Unit 2: Problem Solving

Dr. Ashu Abdul

Assistant Professor
Department of Computer Science and Engineering
SRM University-AP
Andhra Pradesh - 522503, India
Email: ashu.a@srmap.edu.in

Topics to be discussed

1. Problem Solving
2. State Space Search, Blind and Informed Search
3. Constraint Satisfaction
   - Game Tree,
   - Evaluation Function,
   - Mini-Max Search,
   - Alpha-Beta Pruning,
   - Games of Chance

Problem Solving

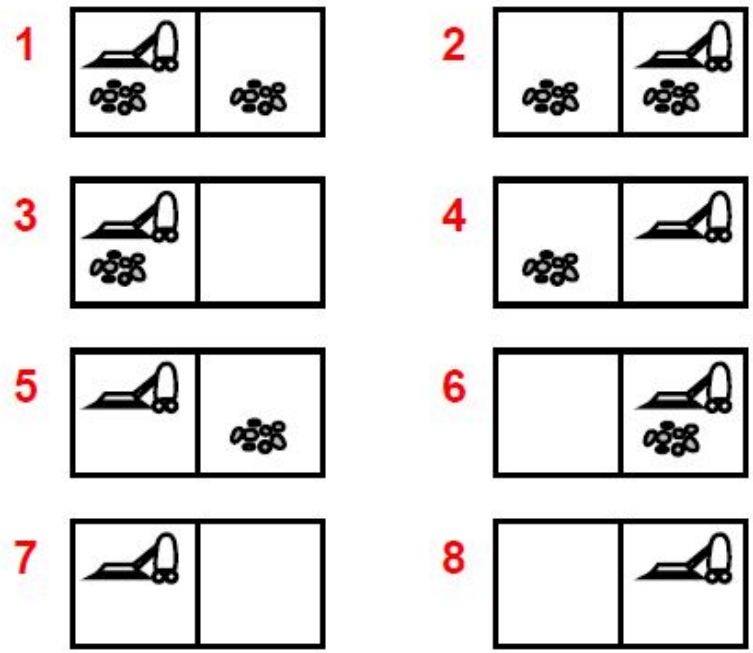
- A **goal** and **a set of means for achieving the goal** are called a **problem**, and the process of exploring what the means can do is called **search**.
- **Problem-solving agents** decide what to do by **finding sequences of actions** that lead to desirable states.
- **Goal formulation**, based on the **current situation**, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider and follows goal formulation.
- We have to "**formulate**, **search**, and **execute phase**" design for the **agent**.
- After formulating a **goal** and a **problem** to solve, the agent calls a **search** procedure to solve it.

Formulating Problems

- There are **four** essentially different types of problems.
  1. Single state problems (deterministic, fully observable)
  2. Multiple-state problems (non-observable)
  3. Contingency problems (nondeterministic and/or partially observable)
  4. Exploration problems (unknown state space - "online")

Formulating Problems

- Let the world contain just **two locations**.
- Each location may or may not contain **dirt**, and the agent may be in one location or the other.
- There are **8 possible world states**.
- The agent has **three possible actions** in this version of the vacuum world: **Left**, **Right** and **Suck**.
- The goal is equivalent to the state set $\{7, 8\}$.



in robotics, the exploration problem deals with the use of a robot to maximize the knowledge over a particular area

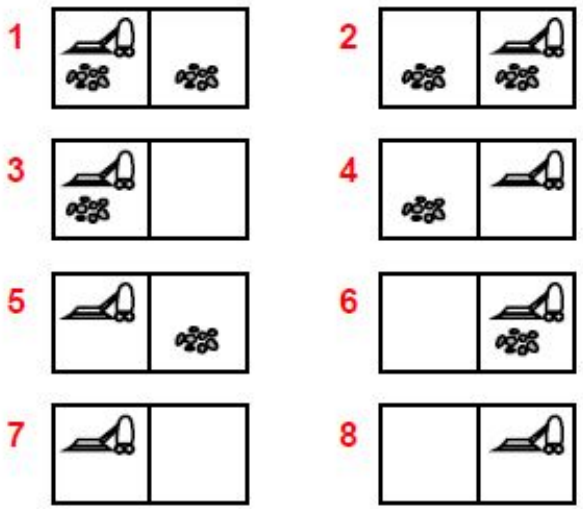
Formulating Problems

Example: vacuum world

Single-state, start in #5. Solution??
[*Right*, *Suck*]

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

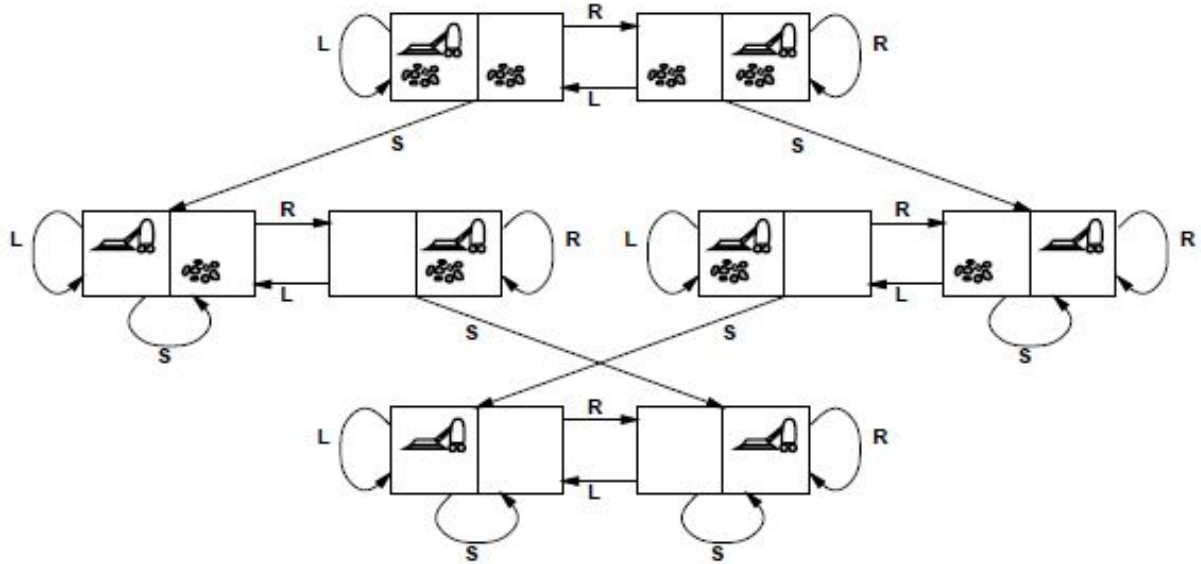
Contingency, start in #5
Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
[*Right*, if *dirt* then *Suck*]



- Guaranteed to reach a goal state: [Right, Suck, Left, Suck]
- The agent learns a "map" of the environment, which it can then use to solve subsequent problems. We call this an **exploration problem**.

Vacuum World State Space Graph

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??: 1 per action (0 for $NoOp$)

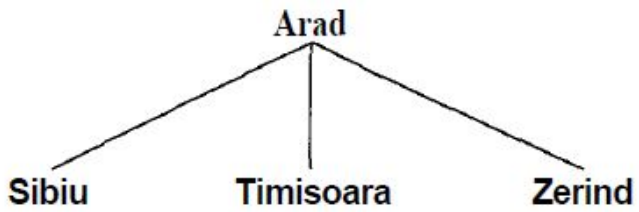
Well-defined Problems and Solutions - State Space Search

- The **initial state** that the agent knows itself to be in.
- **Successor function:** Given a particular state x, $S(x)$ returns the set of states reachable from x by any single action.
- **State space:** The set of all states reachable from the initial state by any sequence of actions.
- A **Path** in the state space is simply any sequence of actions leading from one state to another.
- A **Path cost** function is a function that assigns a cost to a path.
- Together, the **initial state**, **operator set**, **goal test** and **path cost** function define a **problem**.
- The output of a search algorithm is a **solution**, that is, a path from the initial state to a state that satisfies the **goal test**.
- The **total cost** of the search is the **sum** of the **path cost** and the **search cost**.

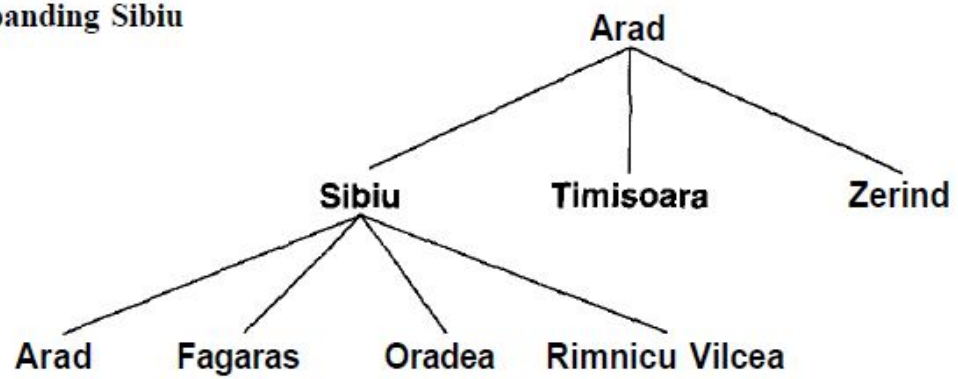
Formulating Problems: Single-State Problem Formulation

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu



Single-state problem formulation

A problem is defined by four items:

initial state e.g., "at Arad"

successor function $S(x)$ = set of action–state pairs
e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

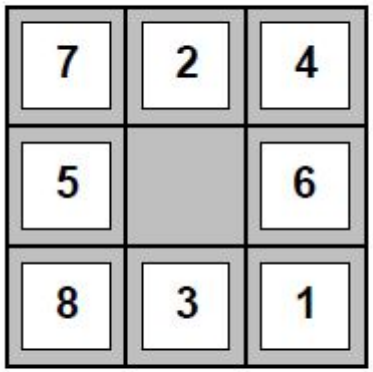
goal test, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., $NoDirt(x)$

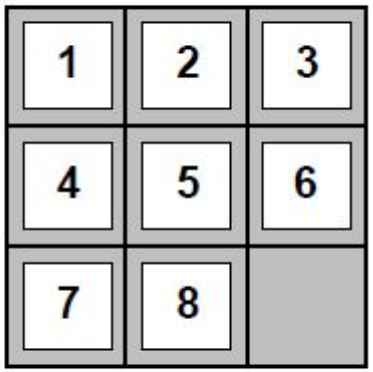
path cost (additive)
e.g., sum of distances, number of actions executed, etc.
$c(x, a, y)$ is the step cost, assumed to be ≥ 0

A solution is a sequence of actions

Example Problems: Toy Problems/The 8-Puzzle

Example: The 8-puzzle





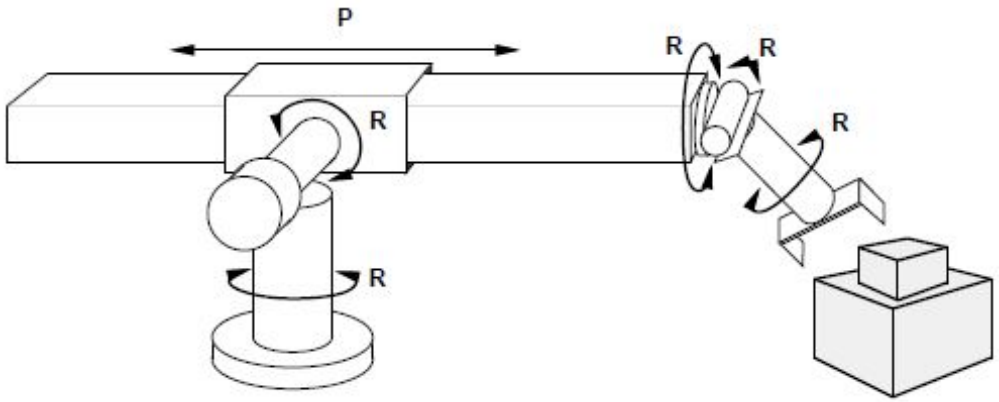
states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

[Note: optimal solution of n-Puzzle family is NP-hard]

- The 8-puzzle belongs to the family of **sliding-block puzzles**.

Example Problems: Robotic Assembly

Example: robotic assembly



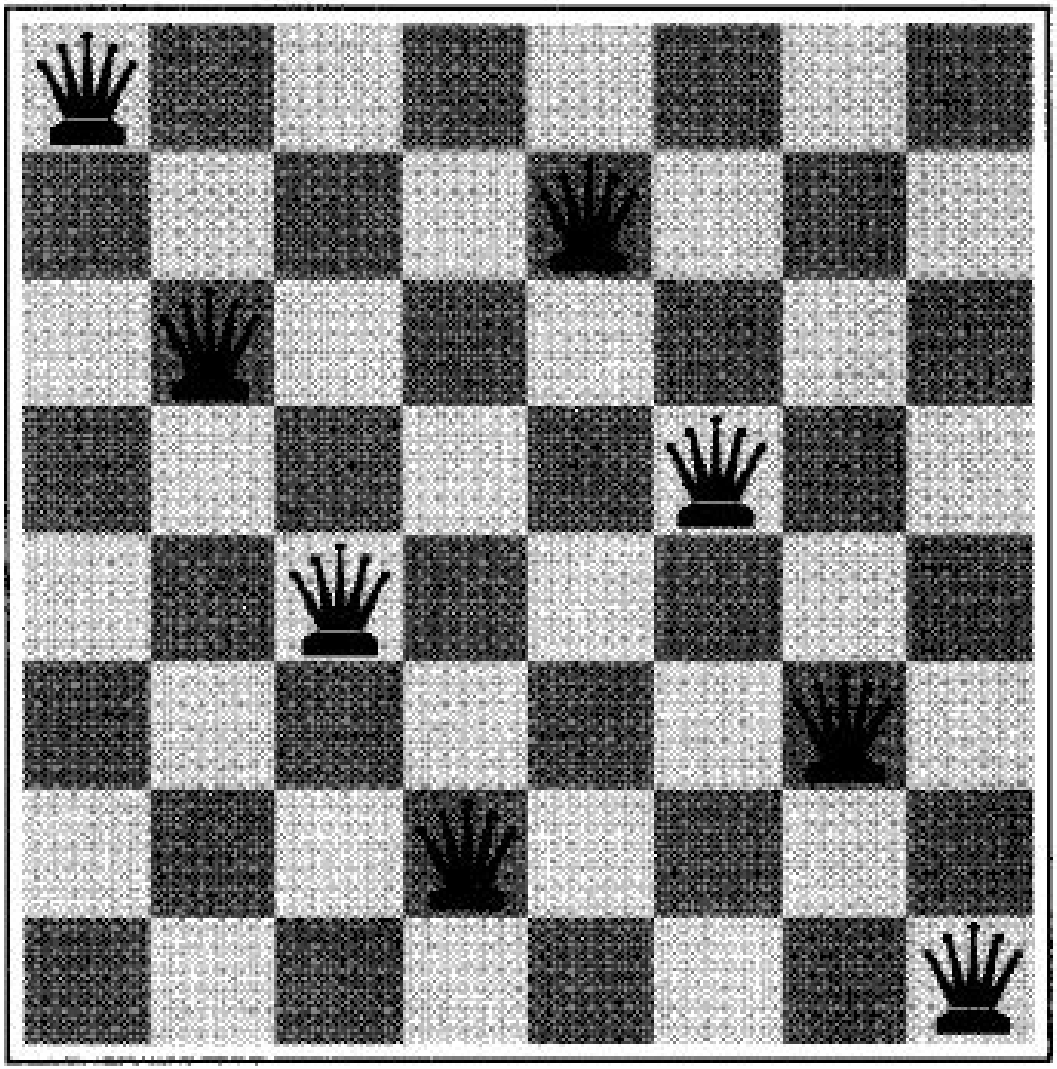
states??: real-valued coordinates of robot joint angles
parts of the object to be assembled

actions??: continuous motions of robot joints

goal test??: complete assembly **with no robot included!**

path cost??: time to execute

Example Problems: 8-Queen Problem



Example Problems: 8-Queen Problem

The goal of the **8-Queen problem** is to place **eight queens** on a **chessboard** such that **no queen attacks any other**. (A queen attacks any piece in the **same row**, **column** or **diagonal**)

The queen in the **rightmost column** is attacked by the queen at **top left**.

Goal test: ?
Path cost: ?
States: ?
Operators: add a queen to any square.
We have . . . possible sequences to investigate.
Operators:?

Example Problems: 8-Queen Problem

The goal of the **8-Queen problem** is to place **eight queens** on a **chessboard** such that **no queen attacks any other**. (A queen attacks any piece in the **same row**, **column** or **diagonal**)

The queen in the **rightmost column** is attacked by the queen at **top left**.

Goal test: 8 queens on board, none attacked.
Path cost: zero
States: any arrangement of 0 to 8 queens on board.
Operators: add a queen to any square.
We have 64^8 possible sequences to investigate.
States: arrangements of 0 to 8 queens **with none attacked**.
Operators: Place a queen in the left-most empty column such that it is not attacked by any other queen.

A quick calculation shows that there are only 2057 possible sequences to investigate.

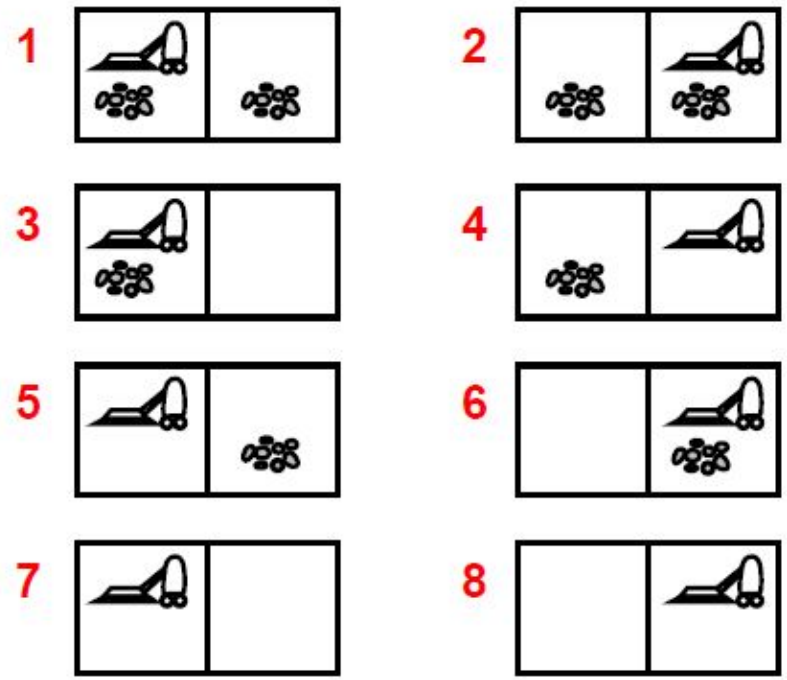
Example Problems: Crypt arithmetic

States: a cryptarithmetic puzzle with some letters replaced by digits.
Operators: Replace all occurrences of a letter with a digit not already appearing in the puzzle.
Goal test: Puzzle contains only digits, and represents a correct sum.
Path cost: Zero. All solutions equally valid.

```
 FORTY      Solution:    29786      F=2,  0=9,  R=7,  etc.
+  TEN                     850
+  TEN                     850
 -----                   -----
 SIXTY                   31486
```

Example Problems: The Vacuum World

States: one of the eight states
Operators: move left, move right, suck.
Goal test: no dirt left in any square.
Path cost: each action costs 1.



Example Problems: Missionaries and Cannibals Problem

Cannibals - A person who eats other human beings
State space - [Missionaries, Cannibals, Boat]
Initial State - [3, 3, 1]
Goal State - [0, 0, 0]
Operators - adding or subtracting the vectors [1, 0, 1], [2, 0, 1], [0, 1, 1], [0, 2, 1] or [1, 1, 1]
Path - moves from [3, 3, 1] to [0, 0, 0]
Path Cost - river trips / number of crossings

Example Problems: Missionaries and Cannibals Problem

Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

Solution

1. Move 2 cannibals to the right
2. Move 1 cannibal back to the left
3. Move 2 cannibals to the right
4. Move 1 cannibal back to the left
5. Move 2 missionaries to the right
6. Move 1 missionary and 1 cannibal back to the left
7. Move 2 missionaries to the right
8. Move 1 cannibal back to the left
9. Move 2 cannibals to the right
10. Move 1 cannibal back to the left
11. Move 2 cannibals to the right

Water Jug Problem

Problem

You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug and the ground on which water may be poured.
Neither jug has any measuring markings on it.
How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State

We will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug.
Note $0 \leq x \leq 4$ and $0 \leq y \leq 3$.
Our initial state: $(0, 0)$

Goal Predicate

State $= (2, y)$ where $0 \leq y \leq 3$.

Water Jug Problem: Operators

We must define a set of operators that will take us from one state to another.

1. Fill 4-gal jug	(x, y) $x < 4$	$\rightarrow$	$(4, y)$
2. Fill 3-gal jug	(x, y) $y < 3$	$\rightarrow$	$(x, 3)$
3. Empty 4-gal jug on ground	(x, y) $x > 0$	$\rightarrow$	$(0, y)$
4. Empty 3-gal jug on ground	(x, y) $y > 0$	$\rightarrow$	$(x, 0)$
5. Pour water from 3-gal jug to fill 4-gal jug	(x, y) $0 < x + y \geq 4$ and $y > 0$	$\rightarrow$	$(4, y - (4 - x))$
6. Pour water from 4-gal jug to fill 3-gal jug	(x, y) $0 < x + y \geq 3$ and $x > 0$	$\rightarrow$	$(x - (3 - y), 3)$
7. Pour all of water from 3-gal jug into 4-gal jug	(x, y) $0 < x + y \leq 4$ and $y \geq 0$	$\rightarrow$	$(x + y, 0)$
8. Pour all of water from 4-gal jug into 3-gal jug	(x, y) $0 < x + y \leq 3$ and $x \geq 0$	$\rightarrow$	$(0, x + y)$

Water Jug Problem: Solution

Gals in 4-gal jug	Gals in 3-gal jug	Rule Applied
0	0	-
4	0	1. Fill 4
1	3	6. Pour 4 into 3 to fill
1	0	4. Empty 3
0	1	8. Pour all of 4 into 3
4	1	1. Fill 4
2	3	6. Pour into 3

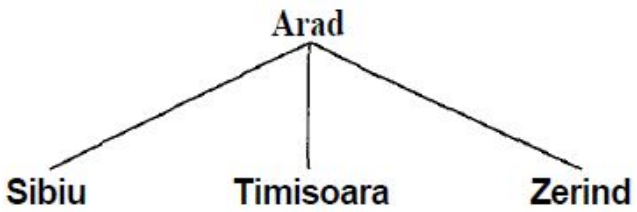
Real-World Problems

- Route Finding
- Touring and travelling salesperson problems
- VLSI layout: cell layout and channel routing
- Robot navigation
- Assembly sequencing

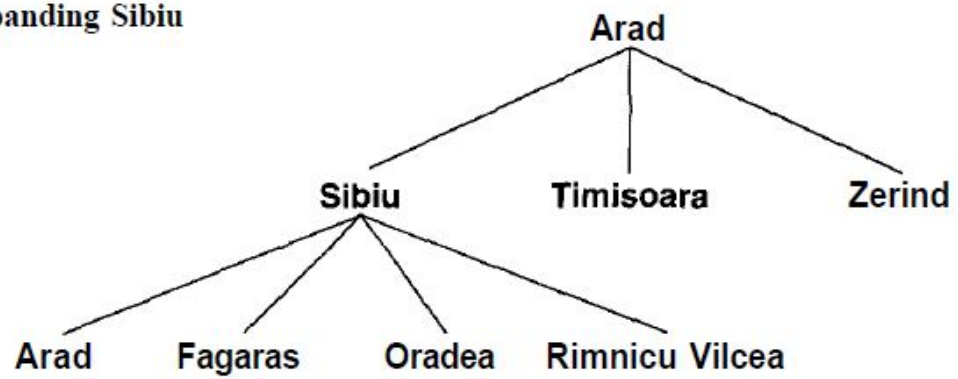
Searching for Solutions

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu



Searching for Solutions

- There are many ways to represent nodes.
- We will assume a **node** is a **data structure** with **five components**.
  1. the **state** in the **state space** to which the node corresponds
  2. the **node** in the **search tree** that generated this node (this is called the **parent node**)
  3. the **operator** that was applied to generate the node;
  4. the **number of nodes** on the path from the **root to this node** (the **depth** of the node);
  5. the **path cost** of the path from the **initial state to the node**
- The collection of nodes that are **waiting to be expanded** is called the **fringe** or **frontier**.

Search Strategies

- Evaluate strategies in terms of **four** criteria.
  1. **Completeness:** is the strategy guaranteed to find a solution when there is one?
  2. **Time complexity:** how long does it take to find a solution?
  3. **Space complexity:** how much memory does it need to perform the search?
  4. **Optimality:** does the strategy find the highest-quality solution when there are several different solutions?

Uninformed/Blind Search

No preference is given to the order of successor node generation and selection.

1. Breadth-First Search
2. Uniform-Cost Search
3. Depth-First Search
4. Depth-Limited Search
5. Iterative Deepening Search

Informed Search

Some information about the problem space is used to compute a preference among the children for exploration and expansion.

1. Heuristic Function
2. Best First Search
3. A$*$ Search
4. AO$*$ Search
5. Hill Climbing Search

Uninformed/Blind Search - Breadth-First Search

Algorithm

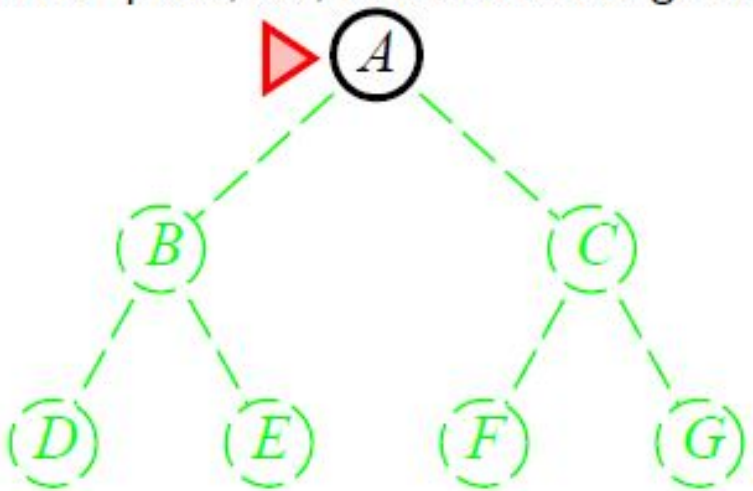
1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element from the queue and place all the children at the end of the queue in **any order**.
5. Return to step 2.

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



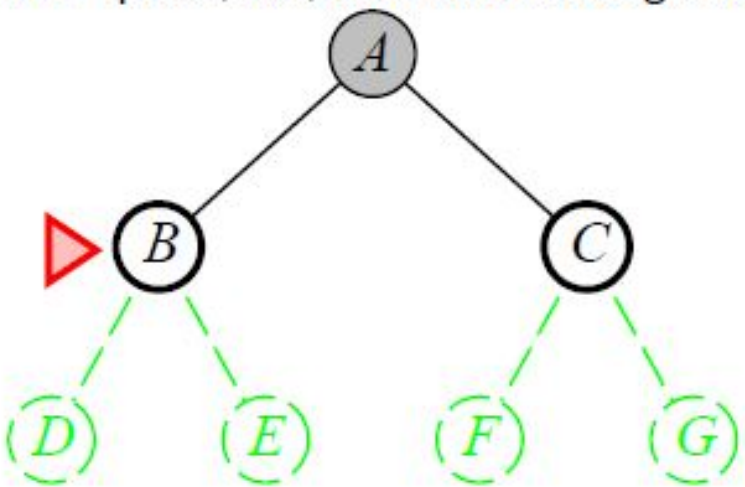
- The root node is expanded first, then all the nodes generated by the root node are expanded next.
- All the nodes at depth d in the search tree are expanded before the nodes at depth $d + 1$

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

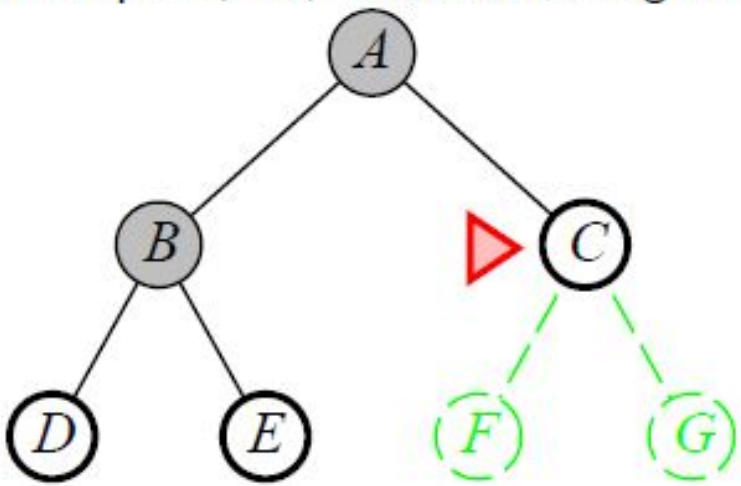


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

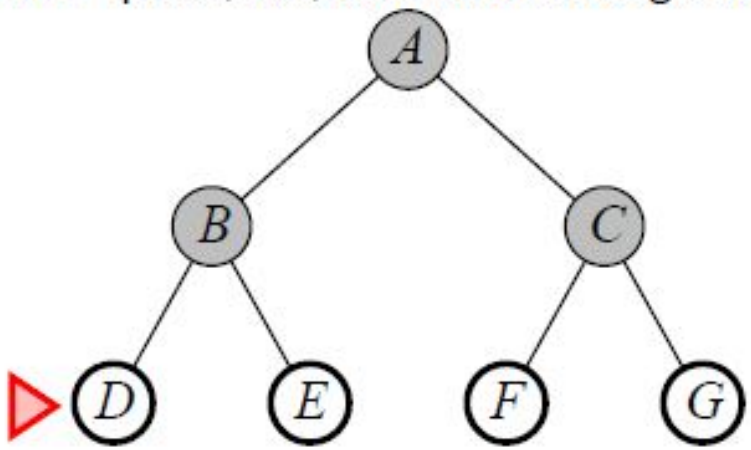


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Uninformed/Blind Search - Breadth-First Search

- b - maximum **branching factor** of the search tree
- d - **depth** of the least-cost solution
- m - maximum depth of the state space (may be ∞)

Properties

1. **Complete:** Yes (if b is finite)
2. **Time:** $1 + b + b^2 + \cdots + b^d = O(b^d)$
3. $O(b^d)$ (keeps every node in memory)
4. **Optimal:** Yes (if cost $= 1$ per step); not optimal in general
5. It is **optimal** provided the **path cost** is a **nondecreasing function** (i.e., 7, 8, 8, 10, 11, 11, 12) of the depth of the node. (This condition is usually satisfied only when **all operators** have the **same cost**.)

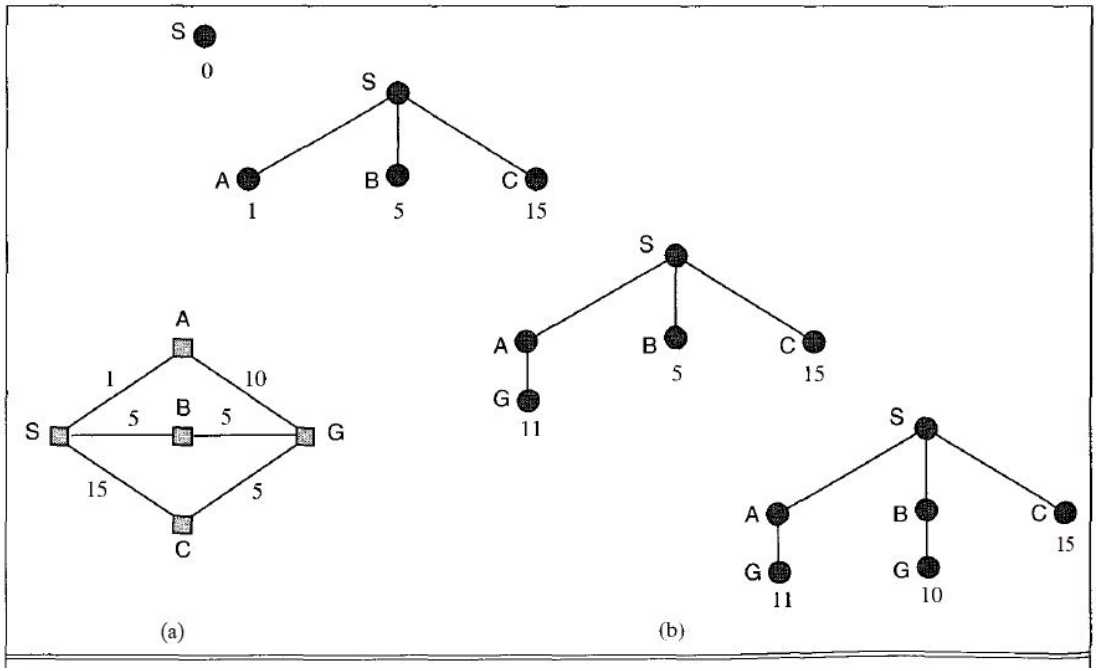
Uninformed/Blind Search - Breadth-First Search

- The memory requirements are a bigger problem for breadth-first search than the execution time.
- $b = 10$, $d = 2$, Nodes $= 1 + 10 + 10^2 = 111$

Depth (d)	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	**111**	**1 seconds**	**11 kilobytes**
4	11111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11111 terabytes

Uninformed/Blind Search - Uniform Cost Search

- Uniform cost search **modifies** the breadth-first strategy by always expanding the **lowest-cost node** on the **fringe** (as measured by the **path cost** $g(n)$), rather than the **lowest-depth node**.
- The problem is to get from S to G and the cost of each operator is marked.
- The strategy first expands the initial state, yielding paths to A, B and C. Because the path to A is cheapest, it is expanded next, generating the **path** SAG, which is **in fact a solution**, though **not the optimal one**.



Uninformed/Blind Search - Uniform Cost Search

- The next step is to expand *SB*, generating *SBG*, which is now the **cheapest path** remaining in the queue, so it is **goal-checked** and returned as the **solution**.
- If every operator has a **nonnegative cost**, then the **cost of a path** can **never decrease** as we go along the path.
- But if some operator had a **negative cost**, then nothing but an **exhaustive search** of all nodes would find the **optimal solution**.

Properties

Same as BFS.

Uninformed/Blind Search - Depth-First Search

Algorithm

1. Place the starting node s on the Stack.
2. If the queue is empty, return failure and stop.
3. If the first element on the Stack is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element, and place the children at the front of the Stack **any order**.
5. Return to step 2.

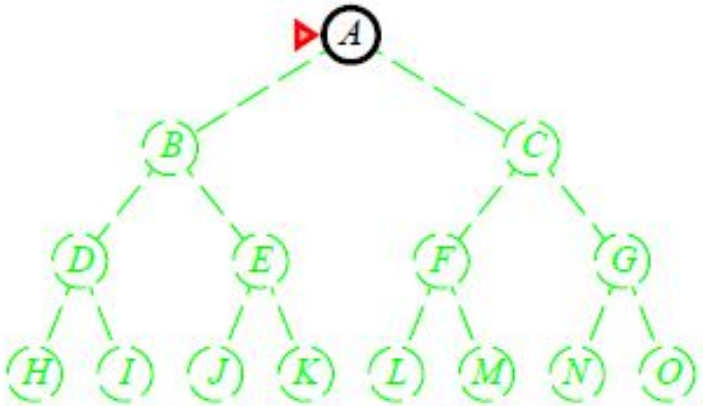
- Depth-first may actually be faster than breadth-first.
- **Implementation:** Stack

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

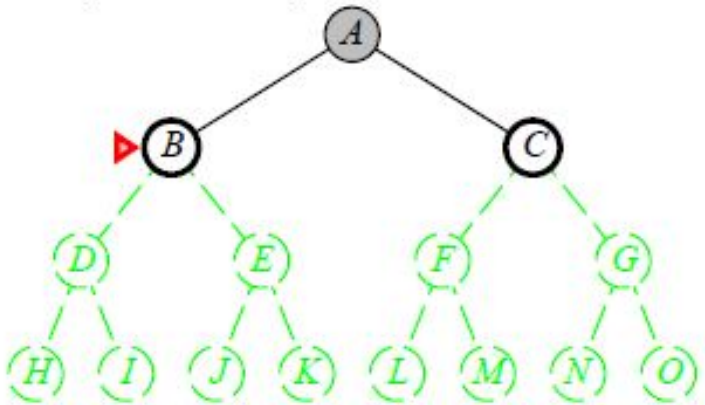


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

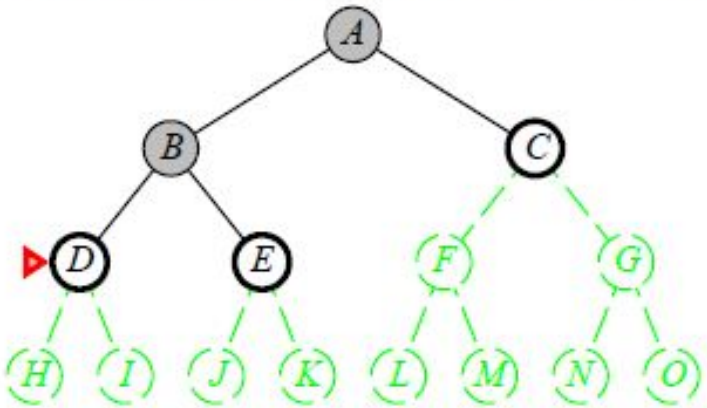


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

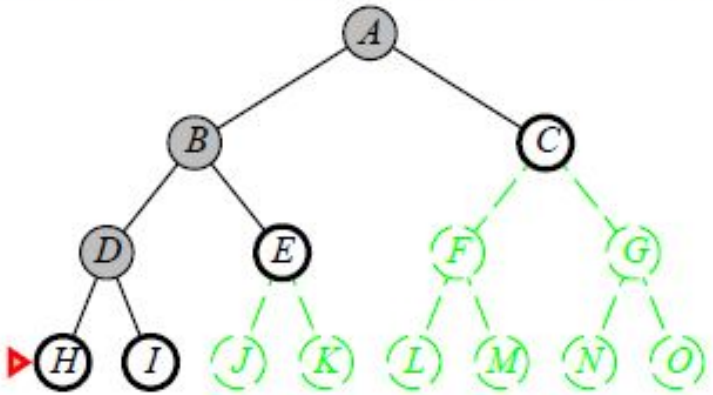


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

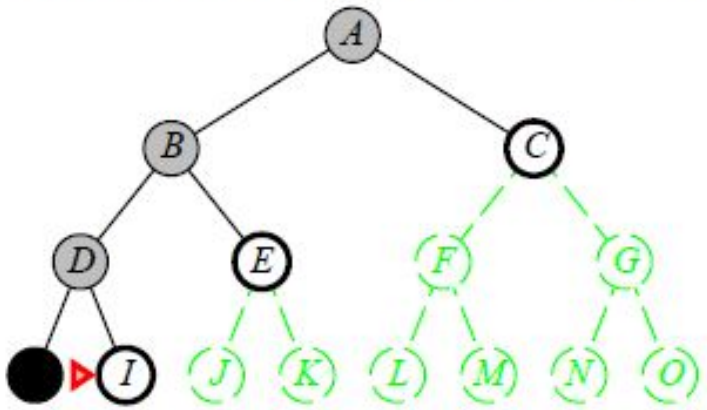


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

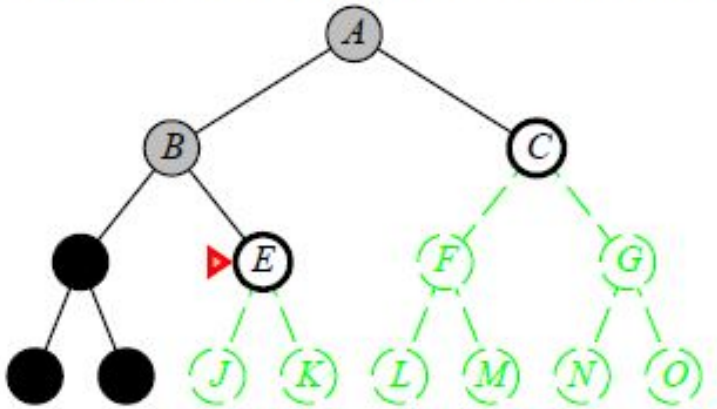


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

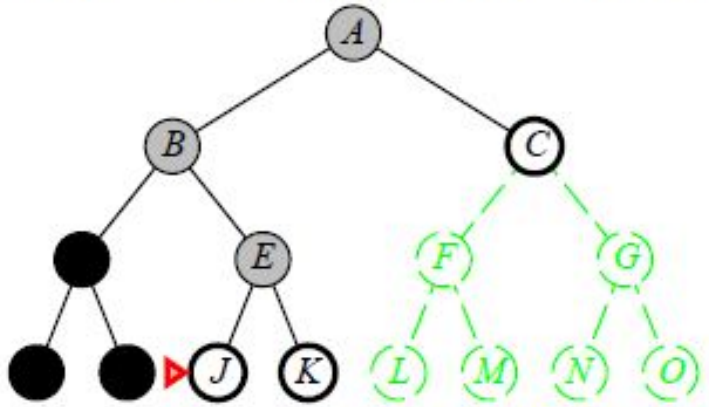


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

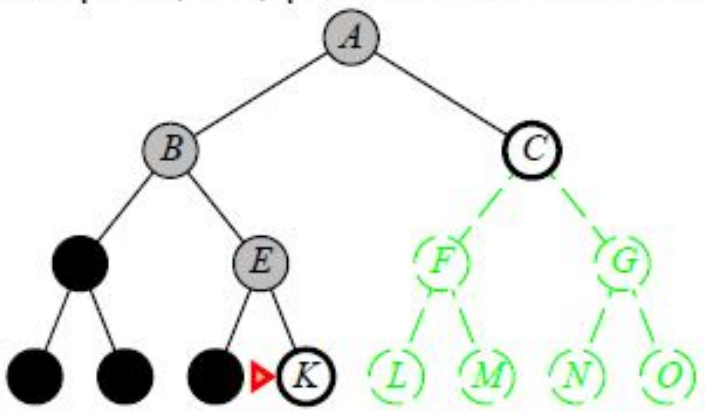


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

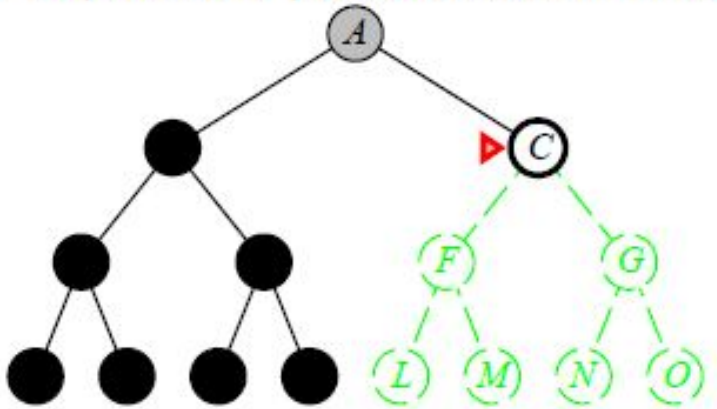


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

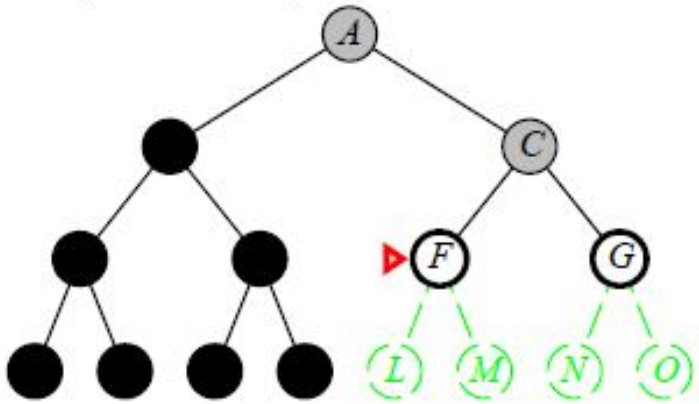


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

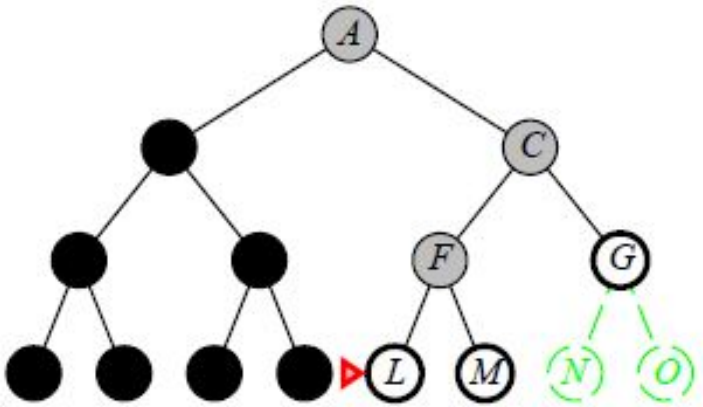


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

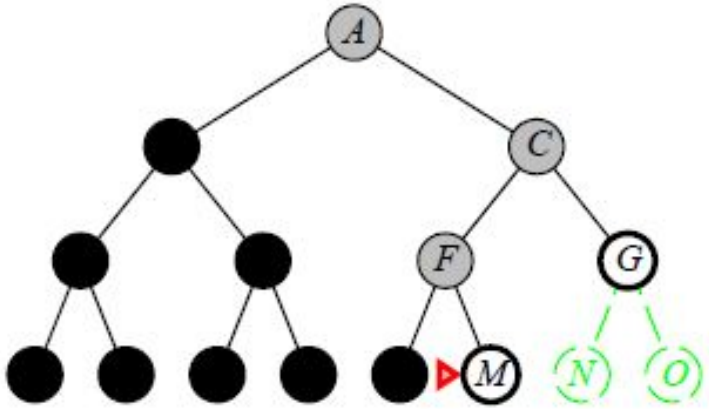


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

- It can get stuck going down the wrong path.
- Depth-first search should be avoided for search trees with large or infinite maximum depths.

Uninformed/Blind Search - Depth-Limited Search

- Depth-limited search **avoids** the **pitfalls** of **depth-first search** by imposing a **cutoff** on the **maximum depth** of a path.
- **Example:** On the map of Romania, there are **20 cities**, so we know that if there is a **solution**, then it must be of **length 19** at the **longest**.
- If you are in **city A** and have **travelled** a path of **less than 19 steps**, then generate a new state in **city B** with a path length that is **one greater**.
- We are guaranteed to find the solution if it exists.

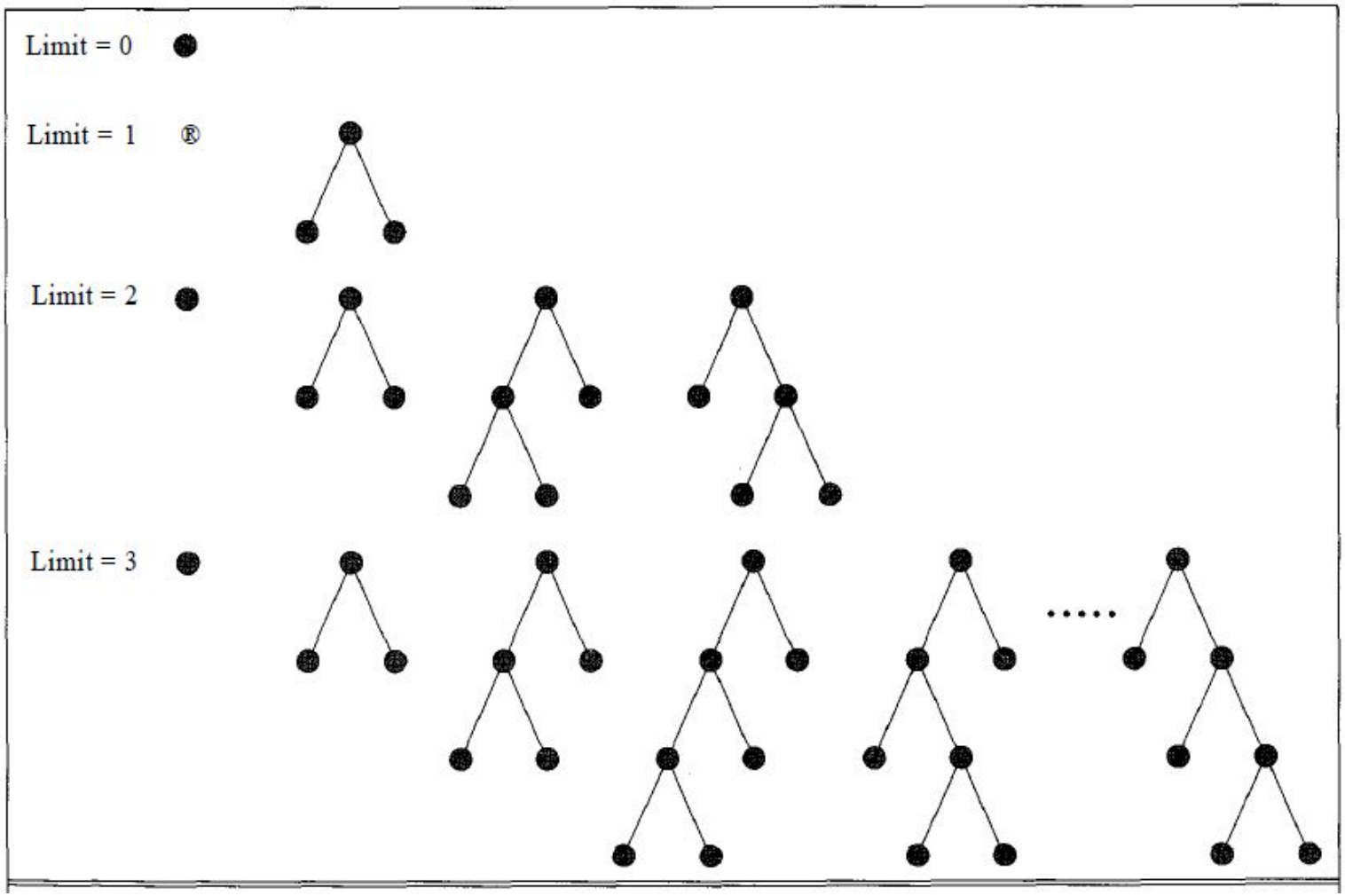
Properties

1. Complete: Yes, if $l > d$
   - If we choose a depth limit that is too small, then depth-limited search is not even complete.
2. Optimal: No
3. Time: $O(b^l)$ // l is the depth limit.
4. Space: $O(bl)$

Uninformed/Blind Search - Iterative Deepening Search

- The hard part about **depth-limited search** is picking a **good limit**.
- We would discover that **any city** can be reached from **any other city** in at most **9 steps** instead of **19**.
- Diameter of the state space $= 9$
- **Iterative deepening search** is a strategy that sidesteps the issue of choosing the **best depth limit** by trying all possible **depth limits**: first **depth 0**, then **depth 1**, then **depth 2**, and so on.
- The order of **expansion of states** is similar to **breadth-first**, except that some states are **expanded multiple times**.
- **Iterative deepening search** may seem **wasteful**, because so many states are expanded multiple times.

Uninformed/Blind Search - Iterative Deepening Search



Uninformed/Blind Search - Iterative Deepening Search

- The number of expansions in a **depth-limited search** to **depth** d with **branching factor** b is

$$1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d$$

- $b = 10$ and $d = 5$

$$1 + 10 + 10^2 + 10^3 + 10^4 + 10^5 = 111111$$

- The total number of expansions in an **iterative deepening search** is

$$(d+1)1 + (d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- $b = 10$ and $d = 5$

$$(5+1)1 + (5)10 + (5-1)10^2 + (5-2)10^3 + (5-3)10^4 + (5-4)10^5$$

$$= 6 + 50 + 400 + 3000 + 20000 + 100000 = 123456 \text{ (11\% more nodes)}$$

Uninformed/Blind Search - Iterative Deepening Search

Properties

1. Complete: Yes
2. Optimal: Yes
3. Time: $O(b^d)$
4. Space: $O(bd)$

- Iterative deepening is the preferred search method when there is a large search space and the depth of the solution is not known.

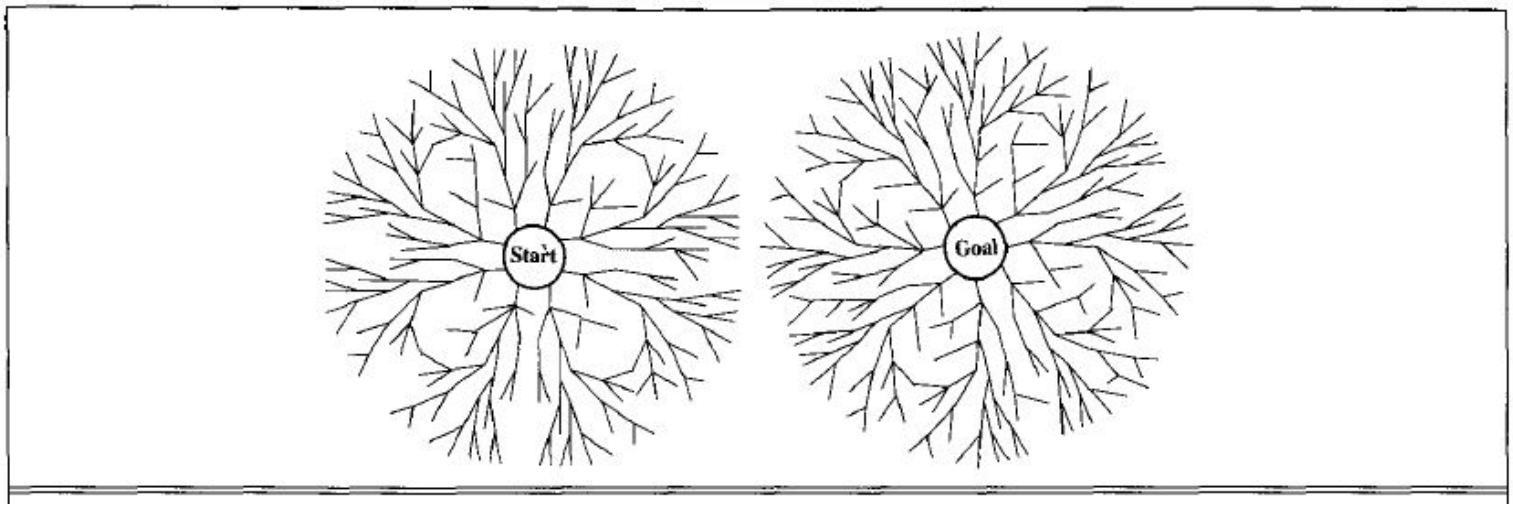
Uninformed/Blind Search - Bidirectional Search

- The idea behind bidirectional search is to simultaneously search both **forward** from the **initial** state j and **backward** from the **goal** and stop when the **two searches** meet in the **middle**.
- The solution will be found in $O(2b^{d/2}) = O(b^{d/2})$ steps because the **forward** and **backward** searches each have to go only **half way**.

Issues

1. What does it mean to search backwards from the goal?
2. When all operators are **reversible**, the predecessor and successor sets are **identical**. calculating **predecessors** can be **very difficult**.
3. What can be done if there are many possible goal states?

Uninformed/Blind Search - Bidirectional Search



Properties

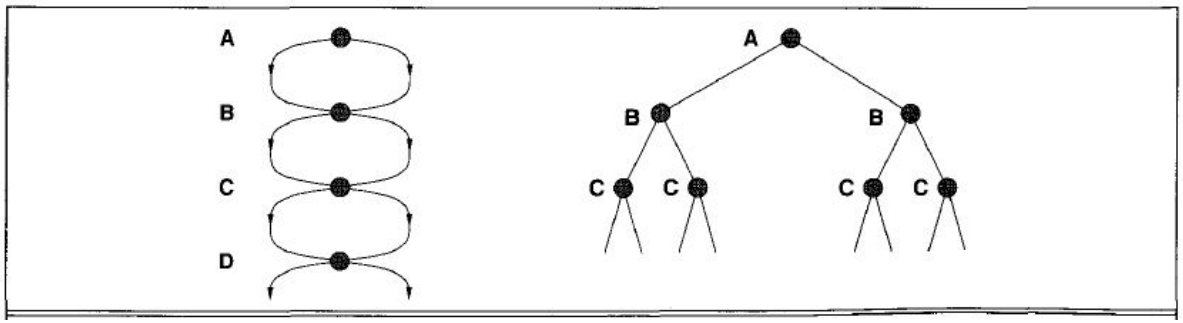
1. Complete: Yes
2. Optimal: Yes
3. Time: $O(b^{d/2})$
4. Space: $O(b^{d/2})$

Avoiding Repeated States

- For many problems, repeated states are unavoidable.
  - Missionaries and Cannibals problem
- The space contains only $m + 1$ states, where m is the maximum depth.

Three ways to deal with repeated states

- Do not return to the state you just came from.
- Do not create paths with cycles in them.
- Do not generate any state that was ever generated before.



Best-first search

Idea: use an evaluation function for each node
– estimate of "desirability"

⇒ Expand most desirable unexpanded node

Implementation:
fringe is a queue sorted in decreasing order of desirability

Special cases:
greedy search
A* search

Greedy search

Evaluation function $h(n)$ (heuristic)
$=$ estimate of cost from n to the closest goal

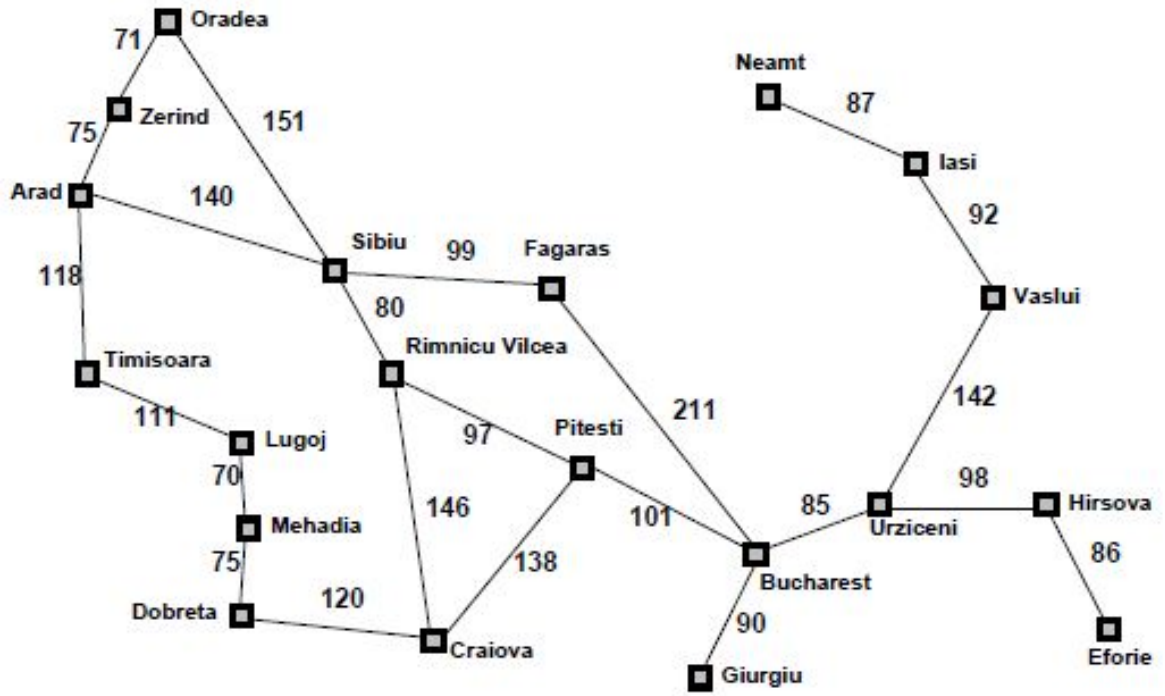
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that appears to be closest to goal

We look at two basic approaches.

- To expand the node closest to the goal
- To expand the node on the least-cost solution path

Romania with step costs in km

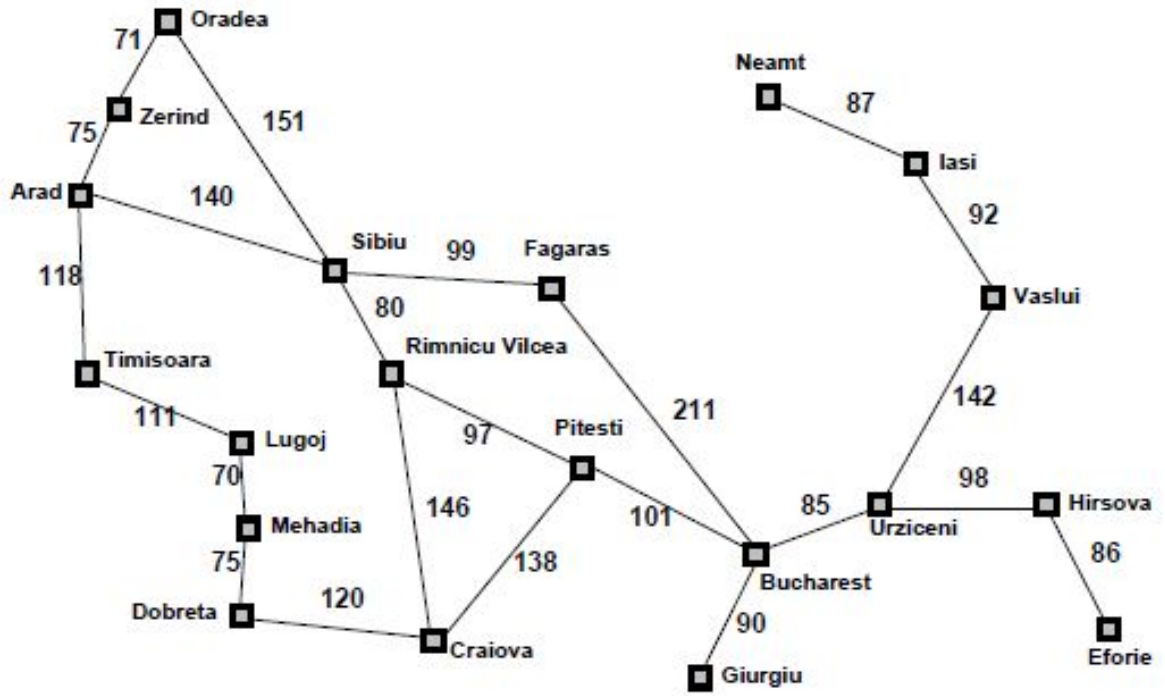


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example

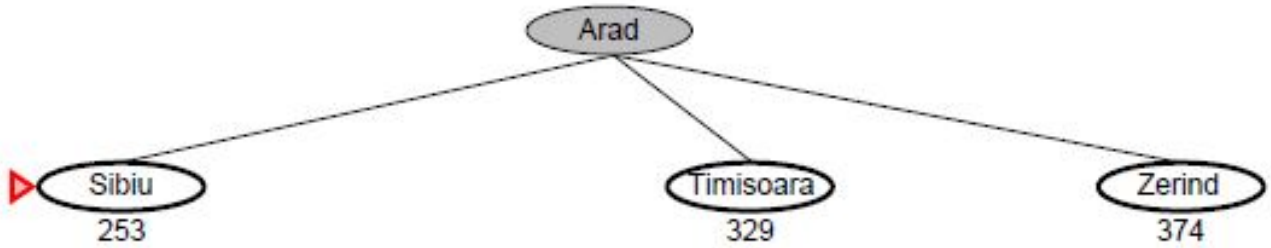
- $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
- $h_{SLD}(n)$ = straight-line distance between n and the goal location.

Romania with step costs in km

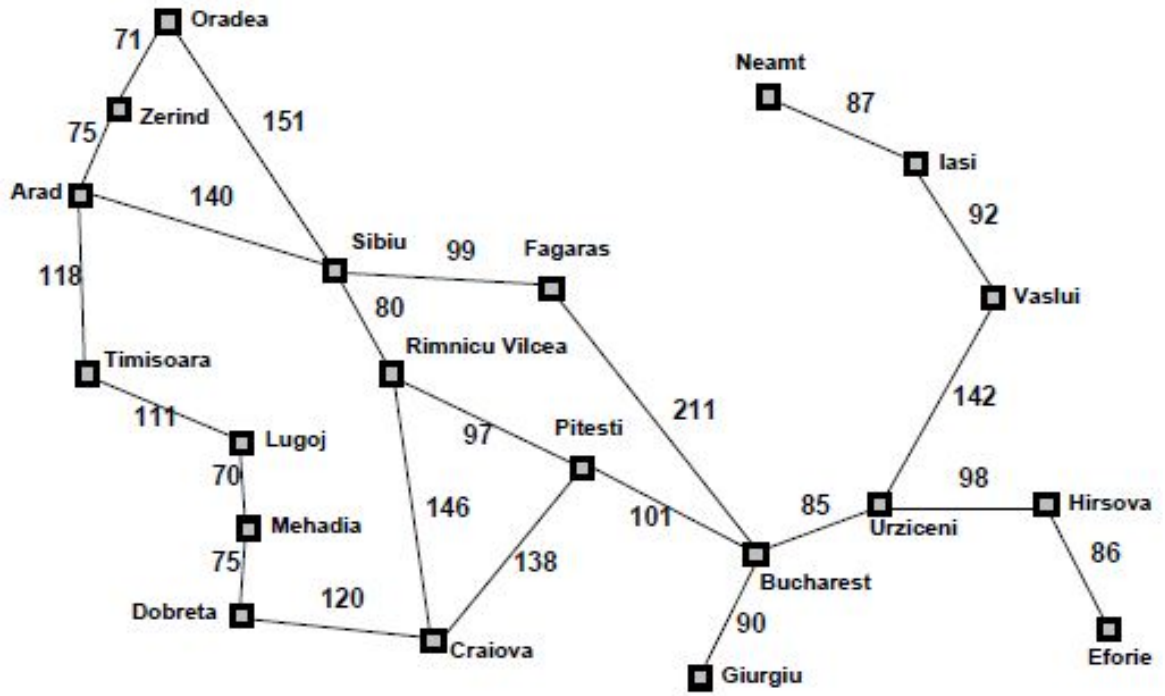


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

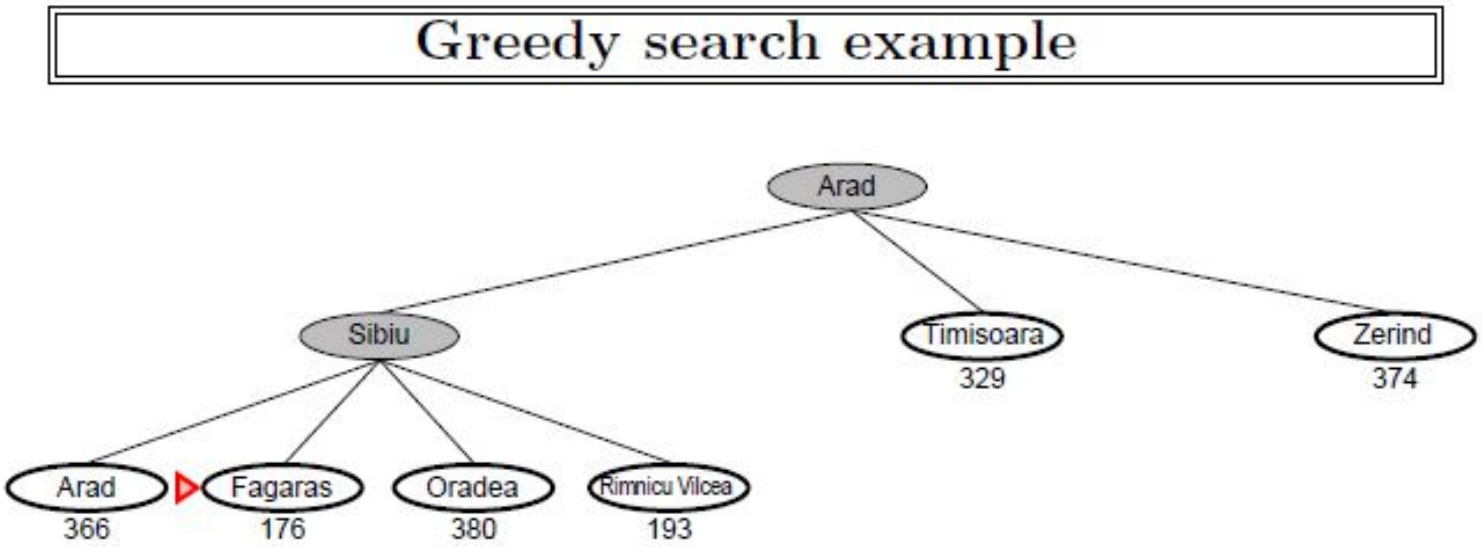
Greedy search example



Romania with step costs in km

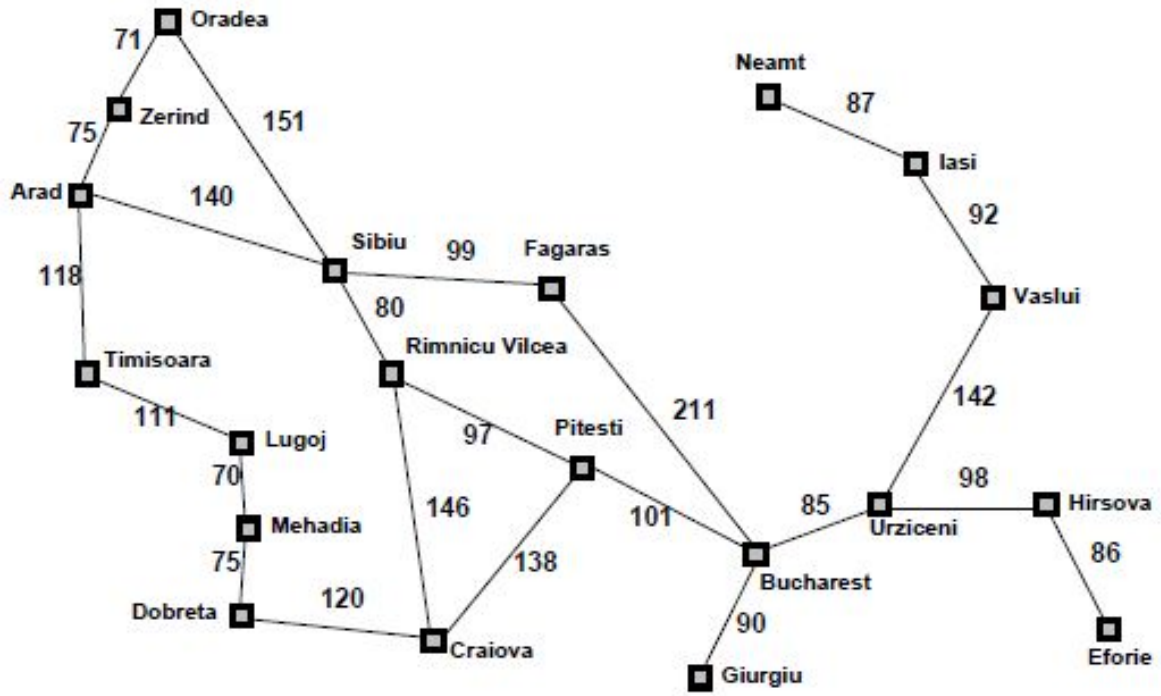


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

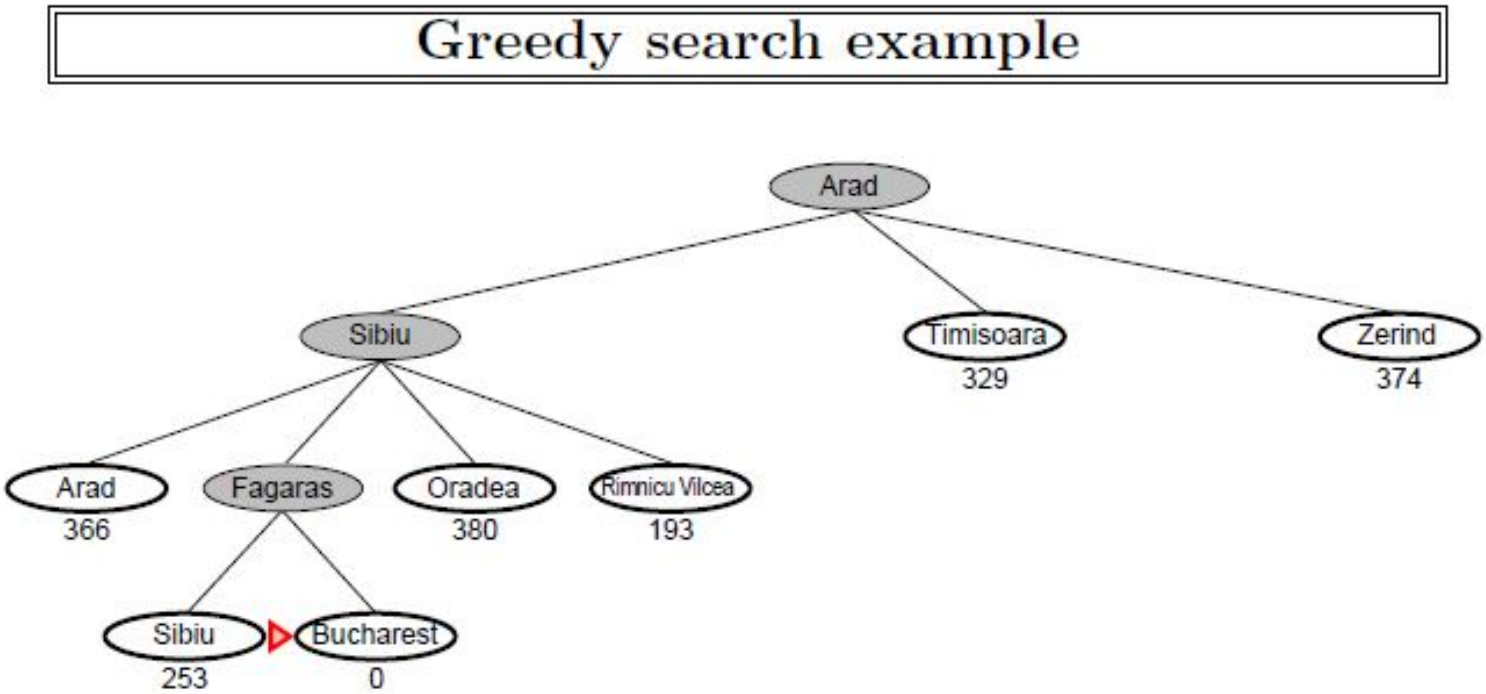

Greedy search example
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193

- **Fagaras:** 178

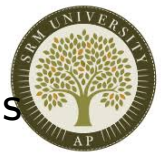
Romania with step costs in km



Straight−line distance to 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



- It suffers from the same defects as **depth-first search**-it is **not optimal**, and it is **incomplete** because it can start down an **infinite path** and never return to try other possibilities.
- The worst-case time complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space.
- Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity.



A* search

Idea: avoid expanding paths that are already expensive

Evaluation function $f(n) = g(n) + h(n)$

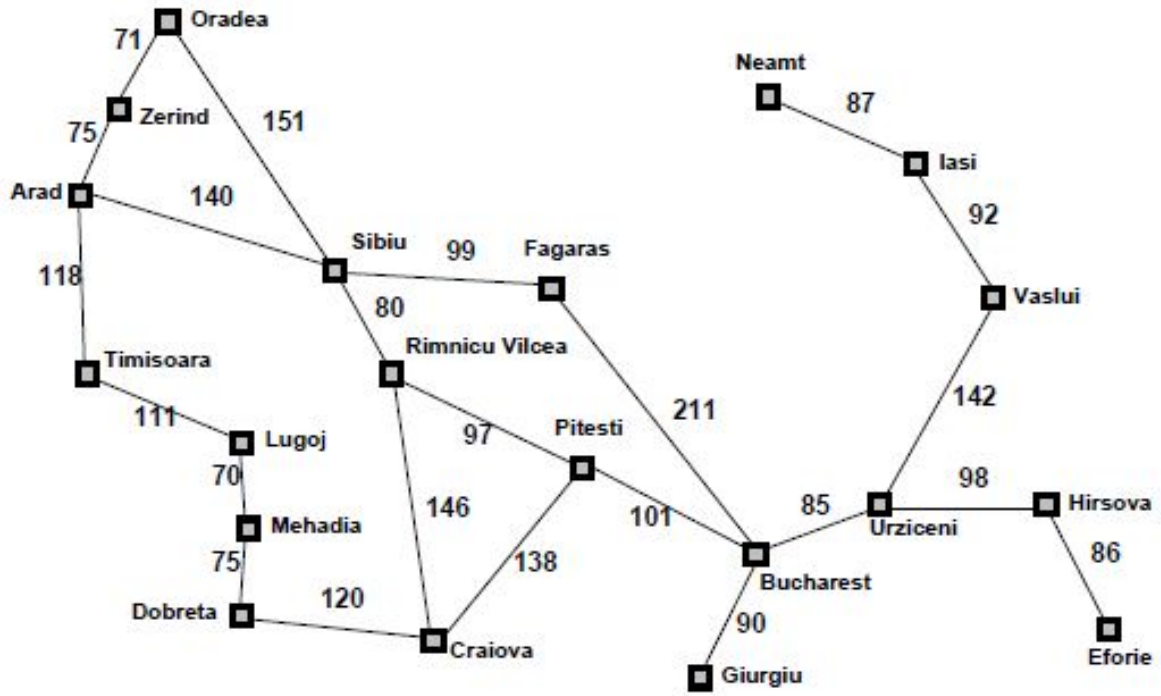
$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

A* search uses an admissible heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n.
(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

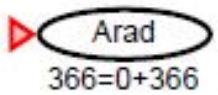
Theorem: A* search is optimal

Romania with step costs in km

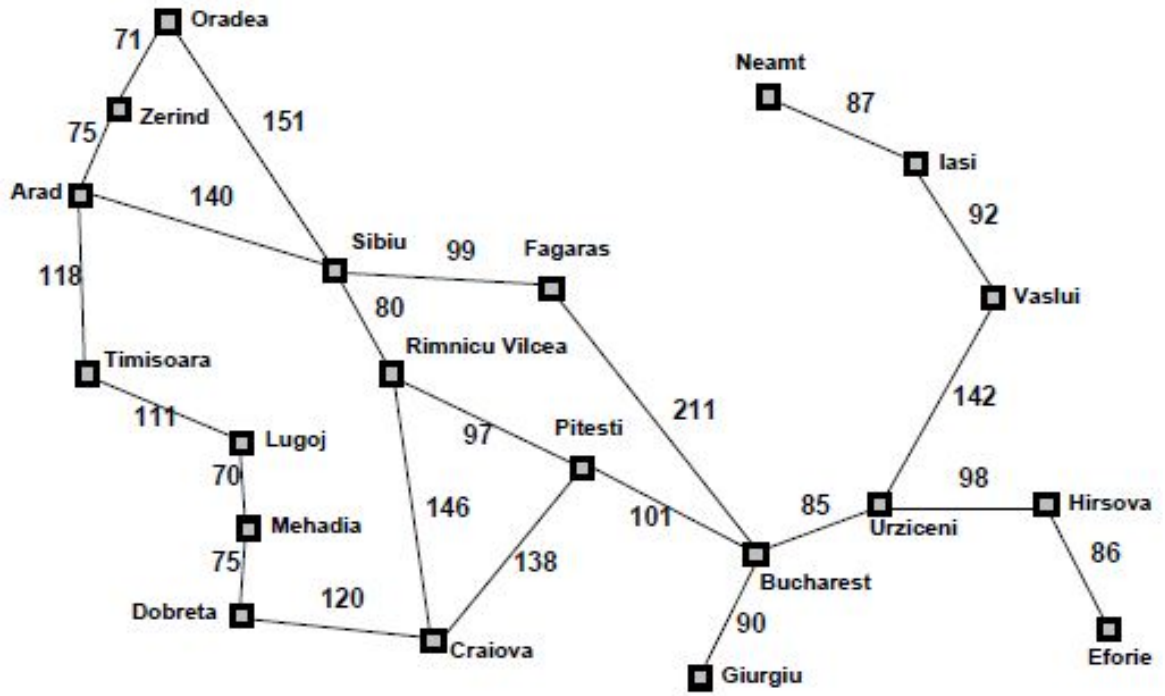


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

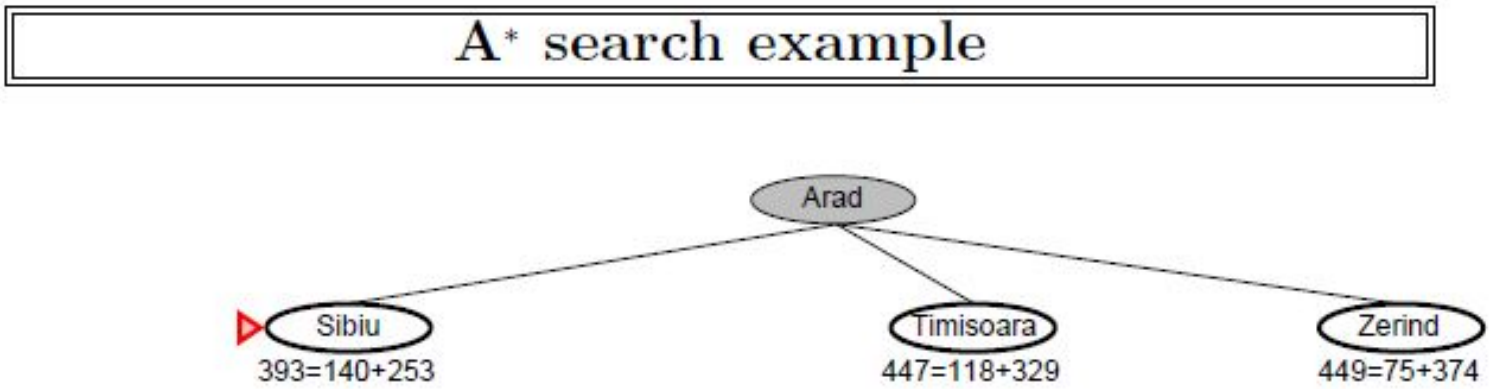
A* search example



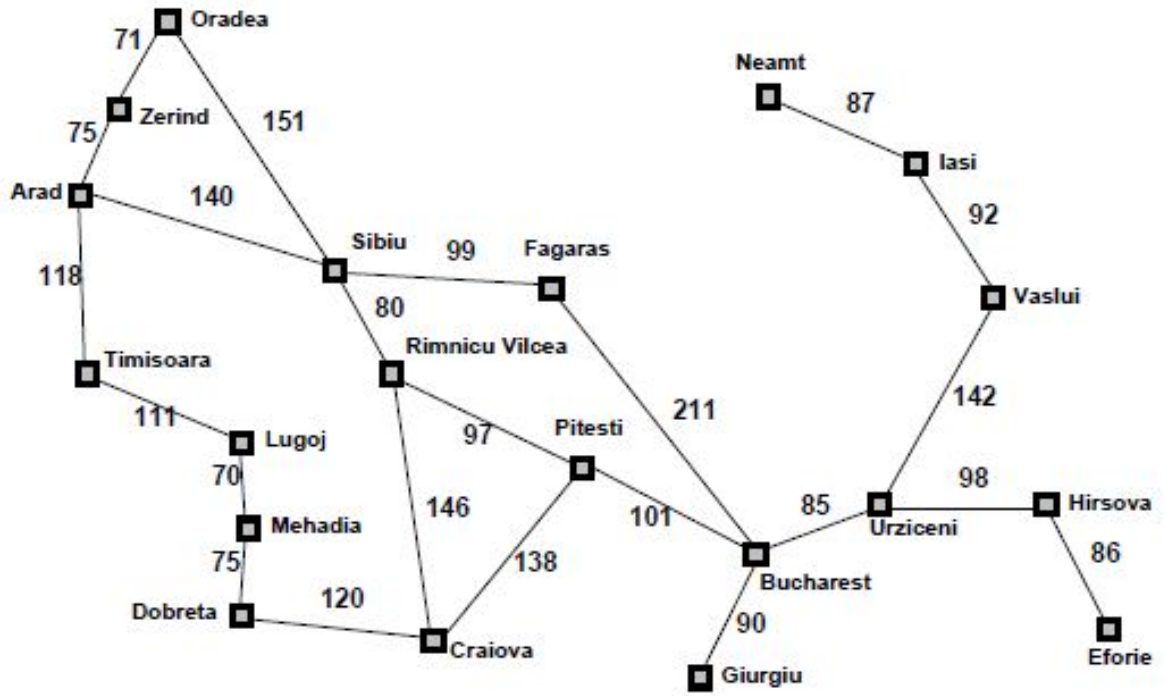
Romania with step costs in km



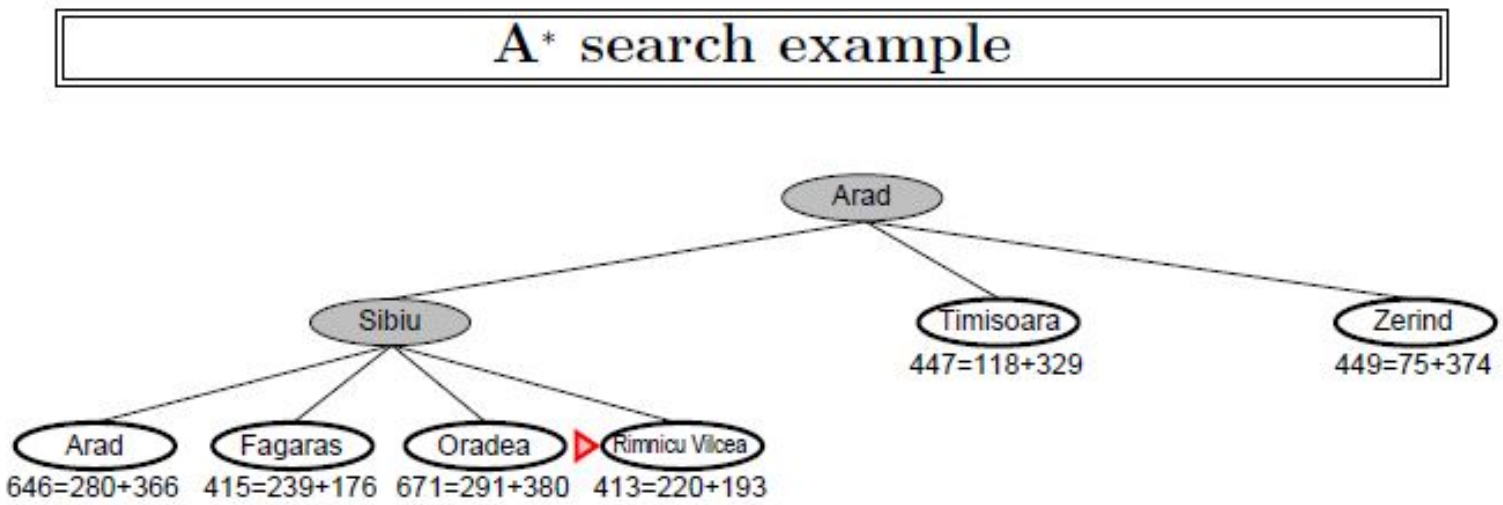
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374


A* search example
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

Romania with step costs in km

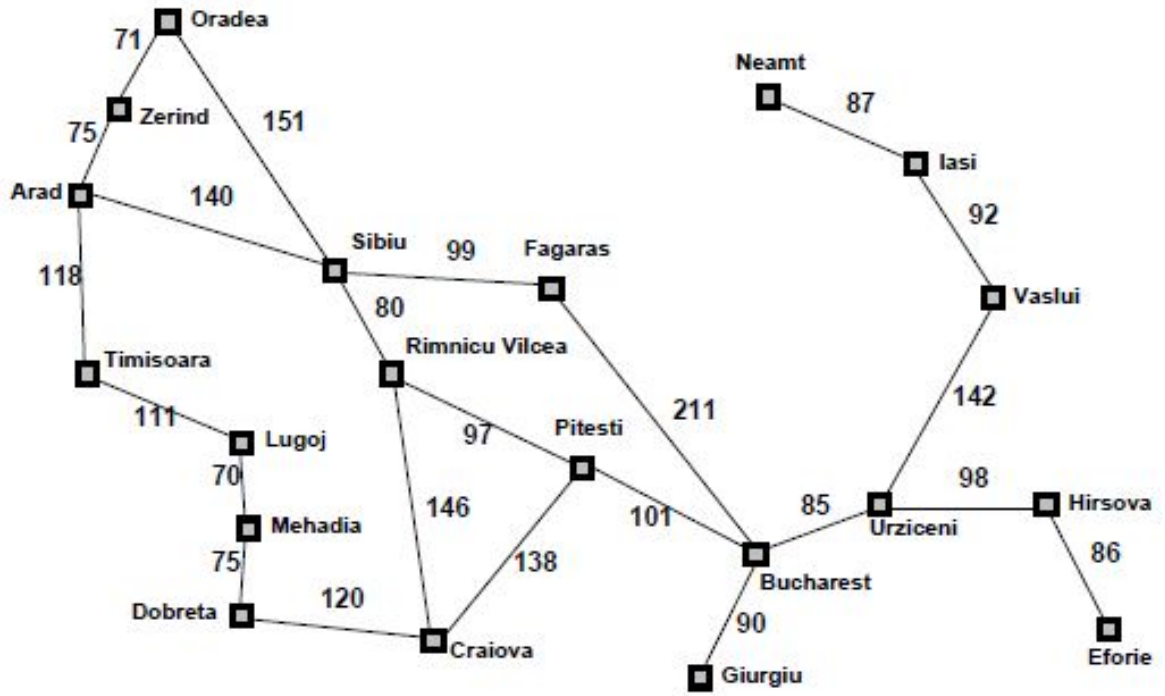


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

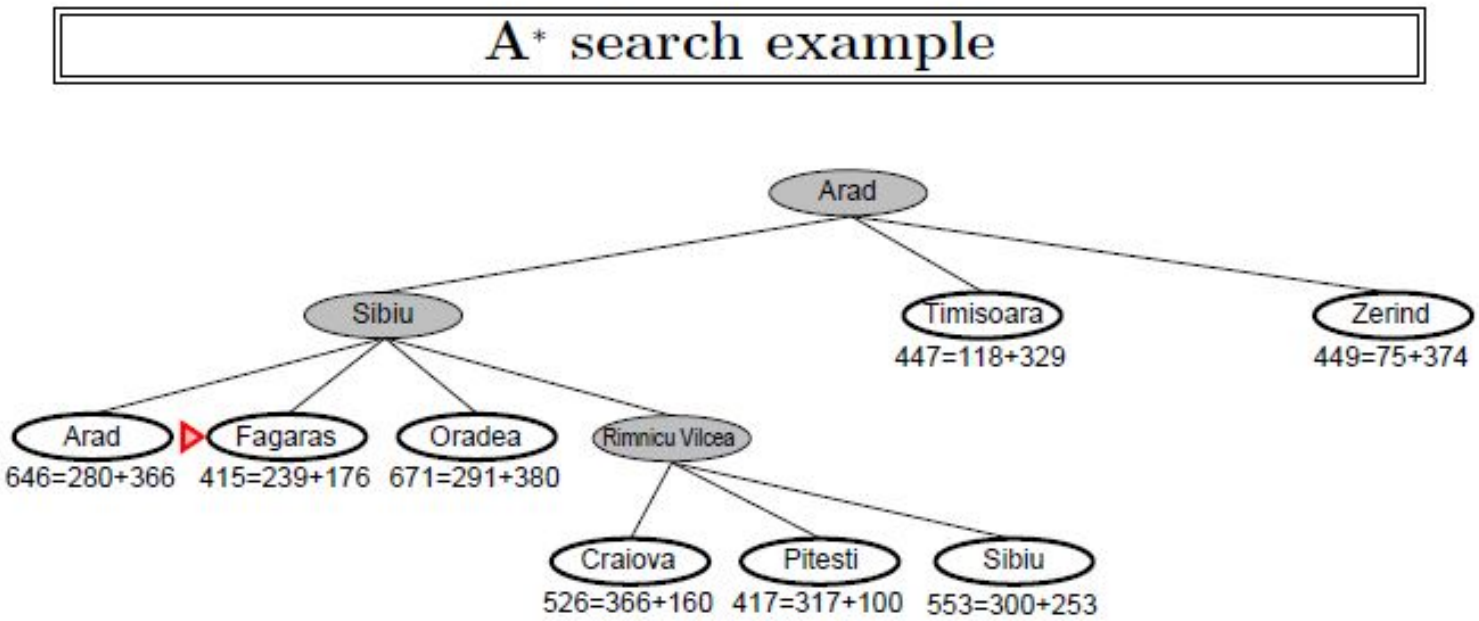


- Fagaras: $417 = 239 + 178$

Romania with step costs in km

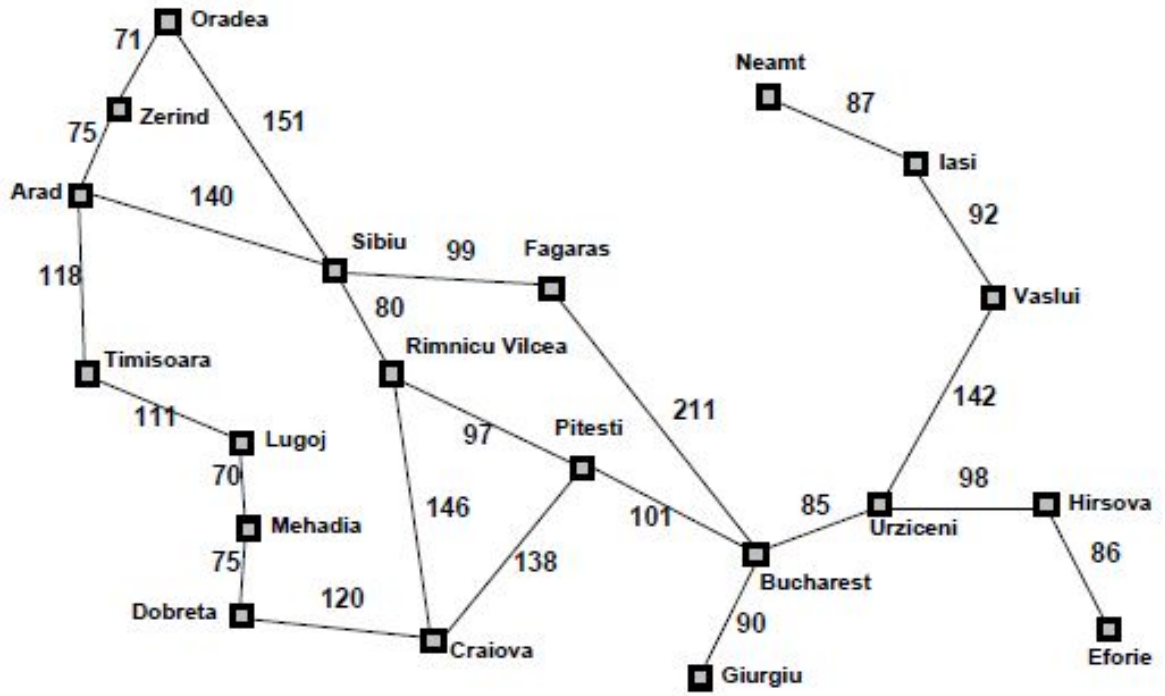


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

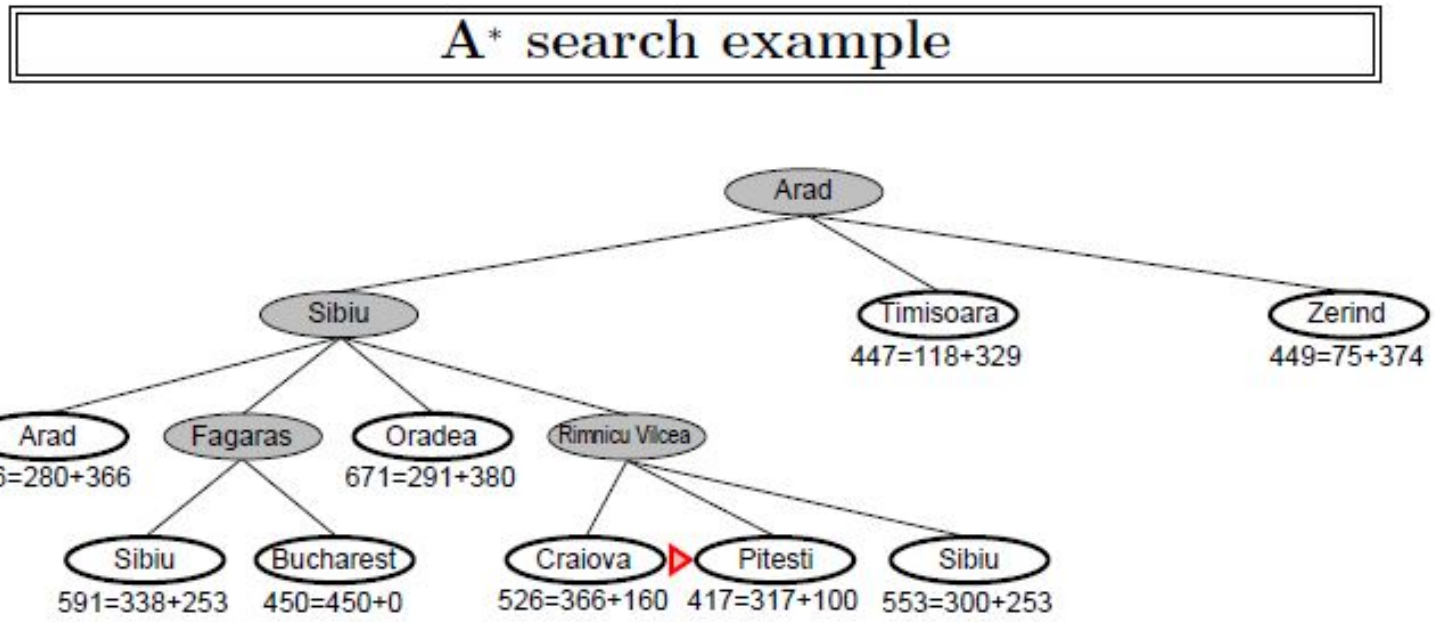


- Pitesti: $415 = 317 + 98$

Romania with step costs in km

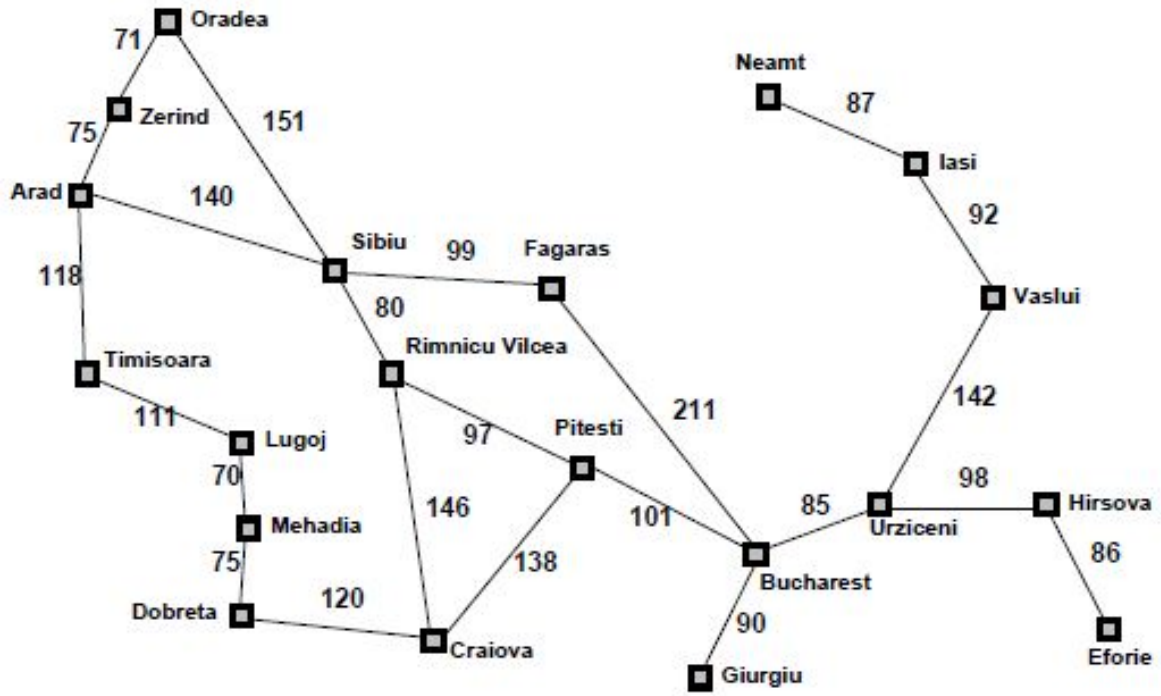


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

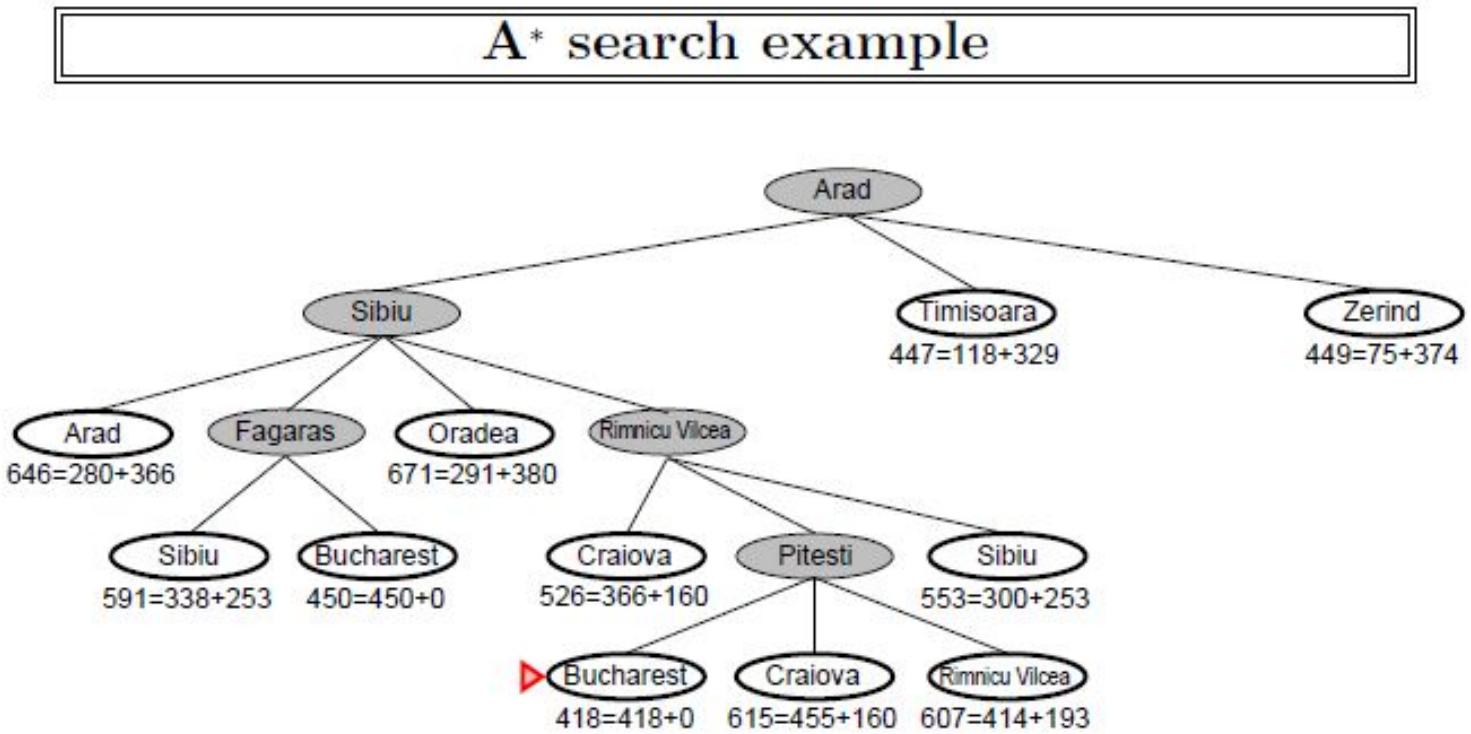


- Pitesti: $415 = 317 + 98$

Romania with step costs in km



Straight-line distance to 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



- Pitesti: $415 = 317 + 98$

A* Search

A* Search

1. Place the starting node s on open.
2. If open is empty, stop and return failure.
3. Remove from open the node n that has the smallest value of $f^*(n)$. If the node is a goal node, return success and stop. Otherwise,
4. Expand n, generating all of its successors n' and place n on closed. For every successor n', if n' is not already on open or closed attach a back-pointer to n, compute $f^*(n')$ and place it on open.
5. Each n' that is already on open or closed should be attached to back-pointers which reflect the lowest $g^*(n')$ path. If n' was on closed and its pointer was changed, remove it and place it on open.
6. Return to step 2.

A* Search

Admissibility Condition

Algorithm A is admissible if it is guaranteed to return an optimal solution when one exists.

Completeness Condition

Algorithm A is complete if it always terminates with a solution when one exists.

Dominance Condition

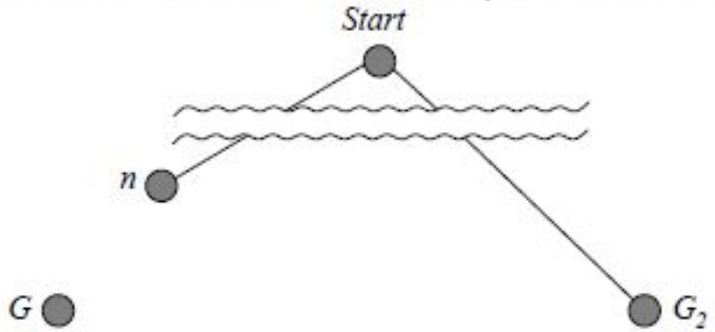
Let A_1 and A_2 be admissible algorithms with heuristics estimation functions h_1^* and h_2^*, respectively. A_1 is said to be more informed than A_2 whenever $h_1^*(n) > h_2^*(n)$ for all n. A_1 is also said to dominate A_2.

Optimality Condition

Algorithm A is optimal over a class of algorithms if A dominates all members of the class.

Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



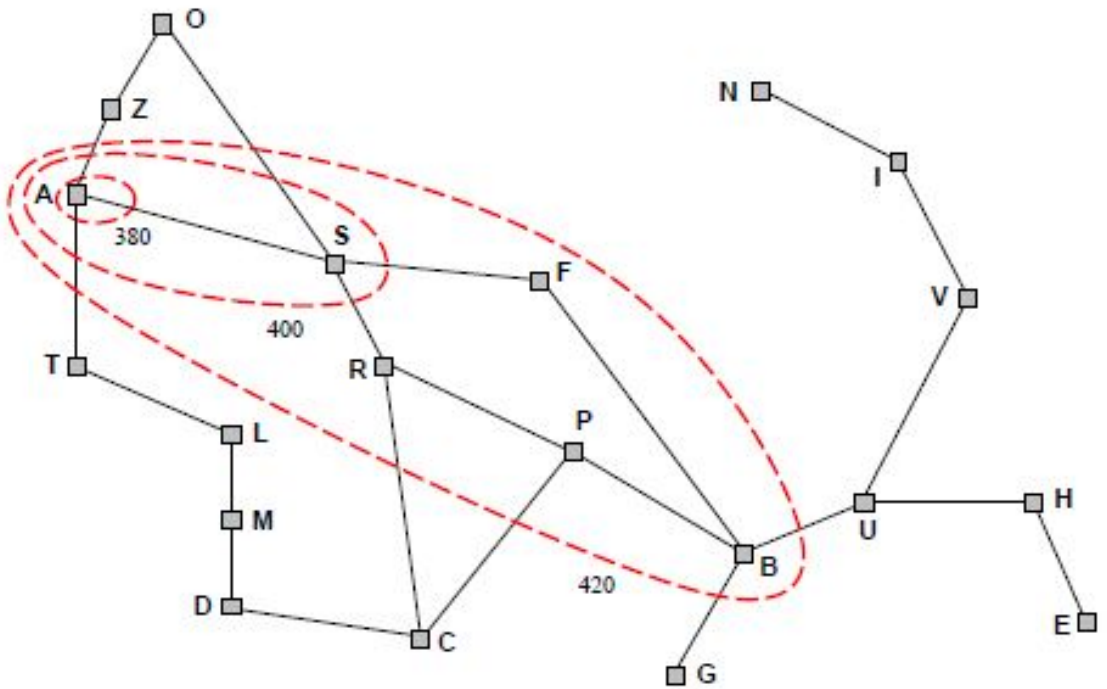
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

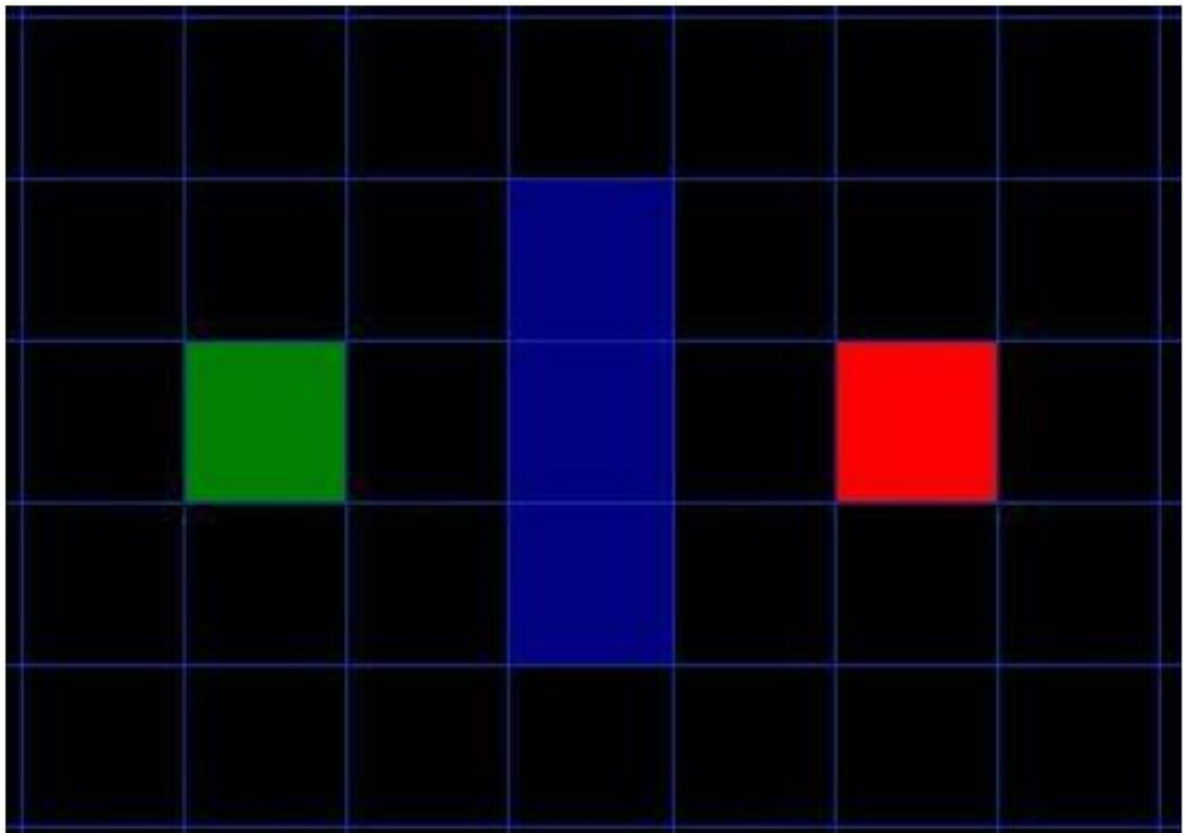
Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$

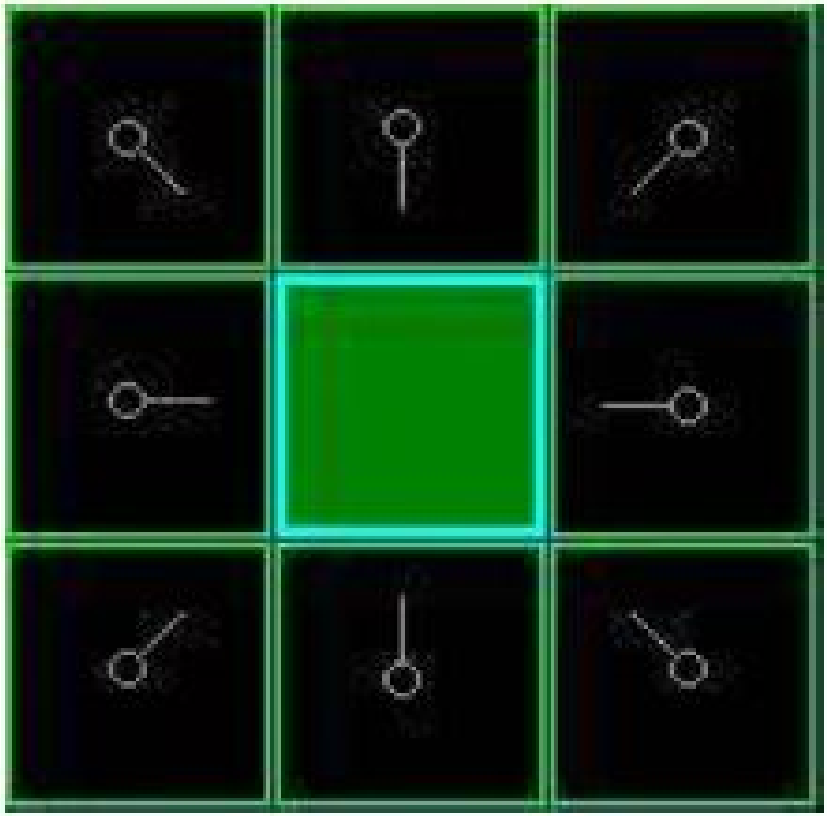
Informed Search - A∗ Search

- Let's assume that we have someone who wants to get from **point A** to **point B**. Let's assume that a wall separates the two points.



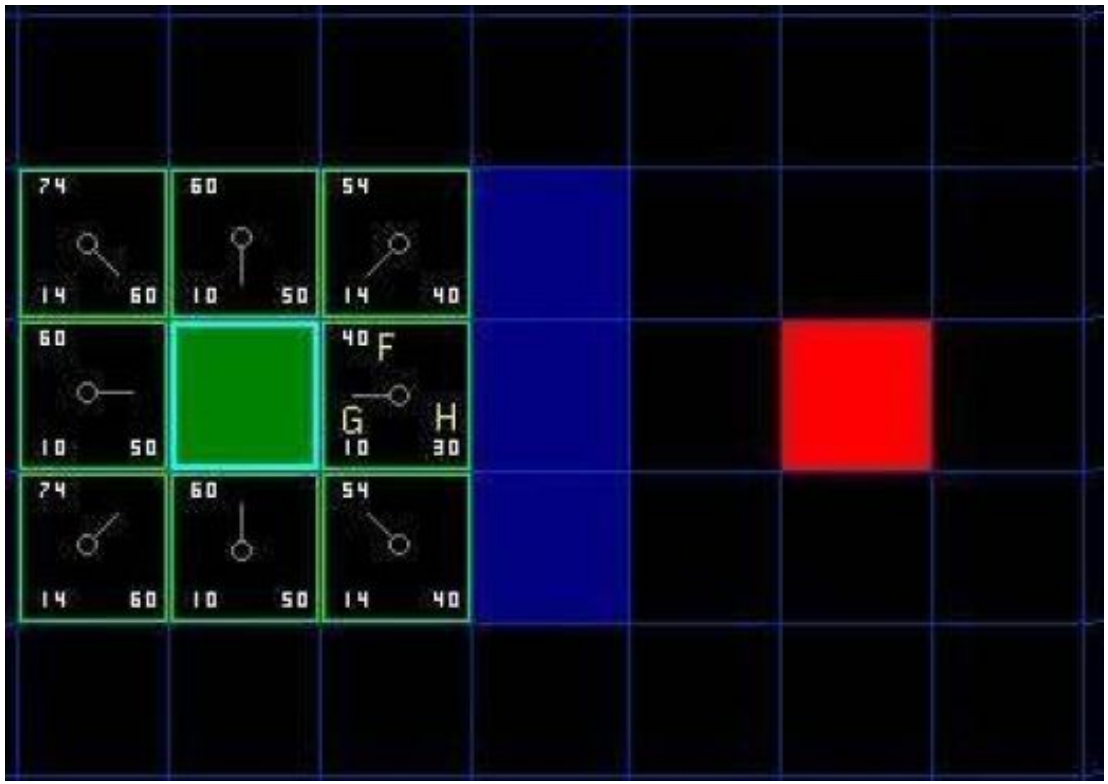
Informed Search - A∗ Search

- Status: Walkable or Unwalkable
- Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain.

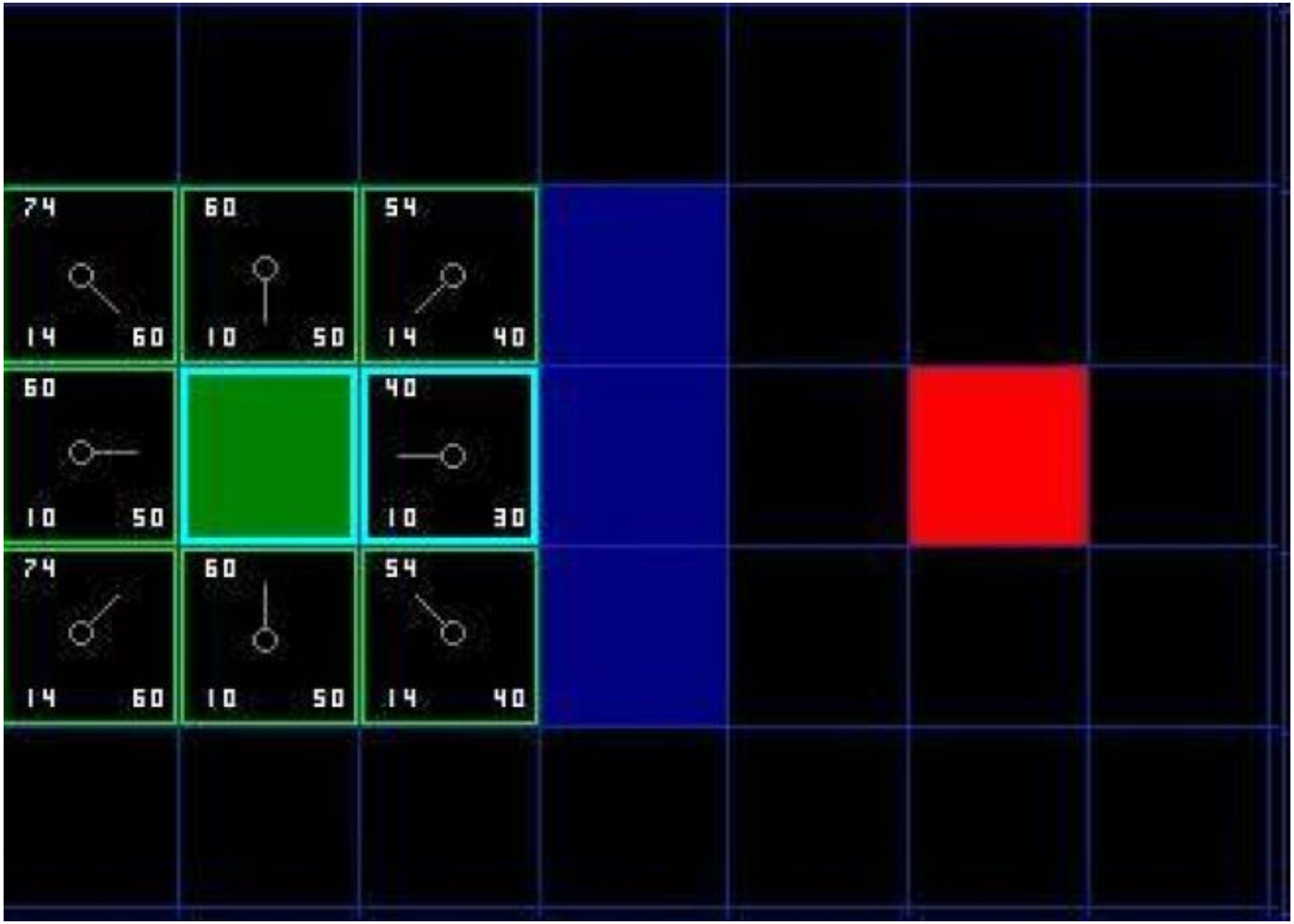


Informed Search - A∗ Search

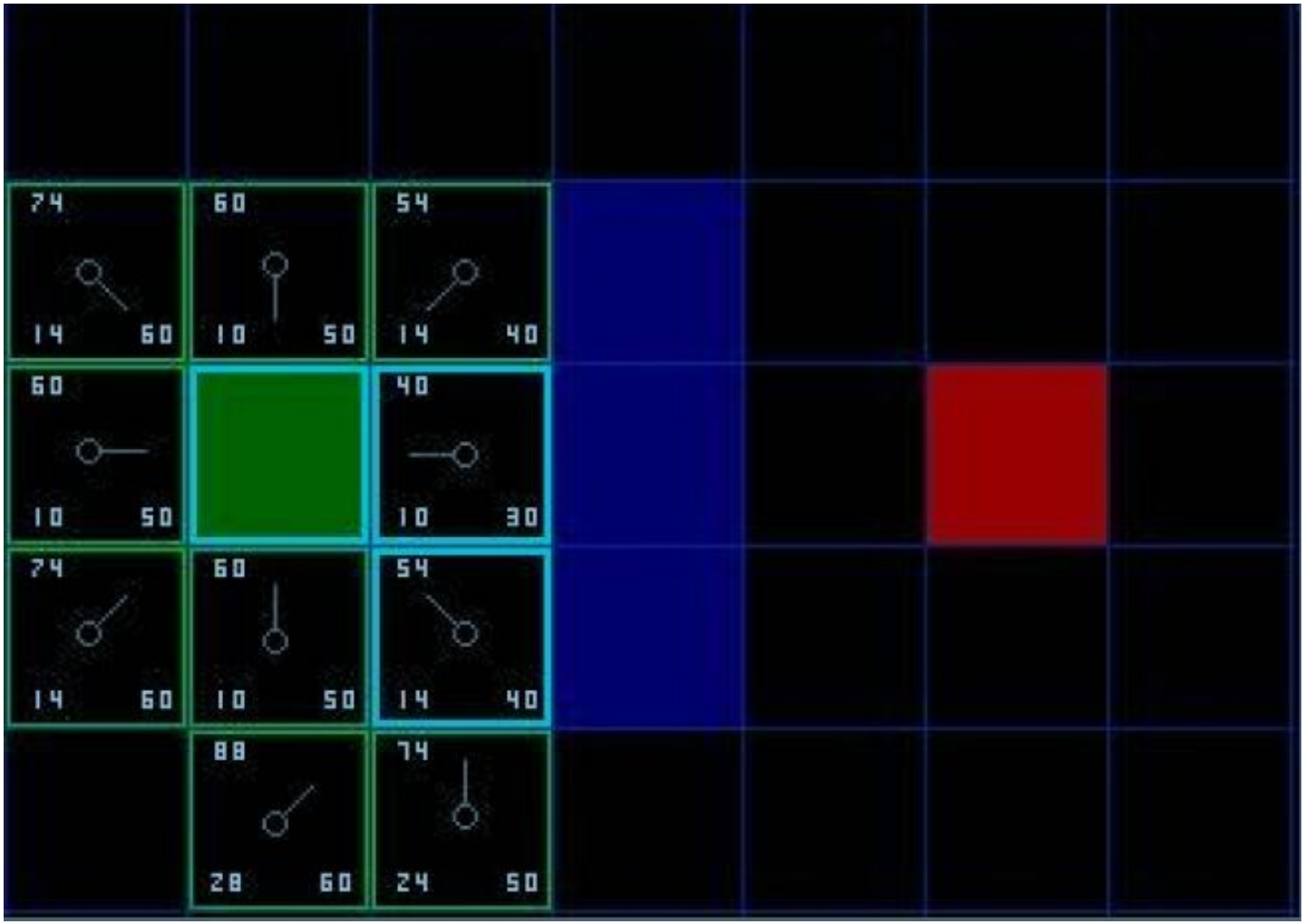
- We will assign a cost of **10** to each **horizontal** or **vertical** square moved and a cost of **14** for a **diagonal** move
- The method we use here is called the **Manhattan method**, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, **ignoring diagonal movement** and **ignoring any obstacles** that may be in the way.



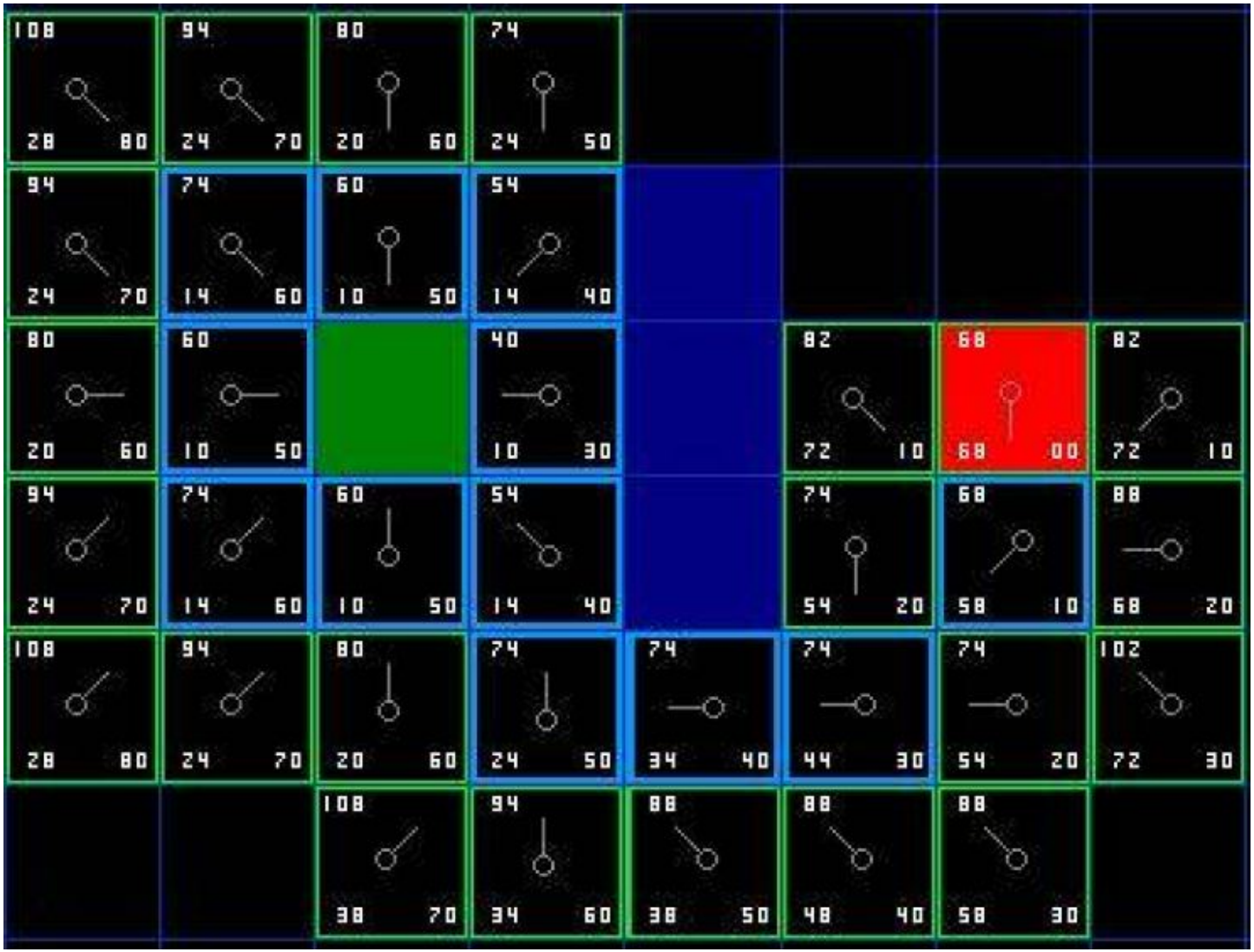
Informed Search - A∗ Search



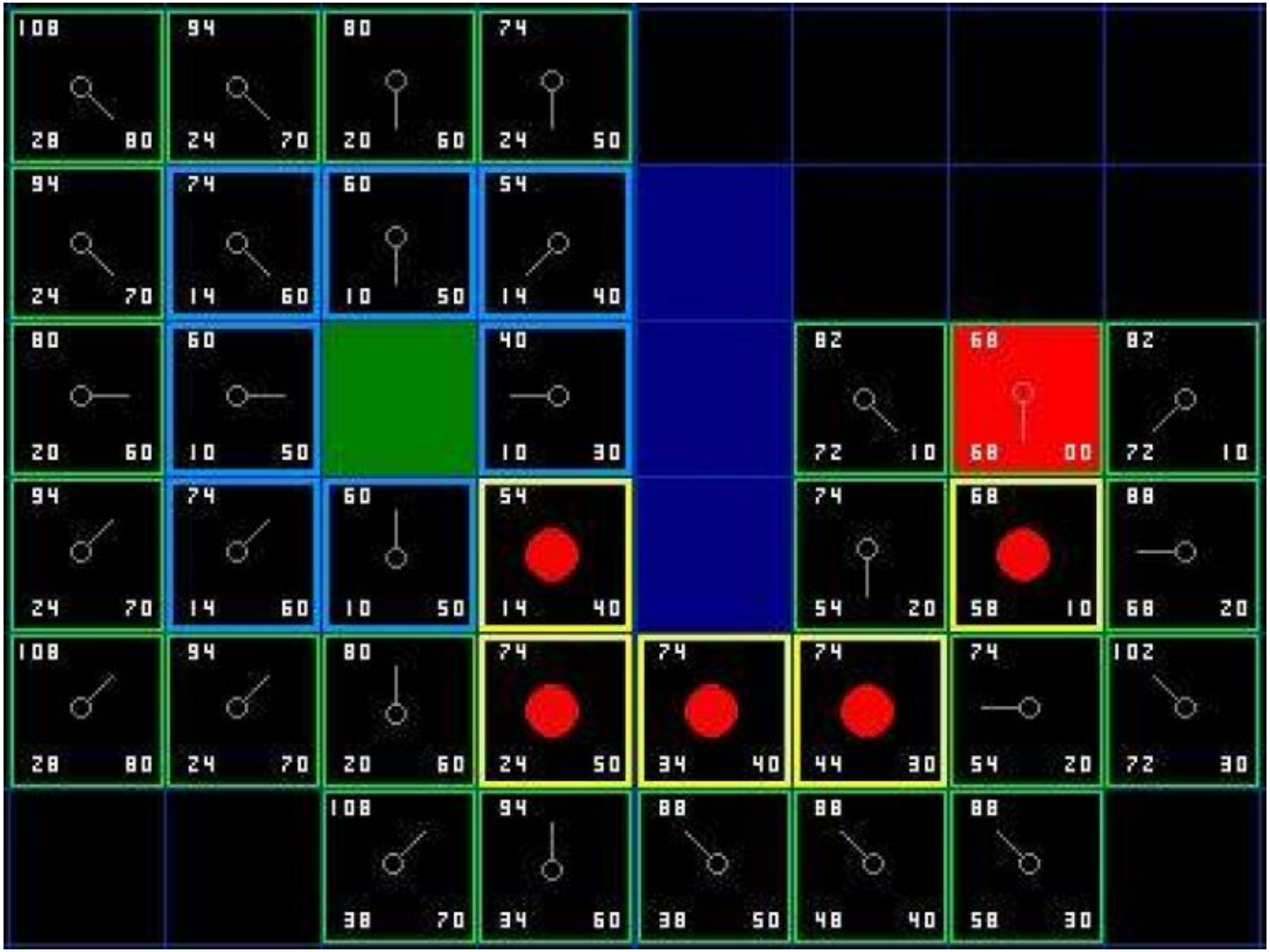
Informed Search - A∗ Search



Informed Search - A∗ Search



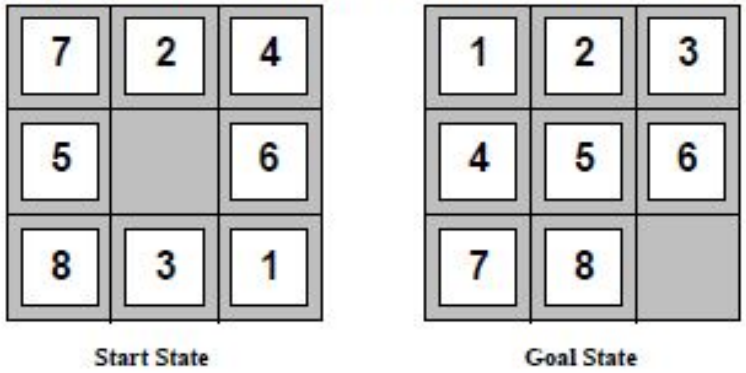
Informed Search - A∗ Search



Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



$h_1(S) =$?? 6
$h_2(S) =$?? 4+0+3+3+1+0+2+1 = 14

Heuristic Function

- A typical solution is about **20 steps**, although this of course varies depending on the **initial state**.
- The **branching factor** is about **3** (when the **empty tile** is in the **middle**, there are **four possible moves**; when it is in a **corner** there are **two**; and when it is along an **edge** there are **three**).
- An exhaustive search to depth **20** would look at about $3^{20} = 3.5 \times 10^9$ states.
- There are only $9! = 362{,}880$ different arrangements of 9 squares.
- Manhattan/City block distance: $|x_1 - x_2| + |y_1 - y_2|$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS $= 3{,}473{,}941$ nodes
$A^*(h_1) = 539$ nodes
$A^*(h_2) = 113$ nodes
$d = 24$ IDS $\approx 54{,}000{,}000{,}000$ nodes
$A^*(h_1) = 39{,}135$ nodes
$A^*(h_2) = 1{,}641$ nodes

Given any admissible heuristics h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

Relaxed problems

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

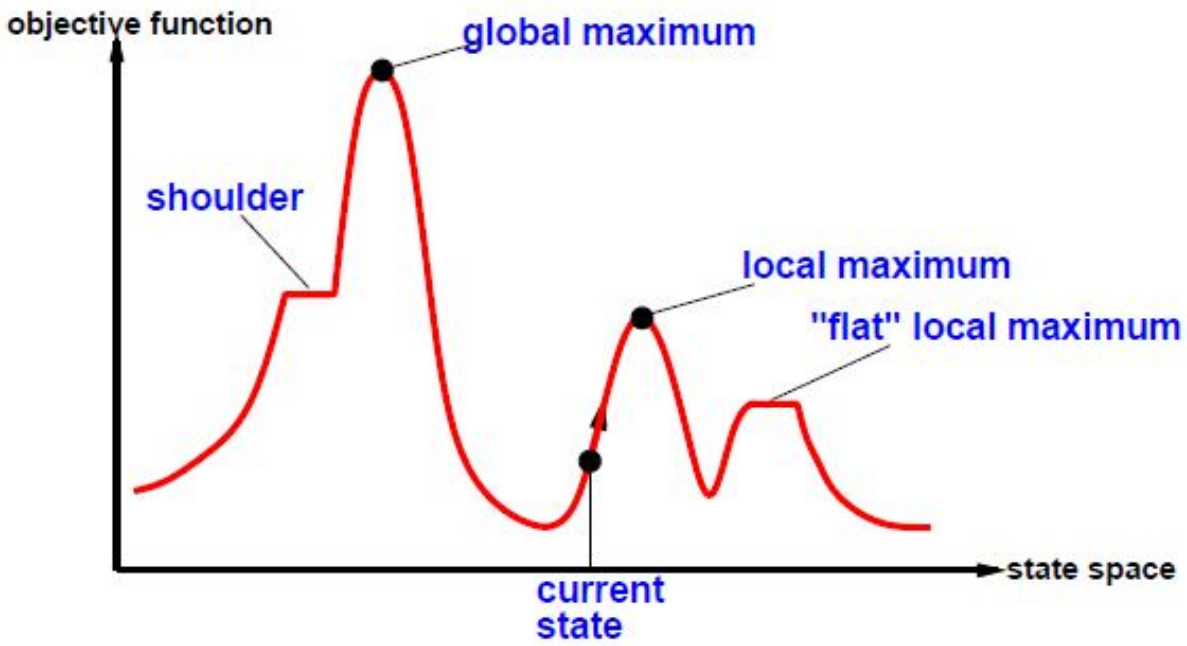
Local Search Algorithms and Optimization Problems

- **Local search algorithms** operate using a single **current state** (rather than **multiple paths**) and generally **move only to neighbors** of that state.
- Typically, the **paths** followed by the search are **not retained**.
- Although local search algorithms are **not systematic**, they have two key advantages:
  1. They use **very little memory**-usually a constant amount
  2. They can often **find reasonable solutions** in large or infinite (continuous) state spaces for which **systematic algorithms are unsuitable**.

Local Search Algorithms and Optimization Problems

- Local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**.
- A landscape has both "**location**" (defined by the **state**) and "**elevation**" (defined by the value of the **heuristic cost function** or **objective function**).
  - If elevation corresponds to **cost**, then the aim is to find the **lowest valley-a global minimum**
  - If elevation corresponds to an **objective function**, then the aim is to find the **highest peak**-a **global maximum**.

Hill-Climbing Search

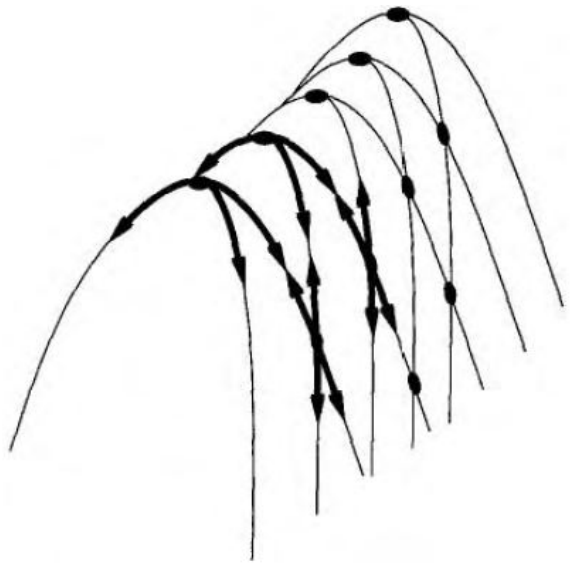


- It is simply a loop that **continually** moves in the direction of **increasing value**-that is, **uphill**.
- It terminates when it reaches a "**peak**" where **no neighbor** has a **higher value**.
- Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

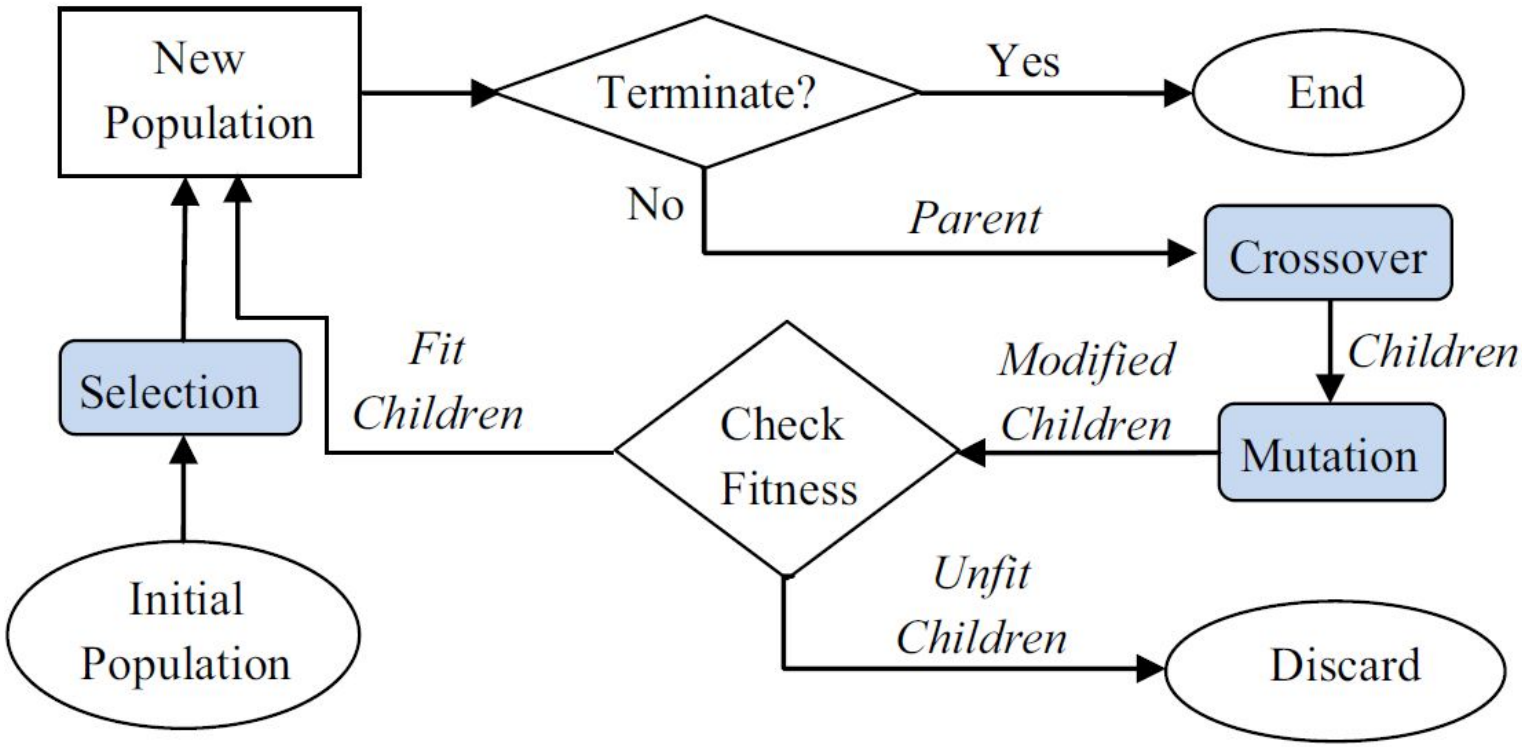
Hill-Climbing Search

Hill climbing often gets stuck for the following reasons

1. **Local maxima:** a **local maximum** is a **peak** that is **higher** than each of its **neighboring states**, but **lower** than the **global maximum**.
2. **Ridges:** Ridges result in **a sequence of local maxima** that is very difficult for **greedy algorithms** to **navigate**.
3. **Plateaux:** A plateau is an area of the state space landscape where the **evaluation function is flat**. It can be a **flat local maximum**, from which no uphill exit exists, or a **shoulder**, from which it is possible to make progress.

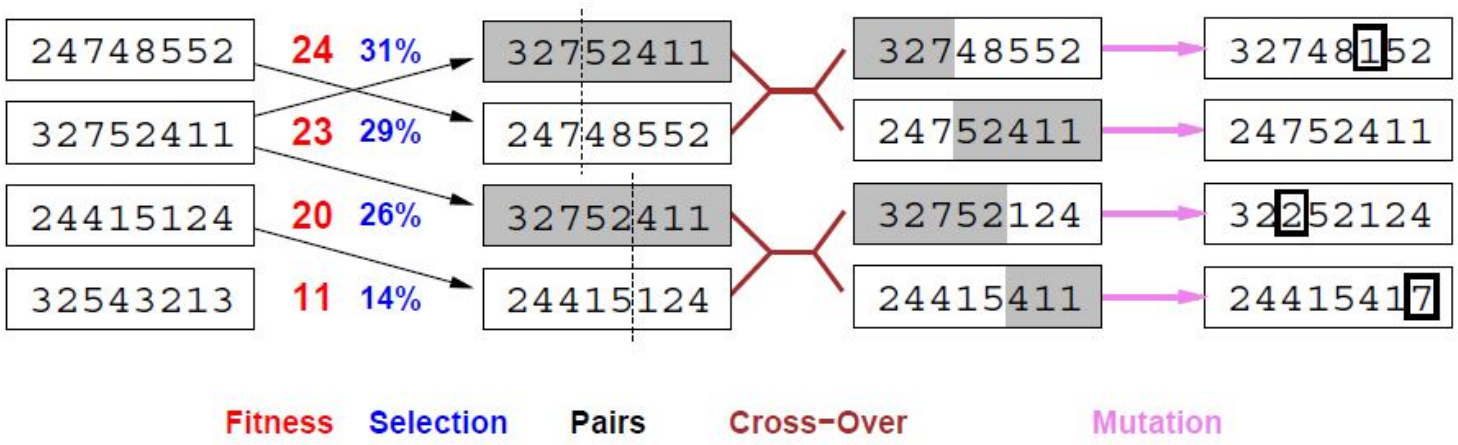


Genetic Algorithm



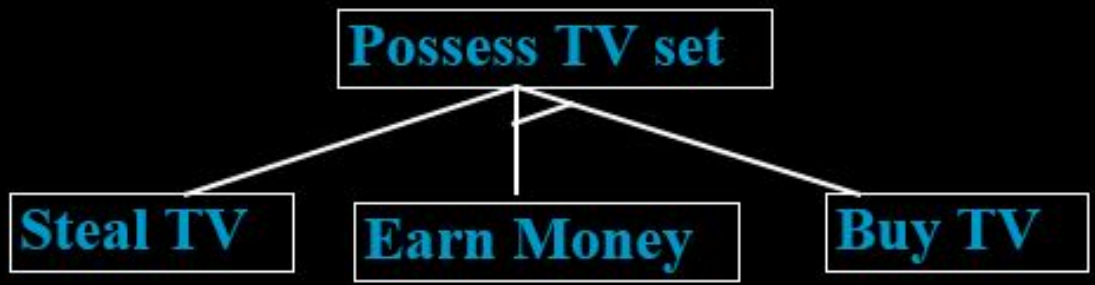
Genetic algorithms

= stochastic local beam search + generate successors from **pairs** of states



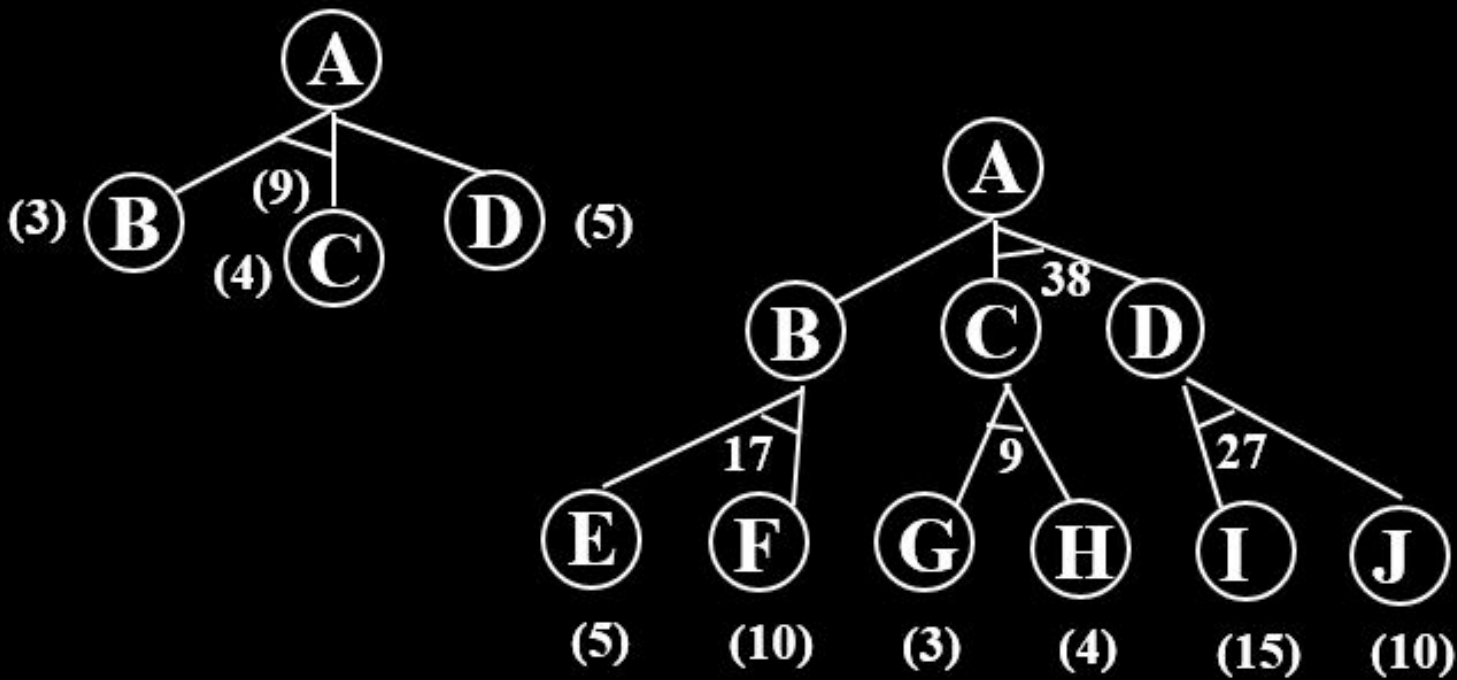
AND/OR graphs

- Some problems are best represented as achieving subgoals, some of which achieved simultaneously and independently (AND)
- Up to now, only dealt with OR options

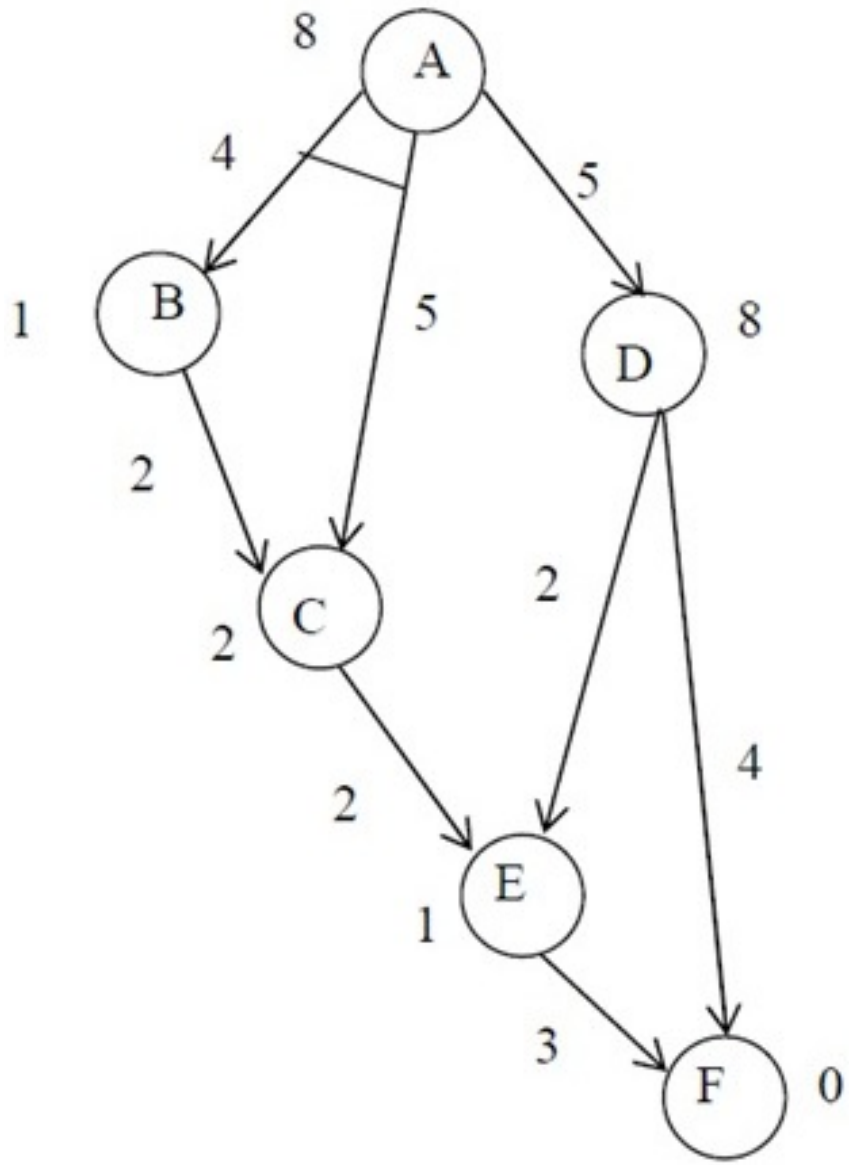


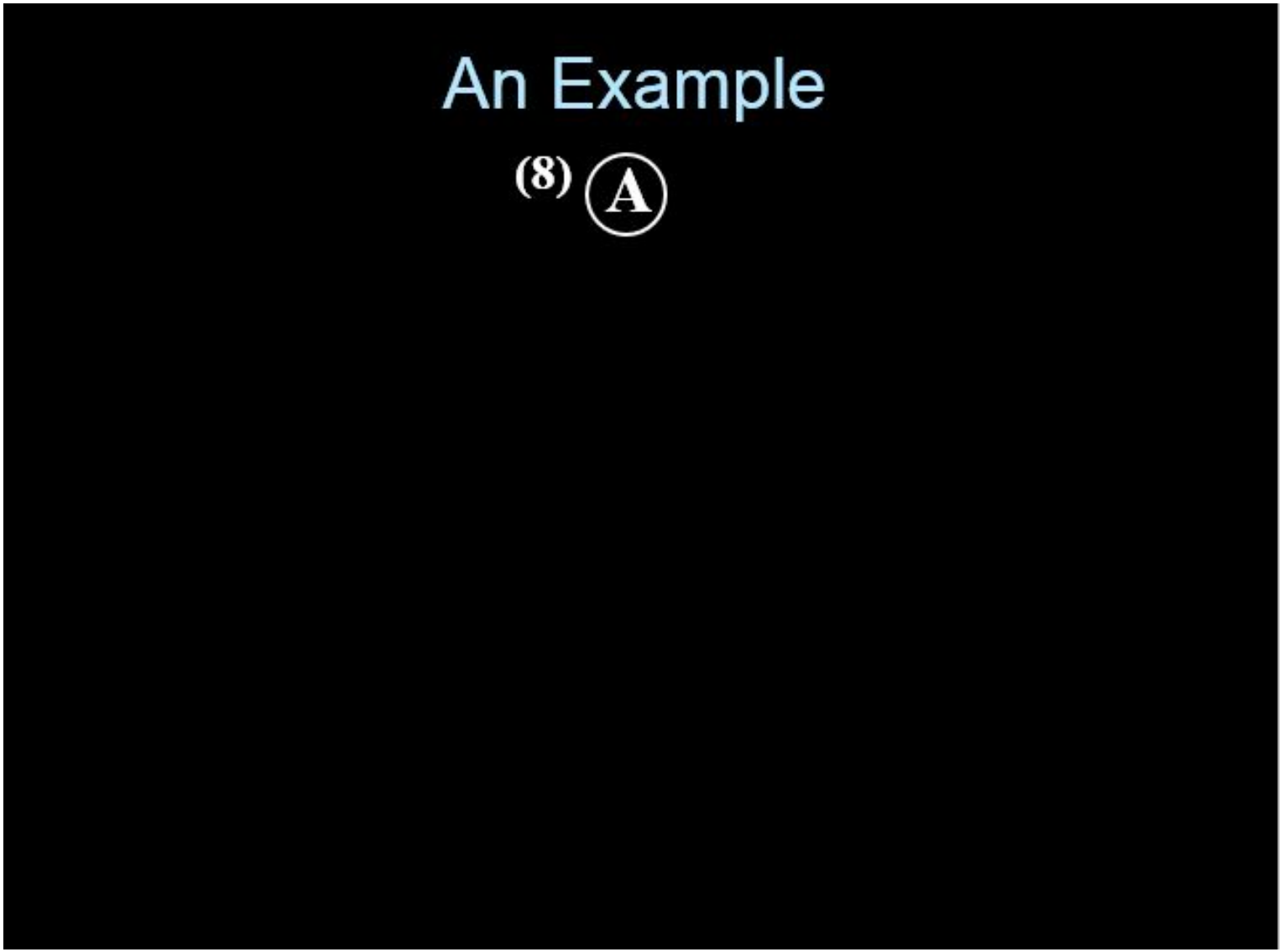
AND/OR search

- We must examine several nodes simultaneously when choosing the next move

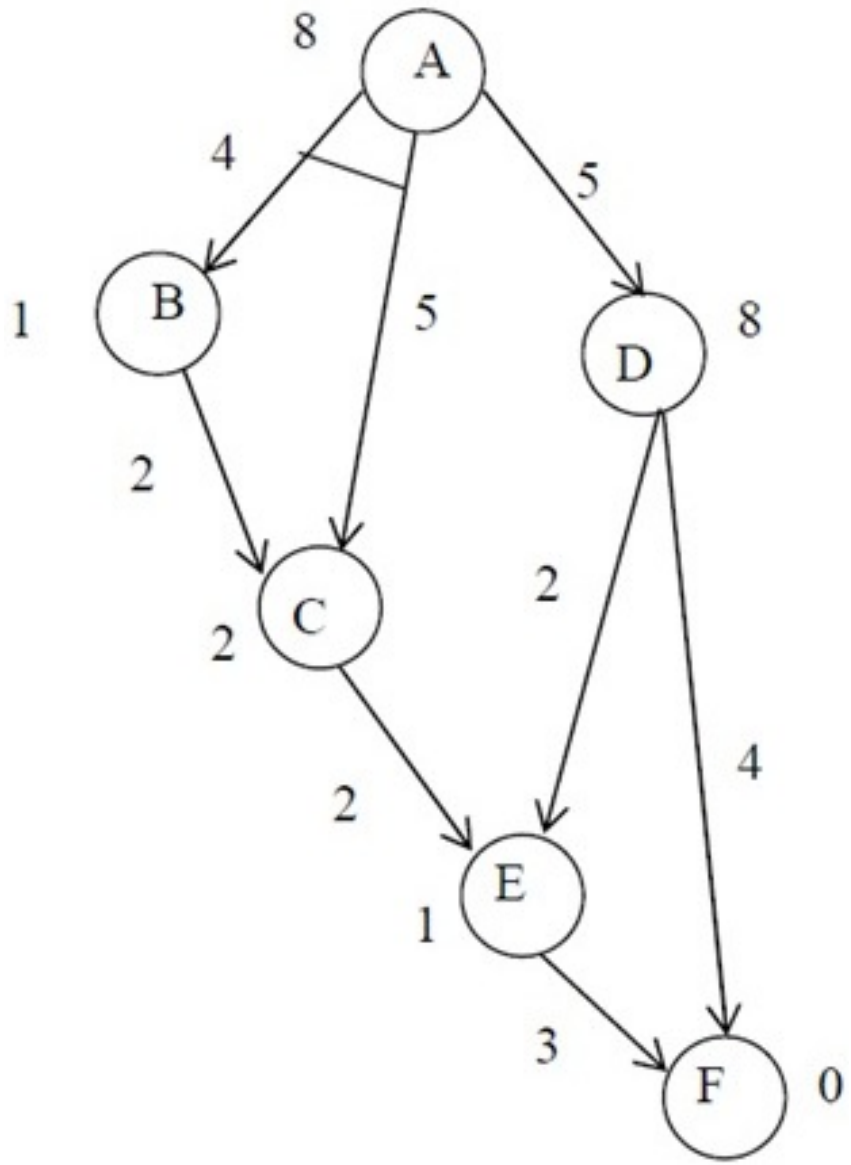


AO* Search

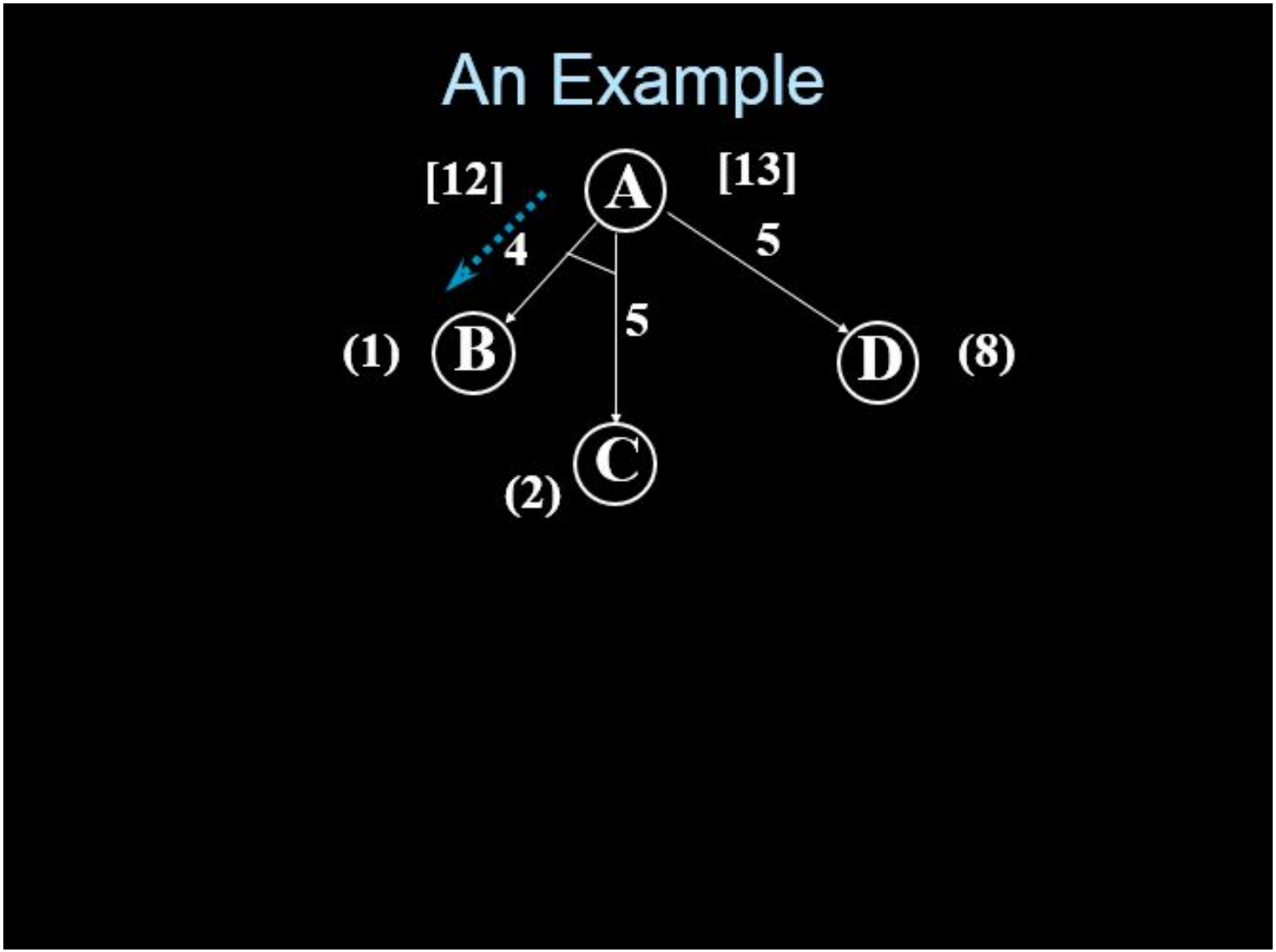



An Example
(8)
A

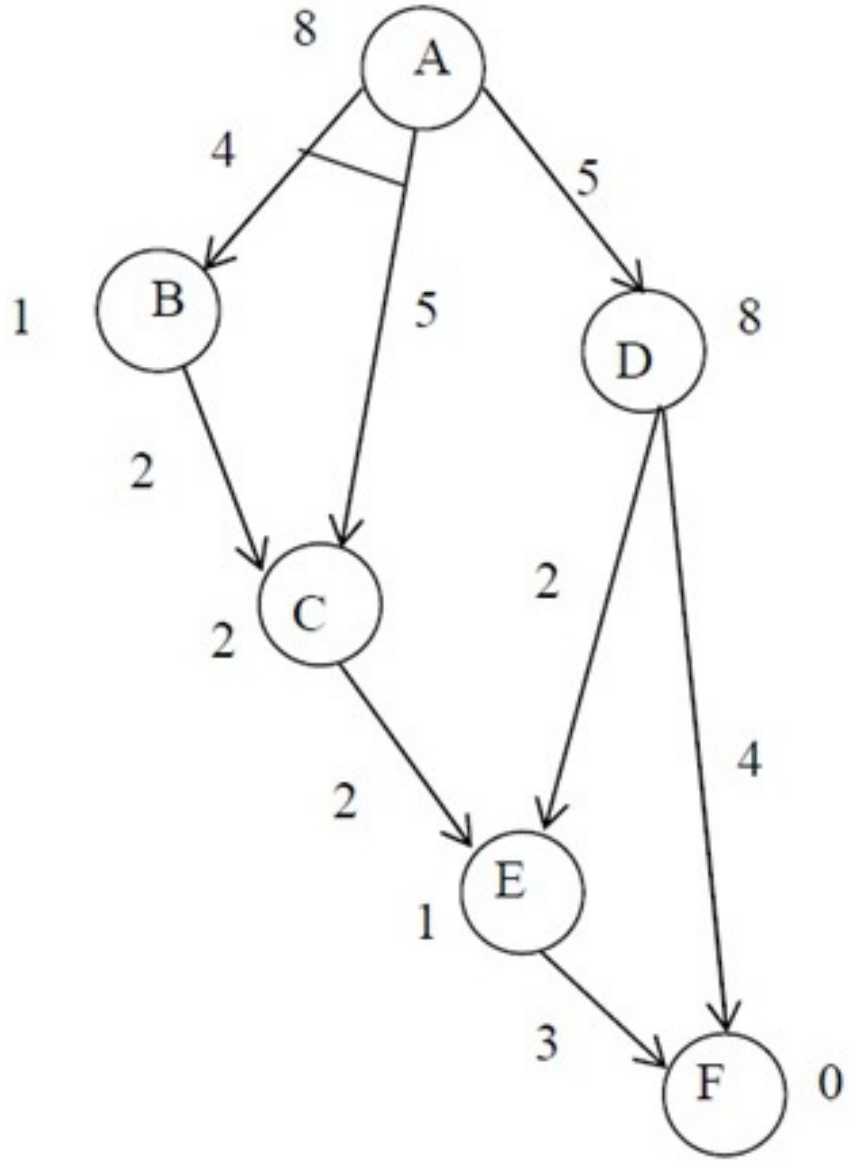
AO* Search

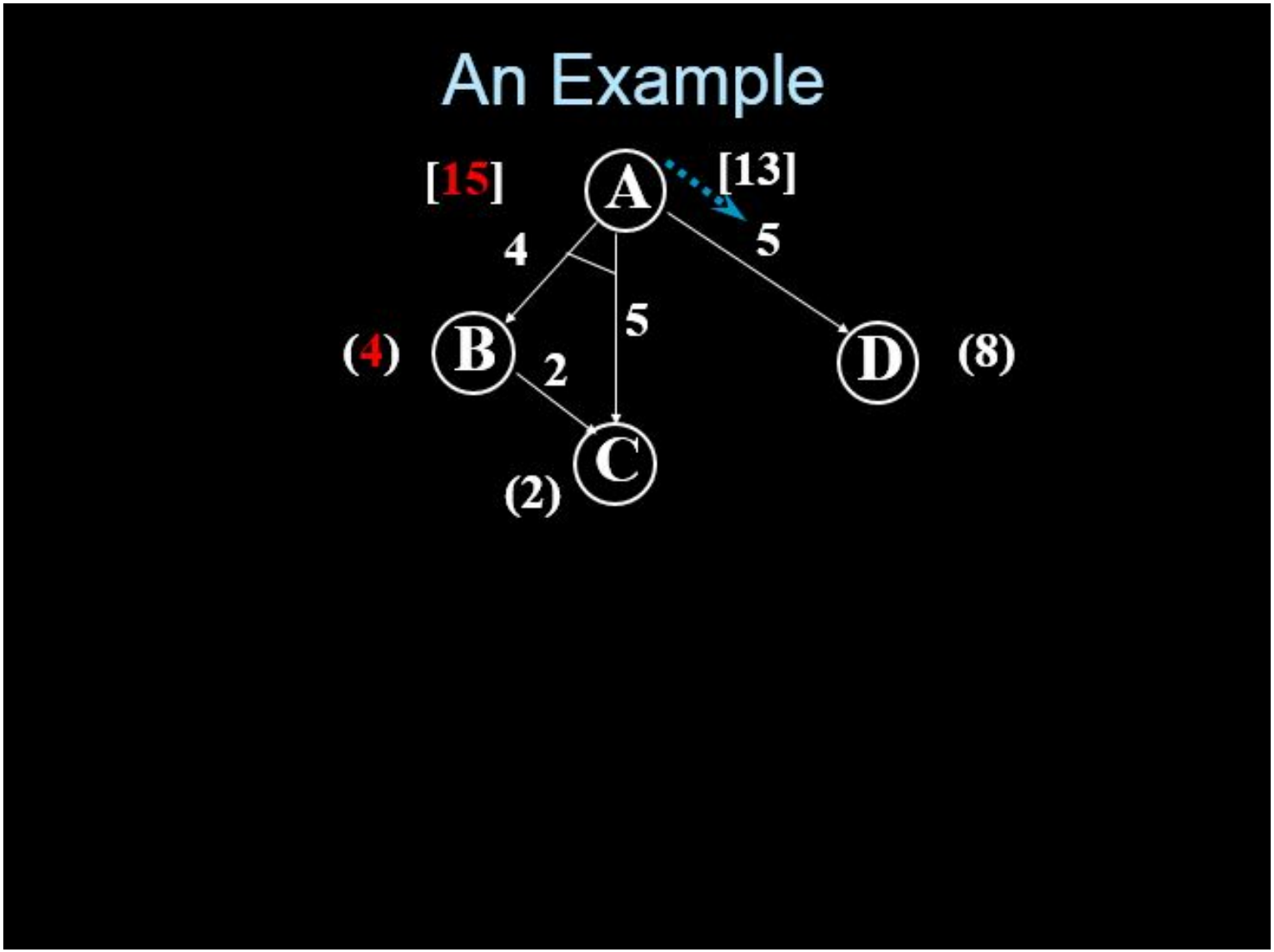


AO* Search

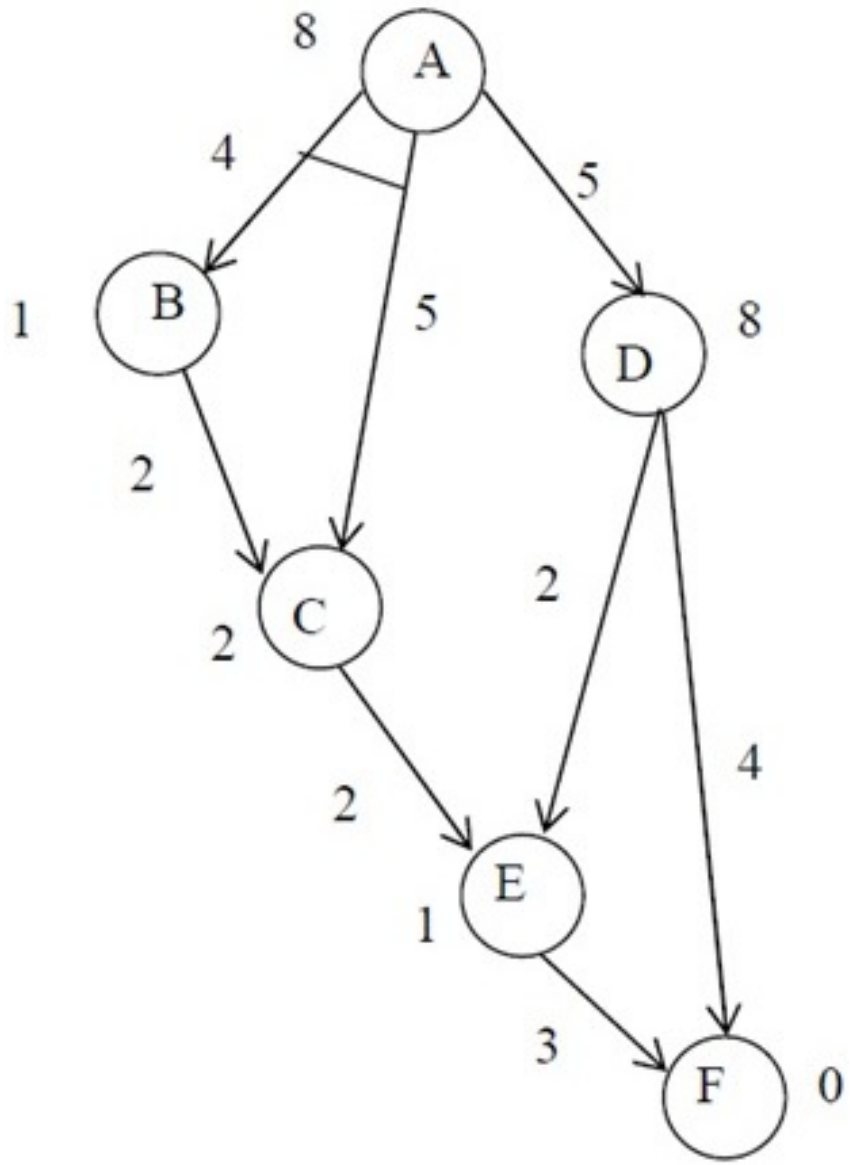


AO* Search

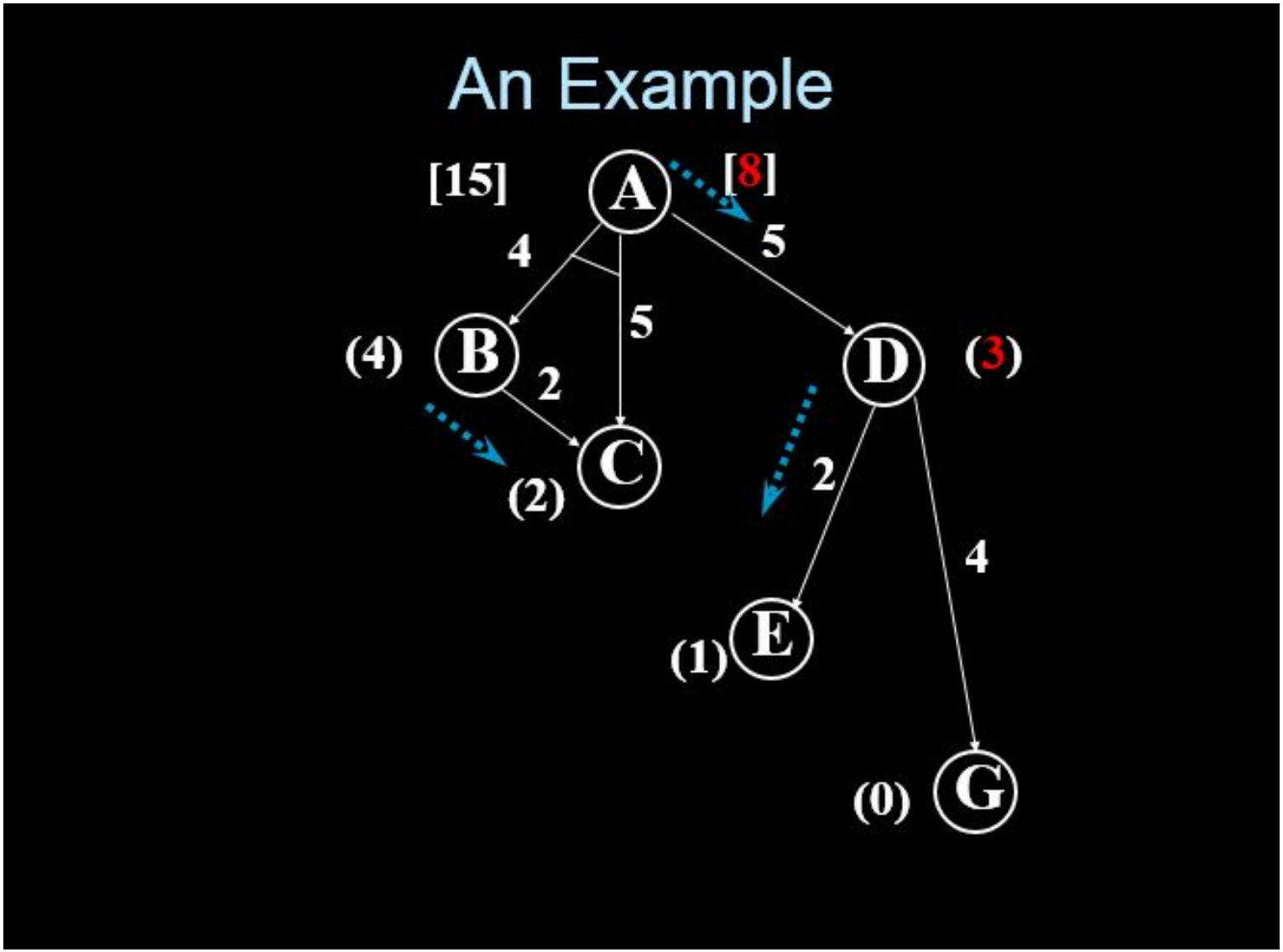



An Example
[15]
A
[13]
4
5
5
(4)
B
2
D
(8)
C
(2)

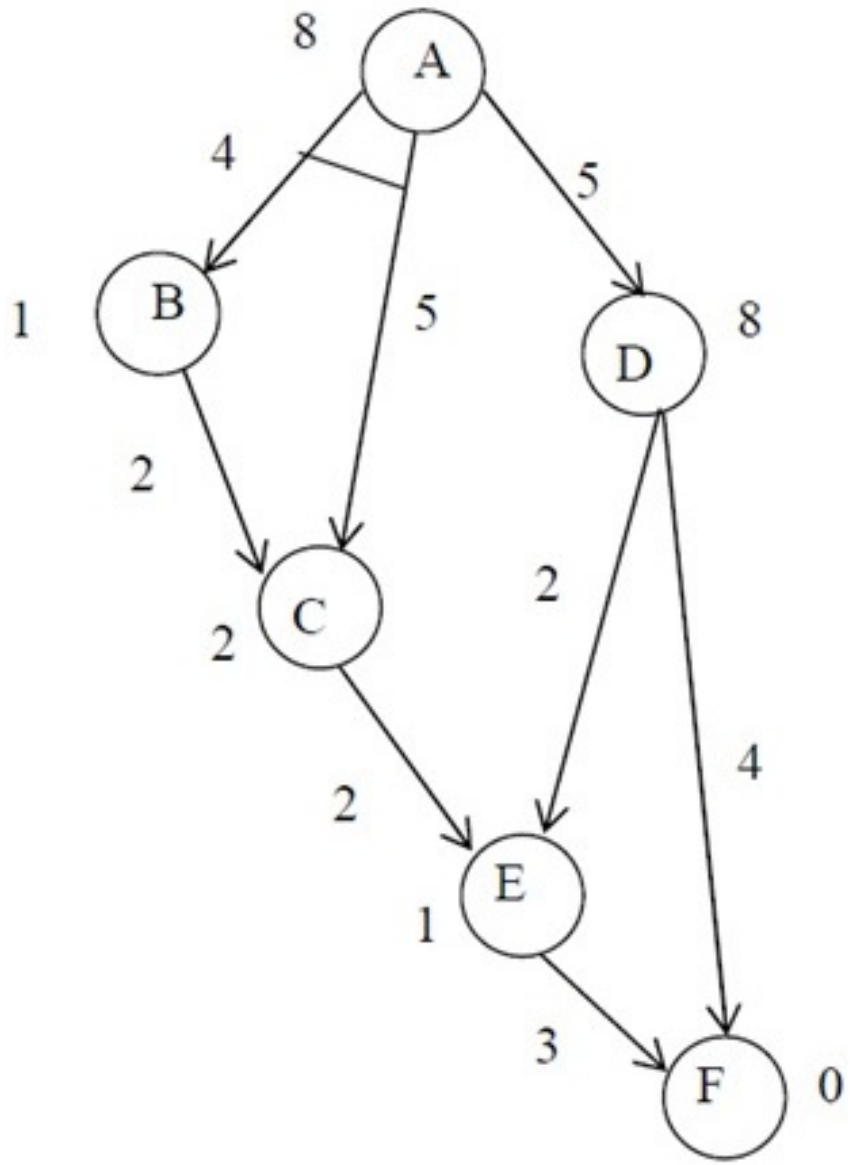
AO* Search



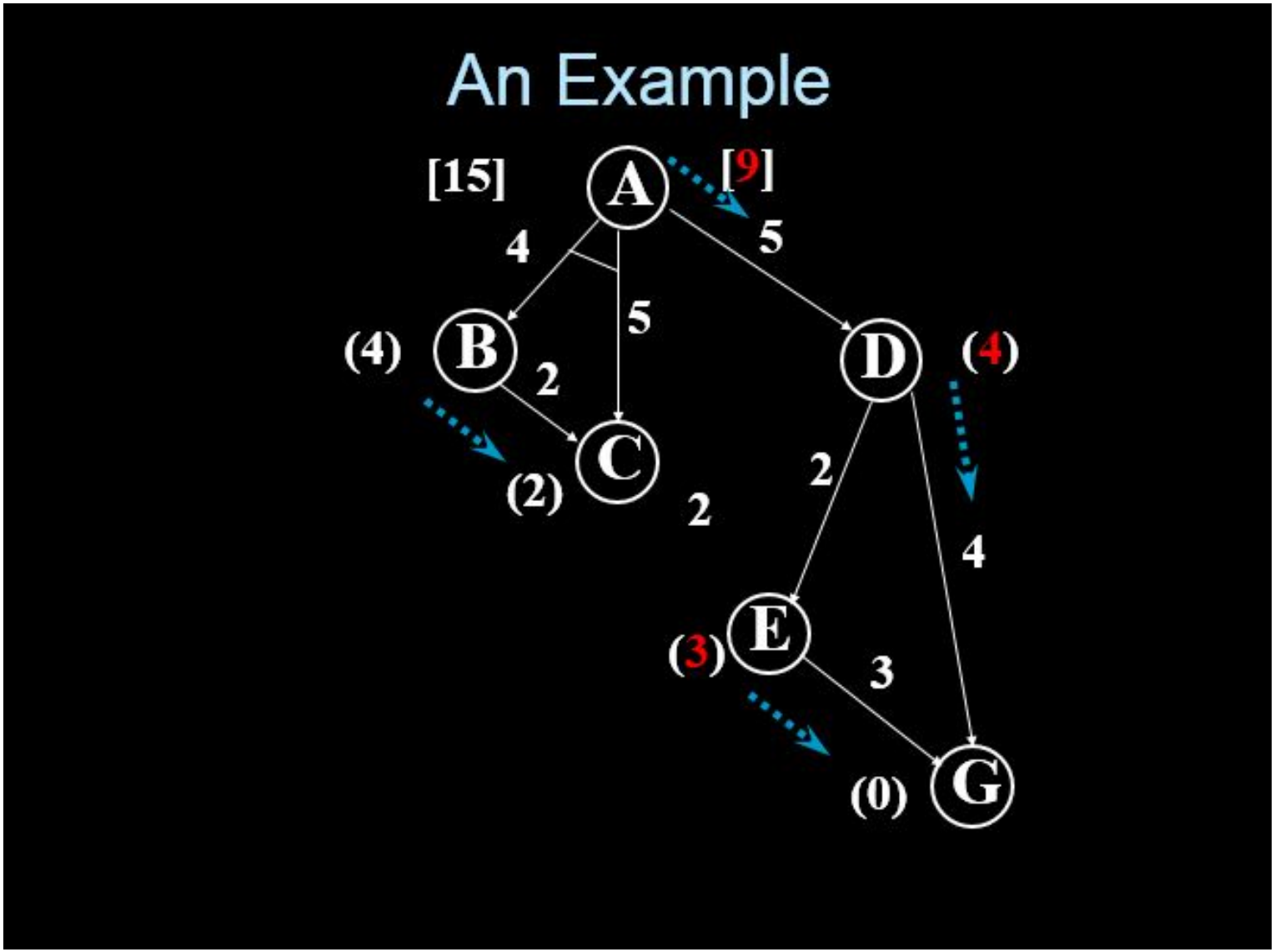
AO* Search



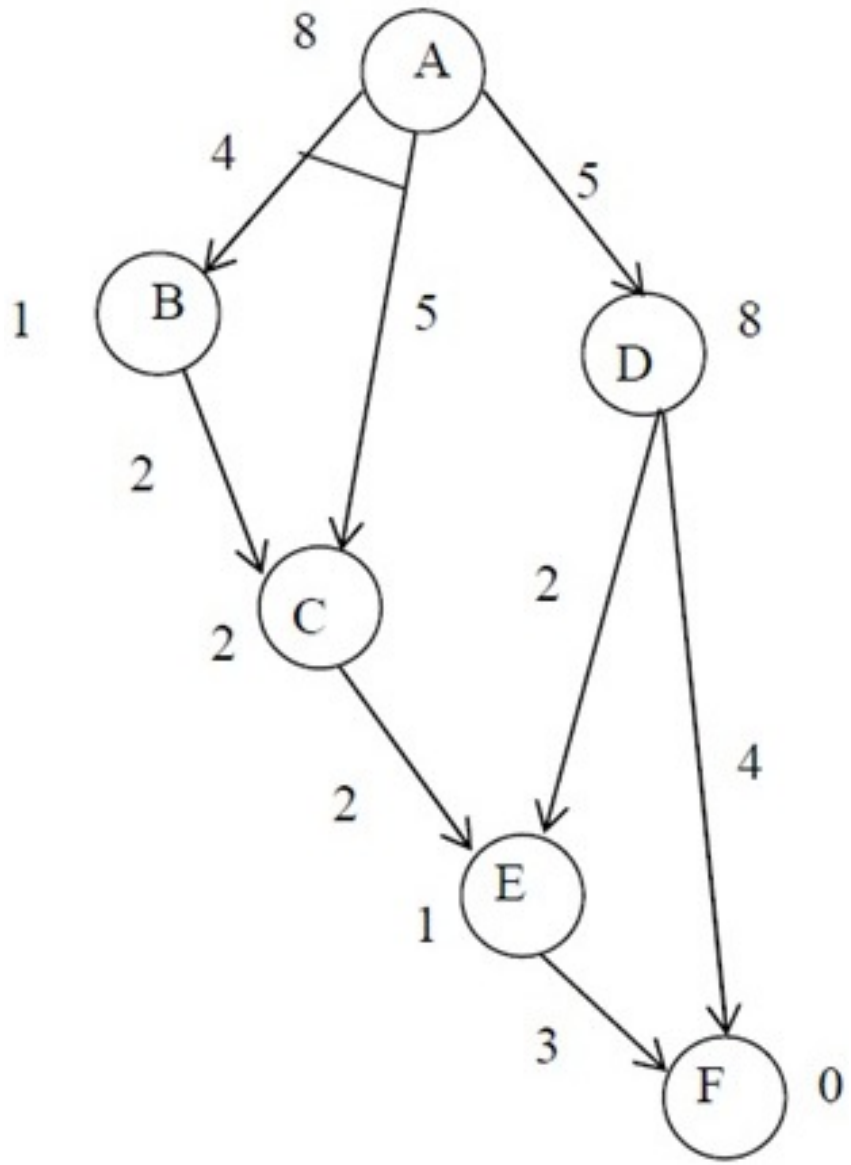
AO* Search



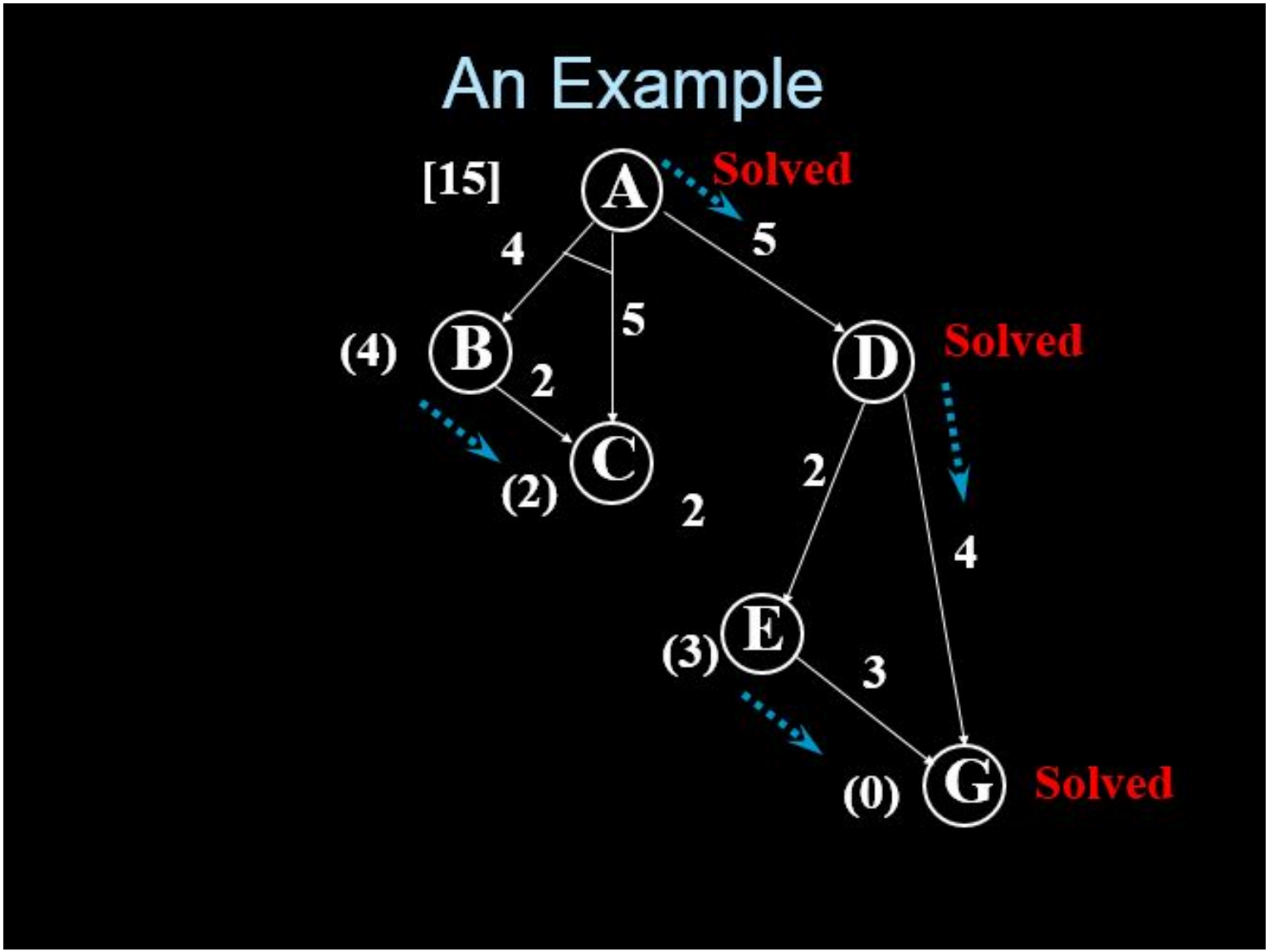
AO* Search

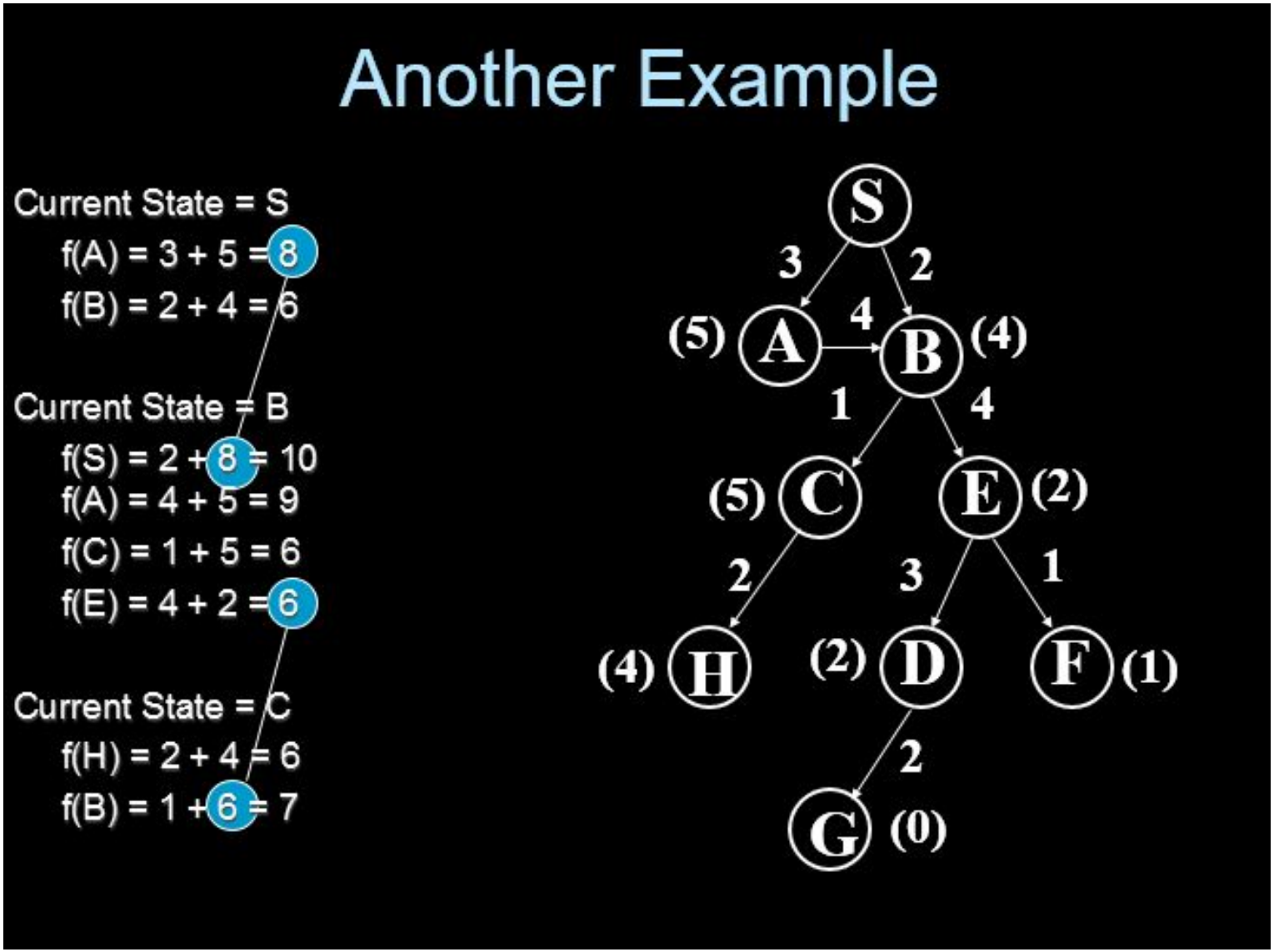


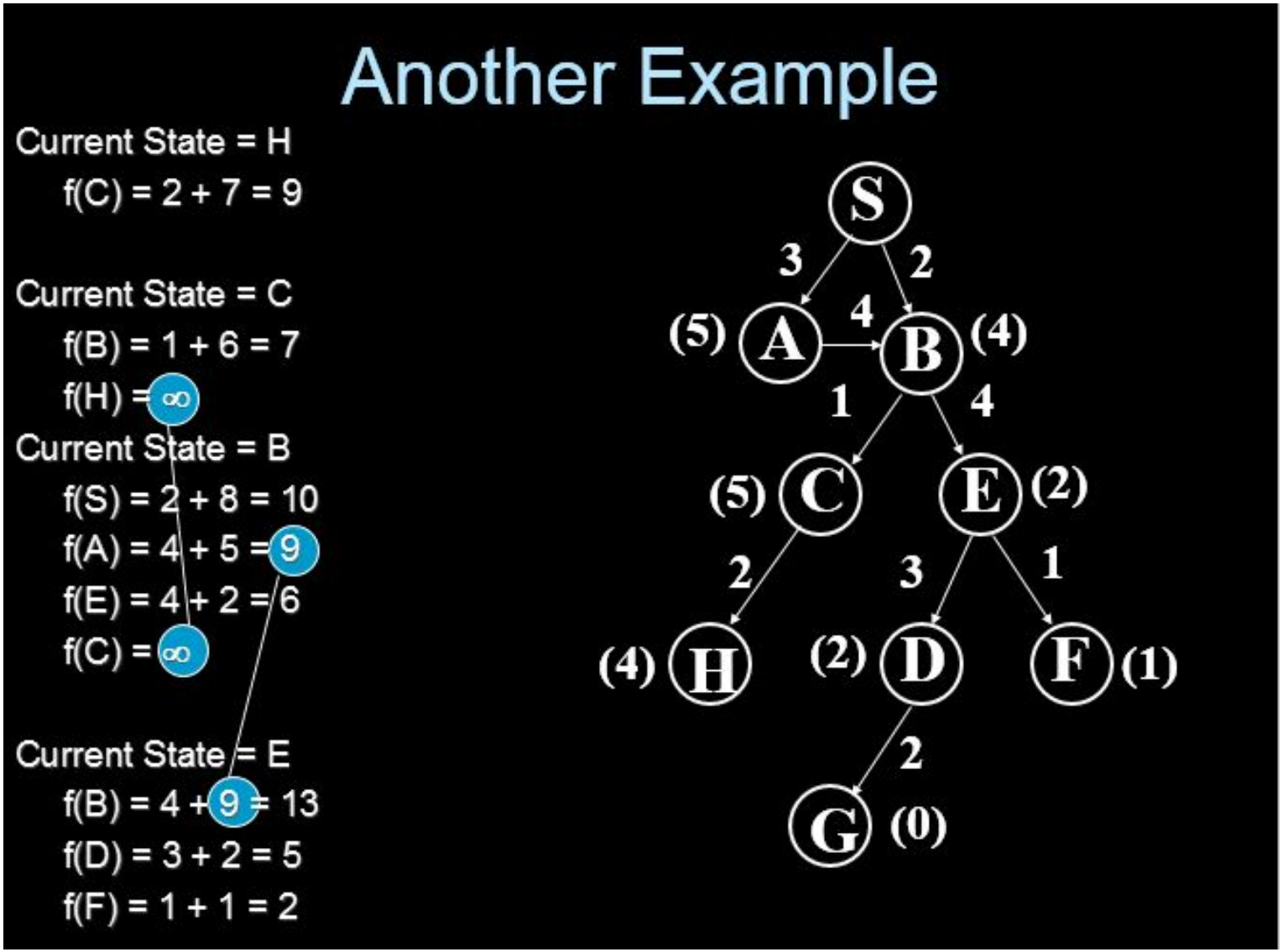
AO* Search



AO* Search




Another Example
Current State = S
f(A) = 3 + 5 = 8
f(B) = 2 + 4 = 6
Current State = B
f(S) = 2 + 8 = 10
f(A) = 4 + 5 = 9
f(C) = 1 + 5 = 6
f(E) = 4 + 2 = 6
Current State = C
f(H) = 2 + 4 = 6
f(B) = 1 + 6 = 7
S
3
2
(5) A
4
B (4)
1
4
(5) C
E (2)
2
3
1
(4) H
(2) D
F (1)
2
G (0)


Another Example
Current State = H
f(C) = 2 + 7 = 9
Current State = C
f(B) = 1 + 6 = 7
f(H) = ∞
Current State = B
f(S) = 2 + 8 = 10
f(A) = 4 + 5 = 9
f(E) = 4 + 2 = 6
f(C) = ∞
Current State = E
f(B) = 4 + 9 = 13
f(D) = 3 + 2 = 5
f(F) = 1 + 1 = 2
S
A
B
C
E
H
D
F
G
(5)
(4)
(5)
(2)
(4)
(2)
(1)
(0)
3
2
4
1
4
2
3
1
2

Another Example

Current State = F

f(E) = 1 + 5 = 6

Current State = E

f(D) = 3 + 2 = 5

f(B) = 4 + 9 = 13

f(F) = ∞

Current State = D

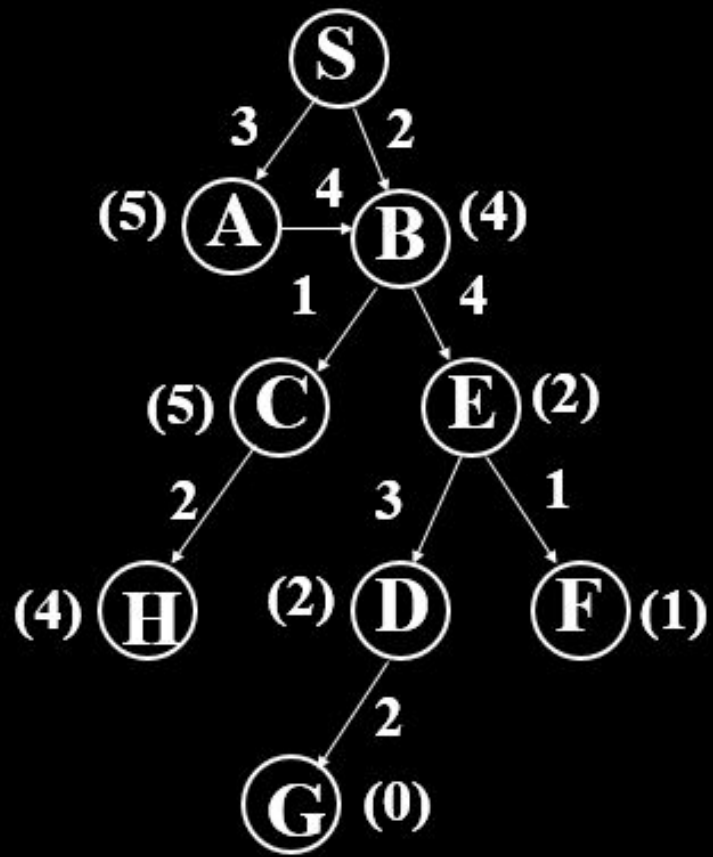
f(G) = 2 + 0 = 2

f(E) = 3 + 13 = 16

Visited Nodes =

S, B, C, H, C, B, E, F, E, D, G

Path = S, B, E, D, G



AO* Search

1. Place the starting node s on open.
2. Using the search tree constructed thus far, compute the most promising solution tree T_0.
3. Select a node n that is both on open and a part of T_0. Remove n from open and place it on closed.
4. If n is a terminal goal node, label n as solved. If the solution of n results in any of n's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where T_0 is the solution tree. Remove from open all nodes with a solved ancestor.
5. If n is not a solvable node (operators cannot be applied), label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of n's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvable ancestors.

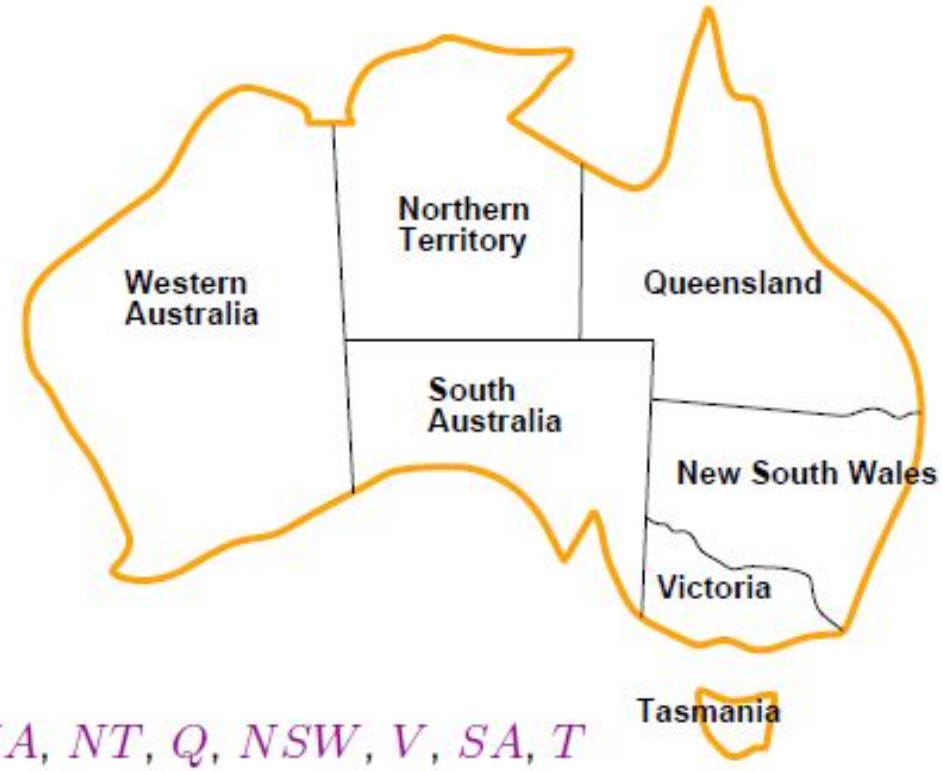
AO* Search

6. Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one subproblem, generate their successors to give individual subproblems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h^* for each newly generated node and place all such nodes that do not yet have descendants on open. Next, recompute the values of h^* at n and each ancestor of n.
7. Return to step 2.

Constraint Satisfaction Problems (CSP)

- A CSP is defined by a set of **variables**, X_1, X_2,..., X_n and a set of **constraints**, C_1, C_2,..., C_m.
- Each variable X_i has a nonempty domain D_i of possible values.
- A state of the problem is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$.
- An assignment that does not violate any constraints is called a **consistent** or legal assignment.

Example: Map-Coloring



Variables WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

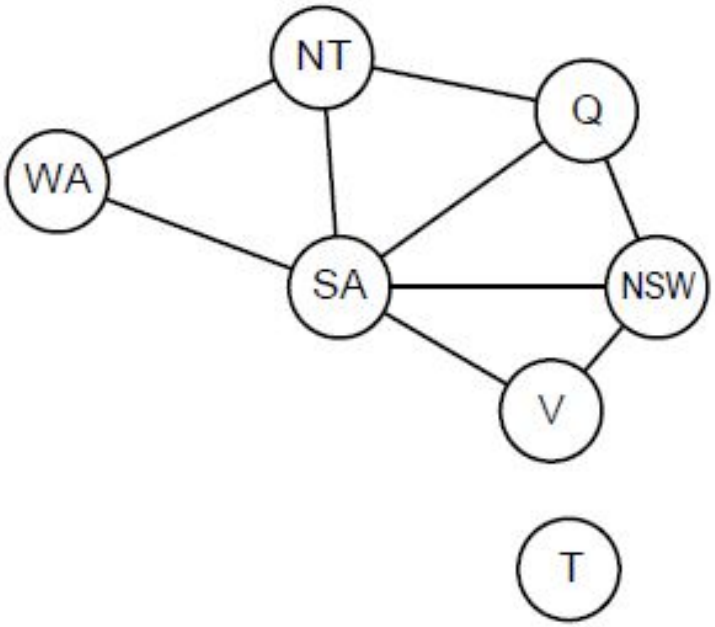
Example: Map-Coloring contd.

Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Varieties of CSPs

Discrete variables

finite domains; size d $\Rightarrow$ $O(d^n)$ complete assignments

- ♢ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ♢ e.g., job scheduling, variables are start/end days for each job
- ♢ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ♢ linear constraints solvable, nonlinear undecidable

Continuous variables

- ♢ e.g., start/end times for Hubble Telescope observations
- ♢ linear constraints solvable in poly time by LP methods

Varieties of constraints

Unary constraints involve a single variable,
e.g., $SA \neq green$

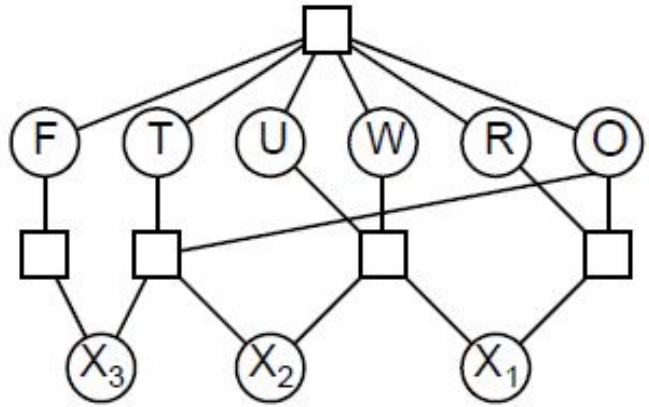
Binary constraints involve pairs of variables,
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,
e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$
often representable by a cost for each variable assignment
$\rightarrow$ constrained optimization problems

Example: Cryptarithmetic

$$\begin{array}{r} \ T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
$alldiff(F,T,U,W,R,O)$
$O+O=R+10\cdot X_1$, etc.

- $X_1 + W + W = U + 10 \times X_2$
- $X_2 + T + T = O + 10 \times X_3$
- $X_3 = F$

Real-world CSPs

Assignment problems
e.g., who teaches what class

Timetabling problems
e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

- Prof. X might prefer teaching in the morning whereas Prof. Y prefers teaching in the afternoon.
- A timetable that has Prof. X teaching at 2 p.m. would still be a **solution**, but would **not** be an **optimal** one.

Backtracking search

Variable assignments are commutative, i.e.,

$[WA = red \text{ then } NT = green]$ same as $[NT = green \text{ then } WA = red]$

Only need to consider assignments to a single variable at each node

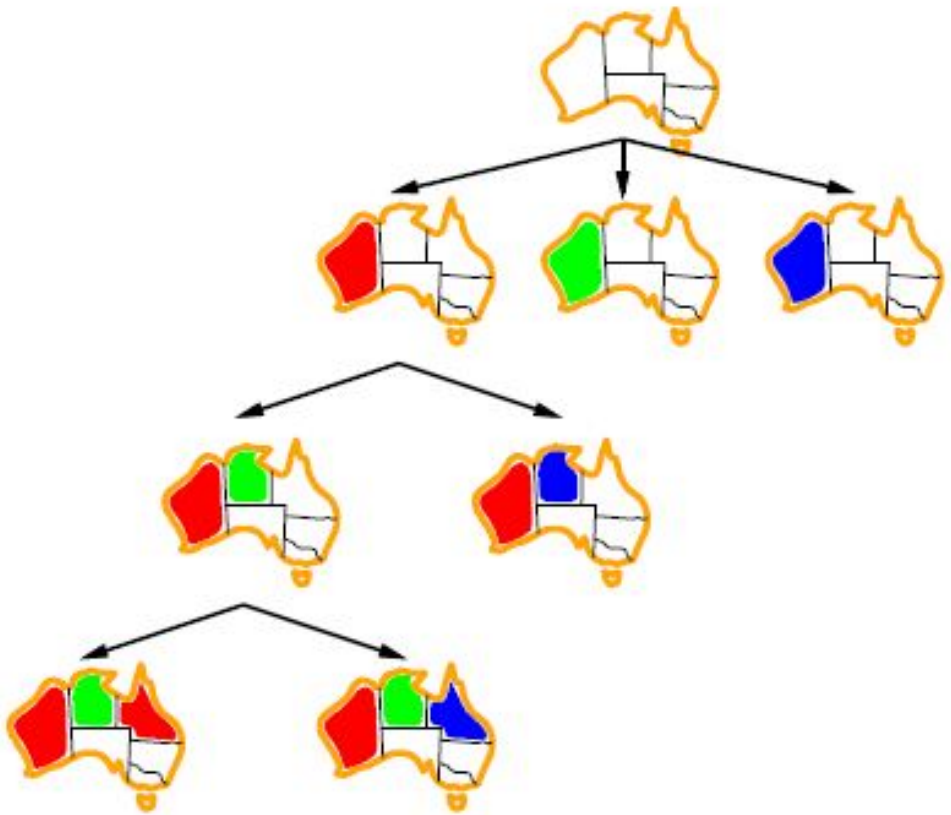
$\Rightarrow \quad b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

Can solve n-queens for $n \approx 25$

Backtracking example



Improving backtracking efficiency

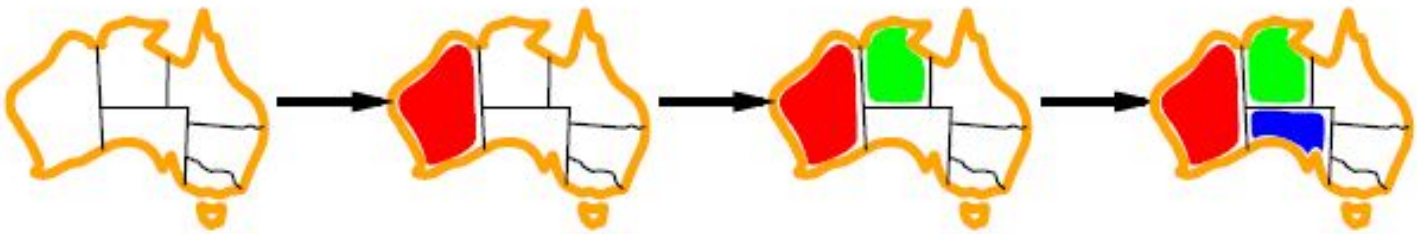
General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Minimum remaining values

Minimum remaining values (MRV):

choose the variable with the fewest legal values

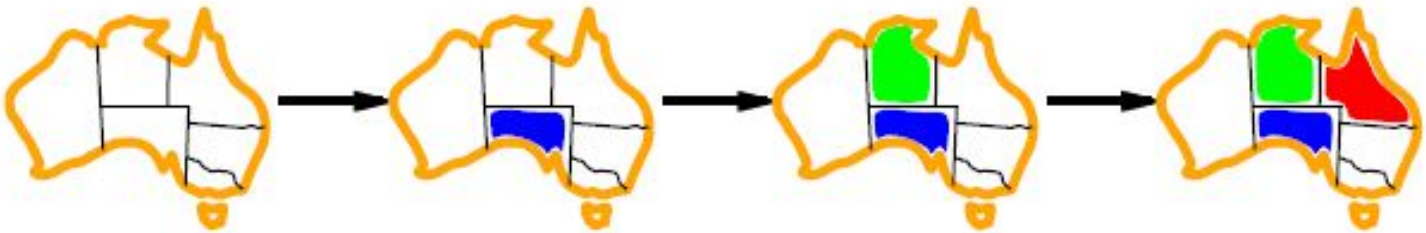


Degree heuristic

Tie-breaker among MRV variables

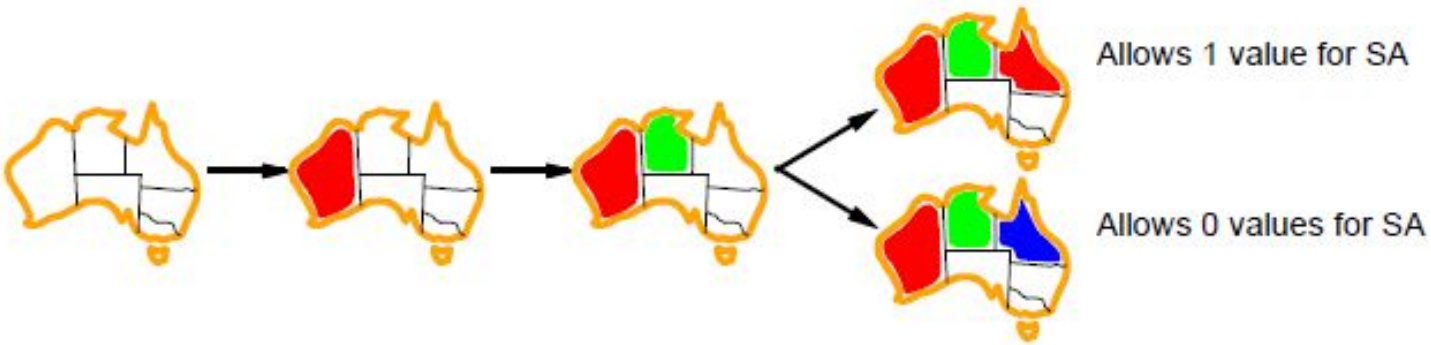
Degree heuristic:

choose the variable with the most constraints on remaining variables



Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



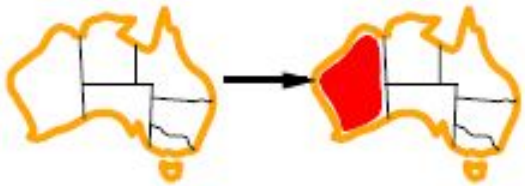
Combining these heuristics makes 1000 queens feasible

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

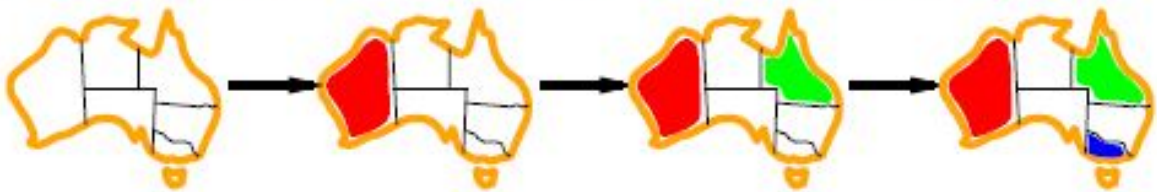


Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc consistency

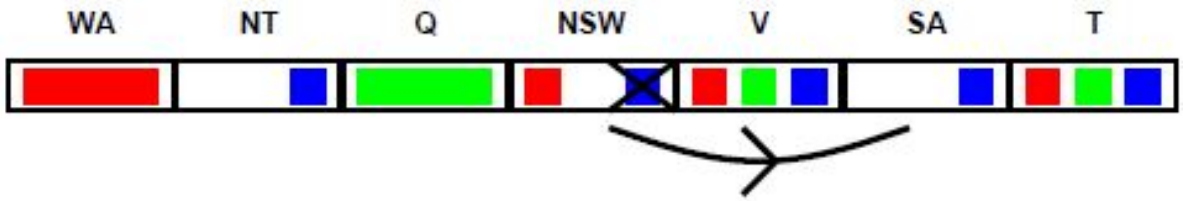
Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff
for every value x of X there is some allowed y

Arc consistency

Simplest form of propagation makes each arc consistent

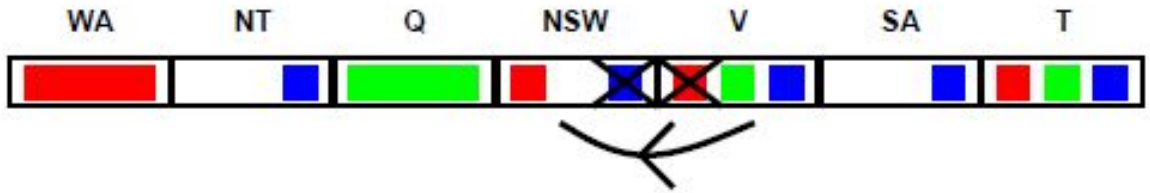
$X \rightarrow Y$ is consistent iff
for every value x of X there is some allowed y



Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff
for every value x of X there is some allowed y



If X loses a value, neighbors of X need to be rechecked

Adversarial Search - Games

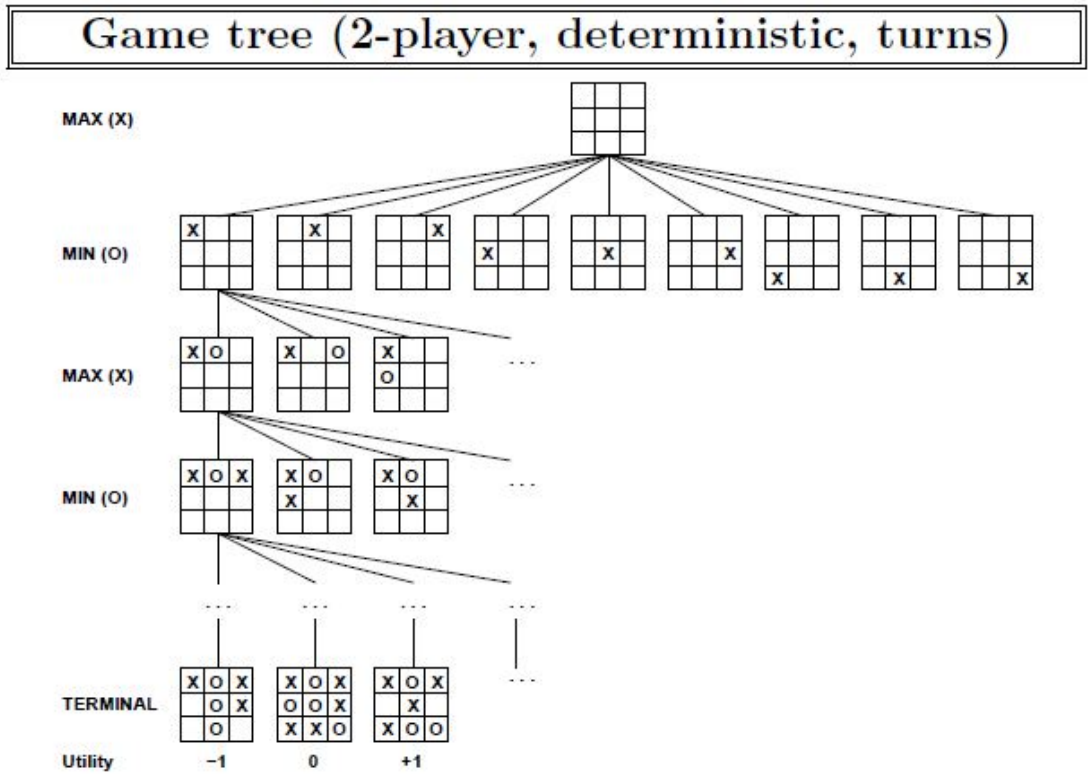
- Distinction between **cooperative** and **competitive** multiagent environments
- **Competitive environments**, in which the agents' goals are in **conflict**, give rise to **adversarial search** problems-often known as **games**.
- It means **deterministic**, **fully observable** environments in which there are **two agents** whose actions must **alternate** and in which the **utility values** at the **end of the game** are always **equal** and **opposite**.
  - If one player **wins** a game of chess (+1), the other player necessarily **loses** (-1).
  - **Chess** has an **average branching factor** of about **35** and games often go to **50 moves** by each player, so the search tree has about $35^{(50+50)}$ or 10^{154} nodes (although the search graph has "only" about 10^{40} distinct nodes).

Adversarial Search - Games

- **Pruning** allows us to **ignore portions** of the search tree that make no difference to the final choice and heuristic **evaluation function** allows us to approximate the true utility of a state without doing a complete search.
- We will consider games with **two players**, whom we will call **MAX** and **MIN**.
- **MAX** moves first and then they take turns moving until **the game is over**.
- At the **end of the game**, **points are awarded** to the **winning player** and **penalties** are given to the **loser**.

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (move, state) pairs, each indicating a **legal move** and the **resulting state**.
- **Terminal test** determines when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an **objective function** or **payoff function**), which gives a **numeric value** for the terminal states. In chess, the outcome is a **win**, **loss**, or **draw**, with values **+1**, **-1**, or **0**.
- Some games have a wider, variety of possible outcomes; the **payoffs in backgammon** range from **+192** to **-192**.

Game Tree - Tic-Tac-Toe



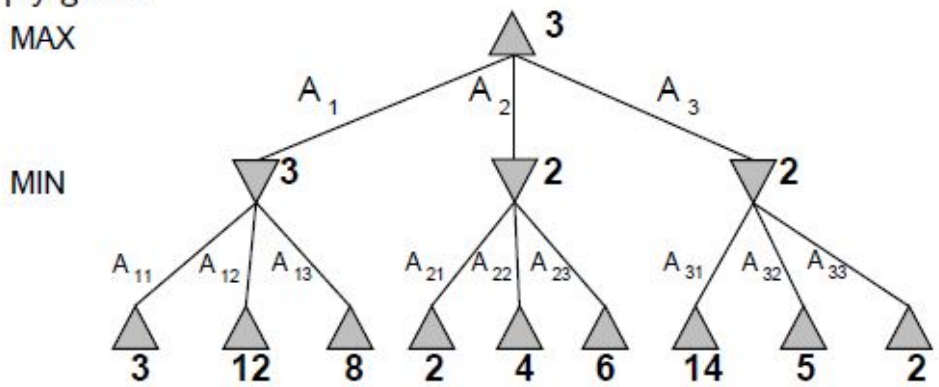
- From the initial state, MAX has **nine** possible moves.
- 9! = 362880 terminal nodes
- Play alternates between MAX'S placing an X and MIN'S placing an 0 until we reach leaf nodes corresponding to terminal states such that one player has **three in a row** or all the **squares are filled**.

Minimax

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



- MAX: Upper triangle and MIN: Lower triangle
- MAX'S best move at the root is A_1, it leads to the highest minimax value and MIN'S best reply is A_{11}, because it leads to the successor with the lowest minimax value.

Optimal Decisions in Games - The minimax Algorithm

- Given a game tree, the optimal strategy can be determined by examining the **minimax value** of each node, which we write as MINIMAX(n).

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if n is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exact solution completely infeasible

But do we need to explore every path?

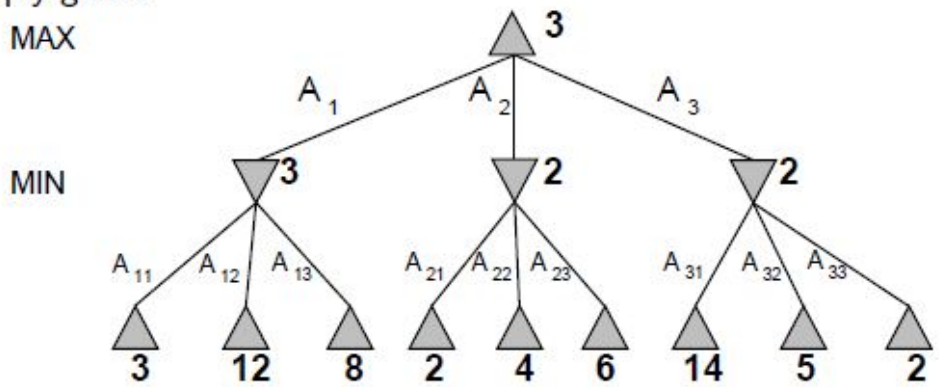
- The first MIN node, has three successors with values 3, 12, and 8, so its minimax value is 3.
- The root node is a MAX node; its successors have minimax values 3, 2, and 2; so it has a minimax value of 3.

Minimax

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



- MAX: Upper triangle and MIN: Lower triangle
- MAX'S best move at the root is A_1, it leads to the highest minimax value and MIN'S best reply is A_{11}, because it leads to the successor with the lowest minimax value.

Optimal Decisions in Games - The minimax Algorithm

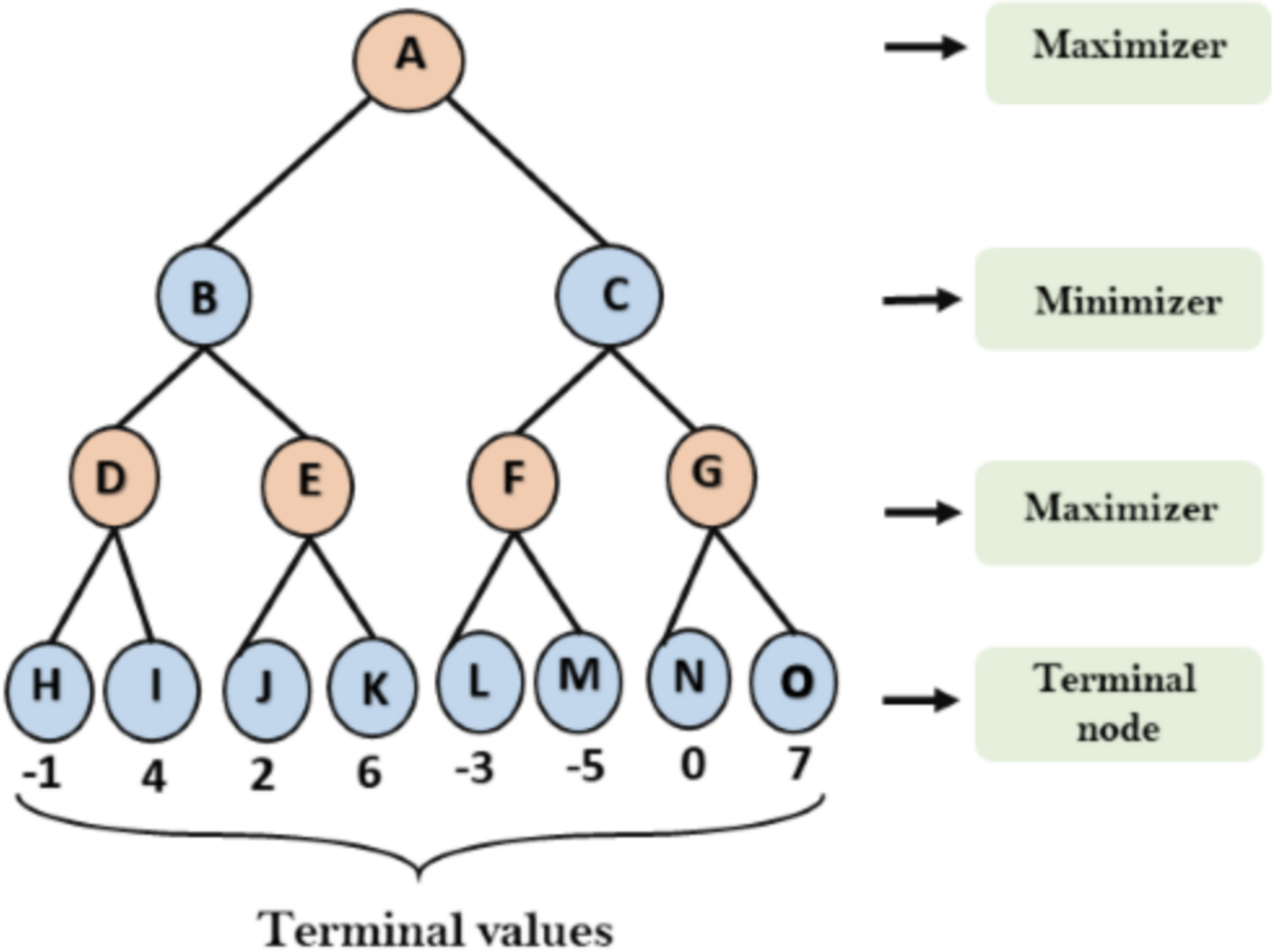
- Given a game tree, the optimal strategy can be determined by examining the **minimax value** of each node, which we write as MINIMAX(n).

Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)       //gives Maximum of the values
return maxEva

else                          // for Minimizer player
 minEva= +infinity
 for each child of node do
 eva= minimax(child, depth-1, true)
 minEva= min(minEva, eva)       //gives minimum of the values
 return minEva
```

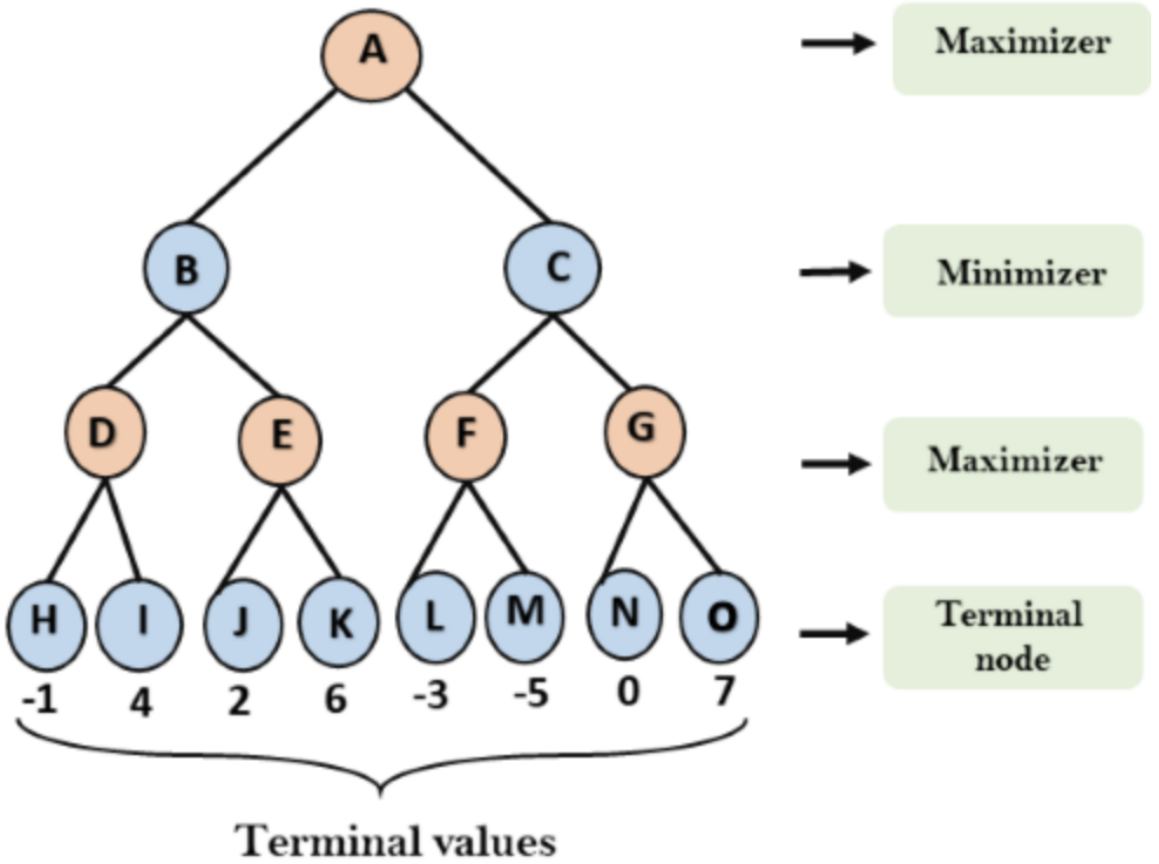
Optimal Decisions in Games - The minimax Algorithm

- Given a game tree, the optimal strategy can be determined by examining the **minimax value** of each node, which we write as MINIMAX(n).



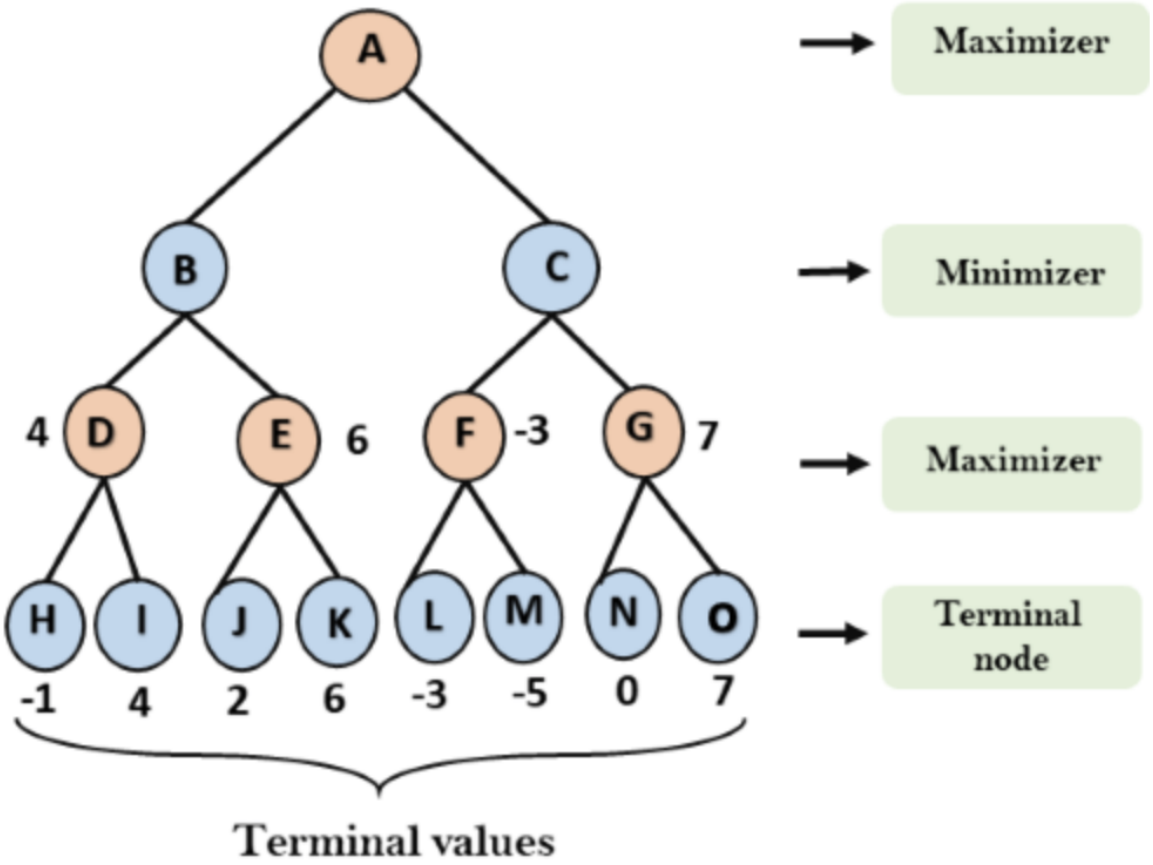
Optimal Decisions in Games - The minimax Algorithm

- Now, first we find the utilities value for the Maximizer, its initial value is $-\infty$, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.



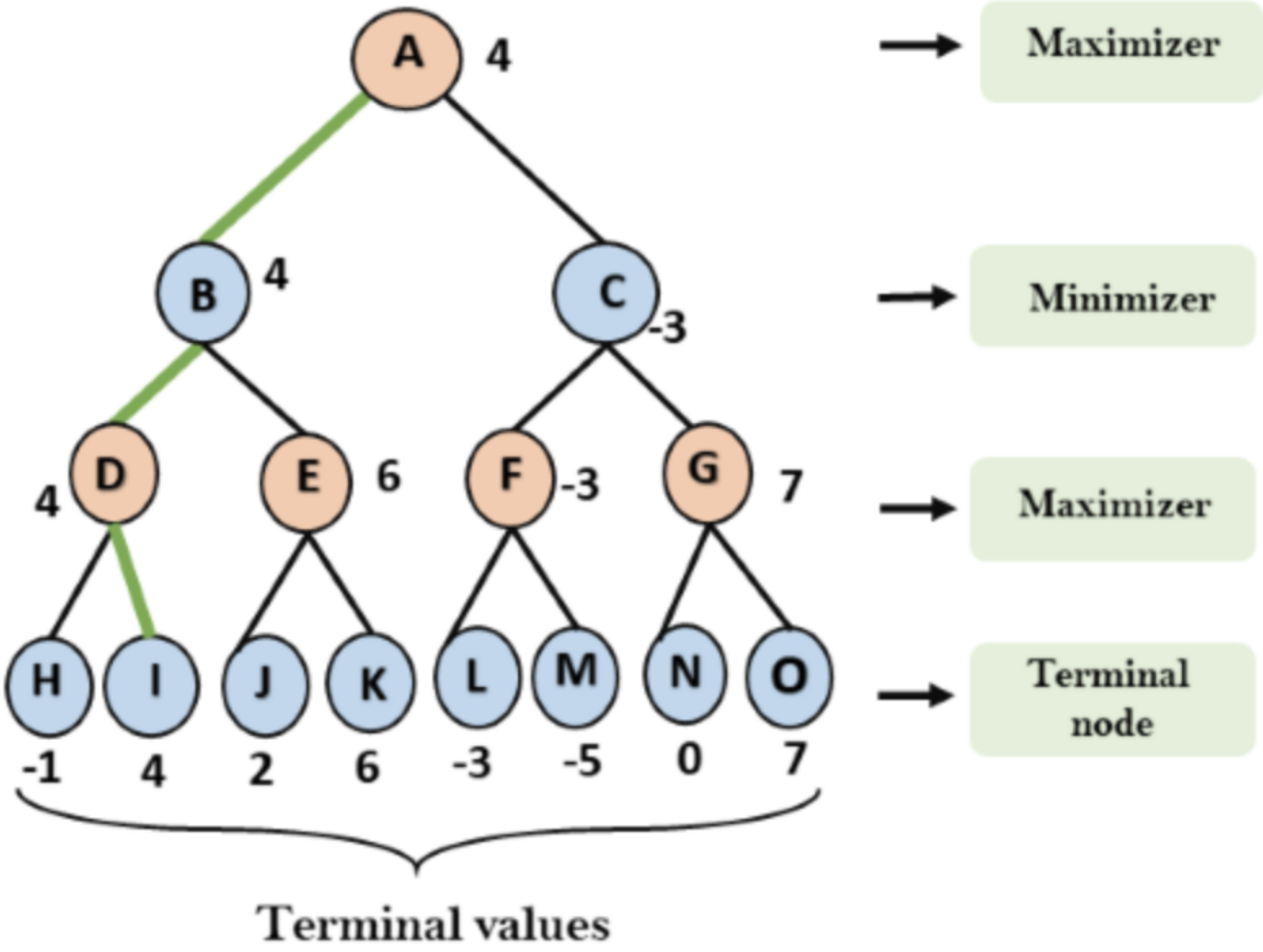
Optimal Decisions in Games - The minimax Algorithm

- For node D max(-1, -∞) = max(-1,4)= 4.
- For Node E max(2, -∞) = max(2, 6)= 6
- For Node F max(-3, -∞) = max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



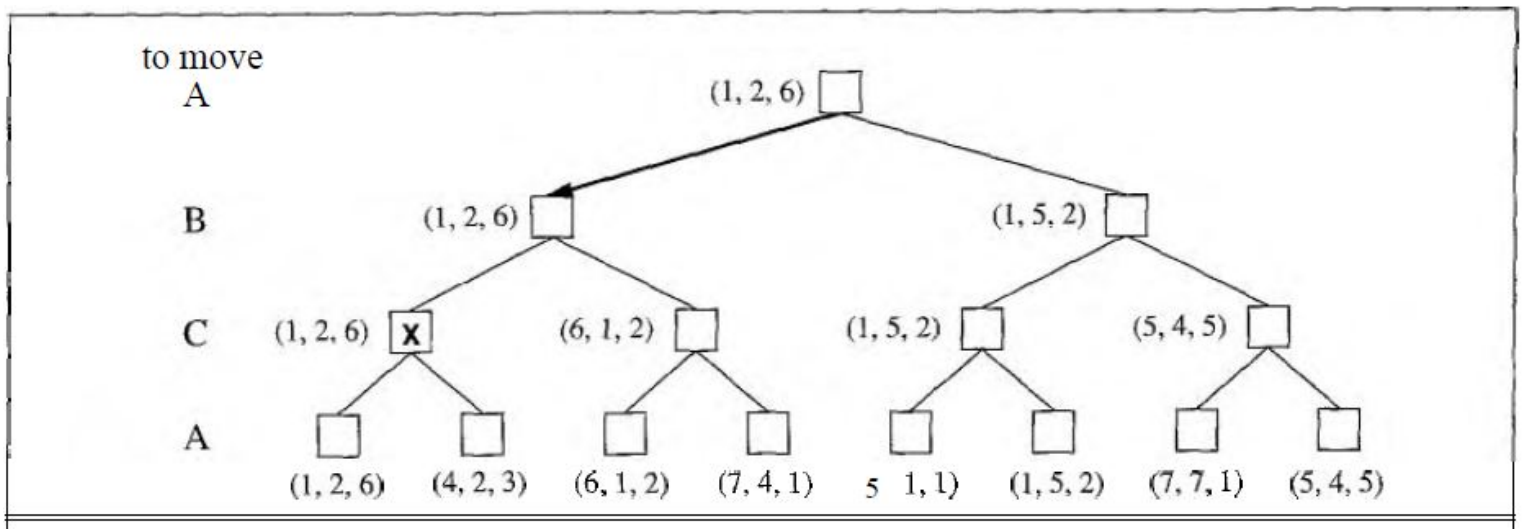
Optimal Decisions in Games - The minimax Algorithm

- For node A max(4, -3)= 4



Optimal Decisions in Multiplayer Games

- Three Players: A, B, C, Vector: $<v_A, v_B, v_C>$
- (1, 2, **6**) and (4, 2, **3**) = (1, 2, 6) (Player C)



Optimal Decisions in Multiplayer Games

- Suppose A and B are in **weak positions** and C is in a **stronger position**.
- Then it is often **optimal** for **both A and B** to **attack C** rather than each other.
- Let **C** destroy each of them **individually**.
- In this way, collaboration emerges from purely **selfish behavior**. Of course, as soon as **C weakens** under the **joint onslaught**, the **alliance loses** its value, and **either A or B** could **violate** the **agreement**.

Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree.
- Since we cannot eliminate the exponent, but we can cut it to half.
- Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning.
- This involves two threshold parameter Alpha and Beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

Alpha-Beta Pruning

- The two-parameter can be defined as:
- Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞
- Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is $+\infty$.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Alpha-Beta Pruning

Conditions for Alpha-Beta Pruning

- The main condition which required for alpha-beta pruning is: $\alpha > \beta$

Key points about alpha-beta pruning:

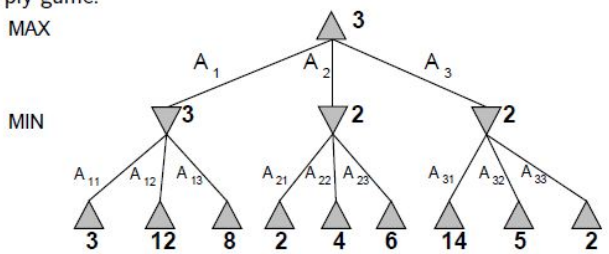
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Minimax

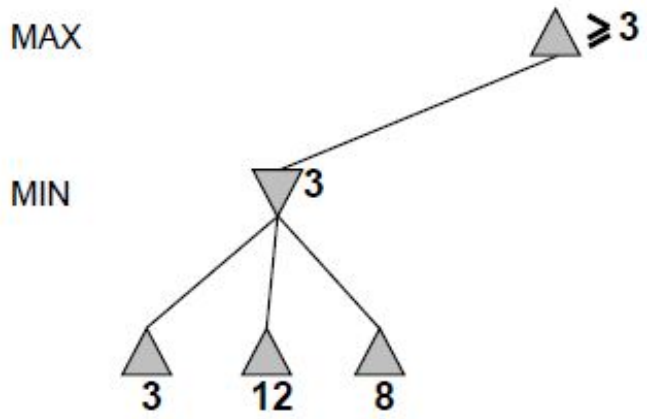
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



α–β pruning example

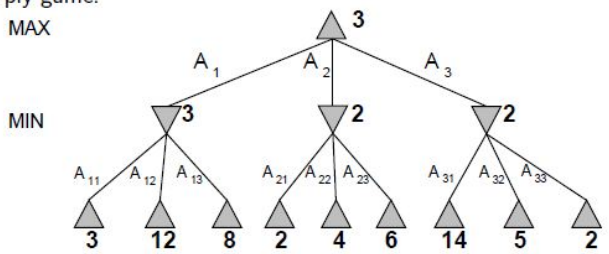


Minimax

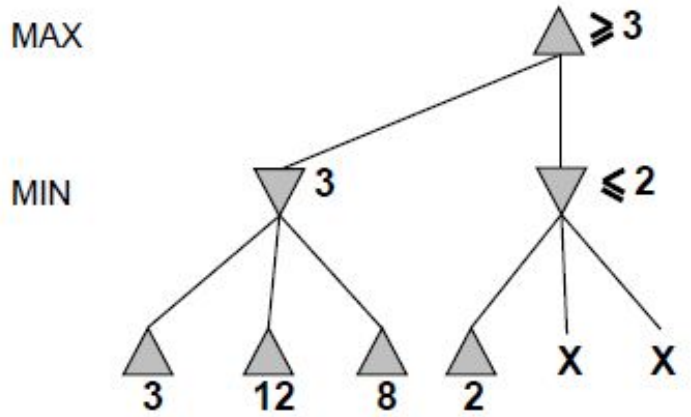
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



α–β pruning example

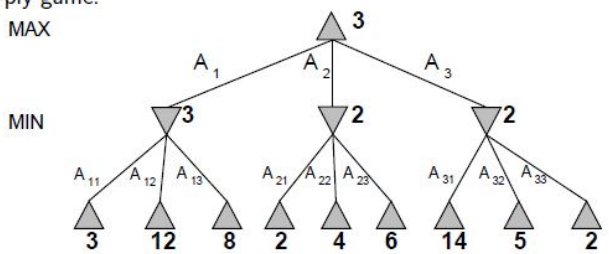


Minimax

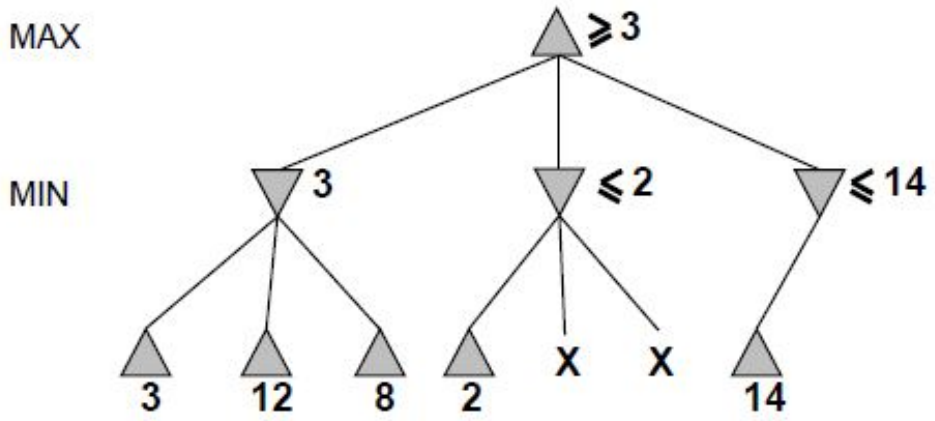
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



α–β pruning example

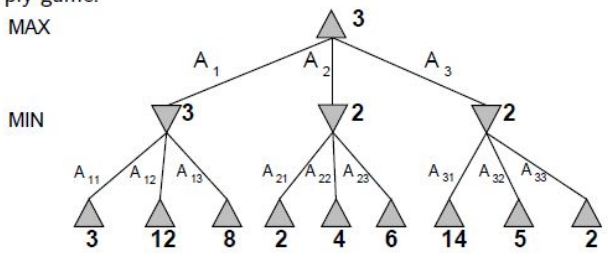


Minimax

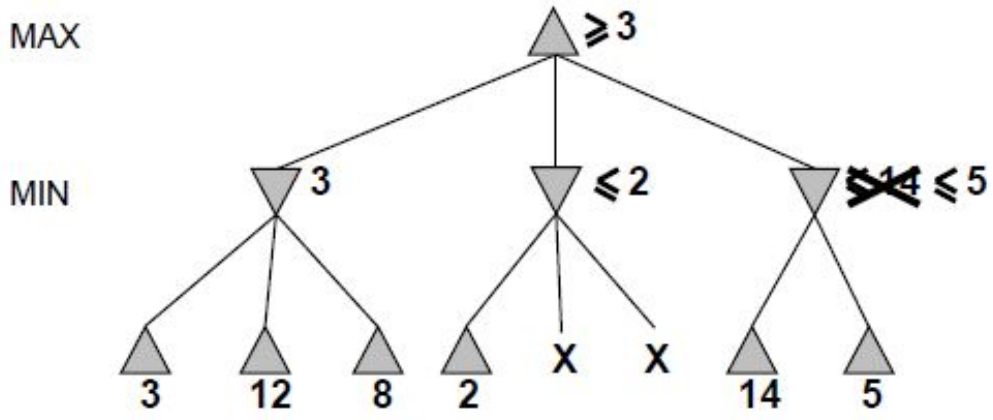
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



α–β pruning example

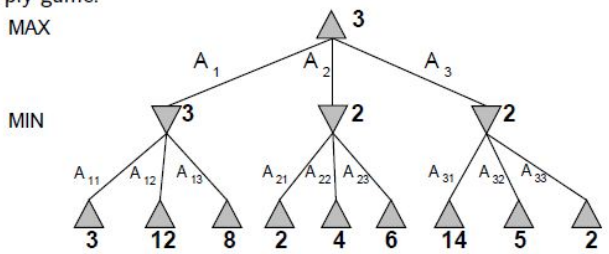


Minimax

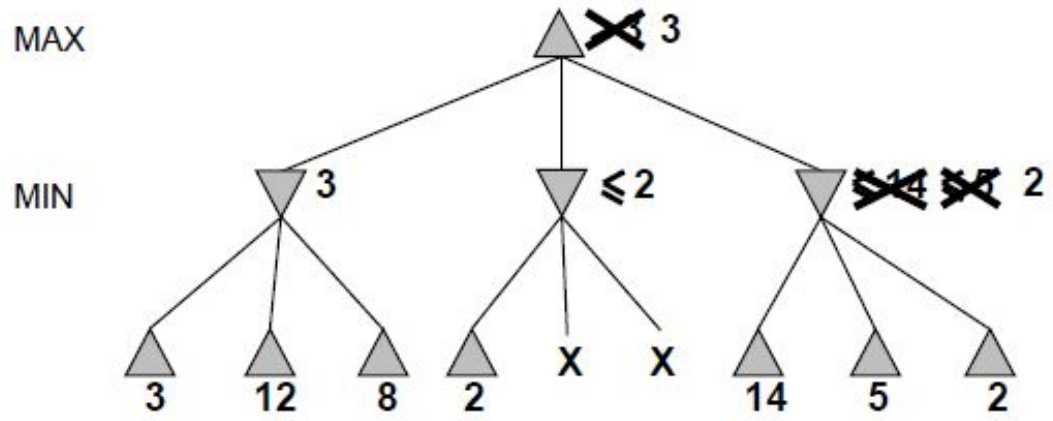
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



α–β pruning example



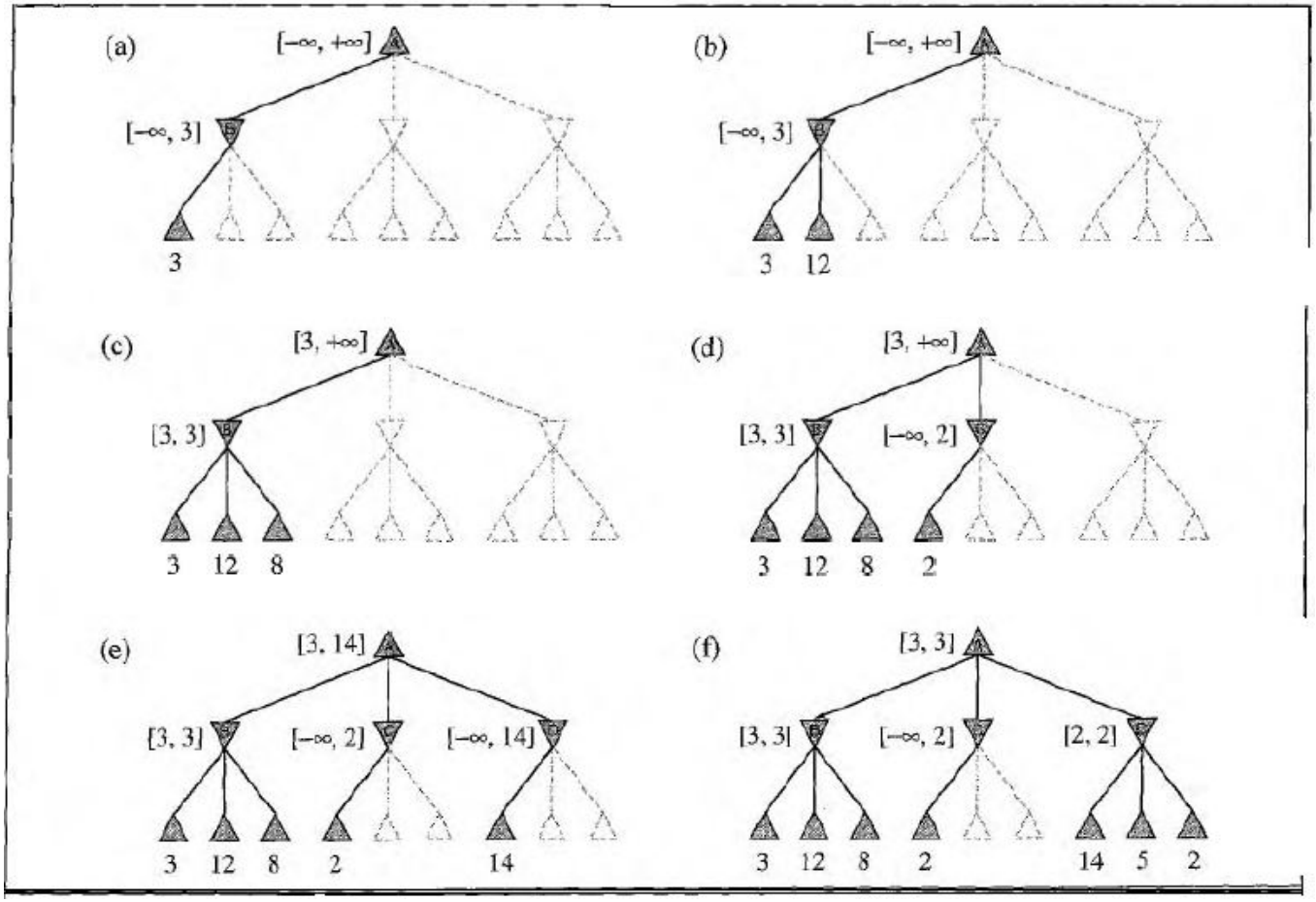
Alpha-Beta Pruning

- When we apply **alpha-beta pruning** to a **standard minimax tree**, it returns the **same move** as **minimax** would, but **prunes** away **branches** that **cannot** possibly **influence** the **final decision**.

Example

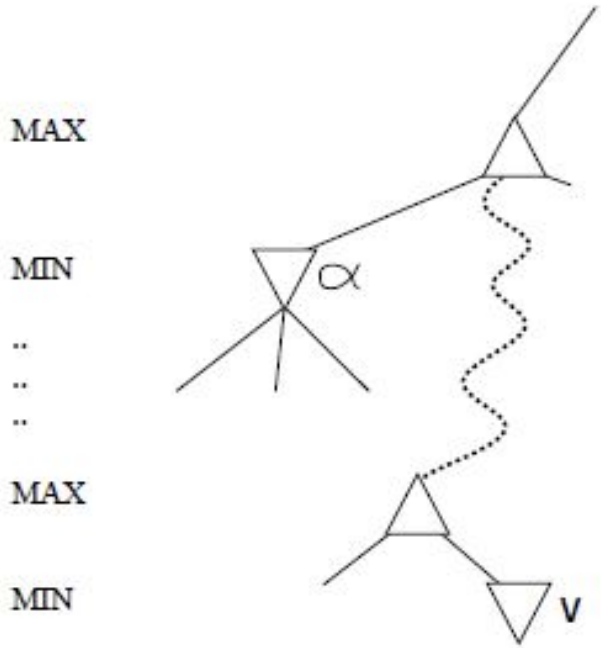
MINIMAX(root) = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))
= max(3, min(2, x, y), 2)
= max(3, z, 2) where z = min(2, x, y) $\leq$ 2
= 3

Alpha-Beta Pruning



- [at least, at most]

Why is it called α–β?



α is the best value (to MAX) found so far off the current path

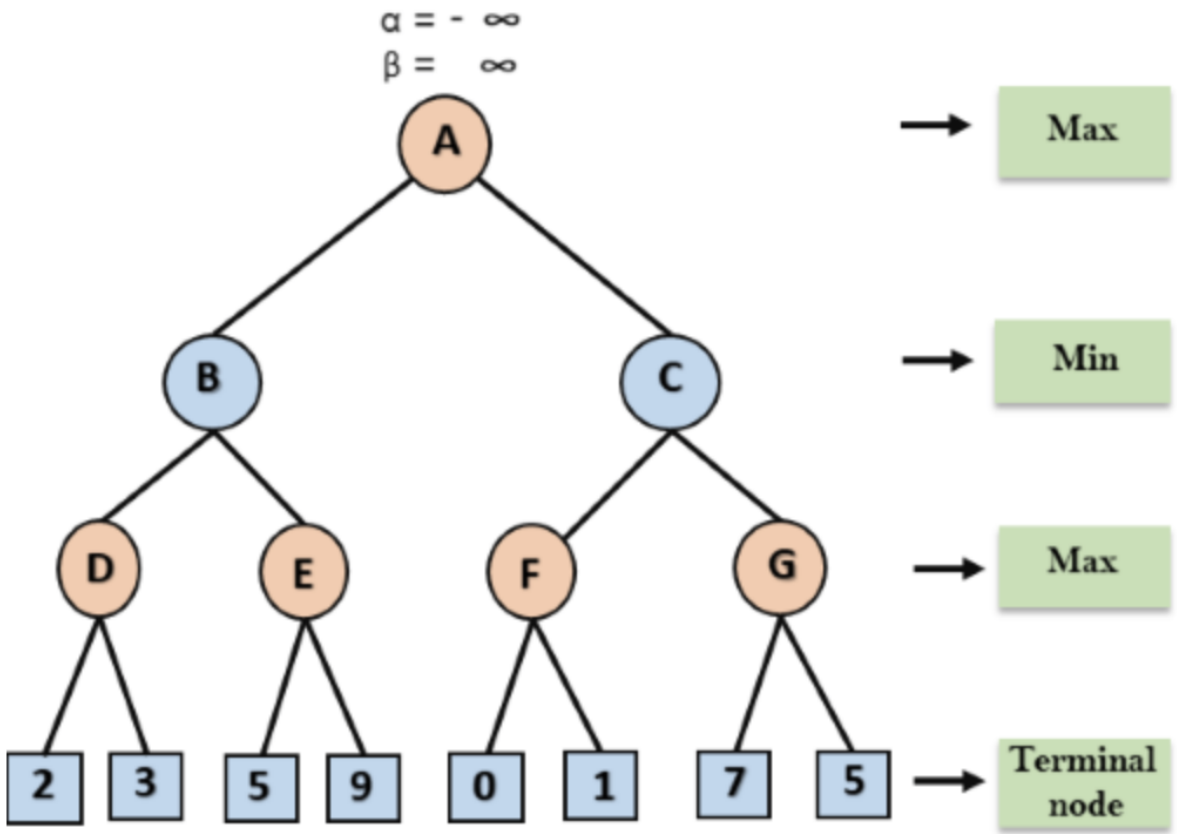
If V is worse than α, MAX will avoid it $\Rightarrow$ prune that branch

Define β similarly for MIN

Pseudo code for Alpha-Beta Pruning

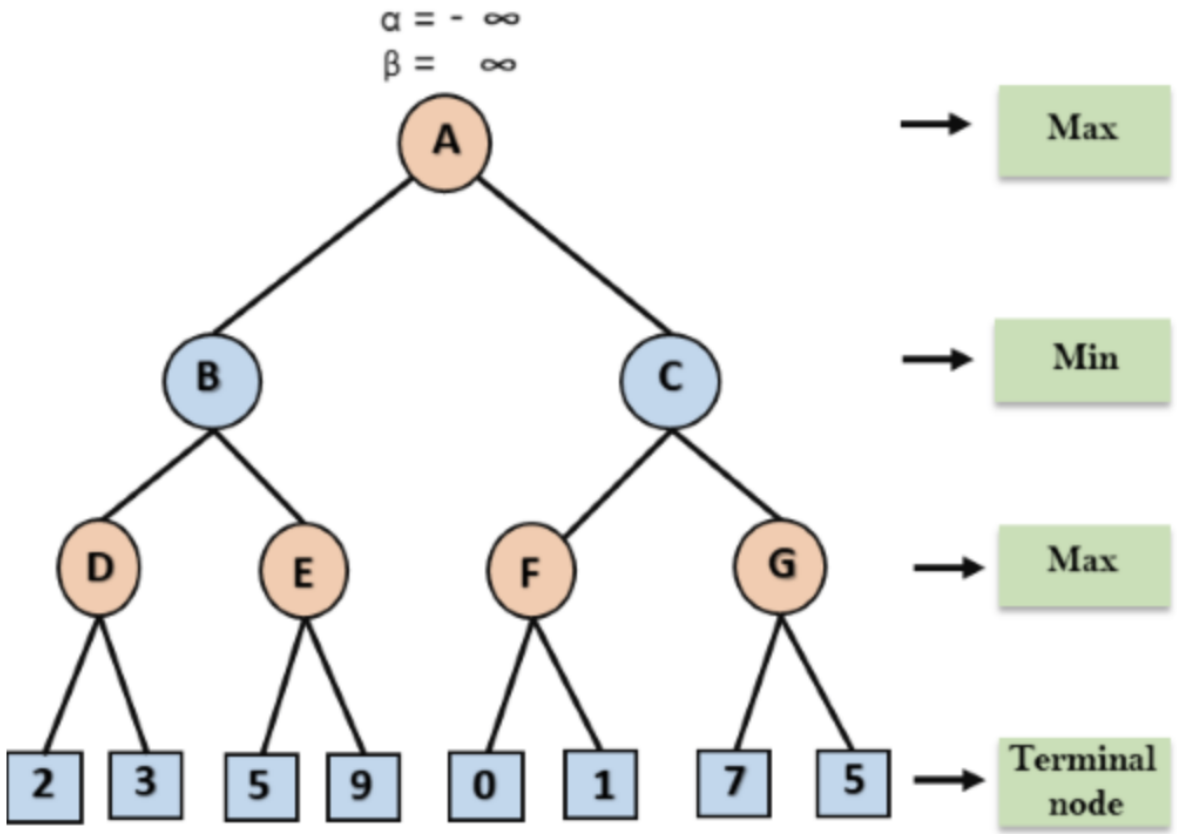
```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
  maxEva= -infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, False)
 maxEva= max(maxEva, eva)
 alpha= max(alpha, maxEva)
  if beta<=alpha
 break
 return maxEva

else                    // for Minimizer player
  minEva= +infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, true)
  minEva= min(minEva, eva)
  beta= min(beta, eva)
   if beta<=alpha
 break
 return minEva
```

Alpha-Beta Pruning



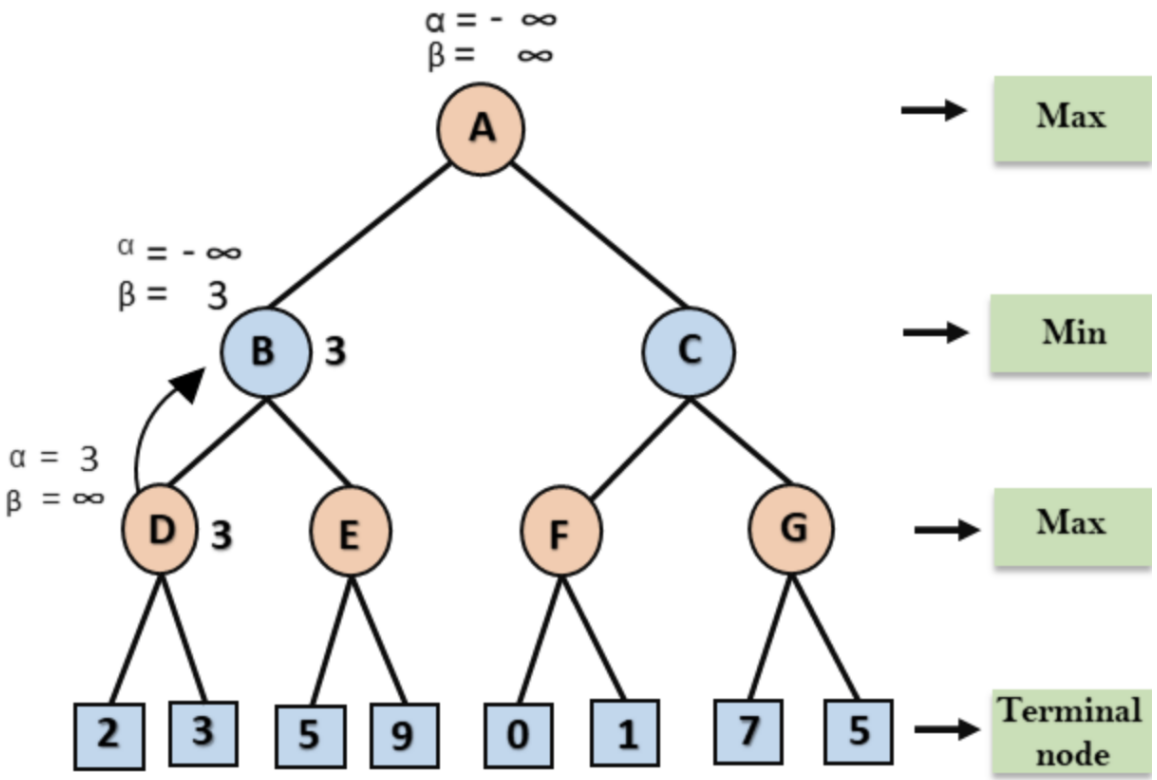
- At the first step the Max player will start first move from node A where $\alpha = -\infty$ and $\beta = +\infty$, these value of α and β are passed down to node B where again $\alpha = -\infty$ and $\beta = +\infty$, and Node B passes the same value to its child D.

Alpha-Beta Pruning



- Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) $= 3$ will be the value of α at node D and node value will be 3.

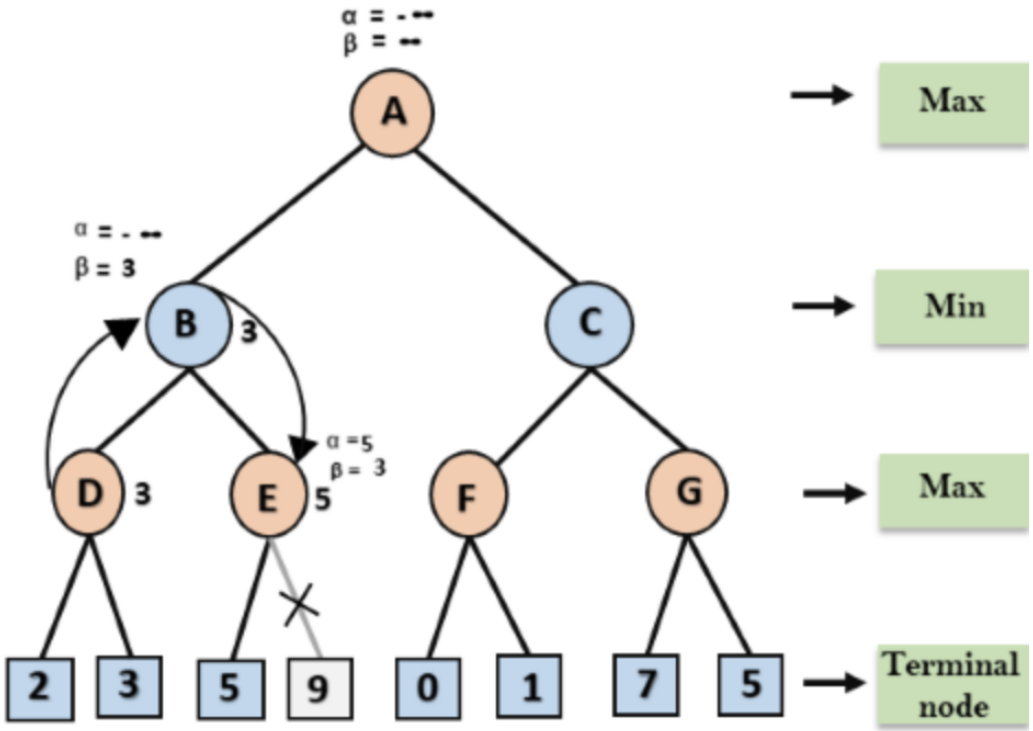
- Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now $\beta = +?$, will compare with the available subsequent nodes value, i.e. min $(\infty, 3) = 3$, hence at node B now $\alpha = -\infty$, and $\beta = 3$.



- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of $\alpha = -\infty$, and $\beta = 3$.

Alpha-Beta Pruning

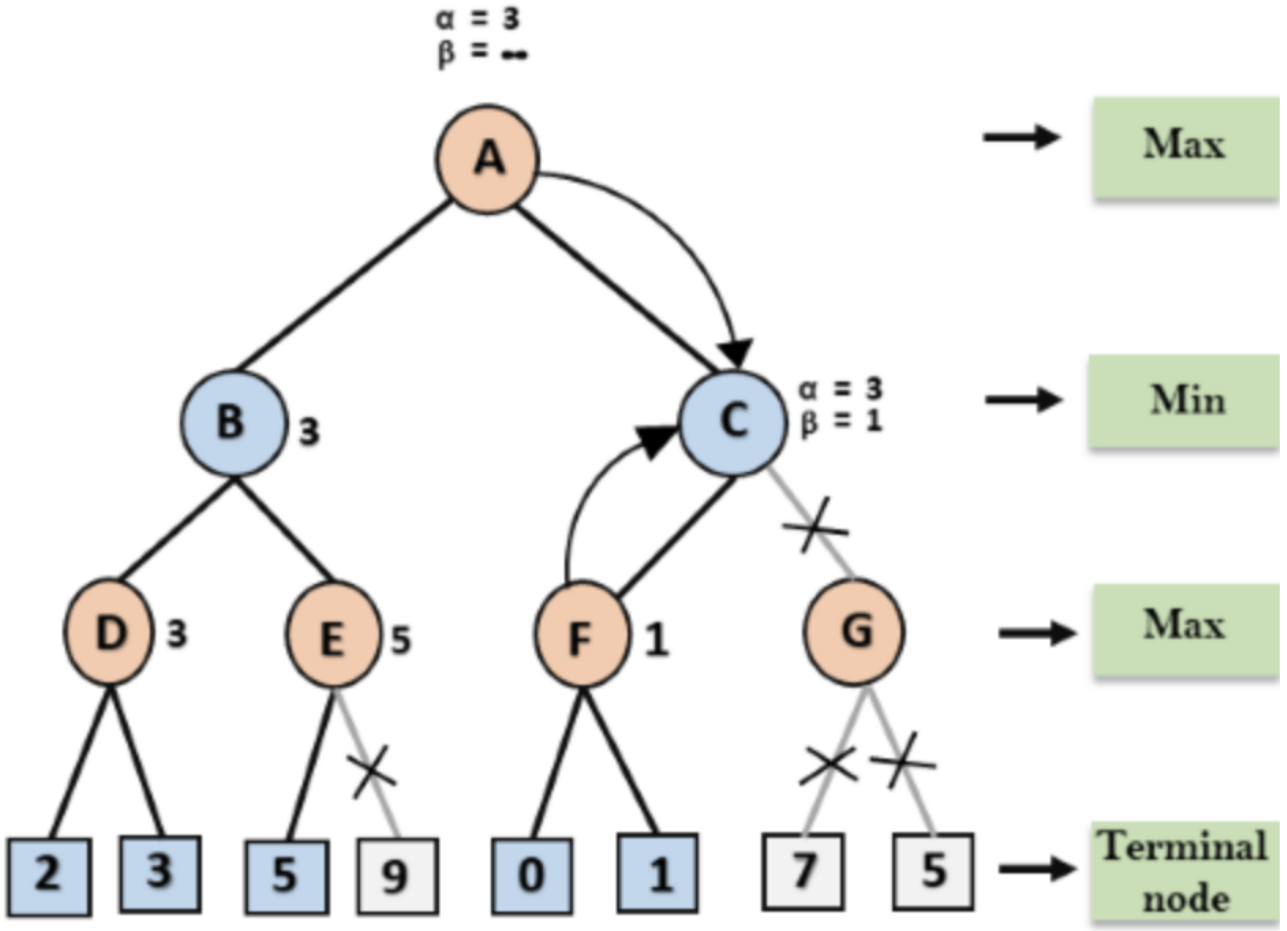
- Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of α will be compared with 5, so max (-∞, 5) $=$ 5, hence at node E $\alpha =$ 5 and $\beta =$ 3, where $\alpha >= \beta$, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



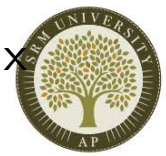
Alpha-Beta Pruning

- Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of α will be changed the maximum available value is 3 as max (-∞, 3)= 3, and $\beta = +\infty$, these two values now passes to right successor of A which is Node C.
- At node C, $\alpha = 3$ and $\beta = +\infty$, and the same values will be passed on to node F.
- Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still ? remains 3, but the node value of F will become 1.
- Step 7: Node F returns the node value 1 to node C, at C α= 3 and β = $+\infty$, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition $\alpha >= \beta$, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.

Alpha-Beta Pruning

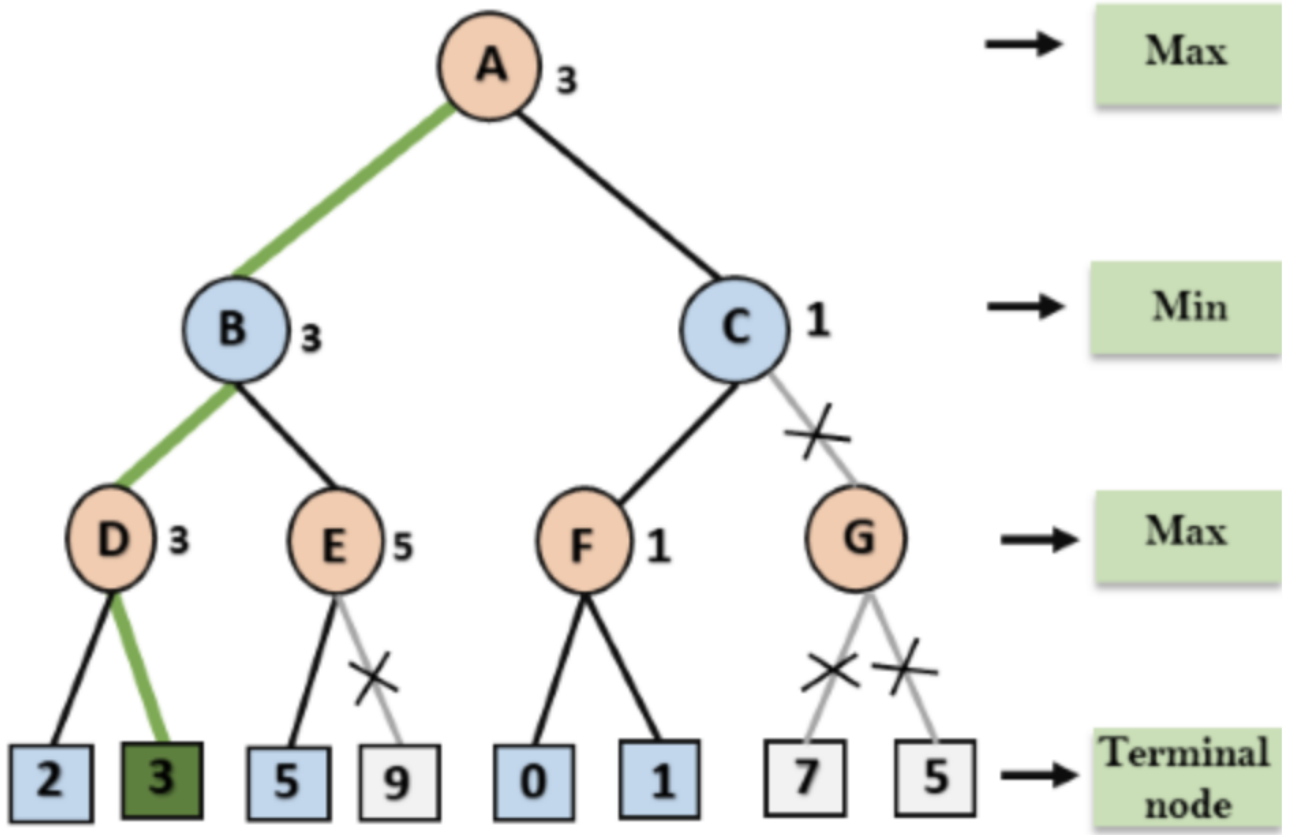


- Step 8: C now returns the value of 1 to A here the best value for A is max $(3, 1) = 3$.



Alpha-Beta Pruning

- Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Alpha-Beta Pruning: Move ordering

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning. It can be of two types:

- Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.
- Alpha-beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.
- This means that the effective branching factor becomes $\sqrt{b}$ instead of b-for chess, about 6 instead of 35.

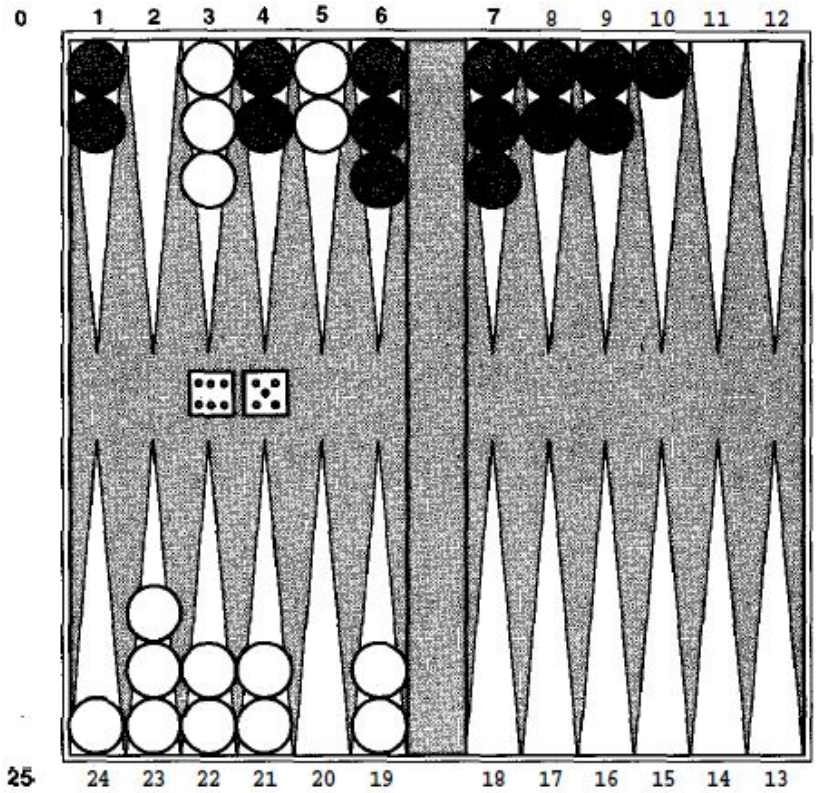
Alpha-Beta Pruning: Rules to find good ordering

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

Games that Include an Element of Chance

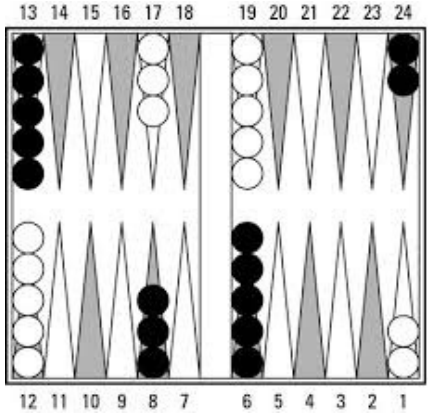
- **Backgammon** is a typical game that combines **luck** and **skill**.
- Two players, 24 narrow triangles, 15 checkers (white/black) per player
- $0 \sim 6$: Black's home board, $7 \sim 12$: outer board
- $19 \sim 25$: White's home board, $13 \sim 18$: outer board



Games that Include an Element of Chance

- The aim of the game is to move all one's pieces off the board.
- **White moves** clockwise toward **25**, and **black moves** counterclockwise toward **0**.
- A piece can **move to any position except one** where **there are two or more of the opponent's pieces**.
- If it **moves to a position with one opponent piece**, that piece is captured and has to **start its journey again from the beginning**.
- In this position, **white has just rolled 6-5** and has four legal moves: (5-10,5-11), (5-11,19-24), (5-10,10-16), and (5-11,11-16).

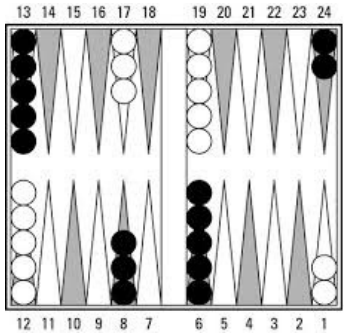
Games that Include an Element of Chance



Initial arrangement of checkers

- **1st player:** 2 on first, 5 on twelve and ninteen, and 3 on seventeen point.
- **2nd player:** 2 on twenty-fourth point, 5 on thirteen and sixth point, and 3 on eigth point.
- Both players have their own pair of dice and a dice cup used for shaking.
- To start the game, each player throws a single die. If equal numbers come up then both players roll again until they roll different numbers.
- The player throwing the higher number now moves his checkers according to the numbers showing on both dice.
- After the first roll, the players throw two dice and alternate turns.

Games that Include an Element of Chance



Players begin with two checkers on their 24-point, three checkers on their 8-point, and five checkers each on their 13-point and their 6-point.

Rules

- A checker may be moved only to an open point one, i.e., not occupied by 2 or more opposing checkers.
- The numbers on the 2 dice constitute separate moves. For example, if a player rolls 5 and 3, he may move one checker 5 spaces to an open point and another checker 3 spaces to an open point or he may move the one checker a total of 8 spaces to an open point, but only if the intermediate point (either 3 or 5 spaces from the starting point) is also open.

Means-end Analysis

Means-end Analysis

1. Comparing the current state S_i to a goal state S_g and computing the difference D_{ig}
2. An operator O_k is then selected to reduce the difference D_{ig}.
3. The operator O_k is applied if possible. If not, the current state is saved, a subgoal is created and means-end analysis is applied recursively to reduce the subgoal.
4. If the subgoal is solved, the saved state is restored and work is resumed on the original problem.

Example

Let the initial PL object $L_i = R \ \& \ (\sim P \rightarrow Q)$ and goal object $L_g = (Q \ \mathrm{V} \ P) \ \& \ R$

$L_i^{'} = (\sim P \rightarrow Q) \ \& \ R = (\sim\sim P \ \mathrm{V} \ Q) \ \& \ R = (P \ \mathrm{V} \ Q) \ \& \ R$

$L_g = (Q \ \mathrm{V} \ P) \ \& \ R$

NOTE: A comparison of the difference that R is on the left in L_i but on the right in L_g. This causes a subgoal to be set up to reduce this difference.