UNIVERSITÉ
JEAN MONNET
SAINT-ÉTIENNE

Introduction to AI

(Part 2 : AI problems and Search Algorithms)

Outline

1. Intro
2. AI Problem Representation
3. Problem solving: Uninformed Search
    1. Depth-first search, Breadth-first search, Uniform search, Iterative deepening…
    2. Problem decomposition (AND/OR tree)
4. Heuristic (informed) Search
    1. A*, AO*…
5. Game Playing

Why AI?

- **Cognitive Science**

As a way to understand how natural minds and mental phenomena work

  - e.g., visual perception, memory, learning, language, etc.

- **Philosophy**

As a way to explore some basic and interesting (and important) philosophical questions

  - e.g., what is consciousness, the mind-body problem, etc.

- **Engineering**

To get machines to do a wider variety of useful things

  - e.g., understand spoken natural language, recognize individual people in visual scenes, find the best travel plan for your vacation, etc.

Weak Vs Strong AI

- **Weak AI**:
  Machines can be made to behave as if they were intelligent

- **Strong AI**:
  Machines can have consciousness

Subject of fierce debate, usually among philosophers and nay-sayers, not so much among AI researchers!

E.g., http://groups.yahoo.com/group/webir/message/1002

AI Characterizations

Discipline that systematizes and automates intellectual tasks to create machines that

Act like humans	Act rationally
Think like humans	Think rationally

Act Like Humans

- AI is the art of creating machines that perform functions that require intelligence when performed by humans
- Methodology: Take an intellectual task at which people are better and make a computer do it
- Turing test

- Prove a theorem
- Play chess
- Plan a surgical operation
- Diagnose a disease
- Navigate in a building

Turing Test

- How it is done
  - Interrogator interacts with a computer and a person via a teletype
  - Computer passes the Turing test if interrogator cannot determine which is which

- Loebner contest
  - Modern version of Turing Test, held annually, with a $100,000 prize
  - Participants include a set of humans, a set of computers and a set of judges
    - Scoring: Rank from least human to most human
    - Highest median rank wins $2000
    - If better than a human, win $100,000 (Nobody yet…)

Chess

Garry Kasparov Vs Deep

Perspective on Chess

Pro: “Saying Deep Blue doesn’t really think about chess is like saying an airplane doesn't really fly because it doesn't flap its wings”

Drew McDermott

Cons: “Chess is the Drosophila of artificial intelligence. However, computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing Drosophila. We would have some science, but mainly we would have very fast fruit flies.”

John McCarthy

Think Like Humans

- How the computer performs functions does matter
- Comparison of the traces of the reasoning steps
- Cognitive science
  - testable theories of the workings of the human mind

•Connection with Psychology
•General Problem Solver (Newell and ifmon)
•Neural networks
•Reinforcement learning

But:
• Role of physical body, senses, and evolution in human intelligence?
• Do we want to duplicate human imperfections?

Think/Act Rationally

- Always make the best decision given what is available (knowledge, time, resources)
- Perfect knowledge, unlimited resources → logical reasoning
- Imperfect knowledge, limited resources → (limited) rationality

•Connection to economics, operational research, and control theory
•But ignores role of consciousness, emotions, fear of dying on intelligence

History of AI

“I personally think that AI is (was?) a rebellion against some form of establishment telling us "Computers cannot perform certain tasks requiring intelligence”

Jean-Claude Latombe

Bits of History

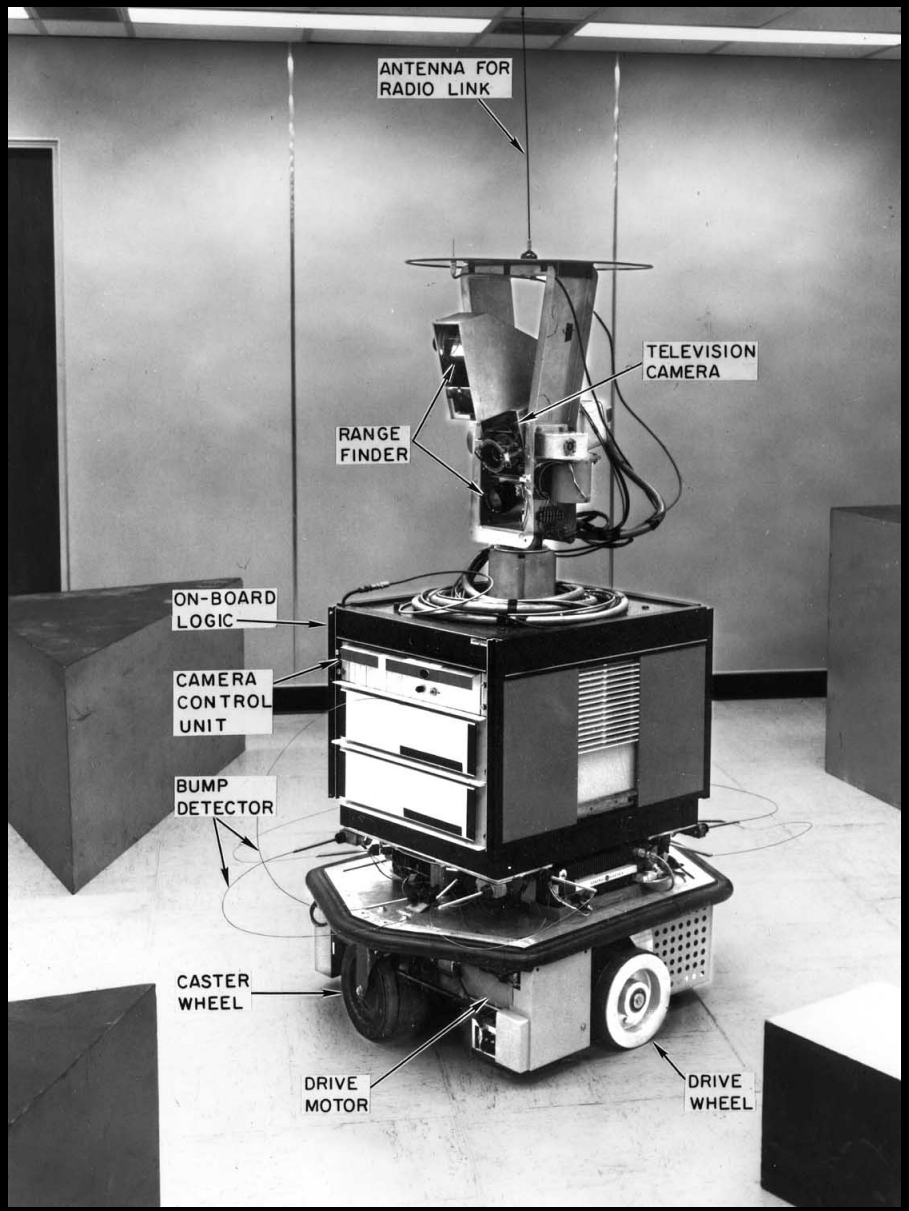
1956: The name “Artificial Intelligence” was coined by John McCarthy

(would “computational rationality” have been better?)

Early period (50’s to late 60’s): basic principles and generality

- General problem solving
- Theorem proving
- Games
- Formal calculus

Bits of History



- 1969-1971: Shakey the robot [Fikes, Hart, Nilsson]
- Logic-based planning (STRIPS)
- Inductive learning (PLANEX)
- Computer vision

Bits of History

Knowledge-is-Power period (late 60's to mid 80's):

- Focus on narrow tasks require expertise
- Encoding of expertise in rule form:

```
If:    the car has off-highway tires and
       4-wheel drive and
       high ground clearance
Then: the car can traverse difficult terrain (0.8)
```

- Knowledge engineering
- 5th generation computer project
- CYC system (Lenat)

Bits of History

AI becomes an industry (80's – present):

- Expert systems: Digital Equipment, Teknowledge, Intellicorp, Du Pont, oil industry, …
- Lisp machines: LMI, Symbolics, …
- Constraint programming: ILOG
- Robotics: Machine Intelligence Corporation, Adept, GMF (Fanuc), ABB, …
- Speech understanding
- Information Retrieval – Google, …

Where are we now? *(1/2)*

- SKICAT: a system for automatically classifying the terabytes of data from space telescopes and identifying interesting objects in the sky *(94% classification accuracy, exceeds human abilities)*
- Deep Blue: the first computer program to defeat champion Garry Kasparov
- Pegasus: a speech understanding program that is a travel agent
- Jupiter: a weather information
- HipNav: a robot hip-replacement surgeon
- Navlab: a Ford escort that steered itself from Washington DC to San Diego *98%* of the way on its own!

Where are we now? (2/2)

- Google news: autonomous AI system that assembles "live" newspaper
- DS1: a NASA spacecraft that did an autonomous flyby an asteroid.
- Credit card fraud detection and loan approval
- Search engines: www.citeseer.com, automatic classification and indexing of research papers.
- Proverb: solves NYT puzzles as well as the best humans.
- AlphaGo: March 2016 AlphaGo played South Korean professional Go player Lee Sedol, ranked 9-dan, one of the best Go players - win four to one.

Surprises in AI research *(1/2)*

Tasks difficult for humans have turned out to be "easy"

Chess

Checkers, Othello, Backgammon

Logistics planning

Airline scheduling

Fraud detection

Sorting mail

Proving theorems

Crossword puzzles

Surprises in AI research *(2/2)*

Tasks easy for humans have turned out to be hard

Speech recognition

Face recognition

Composing music/art

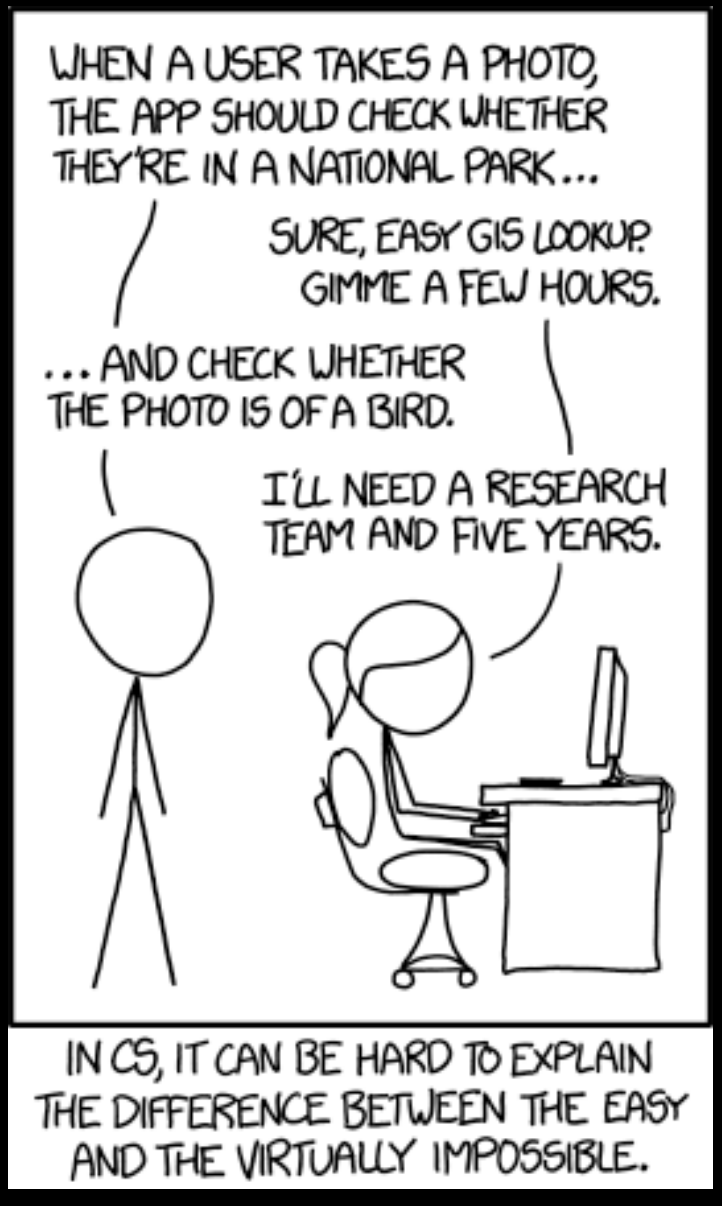
Autonomous navigation

Motor activities *(walking)*

Language understanding

Common sense reasoning

(eg, how many legs does a fish have?)



Why is AI Hard?

simple syntactic manipulation is not enough

Machine Translation:

- Big project in 1957 following Sputnik launch
- Translation of Russian documents
  - '*The spirit is willing but the flesh is weak*'
  - '*The vodka is strong but the meat is rotten*'

ELIZA: a "psychoterapist" program

It simulates a psychotherapist interacting with a patient

- Coded at MIT during 1964-1966 by Joel Weizenbaum
- First script was DOCTOR.
  - a simple collection of syntactic patterns not unlike regular expressions
  - each pattern had an associated reply which might include bits of the input (after simple transformations (my → your)
- Weizenbaum was shocked at reactions:
  - Psychiatrists thought it had potential.
  - People unequivocally anthropomorphized.
  - Many thought it solved the NL problem.

- Try typing "*Meta-x doctor*" in Emacs.
- Recent Loebner Prize winner, http://www.alicebot.org/

Colby's PARRY

- Kenneth Colby modeled a paranoid using the same techniques circa 1968.
- PARRY has basic emotions. If it gets angry, its replies become more hostile.
- In a Turing test, expert psychiatrists were unable to distinguish PARRY's ramblings from those of real paranoids.

H: Good afternoon. My name is Doctor Jones.
P: Pleased to meet you.
H: How are you?
P: All right I guess.
H: Could you tell me your name?
P: Pat Smith.
H: I though you might have some problems.
P: I've been upset lately.
H: By what?
P: People get on my nerves sometimes.
H: Tell me more.
P: Do you know anything about bookies?
...

Parry meets Eliza...

P: I don't understand your motives.

E: Why don't you understand my motives?

P: Shouldn't I be?

E: Why do you say that just now?

P: That's just the way it is.

E: Suppose that's the way it were not.

P: So?

Please tell me *this* isn't AI!

E: I am not sure I understand you fully.

P: You harp on the same questions too much.

E: What does that similarity suggest to you?

P: I don't get you.

E: Do you wish that you did get I?

...

Why is AI Hard?

⇨Computational intractability

- AI goal defined before notion of NP-completeness
  - people thought to solve larger problems we simply need larger/faster computers

ALL the algorithms we will study will be computational intractable (NP-complete as best) in the **WORST** case …

How does the fact that we are dealing with the **REAL WORLD** make solving these computationally challenging problems feasible **IN PRACTICE?**

Complexity Class P

- Deterministic in nature
- Solved by conventional computers in polynomial time
  - $O(1)$ Constant
  - $O(\log n)$ Sub-linear
  - $O(n)$ Linear
  - $O(n \log n)$ Nearly Linear
  - $O(n^2)$ Quadratic
- Polynomial upper and lower bounds

Algorithm vs. Problem Complexity

- **Algorithmic complexity** is defined by the analysis of an algorithm

- **Problem complexity** is defined by
  - An upper bound – defined by an algorithm
  - A lower bound – defined by a proof

Upper and Lower Bounds

- The Upper bound is the best algorithmic solution that has been found for a problem.
  - "What' s the best that we know we can do?"
- The Lower bound is the best solution that is theoretically possible.
  - "What cost can we prove is necessary?"

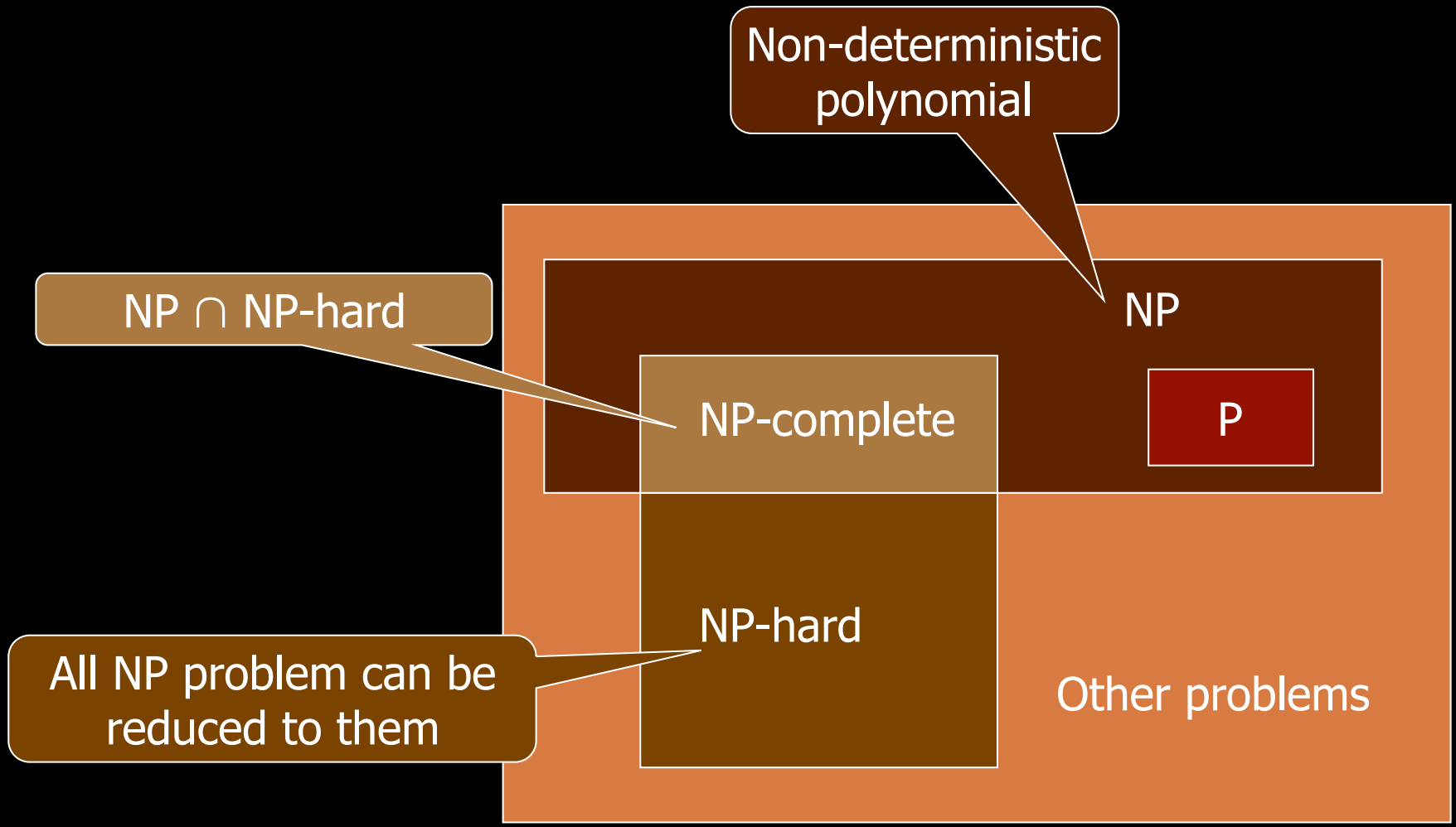
Terminology

- Polynomial algorithms are reasonable
- Polynomial problems are tractable
  - (sort, shortest path, …)

- Exponential algorithms are unreasonable
- Exponential problems are intractable
  - (finding all subset of a set)

- Some pb are neither polynomial nor exponential
  - (planning, TSP, diagnostic, …)

Recall complexity



AI-complete problems

AI-complete: problem would not be solved by a simple specific algorithm.

- Ex cryptarithm: SEND + MORE = MONEY

  10!/2 = 1814400 possibility to test
- Need *heuristics* to integrate expert knowledge, know-how (euriskein = find)

- Ex : chess (10^{160} possible moves)

Ex: Travelling Salesman Problem (TSP)

« *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?* »

- NP-hard problem in combinatorial optimization

Some numbers about the TSP

- TSP with 10 cities: 181 000 possible routes
- TSP with 20 cities:10 000 000 000 000 000 possible routes
- TSP with 50 cities 10^{62} possible routes

- There is (only) 10^{21} liters of water on Earth

Soving an intractable problem with AI

- Do not explore all the search space
- Need « good » solutions instead of optimal ones
- Need to adapt the strategy (heuristics) to the problem, a know-how to the data…
- Aim for a polynomial time even if not optimal

Outline

1. Intro
2. **AI Problem Representation**
3. Problem solving: Uninformed Search
    1. Depth-first search, Breadth-first search, Uniform search, Iterative deepening…
    2. Problem decomposition (AND/OR tree)
4. Heuristic (informed) Search
    1. A*, AO*…
5. Game Playing

Well defined problems

- A problem is generally stated informally (in natural language), it has to be « formalized » before being presented to a machine.
- This problem representation must be « good » because the machine cannot come back on this step.

Example in math

- Informal utterance/statement

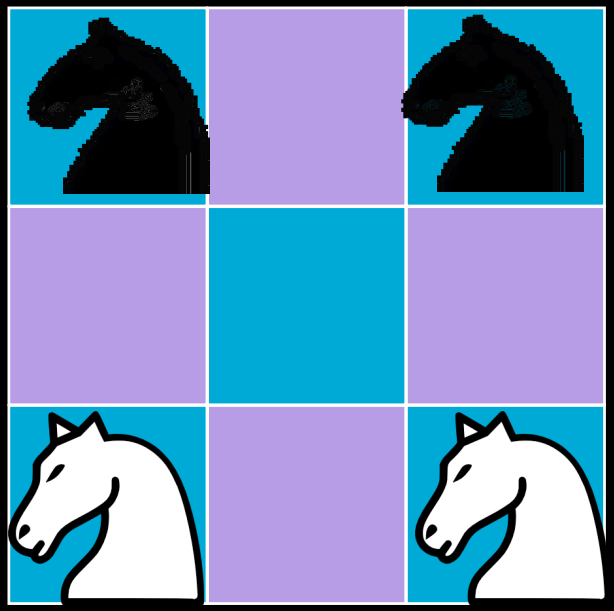
  Find the solution for $x^2 - 9$

- Formal utterance

  Find $\{x \mid x^2 - 9 = 0\}$

- Don't you already give the solution?
- What about $\{x \mid (x-3)(x+3) = 0\}$?

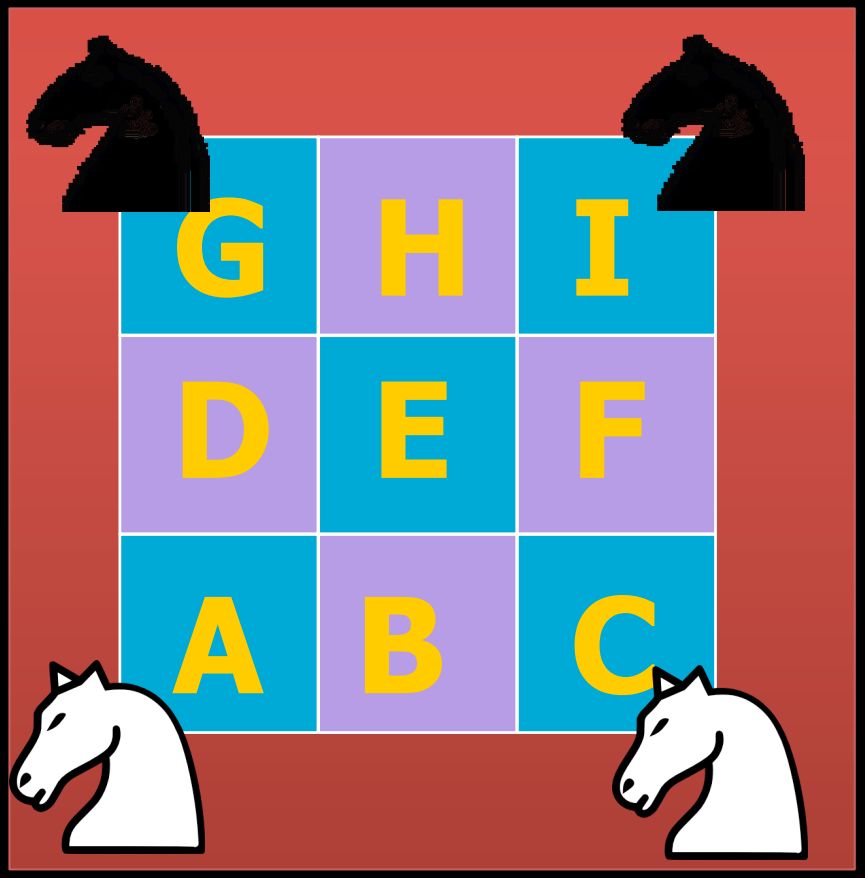
Ex: 4 horses



Informal utterance

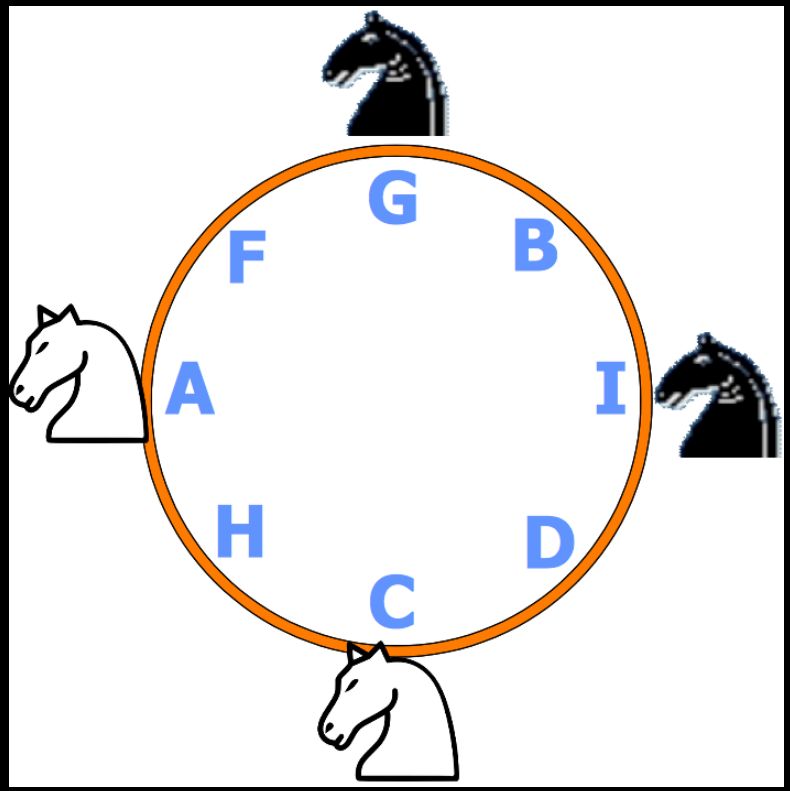
- swap (whenever possible) the 2 white horses with the black ones with as few movement as possible.

4 horses: well defined problem



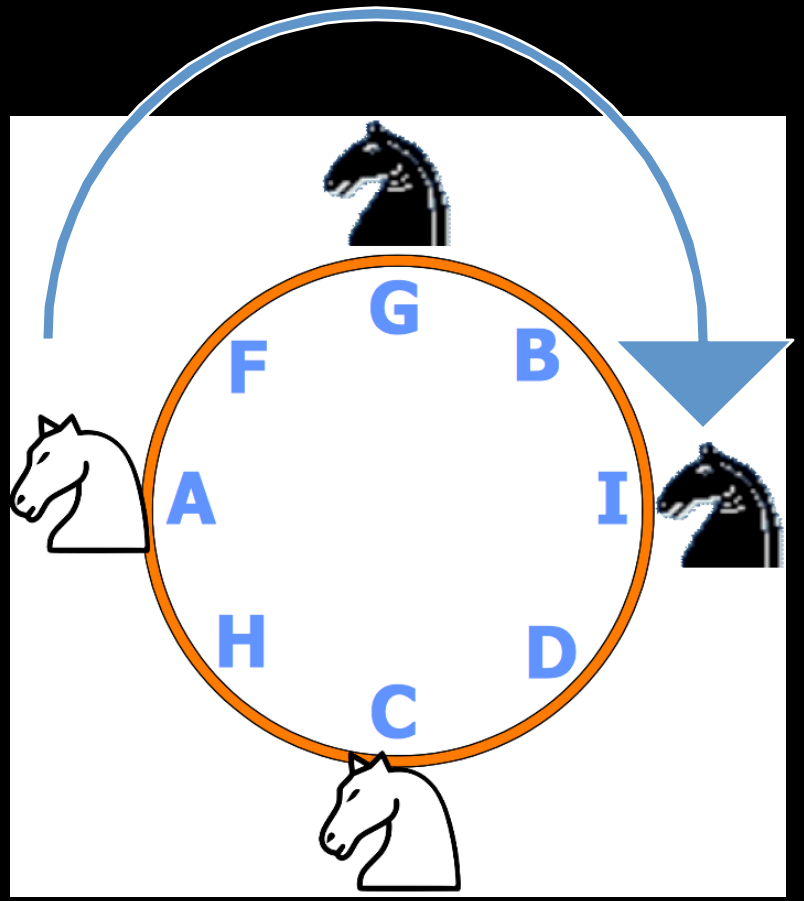
- Get rid of the board
- Model the moves
- Name the different states

4 horses: another representation



- Adjacent points: position reachable in 1 move
- Swap the positions in circle

4 horses: solution



- Make all the pieces do half a turn
  - 4 moves per horse = 16 moves

Well Defined Problems and Solutions

A problem:

- Initial state
- Actions and Successor Function
  - action might have preconditions to be applicable
- Goal test
- Path cost

Example: Water pouring *(1/3)*

Given a 4 gallon bucket and a 3 gallon bucket, how can we measure exactly 2 gallons into one bucket?

- There are no markings on the bucket
- You can empty a bucket completely
- You must fill each bucket completely

Example: Water pouring (2/3):exercise

Initial state:

- The buckets are empty
- Represented by ……..

Goal state:

- One of the buckets has two gallons of water in it
- Represented by …..

Path cost:

- 1 per unit step

Example: Water pouring (3/3) :exercise

Actions and Successor Function

- Fill a bucket
  - -->
  - -->
- Empty a bucket
  - -->
  - -->
- For contents of one bucket into another
  - -->
  - -->

What about preconditions?

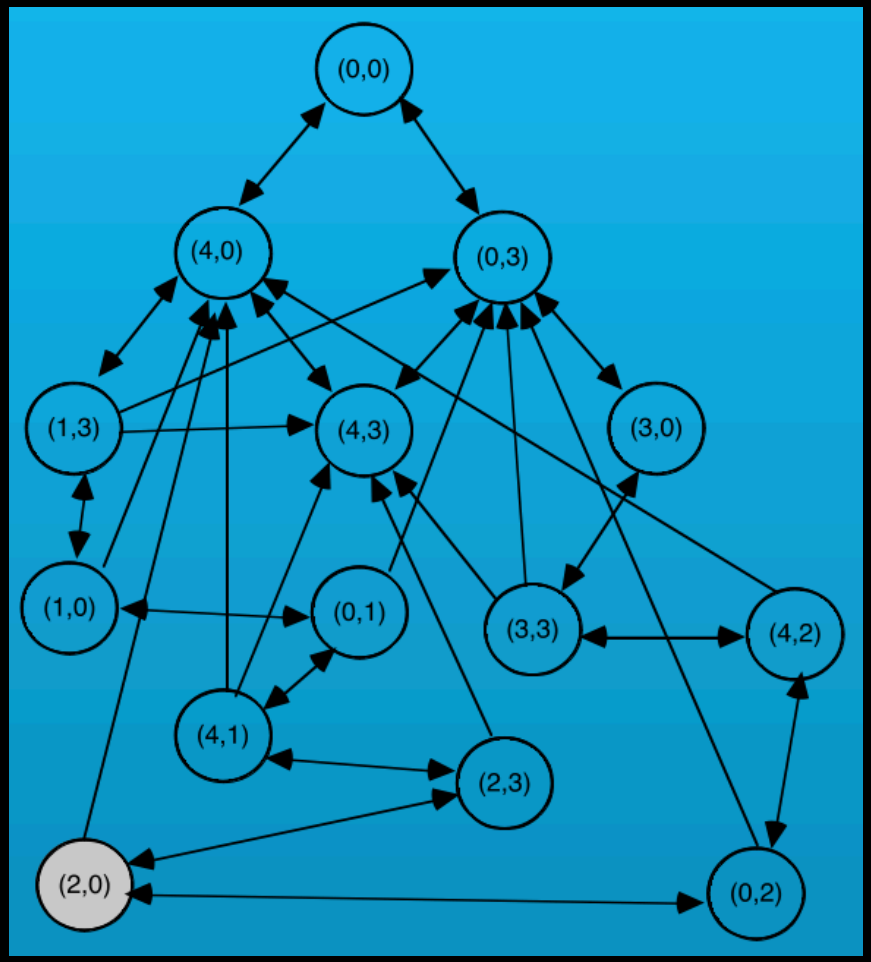
To solve the water jug problem

- Required a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to goal state.

- One solution to the water jug problem :

Shortest such sequence will have a impact on the choice of appropriate mechanism to guide the search for solution.

Solution to the water jug problem



Gallons in the 4-gallon jug	Gallons in the 3-gallon jug	Rule applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5 or 12
0	2	9 0r 11
2	0	

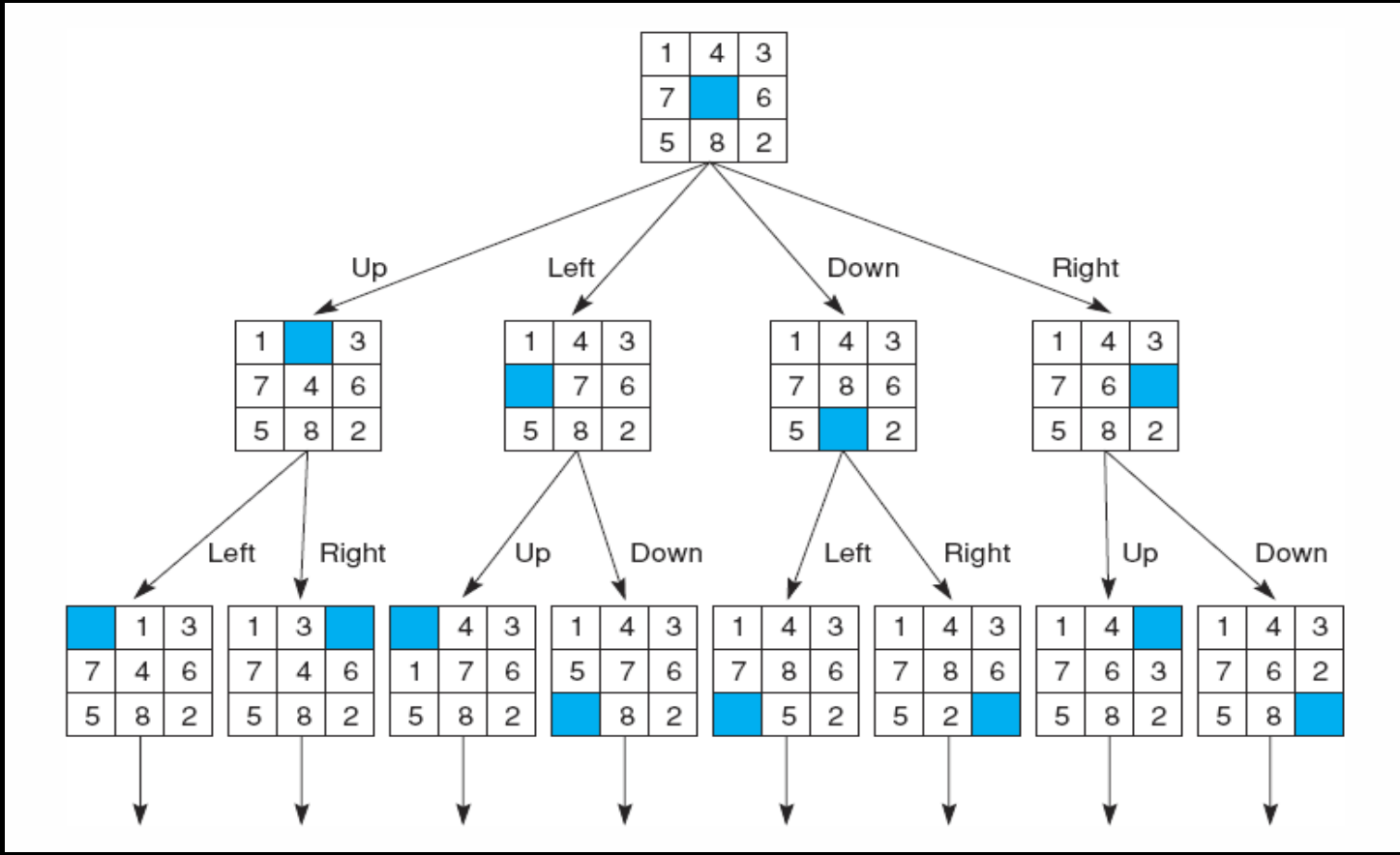
Example: Eight Puzzle

- States:
  - Description of the eight tiles and location of the blank tile
- Successor Function:
  - Generates the legal states from trying the four actions {*Left, Right, Up, Down*}
- Goal Test:
  - Checks whether the state matches the goal configuration
- Path Cost:
  - Each step costs 1

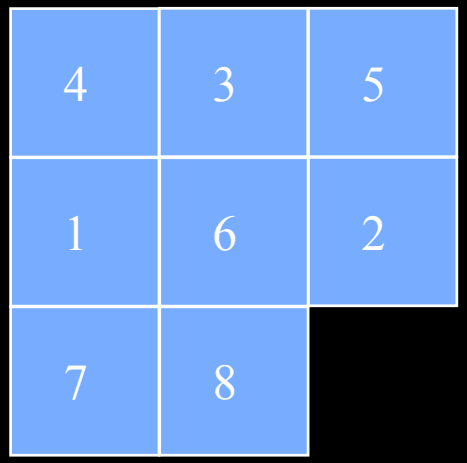
7	2	4
5		6
8	3	1

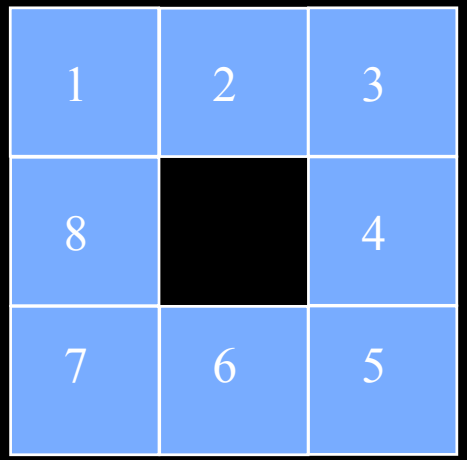
1	2	3
4	5	6
7	8	

Solution Eight Puzzle



More formally (exercise)





Matrix M 3 x 3

Blank tile: (l_v, c_v)

Function UP

- Precondition : $l_v > 1$
- Effect :

  $M(l_v,c_v) \leftarrow M(l_v-1, c_v)$

  $l_v \leftarrow l_v-1$

Write DOWN, LEFT, RIGHT.

Rk on Eight Puzzle

- Eight puzzle is from a family of “sliding –block puzzles”
  - NP Complete
  - 8 puzzle has 9!/2 = 181440 states
  - 15 puzzle has approx. $1.3*10^{12}$ states
  - 24 puzzle has approx. $1*10^{25}$ states

Example: Eight Queens *(1/3)*

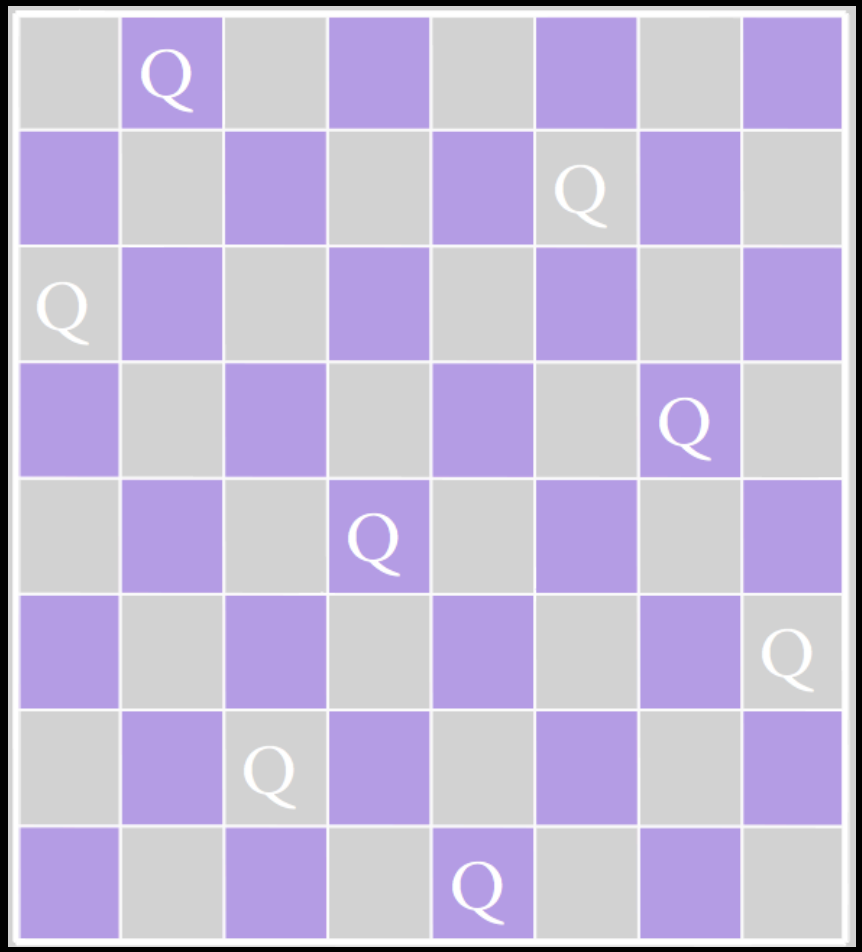
- Place 8 queens on a 8*8 chess board such that no queen can attack another queen

- No path cost because only the final state counts!

- Incremental formulations

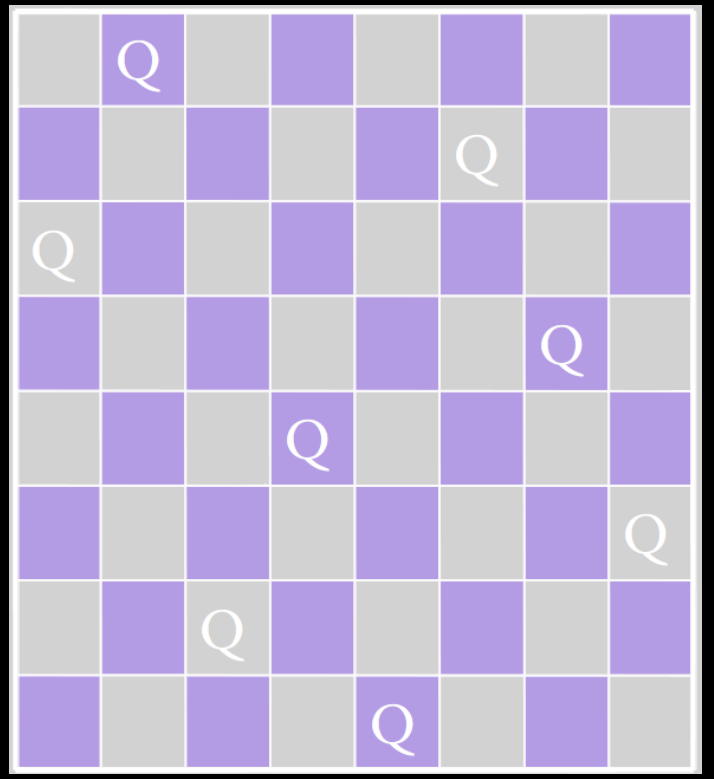
or

- Complete state formulations



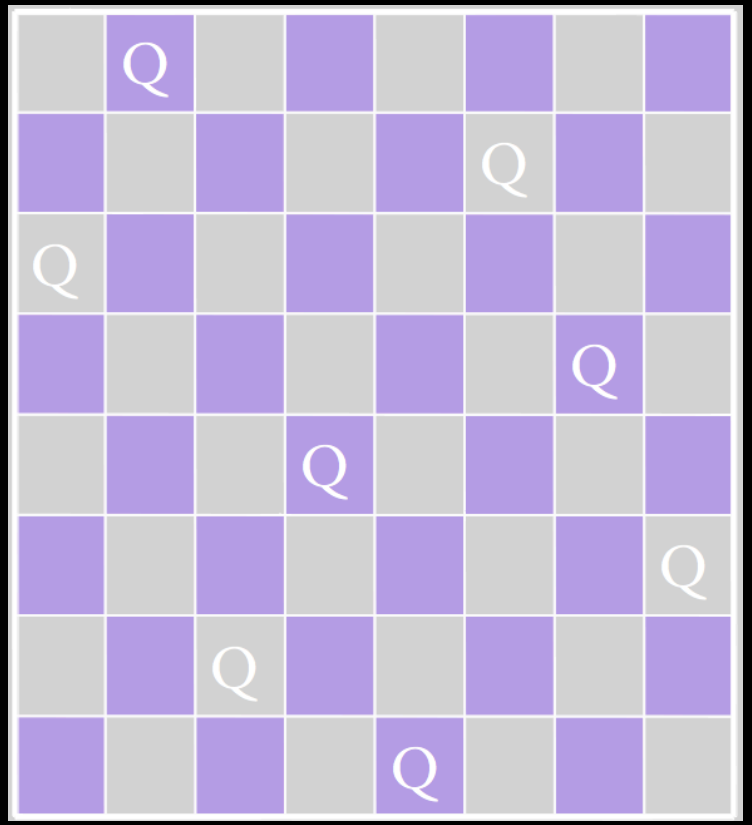
Example: Eight Queens *(2/3)*

States:
Any arrangement of 0 to 8 queens
• Initial state:
No queen on the board
• Successor function:
Add a queen to an empty square
• Goal test:
8 queens on the board and none are attacked
• $64 \times 63 \times \cdots \times 57 = 1.8 \times 10^{14}$ possible sequences. . . OUCH!!!!



Example: Eight Queens *(3/3)*

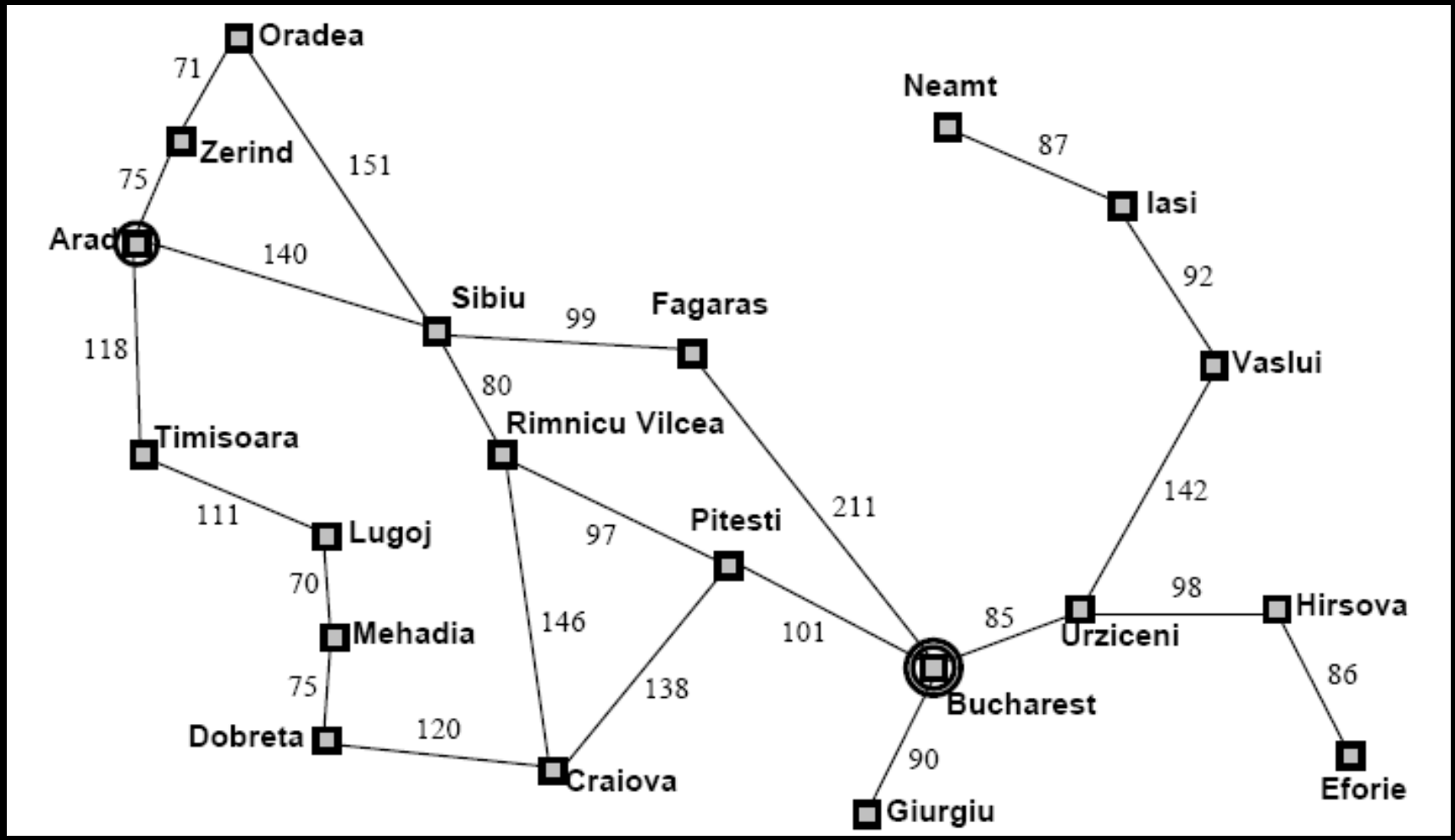
States:
Arrangements of n queens, one per column in the leftmost n columns, with no queen attacking another are states



- Successor function:
Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen

To be formalized!
- 2, 057 sequences to investigate

Example: Map Planning



Map Planning: Searching For Solutions

- Initial State
  - e.g. “At Arad”
- Successor Function
  - A set of action state pairs
  - S(Arad) = {(Arad->Zerind, Zerind), …}
- Goal Test
  - e.g. x = “at Bucharest”
- Path Cost
  - sum of the distances traveled

Searching For Solutions

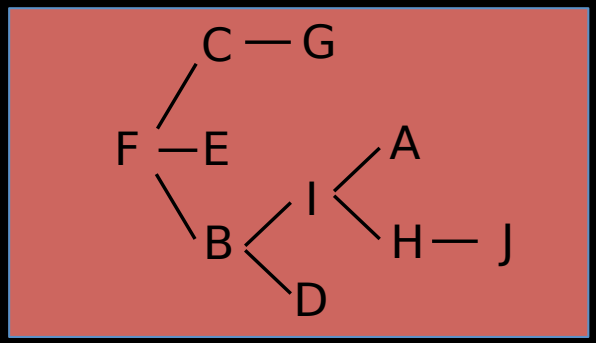
- Having formulated some problems…how do we solve them?
- Search through a state space
- Use a search tree that is generated with an initial state and successor functions that define the state space

Searching the State Space

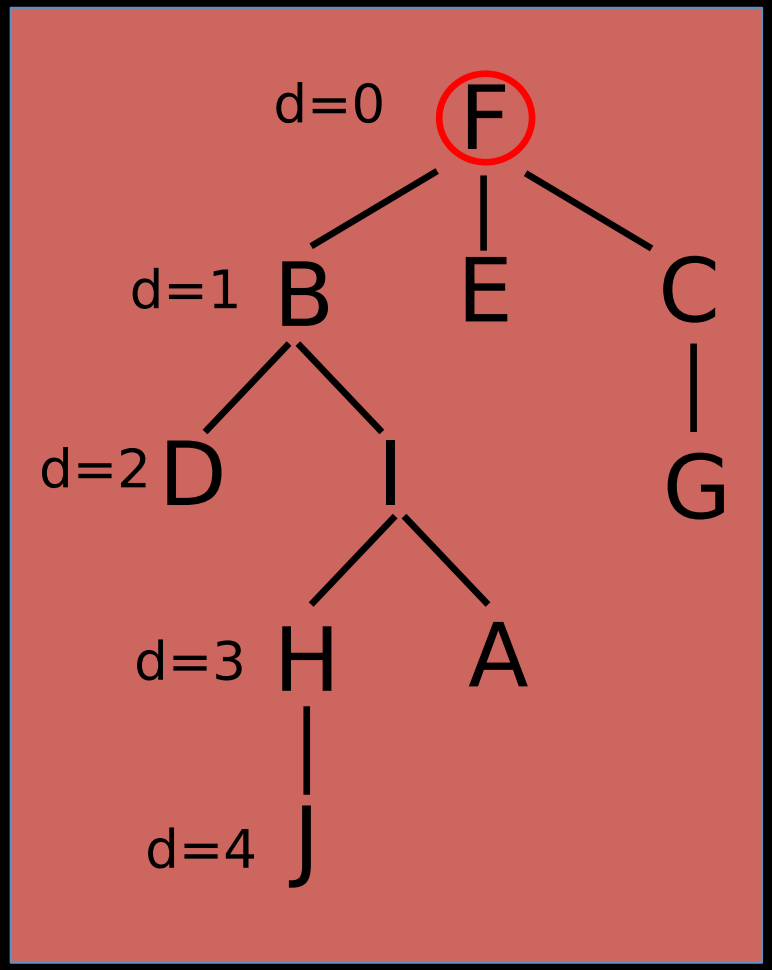
- Representing the entire space of problems as state space provides a powerful tool for measuring the structure and complexity of problems and analyifs of the efficiency, correctness and generality of solution strategies
- Problem solving is a process of searching the state space for a path to a solution
- The choice of which state to expand is determined by the search strategy
- The corresponding sequences of state expansion is represented by a data structure known as a search tree

Search Trees

- A **tree** is a graph that:
  - is connected but becomes disconnected on removing any edge
  - is connected and acyclic
  - has precisely one path between any two nodes
- Property 3, unique paths, makes them much easier to search and so we will start with search on (rooted) trees

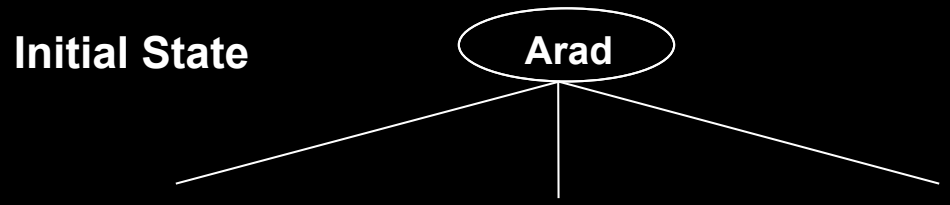


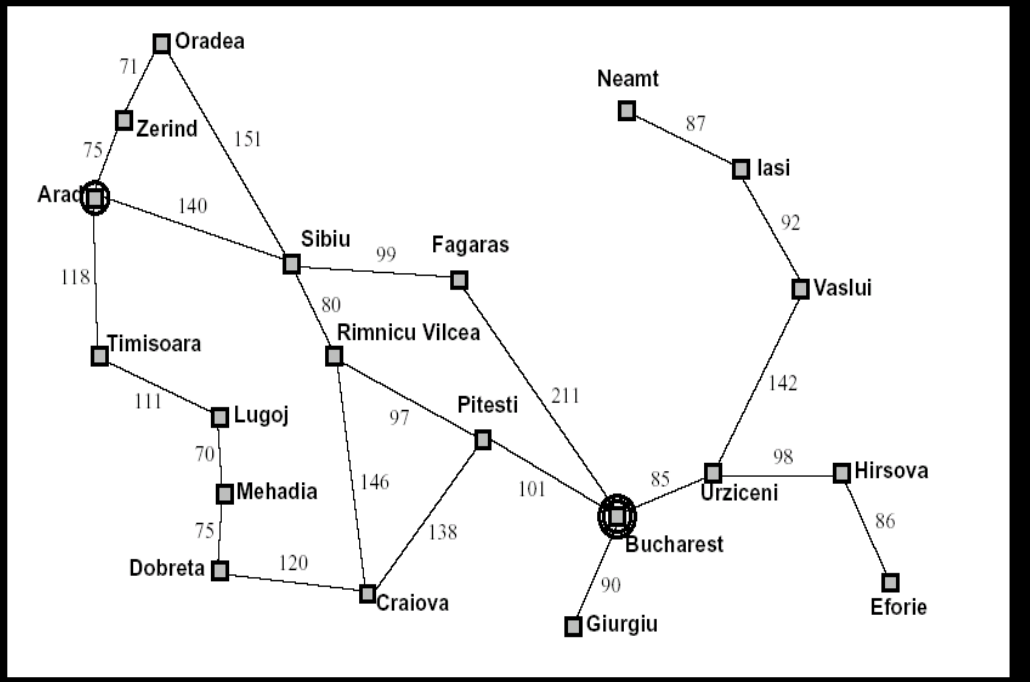
Trees - Terminologies



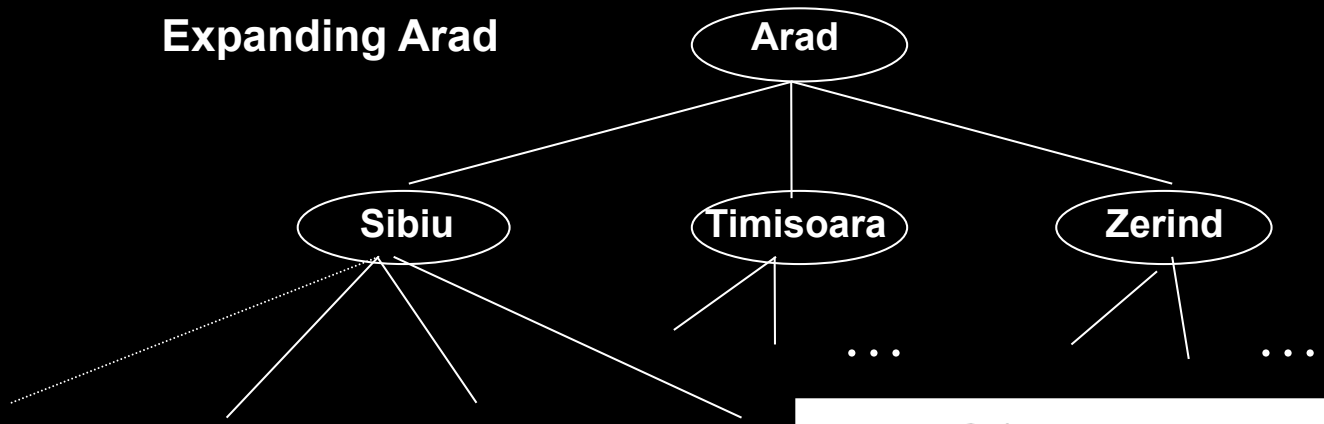
- Nodes
  - Root Node
  - Children Node
  - Parent Node
  - Leaves
- Branching Factor
  - Average number of children for the nodes of a tree
- The depth, d, of a node is just the number of edges away from the root node
- The depth of a tree is the depth of the deepest node
  - in this case, depth = 4

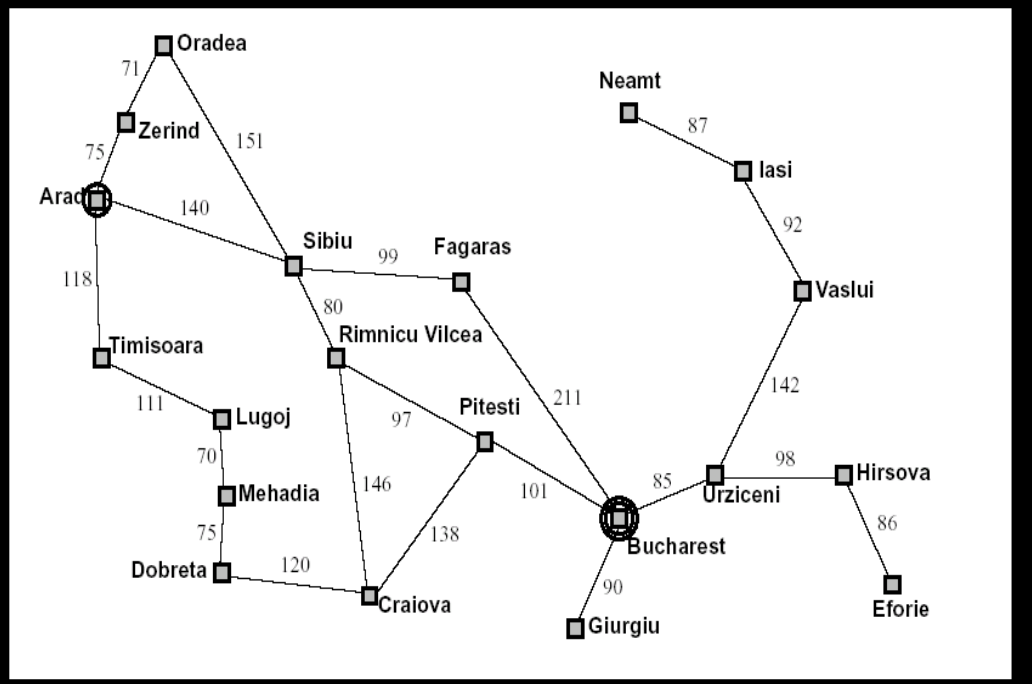
Search Tree - Romania



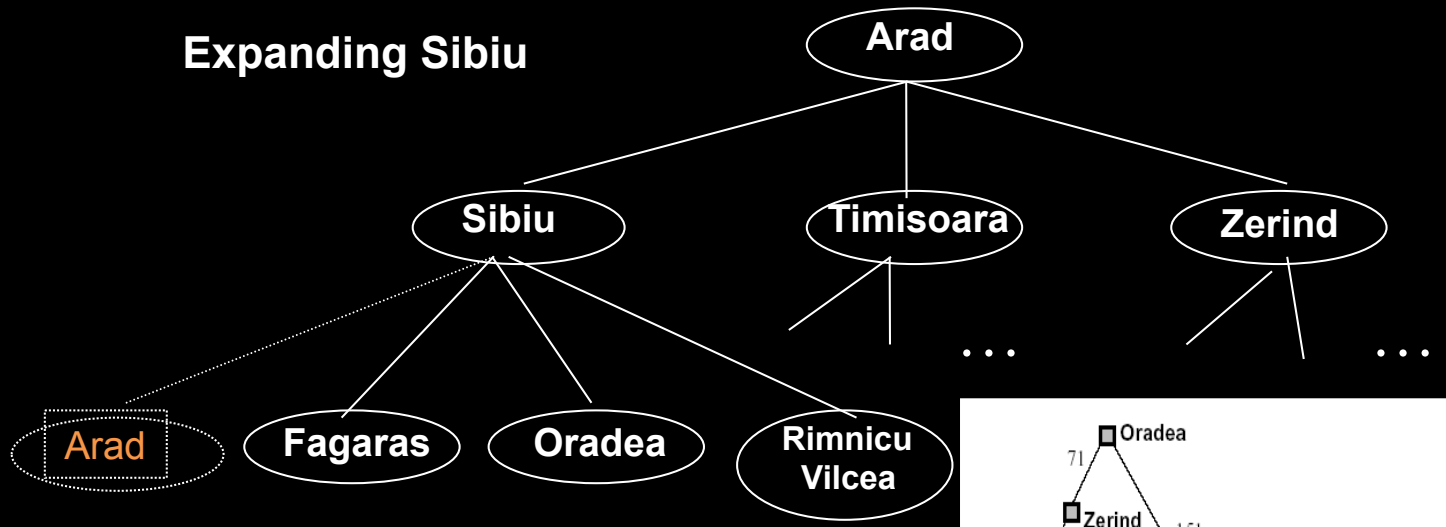


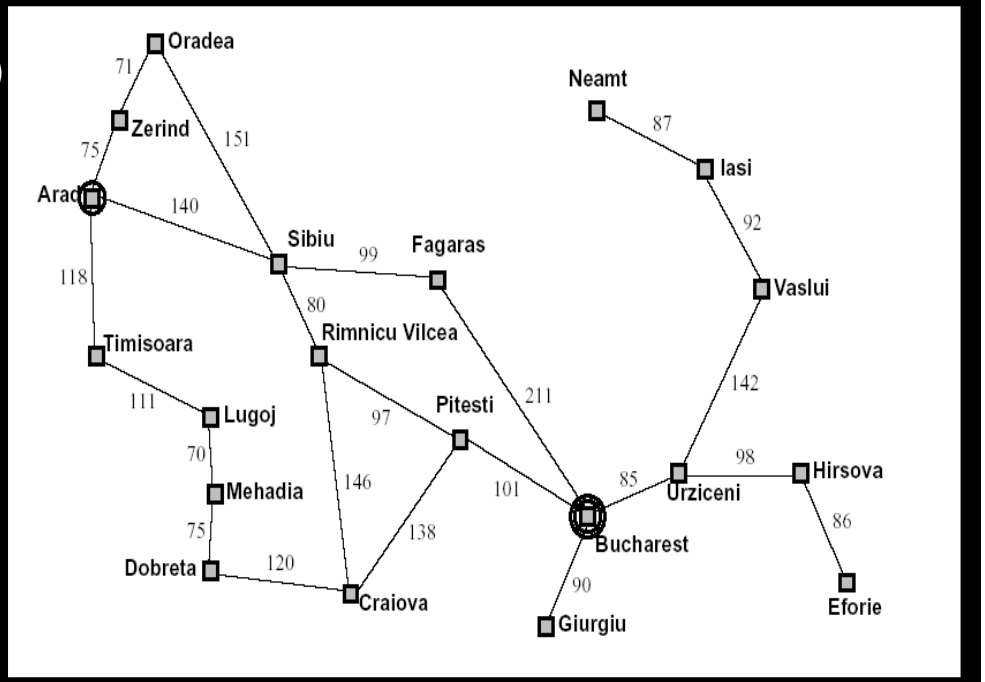
Search Tree - Romania



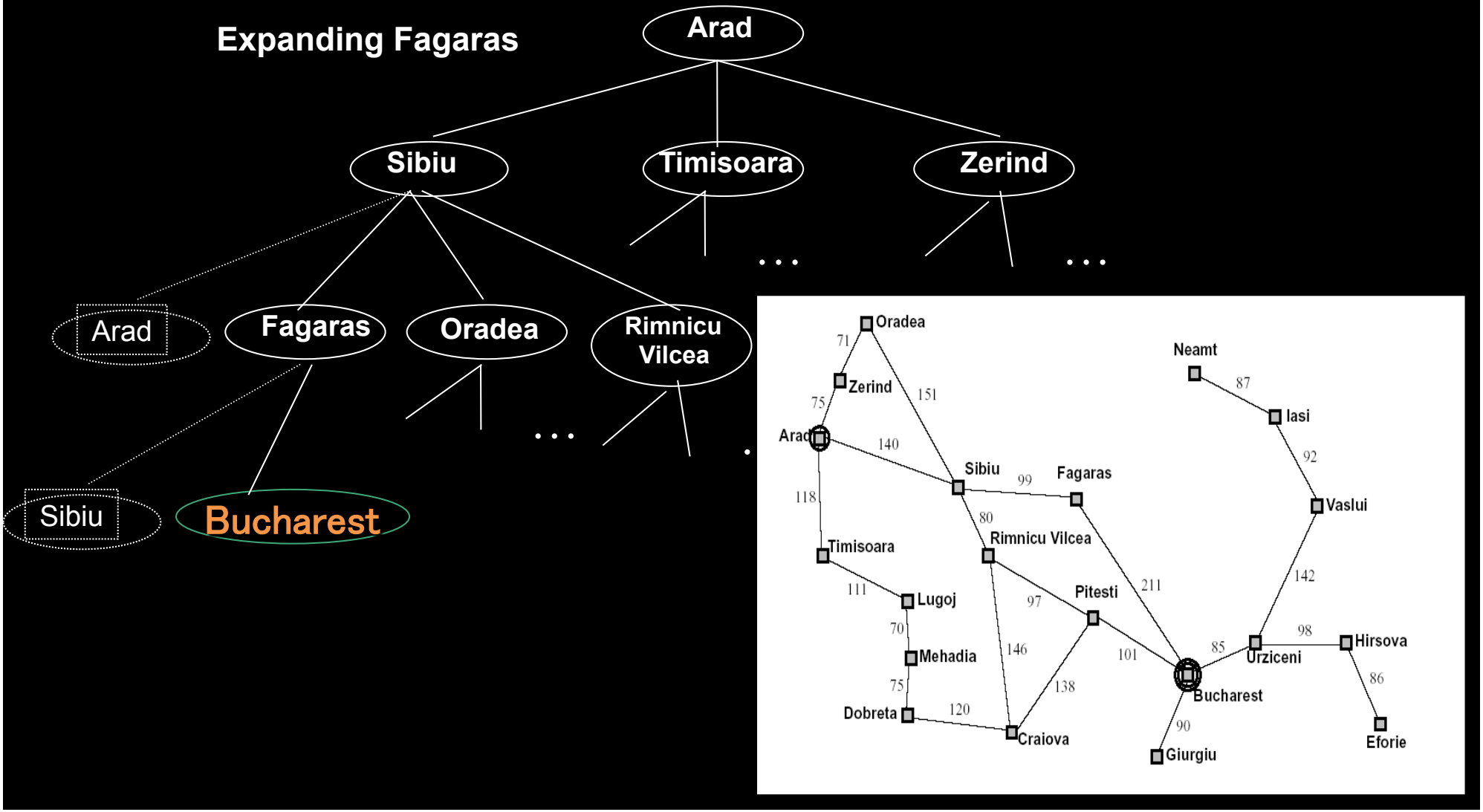


Search Tree - Romania



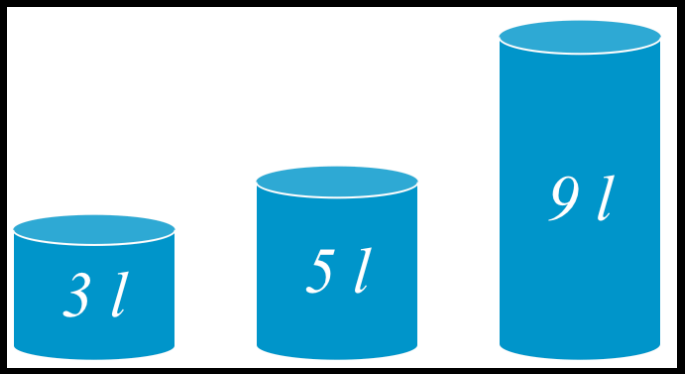


Search Tree - Romania



Exercise: the return of the water pouring problem

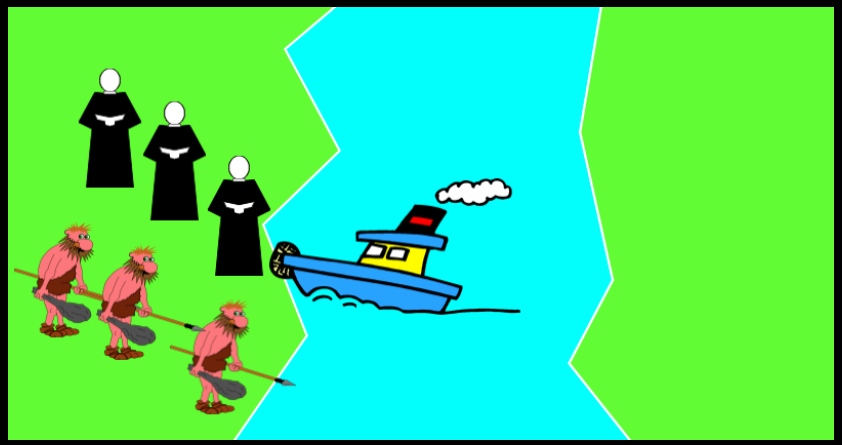
- Measure 7 litters from these **3 buckets** (put 7L in the 9L bucket)
- Same restrictions as before
- The buckets are empty
- Formalize the problem (with precondition for the actions) and show the first level of the search tree



What's the best solution?

- Multiple branches lead to a possible solution
- The best solution depends on the cost function:
  - You might not want to spoil water
  - You may not be strong enough (or it might make you more tired to weigh a bucket)
  - …

Exercise: missionaries and cannibals

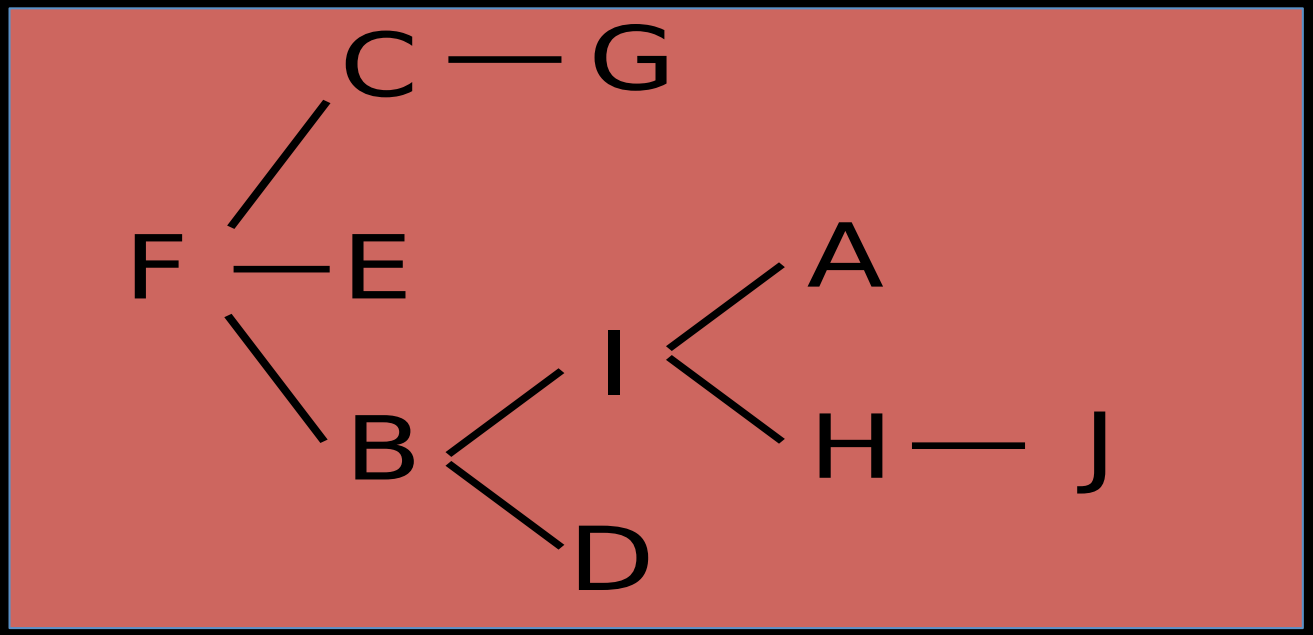


Three missionaries and three cannibals are on one side of a river, with a canoe. Somebody should drive the canoe and everybody must get off the canoe when the canoe reaches one side.They all want to get to the other side of the river. The canoe can only hold one or two people at a time. At no time should there be more cannibals than missionaries on either side of the river, as this would probably result in the missionaries being eaten.

Q: Formulate this problem and with appropriate representation and draw the state space.

Finding Goals in Trees

- Does the following tree contain a node « I »?



- Yes. How did you know that? « read it »?
- « Trivial! »: so why the big deal about search?

Why is Goal Search Not Trivial?

- Because the graph is not given in a nice picture "on a piece of paper"
- At the start of the search, the search algorithm does not know
  - the size of the tree
  - the shape of the tree
  - the depth of the goal states
- How big can a search tree be?
  - say there is a constant branching factor b
  - and one goal exists at depth d (root is a depth 0)
  - search tree which includes a goal can have b^d different branches in the tree (worst case)
- Examples:

  b = 2, d = 10: $b^d = 2^{10} = 1{,}024$

  b = 10, d = 10: $b^d = 10^{10} = 10{,}000{,}000{,}000$

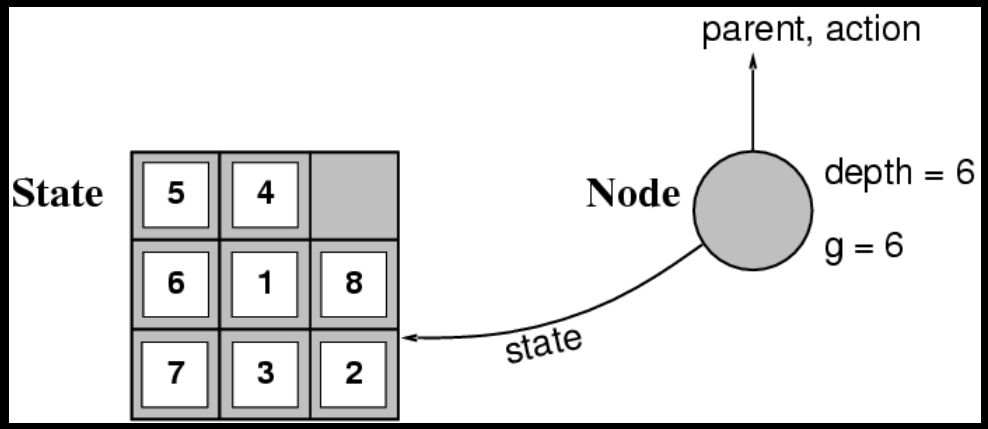
Finding Goals in Trees: Reality

- Does the tree under a given root contain a node “G”?
- All you get to see at first is the root node and a guarantee that it is a tree
- The rest is up to you to discover during the process of search → discover/create “on the fly”

Properties of Search

- We will say a search method is "complete" if it has both the following properties:
  - if a goal exists then the search will always find it
  - if no goal exists then the search will eventually finish and be able to say for sure that no goal exists
- We only look at complete search methods
- Rk: incomplete search methods often work better in practice, but not seen here (ex: evolutionary algorithms in machine learning)

Implementation



- A **node** is a bookkeeping data structure from which a search tree is constructed. It contains information such as: state, parent node, action, path cost g(x), depth
- A **state** is the configuration in the state space to which the node corresponds
- The `Successor-Fn(x)` (successor function) returns all states that can be reached from state x
- **Fringe** is the collection of nodes that have been generated but not (yet) expanded. Each node of the fringe is a leaf node

Implementation
Node Representation

```
Datatype NODE
Components:
```

- STATE,
- PARENT-NODE,
- OPERATOR,
- DEPTH,
- PATH-COST

- STATE: This represents the state in the state space to which a node corresponds
- PARENT-NODE: This points to the node that generated this node. In a data structure representing a tree it is usual to call this the parent node
- OPERATOR: The operator that was applied to generate this node
- DEPTH: The number of nodes from the root
- PATH-COST: The path cost from the initial state to this node

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?

Problem Characteristics

In order to choose the most appropriate method for a particular problem, it is necessary to analyze the problem along several key dimensions:

1. Is the problem decomposable into a set of independent smaller or easier subproblems?
2. Can solution steps be ignored or at least undone if they prove unwise?
3. Is the problem's universe predictable?
4. Is a good solution to the problem obvious without comparison to all other possible solutions?
5. Is the desired solution a state of the world or a path to a state?
6. Is a large amount of knowledge absolutely required to solve the problem or is knowledge important only to constrain the search?
7. Can a computer that is simply given the problem return the solution or will the solution of the problem require interaction between the computer and a person?

Outline

1. Intro
2. AI Problem Representation
3. **Problem solving: Uninformed Search**
    1. Depth-first search, Breadth-first search, Uniform search, Iterative deepening…
    2. Problem decomposition (AND/OR tree)
4. Heuristic (informed) Search
    1. A*, AO*…
5. Game Playing

Sub-Outline

Uninformed strategies use only the information available in the problem definition

- Also known as blind searching

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Key issue: type of queue used for the “fringe” of the search tree

Comparing Uninformed Search Strategies *(1/2)*

Completeness

- Will a solution always be found if one exists?

Time

- How long does it take to find the solution?
- Often represented as the number of nodes searched

Space

- How much memory is needed to perform the search?
- Often represented as the maximum number of nodes stored at once

Optimal

- Will the optimal (least cost) solution be found?

Comparing Uninformed Search Strategies *(2/2)*

Time and space complexity are measured in

- b – maximum branching factor of the search tree
- m – maximum depth of the state space
- d – depth of the least costly solution

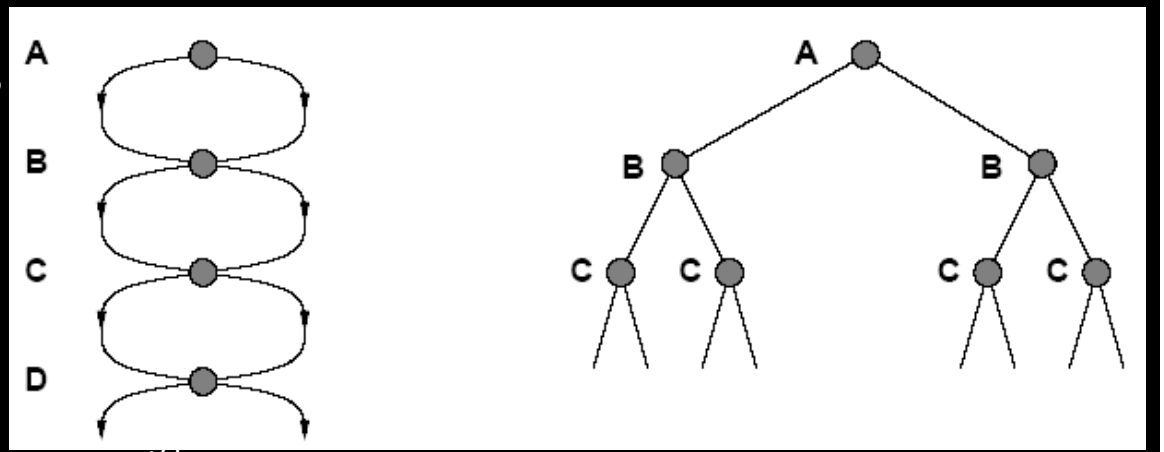
Search Algorithm

```
INSERT(initial-node,FRINGE)
loop
   if FRINGE = ∅ then
      return FAILURE
   end if
   CURRENT ← REMOVE(FRINGE)
   s ← STATE(CURRENT)
   if GOAL?(s) then
      return path or goal state
   end if
   for state s' in SUCCESSORS(s) do
      Create a node N
      INSERT(N,FRINGE)
   end for
end loop
```

Avoiding Repeated States

- Complication of wasting time by expanding states that have already been encountered and expanded before
  - Failure to detect repeated states can turn a linear problem into an exponential one

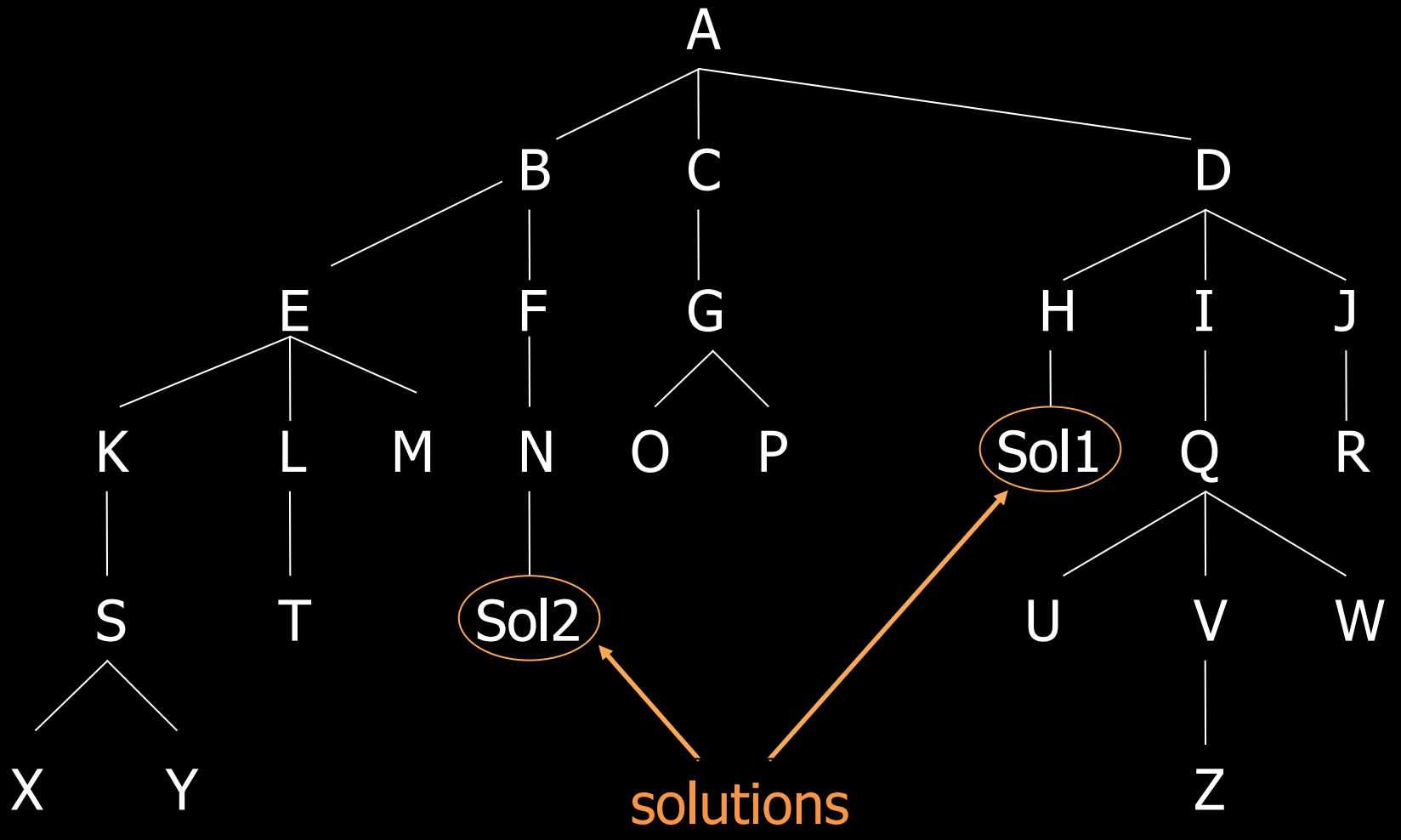
- Sometimes, repeated states are unavoidable
  - Problems where the actions are reversable
    - Route finding
    - Sliding blocks puzzles



Avoiding Repeated States

```
CLOSED ← ∅
FRINGE ← INSERT(initial-node,FRINGE)
loop
   if FRINGE = ∅ then return FAILURE
   CURRENT ← REMOVE-FRONT(FRINGE)
   if GOAL?(CURRENT) then return CURRENT
   if STATE(CURRENT) ∉ CLOSED then
      CLOSED ← CLOSED ∪ STATE(CURRENT)
      FRINGE ← INSERTALL(EXPAND(CURRENT),FRINGE)
   end if
end loop
```

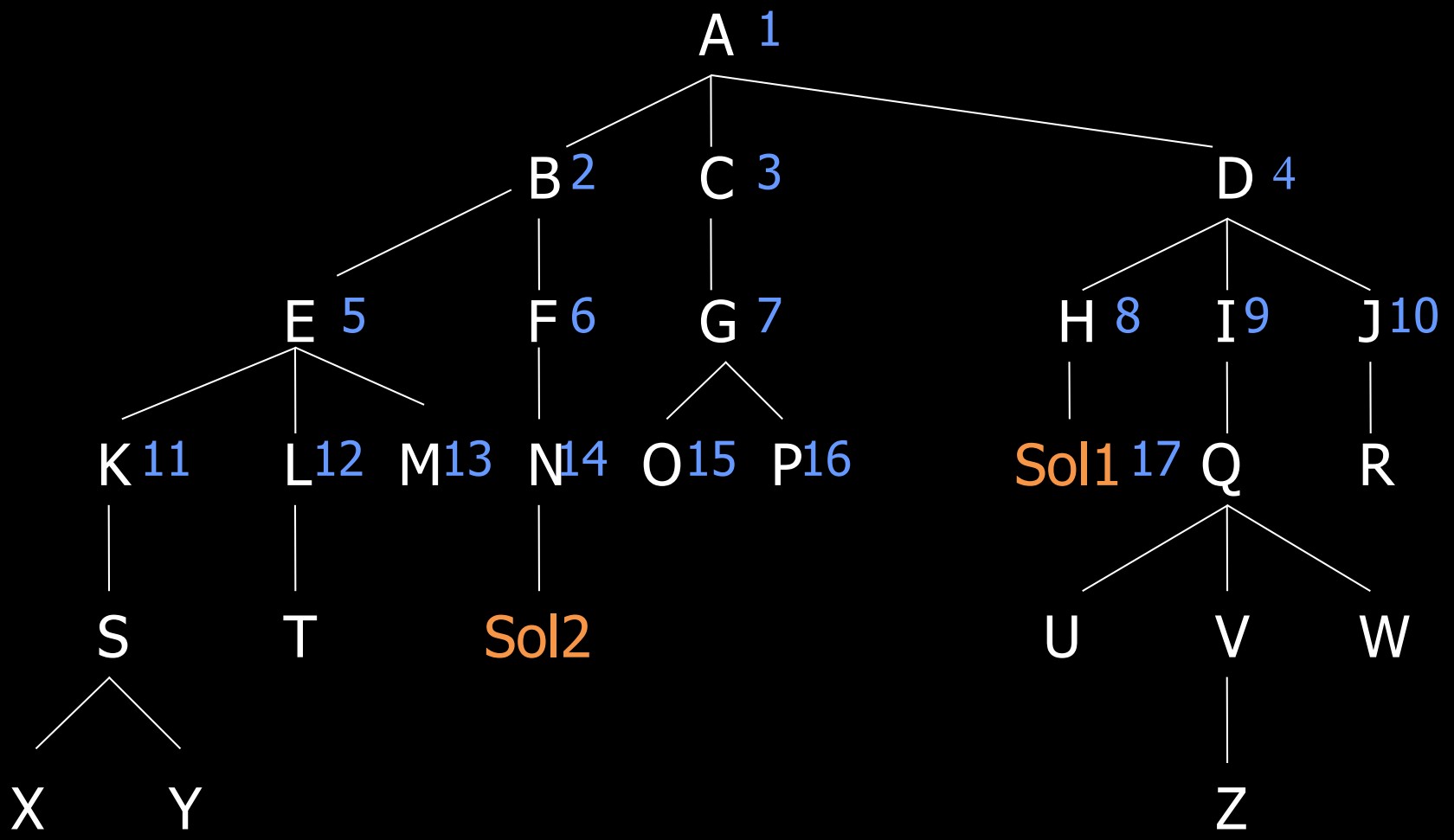
Running Example



Breadth-First Search

- Recall from Data Structures the basic algorithm for a breadth-first search on a graph or tree
- Expand the **shallowest** unexpanded node
- Place all new successors at the end of a FIFO queue

Breadth-first

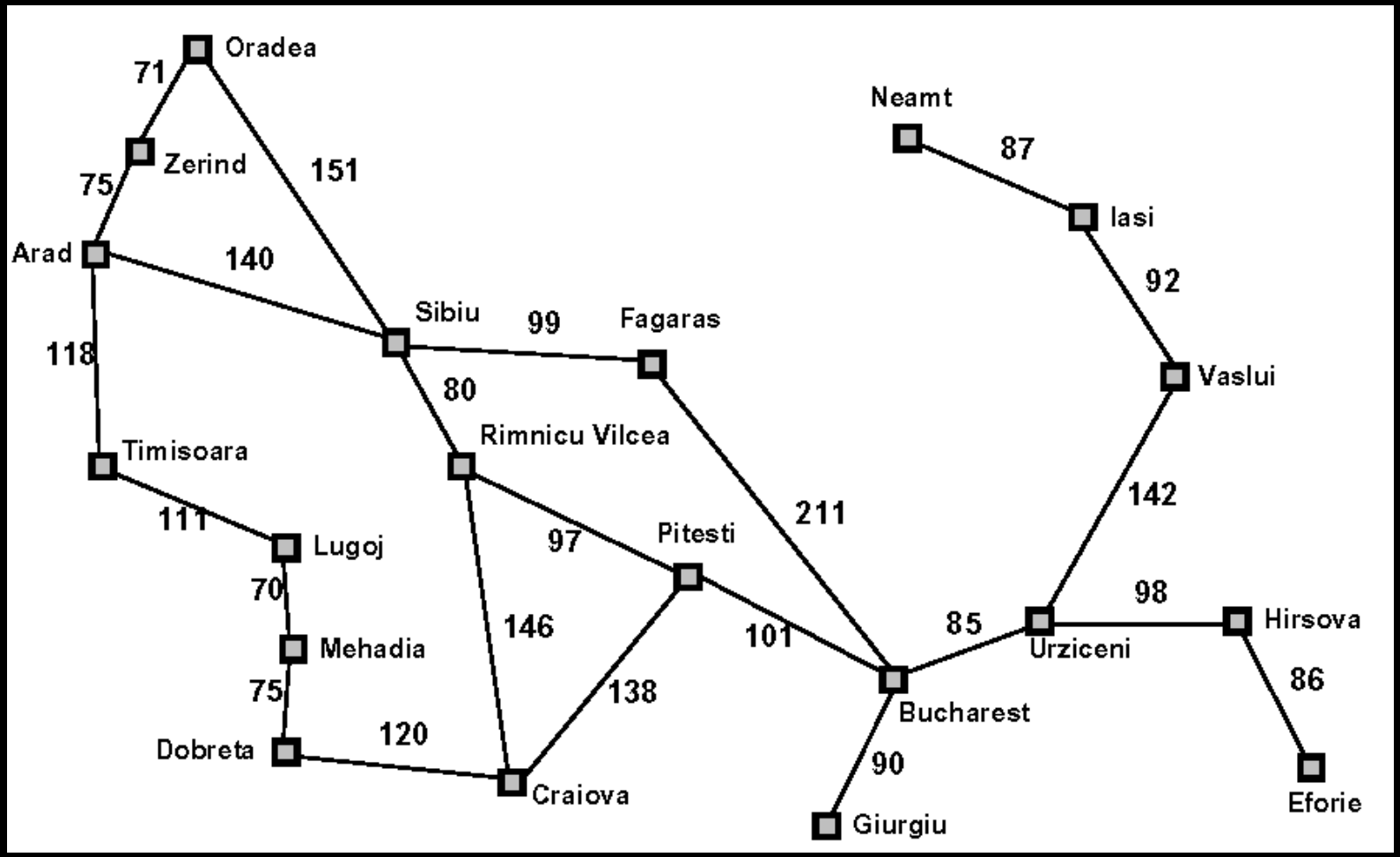


Properties of Breadth-First Search

- Complete
  - Yes if b (max branching factor) is finite
- Time
  - $1 + b + b^2 + \ldots + b(b^{d-1}) + b*b^d = O(b^{d+1})$
  - exponential in d
- Space
  - $O(b^d)$
  - Keeps every node in memory
  - This is the big problem; an agent that generates nodes at 10 MB/sec will produce 860 MB in 24 hours
- Optimal
  - Yes (if cost is 1 per step); not optimal in general

Running Exercise

Travel in Romania (from S. Russel et P. Norvig)



Exercise (questions)

Travel in Romania (from S. Russel et P. Norvig)

- Try to reach Bucharest from Arad
- Try out the breath-first algorithm in this example (without using the distances)
  - evolution of FRINGE and CLOSED
- How many cities did you visit? How many kilometres did you travel?

Lessons From Breadth First Search

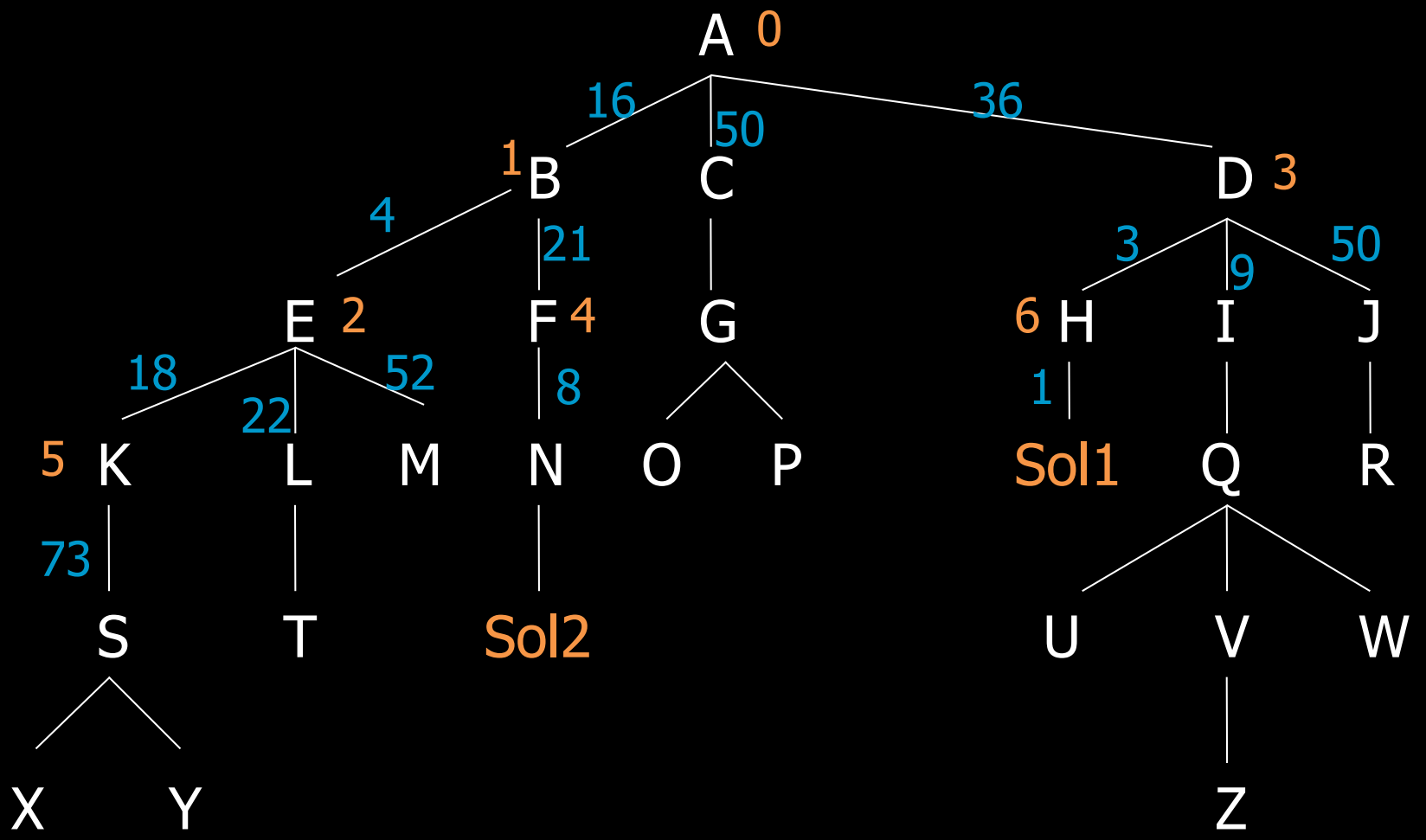
- The memory requirements are a bigger problem for breadth-first search than is execution time

- Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances

Uniform-Cost Search

- Same idea as the algorithm for breadth-first search but…
  - Expand the least-cost unexpanded node
  - FIFO queue is ordered by cost
  - Equivalent to regular breadth-first search if all step costs are equal

uniform cost example

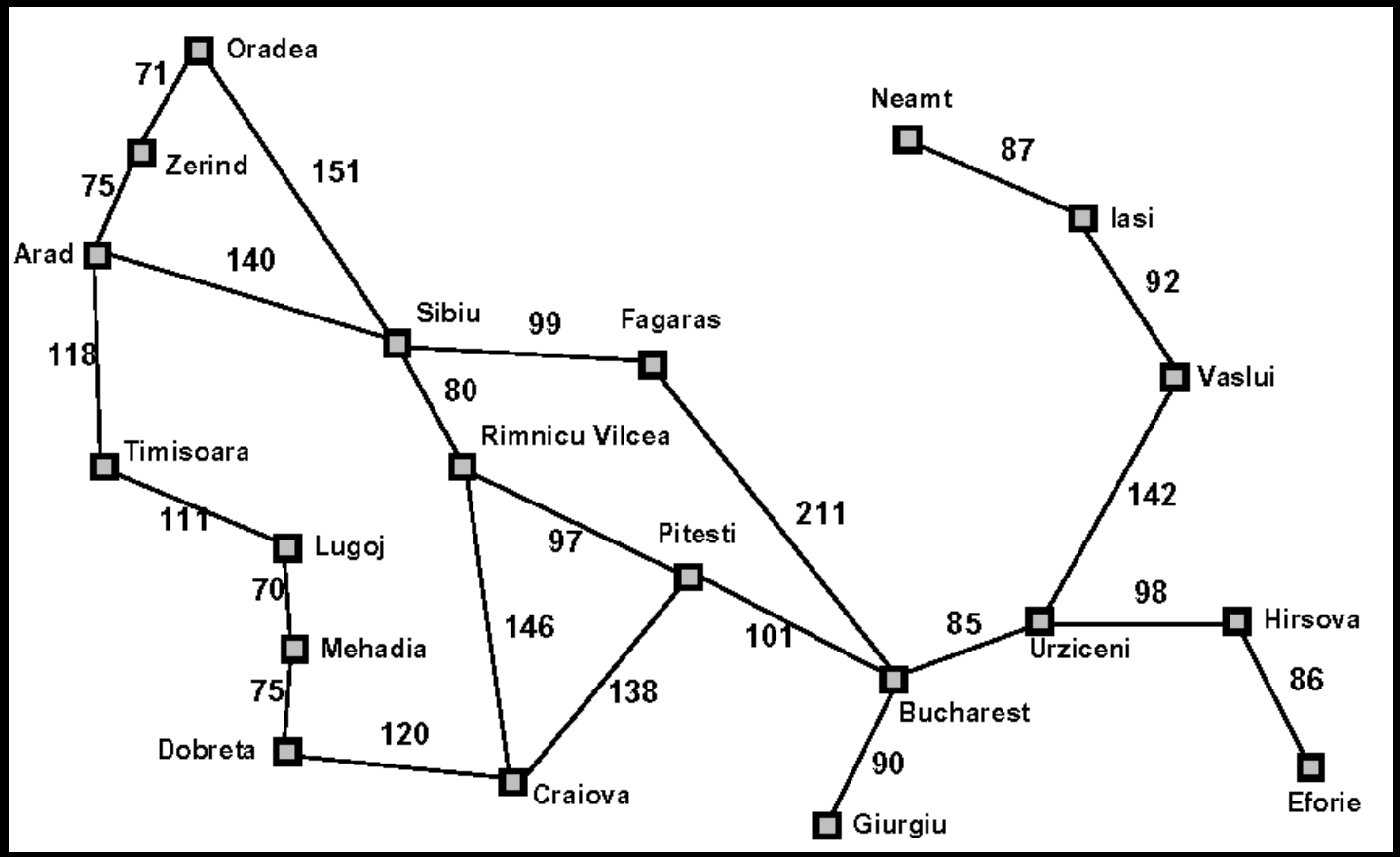


Uniform-Cost Search

- Complete
  - Yes if the cost is greater than some threshold
  - step cost >= ε
- Time
  - Complexity cannot be determined easily by d or d
  - Let C* be the cost of the optimal solution
  - $O(b^{1+[C^*/\varepsilon]})$
- Space
  - $O(b^{1+[C^*/\varepsilon]})$
- Optimal
  - Yes, Nodes are expanded in increasing order

Exercise: uniform cost

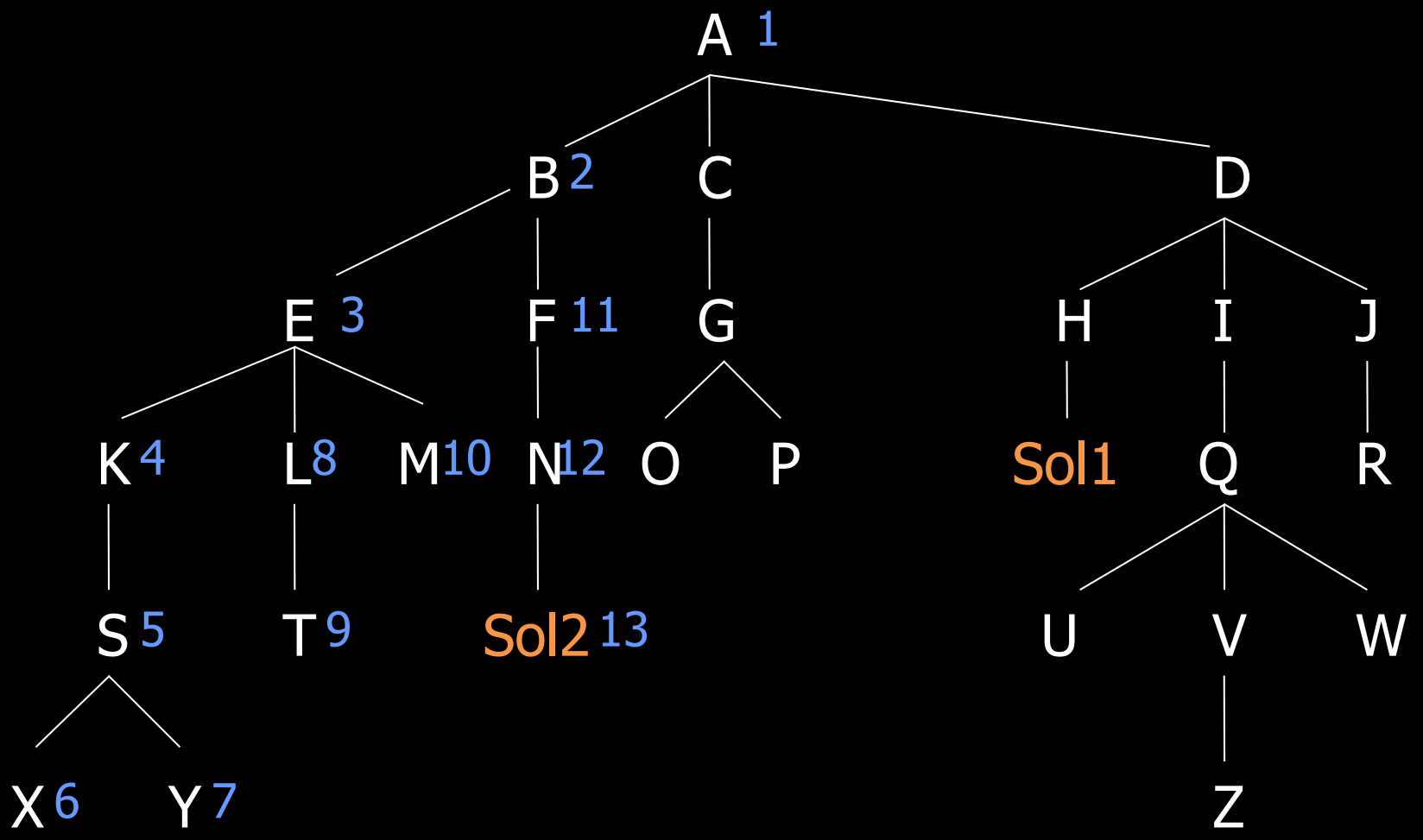
Travel in Romania (same questions as before but with as few kilometres as possible)



Depth-First Search

- Recall from Data Structures the basic algorithm for a depth-first search on a graph or tree
- Expand the **deepest** unexpanded node
- Unexplored successors are placed on a stack (LIFO queue) until fully explored

depth-first
(with or without backtrack)

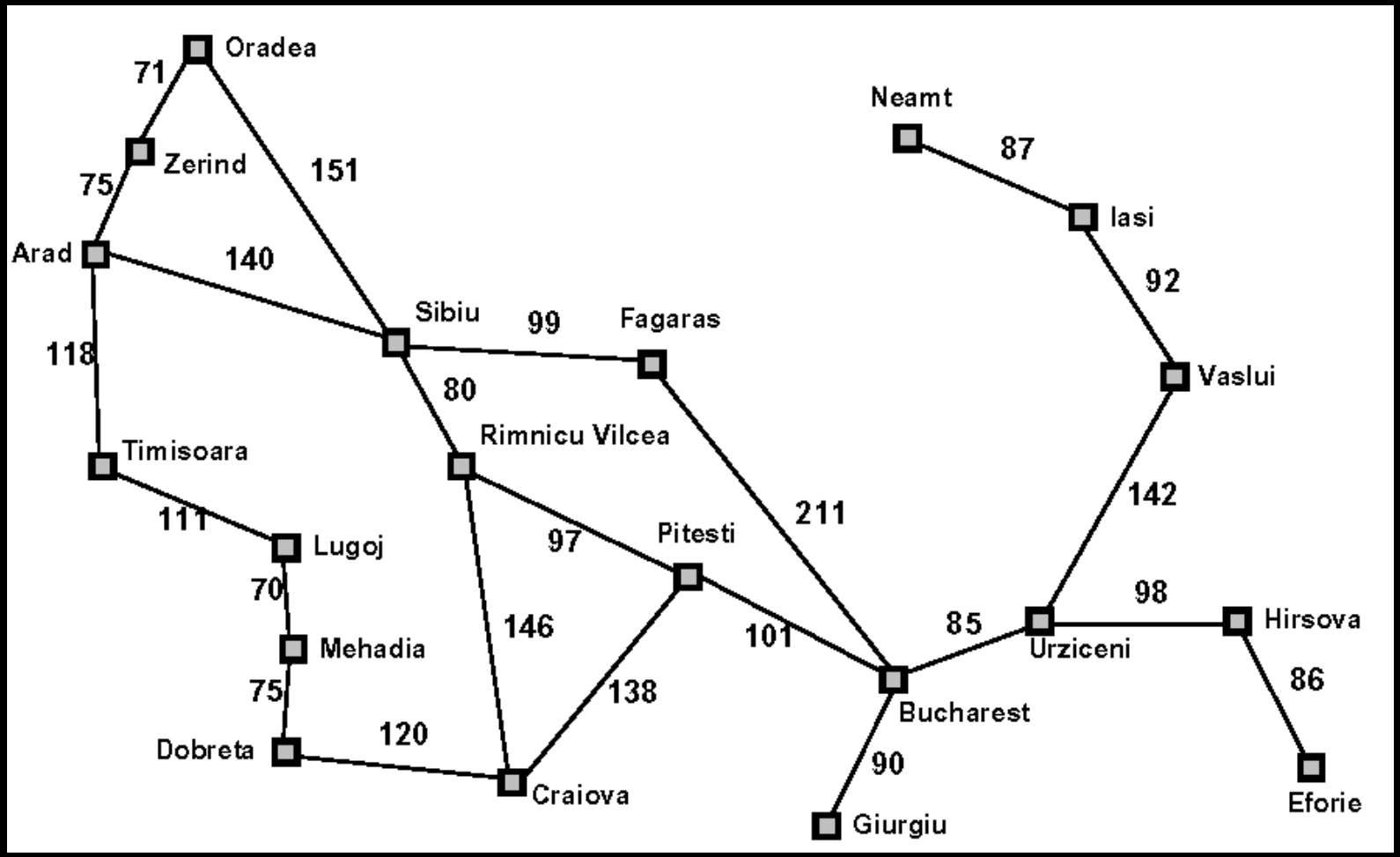


Depth-First Search

- Complete
  - No: fails in infinite-depth spaces, spaces with loops
    - Modify to avoid repeated spaces along path
  - Yes: in finite spaces
- Time
  - $O(b^m)$
  - Not great if m is much larger than d
  - But if the solutions are dense, this may be faster than breadth-first search
- Space
  - O(bm)…linear space
- Optimal
  - No

Depth-first - Exercice

Travel in Romania (same questions as before with as few kilometers as possible)



Depth-Limited Search

- A variation of depth-first search that uses a depth limit
  - Alleviates the problem of unbounded trees
  - Search to a predetermined depth l
  - Nodes at depth l have no successors
- Same as depth-first search if l = ∞
- Can terminate for failure and cutoff

Depth-Limited Search

```
function DEPTH-LIMITED-SEARCH(problem,limit)
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]),problem,limit)
end function
function RECURSIVE-DLS(CURRENT,problem,limit)
  cutoff-occurred? ← FALSE
  if GOAL?[problem](STATE(CURRENT)) then
    return CURRENT
  else if DEPTH(CURRENT) = limit then
    return CUTOFF
  else
    for each SUCCESSOR in EXPAND(CURRENT,problem) do
      result ← RECURSIVE-DLS(SUCCESSOR,problem,limit)
      if result = CUTOFF then
        cutoff-occurred? ← TRUE
      else if result ≠ FAILURE then
        return result
      end if
    end for
  end if
  if cutoff-occurred? then return CUTOFF else return FAILURE
end function
```

Depth-Limited Search

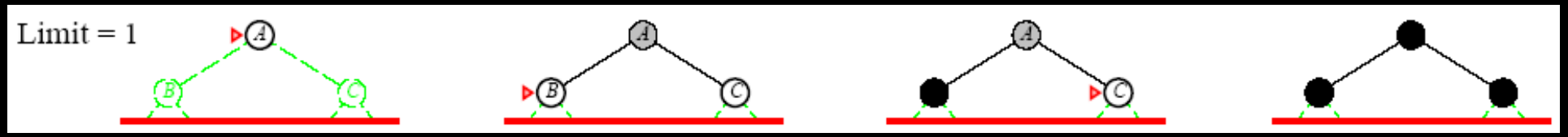
- Complete
  - Yes if l < d
- Time
  - $O(b^l)$
- Space
  - O(bl)
- Optimal
  - No if l > d

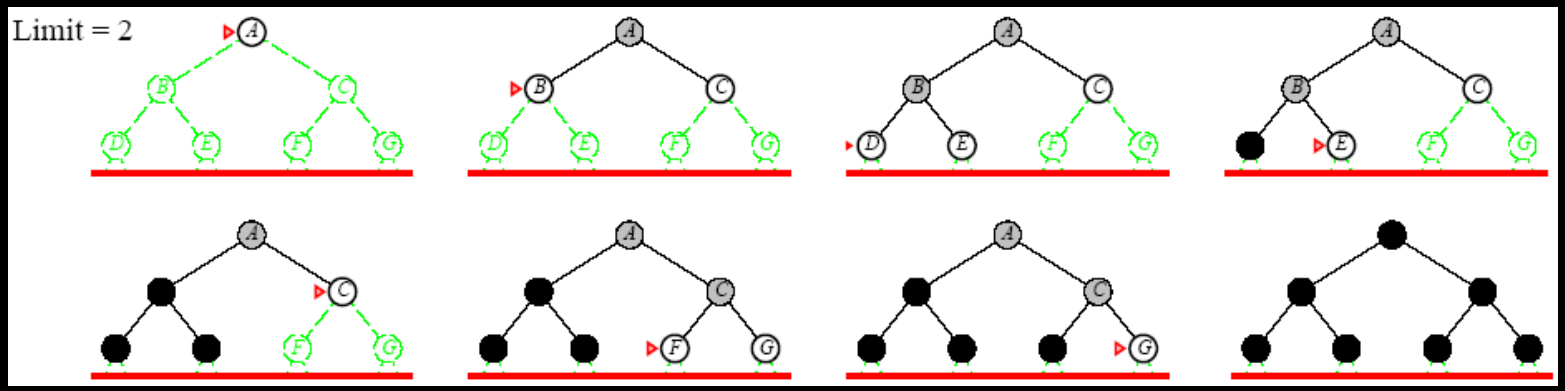
Iterative Deepening Search

- Iterative deepening depth-first search
  - Uses depth-first search
  - Finds the best depth limit
    - Gradually increases the depth limit; 0, 1, 2, … until a goal is found

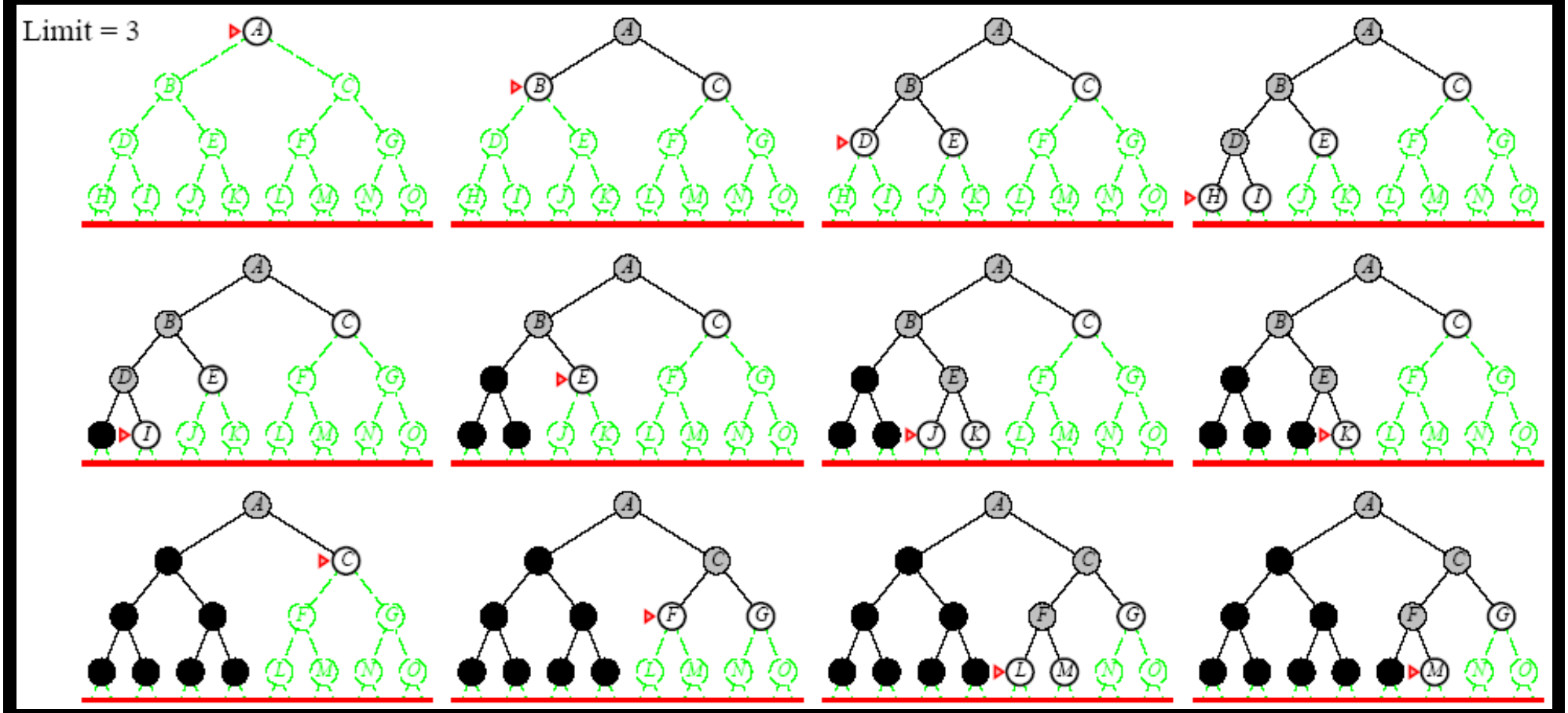
```
function ITERATIVE-DEEPENING-SEARCH(problem)
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem,depth)
    if result ≠ CUTOFF then return result
  end for
end function
```

Iterative Deepening Search

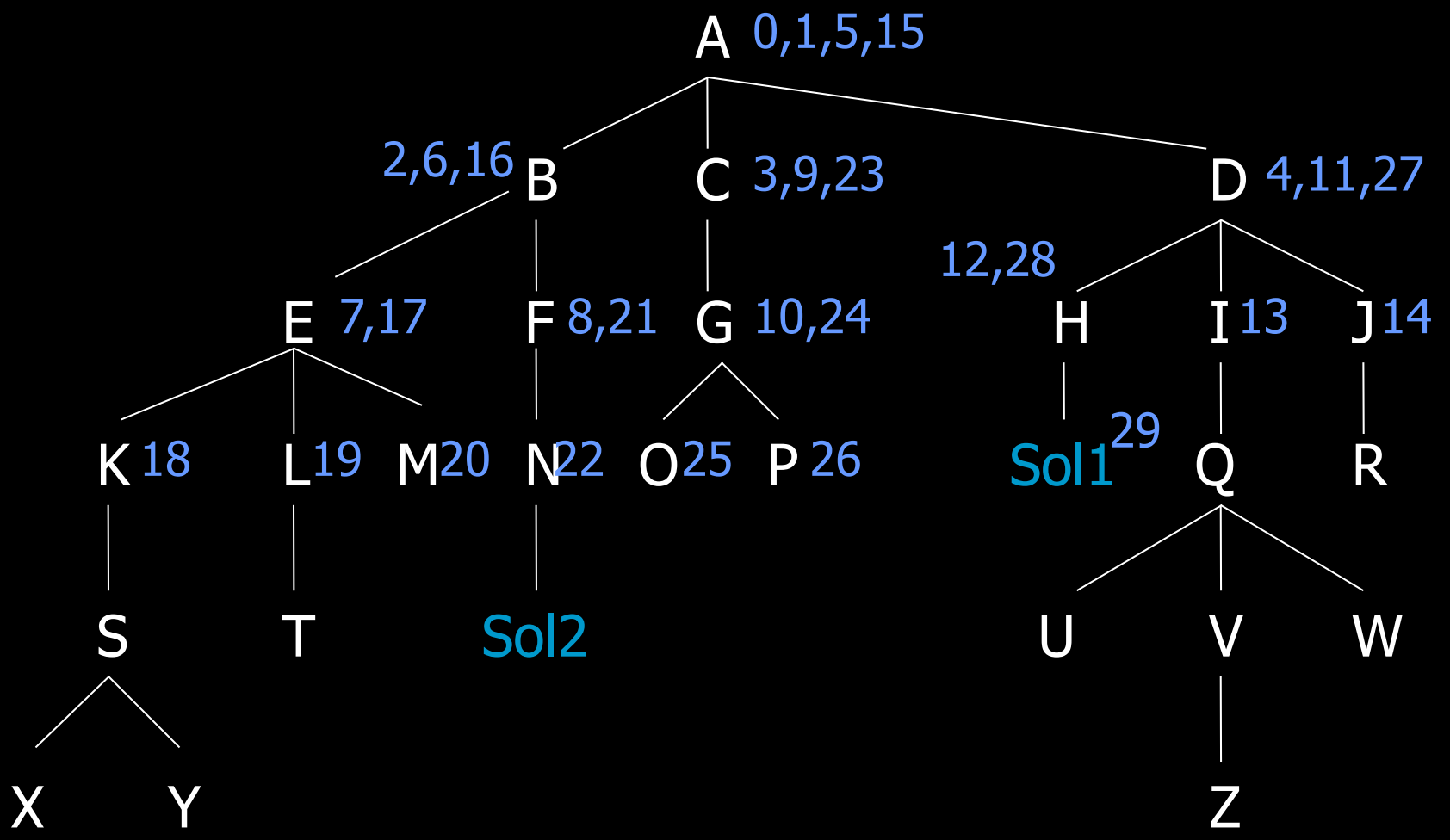




Iterative Deepening Search



Iterative deepening IDS



Iterative Deepening Search

- Complete
  - Yes
- Time
  - $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space
  - O(bd)
- Optimal
  - Yes if step cost = 1
  - Can be modified to explore uniform cost tree

Lessons From Iterative Deepening Search

- Faster than BFS even though IDS generates repeated states
  - BFS generates nodes up to level d+1
  - IDS only generates nodes up to level d

- In general, iterative deepening search is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

(the redundancies are less expensive than developing the last levels of the tree)

Decomposing the pb (OR problem reduction)

Problem Reduction (is the pb decomposable?)

Sometimes problems only seem hard to solve.

- A hard problem may be one that can be reduced to a number of simple problems...and, when each of the simple problems is solved, then the hard problem has been solved.

This is the basic intuition behind the method of problem reduction.

Example: how to get to London?

Planning a trip to London

- you probably don't want to search through all the possible sequences of actions that might get you to London.

- You're more likely to **decompose the problem** into simpler ones - such as getting to the station, then getting a train to London.

- There may be **more than one possible way** of decomposing the problem - an alternative strategy might be to get to the airport, fly to Heathrow, and get the tube from Heathrow into London.

- These different possible plans would have different costs (and benefits) associated with them, and you might have to choose the **best plan**.

Problem Reduction

- The simple state-space search techniques described in the above slides could all be represented using a tree where each successor node represents an alternative action to be taken.
- The graph structure being searched is referred to as an **OR graph**.
- In OR graph we want to find a single path to a goal.
- To represent problem reduction techniques we need to use an AND-OR graph/tree.

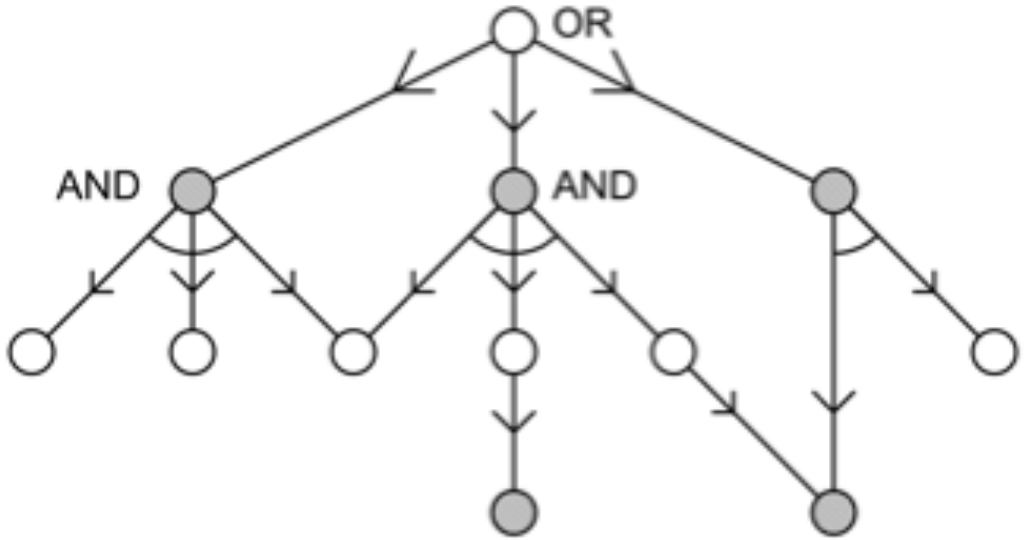
Problem statement

Given

- A goal
- Operators to decompose the goal into sub-goals
- Elementary goals (facts)

Decompose the problem into smaller simplest problem until reaching the elementary goals.

AND/OR graph representation



- Decomposition into sub-problems
- (P_i, O, P_{sol})
- AND/OR graph
- Solved node
- Unsolvable node
- Problem solution

You can have:

AND nodes whose successors must all be achieved: One AND arc may point to a number of successor nodes, all of which must be solved in order for the arc to point to a solution.

OR nodes where one of the successors must be achieved (ie, they are alternatives): several arcs may emerge from a single node, indicating the variety of ways in which the original problem might be solved.

Tower of Hanoï example

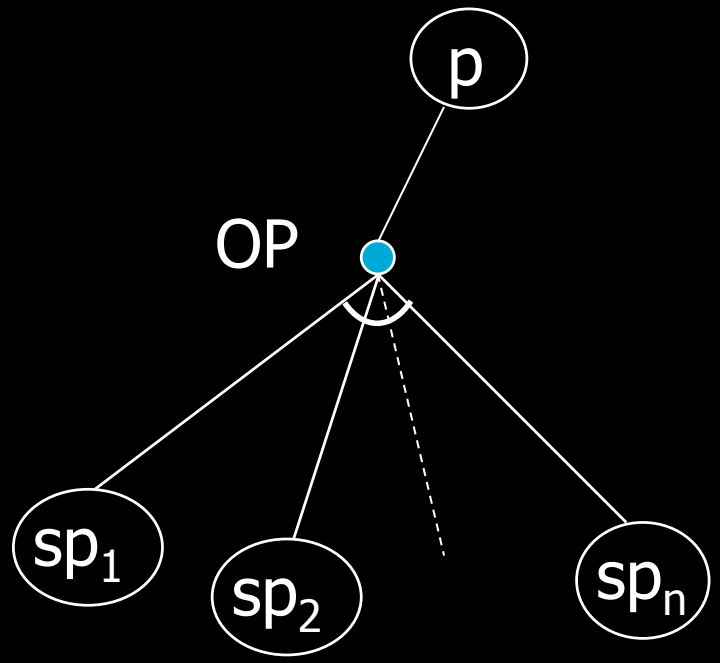
- **Pb**: 64 size (in the original pb) ordered disks occupy one of 3 pegs and must be transferred to one of the other pegs. But, only one disk can be moved at a time; and a larger disk may never be placed on a smaller disk.

- We start at 3 size…

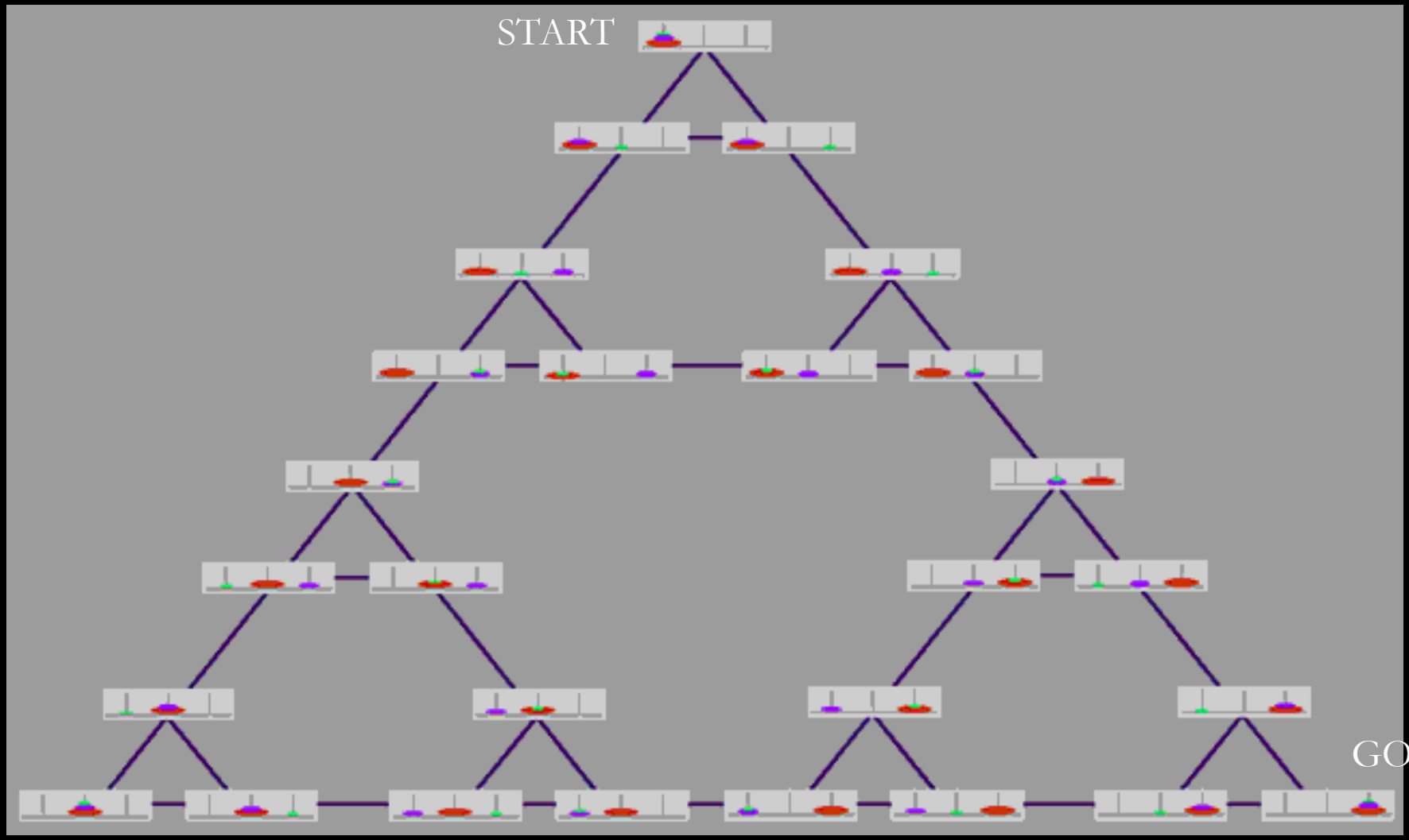
Notations

- Let OP : p ← sp1, sp2, …, spn

- The spi are AND-brothers
- OP is their AND-parent
- p is their OR-parent



State Space for the *3* Disk Tower of Hanoi Problem



About the Hanoï state space

- Recall that in state space search the generators correspond to moves in the state space.
- Thus, the two states below the top state in the triangle of states corresponds to the movement of the smallest disk either to the rightmost peg or to the middle peg.
- The shortest solution to this problem corresponds to the path down the right side of the state space.

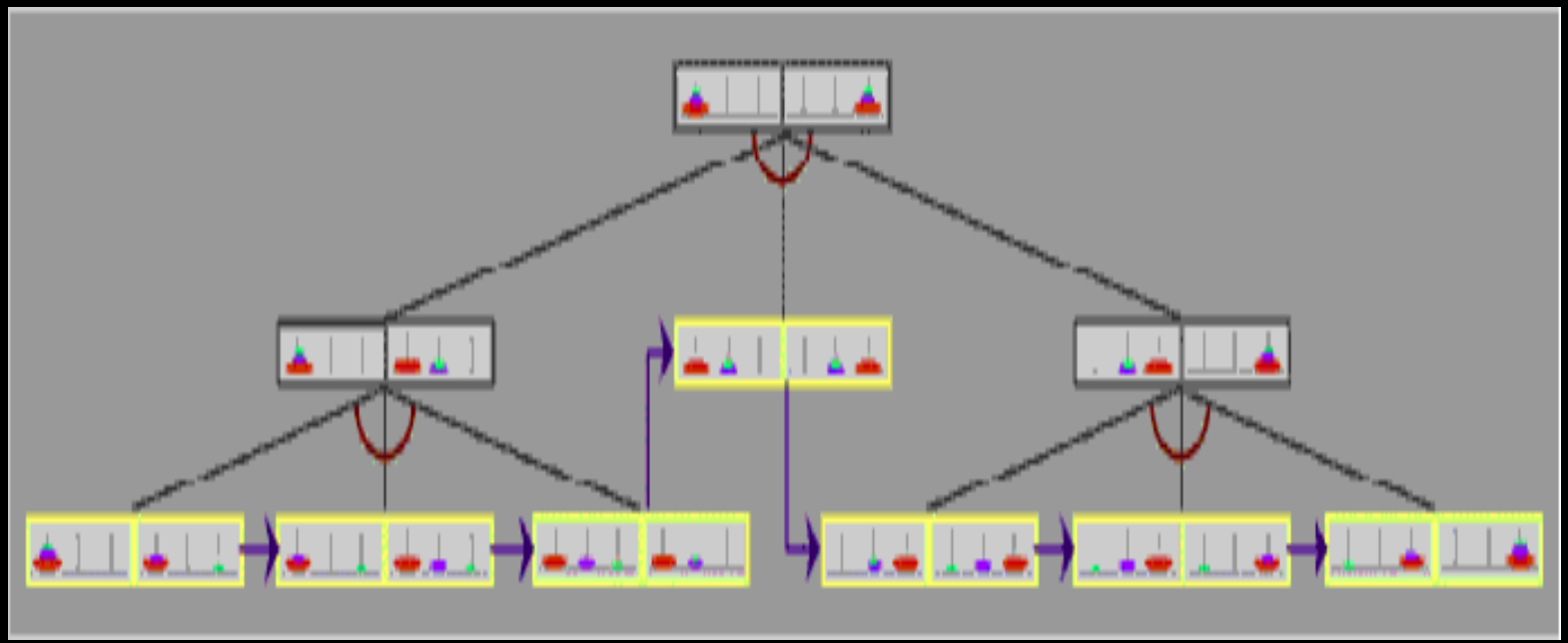
Problem representation with an AND/OR graph

- In problem reduction, searching the problem space consists of an AND/OR graph of (partial) state pairs. These pairs are referred to as (sub)problems.
- The first element of the pair is the starting state of the (sub)problem

  and the second element of the pair is the goal state (sub)problem.

Problem Reduction

- The symmetry of the state space shown above may have led you to suspect that the Tower of Hanoi problem can be elegantly solved using the method of problem decomposition.
- The AND tree that solves the 3 disk problem is shown below
- The yellow states are primitive problems



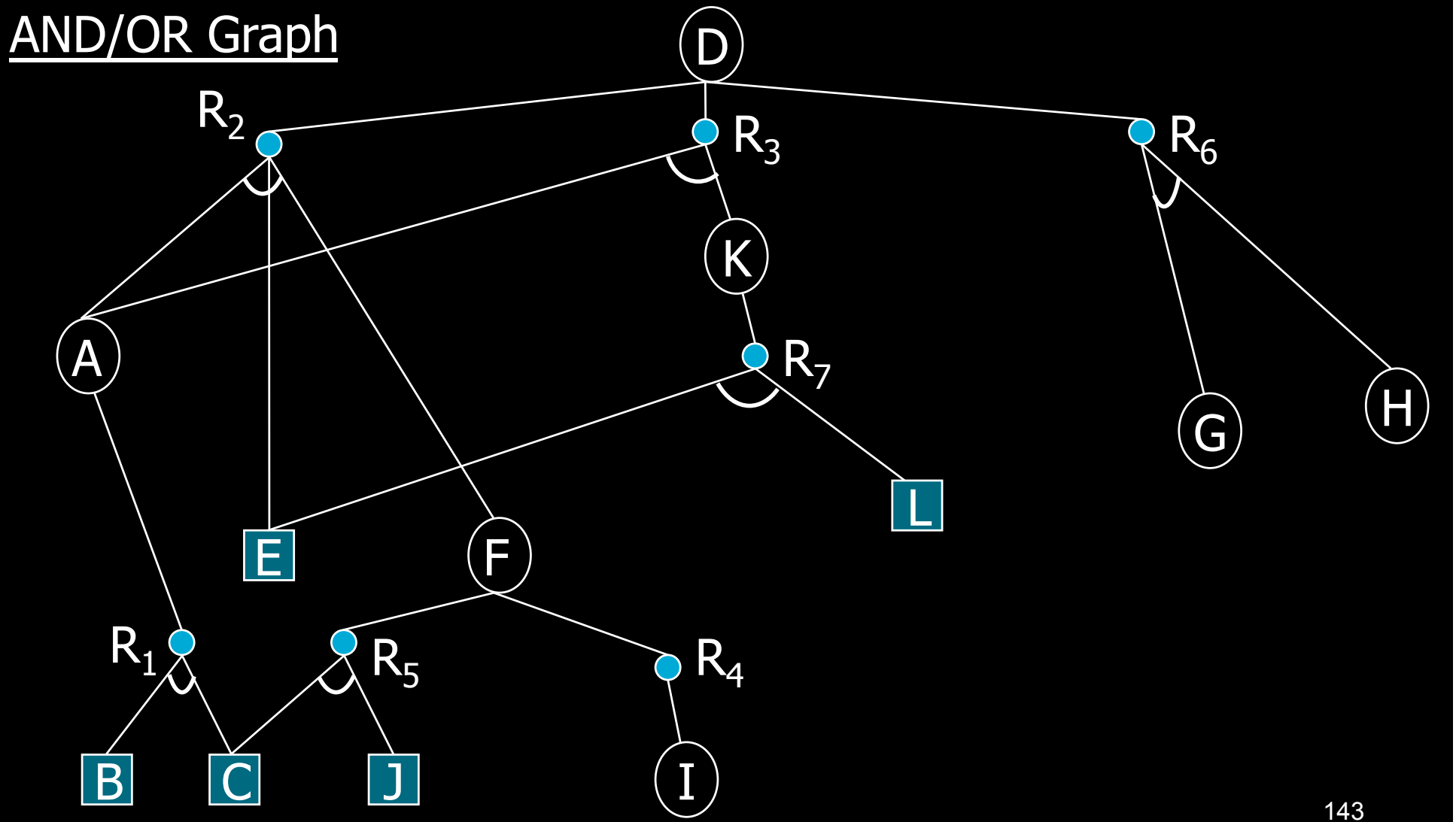
Exercise

- Elementary goals : B, C, E, J, L
- Decomposition operator

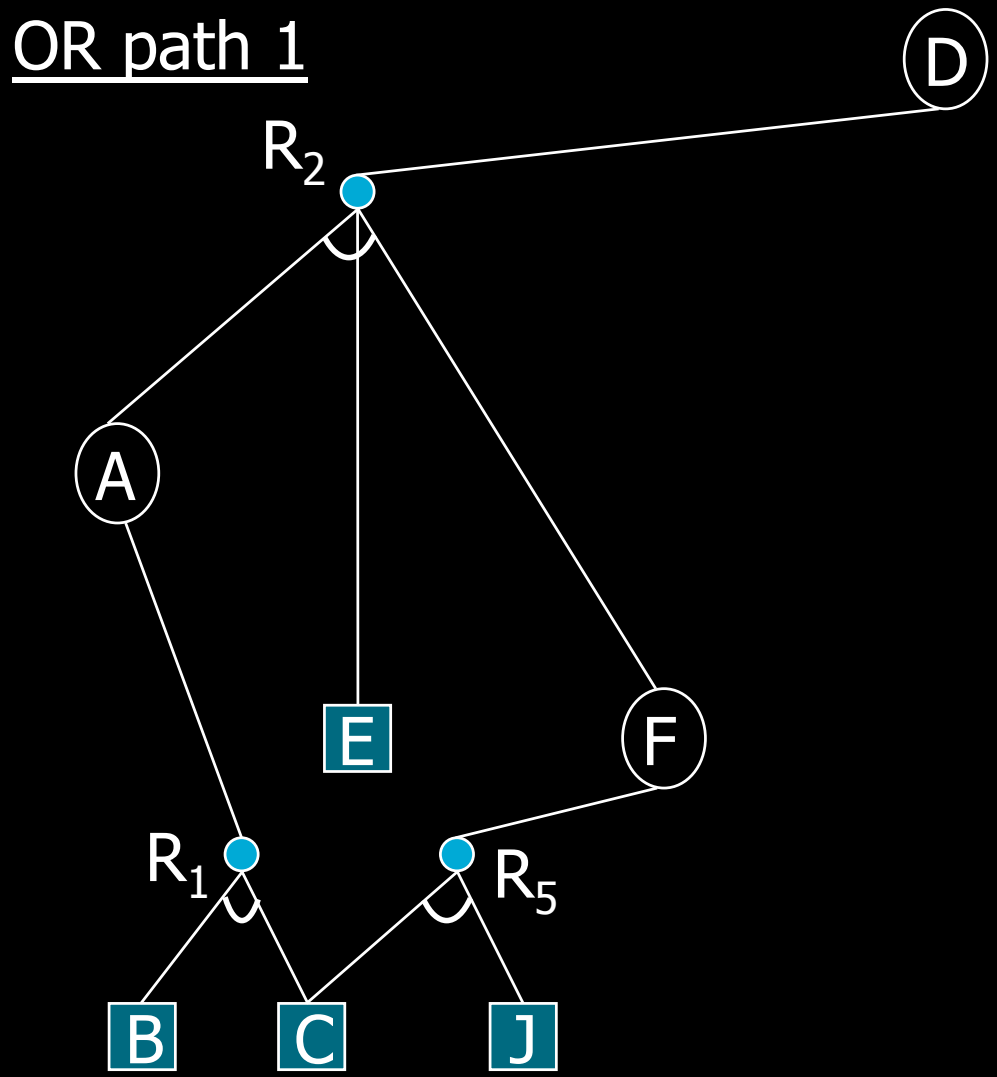
```
R1 : A -> B,C
R2 : D -> A,E,F
R3 : D -> A,K
R4 : F -> I
R5 : F -> C,J
R6 : D -> G,H
R7 : K -> E,L
```

- Solve D
- Draw the AND/OR graph
- Draw the solution subgraphs (OR paths)

Example - solution

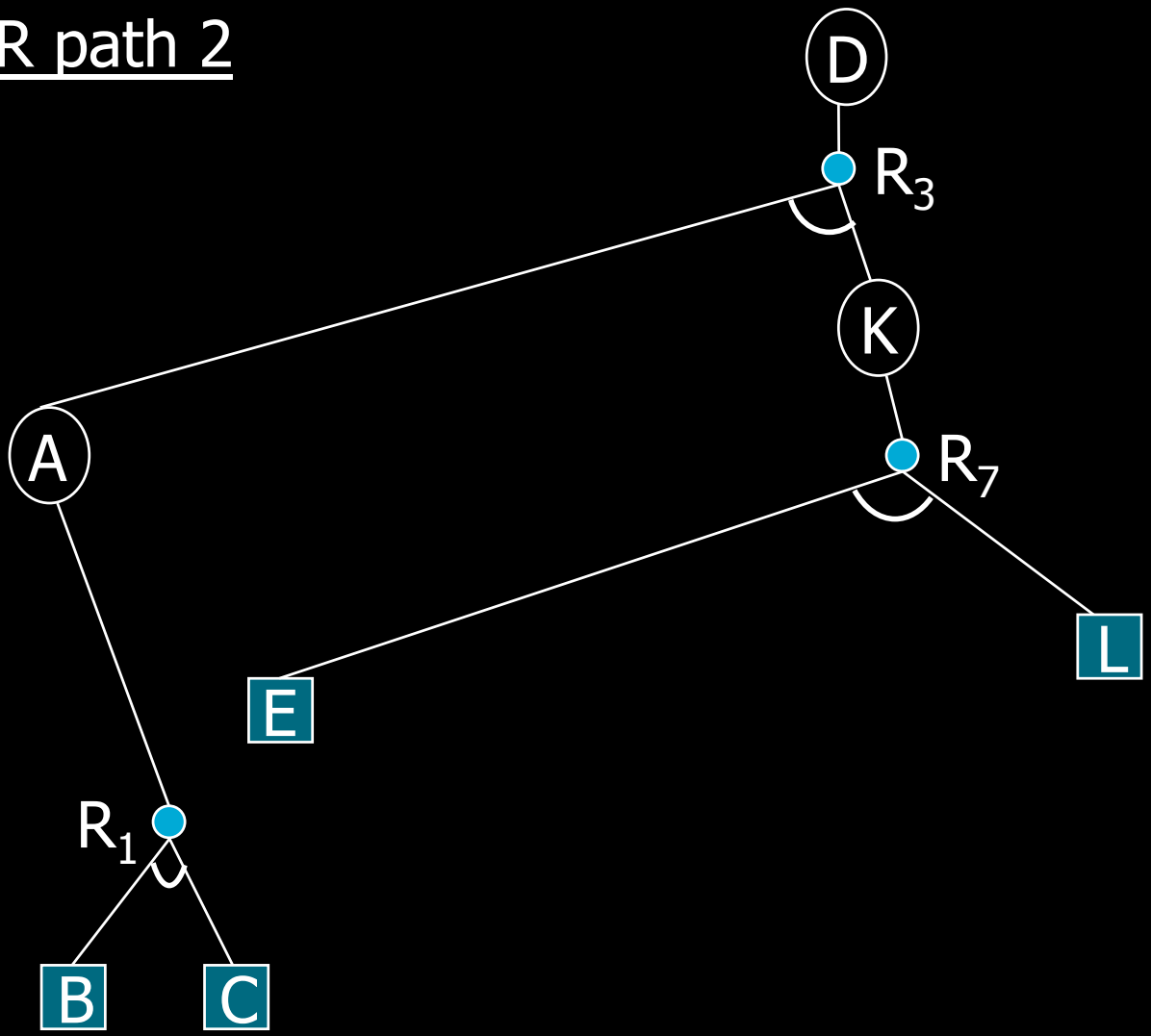


Solution



sol

OR path 2



Search in AND/OR graphs

NB: solutions are sub-graphs (OR-PATH)

- With graph: we construct an implicit sub-graph corresponding to the problem
- AND/OR search graph and OR-path as a solution
- With backtracking: we keep only a OR-path in memory.

Notations for the graph algorithm

- **FRINGE**: set of nodes (sub-problems) that still need to be delt with.
- **CLOSED**: set of nodes already seen.
- p0: initial problem
- Gr-Sol: partial solution (OR-path from the root)
- DURING: subset of FRINGE which contains the current leaves of Gr-Sol

AND-OR search with a graph

FRINGE $\leftarrow \{p_0\}$; CLOSED $\leftarrow \{\ \}$; Gr-Sol $\leftarrow$ OR-path(p_0)
repeat
  choice 1: choice of Gr-Sol[1]; DURING $\leftarrow$ leaves of Gr-Sol
  choice 2: choice of a node p not primitive in DURING and not in CLOSED
  Develop p
  Take[2] p from FRINGE and DURING, and add it to CLOSED
  for all Op s.t. $p \rightarrow sp_1, \ldots, sp_k$ **do**
    Create a AND-node OP (successor of p)
    Create k OR-nodes $sp_1, \ldots, sp_k$
    FRINGE $\leftarrow$ FRINGE $\cup \{sp_1, \ldots, sp_k\}$
  end for
until DURING $= \emptyset$ **OR** all pbs in DURING are primitives
if DURING $= \emptyset$ **then return** FAIL

[1] Depends on the heuristic
[2] Can be done depth of breadth or best-first

NB: we can also use pointers on the parents

Exercise

S → A ; S → B ; A → C,D ; B → D,I ; C → F ; F → H ; H → t1 ; F → G ; G → H,I ; I → t2 ; D → G ; D → F

1. Draw the AND/OR-graph, S is the initial problem, t1 and t2 are primitives.

2. Show the evolution of FRINGE, DURING, CLOSED for a depth-first and a breadth-first search.

AND/OR search backtracking algorithm

- 2 types of backtrack
  - backtrack on ORs (similar to standard depth-first search): when one branch of the OR fails, we need to try another one or gives a failure if there are no more branch.
  - backtrack on ANDs: if one of the sub-problem fails, the whole branch fails (there is not need to try the AND-brothers)

- At each choice point (OR branch), one needs to chose:
  - the operator to apply
  - The order in which the sub problems should be explored.

Backtracking Algorithm

```
function SEARCH-AND/OR(pb)
  if TERMINAL(pb) then return EMPTY
  if IMPASSE(p) OR DEPTH(pb)>Limit then return FAIL (2)
  List-op ← OP-APPL(pb) (3)
  repeat
    if List-op = ∅ then return FAIL
    Op ← FRONT(List-op)
    {sp_1, ..., sp_n} ← OP(p)
    List-op ← QUEUE(List-op)
    ESC ← FALSE
    for i from 1 to n AND not(ESC) do
      DEPTH(sp_i) ← DEPTH(p)+1
      Gr_i ← SEARCH-AND/OR(sp_i)
      if Gr_i=FAIL then ESC ← TRUE
    end for
  until ESC=FALSE
  return OR-path(p,Op,and(Gr_1,...,Gr_n)))
end function
```

Remarks

- Simple implementation
- The algorithm may not terminate (cycles) ⇒ need to include a loop detection or limit the recursion depth arbitrarily
- Line (2) is optional but can avoid to explore a partial solution known to be bad
- We can add a heuristic on Line (3) to "smartly" sort the possible operators

Outline

1. Intro
2. AI Problem Representation
3. Problem solving: Uninformed Search
    1. Depth-first search, Breadth-first search, Uniform search, Iterative deepening…
    2. Problem decomposition (AND/OR tree)
4. **Heuristic (informed) Search**
    1. A*, AO*…
5. Game Playing

Search Algorithms

- Blind search — BFS, DFS, uniform cost
  - no notion concept of the “right direction”
  - can only recognize goal once it’s achieved

- Heuristic search
  we have rough idea of how good various states are, and use this knowledge to guide our search

Best-first search

- Idea: use an evaluation function f(n) for each node
  - Estimate of desirability
- Expand most desirable unexpanded node
- Implementation: fringe is queue sorted by decreasing order of desirability
  - Greedy search
  - A* search

Heuristic

- Webster's Revised Unabridged Dictionary (1913) (web1913)
  - Heuristic \Heu*ris"tic\, a. [Greek. to discover.] Serving to discover or find out.
- The Free On-line Dictionary of Computing (15Feb98)
  - heuristic 1. <programming> A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee feasible solutions and are often used with no theoretical guarantee. 2. <algorithm> approximation algorithm.
- From WordNet (r) 1.6
  - heuristic adj 1: (computer science) relating to or using a heuristic rule 2: of or relating to a general formulation that serves to guide investigation [ant: algorithmic] n : a commonsense rule (or set of rules) intended to increase the probability of solving some problem [syn: heuristic rule, heuristic program]

Informed Search

- Add domain-specific information to select the best path along which to continue searching
- Define a heuristic function, h(n), that estimates the "goodness" of a node n.
- Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
- The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal

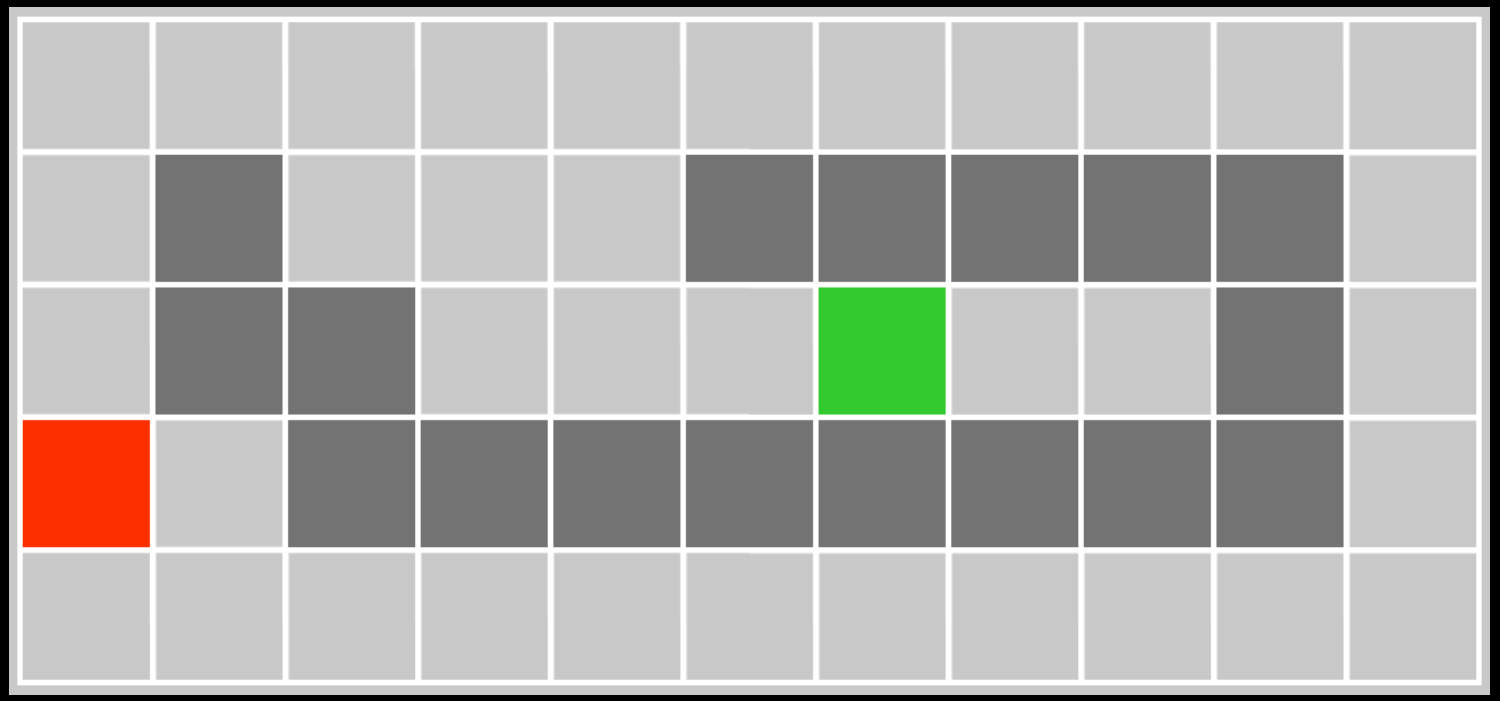
Greedy Search

- Gready best-first: tries to expand the node that is the closest to the goal because it is likely to lead to a solution quickly f(N) = h(N)

```
FRINGE ← {initiale-node}
while FRINGE ≠ ∅ do
   CURRENT ← REMOVE-FRONT(FRINGE)
   if GOAL(CURRENT) then return PATH_TO(CURRENT)
   for all op ∈ List_op(CURRENT) do
      SUCCESSOR←op(CURRENT)
      FRINGE ← INSERT-ORDERED(FRINGE,SUCCESSOR) [1]
   end for
end while
return FAILURE
```

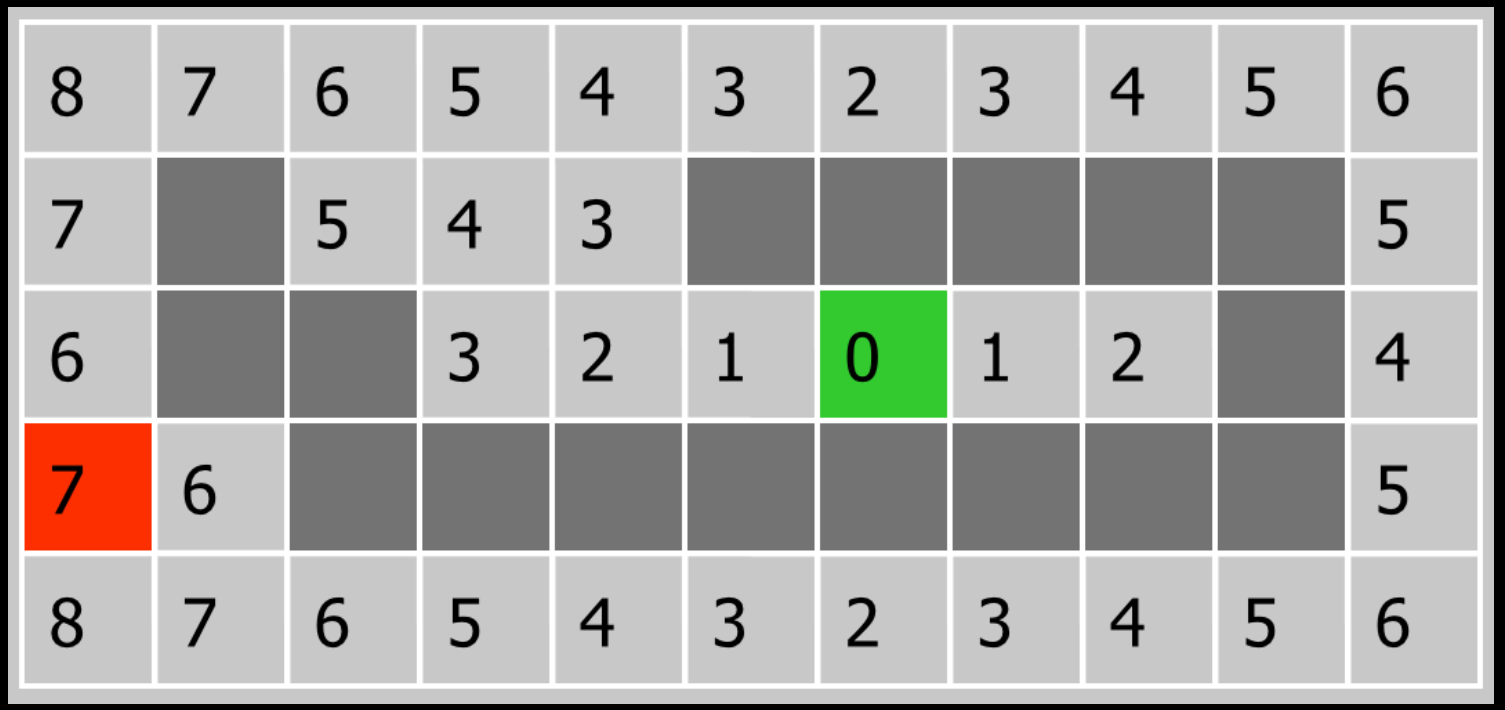
[1] Order the nodes in the fringe in increasing values of h(n)

Ex: Robot Navigation



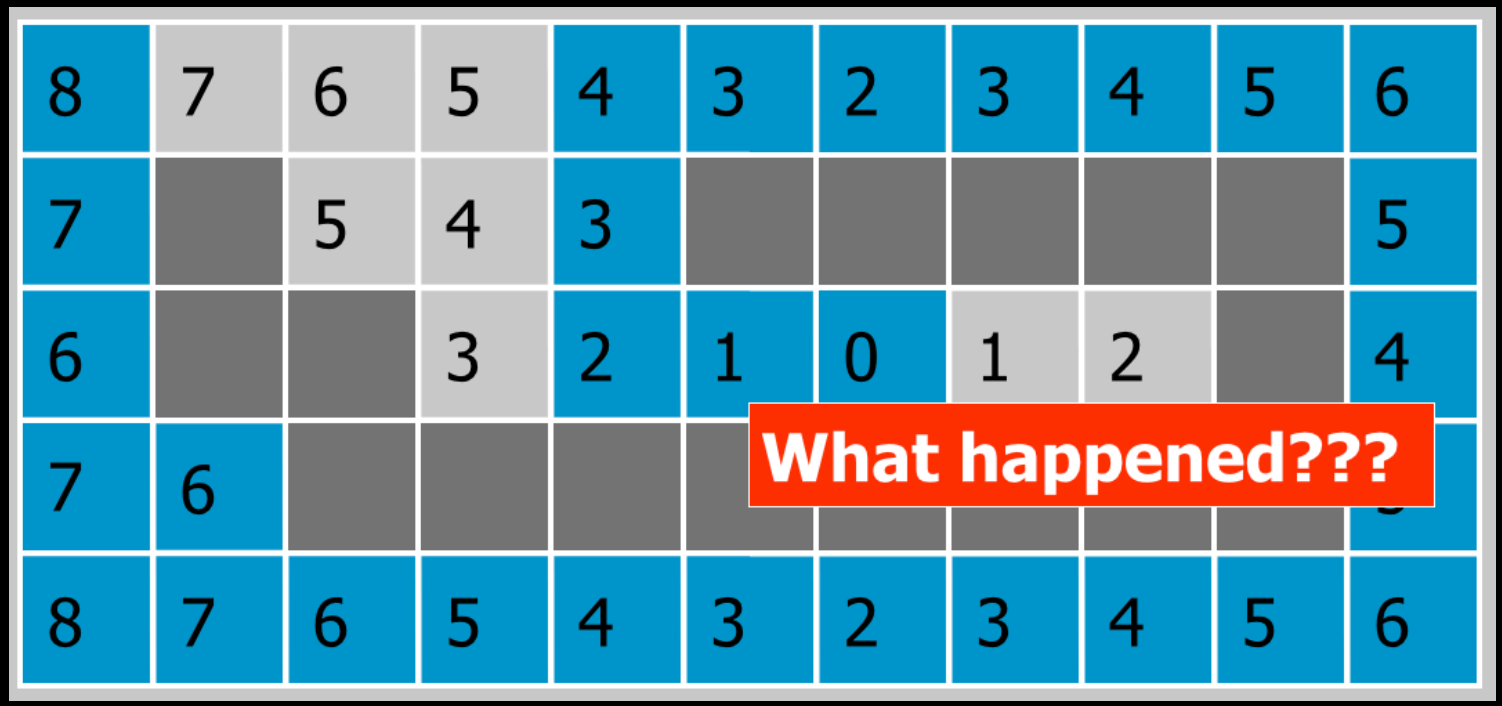
Ex: Robot Navigation

- f(n) = h(n) with h(n) = Manhattan distance to the goal



Ex: Robot Navigation

- f(n) = h(n) with h(n) = Manhattan distance to the goal



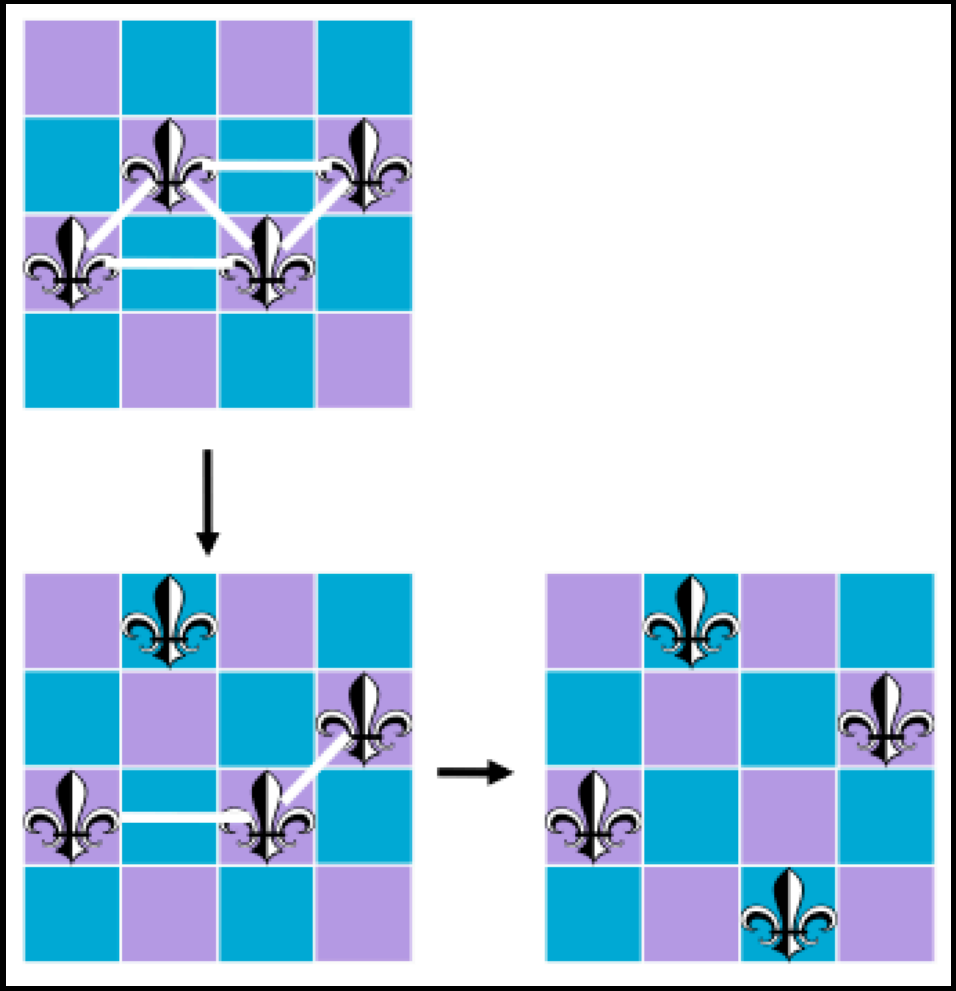
Greedy Search

- f(N) = h(N) → greedy best-first
- Is it complete?
  - If we eliminate endless loops, yes
- Is it optimal?
  - In general, no

About Heuristics

- Heuristics are intended to orient the search along promising paths
- The time spent computing heuristics must be recovered by a better search
- After all, a heuristic function could consist of solving the problem; then it would perfectly guide the search
- Deciding which node to expand is sometimes called meta-reasoning
- Heuristics may not always look like numbers and may involve large amount of knowledge

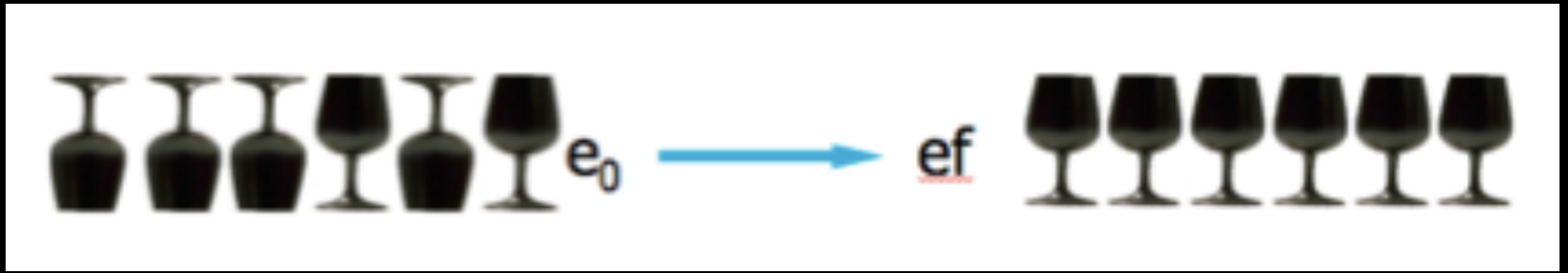
Example with the n queens pb



A possible heuristic?

- From one configuration, we can move a queen to have as few attacks as possible
- We move the most attacked one.

Exercise: the glass pb



- One possible action: flip 2 adjacent glasses
- Show (part of) the search space using
  - f1 = “the number of misplaced glasses” •
- Show (part of) the search space with
  - f 2 = “the distance between the two furthest misplaced glasses” Nb: $f(e_0)=4, f(e_f)=0$

More informed search

- We kept looking at nodes closer and closer to the goal, but were accumulating costs as we got further from the initial state
- Our goal is not to minimize the distance from the current head of our path to the goal, **we want to minimize the overall length of the path to the goal!**

- Let g(N) be the cost of the best path found so far between the initial node and N

- f(N) = g(N) + h(N)

Ex: Robot Navigation

- f(n) = g(n)+ h(n) with h(n) = Manhattan distance to the goal

8+3	7+4	6+5	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Can we Prove Anything?

- If the state space is **finite** and **we avoid** repeated states,
  - the search is **complete**, but in general is **not optimal**
- If the state space is finite and we do not avoid repeated states,
  - the search is in general **not complete**
- If the state space is infinite,
  - the search is in general **not complete**

Admissible heuristic

- Let h*(N) be the true cost of the optimal path from N to a goal node

- Heuristic h(N) is admissible if:

$$0 \leq h(N) \leq h^*(N)$$

NB: an admissible heuristic is always optimistic

A* Search

- Evaluation function:

$$f(N) = g(N) + h(N)$$

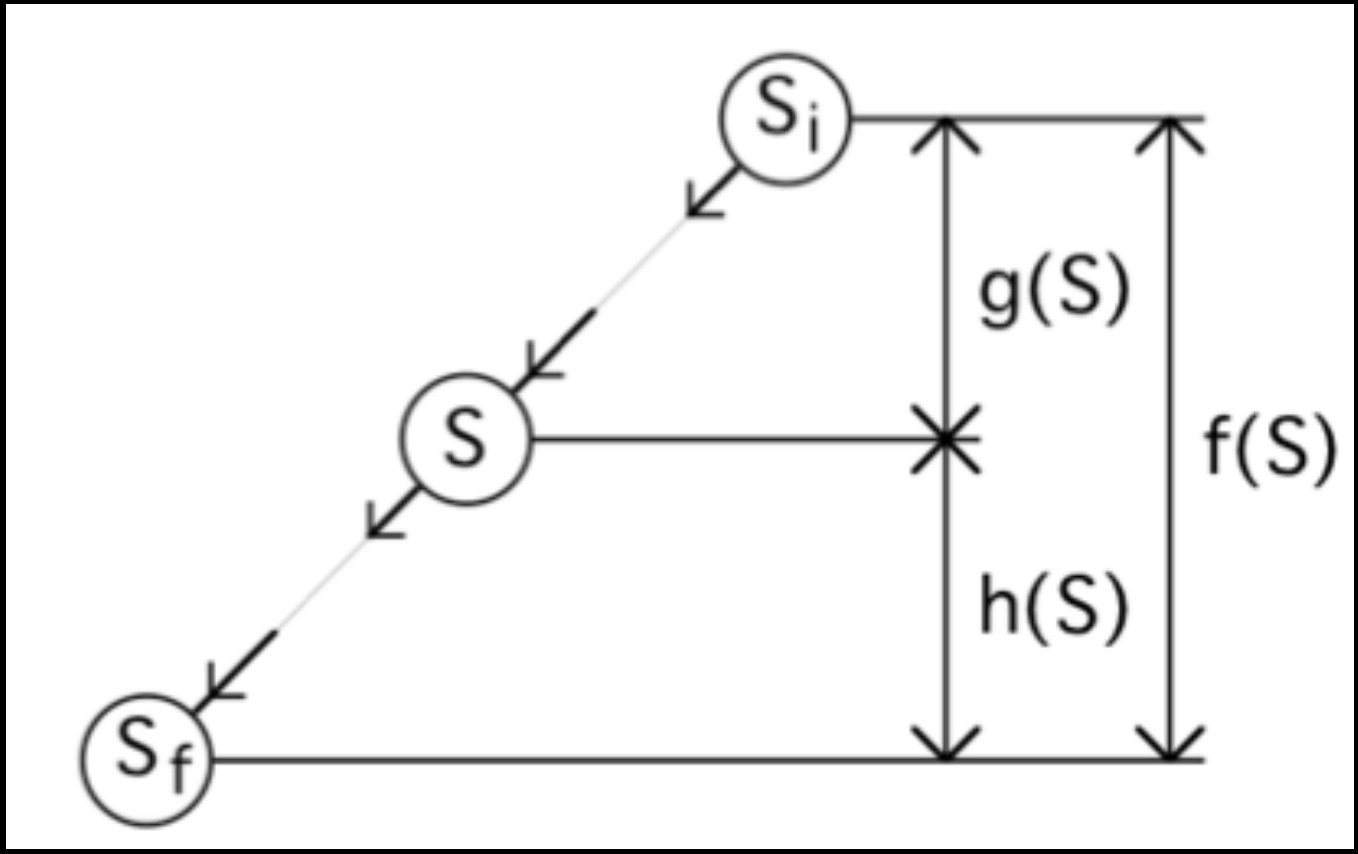
where:

  - g(N) is the cost of the best path found so far to N
  - h(N) is an admissible heuristic

- Then, best-first search with this evaluation function is called **A* search**

- Important AI algorithm developed by Fikes and Nilsson in early 70s. Originally used in Shakey robot.

Components of A*

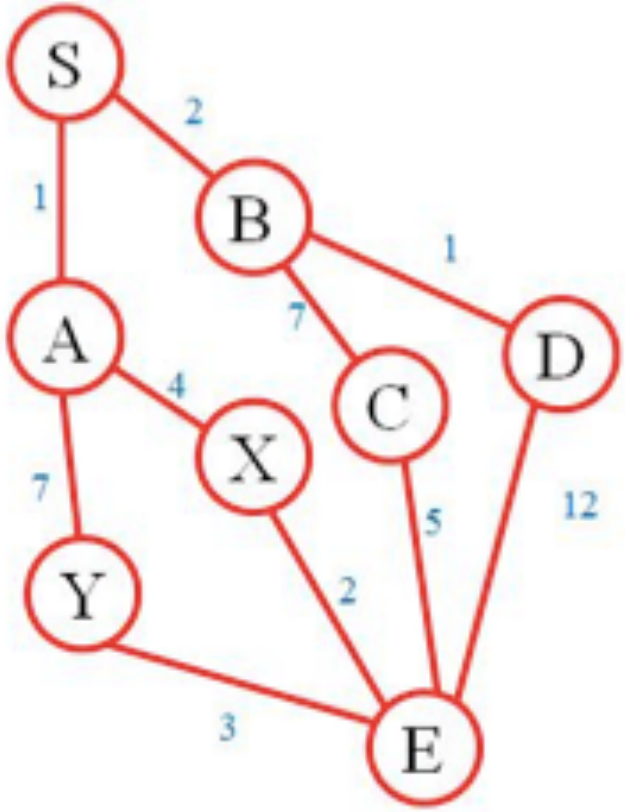


Algorithme A*

FRINGE ← {e0}; CLOSED ← {}
loop
  if FRINGE = ∅ **then return** FAILURE
  CURRENT ← $\underset{X \in \text{FRINGE}}{\text{argmin}}\ (f(X))$ *(X s.t. f(X) is minimum)*
  if CURRENT is a solution **then return** CURRENT
  FRINGE ← FRINGE − {CURRENT}
  CLOSED ← CLOSED ∪ {CURRENT}
  for all op ∈ List_op(CURRENT) **do**
    SUCCESSOR←op(CURRENT)
    if SUCCESSOR ∈ (FRINGE ∪ CLOSED) **AND**
    g(SUCCESSOR) > g(CURRENT) + *cost*(op) [1] **then**
      parent(SUCCESSOR) =CURRENT
      g(SUCCESSOR) = g(CURRENT) + *cost*(op)
      FRINGE ← FRINGE ∪ {SUCCESSOR}
    end if
  end for
end loop

[1] if SUCCESSOR is already seen then the new path to SUCCESSOR must be shorter

Example A*



- Values for h:

  A:5, B:6, C:4, D:15, X:5, Y:8

Expand S

{S,A} f=1+5=6

{S,B} f=2+6=8

Expand A

{S,B} f=2+6=8

{S,A,X} f=(1+4)+5=10

{S,A,Y} f=(1+7)+8=16

Expand B

{S,A,X} f=(1+4)+5=10

{S,B,C} f= (2+7)+4=13

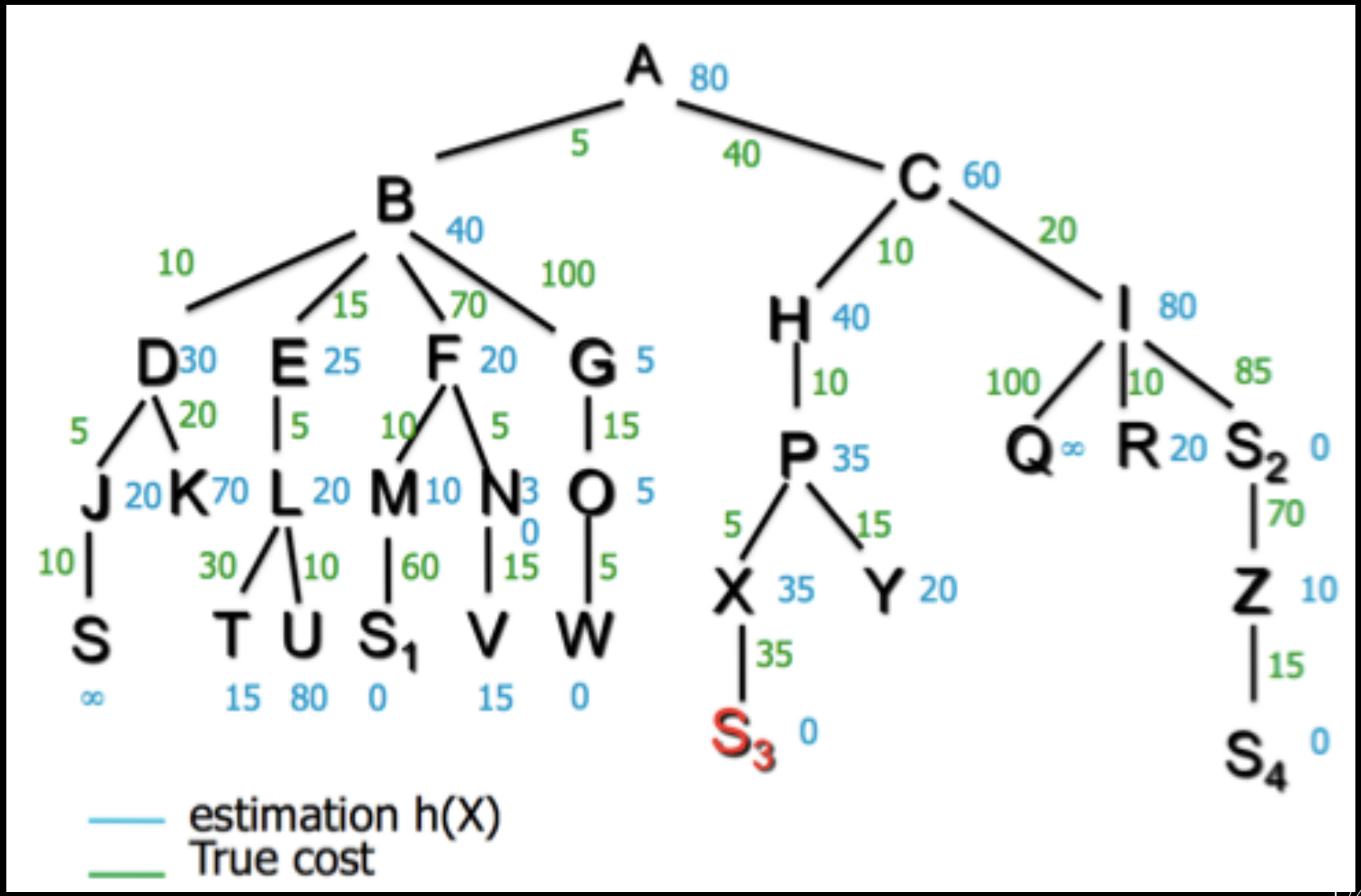
{S,A,Y} f=(1+7)+8=16

{S,B,D} f= (2+1)+15=18

Expand X

{S,A,X,E} is the best path... (costing 7)

Example A* - 2


A 80
5
40
B 40
C 60
10
100
10
20
D 30
E 25
F 20
G 5
H 40
I 80
5
20
5
10
5
15
10
100
10
85
J 20
K 70
L 20
M 10
N 3
O 5
P 35
Q ∞
R 20
S2 0
0
10
30
10
60
15
5
5
15
70
S
T
U
S1
V
W
X 35
Y 20
Z 10
35
15
S3 0
S4 0
∞
15
80
0
15
0
estimation h(X)
True cost

Completeness & Optimality of A*

- Claim 1: If there is a path from the initial to a goal node, A* (with no removal of repeated states) terminates by finding the best path, hence is:
  - Complete
  - Optimal

- requirements:
  - $0 < \varepsilon \leq c(N,N')$
  - Each node has a finite number of successors

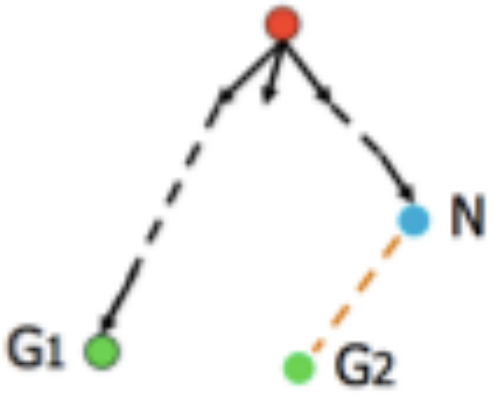
Completeness of A*

- Theorem: If there is a finite path from the initial state to a goal node, A* will find it.

- Proof of Completeness
  - Let g be the cost of a best path to a goal node
  - No path in search tree can get longer than g/ε, before the goal node is expanded

Optimality of A*

- Theorem: If h(n) is admissible, then A* (*using a tree search*) is optimal.



$f(G_1) = g(G_1)$

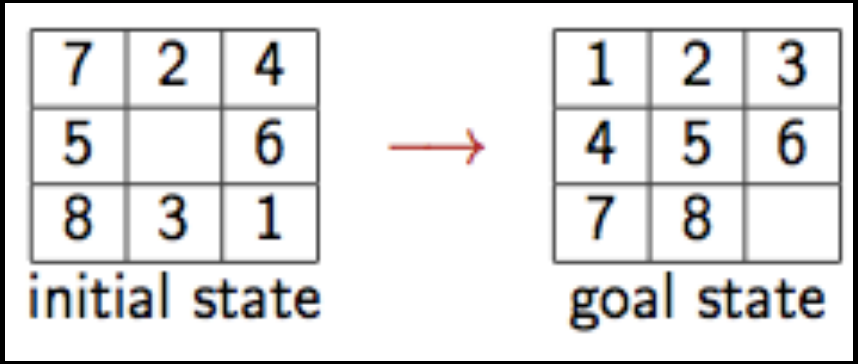
Cost of best path to a goal N

$f(N) = g(N) + h(N) \leq g(N) + h^*(N)$

$f(G_1) \leq g(N) + h(N) \leq g(N) + h^*(N)$

Admissibility of heuristic functions

- Function h(n) that estimates the cost of the cheapest path from node N to goal node
- Example: 8-puzzle

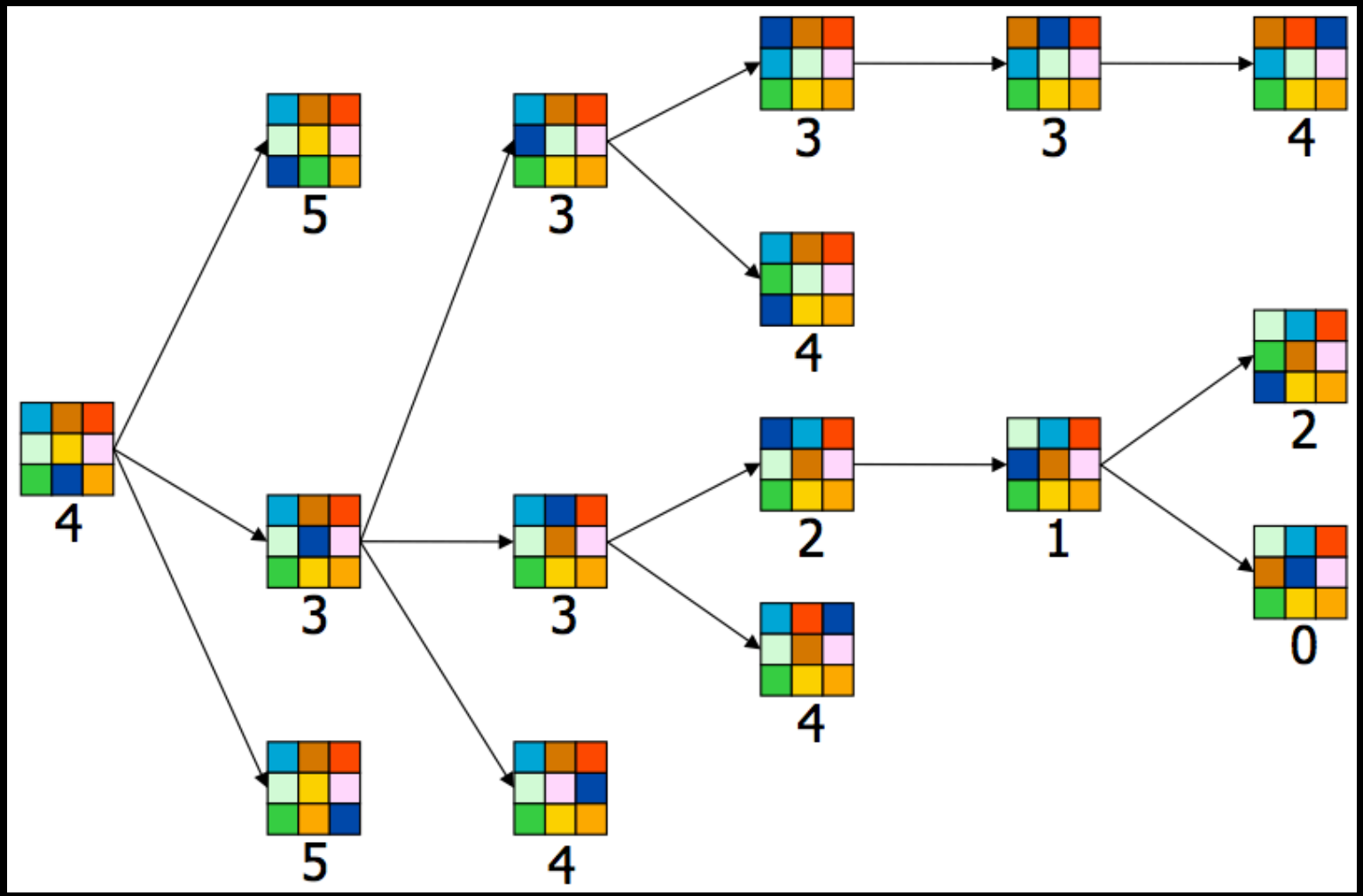


h1(n) = number of misplaced tiles = 6

OR h2(n) = sum of the distances of every tile to its goal position = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14

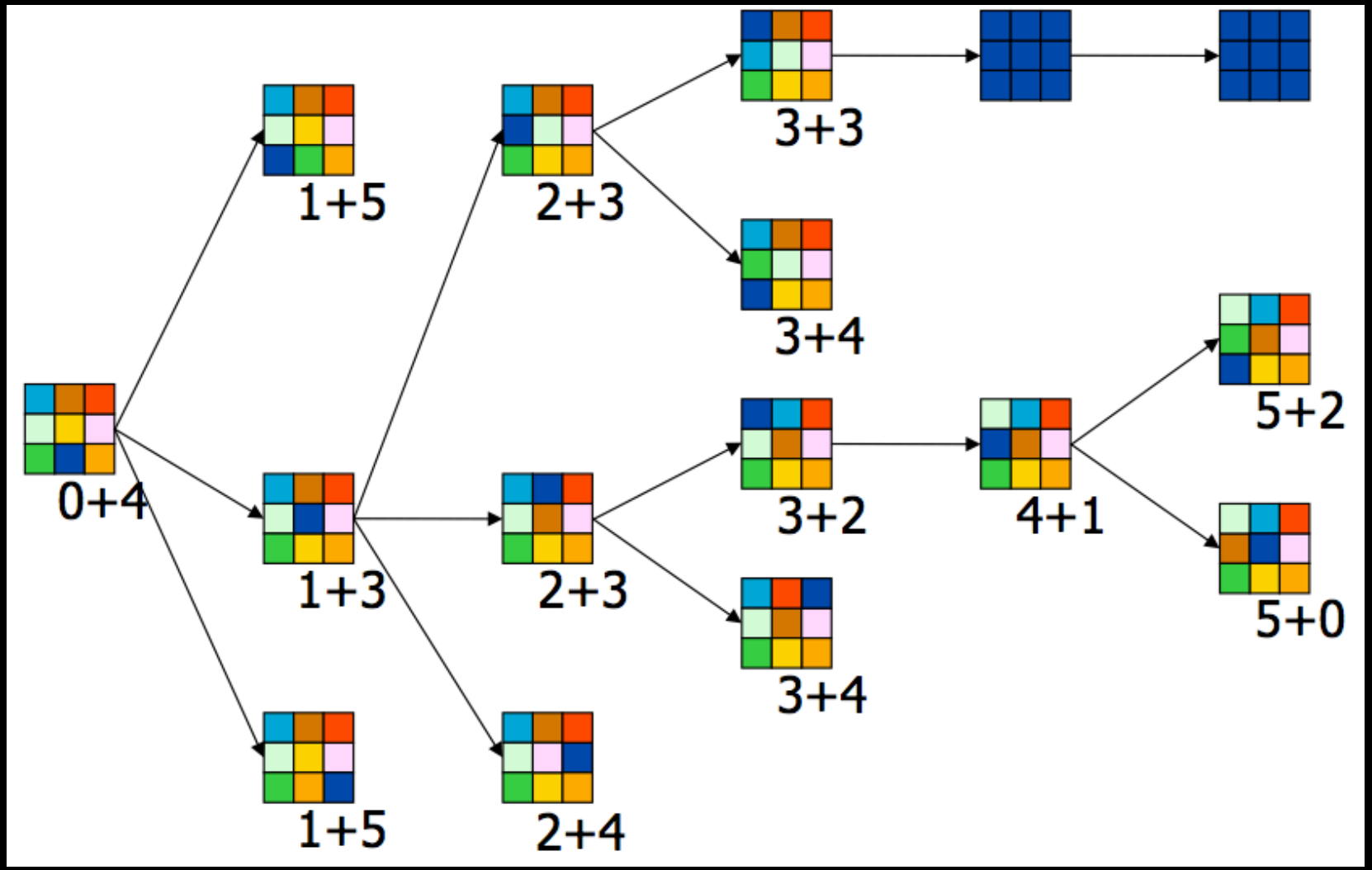
Example: 8-puzzle

- f(n) = h(n) = number of misplaced tiles (no g)



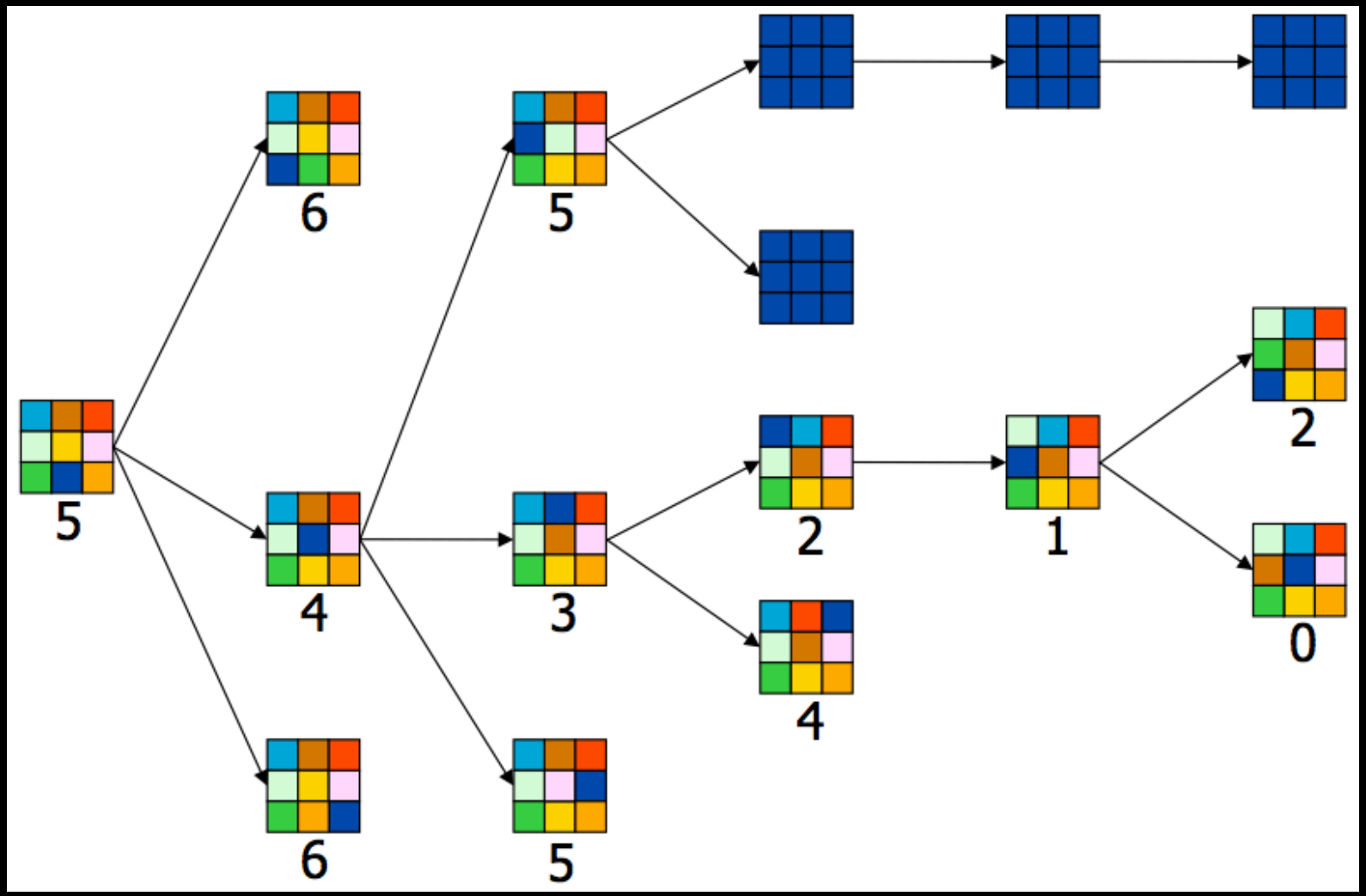
Example: 8-puzzle

- f(n) = g(n) + h(n)



Example: 8-puzzle

- f(n) = g(n) = h(n) = Σ distances of tiles to goal

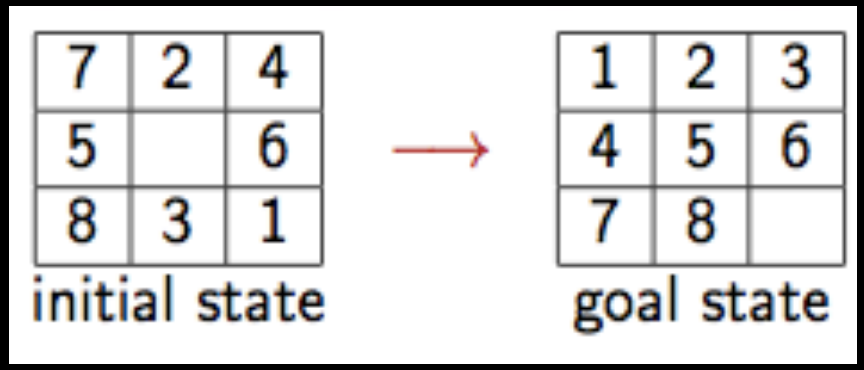


Ex: 8 puzzle

- Another Heuristic:

h(n) = (sum of the distances of each tile to its goal position) + 3 * (sum of score functions for each tile)

Where score function for a non-central tile is 2 if it is not followed by the correct tile in clockwise order and 0 otherwise



h(n) = (2+3+0+1+3+0+3+1) + 3* (2+2+2+0+2+2+0+2)

= 49

Ex: 8 puzzle

- What can we say about the previous heuristic functions?

- $h_1(N)$ = number of misplaced tiles

→ is admissible

- $h_2(N)$ = sum of distances of each tile to goal

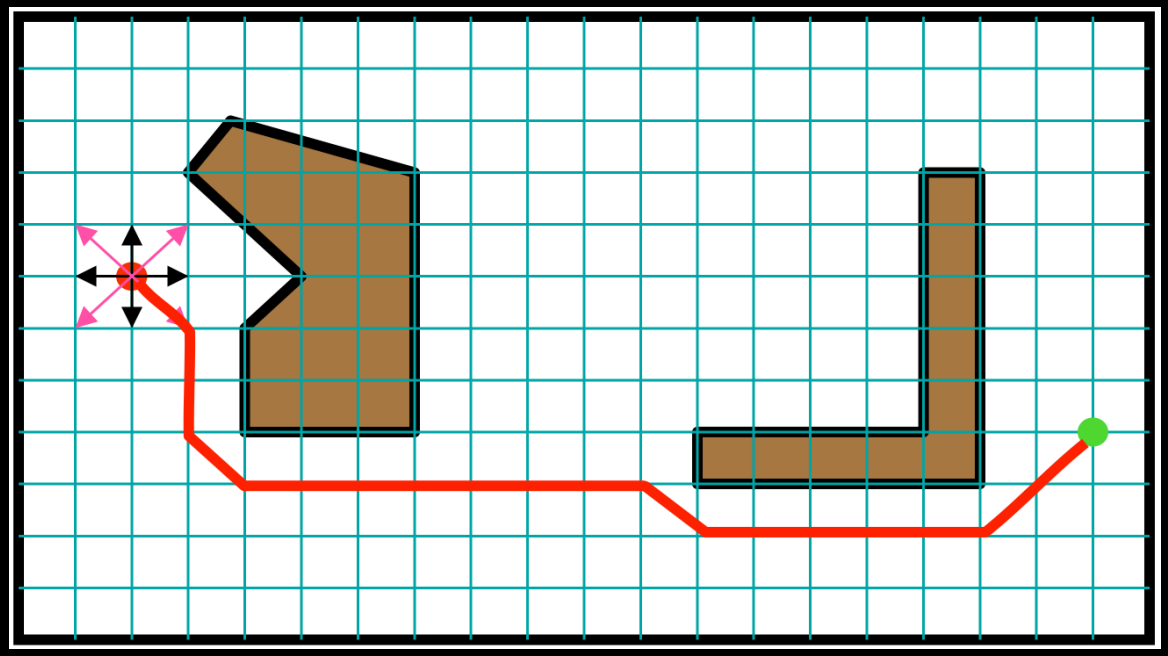
→ is admissible

- $h_3(N)$ = (sum of distances of each tile to goal) +3× (sum of score functions for each tile)

→ is not admissible

Possible evaluation function for the robot path planning pb

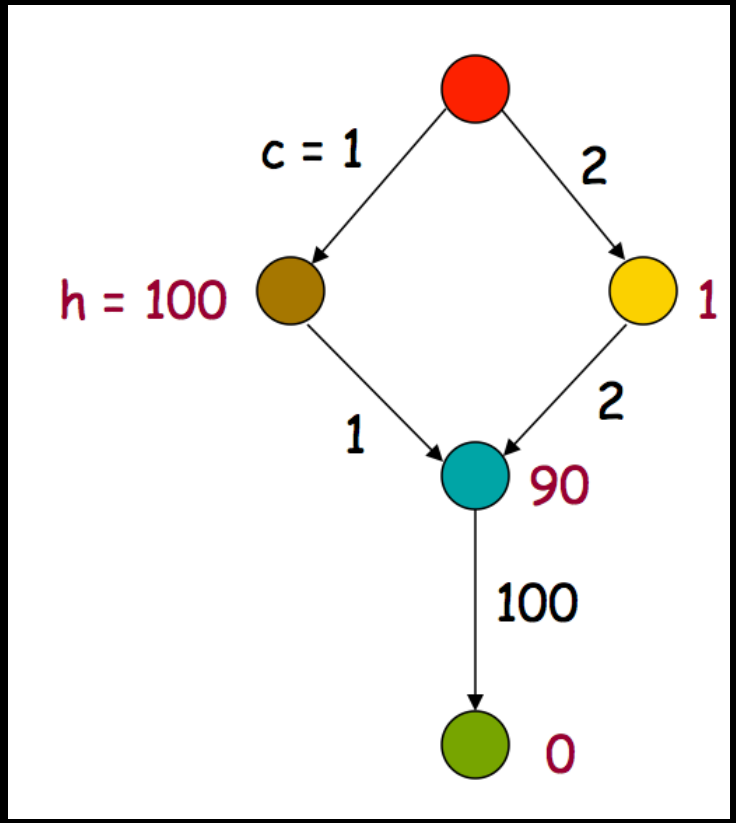
- f (n) = g (n) + h(n), with h(n) = straight-line distance from N to goal



- Cost of one horizontal/vertical step = 1
- Cost of one diagonal step = 2

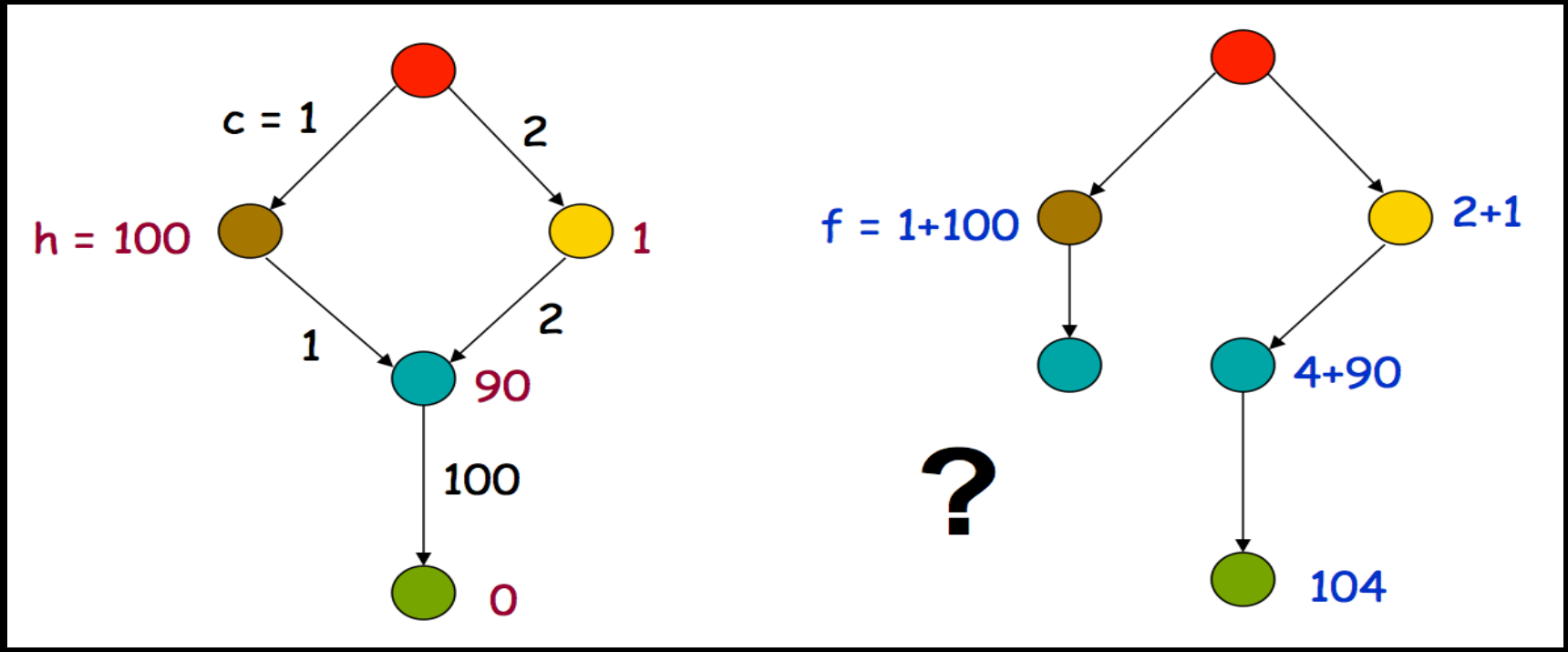
About repeated states

(how to make A optimal even with repeated states)*



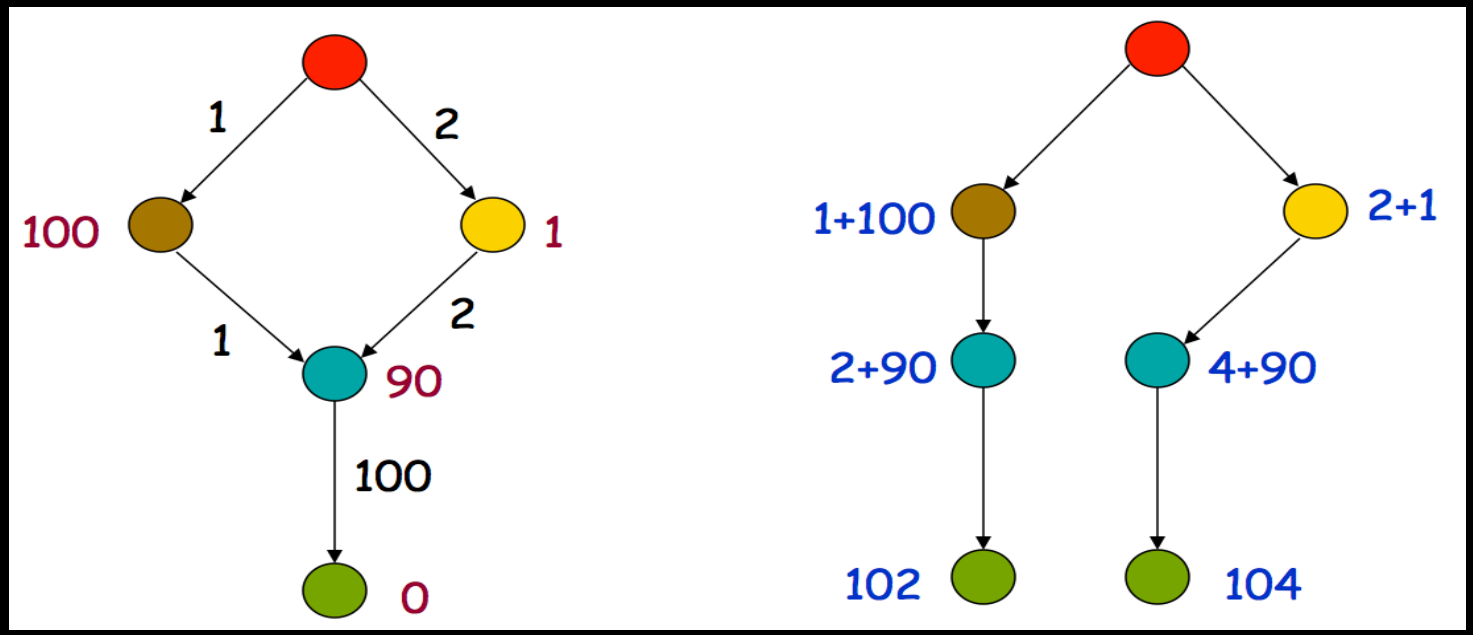
- The heuristic h is clearly admissible

About repeated states



If we discard this new node (if we avoid repeated states), then the search algorithm expands the goal node next and returns a non-optimal solution (cost = 104)

About repeated states

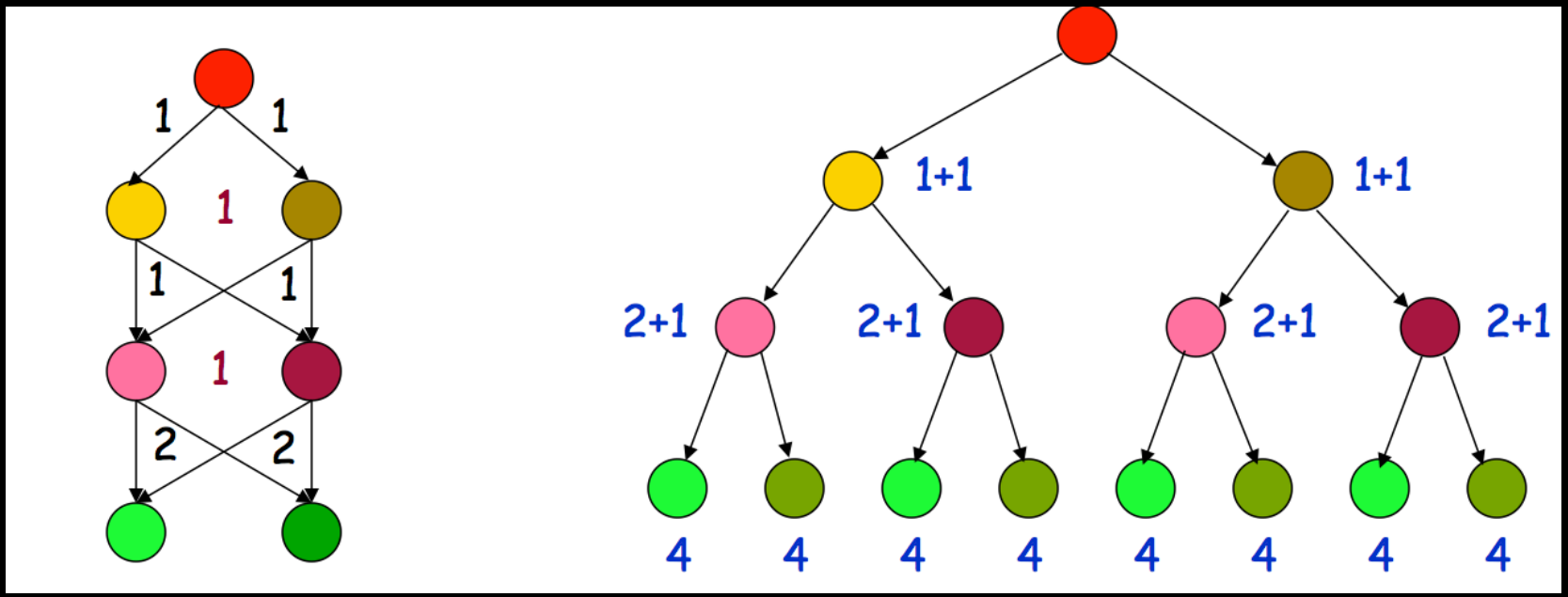


Instead, if we do not discard nodes revisiting states, the search terminates with an optimal solution

About repeated states

But

If we do not discard nodes revisiting states,
⇒ the size of the search tree can be exponential in the number of visited states



2n + 1 states

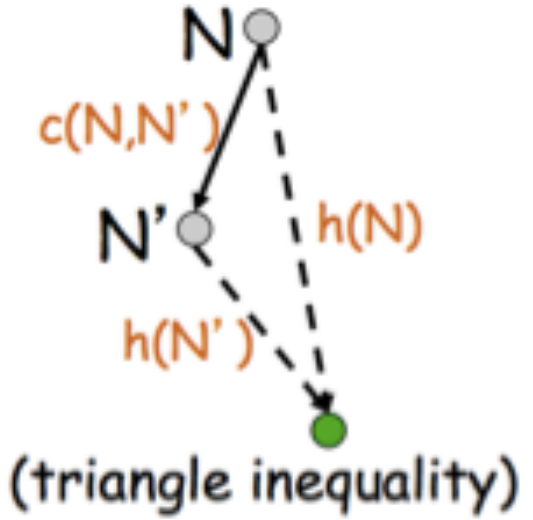
O(2n) nodes

Consistent Heuristic

Consistent Heuristic

The admissible heuristic h is **consistent** (or satisfies the monotone restriction) if for every node N and every successor N'of N:

$$h(N) \leq c(N, N') + h(N')$$



NB: A consistent heuristic is also admissible

Intuition

A consistent heuristic becomes more precise as we get deeper in the search tree

Consistent Heuristic: example

h(N) = 0

- h(N) = 0 for all nodes is admissible and consistent

⇒ Hence, breadth-first and uniform-cost are particular cases of A*!!!

8-puzzle

- h1(N) = number of misplaced tiles
- h2(N) = sum of distances of each tile to goal

⇒ are both consistent

Robot Navigation

- h(N) = straight-line distance from N to goal

⇒ is consistent

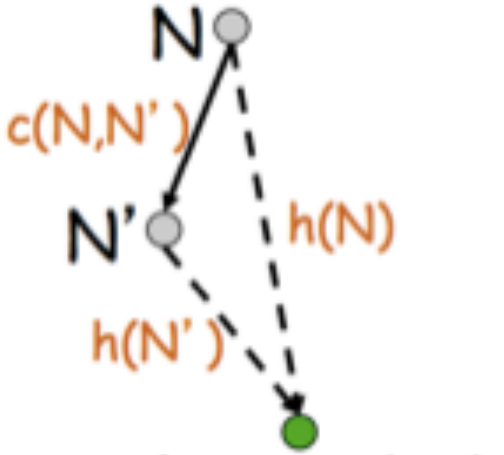
Claims on Consistent Heuristic

Claim 1

If h is consistent,

$\Rightarrow$ the function f along any path is non-decreasing:

$$
\begin{aligned}
f(N) &= g(N) + h(N) \\
f(N') &= g(N) + c(N, N') + h(N') \\
h(N) &\leq c(N, N') + h(N') \\
f(N) &\leq f(N')
\end{aligned}
$$



(triangle inequality)

Claim 2

If h is consistent, then whenever A* expands a node it has already found an optimal path to the state associated with this node

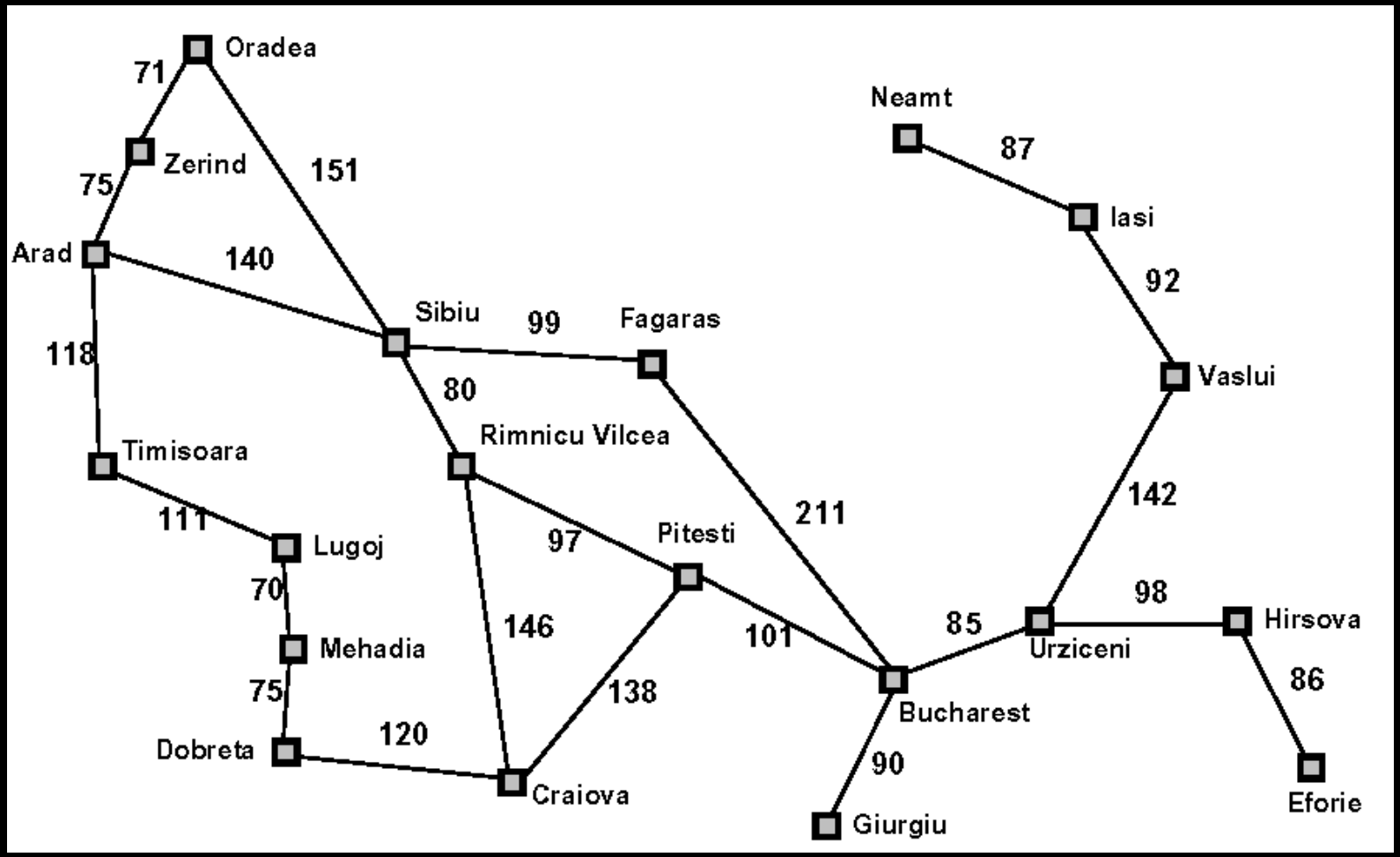
Avoiding Repeated States in A*

- If the heuristic h is consistent, then:
  - Let CLOSED be the list of states associated with expanded nodes

- When a new node N is generated:
  - If its state is in CLOSED, then discard N
  - If it has the same state as another node in the fringe, then discard the node with the largest f

A* - Exercise

Travel in Romania (from S. Russel et P. Norvig)



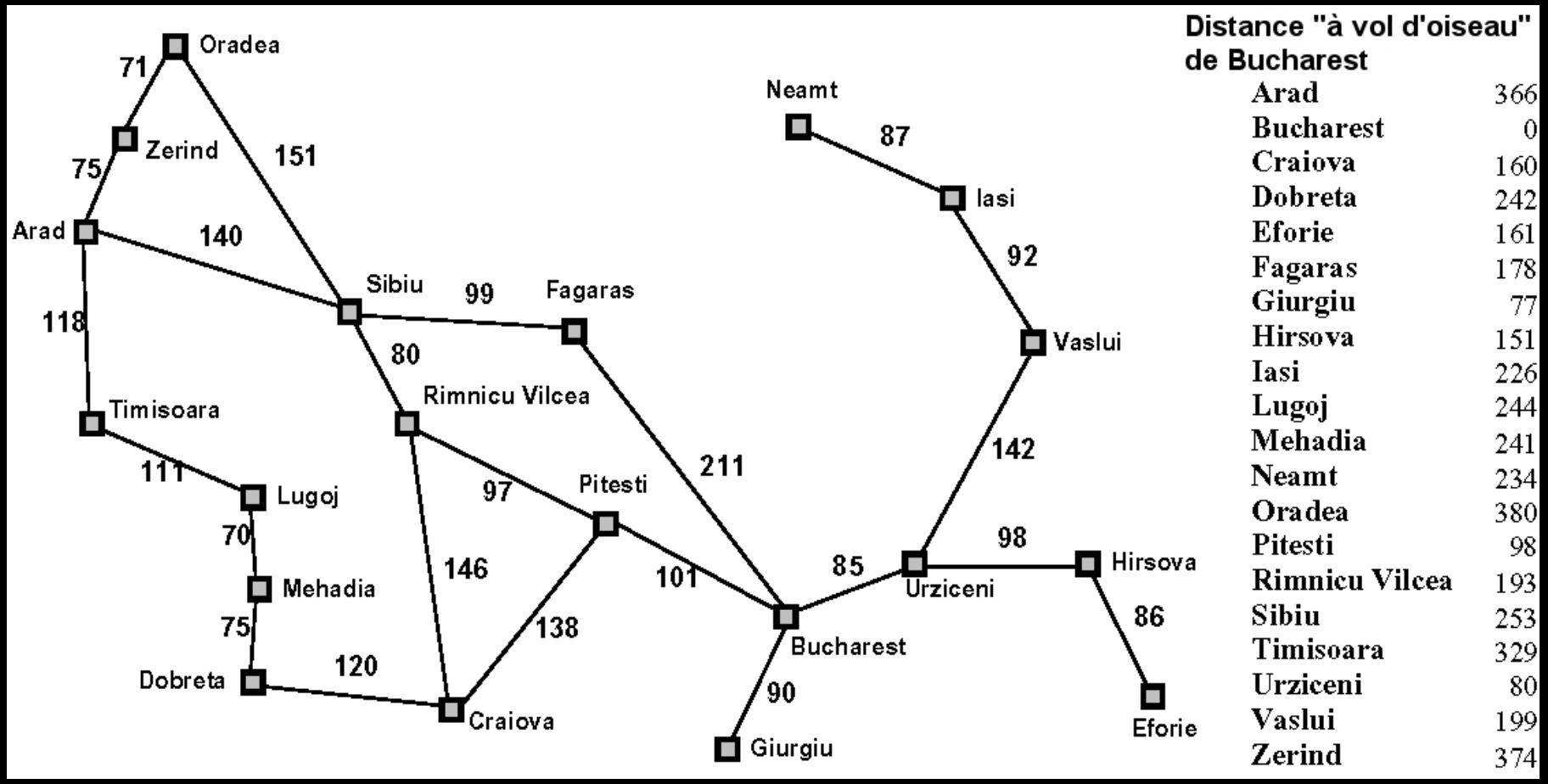
Algorithme A* - exercise

- From Arad to Bucharest

- We want to travel the least number of km.
- We want to use A*
  - What is g? What is h? (one we agree on h, run the algorithm)

Algorithme A* - exercise

From Arad to Bucharest



Distance "à vol d'oiseau" de Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

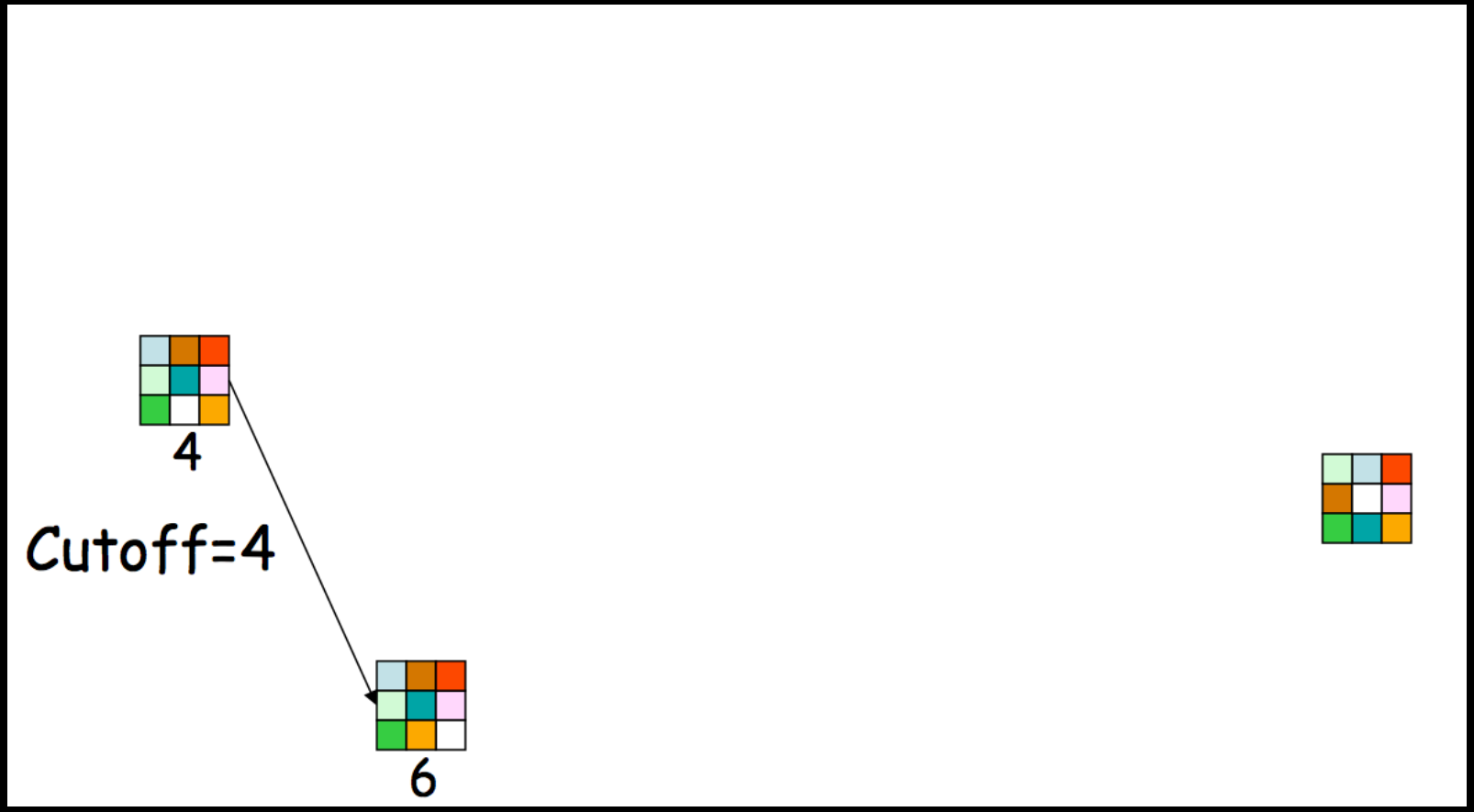
Iterative Deepening A* (IDA*)

- Use f(N) = g(N) + h(N) with admissible and consistent h
- Each iteration is depth-first with cutoff on the value of f of expanded nodes

cutoff ← f(initial-node)
loop
  DEPTH-FIRST by expanding all nodes N s.t. $f(n) \leq$cutoff
  cutoff ← smallest value f of non-expanded (leaf) nodes
end loop

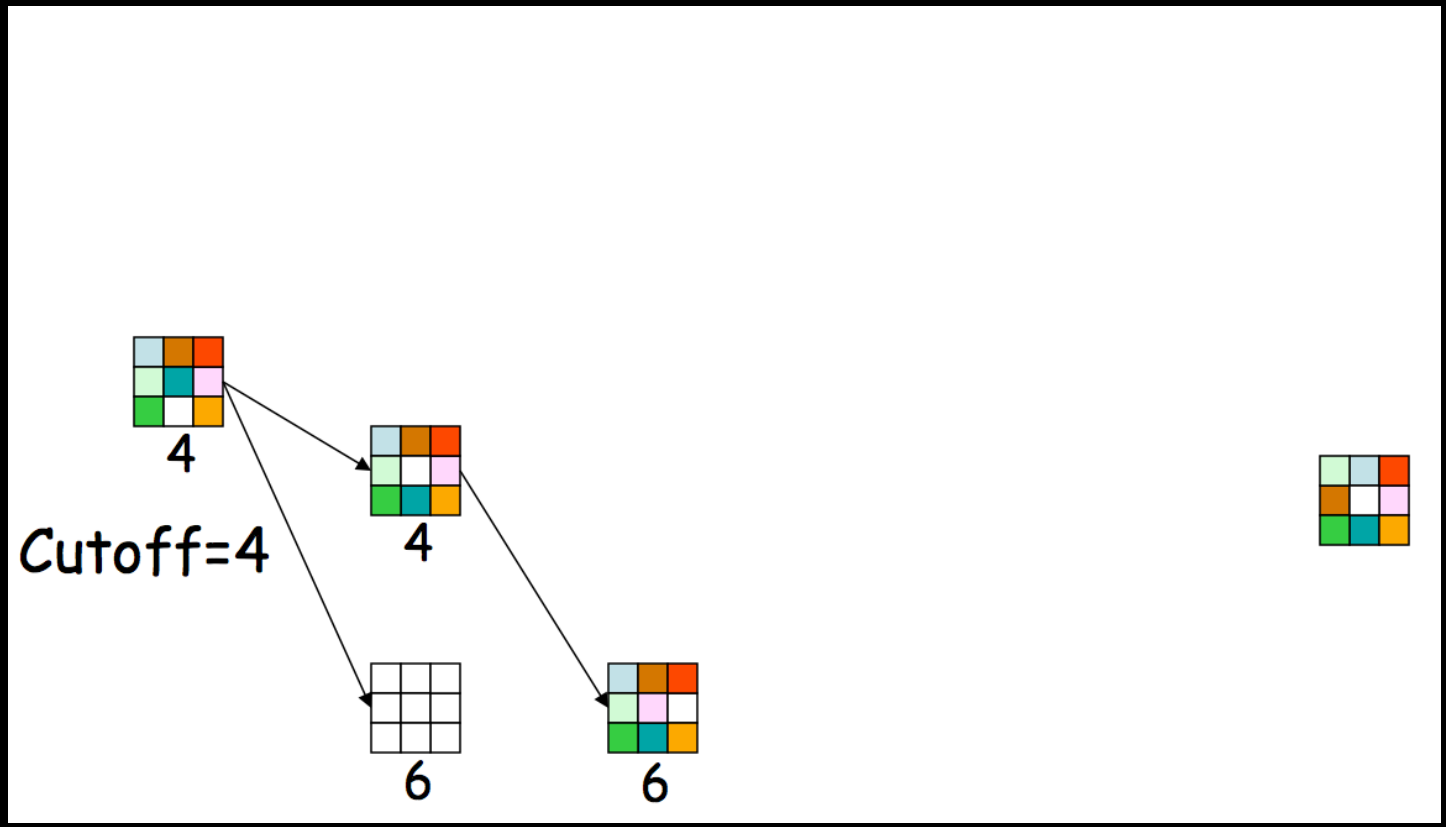
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



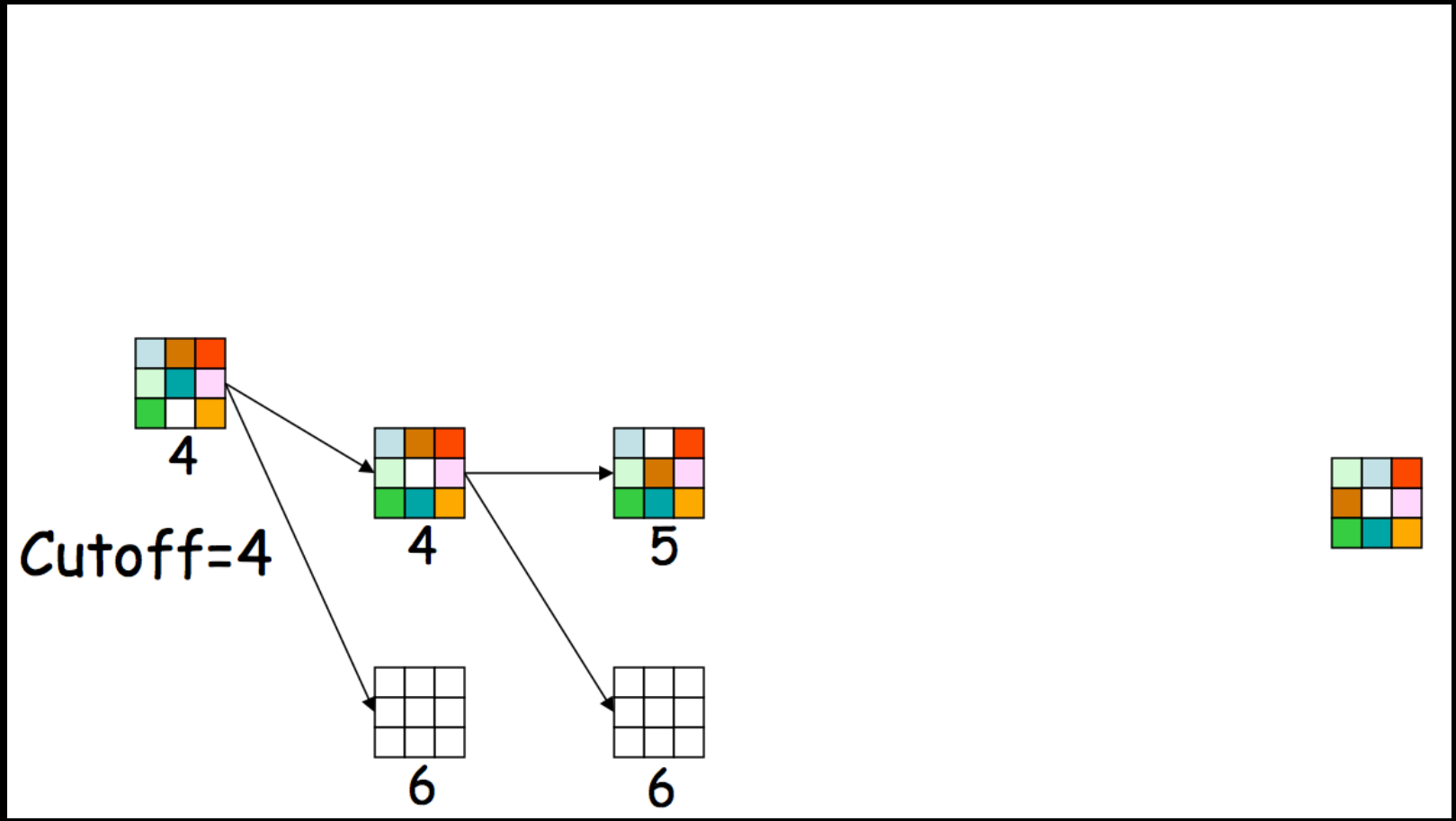
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



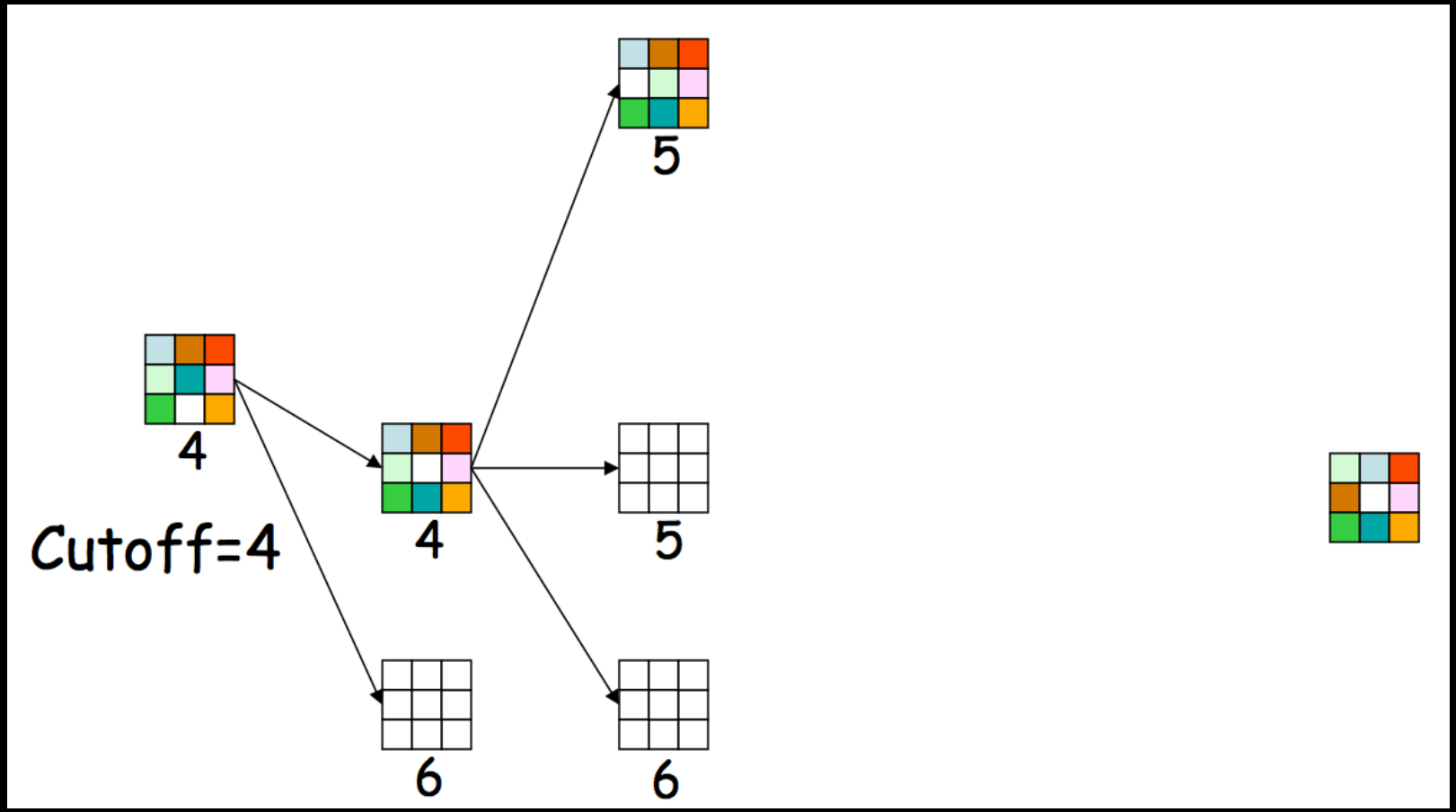
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



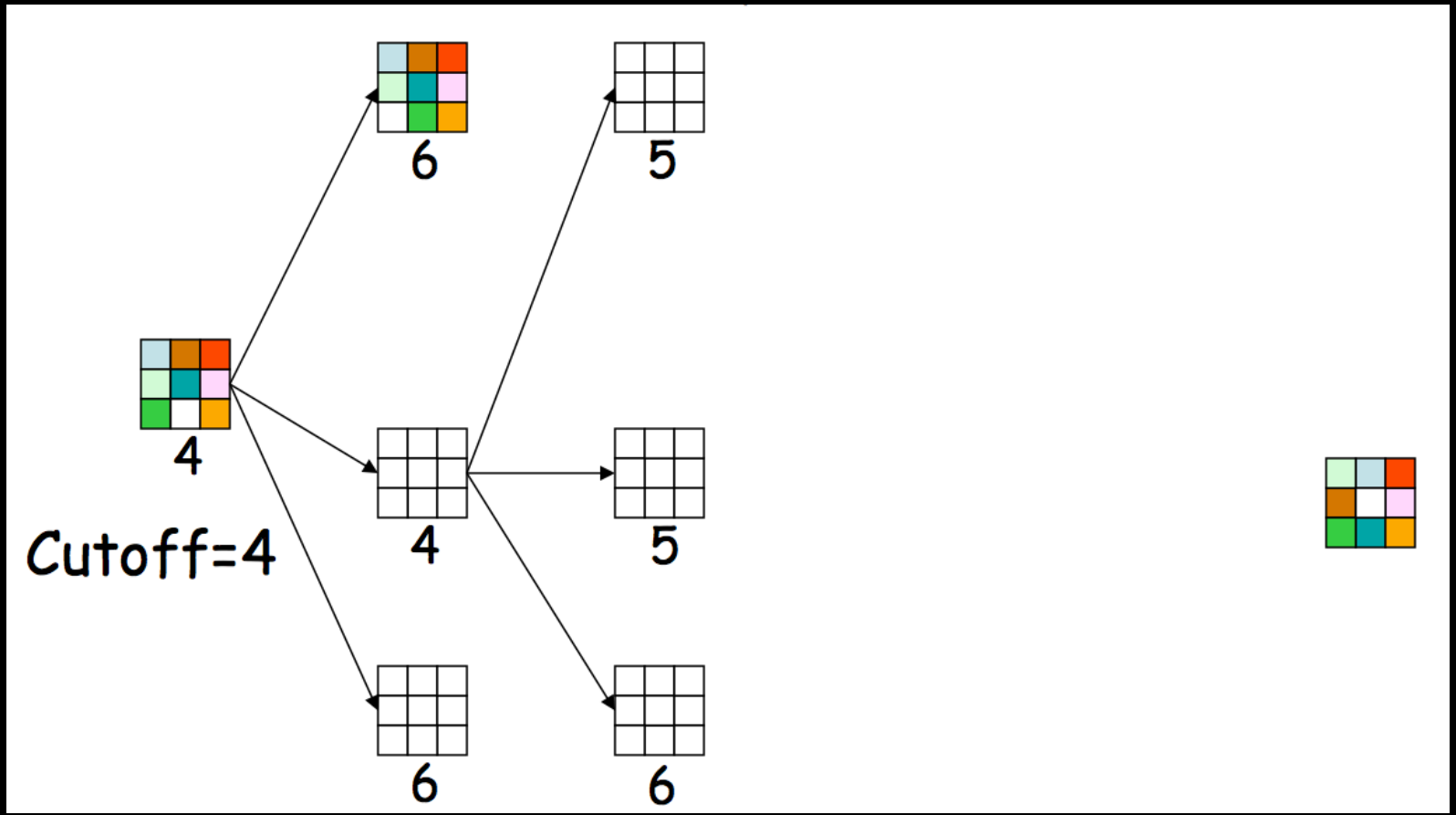
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



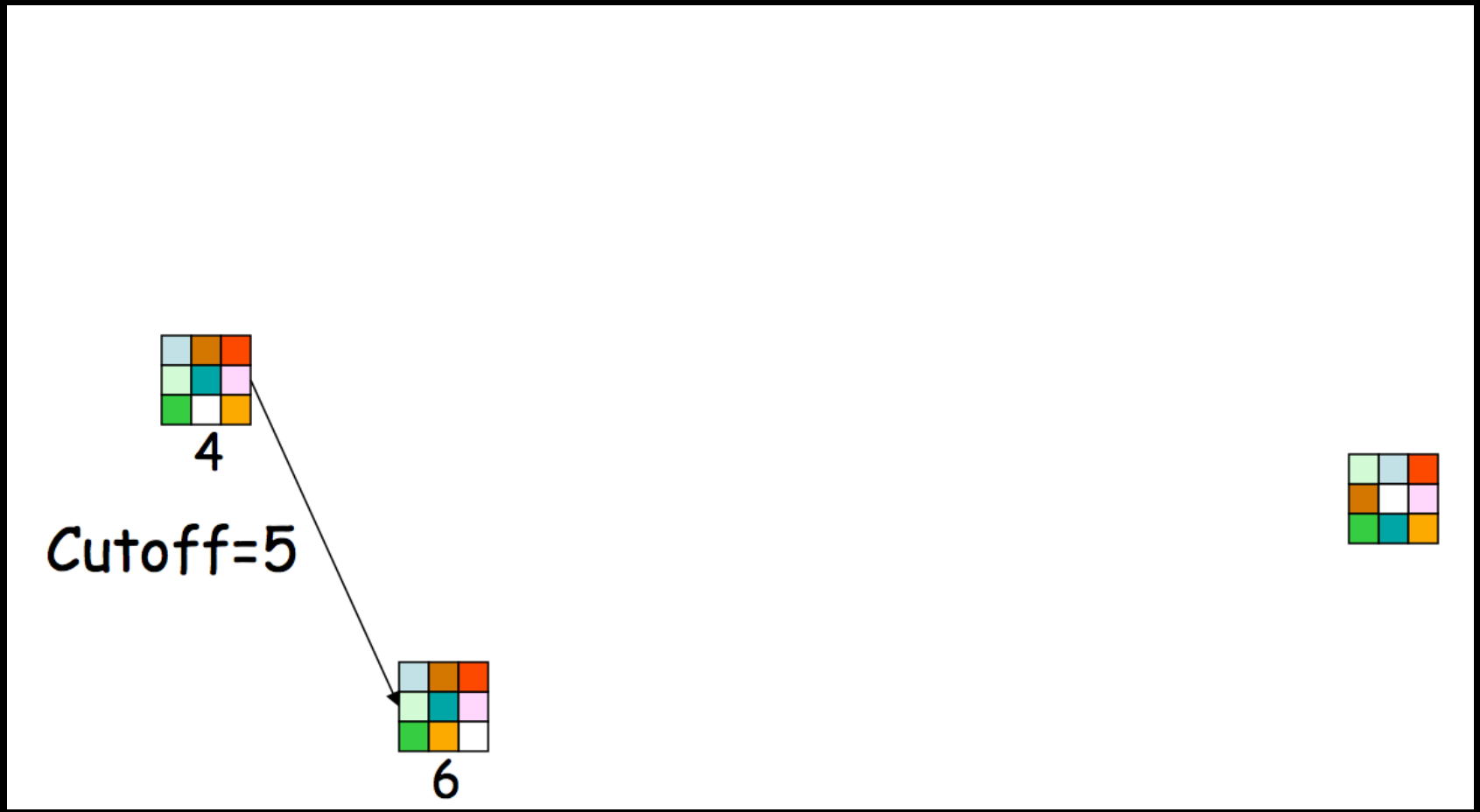
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



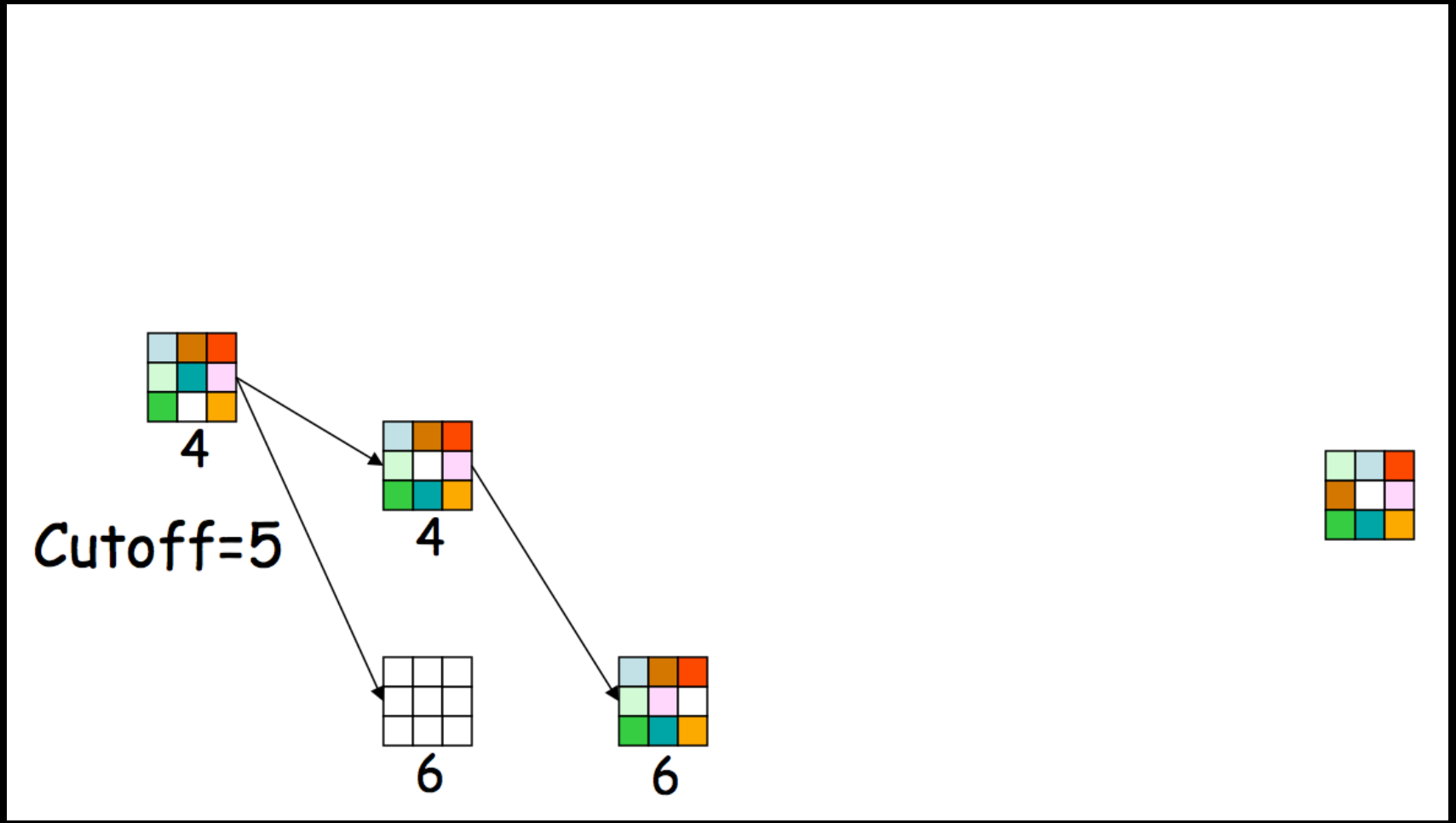
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



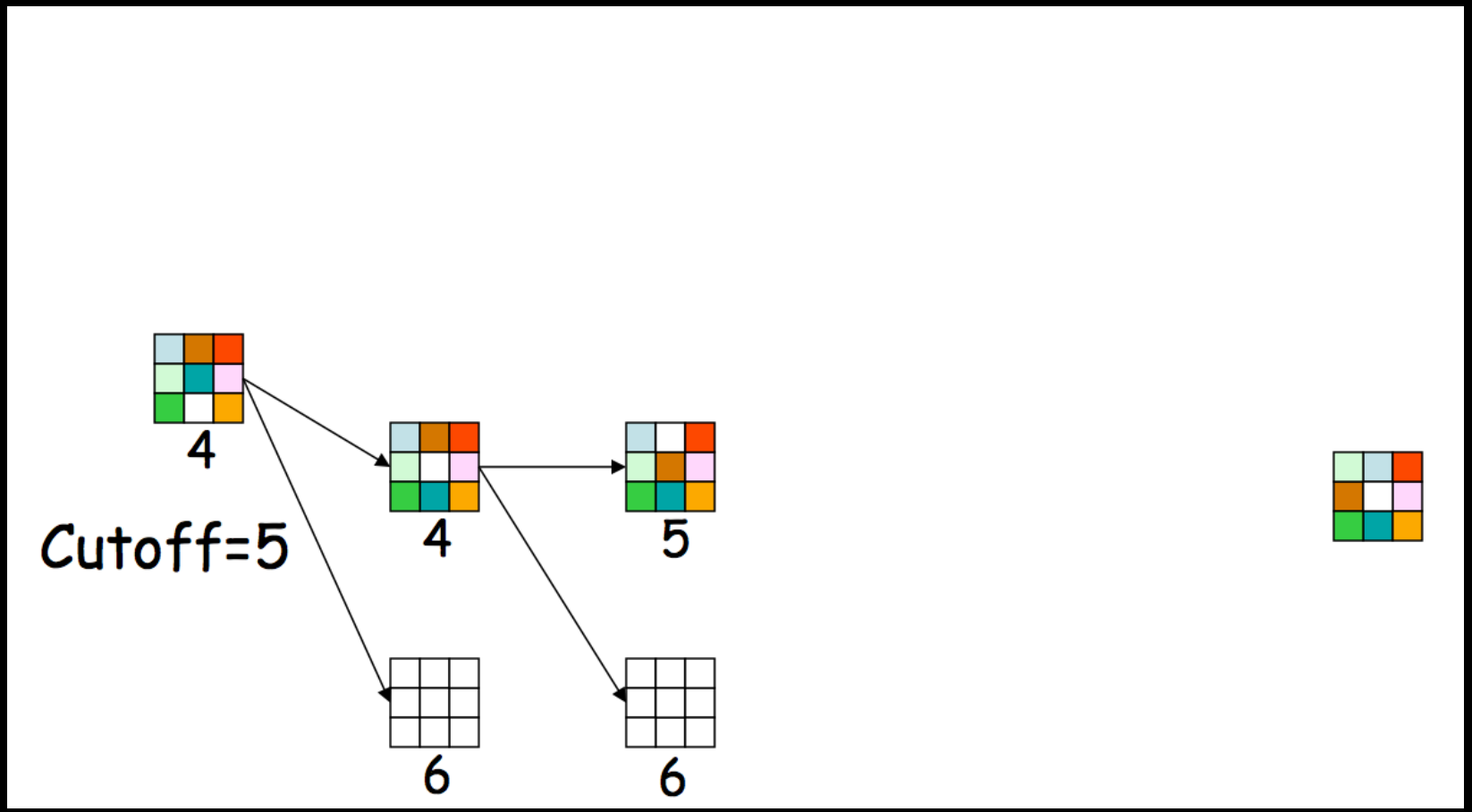
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



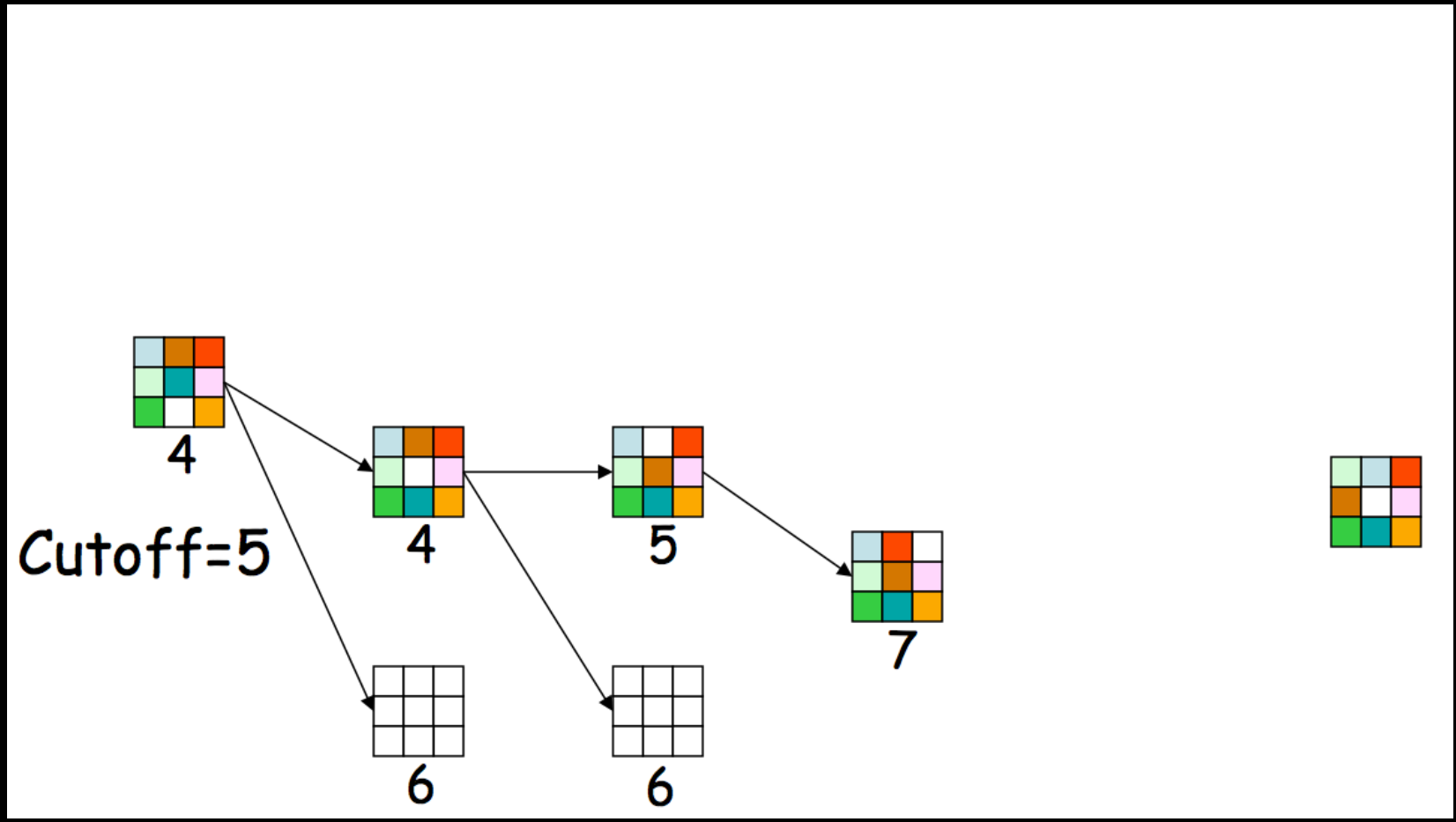
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



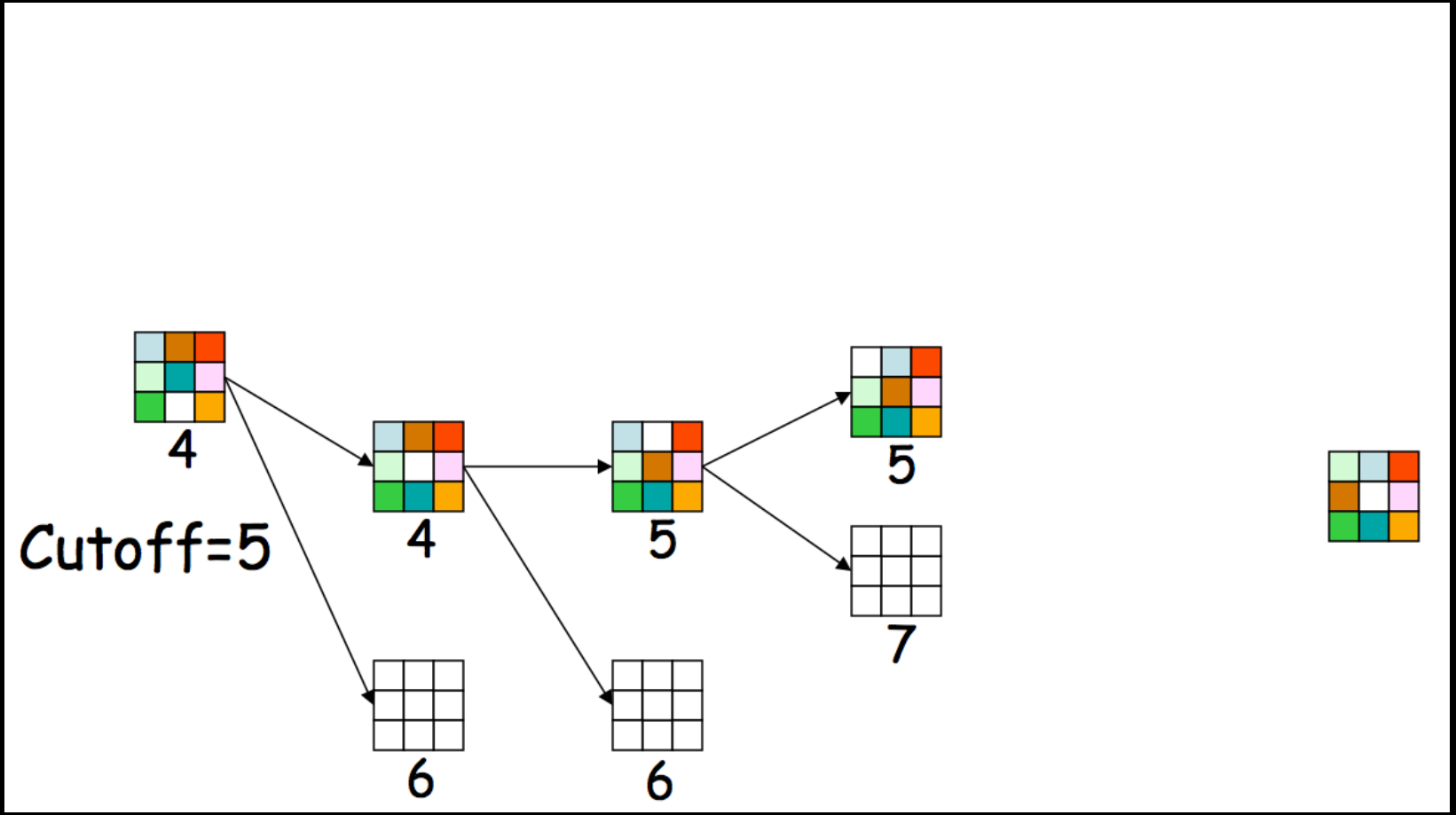
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



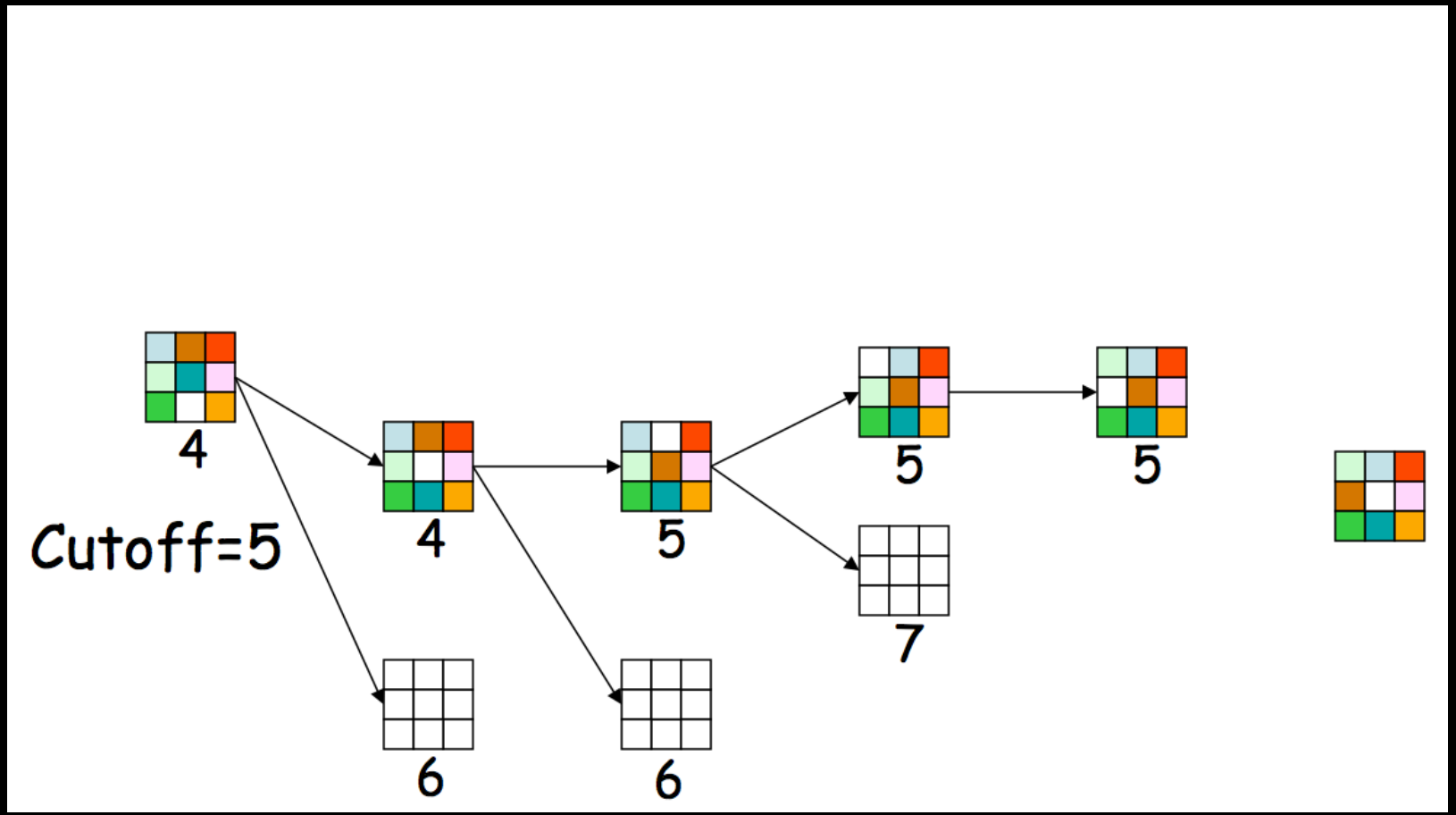
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



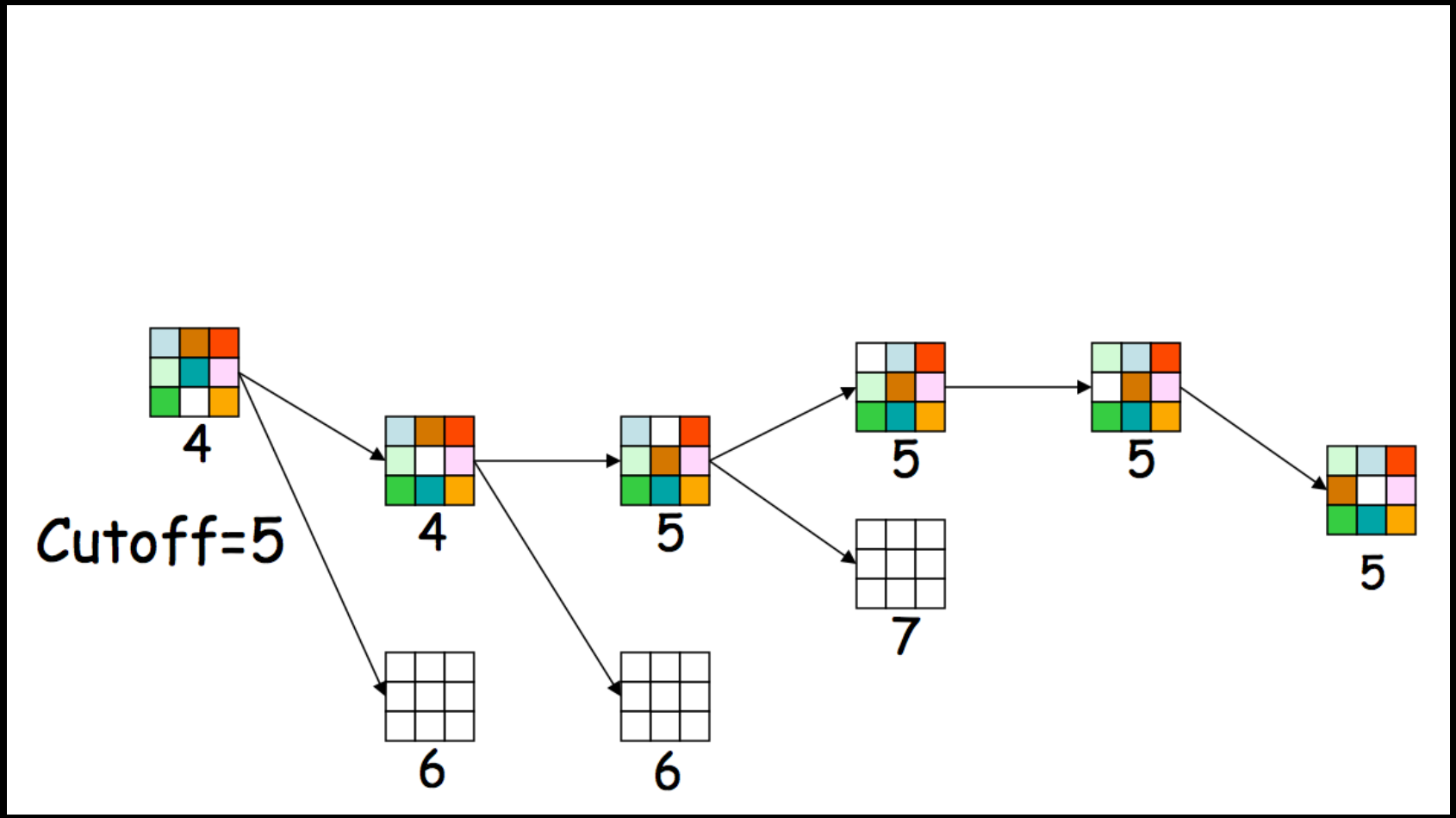
IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



IDA* on 8-puzzle

- f(n) = g(n) = h(n) with h(n) =number of misplaced tiles



Advantages and Drawbacks of IDA*

- Pros
  - Still complete and optimal
  - Requires less memory than A*
  - Avoid the overhead to sort the fringe
- • Cons
  - Can’t avoid revisiting states not on the current path
  - Available memory is poorly used

AO* = A* for AND/OR graphs
Cost g and heuristique h

- Recursive definition of the cost of a solution graph G(p)
  - if p is terminal, C(G(p)) = 0
  - if not, $C(G(p)) = k(Op) + \sum_{i=1}^{n} C(G(spi))$ where Op is such that Op : p → sp1, …, spn
  - Ex: k(Op) = n

- Heuristic h such that h(p) is the static estimation of the minimal cost to solve the problem p (h(t) = 0 if t is terminal)
  - Ex: distance as the crow flies btw 2cities.

Algorithm AO* - Principle

« best problem first »

1. Develop at each step the best problem according to the fact that it belongs to the best known-so-far solution graph and according to h.

2. Refine (update) the scores of each nodes according to the development done so far, i.e. replace h(p) by a dynamical estimation which takes the cost of the decomposition applicable to p into account:

$$f(p) = \min_{\{Op\}} \left(k(Op) + \sum_{i=1}^{n} f(spi) \right)$$

avec Op : p → sp1,…, spn

NB when the node is not yet developed: f(spi) = h(spi)

AO* Algorithm (sketch)

FRINGE $\leftarrow \{p0\}$ CLOSED $\leftarrow \emptyset$ DURING $\leftarrow \{p0\}$
while DURING $\neq \emptyset$ **AND** nodes in DURING are not primitives **do**
  p $\leftarrow$ REMOVE(DURING)
  for all $op_i \in$ List_op(p) **do**
    sp $\leftarrow$ op(p)
    FRINGE $\leftarrow$ INSERT(FRINGE,sp)
    $f_i \leftarrow cost(op_i) + c(sp_i)$
  end for
  FRINGE $\leftarrow$ REMOVE(FRINGE,p)
  CLOSED $\leftarrow$ INSERT(CLOSED,p)
  f(p) $\leftarrow \min(f_1, \ldots, f_n)$
  for all PARENT $\in$ GRAND-PARENTS(p) **do**
    f(PARENT) $\leftarrow$ UPDATE-COST(PARENT)
  end for
  for all node $\in$ CLOSED $\cap$ Gr-Sol **do**
    update the pointer associated to node
  end for
  for all LEAVE $\in$ Gr-Sol **do**
    DURING $\leftarrow$ INSERT(DURING,LEAVE)
  end for
end while

AO* - Properties

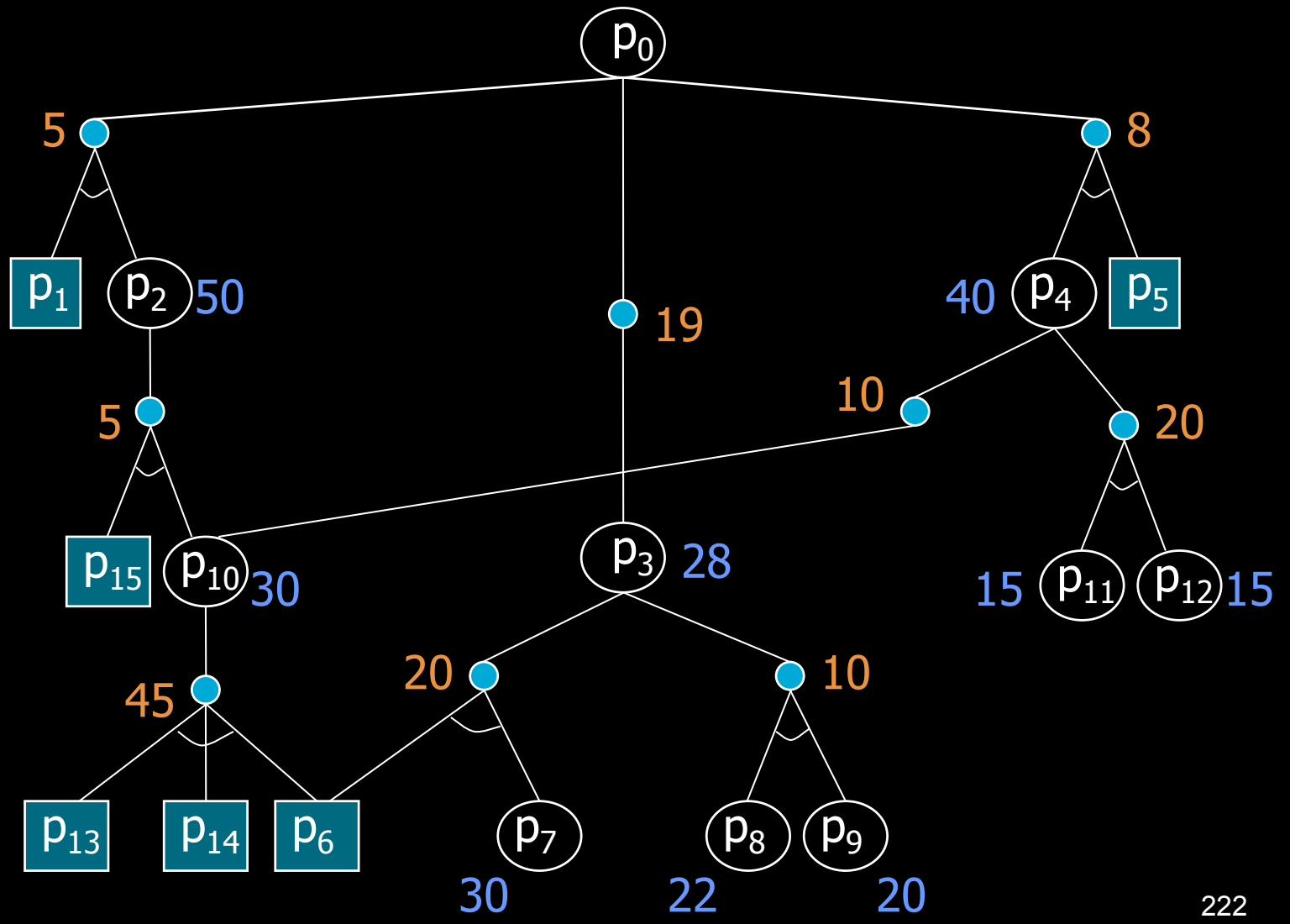
- Hypotheses: no cycle and independent sub-problems.

- If h is admissible and consistent
  - AO* finds the optimal solution (according to the cost function)
- Equivalent to A* for AND/OR graph (except for the function g)

AO* - Example *(1/6)*

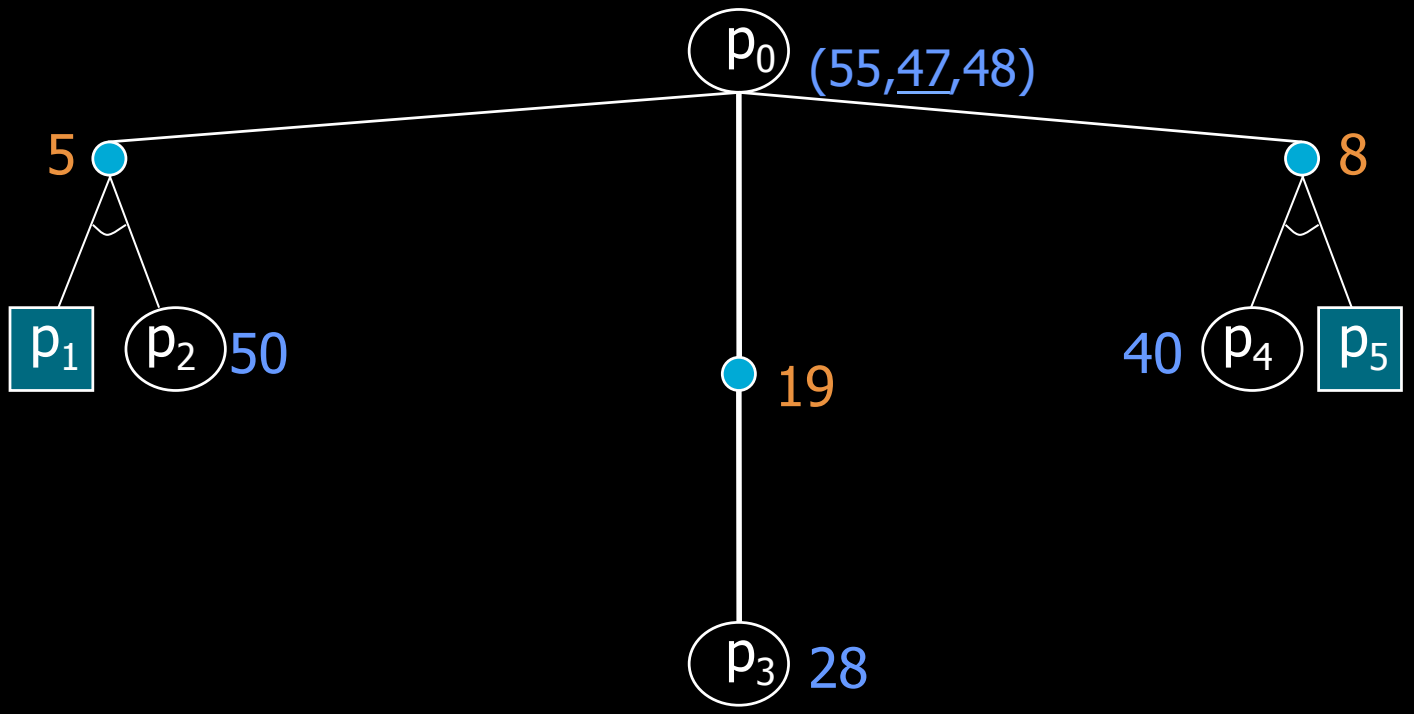
k in orange

h(f) in blue



AO* - Example (2/6)

Step 1



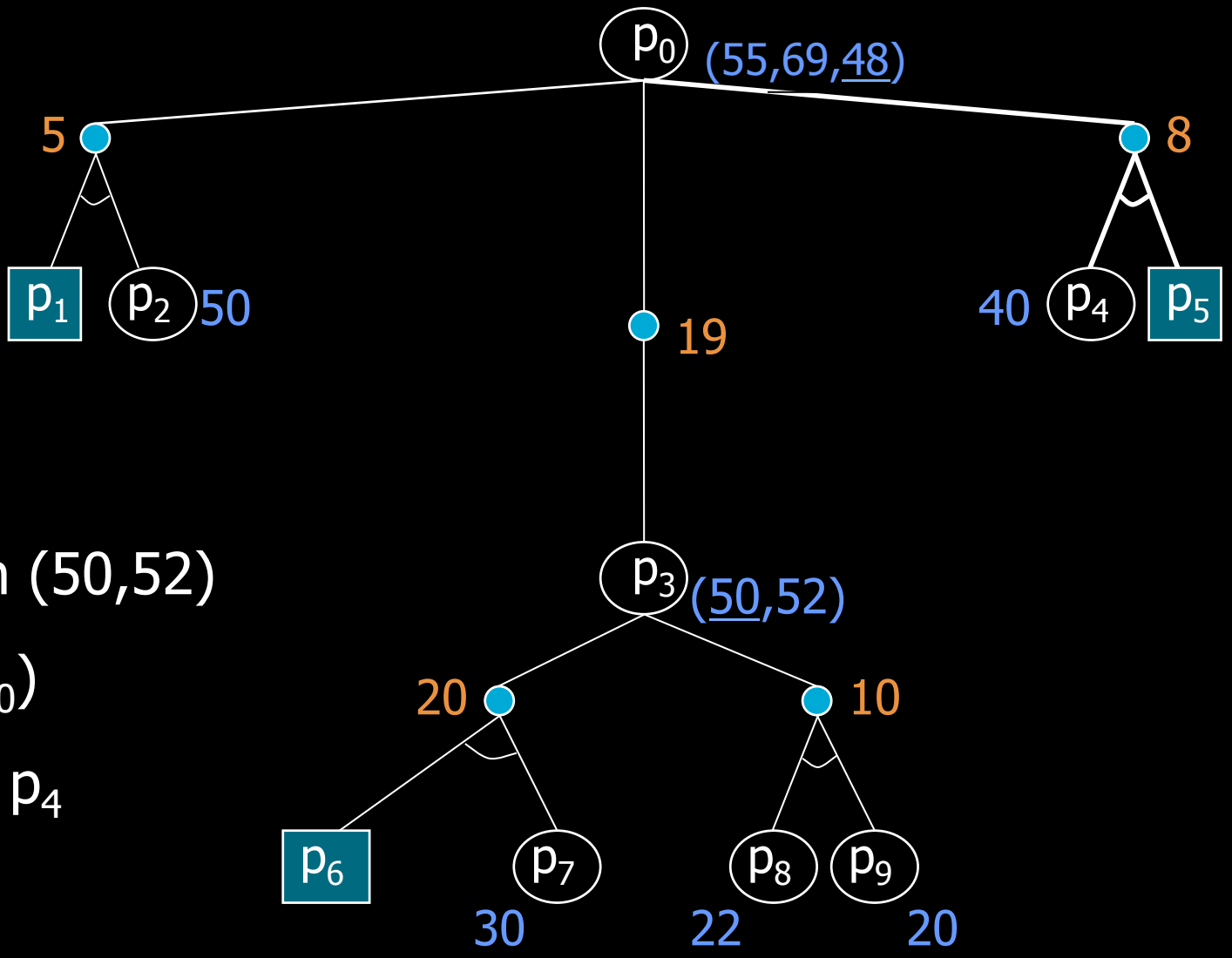
Development of p_0

f(p0) = min ((5+50+0), (19+28), (8+40)) = min (55,47,48)

- we chose to develop p_3

AO* - Example *(3/6)*

Step 2



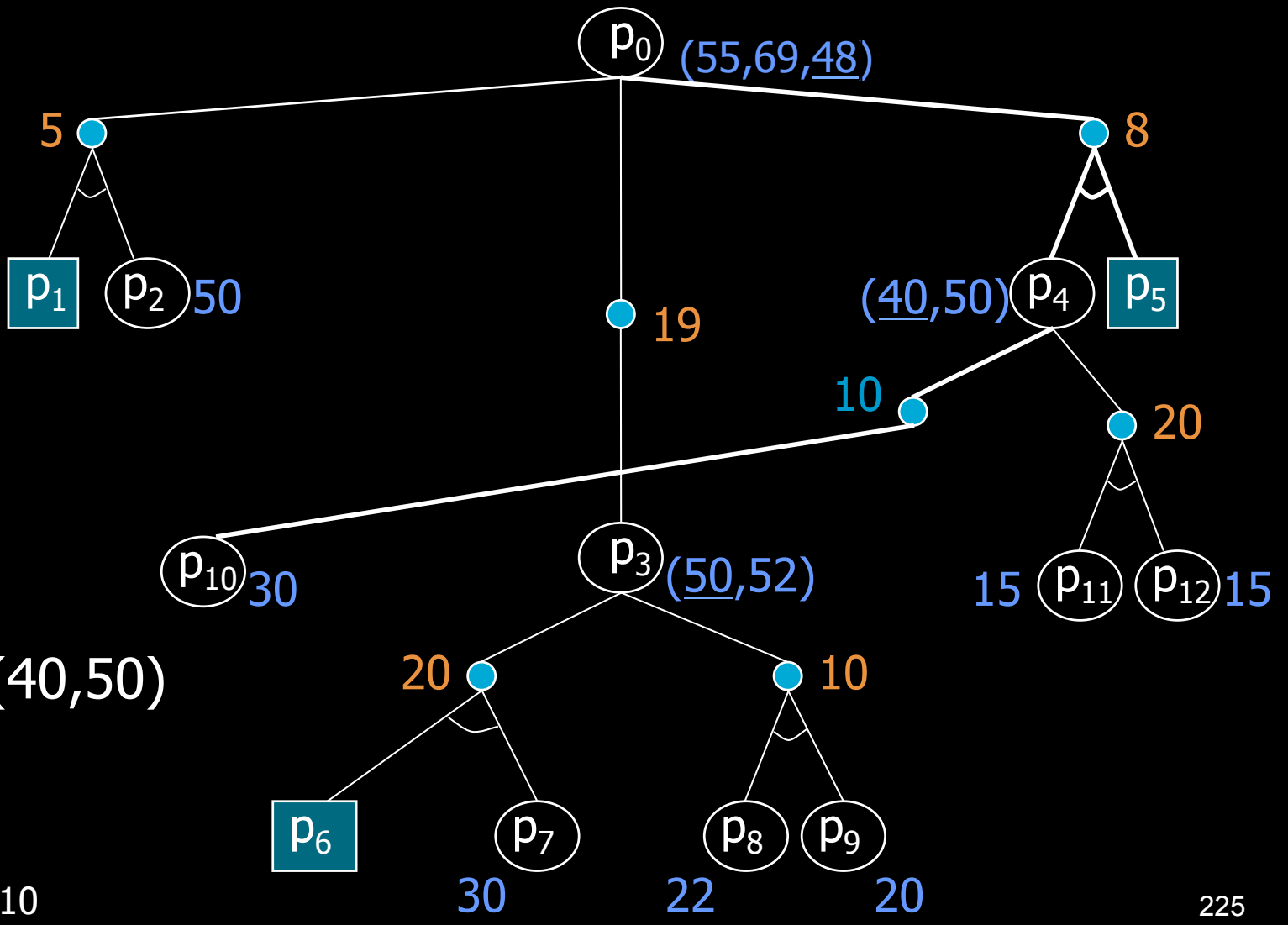
$f(p_3) = \min(50,52)$

Update $f(p_0)$

- Develop p_4

AO* - Example *(4/6)*

Step 3



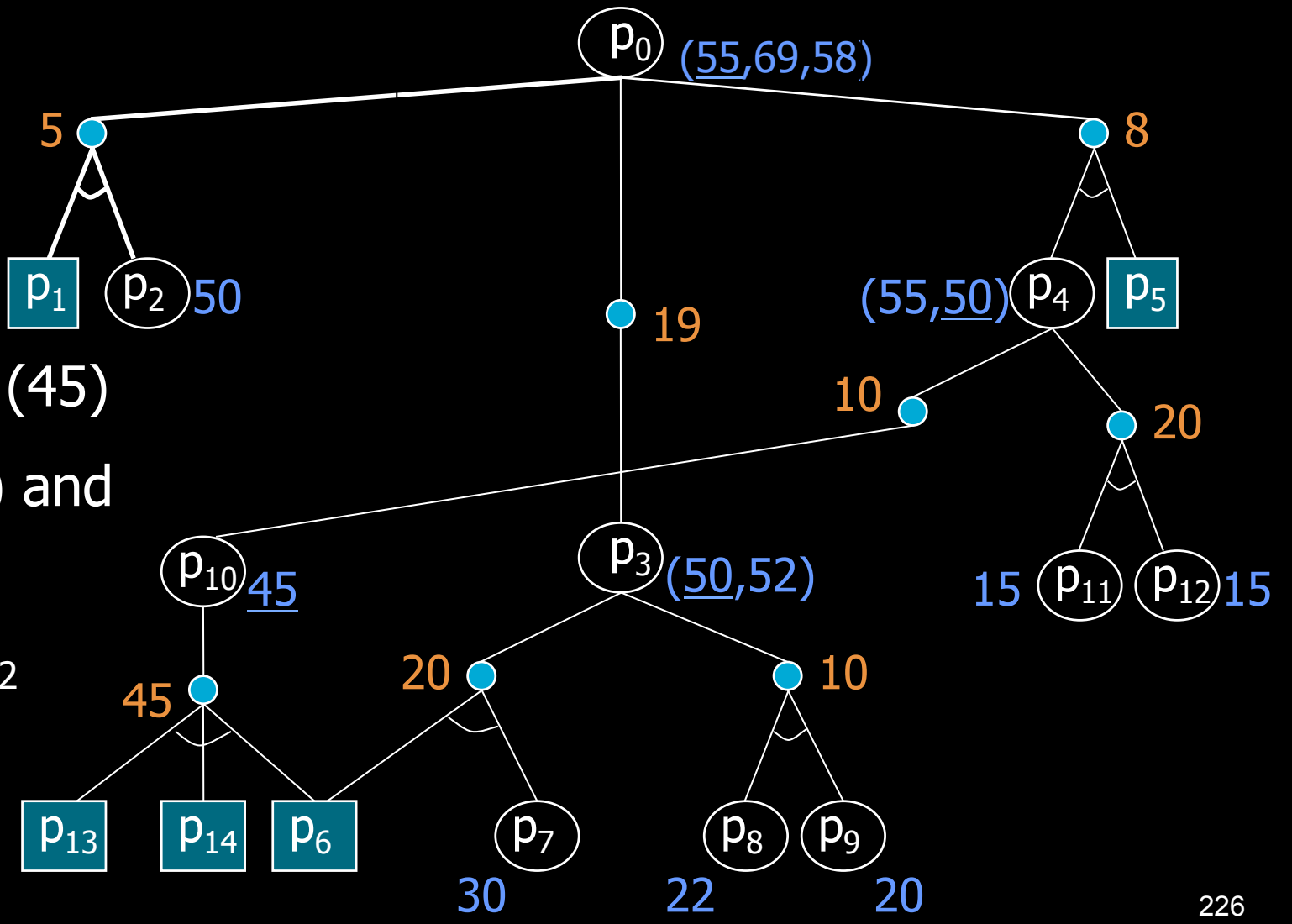
$f(p_4) = \min(40,50)$

Update $f(p_0)$

- Develop p_{10}

AO* - Example *(5/6)*

Step 4



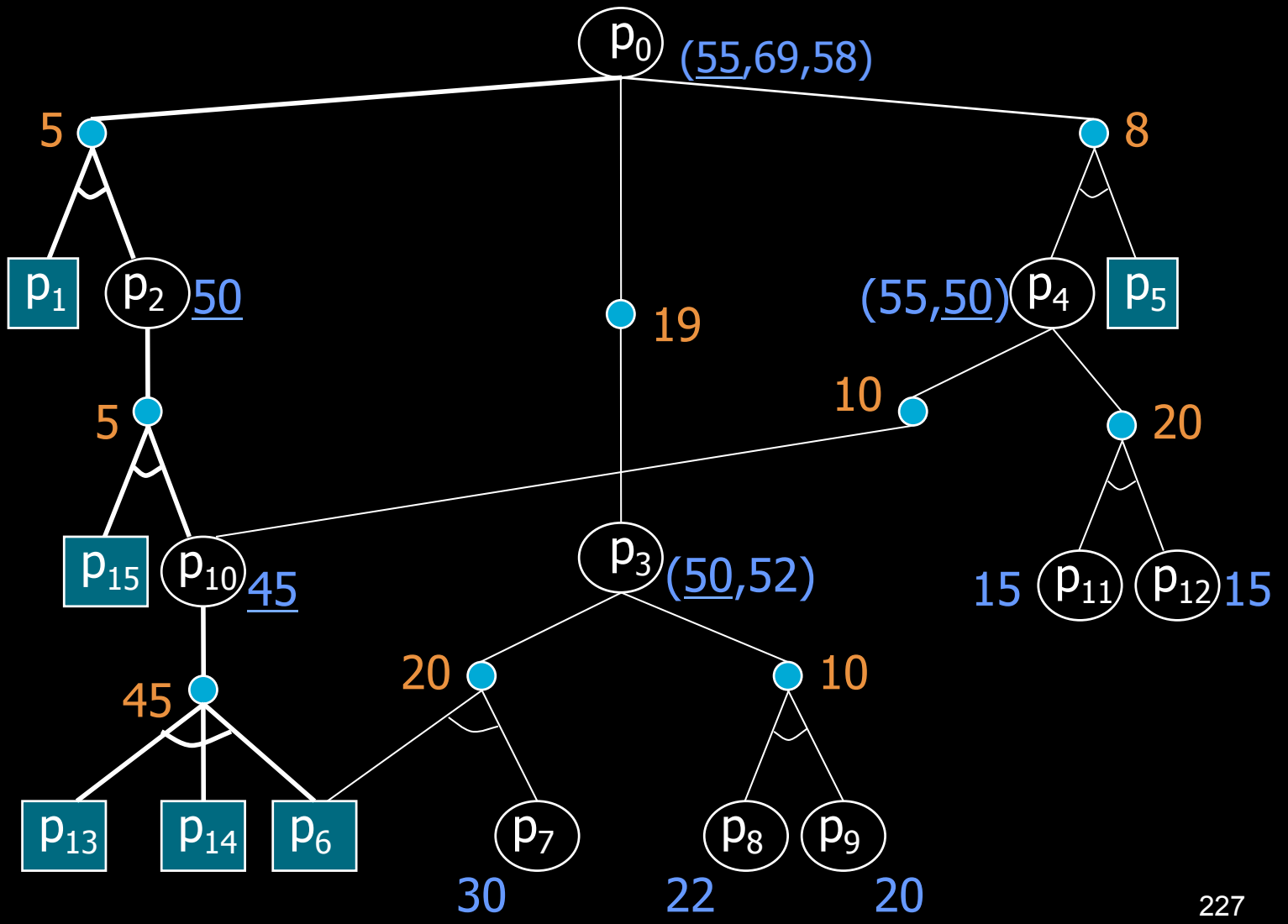
$f(p_{10}) = \min(45)$

Update f(p4) and $f(p_0)$

- Develop p_2

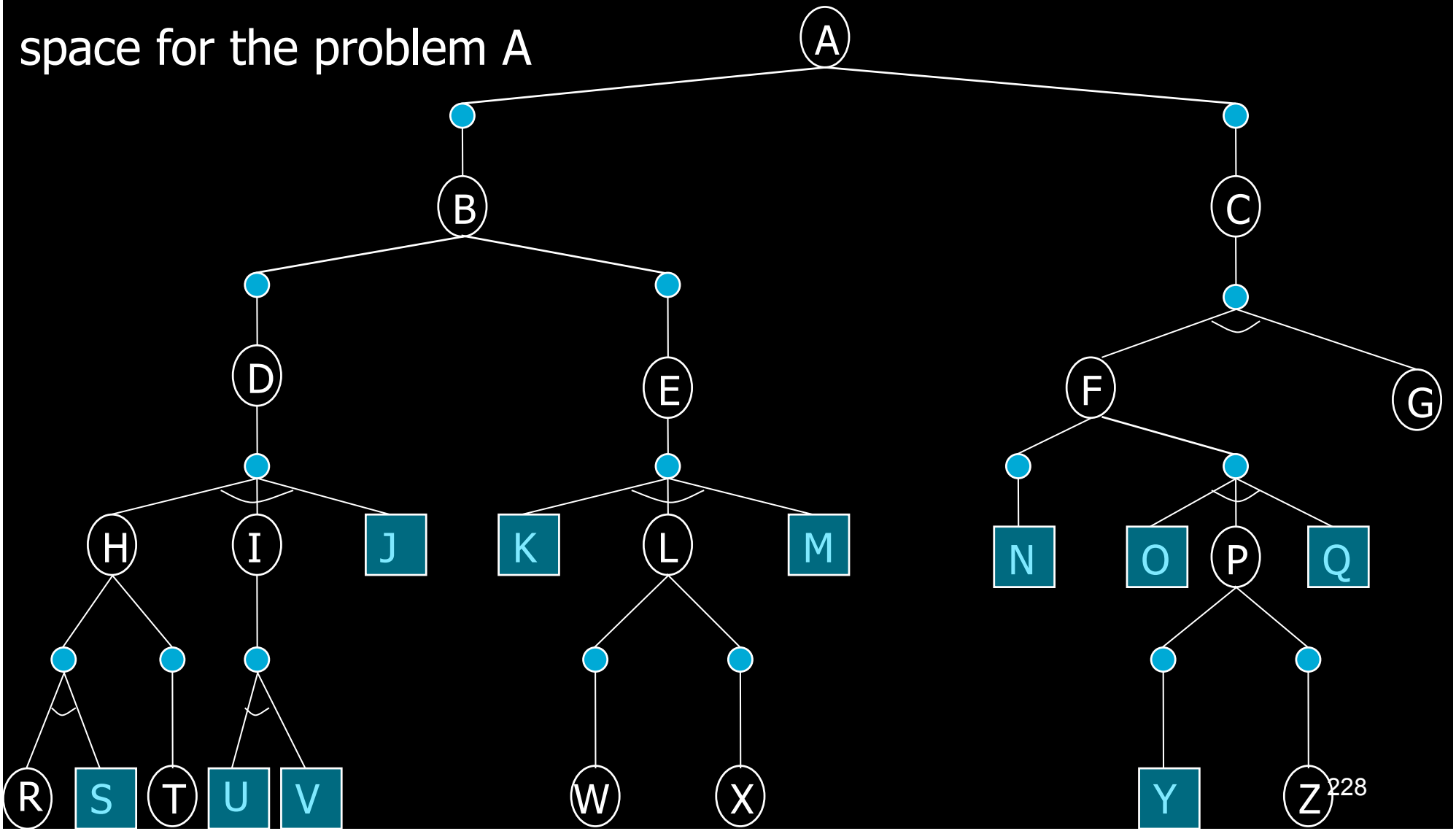
AO* - Example *(6/6)*

Step 5



AO* - Exercise

Intermediate search space for the problem A



AO* Exercise (questions)

1. What are the decomposition rules for the previous AND/OR graph?
2. Let h a heuristic such that h(G)=h(T)=h(Z)=∞ (insolvable pb) h(J)= h(K) = h(M) = h(N) = h(O)= h(Q)=h(S)=h(U)=h(V)=h(Y)=0 (trivial pb), h(R)=1, h(W)=3 et h(X)=7 ; what are the solved sub-graphs? (OR-path from anywhere in the graph)
3. If the cost k of each operator is the number of generated sub-problems, what is the best partial solution for the previous intermediate search space, what is its cost?
4. What would be the next developed node using AO*?

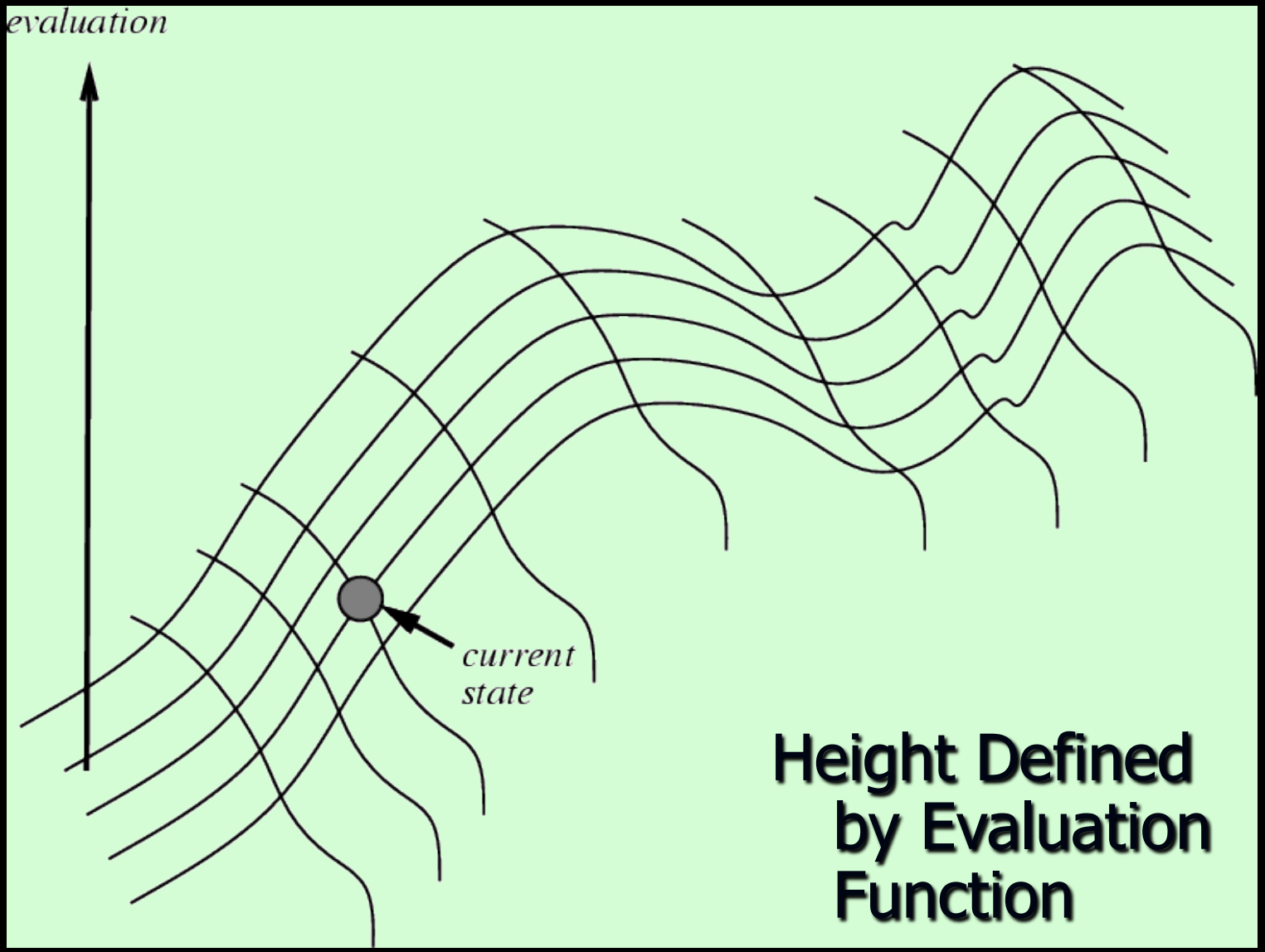
Other approaches (skipped if late on the schedule)

- Optimization problems
  - rather than constructing an optimal solution from scratch, start with a suboptimal solution and iteratively improve it
- Local Search Algorithms
  - Hill-climbing or Gradient descent
  - Potential Fields
  - Simulated Annealing
  - Genetic Algorithms, others…

Hill-climbing search

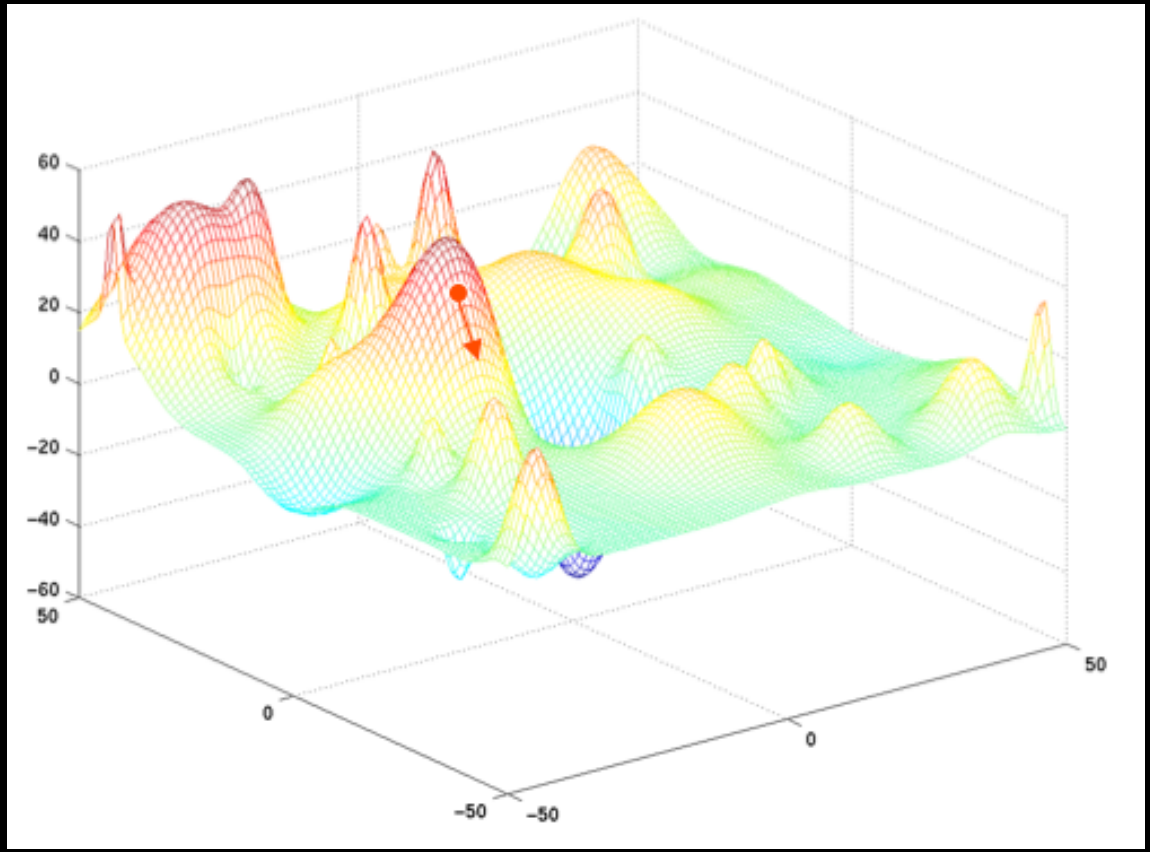
- If there exists a successor s for the current state n such that
  - h(s) < h(n)
  - h(s) <= h(t) for all the successors t of n,
- then move from n to s. Otherwise, halt at n.
- Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor.
- similar to Greedy search in that it uses h, but does not allow backtracking or jumping to an alternative path since it doesn't "remember" where it has been.
- Not complete since the search will terminate at "local minima," "plateaus," and "ridges."

Hill climbing on a surface of states



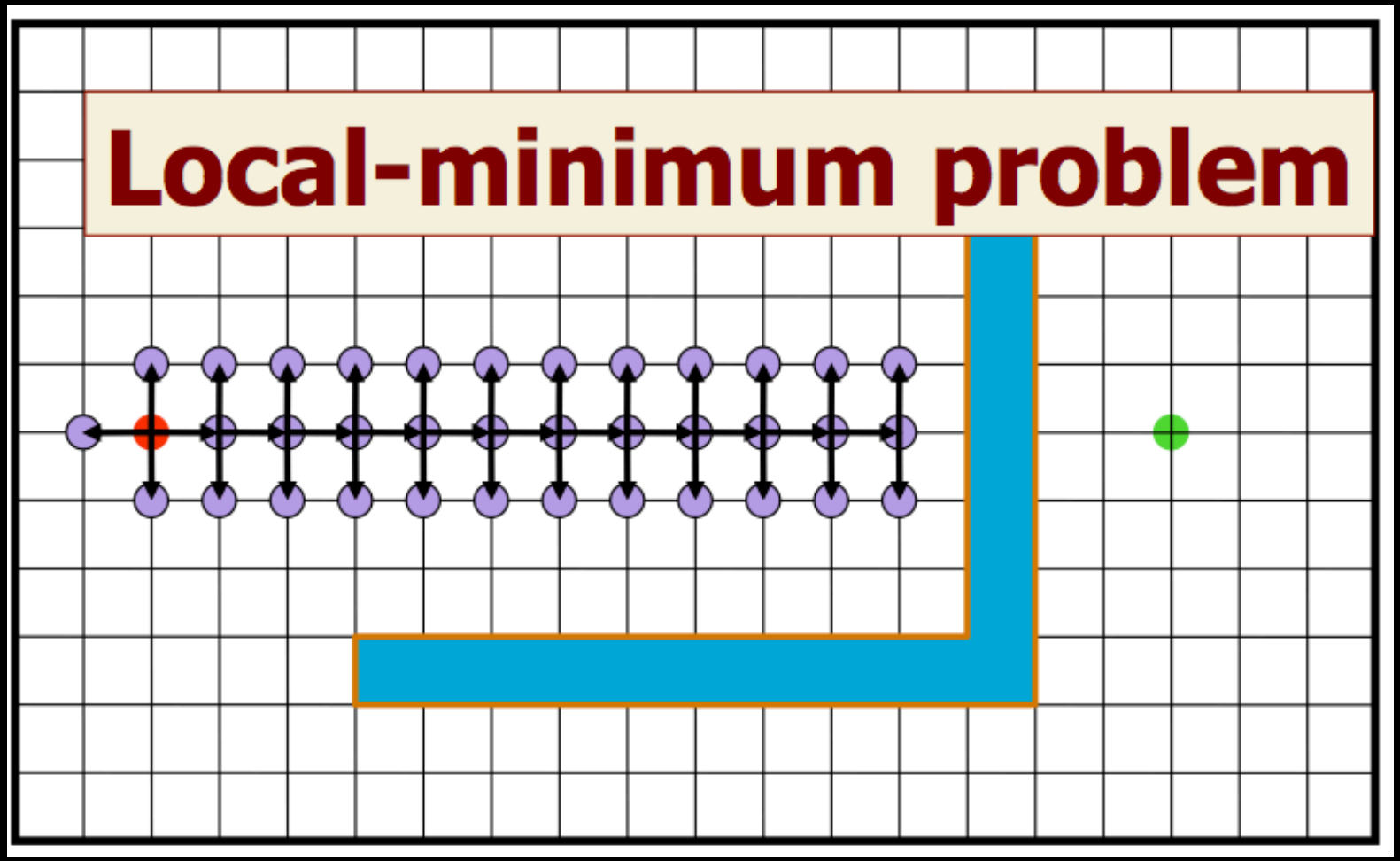
Hill climbing

- Steepest descent (~ greedy best-first with no search) → may get stuck into local minimum



Robot navigation

- f(n) = h(n) = « straight distance to the goal »



Drawbacks of hill climbing

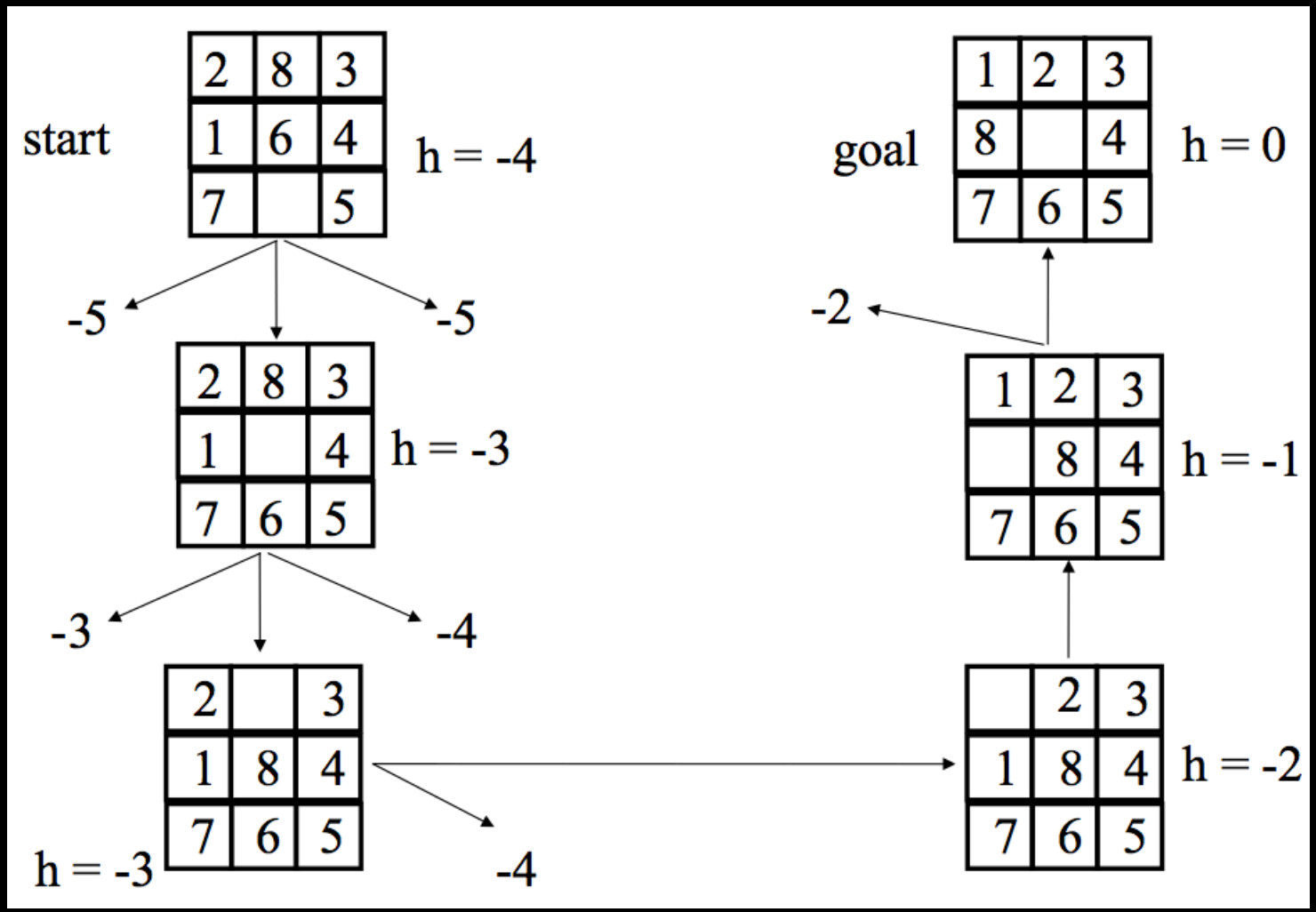
- Problems:
  - Local Maxima: peaks that aren't the highest point in the space
  - Plateaus: the space has a broad flat region that gives the search algorithm no direction (random walk)
  - Ridges: flat like a plateau, but with dropoffs to the sides; steps to the North, East, South and West may go down, but a step to the NW may go up.
- Remedy:
  - Introduce randomness
    Ex: Random restart.
- Some problem spaces are great for hill climbing and others are terrible.

What's the Issue?

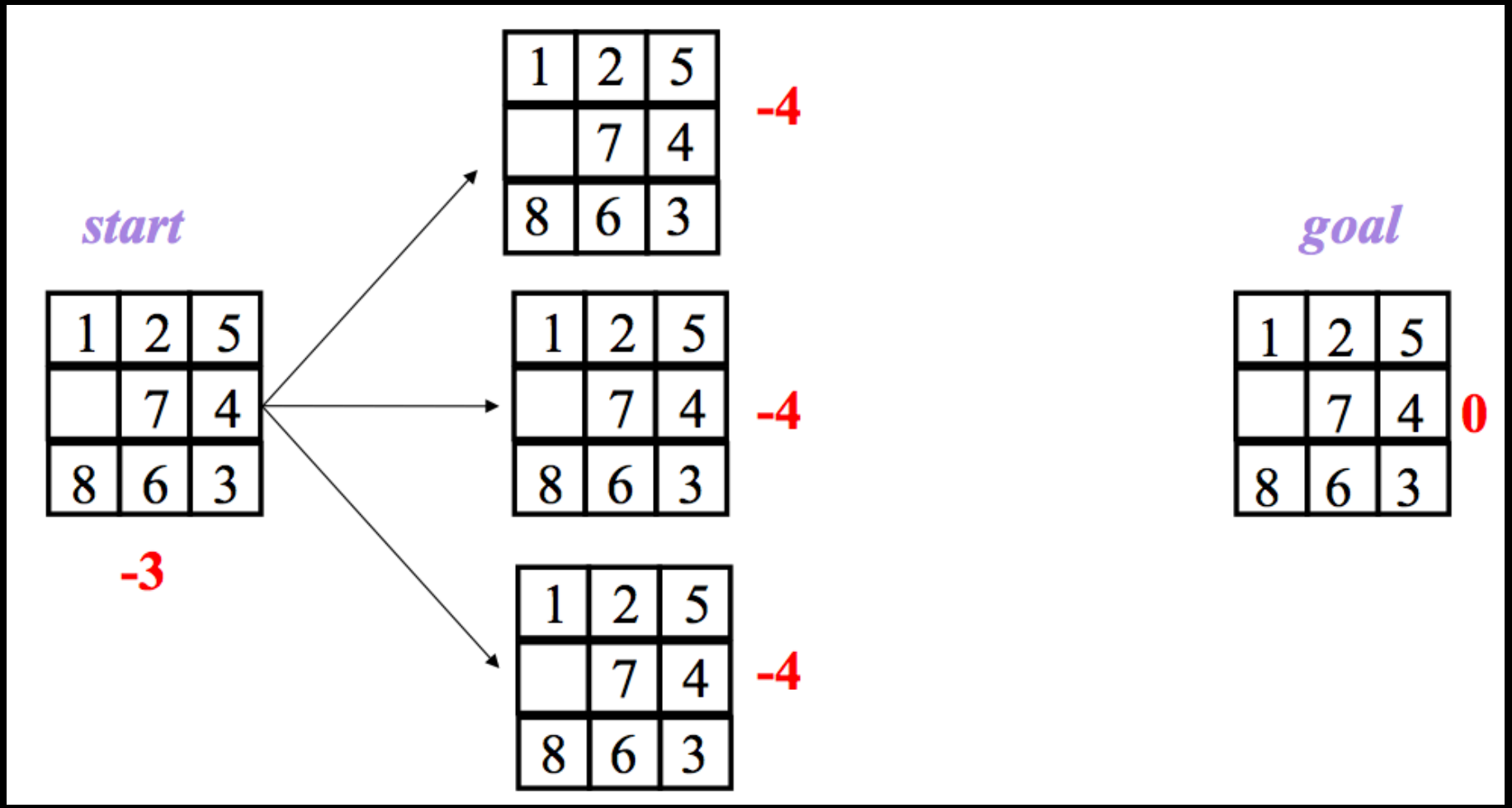
- Search is an iterative **local** procedure
- Good heuristics should provide some **global look-ahead** (at low computational cost)

Ex: Hill climbing on 8-puzzle

f(n) = -(number of tiles out of place)



Ex of local maximum



Potential Fields

- **Idea**: modify the heuristic function
- Goal is gravity well, drawing the robot toward it
- Obstacles have repelling fields, pushing the robot away from them
- This causes robot to “slide” around obstacles
- Potential field defined as sum of attractor field which get higher as you get closer to the goal and the individual obstacle repelling field (often fixed radius that increases exponentially closer to the obstacle)

Does it always work?

- No.
- But, it often works very well in practice
- Advantage #1:
  - can search a very large search space without maintaining fringe of possibilities
  - Scales well to high dimensions, where no other methods work
  - Example: motion planning
- Advantage #2: local method. Can be done online

Example: RoboSoccer

All robots have same field: attracted to the ball

- repulsive potential to other players
- Kicking field: attractive potential to the ball and local repulsive potential if close to the ball, but not facing the direction of the opponent's goal. Result is tangent, player goes around the ball.
- single team: kicking field + repulsive field to avoid hitting other players + player position fields (parabolic if outside your area of the field, 0 inside). Player nearest to the ball has the largest attractive coefficient, avoids all players crowding the ball.
- Two teams: identical potential fields.

Simulated annealing

- Simulated annealing (SA) exploits an analogy between the way in which a metal cools and freezes into a minimum-energy crystalline structure (the annealing process) and the search for a minimum [or maximum] in a more general system.
- SA can avoid becoming trapped at local minima.
- SA uses a random search that accepts changes that increase objective function f, as well as some that decrease it.
- SA uses a control parameter T, which by analogy with the original application is known as the system "temperature."
- T starts out high and gradually decreases toward 0.

simulated annealing (cont.)

- A “bad” move from A to B is accepted with a probability

$$e^{((f(B)-f(A)) / T)}$$

- The higher the temperature, the more likely it is that a bad move can be made.
- As T tends to zero, this probability tends to zero, and SA becomes more like hill climbing
- If T is lowered slowly enough, SA is complete and admissible.

The simulated annealing algorithm

CURRENT $\leftarrow$ INITIAL-NODE
for $t \leftarrow 1$ **to** ∞ **do**
  $T \leftarrow$ SCHEDULE[t] (SCHECULE is a mapping from time to temperature)
  if $T = 0$ **then return** CURRENT
  SUCCESSOR $\leftarrow$ a randomly selected successor of CURRENT
  $\Delta E \leftarrow f(\text{SUCCESSOR}) - f(\text{CURRENT})$
  if $\Delta E > 0$ **then**
    CURRENT $\leftarrow$ SUCCESSOR
  else
    CURRENT $\leftarrow$ SUCCESSOR only with probability $\exp\left(\frac{\Delta E}{T}\right)$
  end if
end for

NB: Successful application on circuit routing, TSP

Summary: Local Search Algorithms

- Steepest descent (~ greedy best-first with no search)
  - may get stuck into local minimum
- Better Heuristics: Potential Fields
- Simulated annealing
- Genetic algorithms

When to Use Search Techniques?

- The search space is small, and
  - There is no other available techniques,

  or
  - It is not worth the effort to develop a more efficient technique

- The search space is large, and
  - There is no other available techniques,

  and
  - There exist “good” heuristics

Outline

1. Intro
2. AI Problem Representation
3. Problem solving: Uninformed Search
    1. Depth-first search, Breadth-first search, Uniform search, Iterative deepening…
    2. Problem decomposition (AND/OR tree)
4. Heuristic (informed) Search
    1. A*, AO*…
5. **Game Playing**

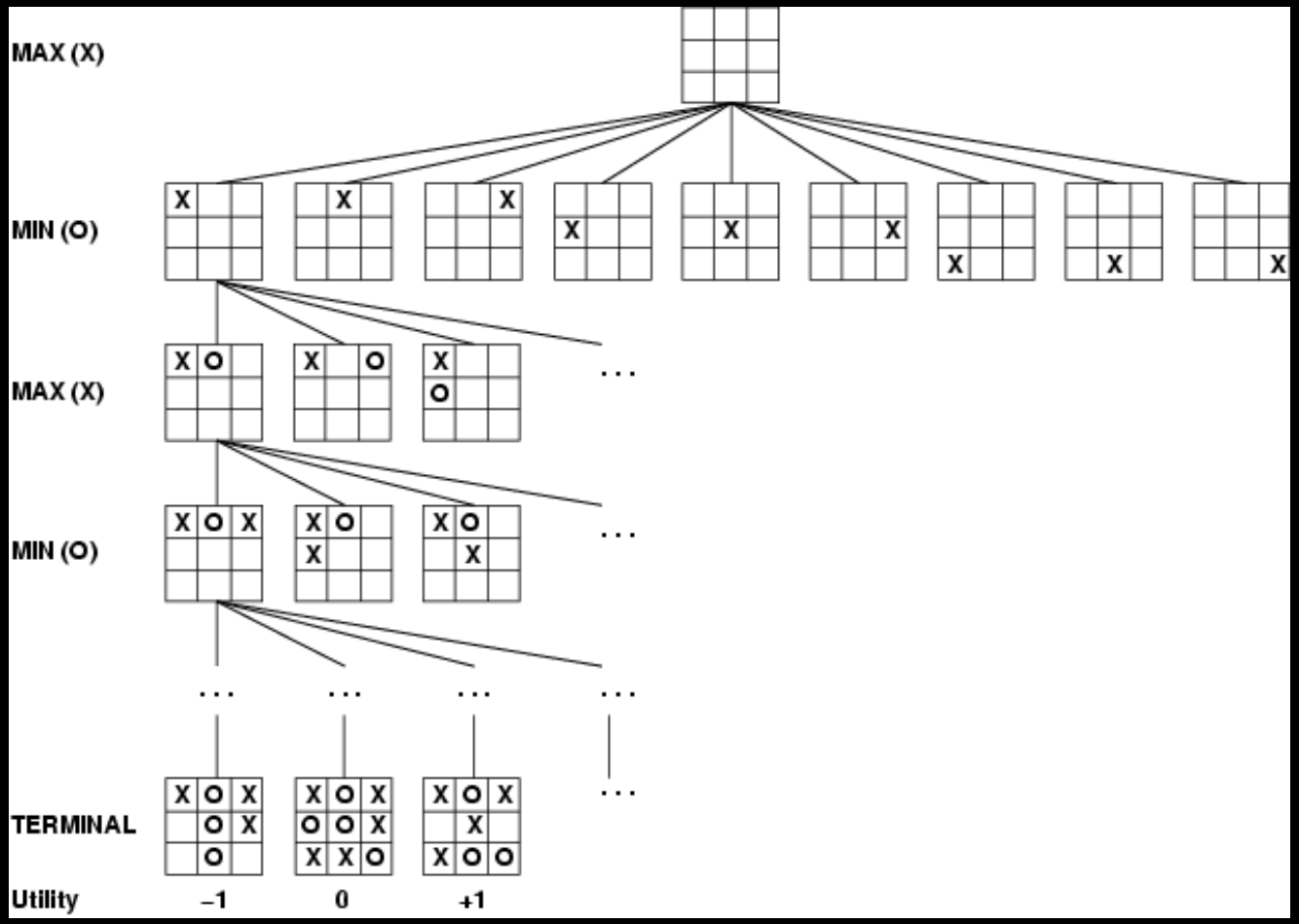
Game Playing

1. Optimal decisions
2. α-β pruning
3. Imperfect, real-time decisions

Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate

Game tree
(2-player, deterministic, turns)



Optimal strategy

- Perfect play for deterministic games
- **Minimax Value** for a node n

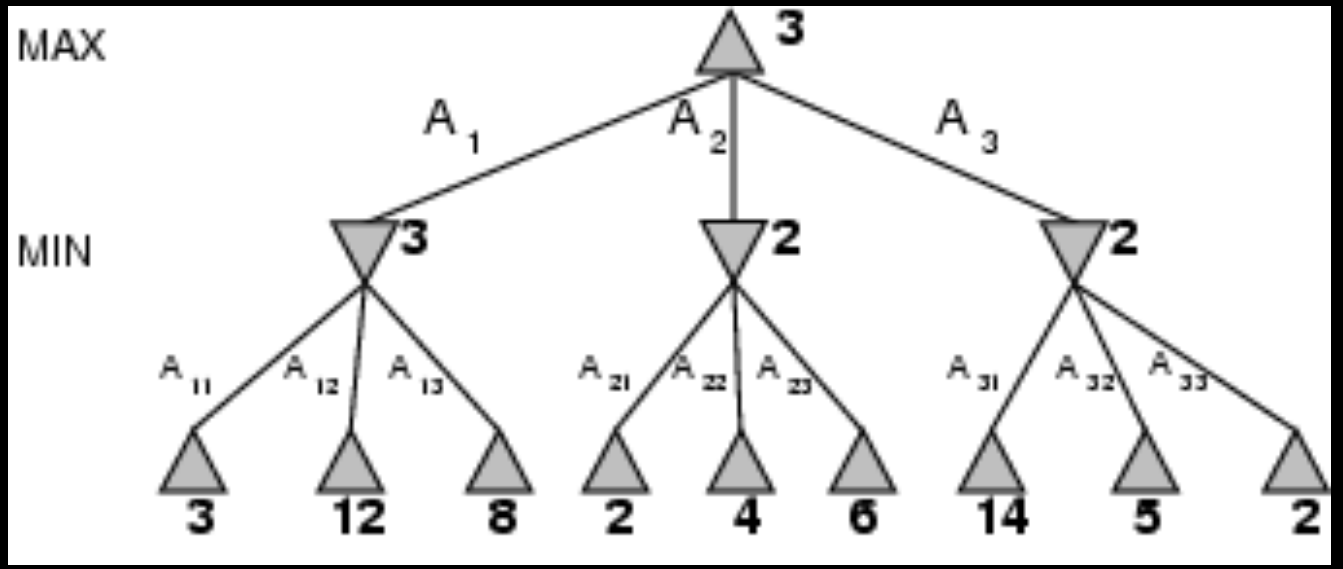
$$
\begin{array}{ll}
\text{MINIMAX-VALUE}(n) = & \\
\quad \text{UTILITY}(n) & \text{If } n \text{ is a terminal} \\
\quad \max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a max node} \\
\quad \min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a min node}
\end{array}
$$

- This definition is used recursively

- Idea: minimax value is the **best achievable payoff** against best play

Minimax example

- Perfect play for deterministic games
- **Minimax Decision** at root: choose the action a that lead to a maximal minimax value
- MAX is guaranteed for a utility which is at least the minimax value – if he plays rationally.



Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

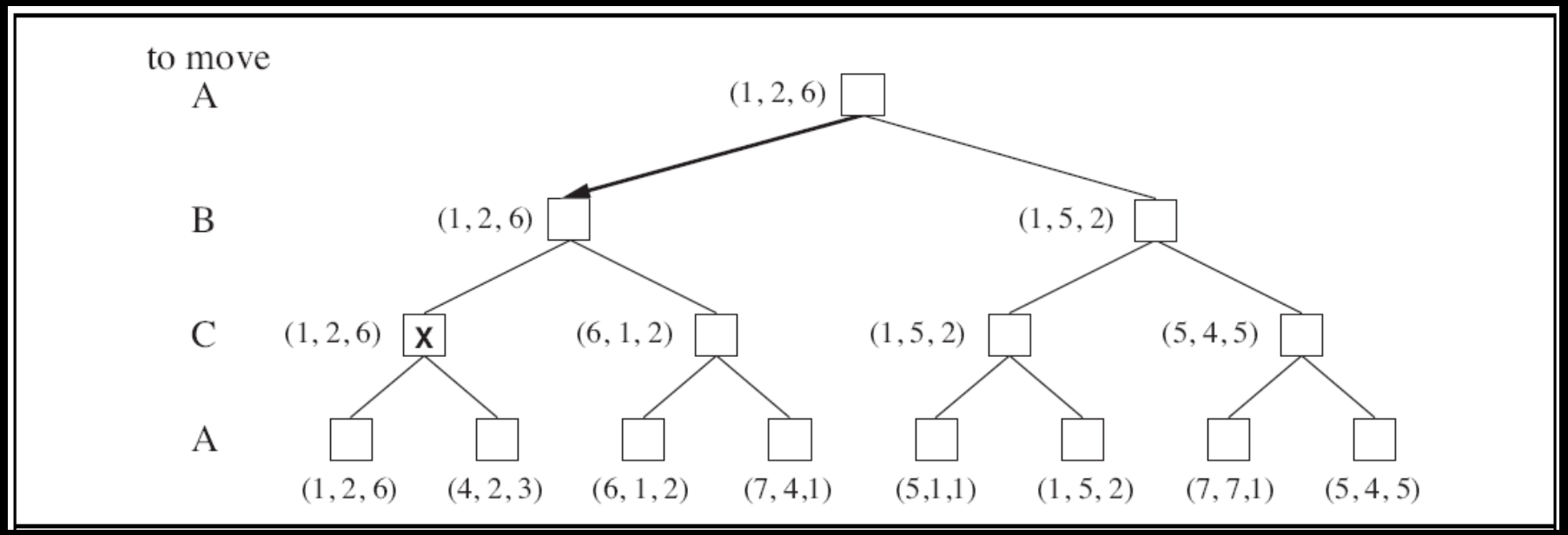
Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

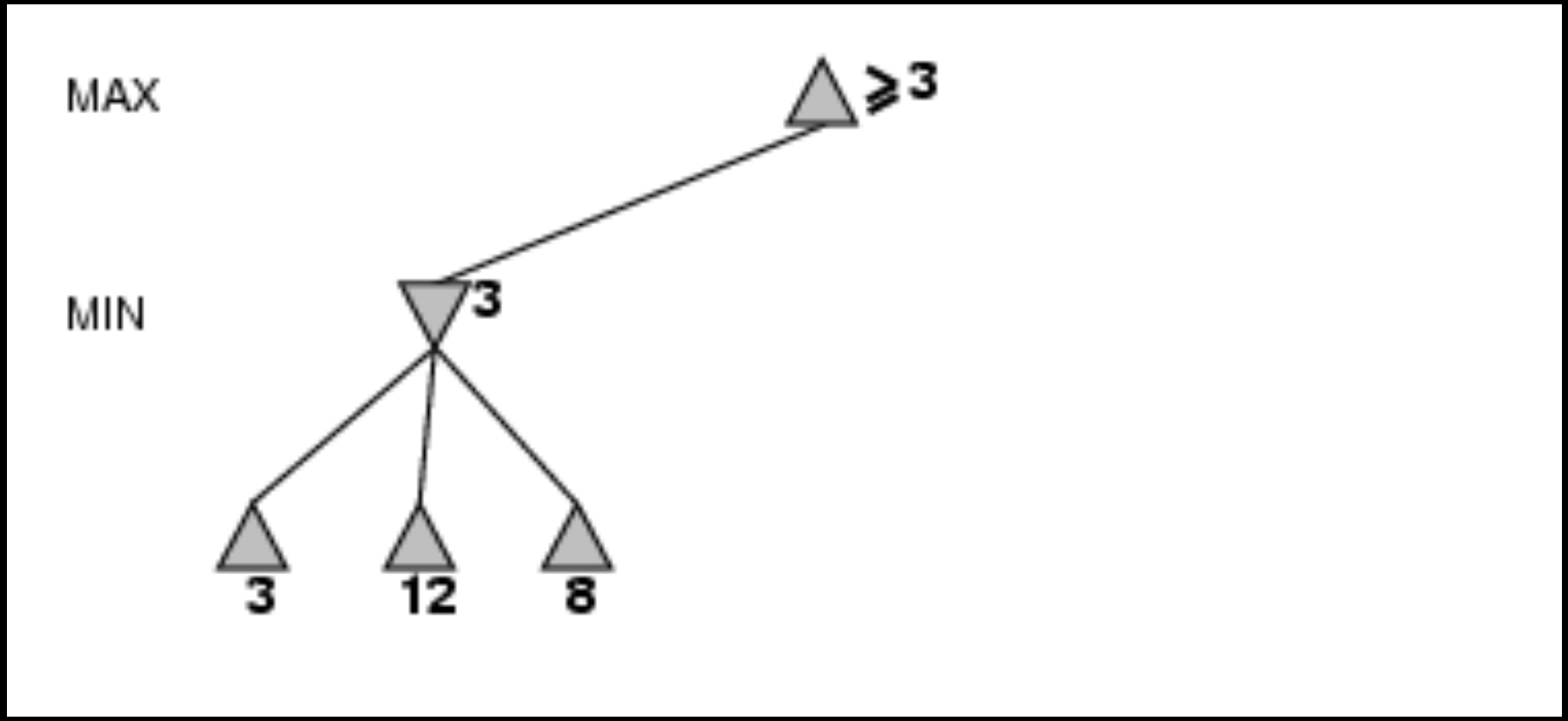
- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

Multi-player games

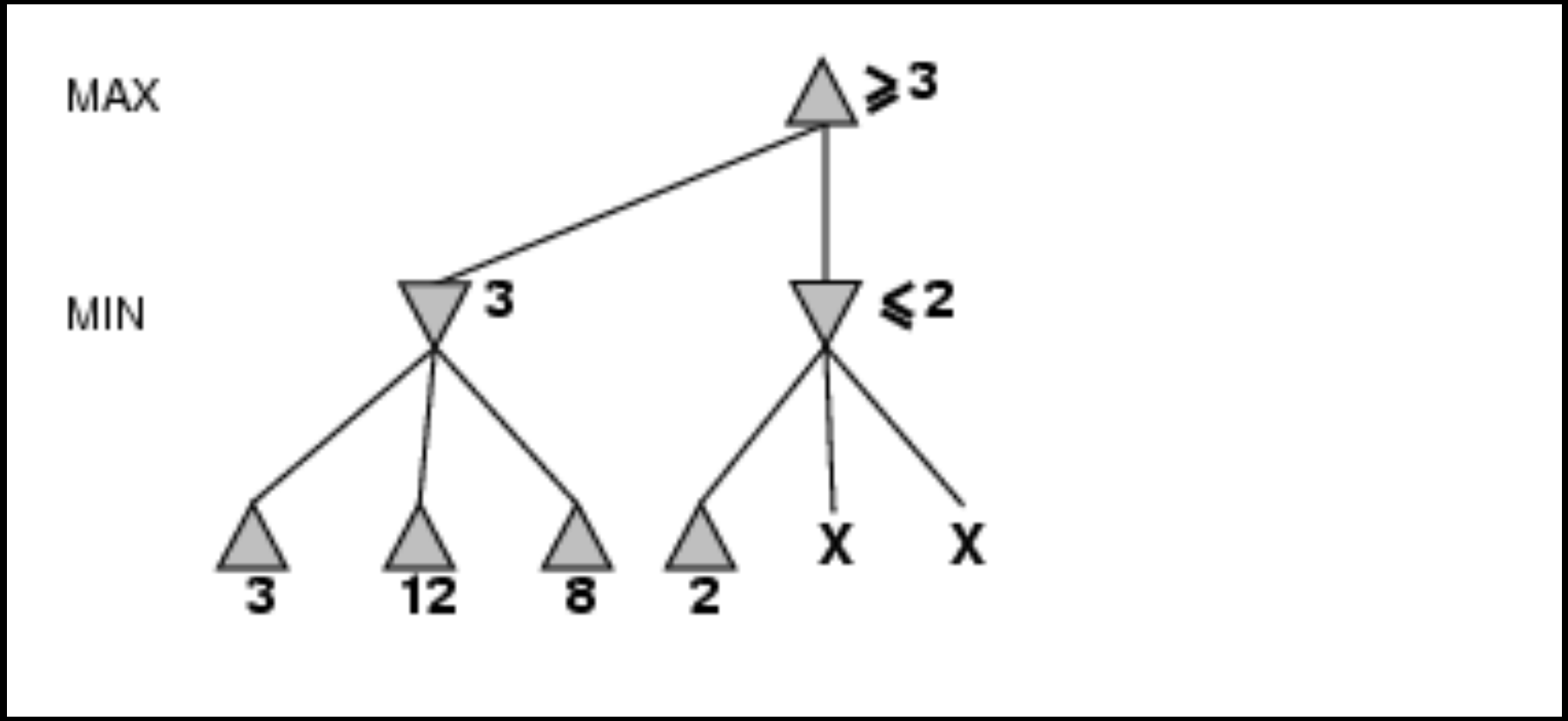
- Each node must hold a vector of values
- For example, for three players A, B, C (v_A, v_B, v_C)
- The backed up vector at node *n* will always be the one that maximizes the payoff of the player choosing at *n*



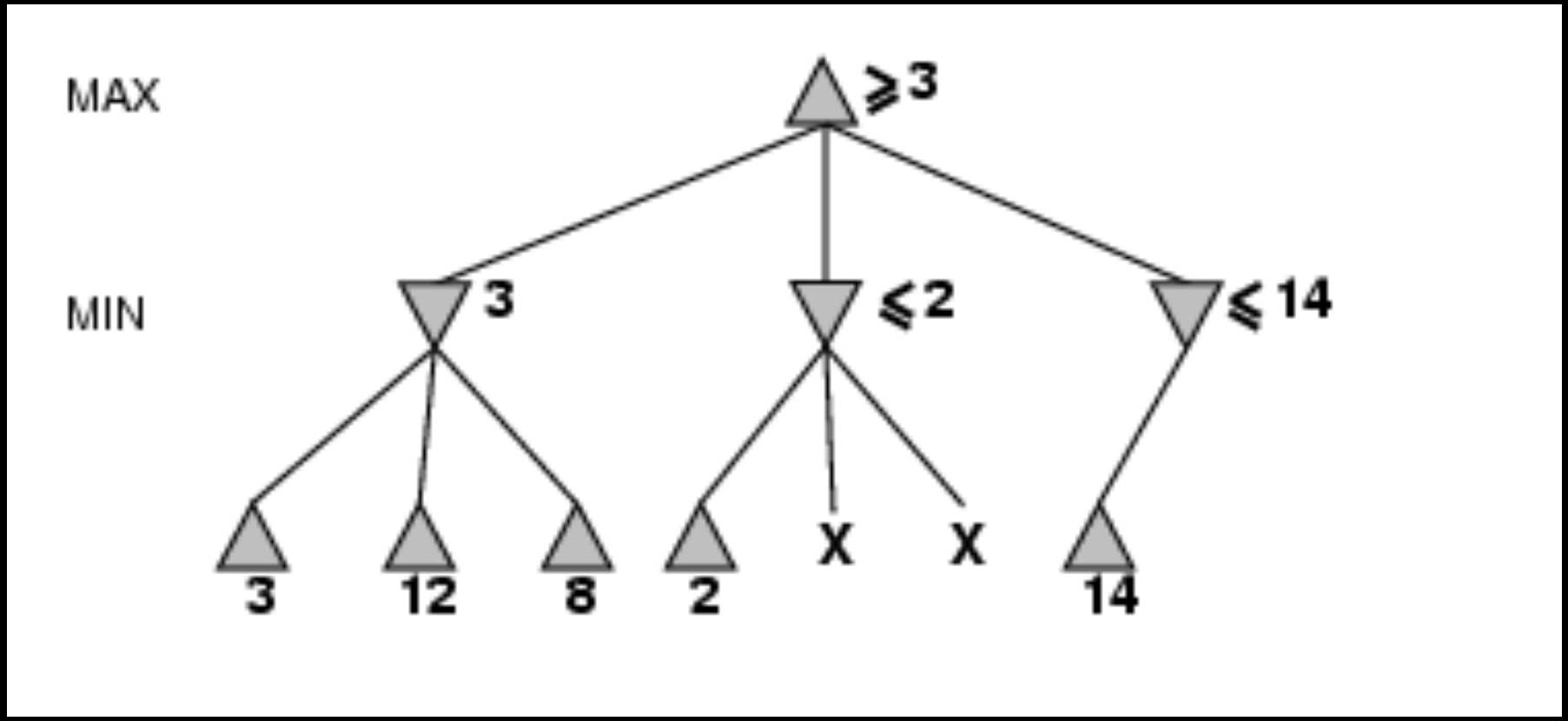
α - β pruning example



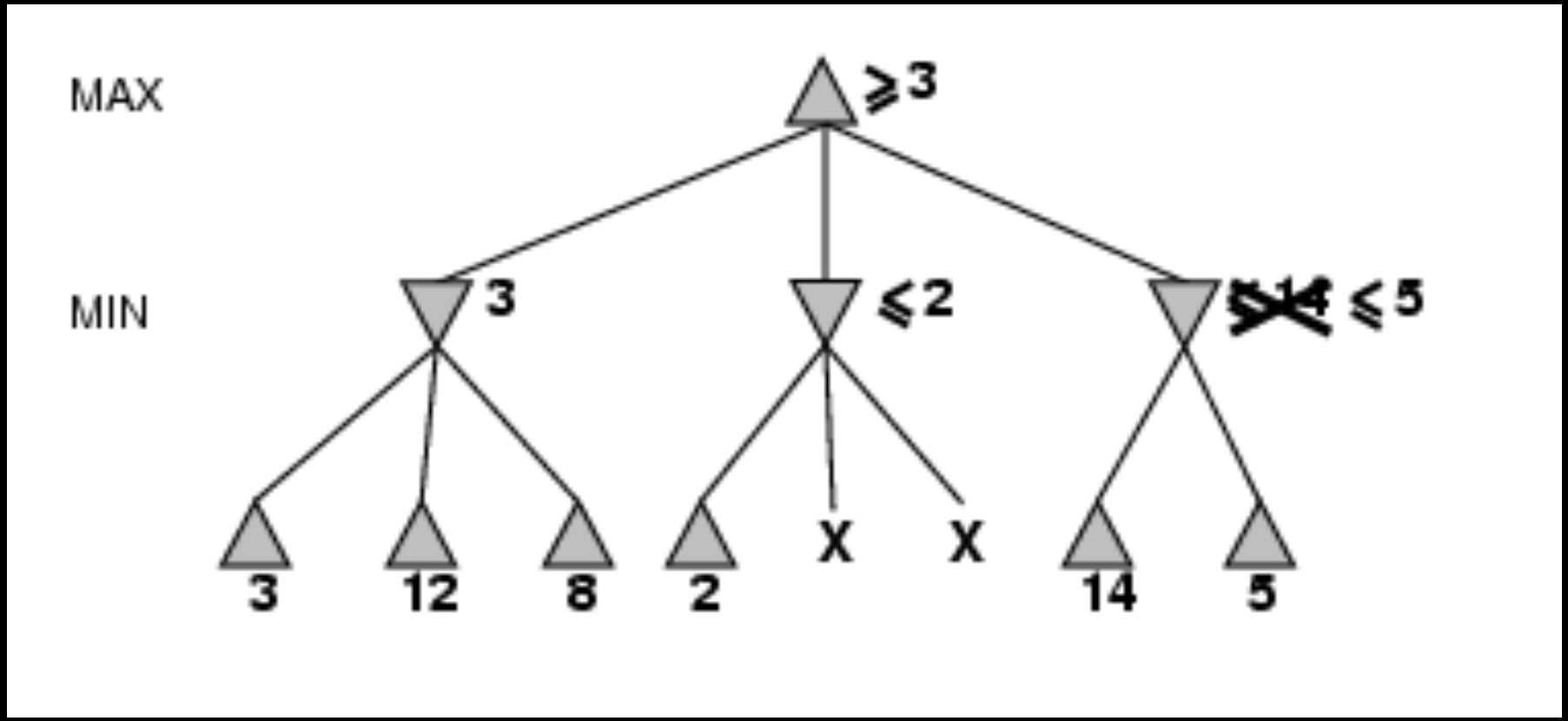
α - β pruning example



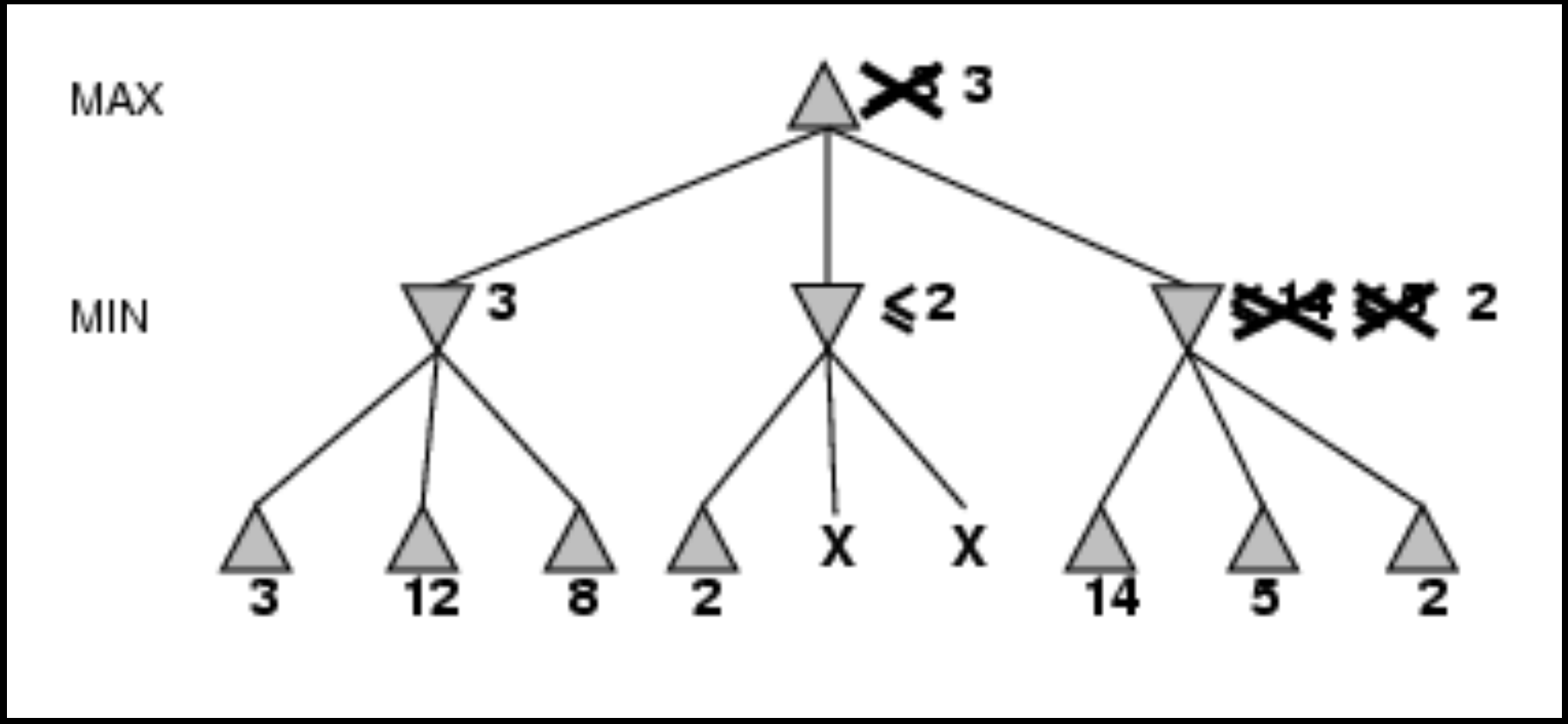
α - β pruning example



α - β pruning example



α - β pruning example



Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  → doubles depth of search
- A simple example of the value of reasoning about which computations are relevant (a form of meta-reasoning)

The α-β algorithm

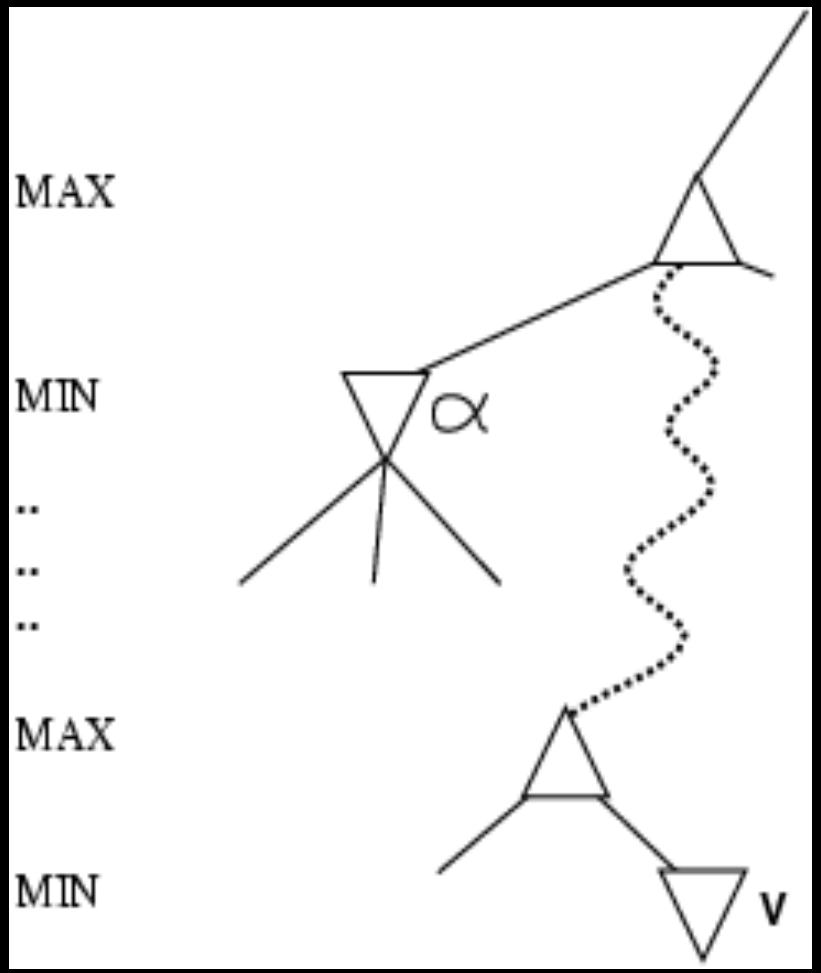
```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

---

function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

The α-β algorithm

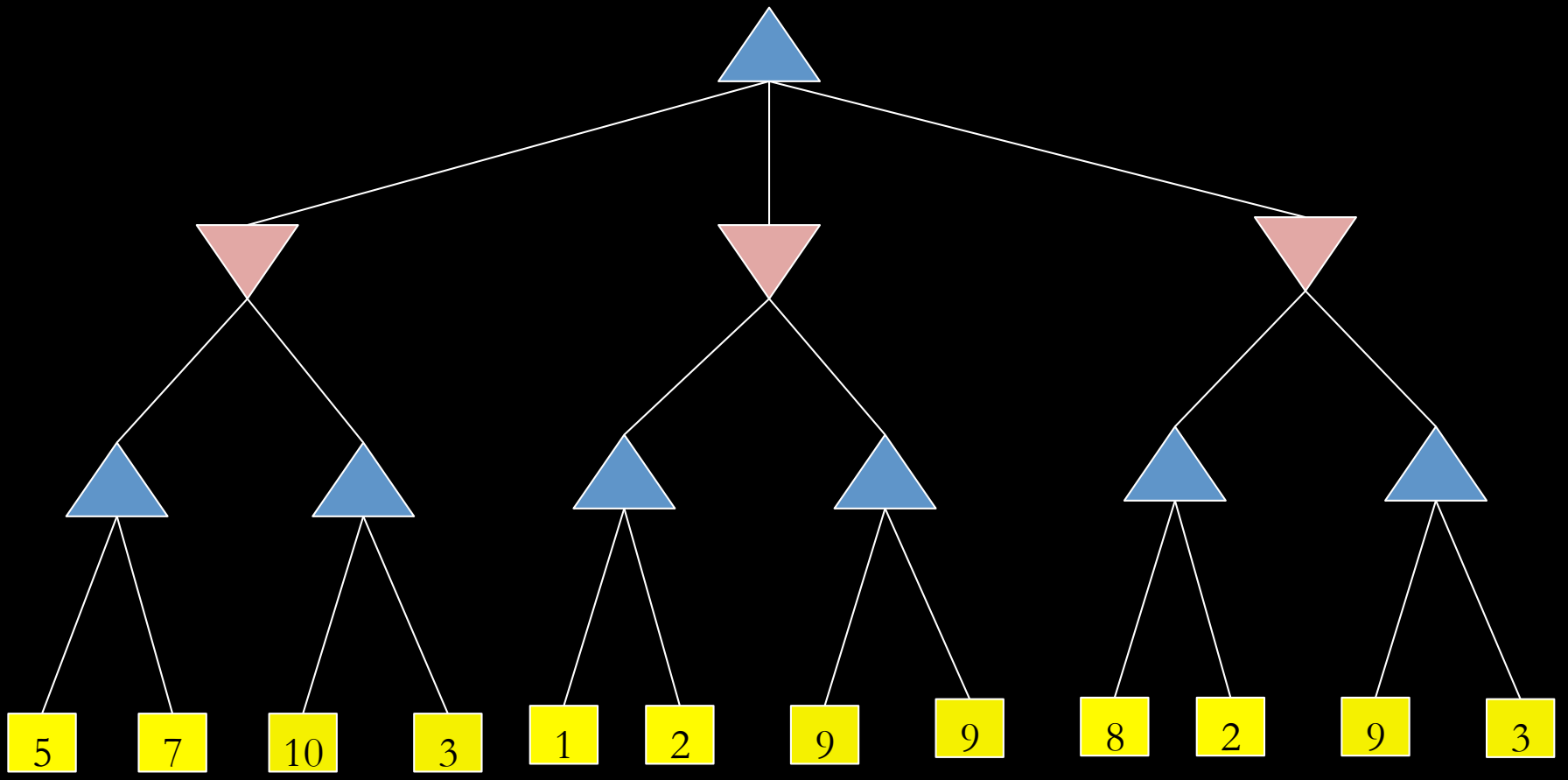
```
function MIN-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Why is it called α - β ?

- α is the value of the best (i.e., highest-value) choice found so far for *MAX* at any choice point along the path to the root.
- If *v* is worse than α, *MAX* will avoid it
  → prune that branch
- β is the value of the best (i.e., lowest-value) choice found so far for MIN at any choice point along the path for to the root.



Another example



How much do we gain?

- Assume a game tree of uniform branching factor b
- Minimax examines $O(b^d)$ nodes, so does alpha-beta in the worst-case
- The gain for alpha-beta is maximum when:
  - The MIN children of a MAX node are ordered in decreasing backed up values
  - The MAX children of a MIN node are ordered in increasing backed up values
- Then alpha-beta examines $O(b^{d/2})$ nodes [Knuth and Moore, 1975]
- But this requires an oracle (if we knew how to order nodes perfectly, we would not need to search the game tree)
- If nodes are ordered at random, then the average number of nodes examined by alpha-beta is $\sim O(b^{3d/4})$

Heuristic Ordering of Nodes

- Order the nodes below the root according to the values backed-up at the previous iteration
- Order MAX (resp. MIN) nodes in decreasing (increasing) values of the evaluation function computed at these nodes

Games of imperfect information

- Minimax and alpha-beta pruning require too much leaf-node evaluations.
- May be impractical within a reasonable amount of time.
- SHANNON (1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cutting off search

- Change:
  - **if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

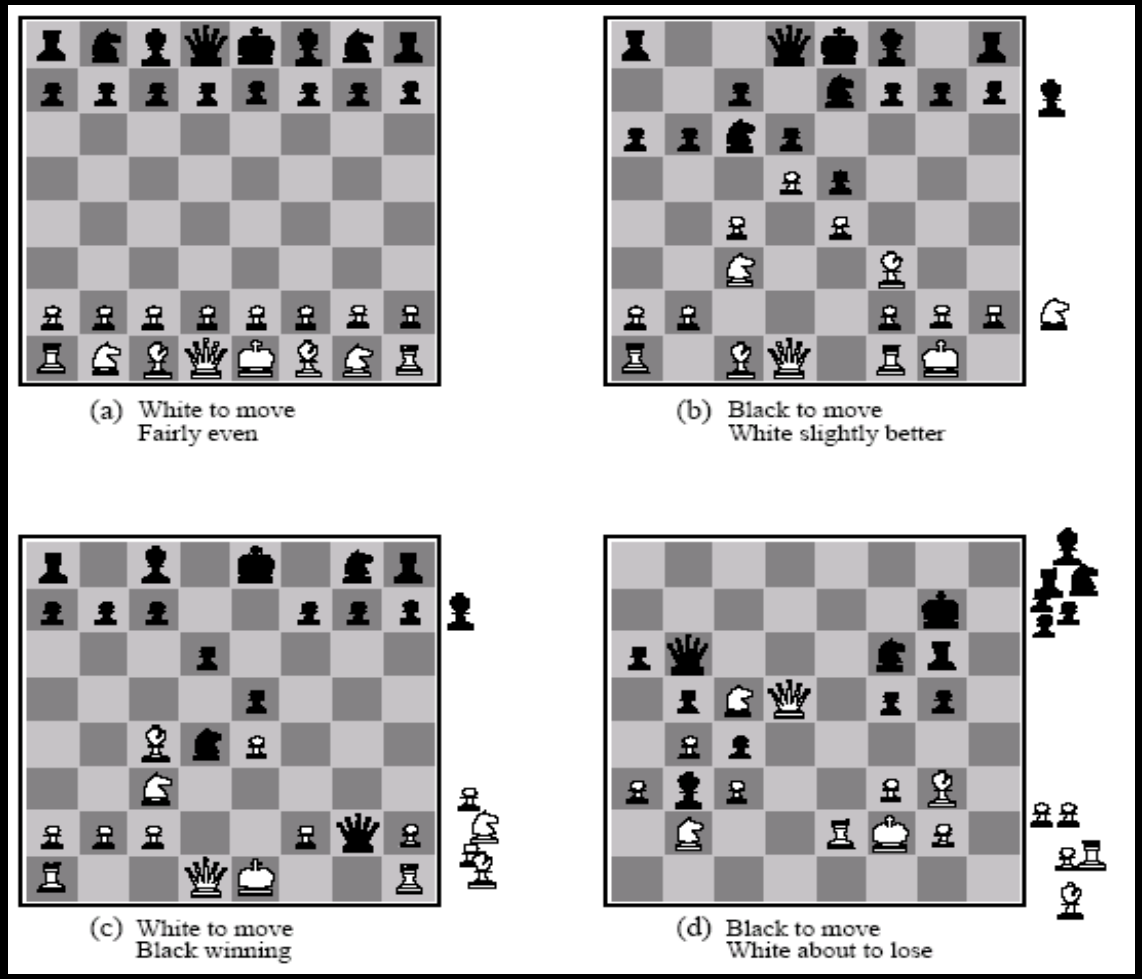
  into
  - **if** CUTOFF-TEST(*state,depth*) **then return** EVAL(*state*)

- Introduces a fixed-depth limit *depth*
  - Is selected so that the amount of time will not exceed what the rules of the game allow.
- When cutoff occurs, the evaluation is performed.

Heuristic EVAL

- Idea: produce an estimate of the expected utility of the game from a given position.
- Performance depends on quality of EVAL.
- Requirements:
  - EVAL should order terminal-nodes in the same way as UTILITY.
  - Computation may not take too long.
  - For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.
- Only useful for quiescent (no wild swings in value in near future) states

Heuristic EVAL example



(a) White to move
Fairly even

(b) Black to move
White slightly better

(c) White to move
Black winning

(d) Black to move
White about to lose

Addition assumes independence

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Heuristic difficulties

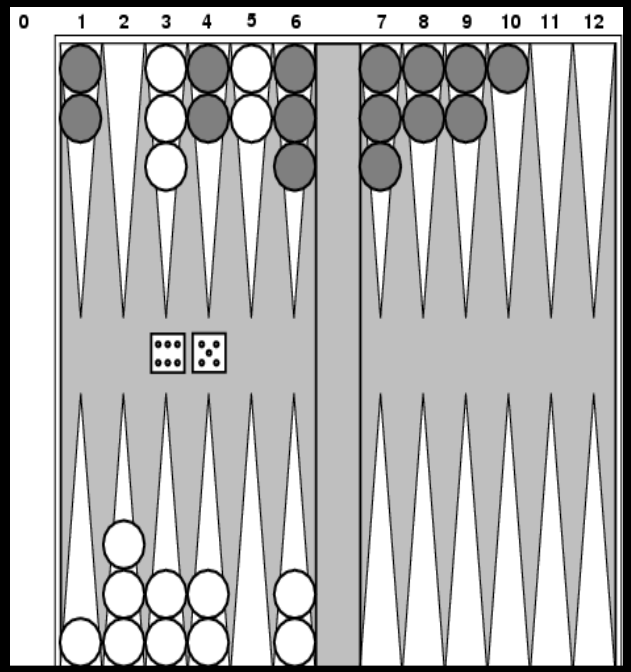
(a) White to move

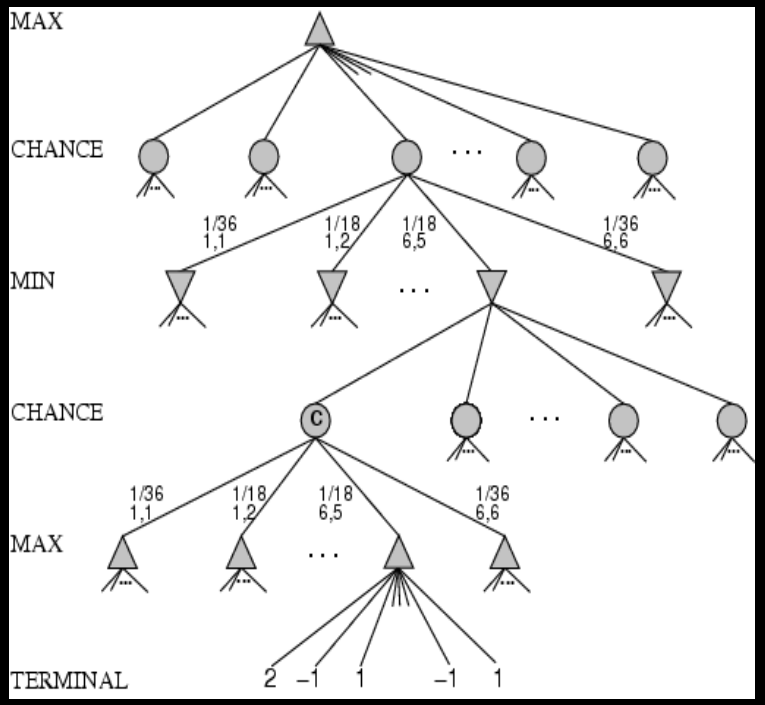
(b) White to move

Horizon effect

Fixed depth search
Makes black think it can avoid the queening move of White pawn

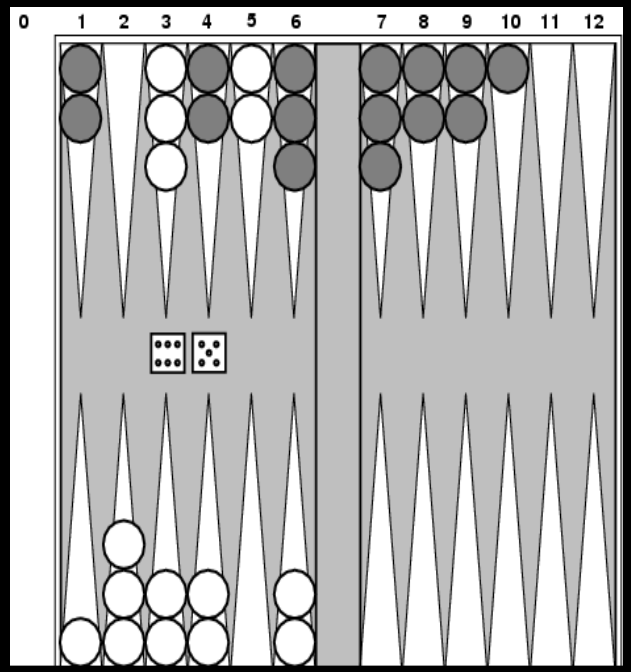
Games that include chance

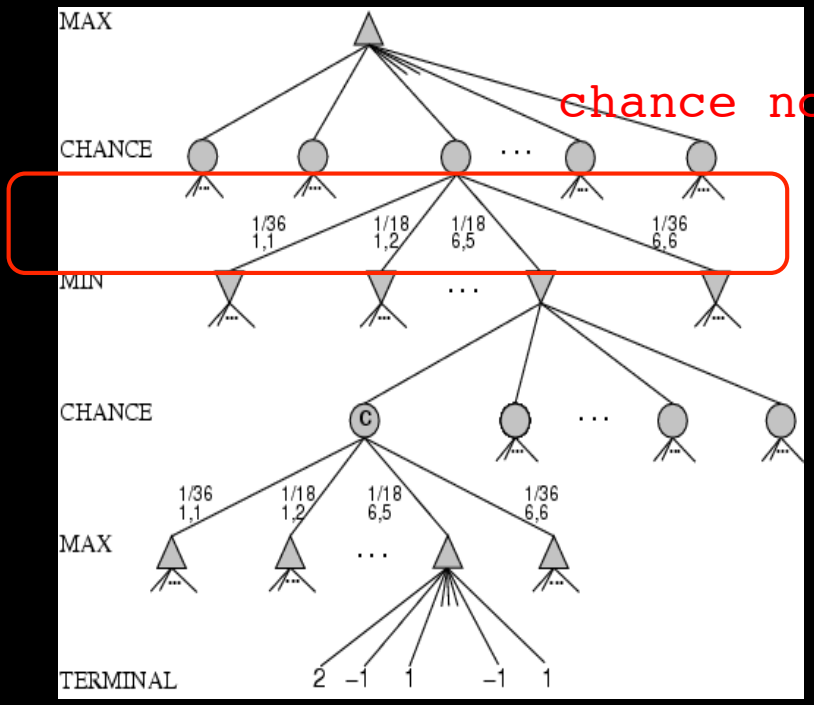




- Possible moves (5-10,5-11), (5-11,19-24), (5-10,10-16) and (5-11,11-16)

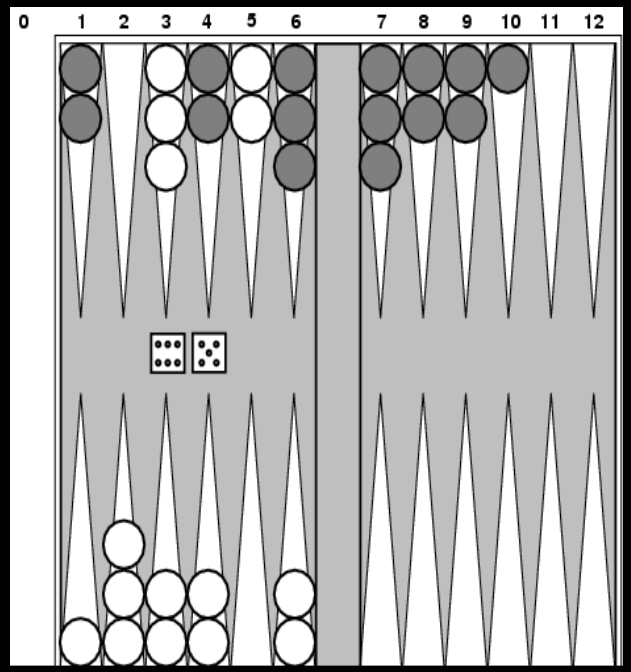
Games that include chance

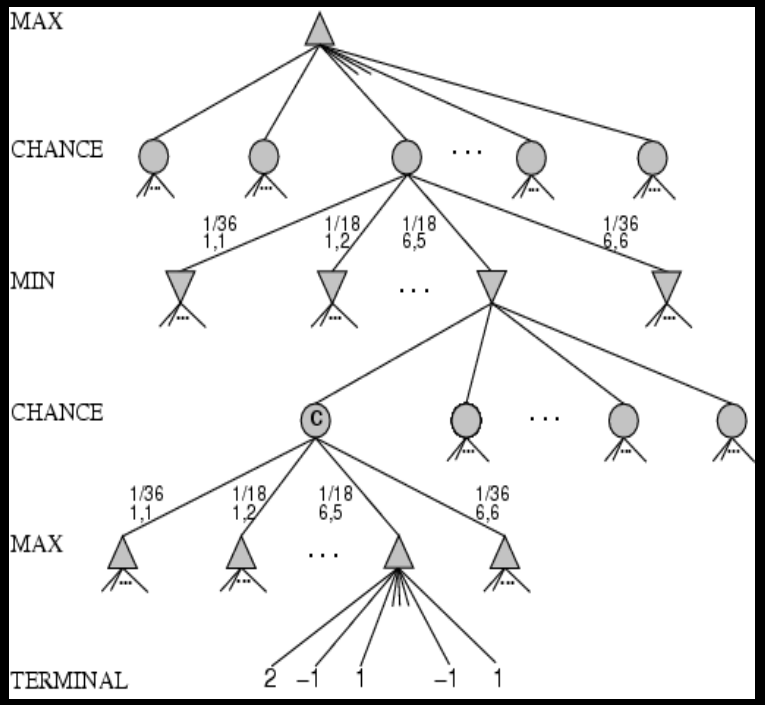




- Possible moves (5-10,5-11), (5-11,19-24),(5-10,10-16) and (5-11,11-16)
- [1,1], [6,6] chance 1/36, all other chance 1/18

Games that include chance





- [1,1], [6,6] chance 1/36, all other chance 1/18
- Can not calculate definite minimax value, only *expected* value

Expected minimax value

EXPECTED-MINIMAX-VALUE(n)=

UTILITY(n) if n is a terminal

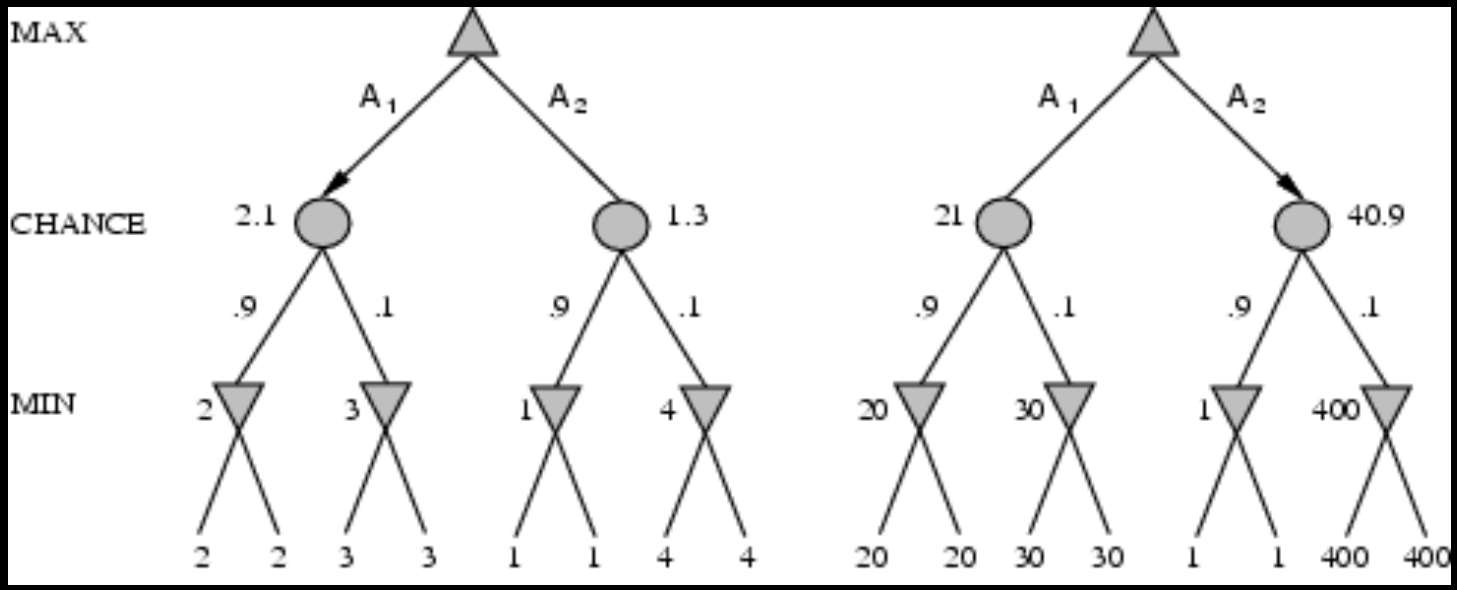
$\max_{s \in successors(n)}$ MINIMAX-VALUE(s) if n is a max node

$\min_{s \in successors(n)}$ MINIMAX-VALUE(s) if n is a max node

$\sum_{s \in successors(n)} P(s)$. EXPECTEDMINIMAX(s) if n is a chance node

These equations can be backed-up recursively all the way to the root of the game tree.

Position evaluation with chance nodes



- Left, A1 wins
- Right A2 wins
- Outcome of evaluation function may not change when values are scaled differently.
- Behavior is preserved only by a *positive linear* transformation of EVAL.

Other Types of Games

- Multi-player games, with alliances or not
- Games with randomness in successor function (e.g., rolling a dice)
  → Expectminimax algorithm
- Games with partially observable states (e.g., card games)
  → Search of belief state spaces

See R&N to know more…

Summary

- Heuristic function
- Best-first search
- Admissible heuristic and A*
- A* is complete and optimal
- Consistent heuristic and repeated states
- Heuristic accuracy
- AO*, IDA*
- Game playing (MinMax, alpha/beta, eval, …)