- Node, link (or edge), branch, arc
- Parent, child, ancestor, descendant
- Root node, goal node
- Expand / Open node / Closed node / Branching factor

Using a Tree – The Obvious Solution?

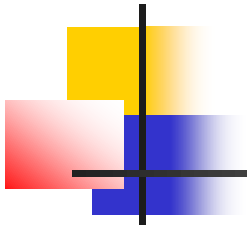


- **But**
  - It can be wasteful on space
  - It can be difficult to implement, particularly if there are varying number of children (as in tic-tac-toe)
  - It is not always obvious which node to expand next. We may have to search the tree looking for the best leaf node (sometimes called the fringe or frontier nodes). This can obviously be computationally expensive



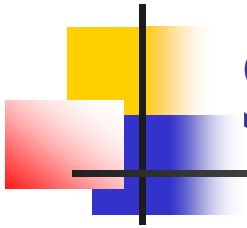
How good is a solution?

- Does our search method actually find a solution?
- Is it a good solution?
  - Path Cost
  - Search Cost (Time and Memory)
- Does it find the optimal solution?
  - But what is optimal?



Evaluating a search

- Completeness
  - Is the strategy guaranteed to find a solution?
- Time Complexity
  - How long does it take to find a solution?
- Space Complexity
  - How much memory does it take to perform the search?
- Optimality
  - Does the strategy find the optimal solution where there are several solutions?



Search Trees

- Some issues:
  - Search trees grow very quickly
  - The size of the search tree is governed by the branching factor
  - Even this simple game tic-tac-toe has a complete search tree of 984,410 potential nodes
  - The search tree for chess has a branching factor of about 35