AI SEARCH TECHNIQUES

PREPARED BY

-ANOOJA JOY

PROBLEM SOLVING

- Problem solving is a process of generating solutions from observed data. It is selection of r**elevant set of states** to consider and **feasible set of operators** for moving from one state to another state.
- A ‘problem’ is characterized by a set of **goals**, a set of objects, and a s**et of operations.** Goals help in organizing behavior of AI agent by limiting the objectives that the agent is trying to achieve
- **Goal formulation** is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider and follows goal formulation.
- **A ‘problem space’** is an abstract space. A solution is a combination of operations and objects that achieve the goals.
- A **‘search’** refers to the search for a solution in a problem space. The process of genearting sequence of states by applying operators to initial state and checking which sequence has reached goal state.
- Search proceeds with different types of ‘search control strategies’. A search algorithm takes a problem as input and returns a solution in the form of an action sequence.

PROBLEM SOLVING AGENTS

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

SEARCH ALGORITHMS

WHY SEARCH IS IMPORTANT?

Problem-solving agents are the goal-based agents and use atomic representation. Agent will have set of actions to achieve goals.

Rational agents or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result.

Search techniques are universal problem-solving methods and is a key computational mechanism in many AI agents

Problem: Find a path from START to GOAL and find the minimum number of transitions

A search problem finds **sequence of actions** which transforms agent from **initial state to goal state**

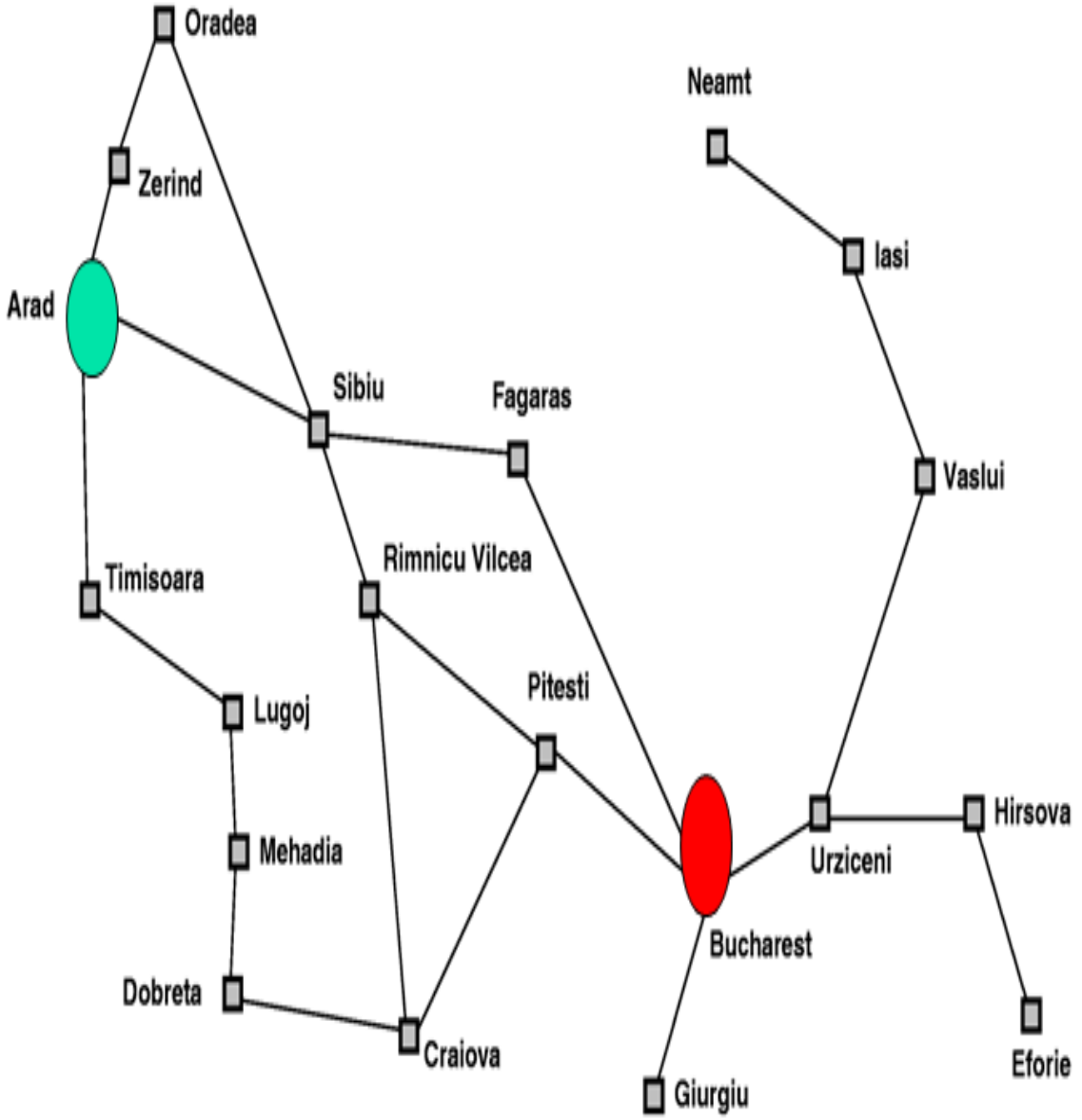
A search problem is represented using directed graph where **edges are the operators** and **nodes are states.** Searching process checks **allowable operation from one state to other states.**

ELEMENTS OF PROBLEM SOLVING

- **Problems** can be represented as **set of states** and how **state changes happens by application of some rules**.
- A problem space **encompasses all valid states that can be generated by the application of any combination of operators on any combination of objects**. The problem space may contain one or more solutions.
- **State space of a problem** is 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.
- **Problem State space representation have following things:**
  - **State(S)** is an abstract representation of agents environment.
  - **Initial State(S0):** description of starting configuration of agent.
  - **Action/Operator(A: S->Set of actions):** takes agent from one state to another state.
  - **Plan:** sequence of actions
  - **Goal:** description of desirable set of states. Its often specified by **Goal test** which any goal must satisfy.
  - **Goal test:** A function that looks at the current state returns whether or not it is the goal state.
  - **Path Cost:** A path cost function is a function that assigns a cost to a path.
  - **Solution:** a path from initial state to goal state

ROUTE FINDING/ MAP SEARCHING

- Imagine an agent on touring holiday in the city of Arad in Romania. The agent has a ticket to fly to Bucharest tomorrow so he can go to his home country from there. The ticket is nonrefundable, the agent's visa is about to expire after tomorrow, there are no seats available for six weeks. How to reach Bucharest?



STATE SPACE REPRESENTATION

- **State space:** Cities
- **Successor function:** Roads: Go to adjacent city with cost = distance

S(x) = set of action–state pairs

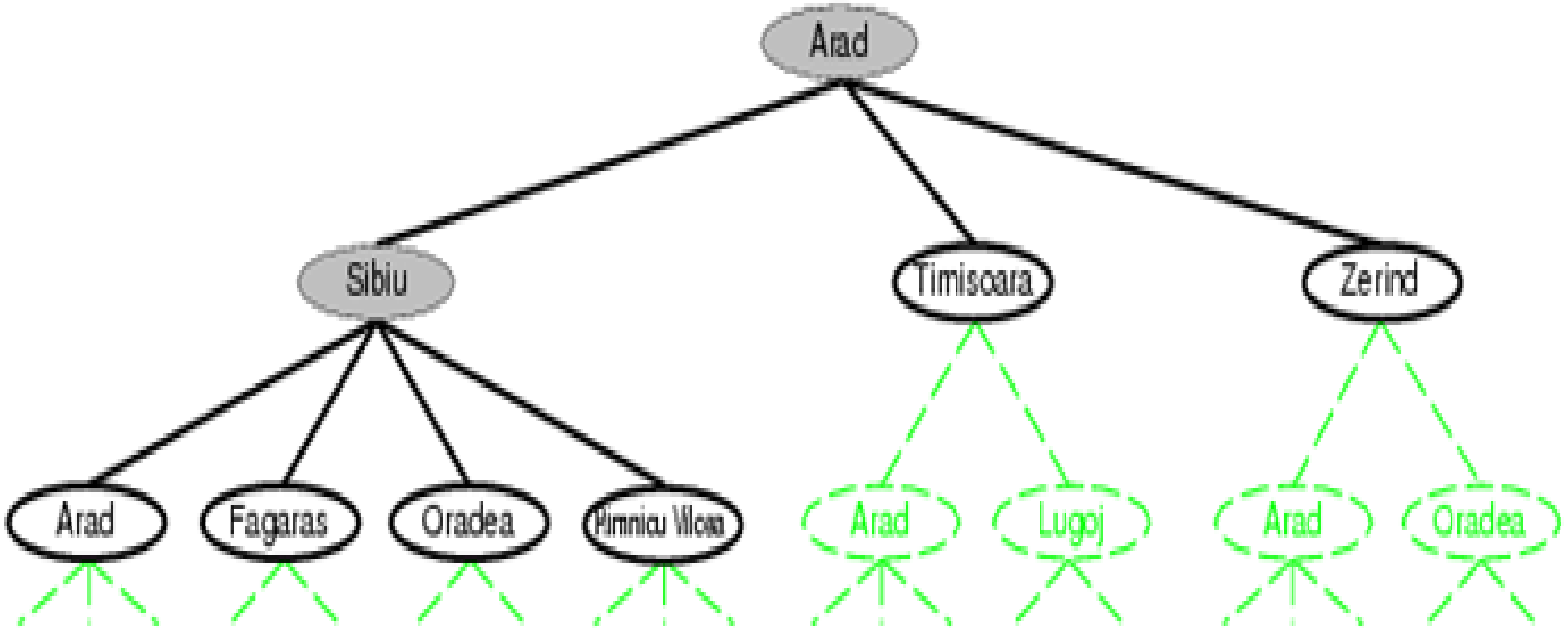
e.g., *S(Arad)* = {<*Arad à Zerind, Zerind*>, … }

- **Start state:** Arad
- **Goal test:** Is state == Bucharest?
- **Solution?**

ABSTRACTION/MODELING

- **State space** must be **abstracted** for problem solving
  - n(Abstract) state = set of real states
  - n(Abstract) action = complex combination of real actions
- ***Abstraction is the process of removing irrelevant detail to create an abstract representation: ``high-level", ignores irrelevant details***
- Abstraction is critical for automated problem solving as good abstractions retain all important details.
- Abstraction is to create an approximate, simplified, model of the world for the computer to deal with: real-world is too detailed to model exactly

STATE SPACE GARPH

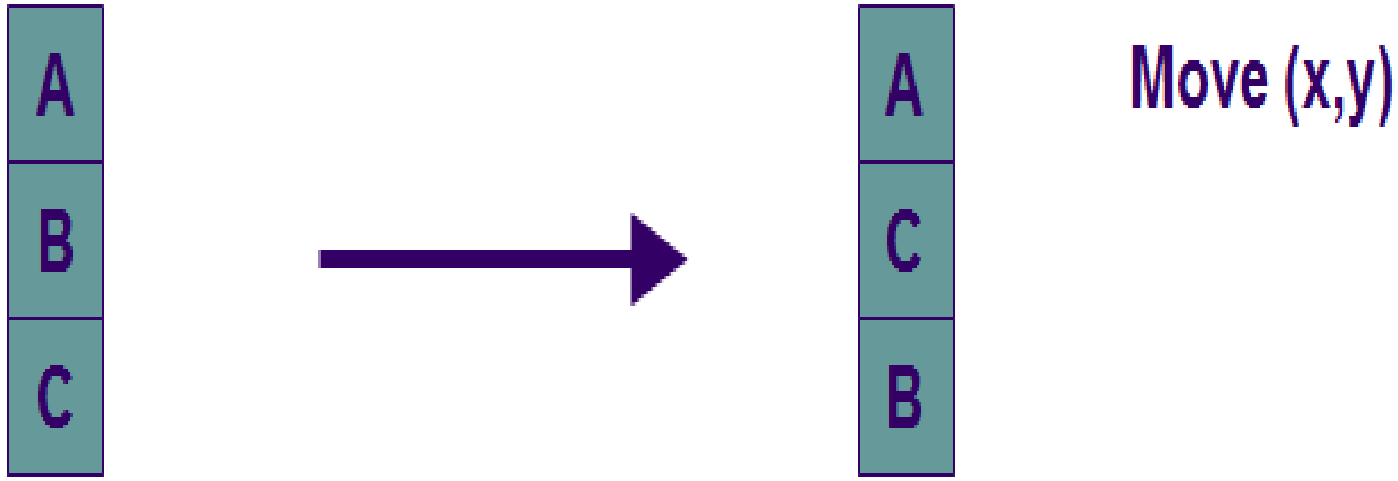


GENERAL SEARCH ALGORITHM

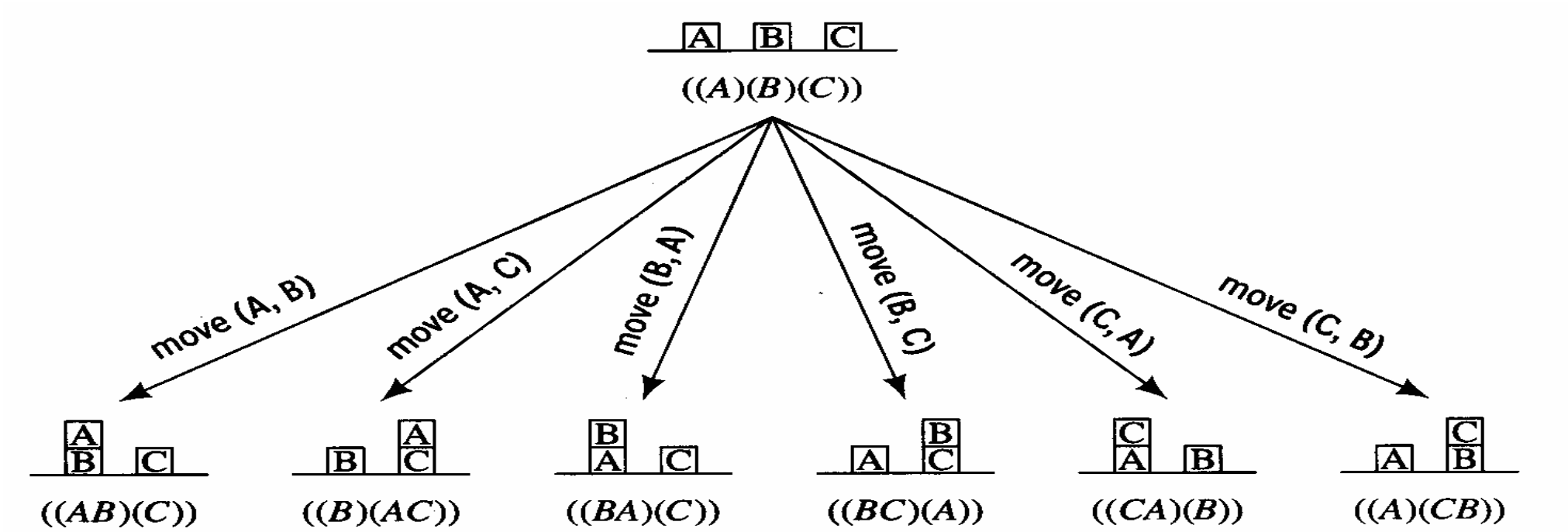
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

ROBOT BLOCK WORLD PROBLEM

- Given a set of blocks in a certain configuration,
- Move the blocks into a goal configuration.
- Example : (c,b,a) --> (b,c,a)



STATE SPACE GRAPH



Effects of Moving a Block

STATE SPACE GRAPHS

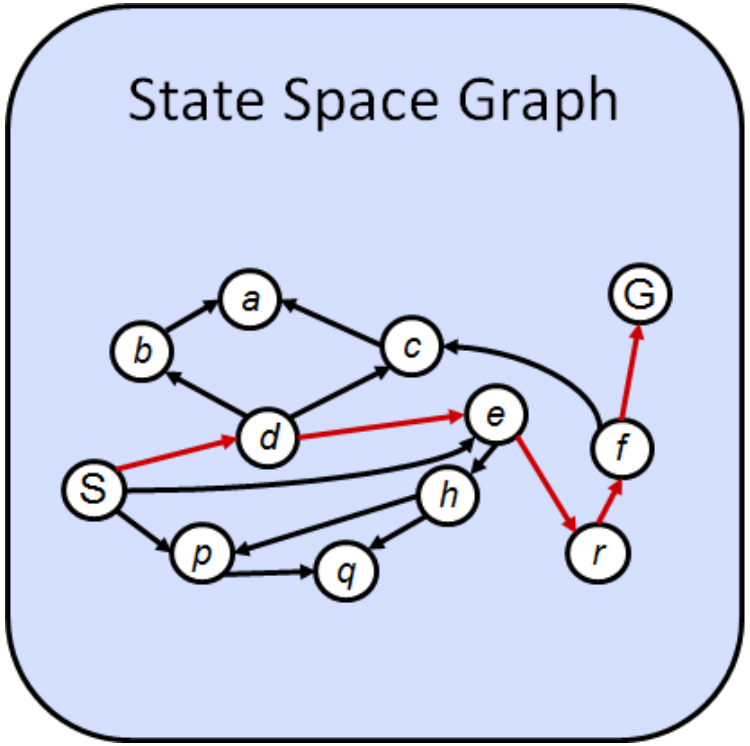
State space graph: A mathematical representation of a search problem

Nodes are (abstracted) world configurations	Arcs represent successors (action results)	Goal test is a set of goal nodes (maybe only one)

In a state space graph, each state occurs only once!

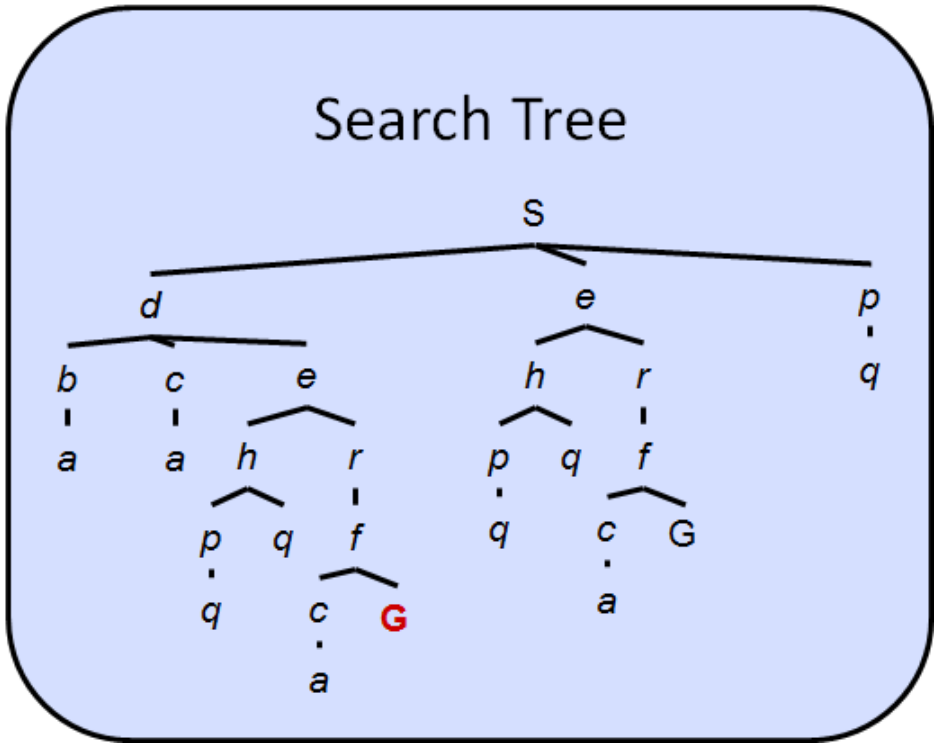
We can rarely build this full graph in memory (it's too big), but it's a useful idea

STATE SPACE GRAPHS VS. SEARCH TREES



Each NODE in in the search tree is an entire PATH in the state space graph.

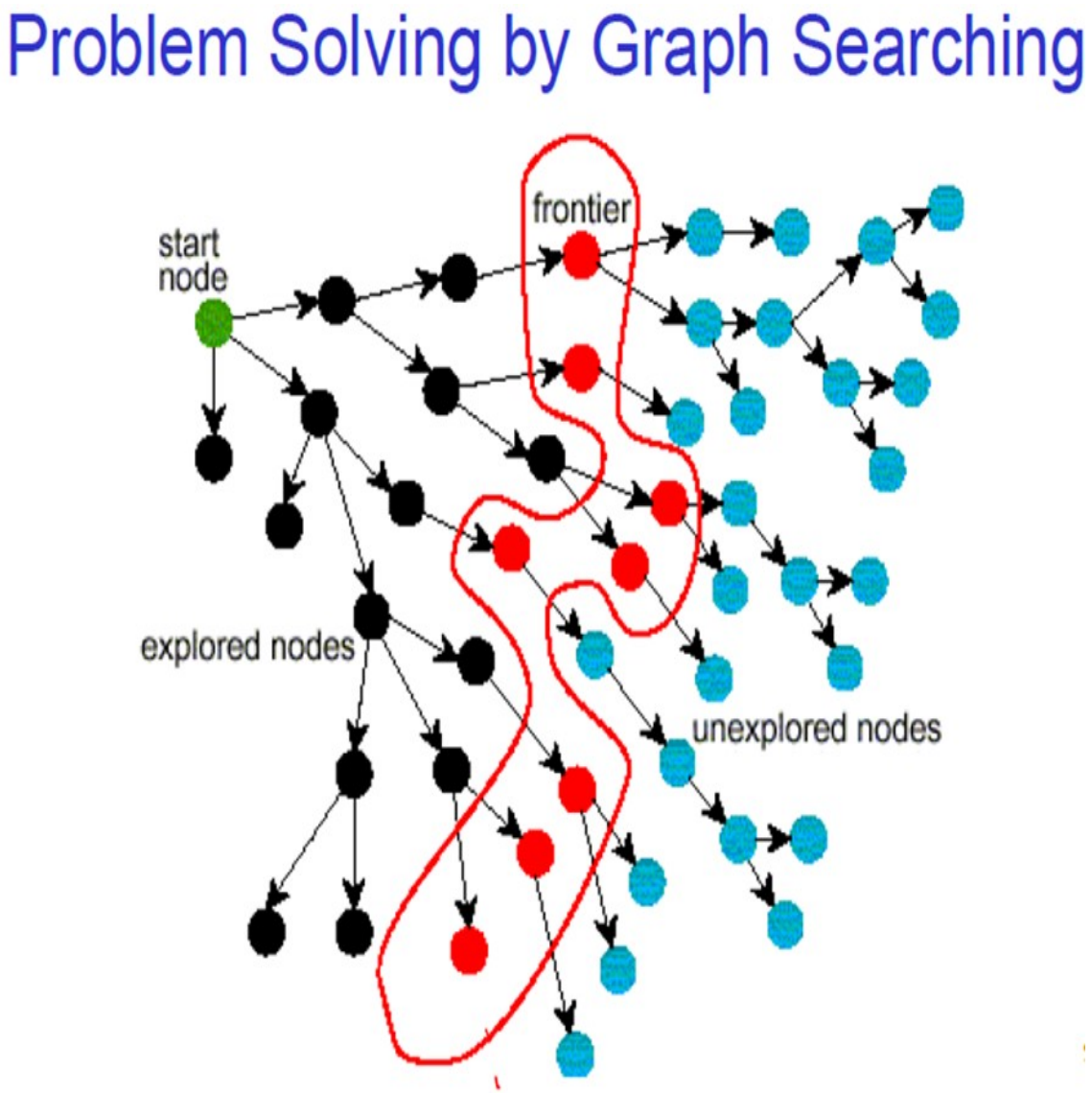
We construct both on demand – and we construct as little as possible.



Lots of repeated structure in the search tree!

CONSTITUENTS OF SEARCH ALGORITHMS

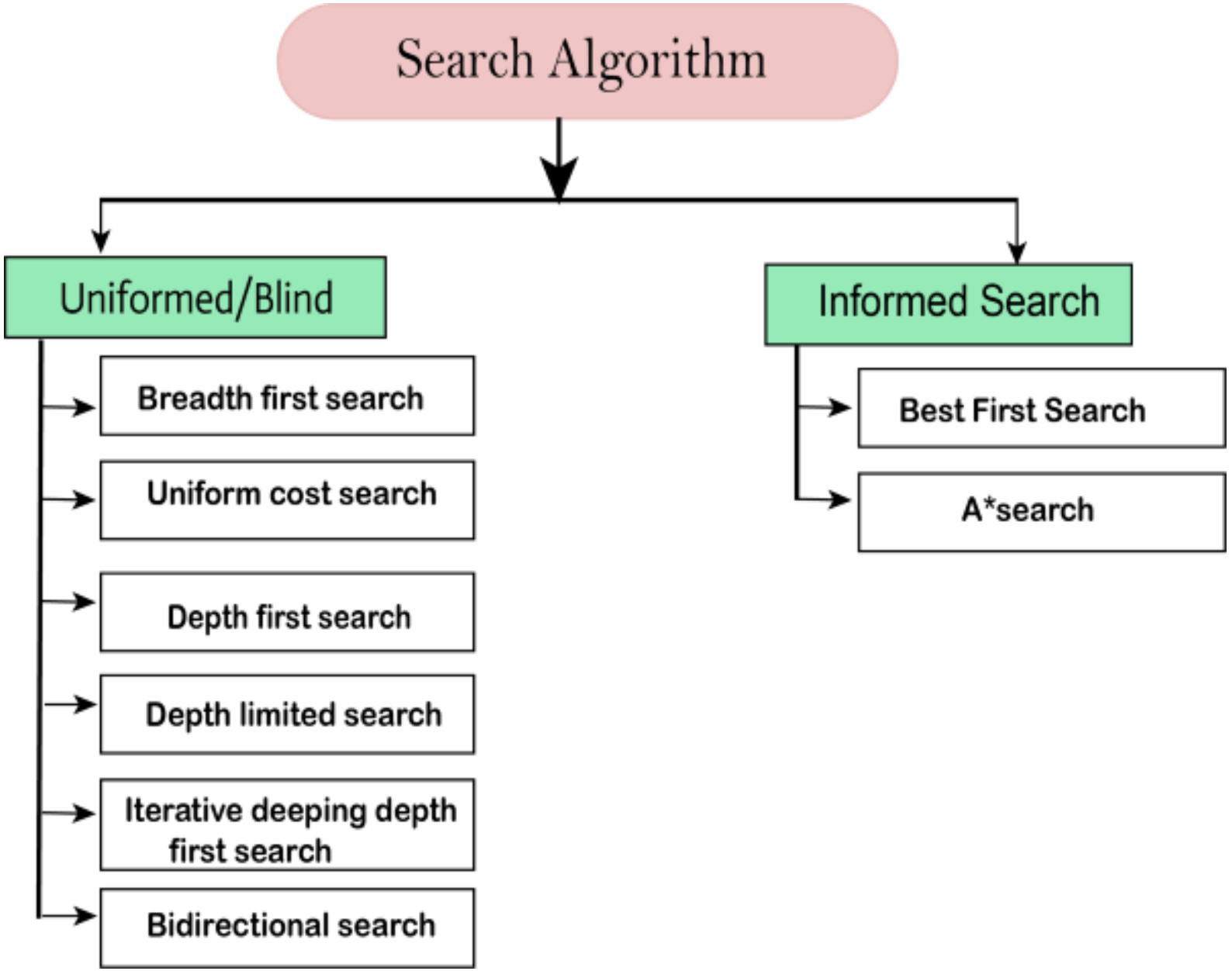
- A problem **graph,** containing the start node S and the goal node G.
- A **strategy,** describing the way the graph will be traversed to get to G .
- **Frontier**: - The set of paths which could be explored next. The way in which the frontier is expanded define the search strategy
- A **fringe,** which is a data structure used to store all the possible states (nodes) that you can go from the current states



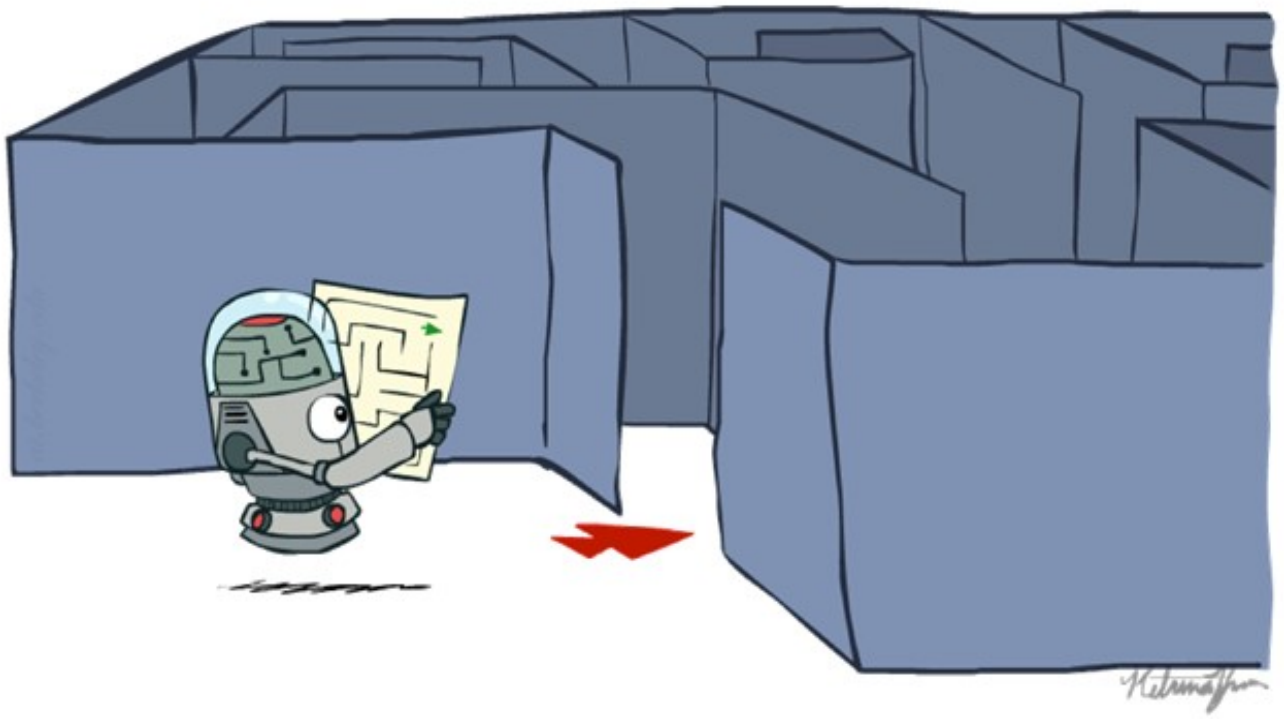
- The **total cost** of the search is the sum of the **path cost** and the **search cost**
- **Search cost** associated is the **time** and **memory** required to find a solution
  - **Time complexity**—number of nodes generated/expanded
  - **Space complexity—**maximum number of nodes in memory.
- Time and space complexity are measured in terms of
    - **b**—maximum **branching factor** of the search tree
    - **d**—**depth** of the **least-cost solution**
    - **m**—maximum depth of the state space (may be ∞)

MEASURING PROBLEM-SOLVING PERFORMANCE

TYPES OF SEARCH ALGORITHMS



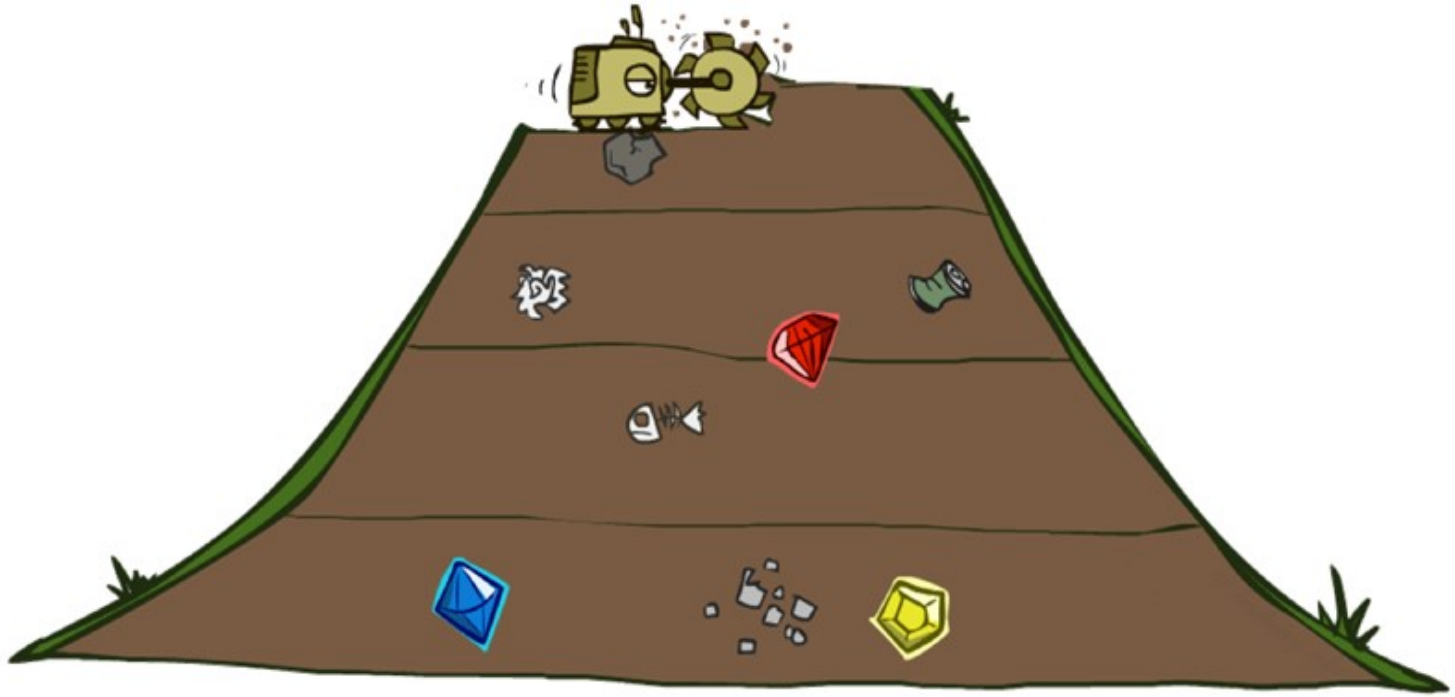
UNINFORMED SEARCH STRATEGIES



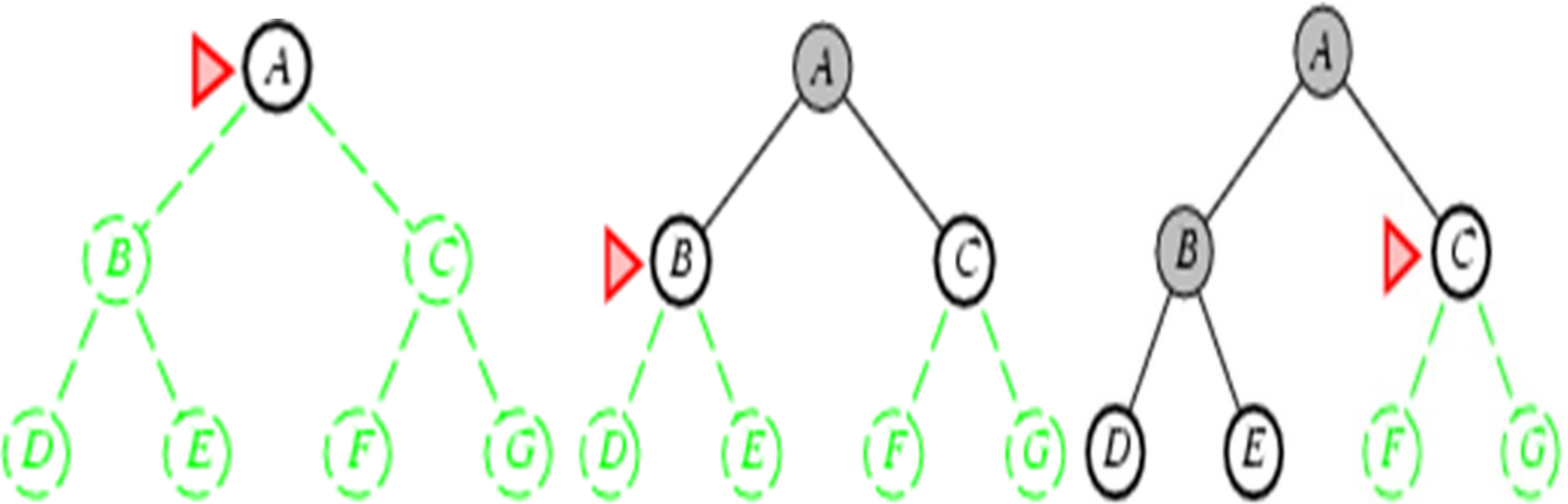
UNINFORMED SEARCH

- Uninformed search have no information about the number of steps or the path cost from the current state to the goal—all they can do is distinguish a goal state from a nongoal state.
- Uninformed search is also sometimes called blind search.
- The uninformed search **does not contain any domain knowledge** such as **closeness, the location of the goal ie** no additional information on the goal node other than the one provided in the problem definition.
- **Uninformed (blind) search strategies use only the information available in the problem definition.** All they can do is generate successors and differentiate between goal and non-goal states.
- Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goa**.**
- It operates in a **brute-force way** as it only includes information about **how to traverse the tree** and **how to identify leaf and goal nodes**.

BREADTH FIRST SEARCH



BREADTH FIRST SEARCH

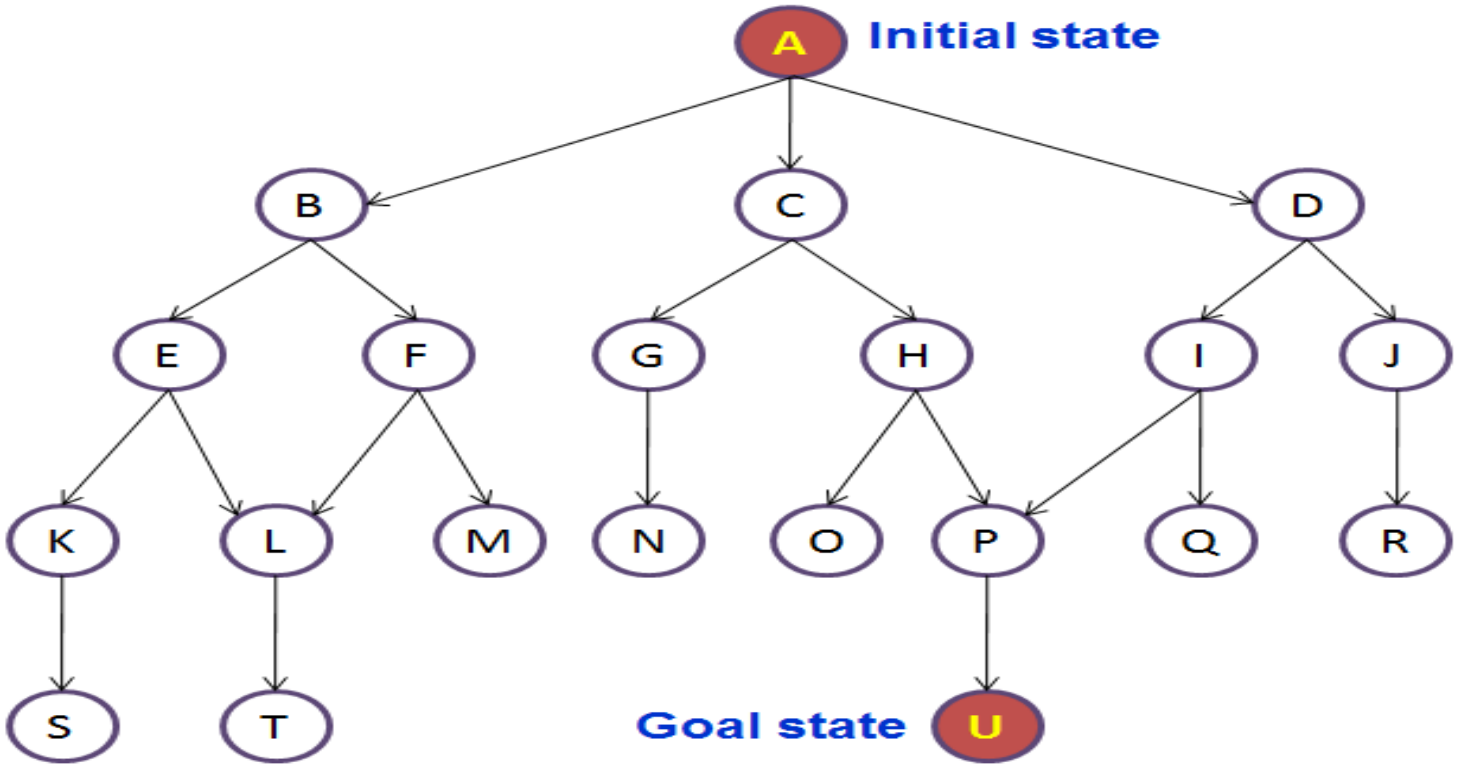


OPEN AND CLOSED LIST IN SEARCH STRATEGIES

- During search, **a node** can be in one of the **three categories**:
  1. **Not generated yet** (has not been made explicit yet)
  2. **OPEN**: generated but not expanded
  3. **CLOSED**: expanded
- **Search strategies differ** mainly on **how to select an OPEN node for expansion** at each step of search

B F S

Breadth First Search examines the nodes in the following order:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U

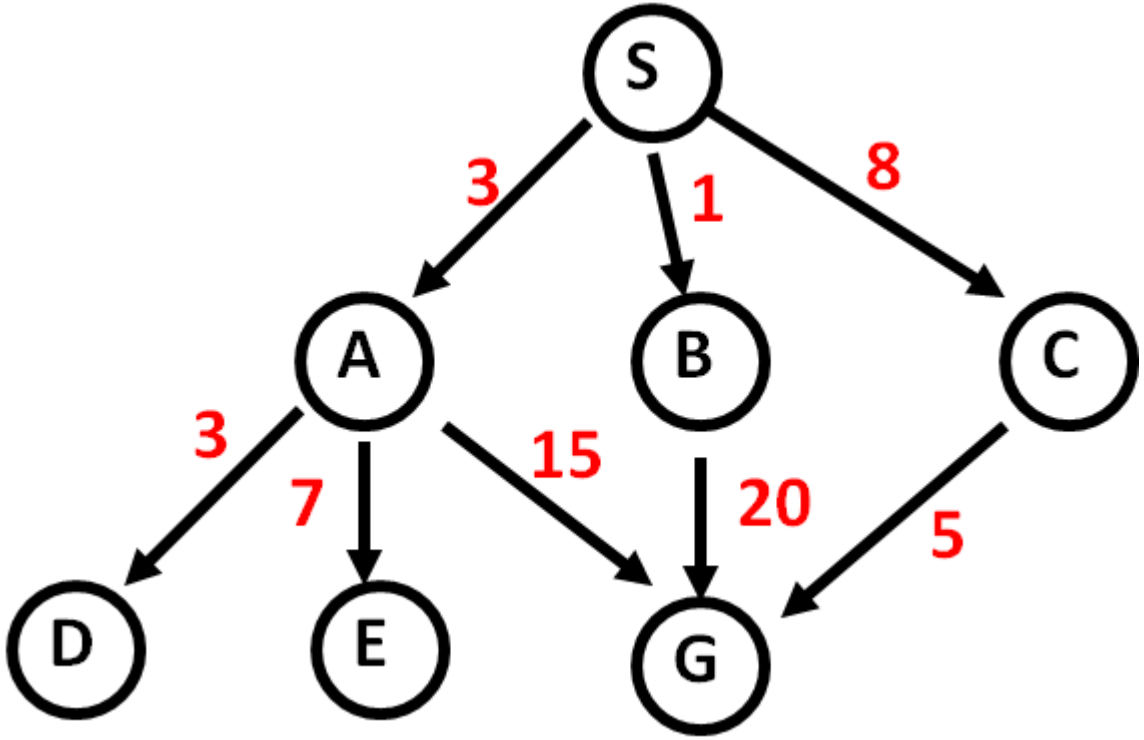


OPEN	CLOSE
A	
BCD	A
CDEF	BA
DEFGH	CBA
EFGHIJ	DCBA
FGHIJKL	EDCBA
GHIJKLM	FEDCBA
HIJKLMN	GFEDCBA
IJKLMNOP	HGFEDCBA
JKLMNOPQ	IHGFEDCBA
KLMNOPQR	JIHGFEDCBA
LMNOPQRS	KJIHGFEDCBA
MNOPQRST	LKJIHGFEDCBA
NOPQRST	MLKJIHGFEDCBA
OPQRST	NMLKJIHGFEDCBA
PQRST	ONMLKJIHGFEDCBA
QRSTU	PONMLKJIHGFEDCBA
RSTU	QPONMLKJIHGFEDCBA
STU	RQPONMLKJIHGFEDCBA
TU	SRQPONMLKJIHGFEDCBA
U	TSRQPONMLKJIHGFEDCBA
U	hold

BREADTH FIRST SEARCH

- **Root node** is **expanded first,** then **all the nodes generated by the root node** are expanded next, and then their successors, and so on.
- All the **nodes at depth d** in the search tree are expanded **before the nodes at depth d + 1.**
- Expand the **shallowest unexpanded node**
- **Implementation:** frontier/fringe is a **FIFO queue**, i.e., new successors go at end
- Problems involving search of **exponential complexity** (like chess game) **cannot be solved** by uninformed methods since the size of the data being too big.

BFS EXAMPLE



B F S

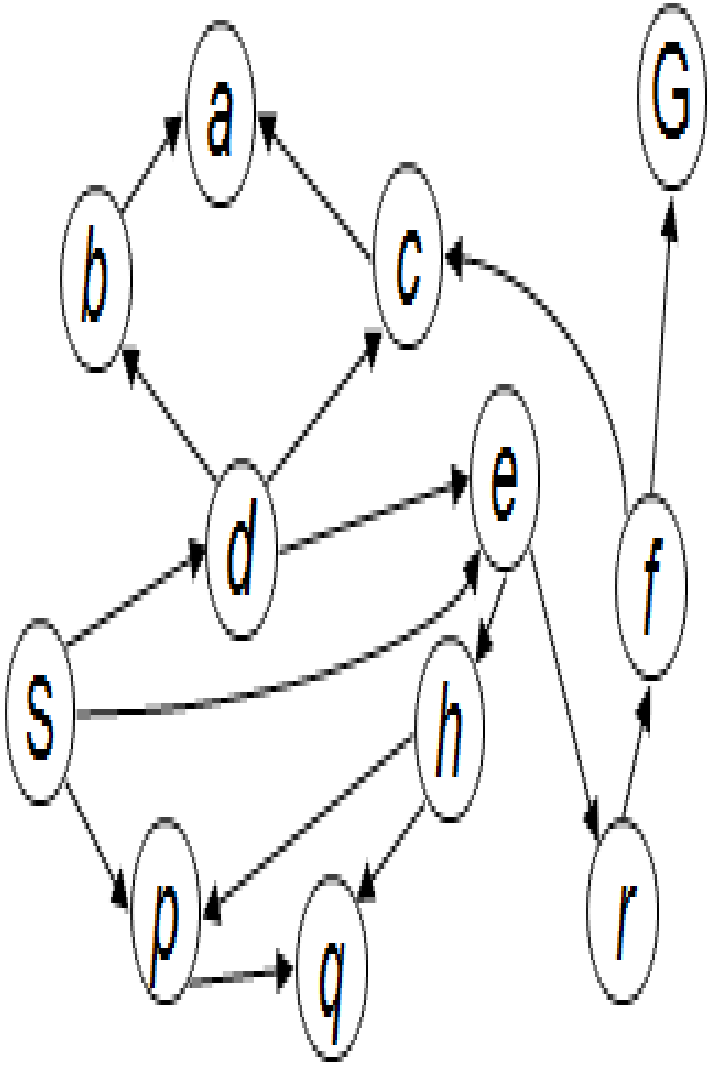
	Expanded node	Nodes list
		{ S^0 }
1	S^0	{ A^3 B^1 C^8 }
2	A^3	{ B^1 C^8 D^6 E^{10} G^{18} }
3	B^1	{ C^8 D^6 E^{10} G^{18} G^{21} }
4	C^8	{ D^6 E^{10} G^{18} G^{21} G^{13} }
5	D^6	{ E^{10} G^{18} G^{21} G^{13} }
6	E^{10}	{ G^{18} G^{21} G^{13} }
7	G^{18}	{ G^{21} G^{13} }

- Solution path found is S A G , cost 18
- Number of nodes expanded (including goal node) = 7

BFS

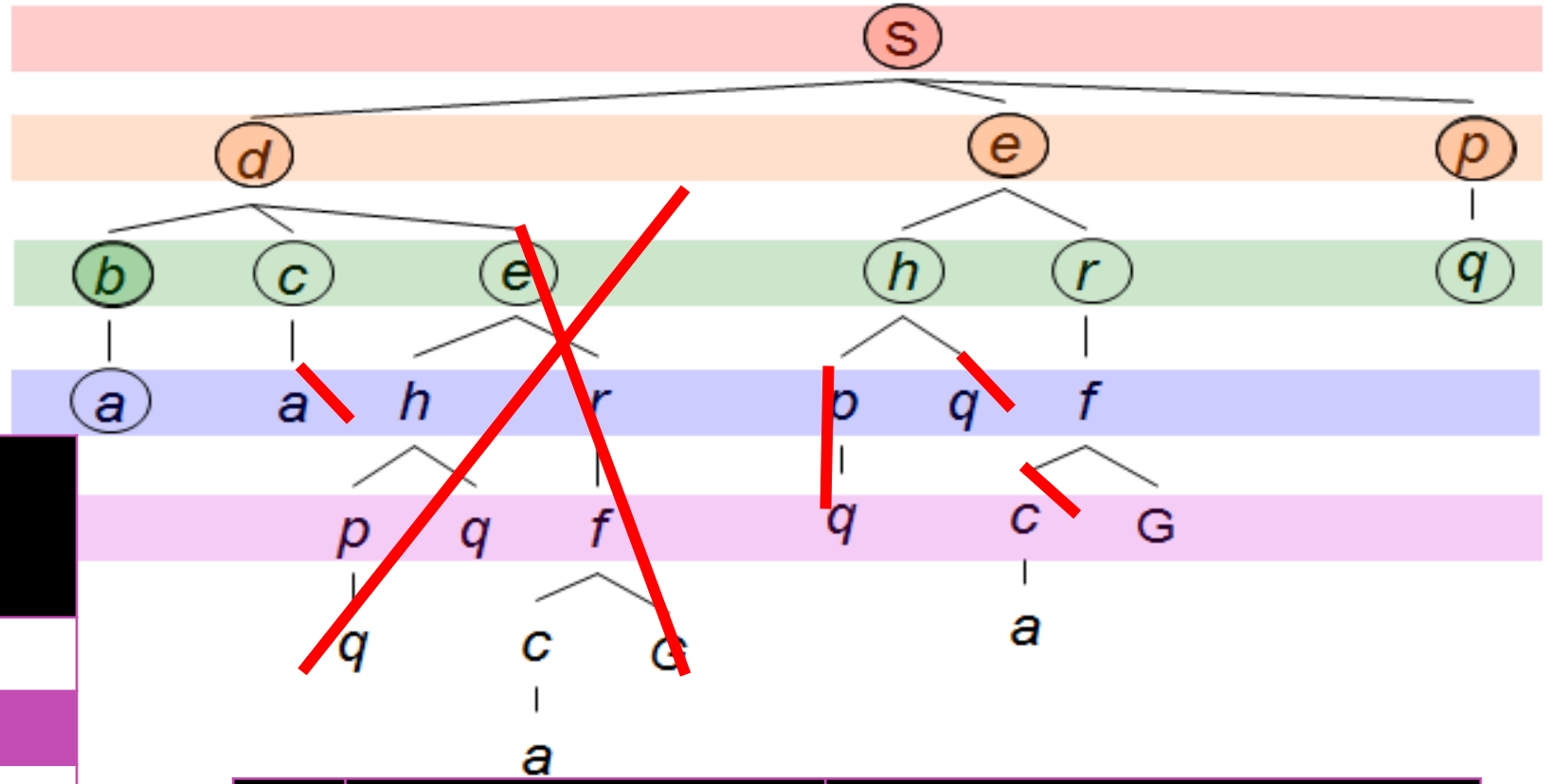
Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue



B F S

Search Tiers



	Expanded node	Nodes list/ open list
		{ S^0 }
1	s^0	{ d^1 e^1 p^1 }
2	d^1	{ e^1 p^1 b^2 c^2 e^2 }
3	e^1	{ p^1 b^2 c^2 e^2 h^3 r^3 }
4	p^1	{b^2 c^2 e^2 h^2 r^2 q^2}
5	b^2	{ c^2 e^2 h^2 r^2 q^2 a^3}
6	c^2	{e^2 h^2 r^2 q^2 a^3 a^3}
7	h^2	{ r^2 q^2 a^3 a^3 p^3 q^3}

	Expanded node	Nodes list/ open list
8	r^2	{ q^2 a^3 a^3 p^3 q^3 f^3 }
9	q^2	{a^3 a^3 p^3 q^3 f^3 }
10	a^3	{ a^3 p^3 q^3 f^3 }
11	f^3	{ $c^4$$g^4$ }
12	g^4	{ }

PROPERTIES OF BREADTH-FIRST SEARCH

Complete??

- **Yes** (If there is a solution that exist, breadth-first search is guaranteed to find it, and if there are several solutions, breadth-first search will always find the shallowest goal state first. It fails if graph or tree is cyclic and if child node have infinite successors.)

Space Complexity?

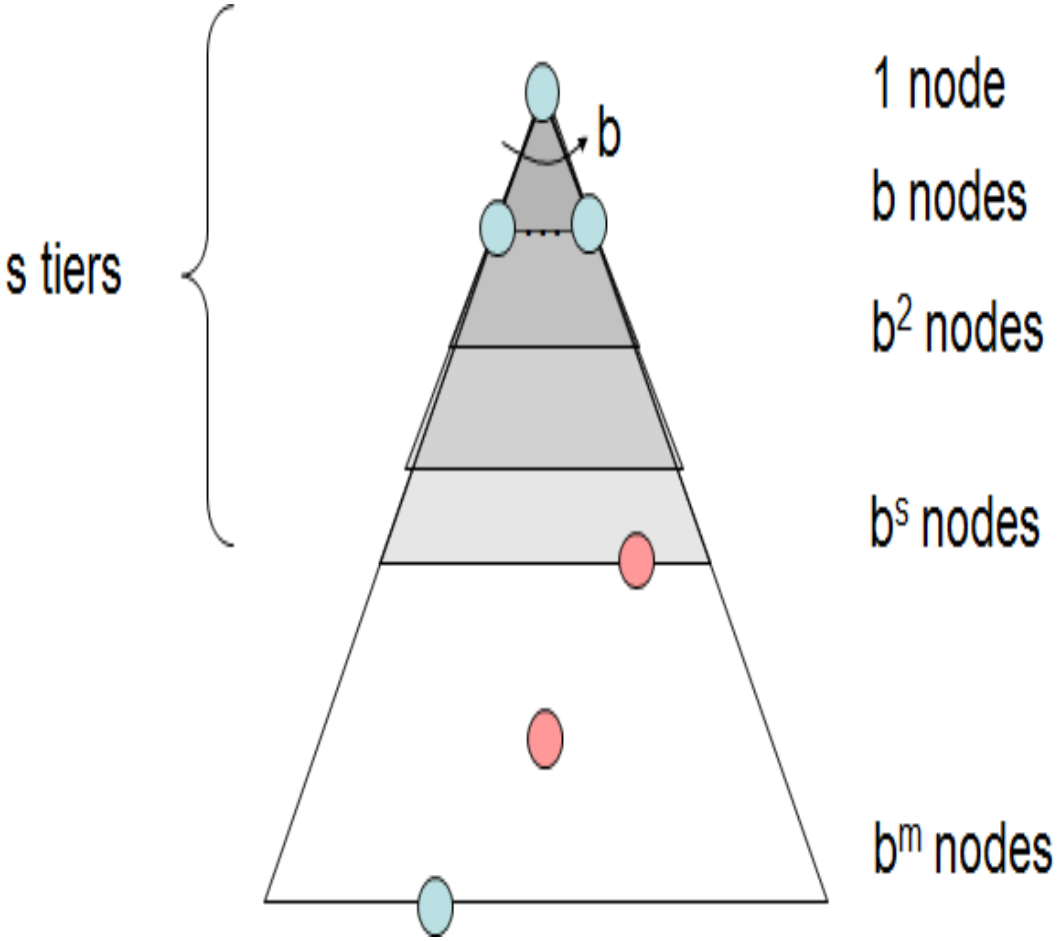
- Has roughly the last tier, so **$O(b^s)$.** Keeps every node in memory. s is the depth of the shallowest solution and b is the branching factor (i.e., number of children per node) at each node. Space is the big problem: it can easily generate 1M nodes/second so after 24hrs it has used 86,000GB (and then it has only reached depth 9 in the search tree)

Time??

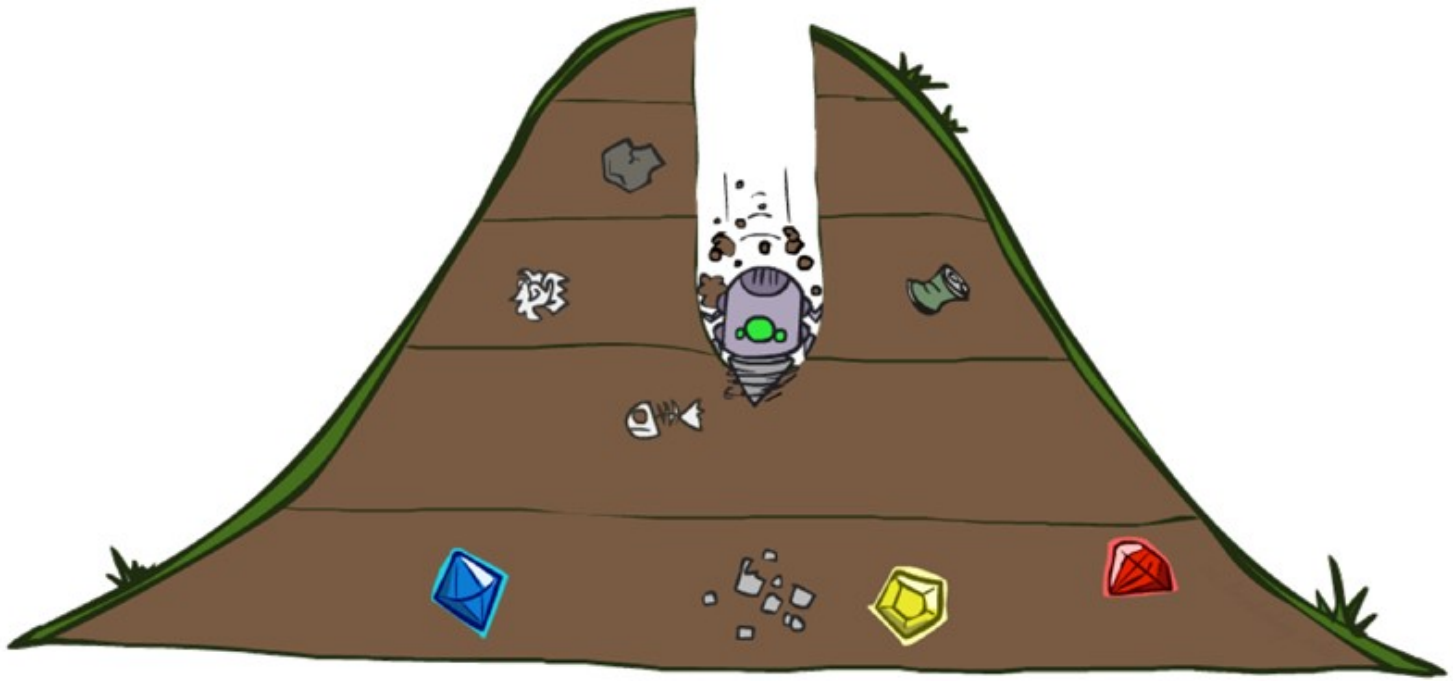
- $1 + b + b^2 + b^3 + \ldots + b^s = b(b^{s-1}) = \mathbf{O(b^s)}$

Is it optimal?

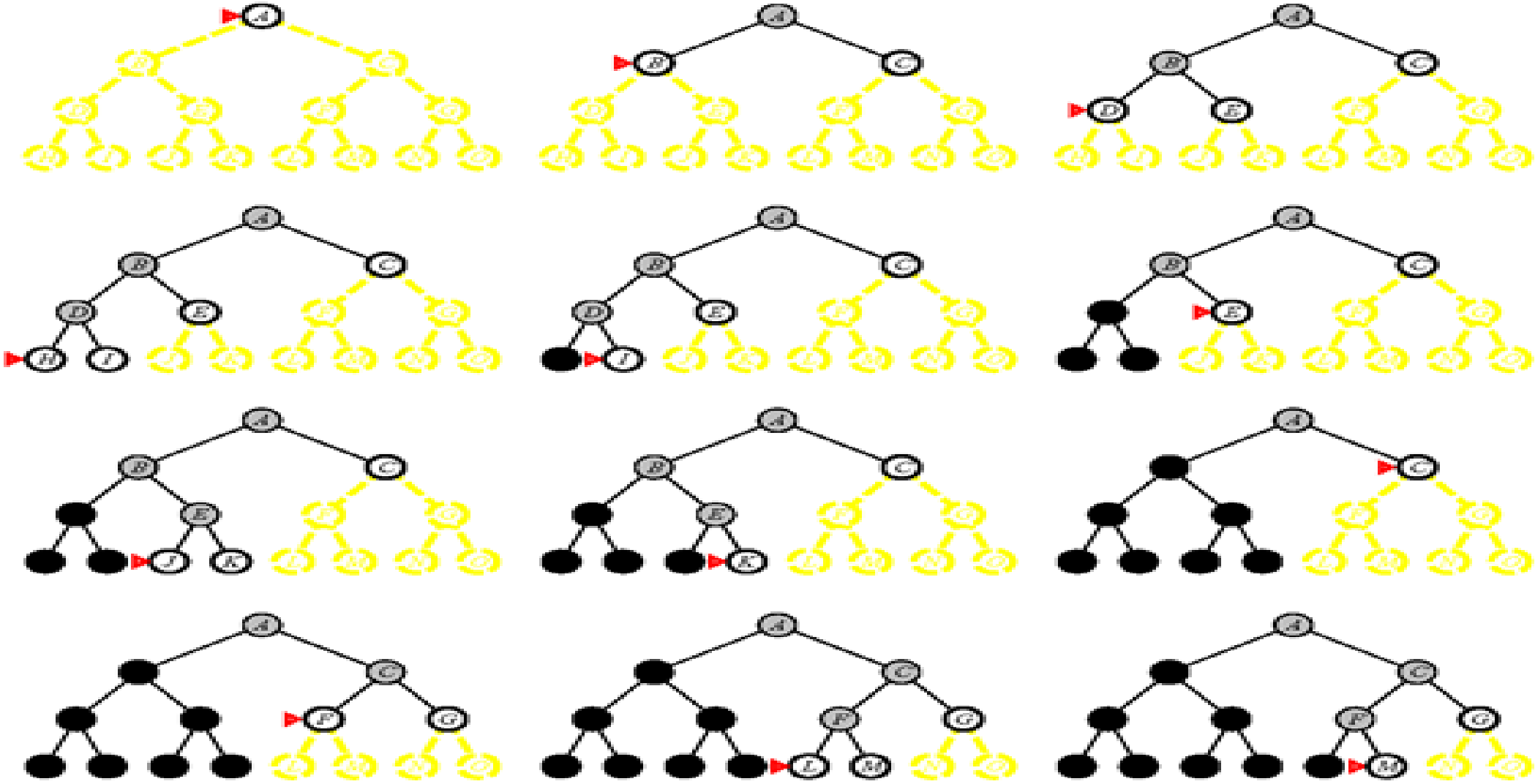
- **Yes.** it is optimal provided the path cost is a nondecreasing function of the depth of the node. Only if costs are all 1 (for all operators have the same cost)



DEPTH FIRST SEACRH



DEPTH FIRST SEARCH



DEPTH FIRST SEARCH

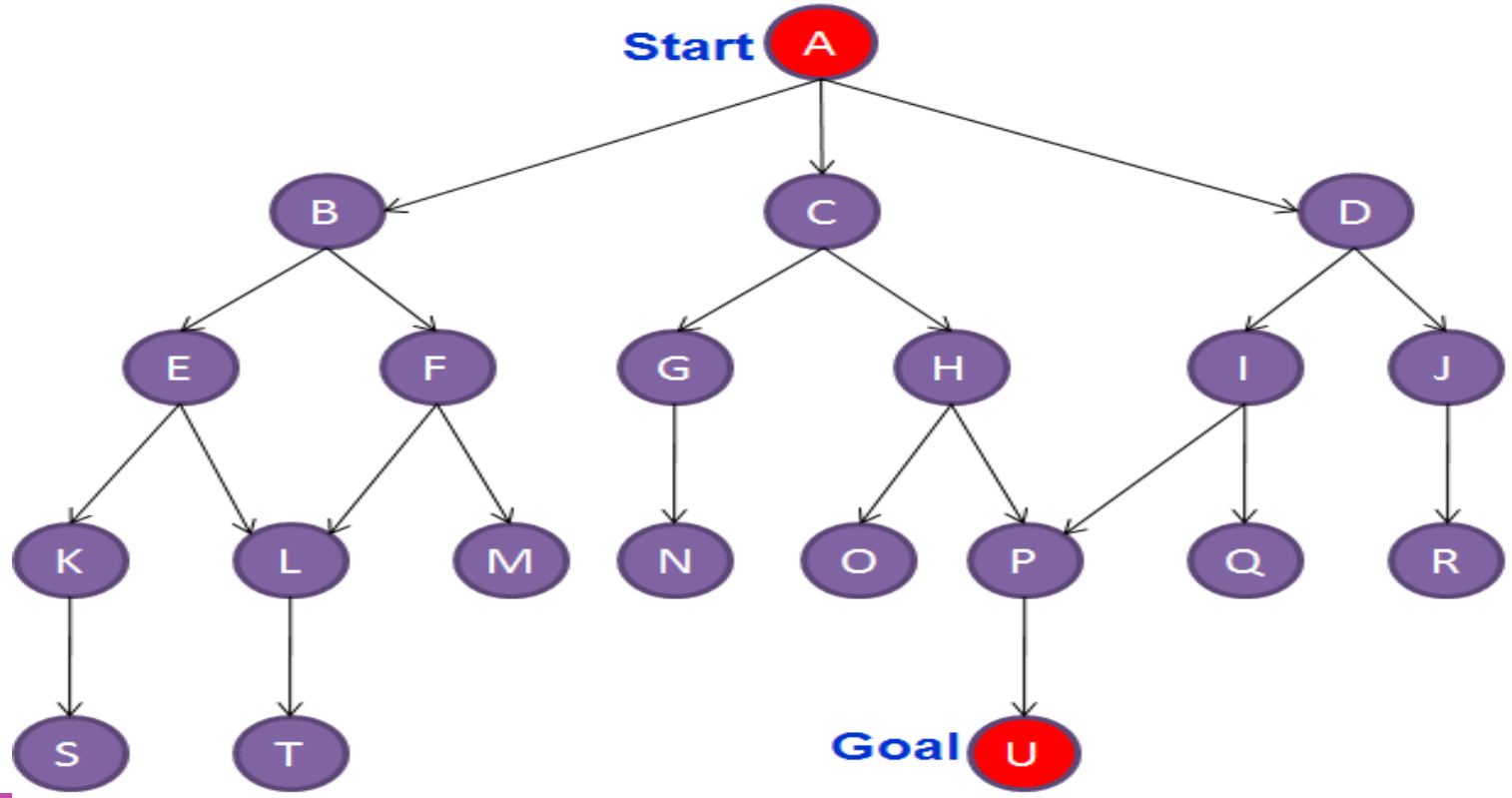
- Depth-first search always expands one of the **nodes at the deepest level of the tree**.
- Enqueue nodes on nodes in **LIFO** (last-in, first-out) order. That is, nodes used as a **stack data structure to order nodes**.
- Depth-first search has very **modest memory requirements.**
- **Implementation:** frontier = **LIFO queue**, i.e., put successors at front
- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D

Algorithm outline:

1. Always select from the OPEN the node with the greatest depth for expansion, and put all newly generated nodes into OPEN
2. OPEN is organized as **LIFO** (last-in, first-out) list.
3. Terminate if a node selected for expansion is a goal

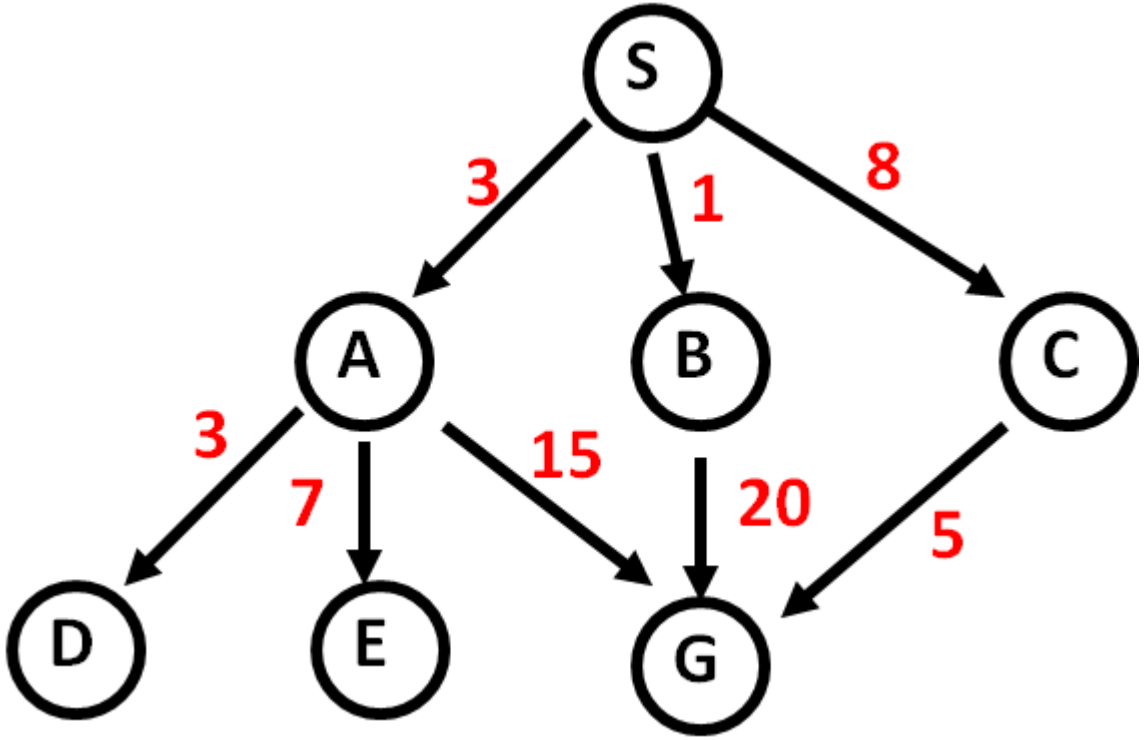
D F S

Depth First Search examines the nodes in the following order:
A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R



OPEN	CLOSED
A	
B C D	A
E F C D	B A
K L F C D	E B A
S L F C D	K E B A
L F C D	S K E B A
T F C D	L S K E B A
F C D	T L S K E B A
M C D	F T L S K E B A
C D	M F T L S K E B A
G H D	C M F T L S K E B A
N H D	G C M F T L S K E B A
H D	N G C M F T L S K E B A
O P D	H N G C M F T L S K E B A
P D	O H N G C M F T L S K E B A
U D	P O H N G C M F T L S K E B A

DFS EXAMPLE



D F S

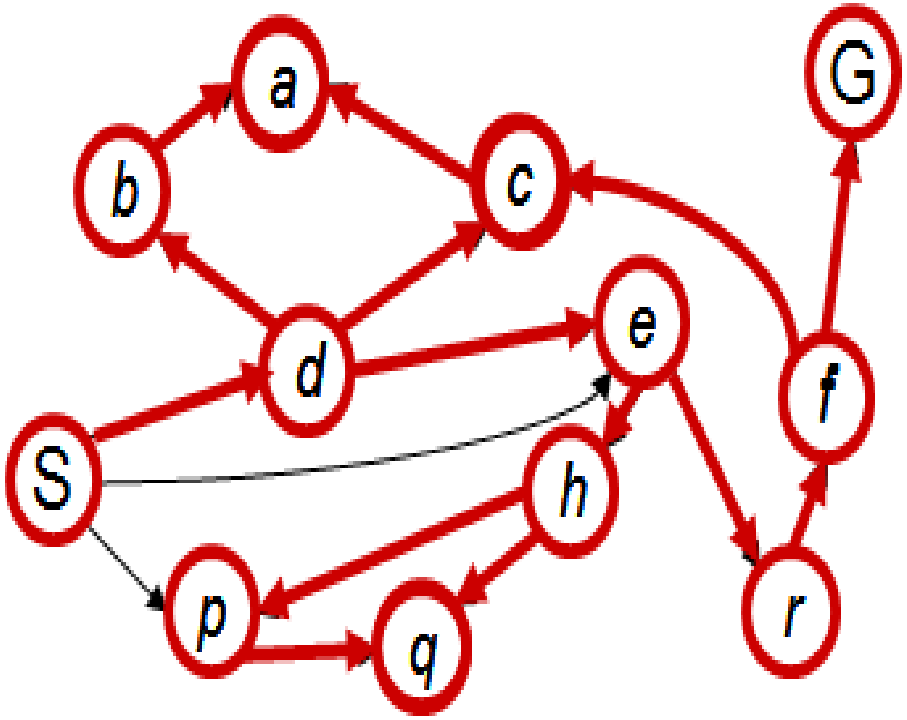
	Expanded node	Nodes list
		{ S^0 }
1	S^0	{ A^3 B^1 C^8 }
2	A^3	{ D^6 E^{10} G^{18} B^1 C^8 }
3	D^6	{ E^{10} G^{18} B^1 C^8 }
4	E^{10}	{ G^{18} B^1 C^8 }
5	G^{18}	{ B^1 C^8 }

- Frontier: LIFO Queue(Equivalent to stack)
- Solution path found is S A G, cost 18
- Number of nodes expanded (including goal node) = 5

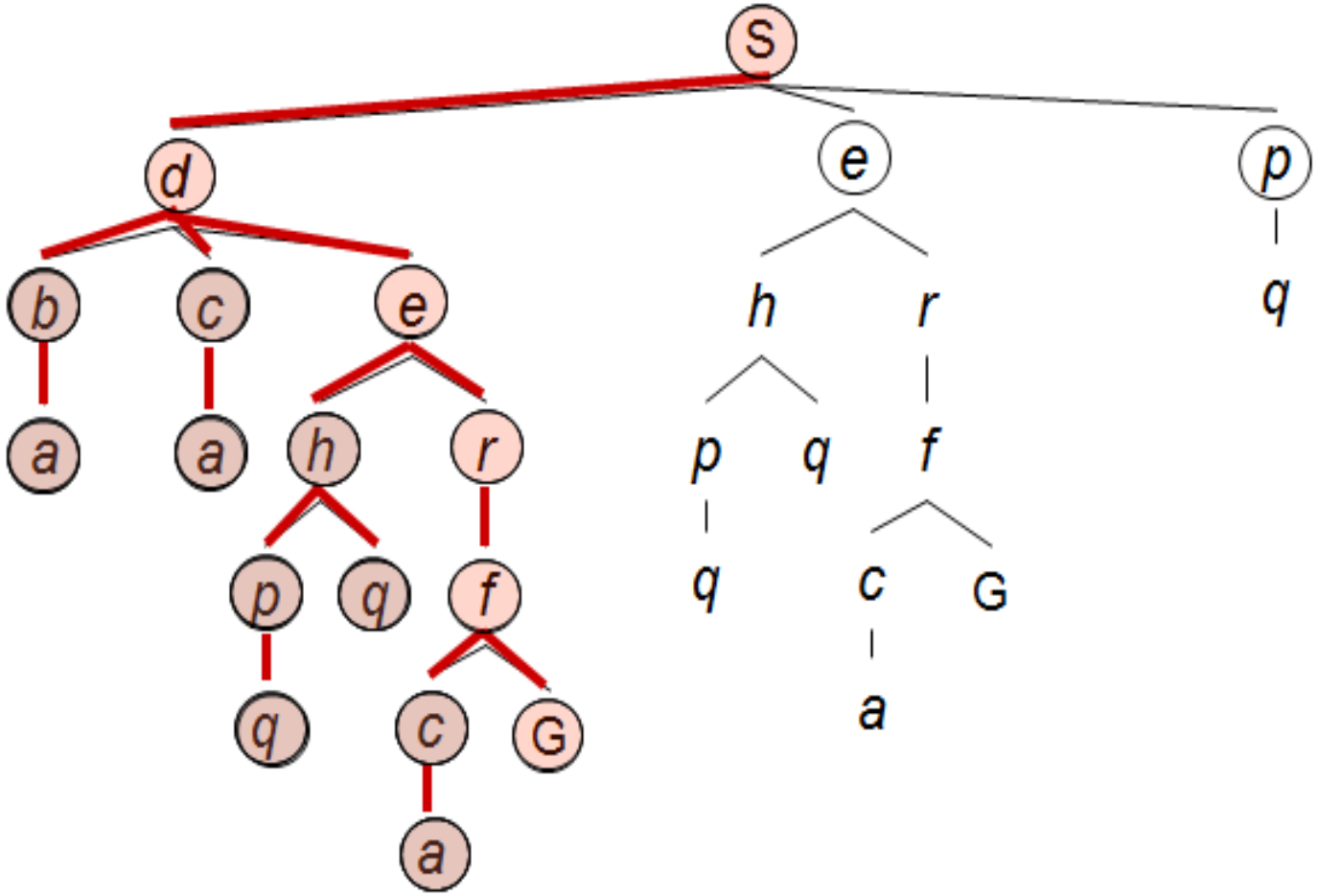
DEPTH FIRST SEARCH

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack



D F S



	Expanded node	Nodes list/ open list
		{ S^0 }
1	s^0	{ d^1 e^1 p^1 }
2	d^1	{b^2 c^2e^2 e^1 p^1}
3	b^2	{ a^3 c^2e^2 e^1 p^1 }
4	a^3	{ c^2e^2 e^1 p^1}
5	c^2	{ a^3 e^2 e^1 p^1 }
6	e^2	{h^3 r^3 e^1 p^1}
7	h^3	{ p^3 q^3 r^3 e^1 p^1}
8	p^3	{ q^4 q^3 r^3 e^1 p^1}
9	q^4	{ q^3 r^3 e^1 p^1}
10	r^3	{ f^4 e^1 p^1}
11	f^4	{ c^5g^5 e^1 p^1}
12	g^5	{ e^1 p^1}

DFS PROPERTIES

What nodes DFS expand?

Some left prefix of the tree.

Could process the whole tree!

If m is finite, takes time $O(b^m)$

How much space does the fringe take?

how many nodes can be in the queue (worst-case)?

At depth l < d (max) we have b-1 nodes

So **O(b*d)**linear space. terrible if m is much larger than d but if solutions are dense, it may be much faster than breadth-first

Time?

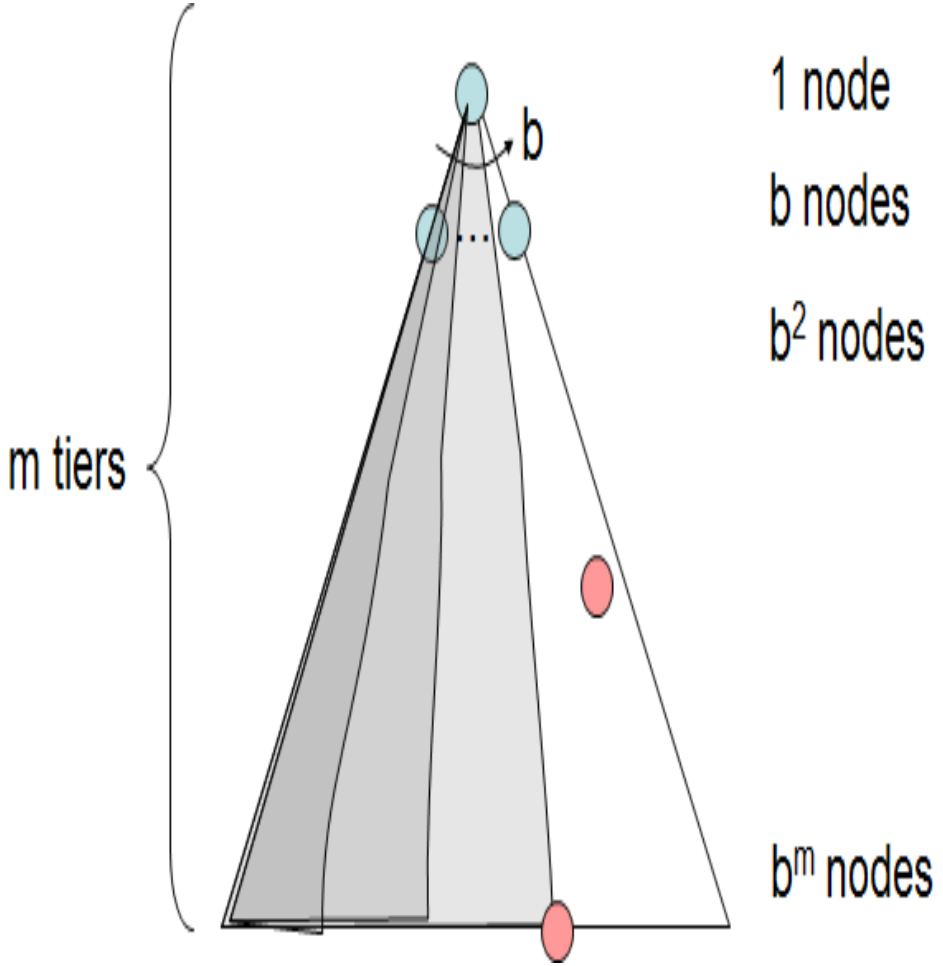
$O(b^m)$

Is it complete?

No. It fails in infinite-depth spaces it also fails in finite spaces with loops. but if we modify the search to avoid repeated states ⇒ complete in finite spaces (even with loops)

Is it optimal?

No, it finds the “leftmost” solution, regardless of depth or cost



Criterion	Breadth-First	Depth-First
Time	b^d	b^m
Space	b^d	bm
Optimal?	Yes if we guarantee that deeper solutions are less optimal, e.g. step-cost=1	No It may find a non-optimal goal first
Complete?	Yes it always reaches goal (if b is finite)	No Fails in infinite-depth spaces
Advantages	Guaranteed to find a solution (if one exists) - complete;	Simple to implement;
	Depending on the problem, can be guaranteed to find an ***optimal*** solution.	Needs relatively **small memory** for storing the state-space.
Disadvantages	Needs a **lot of memory** for storing the state space if the search space has a high branching factor.	Sometimes fail to find a solution (may be get stuck in an infinite long branch) - **not *complete***;
	More **complex** to implement;	**Not** guaranteed to find an ***optimal*** solution (may not find the shortest path solution);
		Can take a lot **longer** to find a solution.

DEPTH LIMITED SEARCH

- Depth-limited search avoids the pitfalls of depth-first searchwhich is infinite path by **imposing a cutoff on the maximum depth of a path**.
- Depth-first search with depth limit l. Algorithm treats the node at the depth limit l as it has no successor nodes further.
- In this algorithm, Depth-limited search can be terminated with two Conditions of failure:
  - **Standard failure value:** It indicates that problem does not have any solution.
  - **Cutoff failure value**: It defines no solution for the problem within a given depth limit.

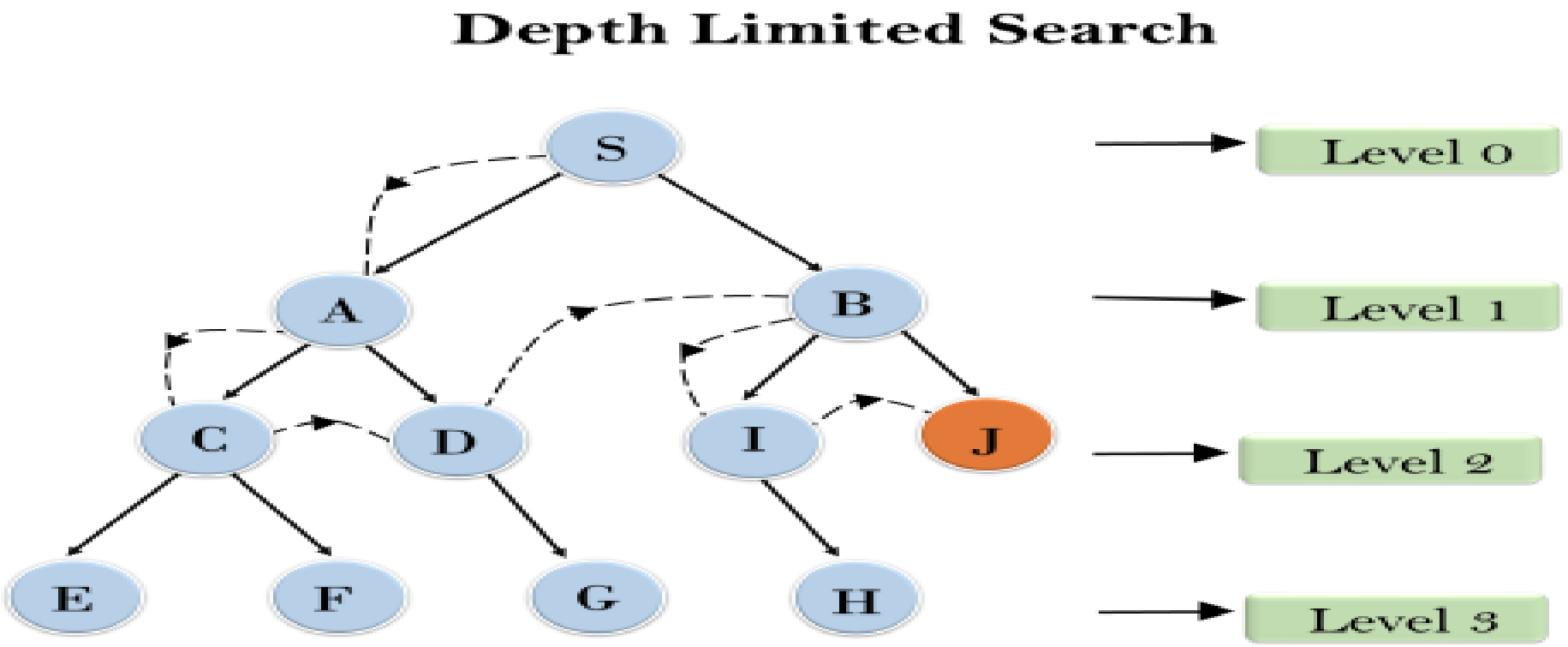
Advantages:

- Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

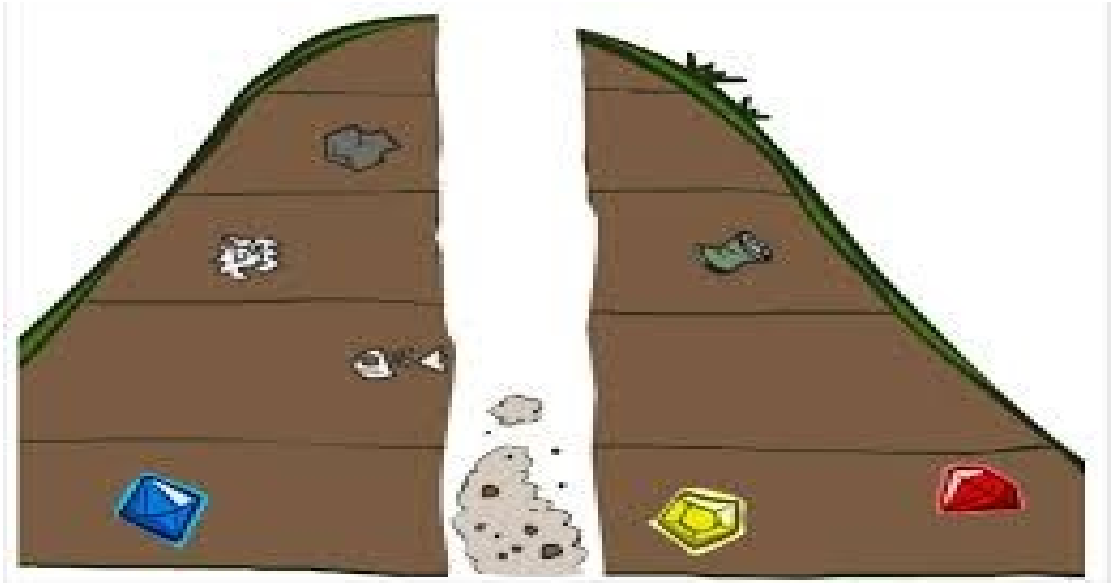
DEPTH LIMITED SEARCH



PROPERTIES OF DLS

- **Complete?**
- **Yes** (unless the node is within the depth l)
- **Time?**
- **$O(b^l)$** Exponential
- **Space?**
- **O(bl)** Keeps all nodes in memory
- **Optimal?**
- **No** (depending upon search algo and heuristic property)

ITERATIVE DEEPENING SEARCH



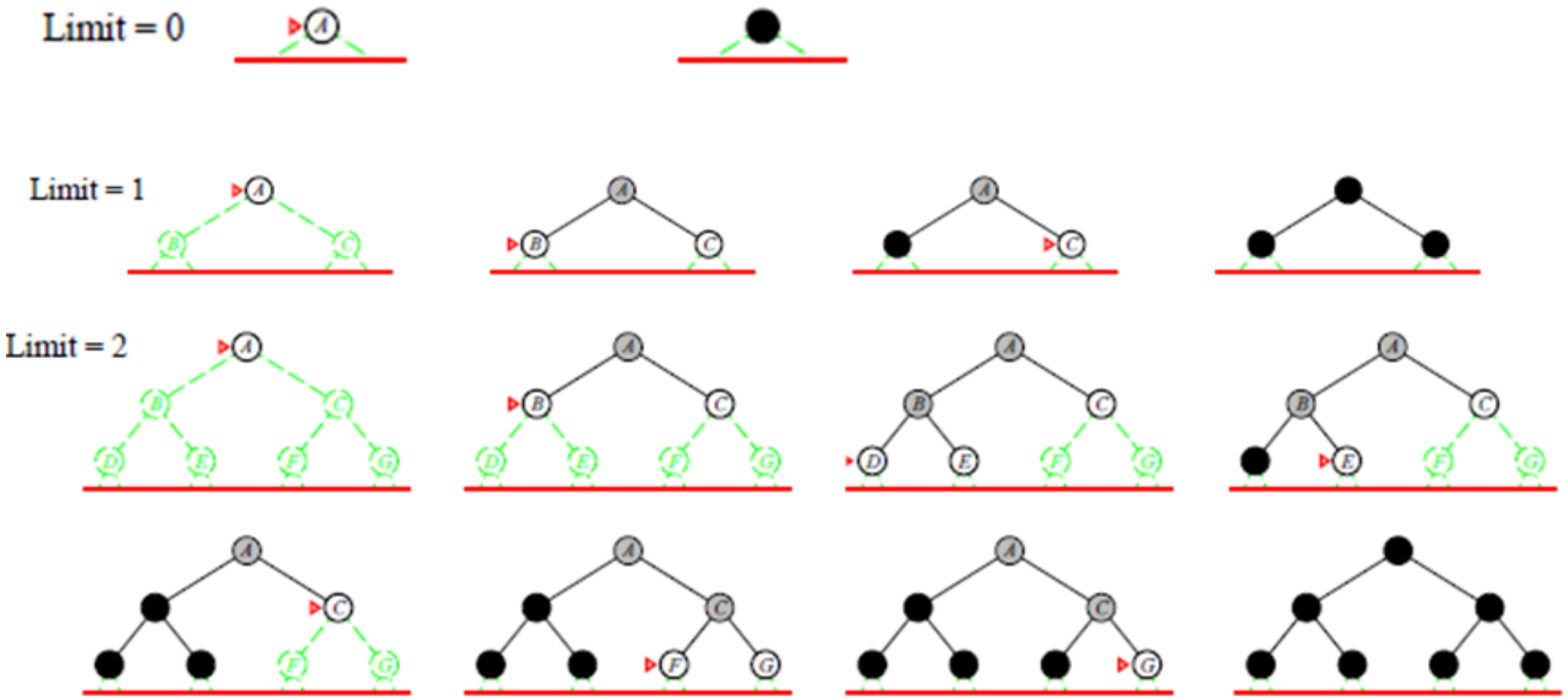
ITERATIVE DEEPENING SEARCH

- The hard part about depth-limited search is picking a good limit, which is known as diameter of the state space. for most problems, we will not know a good depth limit until we have solved the problem.
- 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 iterative deepening algorithm is a combination of **DFS and BFS** algorithms. This search algorithm **finds out the best depth limit** and does it by iteratively increasing the depth limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don’t expand beyond depth L.

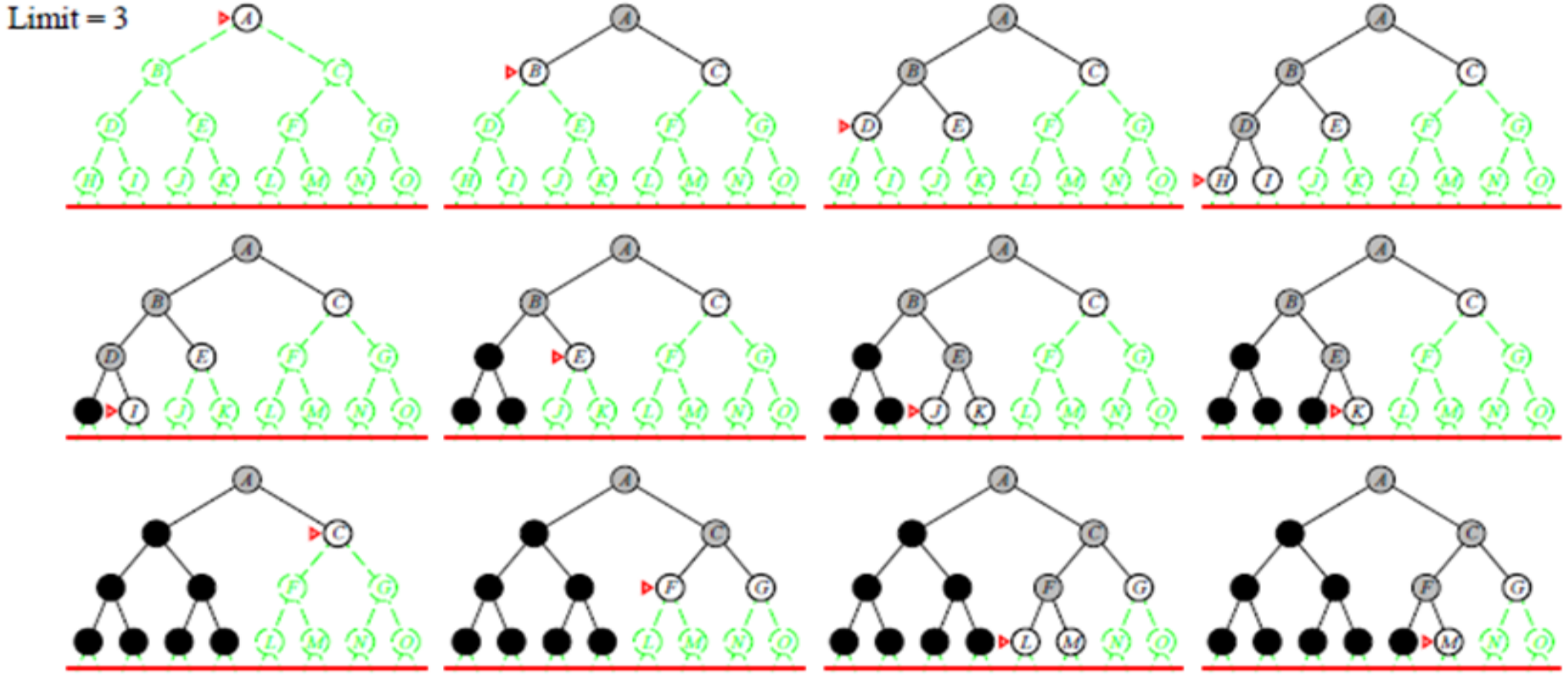
ITERATIVE DEEPENING SEARCH ALGORITHM

- Explore the nodes in DFS order.
- Set a LIMIT variable with a limit value.
- Loop each node up to the limit value and further increase the limit value accordingly.
- Terminate the search when the goal state is found.

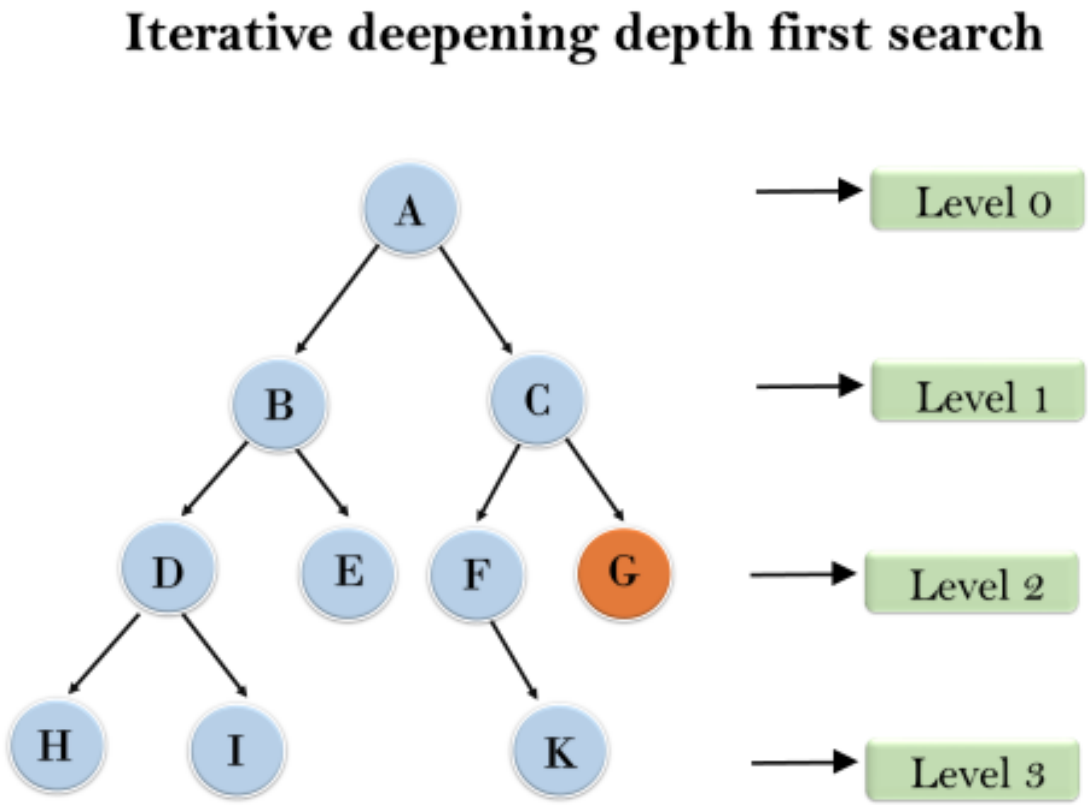
ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH

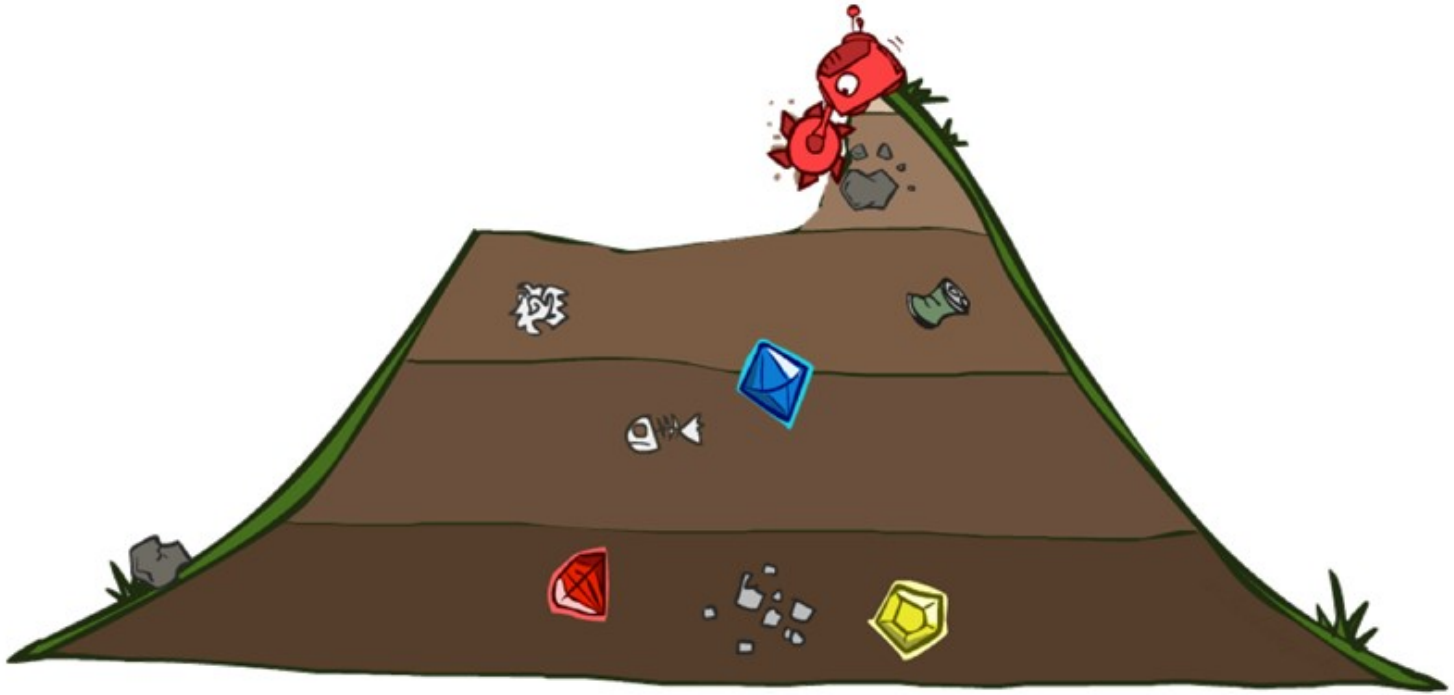


1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

PROPERTIES OF ITERATIVE DEEPENING SEARCH

- Complete?? **Yes**
- Time?? $(d + 1)\, b^0 + db^1 + (d - 1)\, b^2 + \ldots + b^d = O(b^d)$
- Space?? **O(bd)**
- Optimal?? **Yes,** if step cost = 1 it can be modified to explore a uniform-cost tree. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- **Disadvantages of Iterative deepening search**
- The drawback of iterative deepening search is that it seems wasteful because it generates states multiple times.
- **Note:** Generally, iterative deepening search is required when the search space is large, and the depth of the solution is unknown.

UNIFORM COST SEARCH



UNIFORM COST SEARCH

- The primary goal of the uniform-cost search is to **find a path** to the goal node which has the **lowest cumulative cost** ie sort by the **cost-so-far.**
- A uniform-cost search algorithm is implemented by the **priority queue**. It gives **maximum priority to the lowest cumulative cost** and Enqueue nodes by **path cost**.
- Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe. Uniform cost search is **equivalent to BFS algorithm** if the **path cost of all edges is the same.**
- **Algorithm outline:** Let **g(n) = cost of the path from the start node to the current node n**. Sort nodes by increasing value of g
  - Always select from the OPEN the node with the least g(.) value for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(.) values (in ascending order)
  - Terminate if a node selected for expansion is a goal
- Called *"Dijkstra's Algorithm"* in the algorithm's literature and similar to *"Branch and Bound Algorithm"* in operations research literature

UNIFORM COST SEARCH

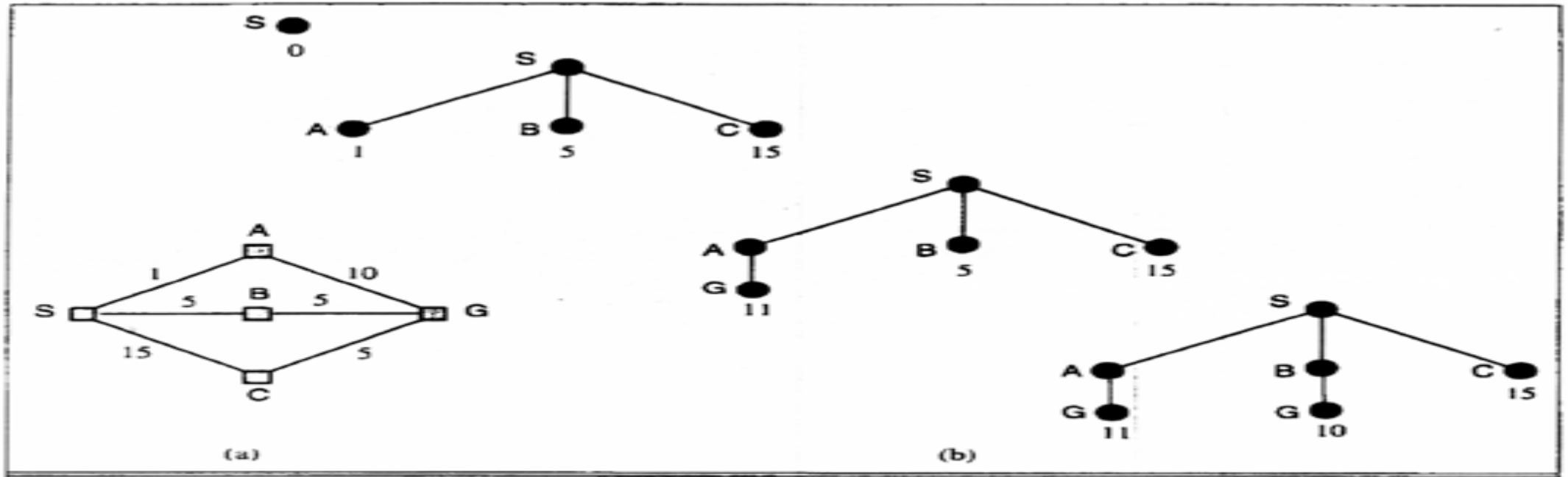
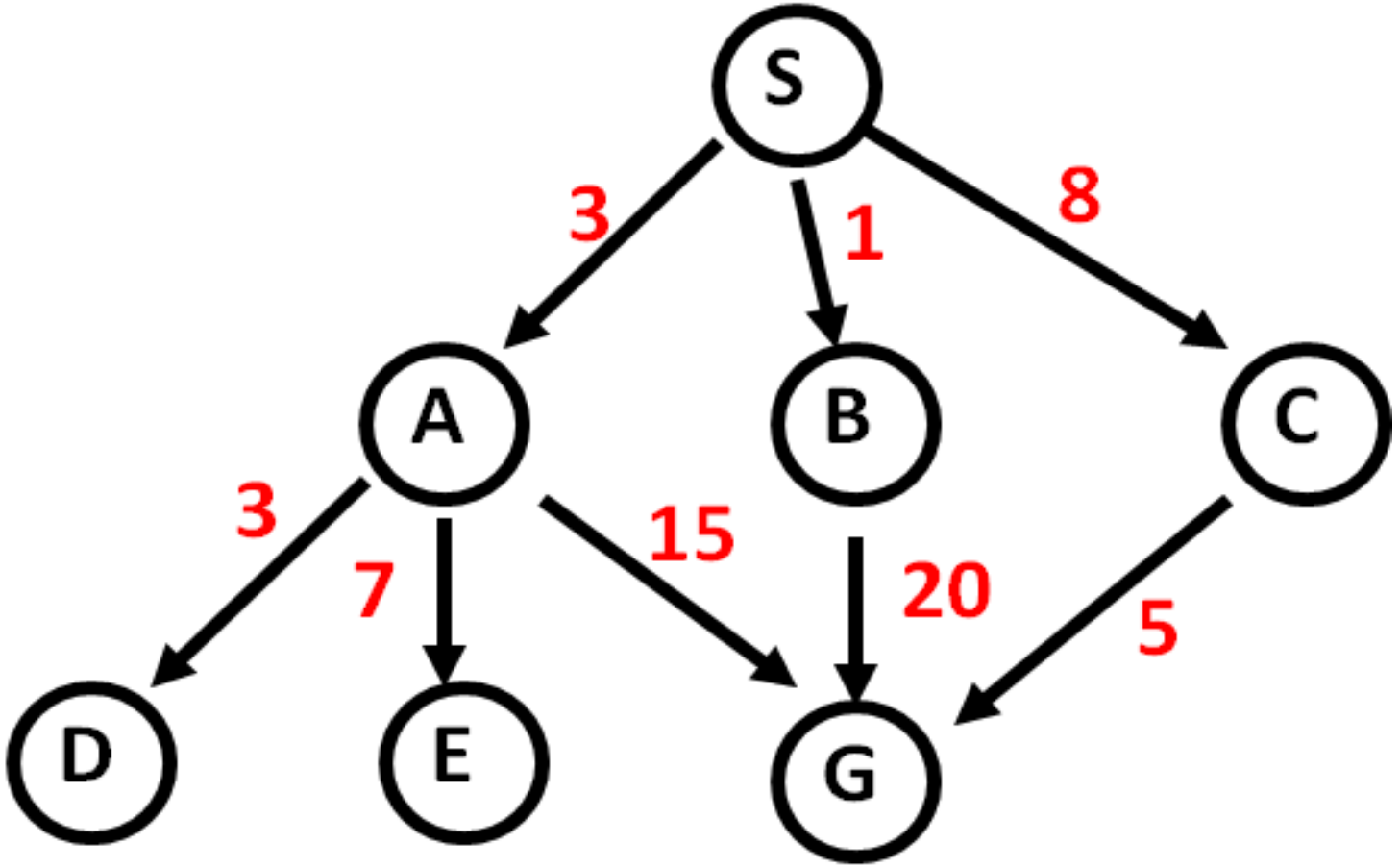


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

UNIFORM COST SEARCH



UNIFORM COST SEARCH

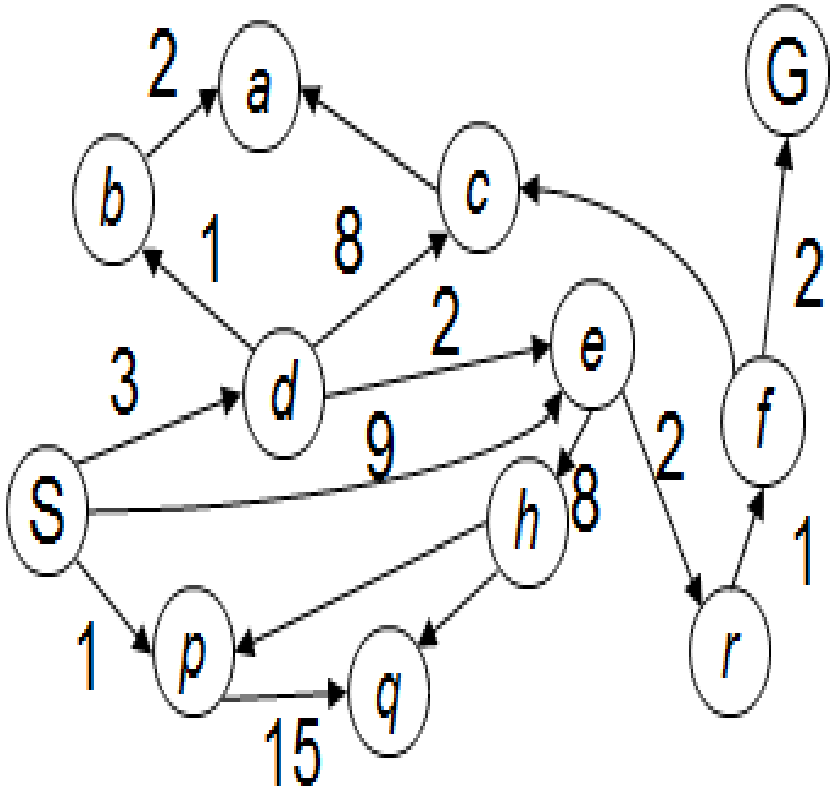
	Expanded node	Nodes list/open list
		$\{ S^0 \}$
1	S^0	$\{ B^1\ A^3\ C^8 \}$
2	B^1	$\{ A^3\ C^8\ G^{21} \}$
3	A^3	$\{ D^6\ C^8\ E^{10}\ G^{18}\ G^{21} \}$
4	C^8	$\{ C^8\ E^{10}\ G^{18}\ G^{21} \}$
5	C^8	$\{ E^{10}\ G^{13}\ G^{18}\ G^{21} \}$
6	E^{10}	$\{ G^{13}\ G^{18}\ G^{21} \}$
7	G^{18}	$\{ G^{21}\ G^{13} \}$

- Solution path found is S B G, cost 13
- Number of nodes expanded (including goal node) = 7

UNIFORM COST SEARCH

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



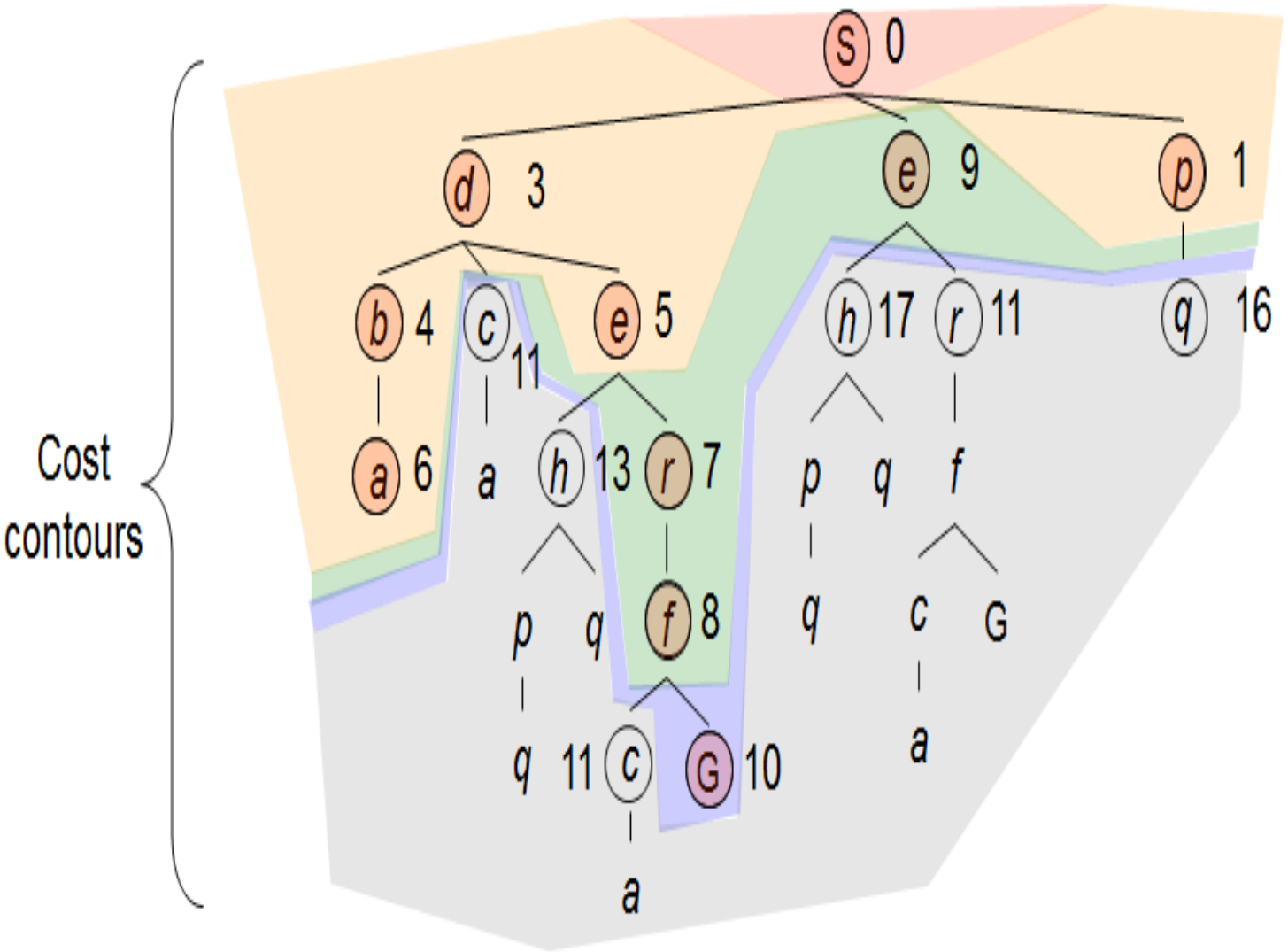
The good:

UCS is complete and optimal!

The bad:

Explores options in every “direction”. No information about goal location

UNIFORM COST SEARCH



	Expanded node	Nodes list/ open list
		{ S^0 }
1	s^0	{ p^1 d^3 e^9 }
2	p^1	{d^3 e^9q^{16}}
3	d^3	{ b^4 e^5e^9 $c^{11}q^{16}$ }
4	b^4	{ e^5 a^6 e^9 $c^{11}q^{16}$}
5	e^5	{a^6 r^7e^9 c^{11} $h^{13}q^{16}$ }
6	a^6	{ r^7e^9 c^{11} $h^{13}q^{16}$ }
7	r^7	{ f^8e^9 c^{11} $h^{13}q^{16}$}
8	f^8	{ e^9 $g^{10}c^{11}$ $h^{13}q^{16}$}
9	g^{10}	{ c^{11} $h^{13}q^{16}$}

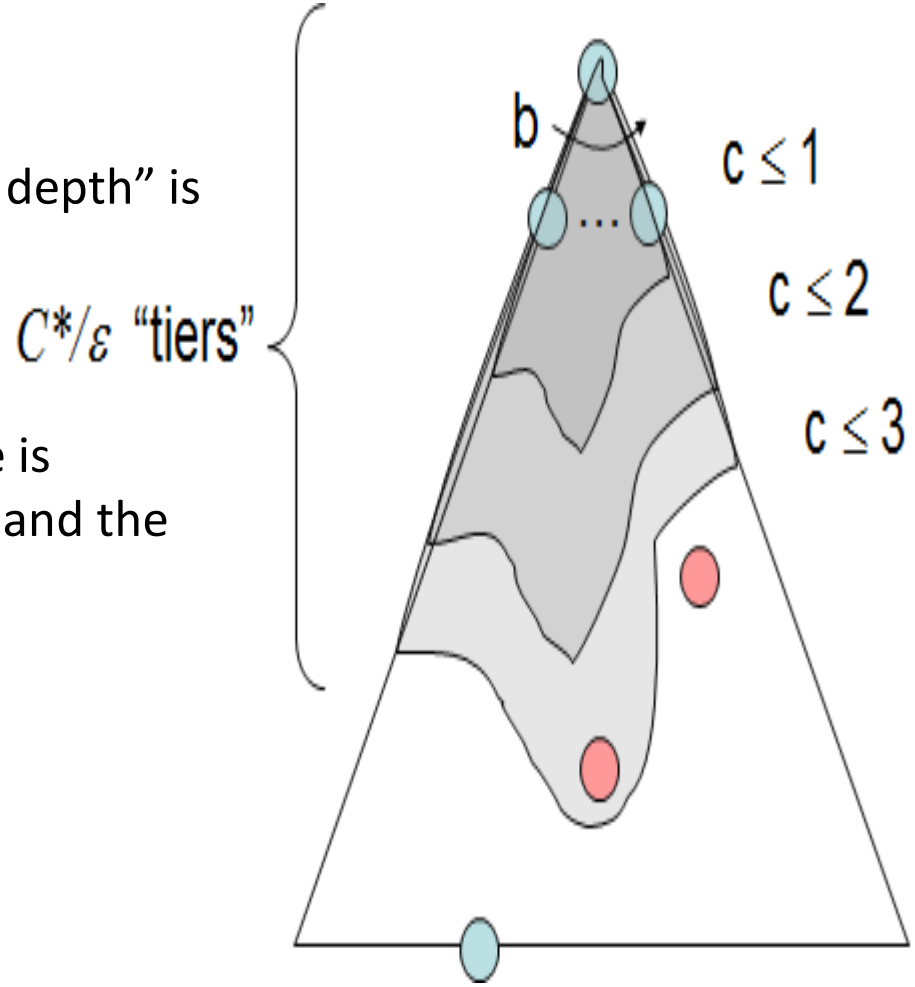
UCS PROPERTIES

What nodes does UCS expand?

Processes all nodes with cost less than cheapest solution!

If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε

Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



Optimal?

Yes. Optimality depends on the goal test being applied when a node is removed from the nodes list, not when its parent node is expanded and the node is first generated

Time?

Exponential time and space complexity, **$O(b^{C^*/\varepsilon})$**

How much space does the fringe take?

Has roughly the last tier, so **$O(b^{C^*/\varepsilon})$**

Is it complete?

Assuming best solution has a finite cost and minimum arc cost is positive, **yes**!

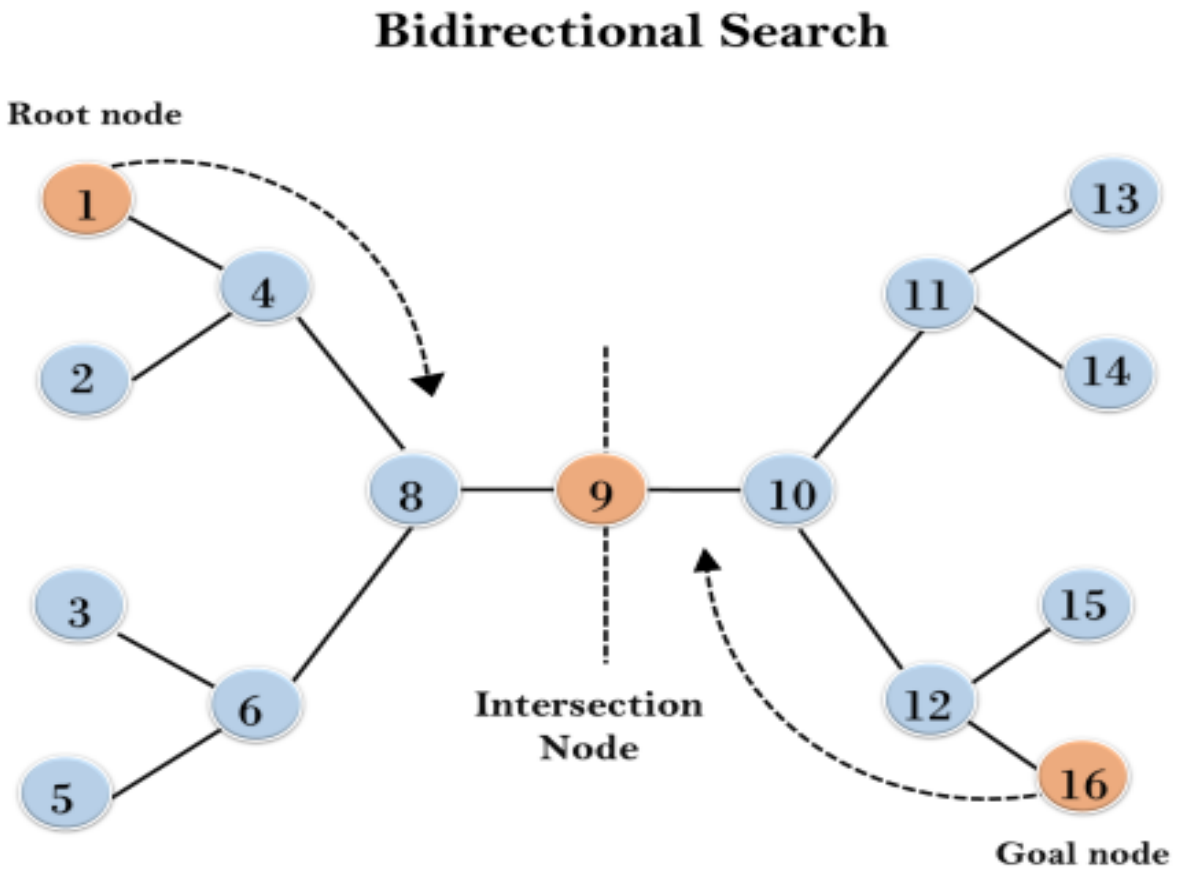
BIDIRECTIONAL SEARCH

- Bidirectional search algorithm runs **two simultaneous searches**, **one form initial state** called as **forward-search** and **other from goal node** called a**s backward-search**, to find the goal node. The search stops when these two graphs intersect each other.
- It is implemented by replacing goal test with a test to see whether **frontiers of 2 searches intersect.**
- Bidirectional search can use search techniques such as BFS, DFS, DLS, Iterative deepening search etc.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- **Idea**
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes. need a way to specify the predecessors of G this can be difficult

When to use bidirectional approach?

- Both initial and goal states are unique and completely defined.
- The branching factor is same in both directions.

BIDIRECTIONAL SEARCH



PROPERTIES OF BIDIRECTIONAL SEARCH

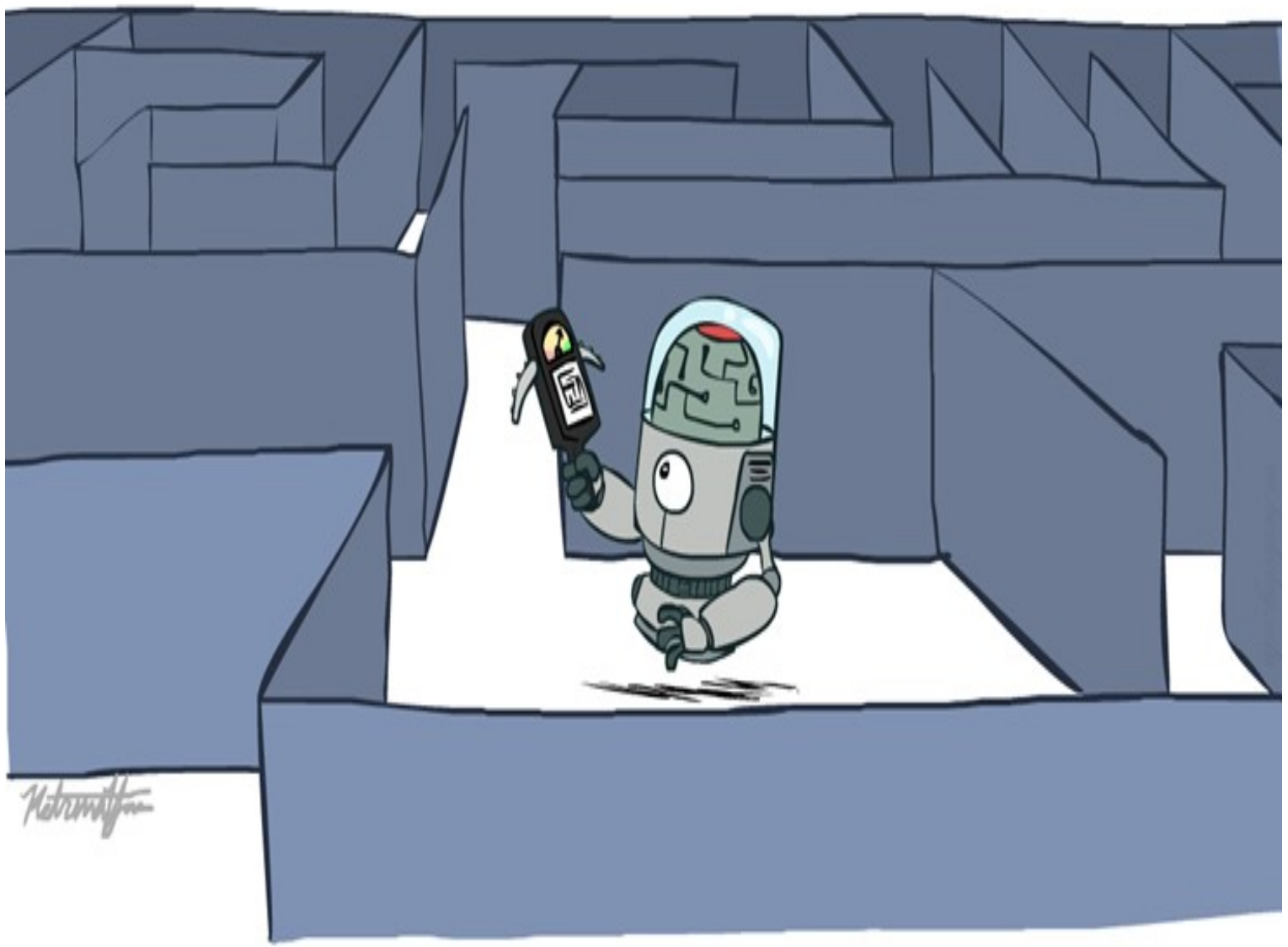
- Complete: Yes. Bidirectional search is complete.
- Optimal: Yes. It gives an optimal solution.
- Time and space complexity: Bidirectional search has $\mathbf{O(b^{d/2})}$
- Disadvantage of Bidirectional Search
- **Though time complexity is reduced increased search complexity is a weakness owing to intersection check and how to search backward.**
- If the goal is an abstract description like for N-Queens problem its very difficult to use.

COMPARISON

Criterion	BFS	DFS	Limited Depth	Iterative deepening	Bidirectional	Uniform Cost Search
Time	B^d	B^m	B^l	B^d	$B^{d/2}$	$B^{c*/\square}$
Space	B^d	B*m	B*l	Bd	$B^{d/2}$	$B^{c*/\square}$
Optimality?	Yes	No	No	Yes	Yes	Yes
Completeness	Yes	No	Yes if l≥d	Yes	Yes if e≥0	Yes if □≥0

- **m** – maximum depth of the tree, **l** –depth limit of search (AdMax
- □-smallest step cost
- C* -cost of optimal solution

INFORMED SEARCH STRATEGIES



INFORMED SEARCH

- Informed search algorithms **use domain knowledge.** In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution **more efficiently than an uninformed search strategy.** Informed search is also called a **Heuristic search.**
- A **heuristic** is a way which **might not always be guaranteed for best solutions** but **guaranteed to find a good solution in reasonable time**.
- A node is selected for expansion in informed search algorithm based on an evaluation function that estimates cost to goal.
- Informed search can solve much complex problem which could not be solved in another way.
- It contains the problem description as well as extra information like how far is the goal node.
- It might not give the optimal solution always but it will definitely give a good solution in a reasonable time. It can solve complex problems more easily than uninformed.

Example: traveling salesman problem, Greedy Search, A* Search

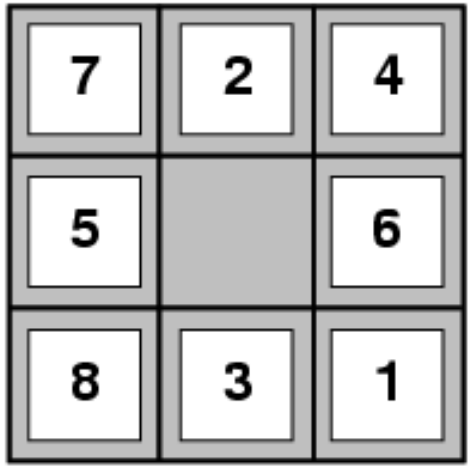
HEURISTIC FUNCTIONS

- A **heuristic function, h(n)**, is the **estimated cost of the cheapest path from the state at node n, to a goal state**.
- The value of the heuristic function is **always positive.**
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
- **Examples:** Manhattan distance, Euclidean distance for pathing
- Heuristic is a function which is used in Informed Search finds the most promising path.
- Heuristic functions are very much dependent on the domain used. h(n) might be the estimated number of moves needed to complete a puzzle, or the estimated straight-line distance to some town in a route finder.
- Choosing an appropriate function greatly affects the effectiveness of the state-space search, since it tells us which parts of the state-space to search next.
- A heuristic evaluation function which accurately represents the actual cost of getting to a goal state, tells us very clearly which nodes in the state-space to expand next, and leads us quickly to the goal state.

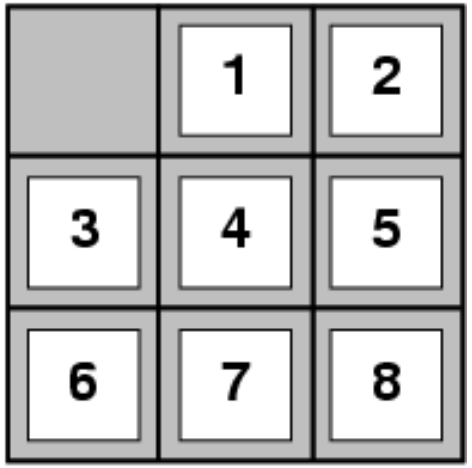
EXAMPLE HEURISTICS

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)**
- $h_1(S)$ = ?
- $h_2(S)$ = ?



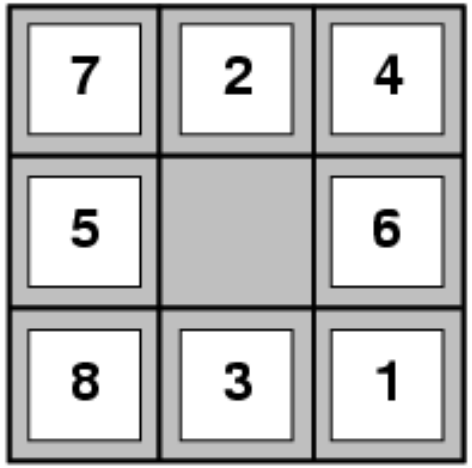
Start State



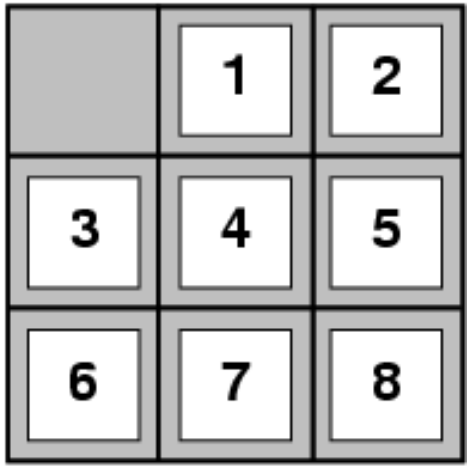
Goal State

EXAMPLE 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)
- $h_1(S) = 8$
- $h_2(S) = 3+1+2+2+2+3+3+2 = 18$



Start State



Goal State

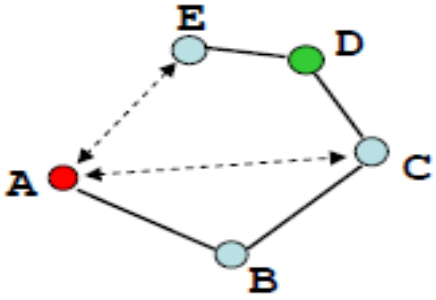
EXAMPLE HEURISTICS

- **For Graph Search problem**
- **Straight-line distance :** The distance between two locations on a map can be known without knowing how they are linked by roads (i.e. the absolute path to the goal).

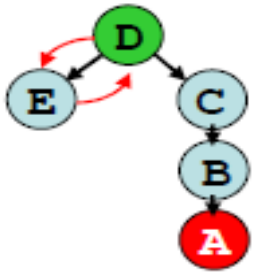
$$d(\mathbf{p},\mathbf{q}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

- This is known as Euclidian distance formula
- In three-dimensional Euclidean space, the distance is

$$d(\mathbf{p},\mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}.$$



Problem Space

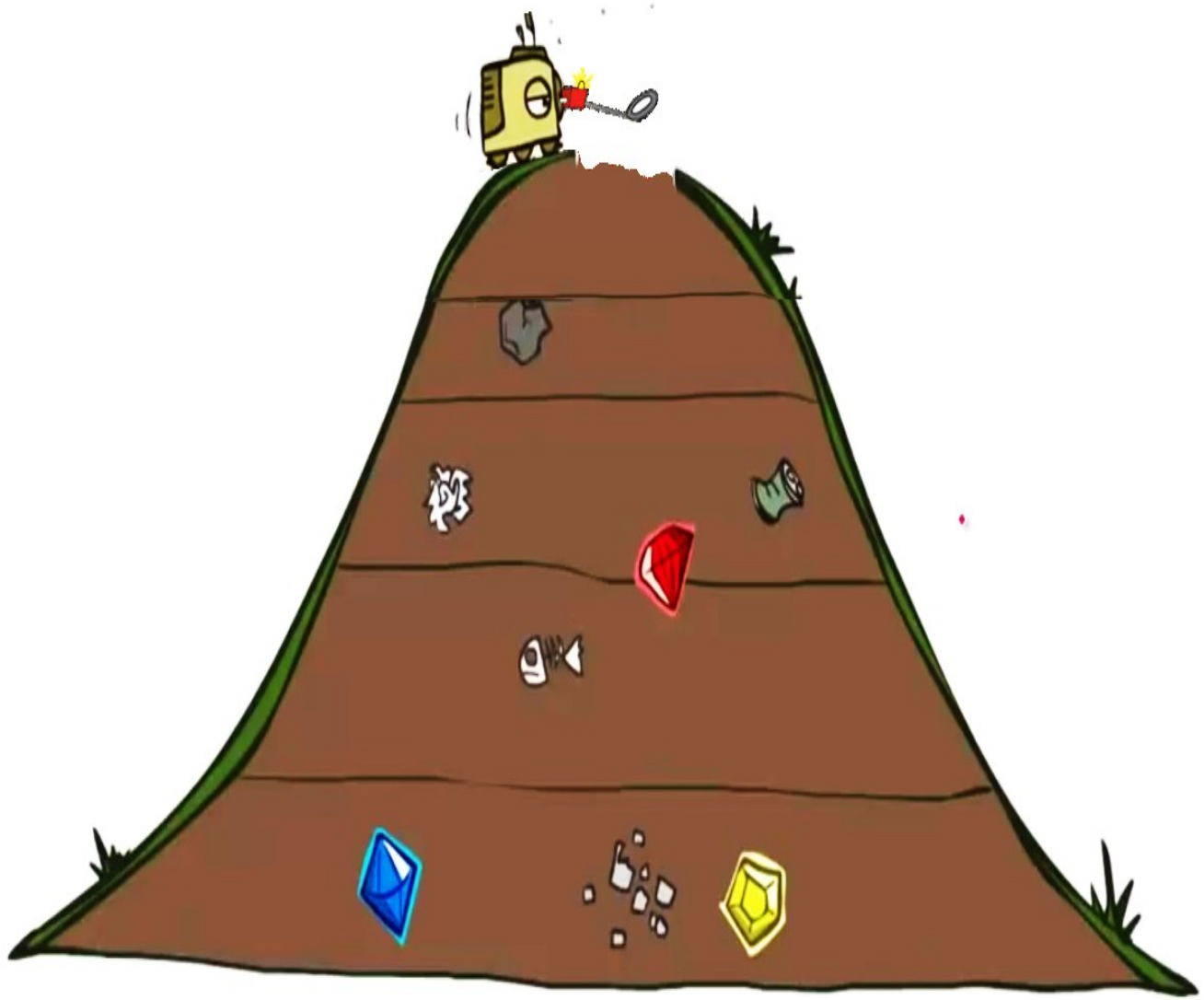


Search Tree

EFFICIENT HEURISTIC ACCURACY ON PERFORMANCE

- Quality of a heuristic is determined by the effective branching factor b*.
- Total no: of nodes generated by A* is N at a depth d with a branching factor b* then,
- $N+1=1+b^*+(b^*)^2+\ldots+(b^*)^d$
- A well defined heuristic should have value of b* close to 1.
- A dominant heuristic will be a better choice.
- A problem with fewer restrictions on the graph is called relaxed problem and cost of an optimal solution to a relaxed problem is admissible heuristic for original problem.

BEST FIRST ERACH



BEST FIRST SEARCH

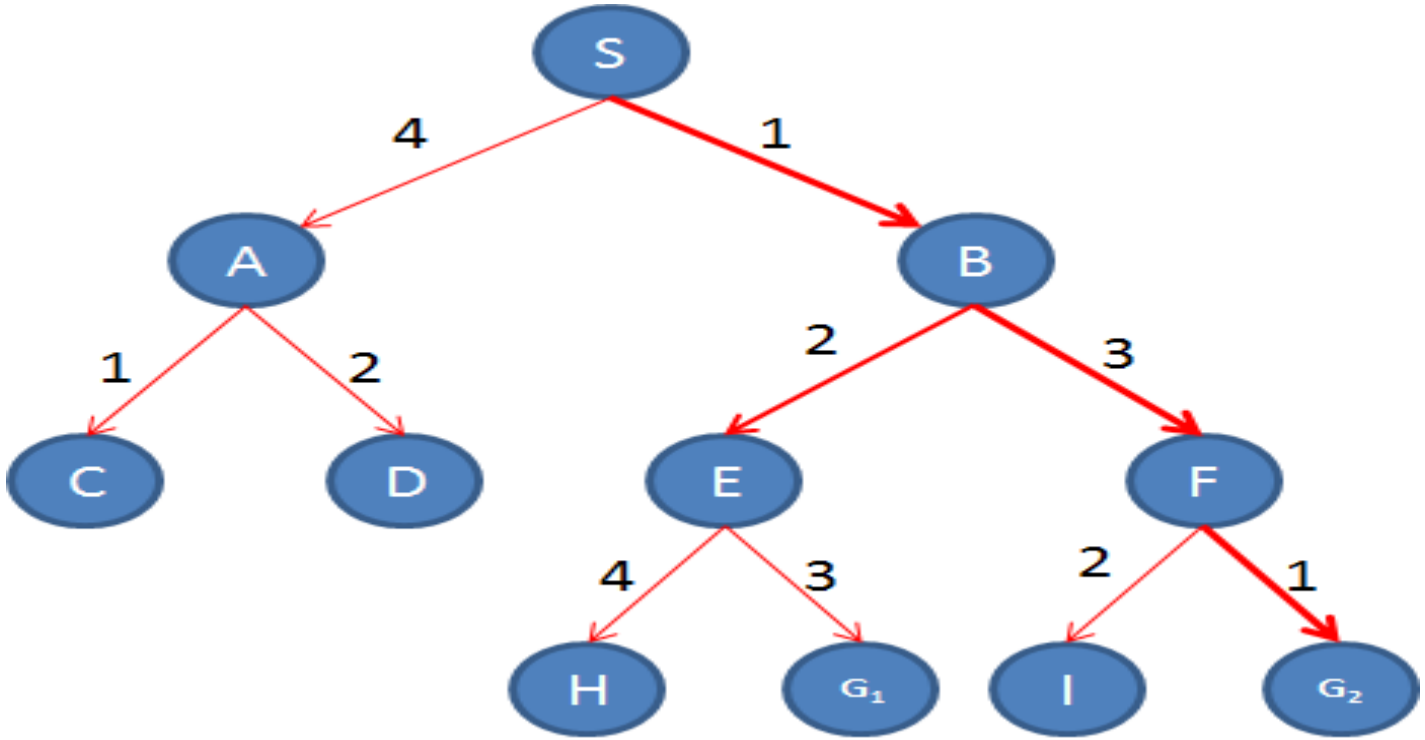
- A search strategy is defined by picking the **order of node expansion** by using an **evaluation function *f(n).*** It is used to assign score for each node.
- **Best-first search** is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function, f (n)**
- The evaluation function **f(n)** is a cost estimate that provides an **estimate for the total cost** also known as **estimate of "desirability“**. Expand the **node n with smallest f(n)** ie most desirable unexpanded node
- Implementation is like uniform cost search where f replaces g in priority queue.
- Choice of f determines search strategy, and most searches use heuristic function as f.

Implementation:

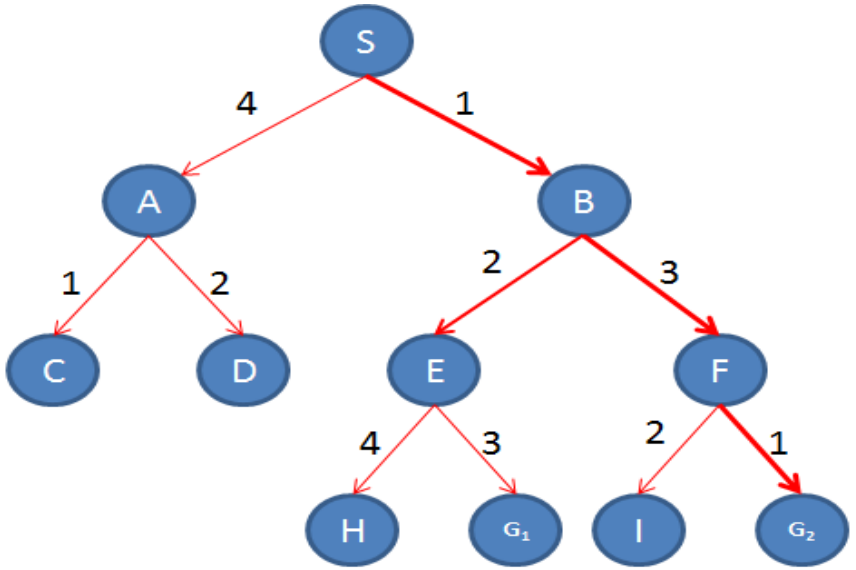
  - Order the nodes in fringe **increasing order of cost**.
  - The algorithm maintains two lists, one containing a list of candidates yet to explore (**OPEN**), and one containing a list of already visited nodes (**CLOSED**). States in OPEN are ordered according to some heuristic estimate of their “closeness” to a goal. This ordered OPEN list is referred to as ***priority queue***.
  - The algorithm always chooses the best of all unvisited nodes that have been graphed

- The advantage of this strategy is that if the algorithm reaches a dead-end node, it will continue to try other nodes.`

BEST FIRST SEARCH EXAMPLE



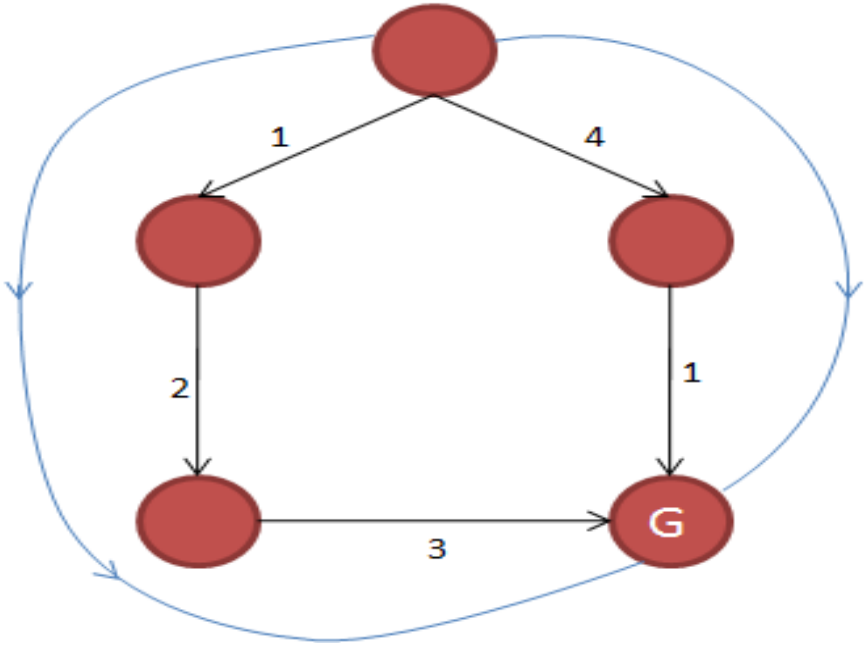
BEST FIRST SEARCH SOLUTION

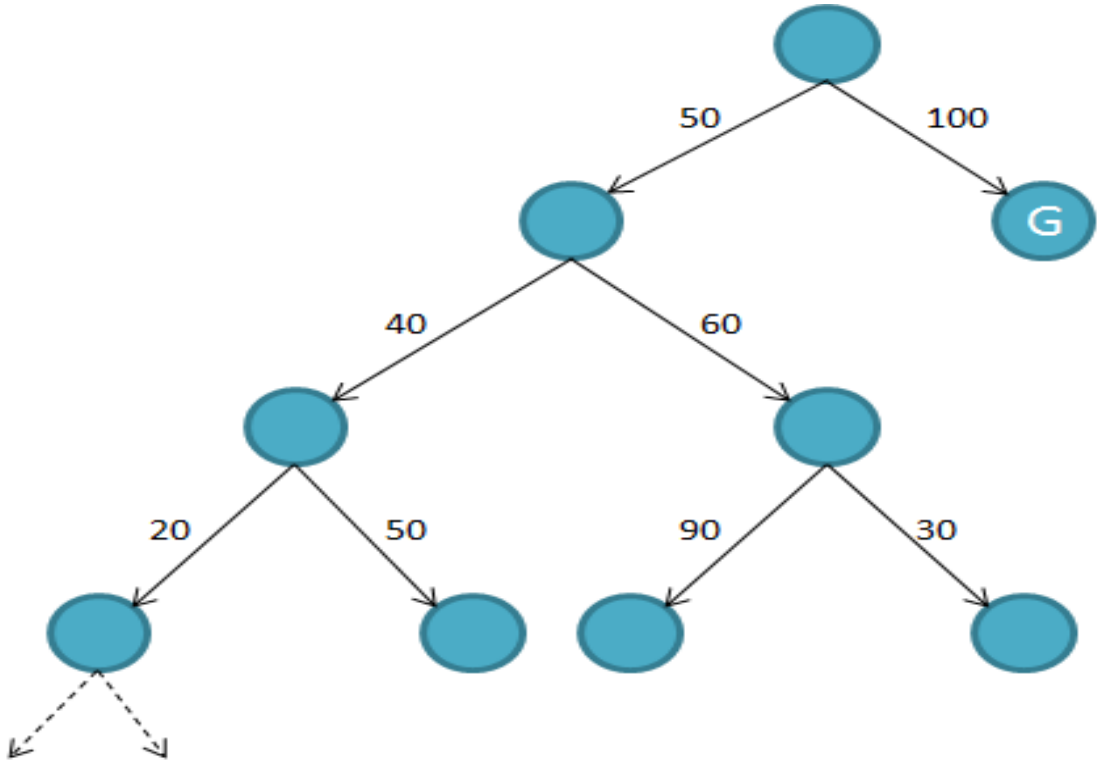


open=[S_0]; closed=[ ]
open=[B_1 , A_4]; closed=[S_0]
open=[E_3 , A_4, F_4]; closed=[S_0 , B_1]
open=[A_4 , F_4 , $G1_6$, , H_7]; closed=[S_0 , B_1 , E_3]
open=[F_4 , C_5 , $G1_6$, D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4]
open=[C_5 , $G2_5$, $G1_6$, I_6 , D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4 , F_4]
open=[$G2_5$, $G1_6$, I_6 , D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4 , F_4 , C_5]

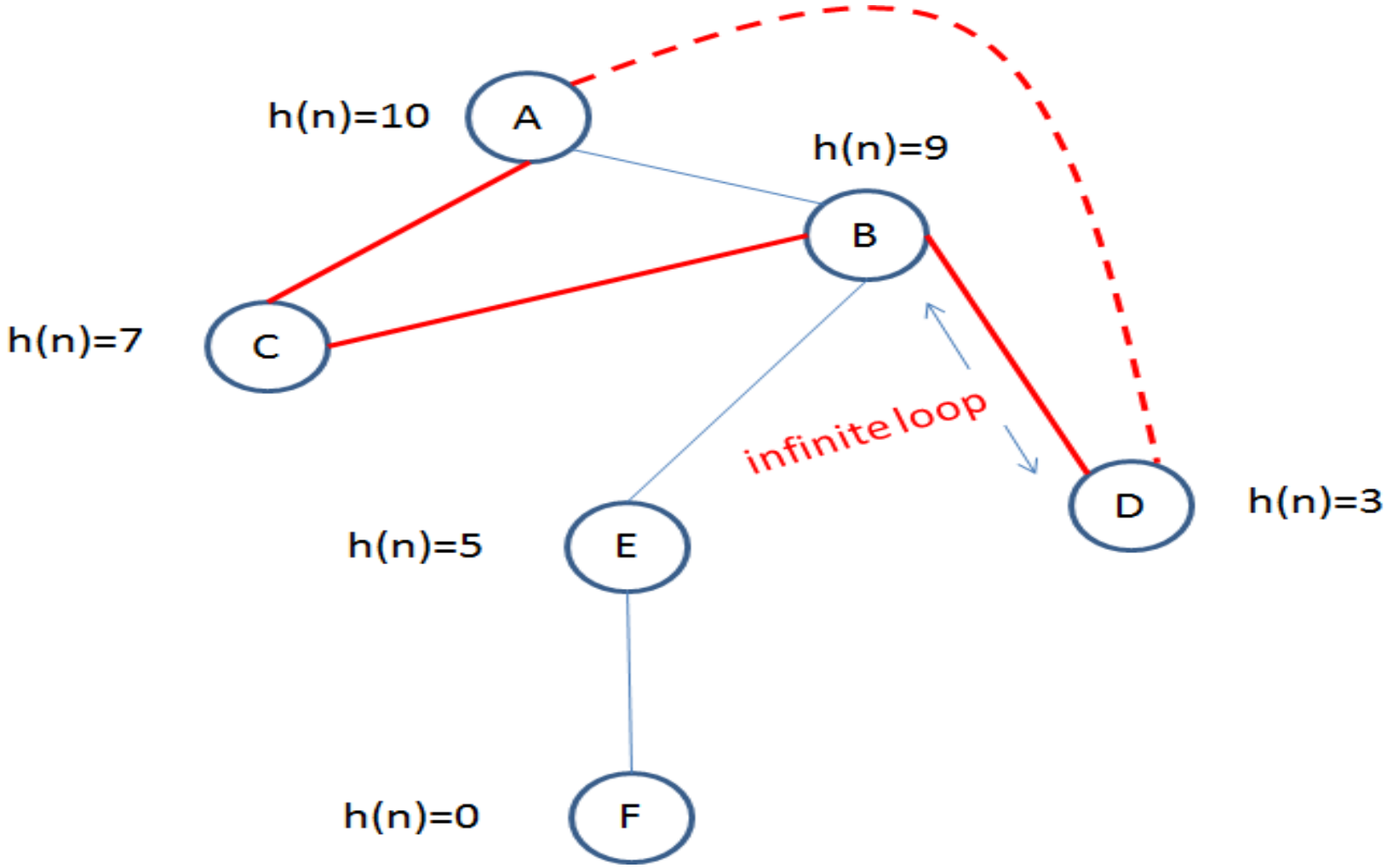
Cost = 1+3+1=5

DOES BEST FIRST ALGORITHM ALWAYS GUARANTEE TO FIND SHORTEST PATH?





GBFS INFINITE LOOP



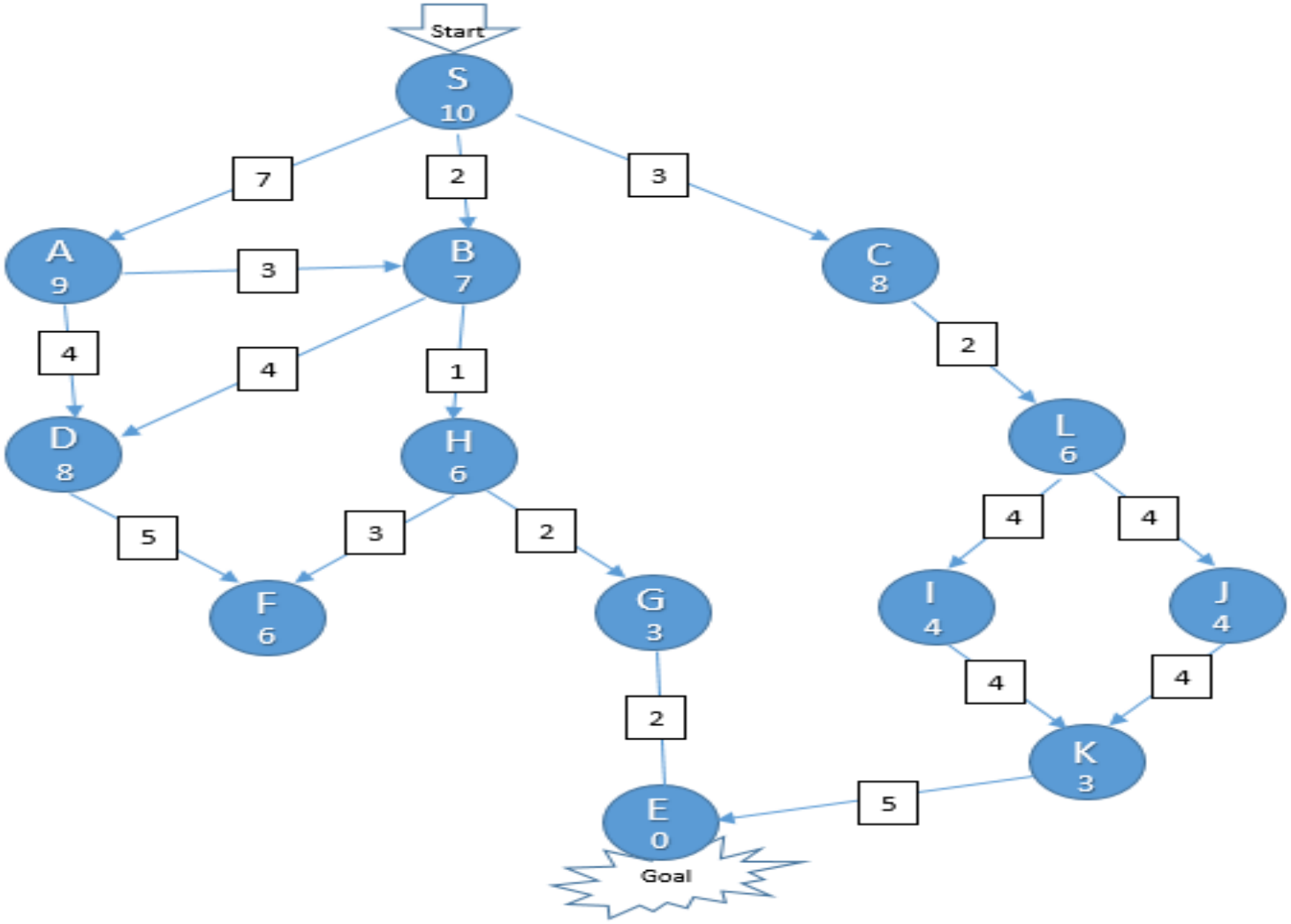
PROPERTIES OF BEST FIRST SEARCH

- It may **get stuck** in an infinite branch that doesn't contain the goal .
- It does **not guarantee** to find the shortest path solution .

Memory requirement :

- **In best case :** as depth first search.
- **In average case :** between depth and breadth.
- **In worst case :** as breadth first search

PROBLEM


Start
S
10
7
2
3
A
9
3
B
7
C
8
4
4
1
2
D
8
H
6
L
6
5
3
2
4
4
F
6
G
3
I
4
J
4
4
4
2
K
3
5
E
0
Goal

GREEDY BFS

GREEDY BEST FIRST SEARCH

- Greedy best-first search expands the node that appears to be closest to goal.
- **Greedy best-first search is a best first search but uses heuristic estimate h(n) rather than cost function. It** sorts by the **cost of getting to the goal from that state**
- **Evaluation function f(n) = h(n) (heuristic) = estimate of cost from n to goal**

E.g., hSLD(n) = straight-line distance from n to Bucharest

- Greedy best-first search algorithm always selects the path which appears best at that moment.

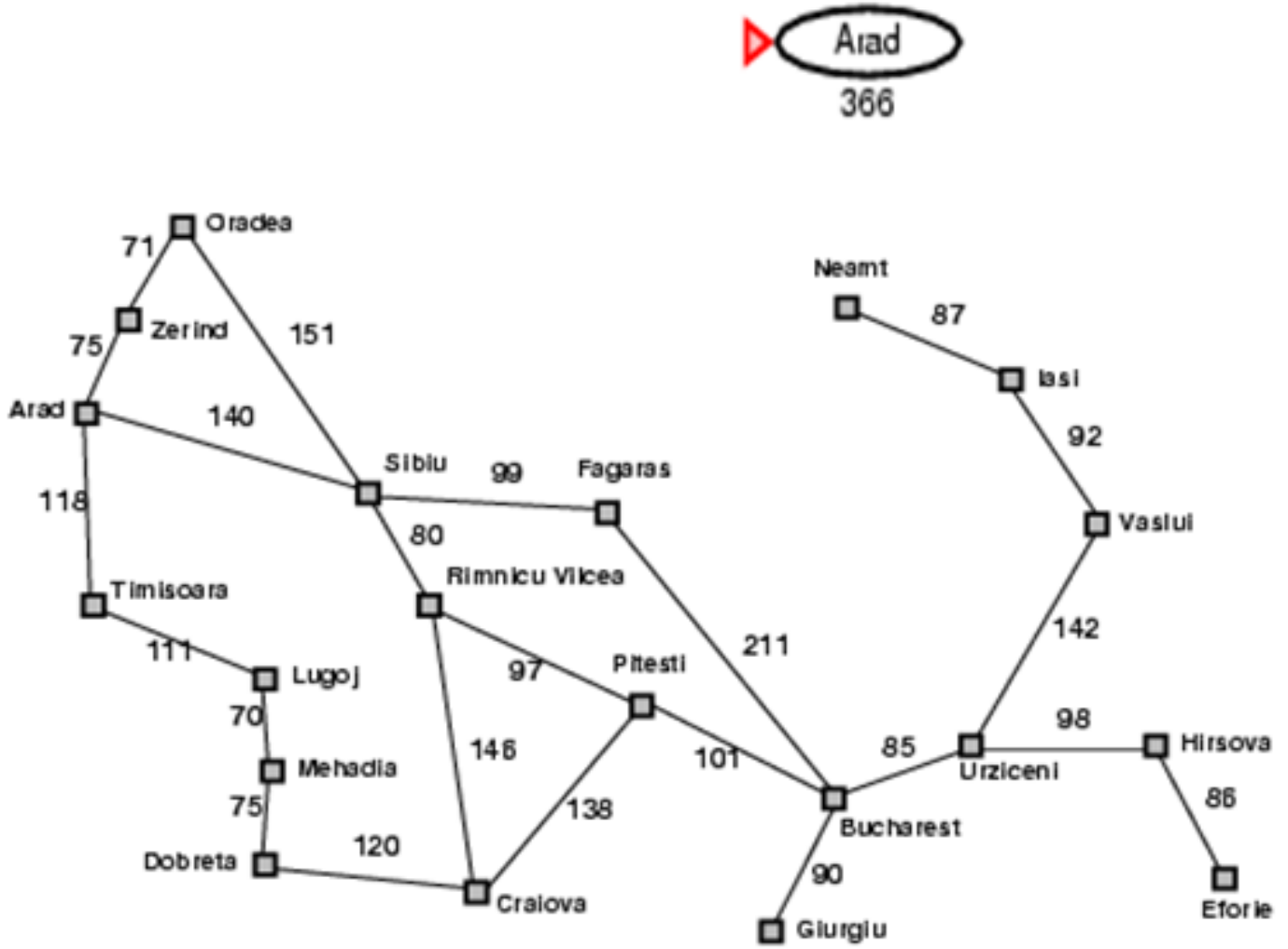
PROBLEM

- It doesn't take account of the cost so far like best first instead uses heuristic function, so it **isn't optimal**, and can **wander into dead-ends**, like depth-first search.
- In most domains, we also don't know the cost of getting to the goal from a state. So we have to guess, using a heuristic evaluation function.
    - If we knew how far we were from the goal state we wouldn’t need to search for it!

GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier queue:

Arad 366



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

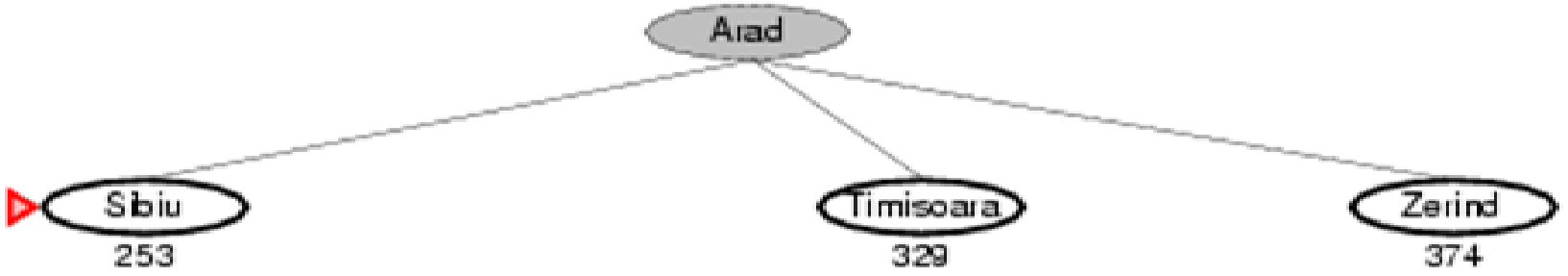
GREEDY BEST-FIRST SEARCH EXAMPLE

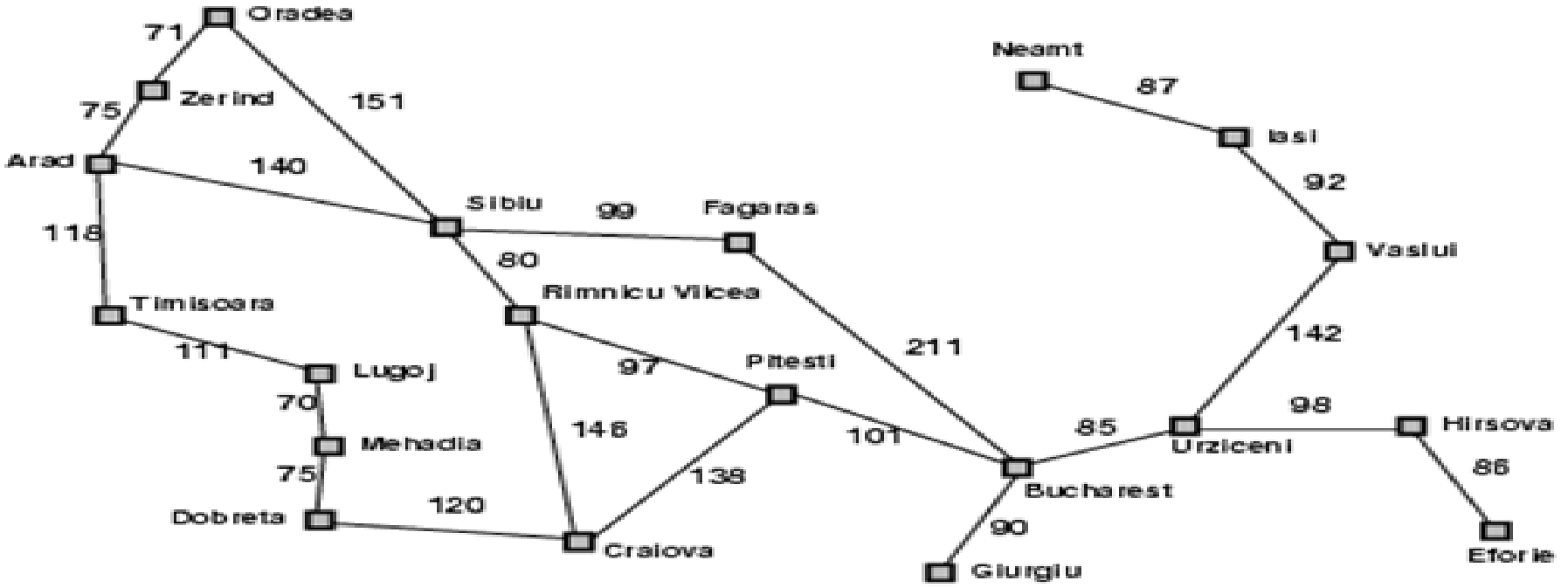
Frontier queue:

Sibiu 253

Timisoara 329

Zerind 374





Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier queue:

Fagaras 176

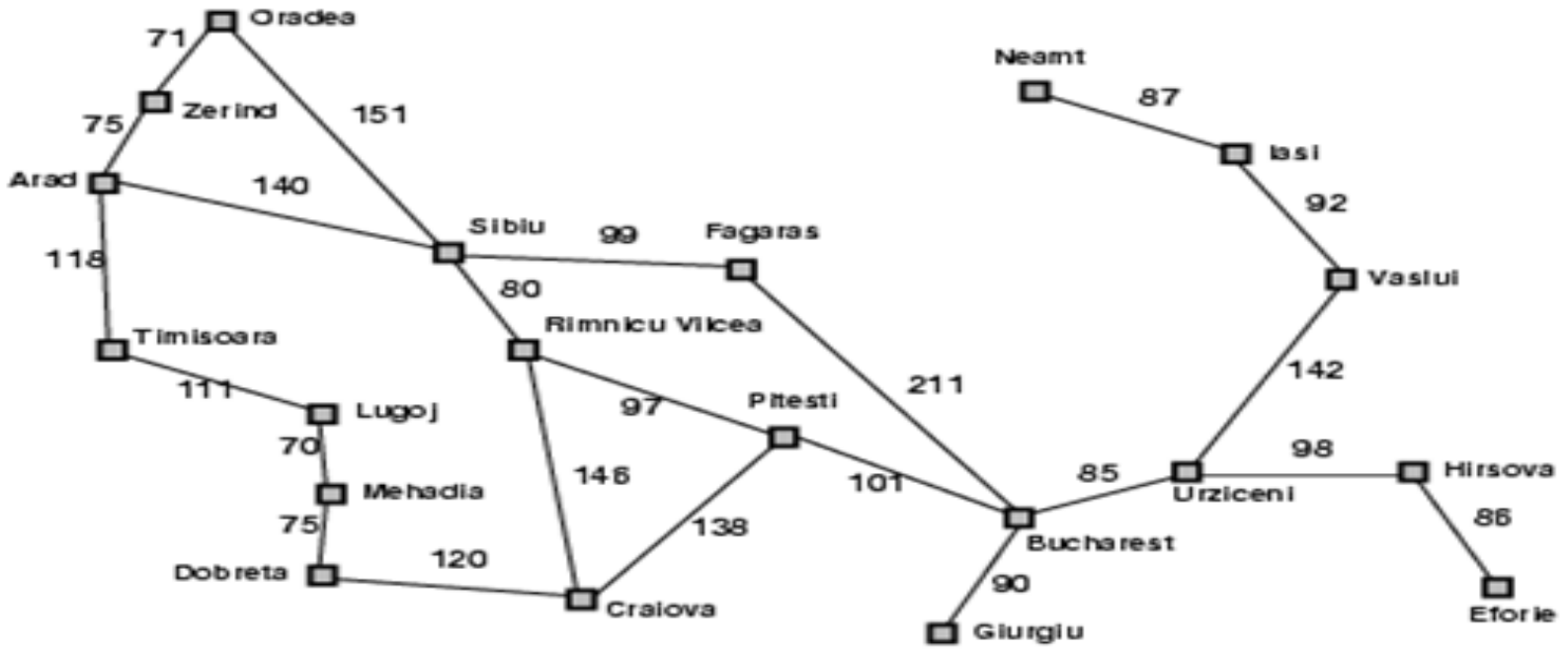
Rimnicu Vilcea 193

Timisoara 329

Arad 366

Zerind 374

Oradea 380



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier queue:

Bucharest 0

Rimnicu Vilcea 193

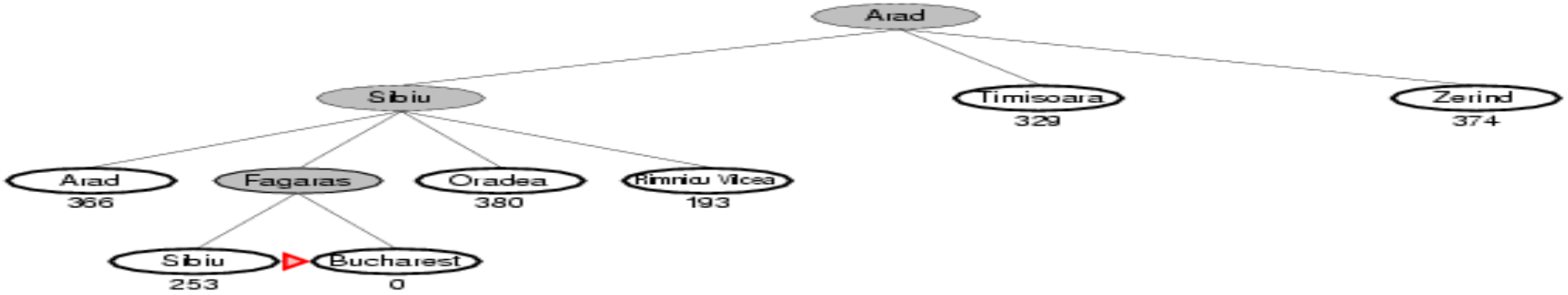
Sibiu 253

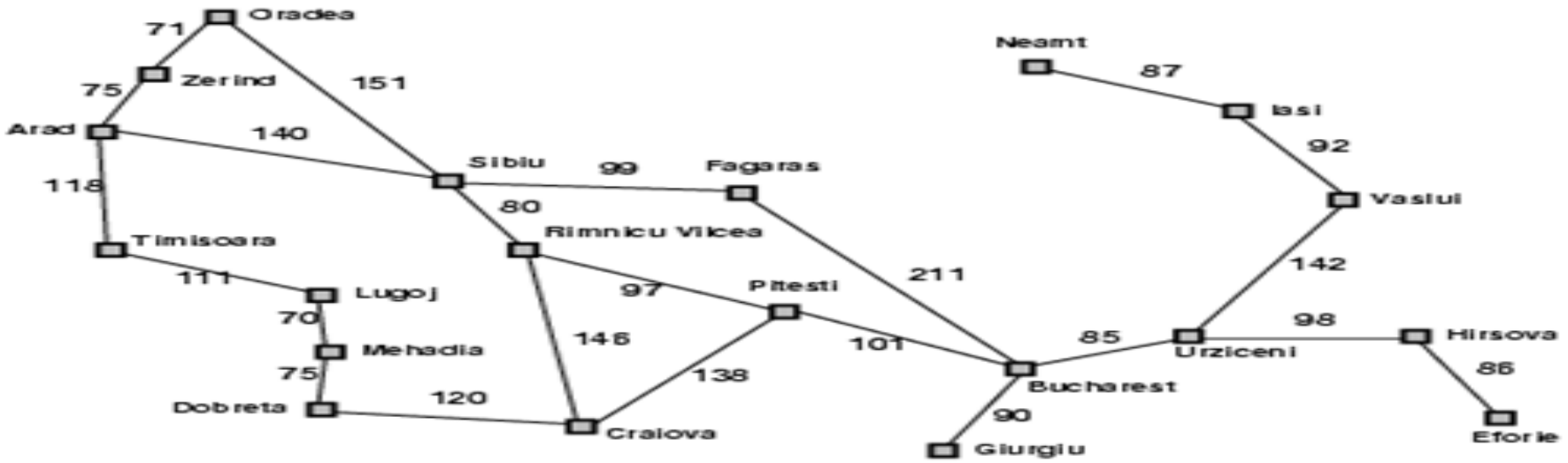
Timisoara 329

Arad 366

Zerind 374

Oradea 380

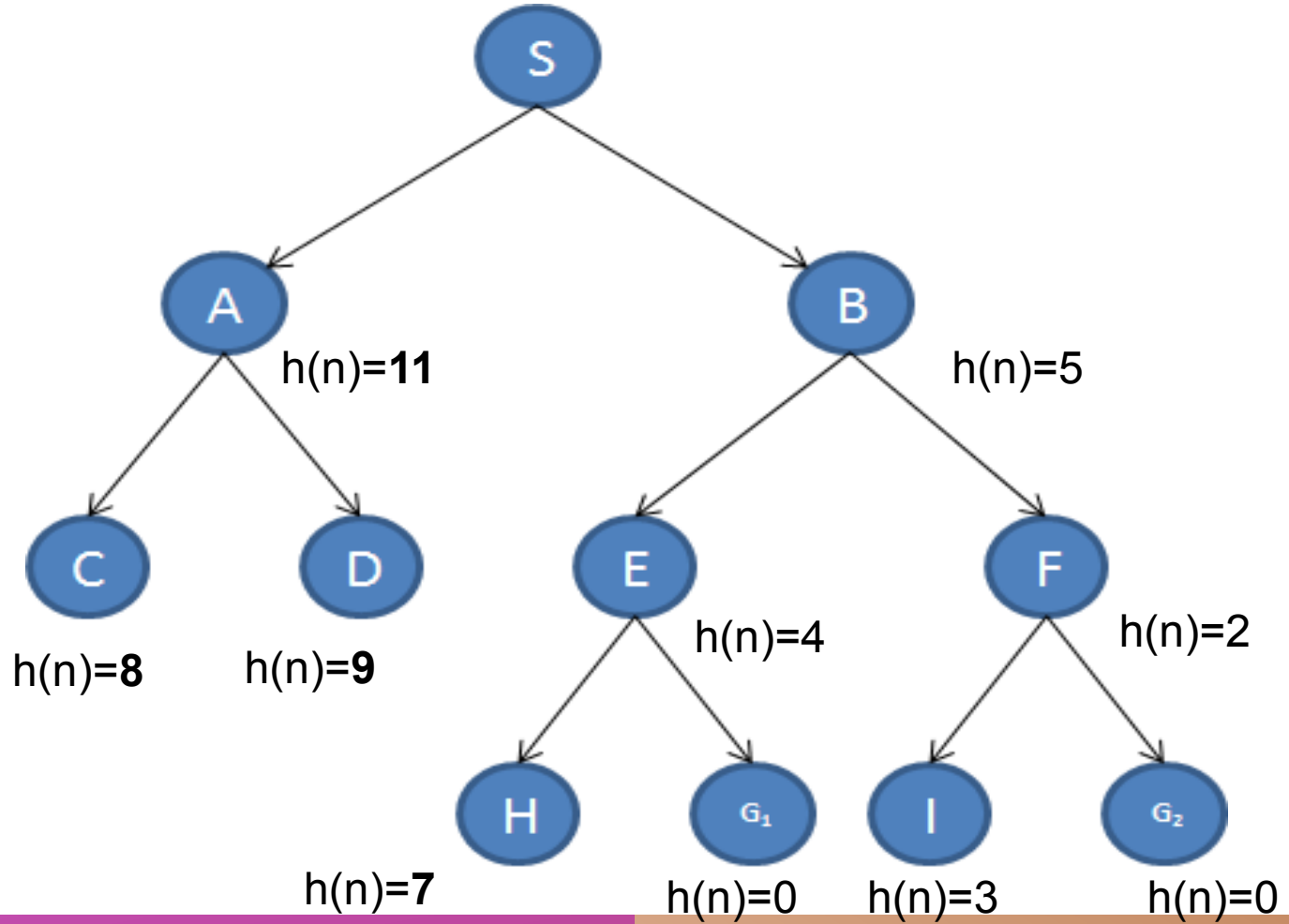




Straight-line distance to Bucharest

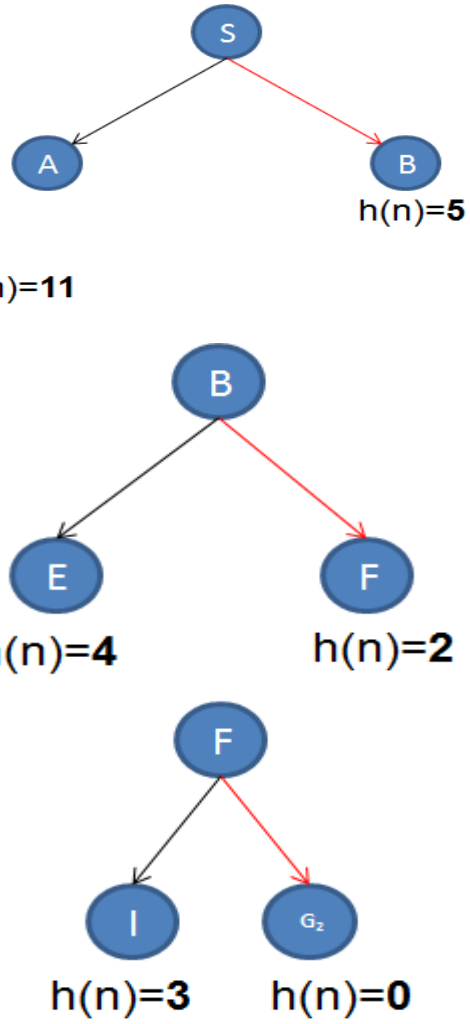
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

GREEDY BFS EXAMPLE



node	h(n)
A	11
B	5
C	9
D	8
E	4
F	2
H	7
I	3

GREEDY BFS SOLUTION



open=[S]; closed=[ ]

open=[B_5 , A_{11}]; closed=[S]

open=[F_2 , E_4, A_{11}]; closed=[S , B]

open=[$G2_0$, I_3 , E_4 , A_{11}]; closed=[S , B , F]

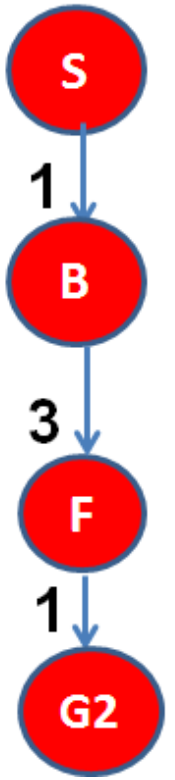
open=[I_3 , E_4 , A_{11}]; closed=[S , B , F , G2]

Solution:

S , B , F , G_2

Cost = 1+3+1=**5**

Search Path



PROPERTIES OF GREEDY BEST-FIRST SEARCH

Complete?

- No – can get stuck in loops,

Time?

- O(bm), but a good heuristic can give dramatic improvement

Space?

- O(bm) -- keeps all nodes in memory

Optimal?

- No
- **Obtain best solution than best-first. But not guaranteed the optimum solution**

A*SEARCH

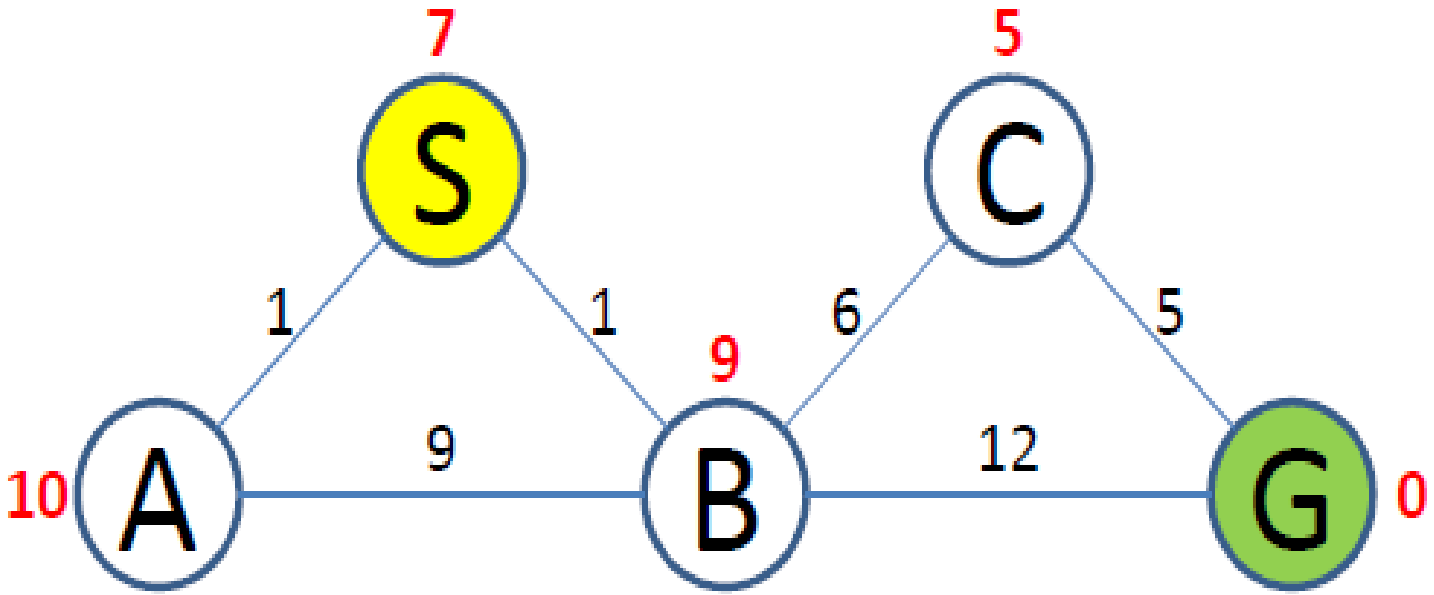
A* SEARCH ALGORITHM

- GBFS uses minimal estimated cost h(n) to goal node reduce search cost considerably. Unfortunately, it is neither optimal nor complete cost to the goal. Uniform-cost search, uses minimal cost of the path so far, g(ri); it is optimal and complete, but can be very inefficient.
- A* combines the two evaluation functions. A* is the best search for minimizing the total estimated solution cost.
- **Idea:** avoid expanding paths that are already expensive
- **Evaluation function f(n) = g(n) + h(n)** // Equals to traveled distance plus predicted distance ahead.
  - **g(n)** = cost so far to reach n . g(n) is called the **backward cost**
  - **h(n)** = estimated cost from n to goal . h(n) is called the **forward cost**
  - **f(n)** = estimated total cost of path through n to goal

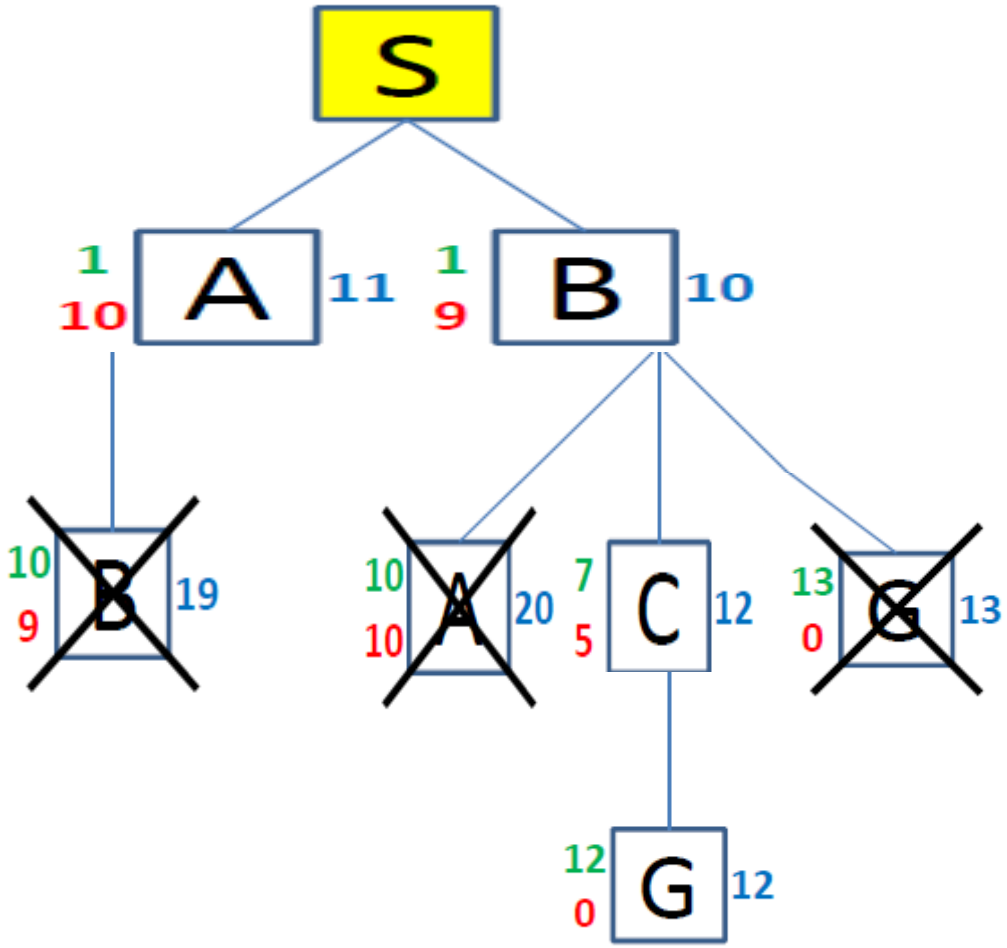
A* AND ADMISSIBILTY OF HEURISTIC FUNCTION

- The choice of an appropriate heuristic evaluation function, h(n), is still crucial to the behavior of this algorithm.
- If its guaranteed that there is **no overestimate** for **h(n) ie** If **h(n) <= h*(n) for all n**, then **f(n) = h(n) + g(n)** gives the **shortest path**, where **h(n)** is theis the **heuristic estimate of the cost of getting to a goal node from n** and **h*(n)** is the **cost of an optimal path from n to a goal node**. If **h(n) <= h*(n)** then **h(n)** is called ***Admissible heuristic***
- Choose a heuristic evaluation function, h(n) which is as close as possible to the actual cost of getting to a goal state. Ie, choose a function h(n) which never overestimates the actual cost of getting to the goal state.
- An **admissible heuristic never overestimates the cost to reach the goal**, i.e., it is optimistic
- Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs
- Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe

EXERCISE-1 ON A* ALGORITHM



SOLUTION



QUEUE:
S7
B10 A11
A11 C12 G13
C12 G12 G13 B19 A20
G12 G13 B19 A20

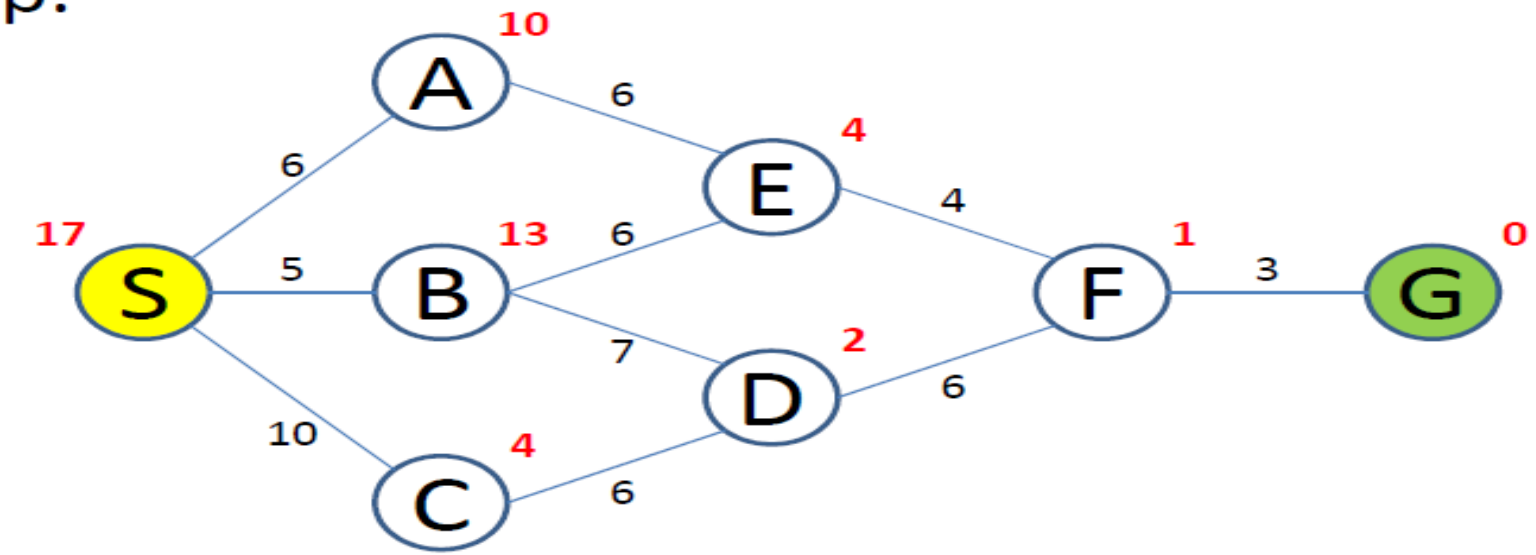
- **Frontier: Priority Queue(with priority to least F(n))**
- **Solution path found is S B C G, cost 12**
- **Number of nodes expanded (including goal node) = 4**

THINGS TO BE TAKEN CARE WHILE IMPLEMENTING A*

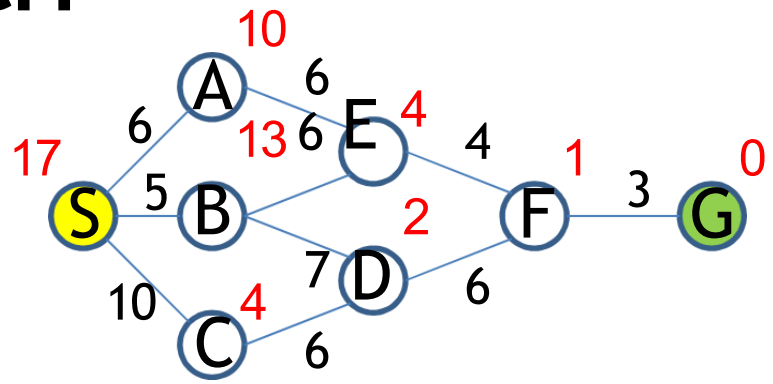
1. Push a node n which is already visited to fringe if the f(n) value is lessser than previous f(n) value. Else optimality wont be achieved.(else G19 since E15 cant be pushed in)ie We must not exclude states we
   have seen before, but update their information when their information when the new path is better.
2. Always check the expanded list and see if the node is visited and expand a node if F(n) value is less(refer case of D in coming)
3. If its undirected graph all possible paths should be explored out even explored nodes, but exclude the predecessors in a branch. Else infinite path will arise(completeness can't be achieved)

EXERCISE-2 ON A* ALGORITHM

Perform the A* Algorithm on the following figure. Explicitly write down the queue at each step.



A* Search

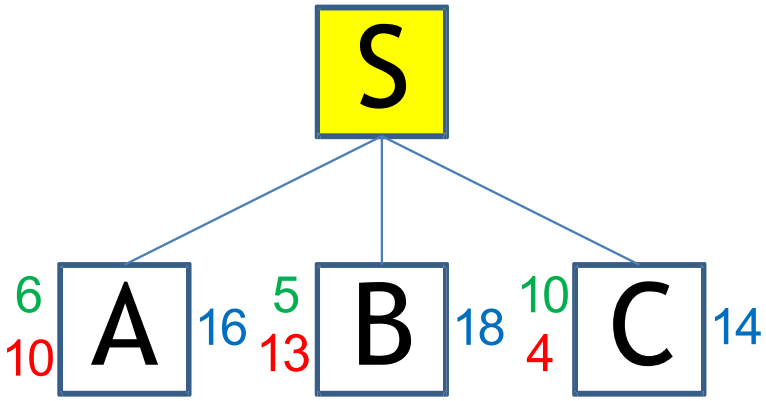


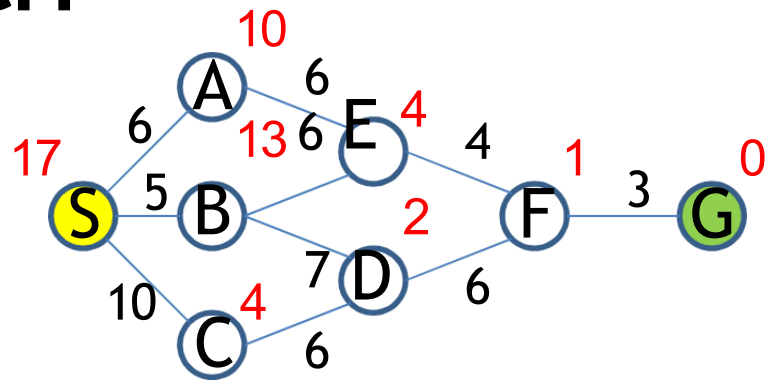
<u>QUEUE:</u>

S^{17}

Expanded Node	Fringe Queue
--	S^{17}

A* Search



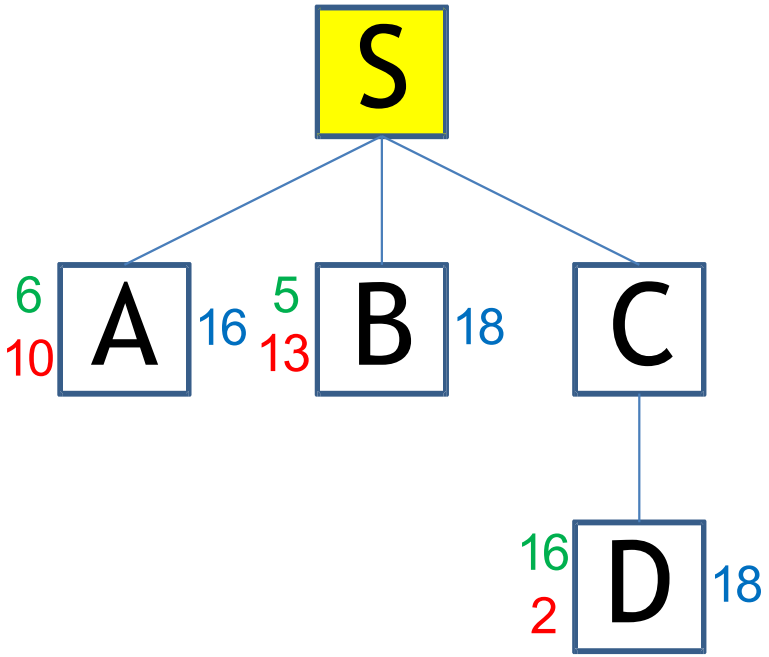


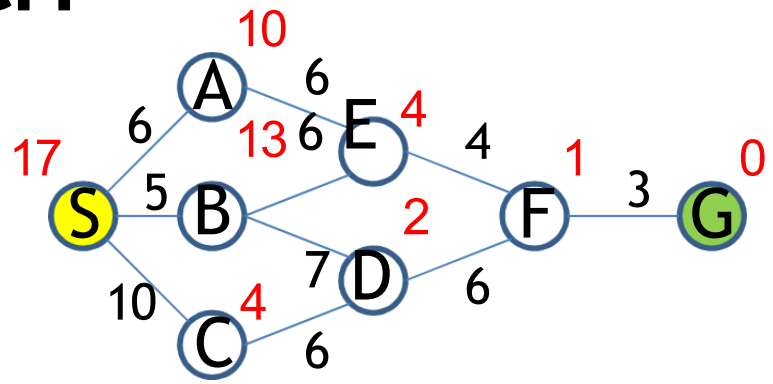
QUEUE:

S17 C14 A16 B18

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$

A* Search



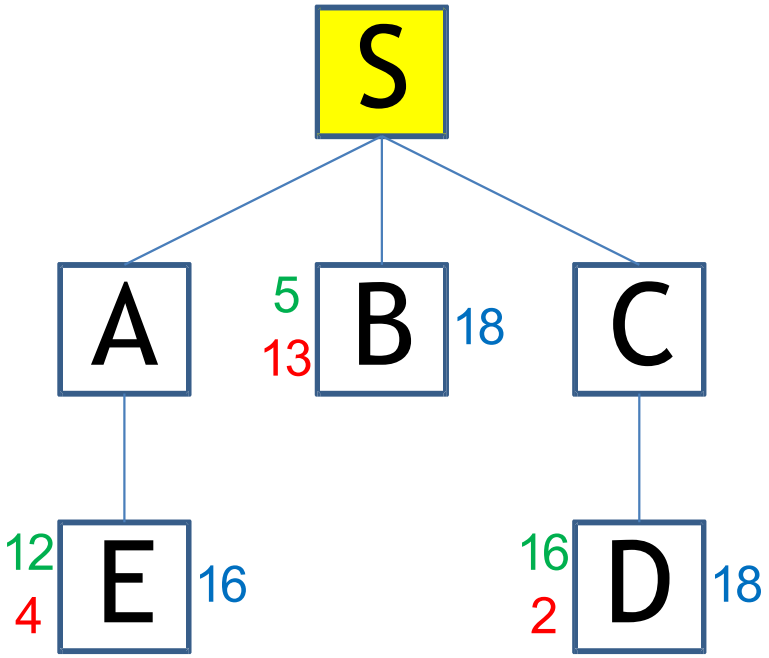


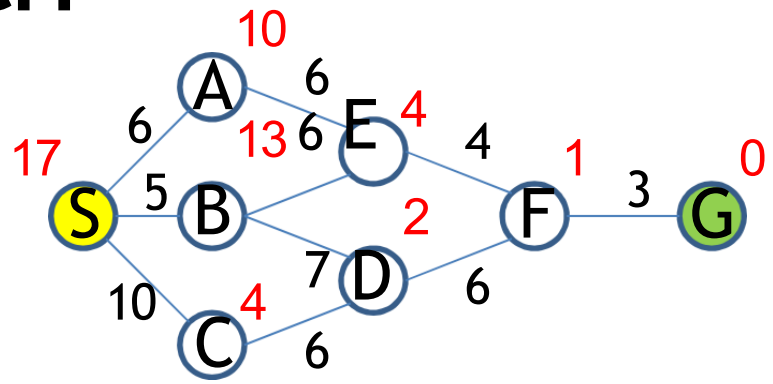
QUEUE:

$S^{17}\ C^{14}\ A^{16}\ B^{18}\ D^{18}$

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14}\ A^{16}\ B^{18}$
C^{14}	$A^{16}\ B^{18}\ D^{18}$

A* Search



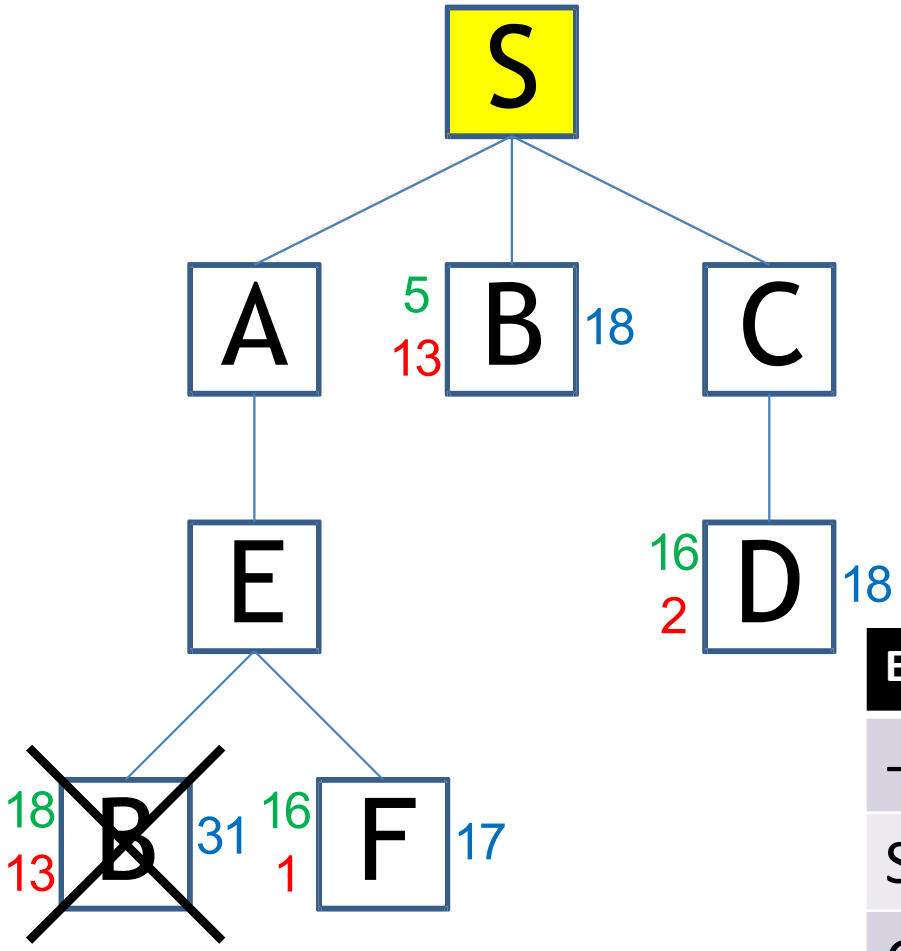


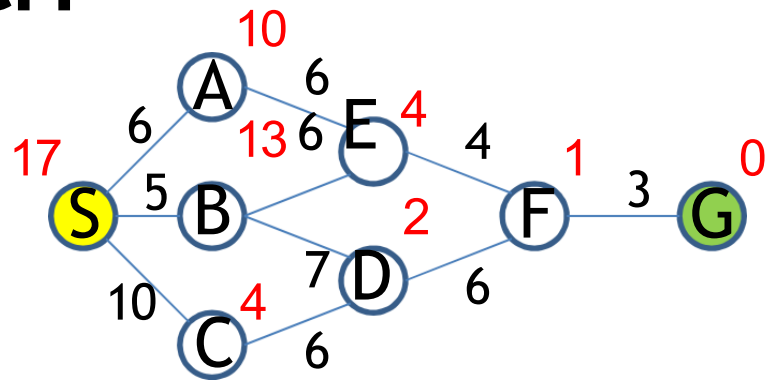
QUEUE:

S^{17} C^{14} A^{16} E^{16} B^{18} D^{18}

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	C^{14} A^{16} B^{18}
C^{14}	A^{16} B^{18} D^{18}
A^{16}	E^{16} B^{18} D^{18}

A* Search



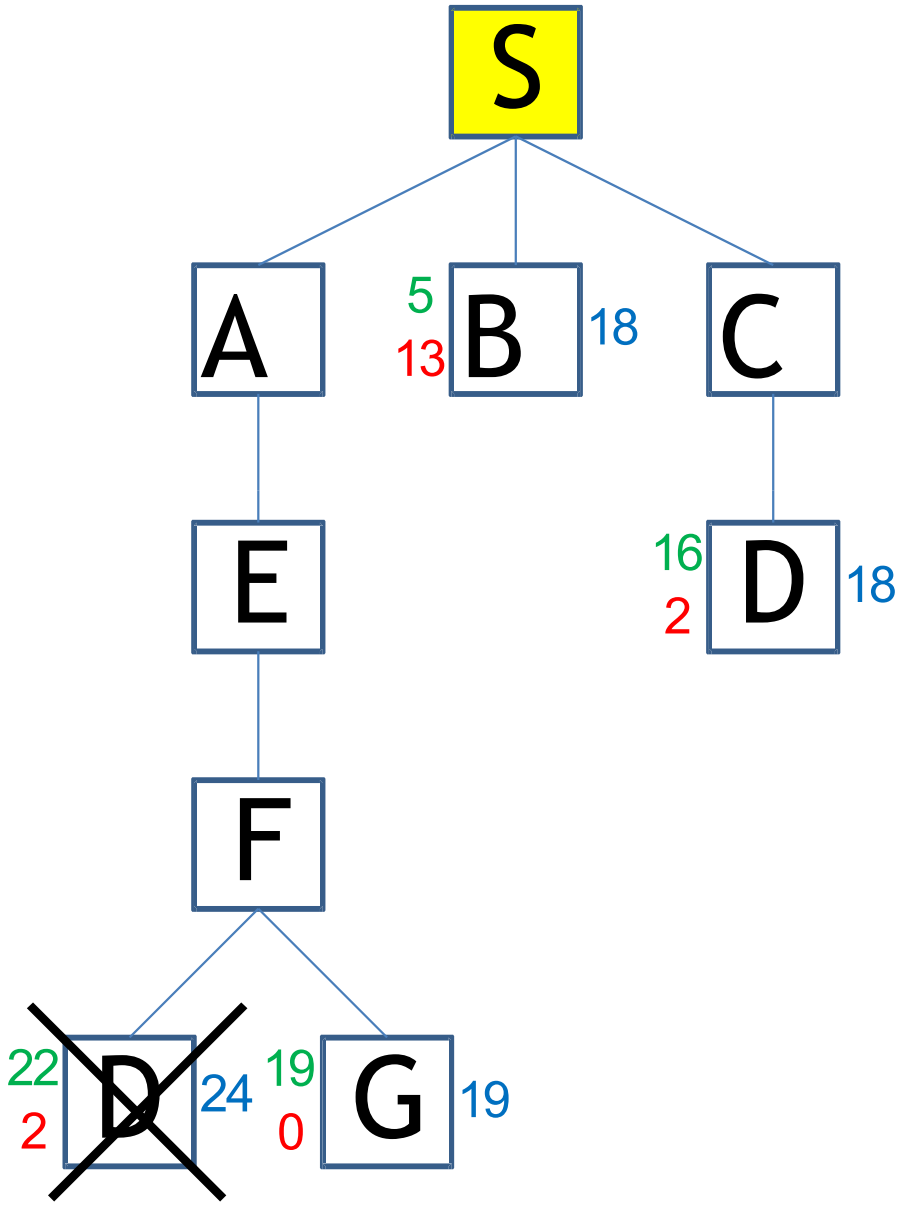


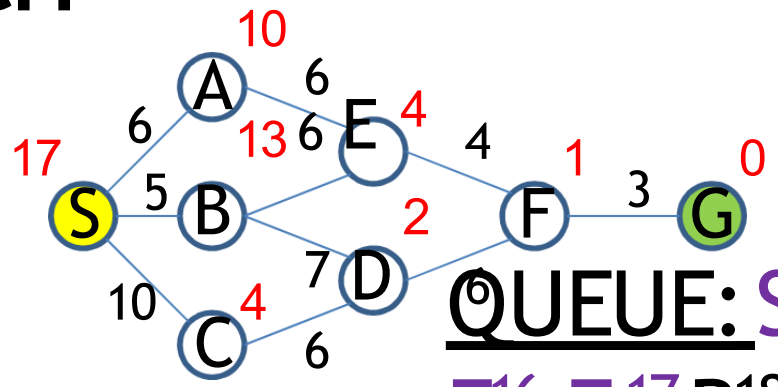
QUEUE:

S^{17} C^{14} A^{16} E^{16} F^{17} B^{18} D^{18} B^{31}

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	C^{14} A^{16} B^{18}
C^{14}	A^{16} B^{18} D^{18}
A^{16}	E^{16} B^{18} D^{18}
E^{16}	F^{17} B^{18} D^{18}

A* Search

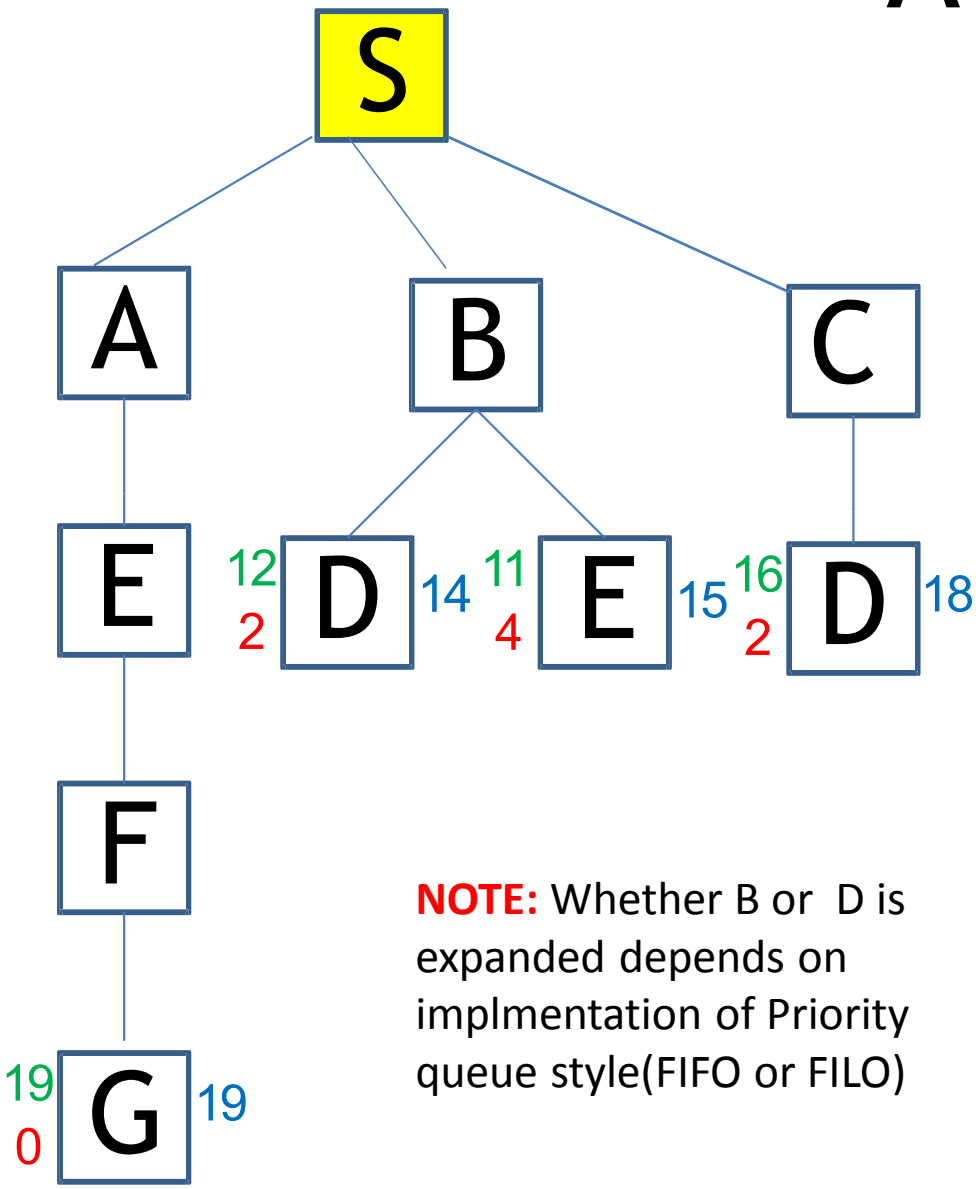


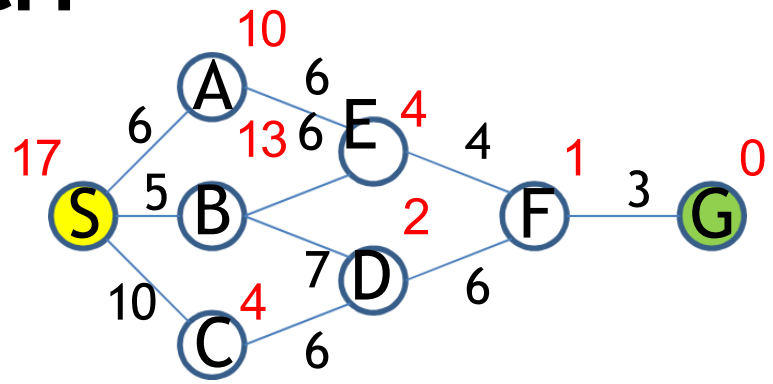


QUEUE: S^{17} C^{14} A^{16} E^{16} F^{17} B^{18} D^{18} G^{19} D^{24}

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	C^{14} A^{16} B^{18}
C^{14}	A^{16} B^{18} D^{18}
A^{16}	E^{16} B^{18} D^{18}
E^{16}	F^{17} B^{18} D^{18}
F^{17}	B^{18} D^{18} G^{19} D^{24}

A* Search



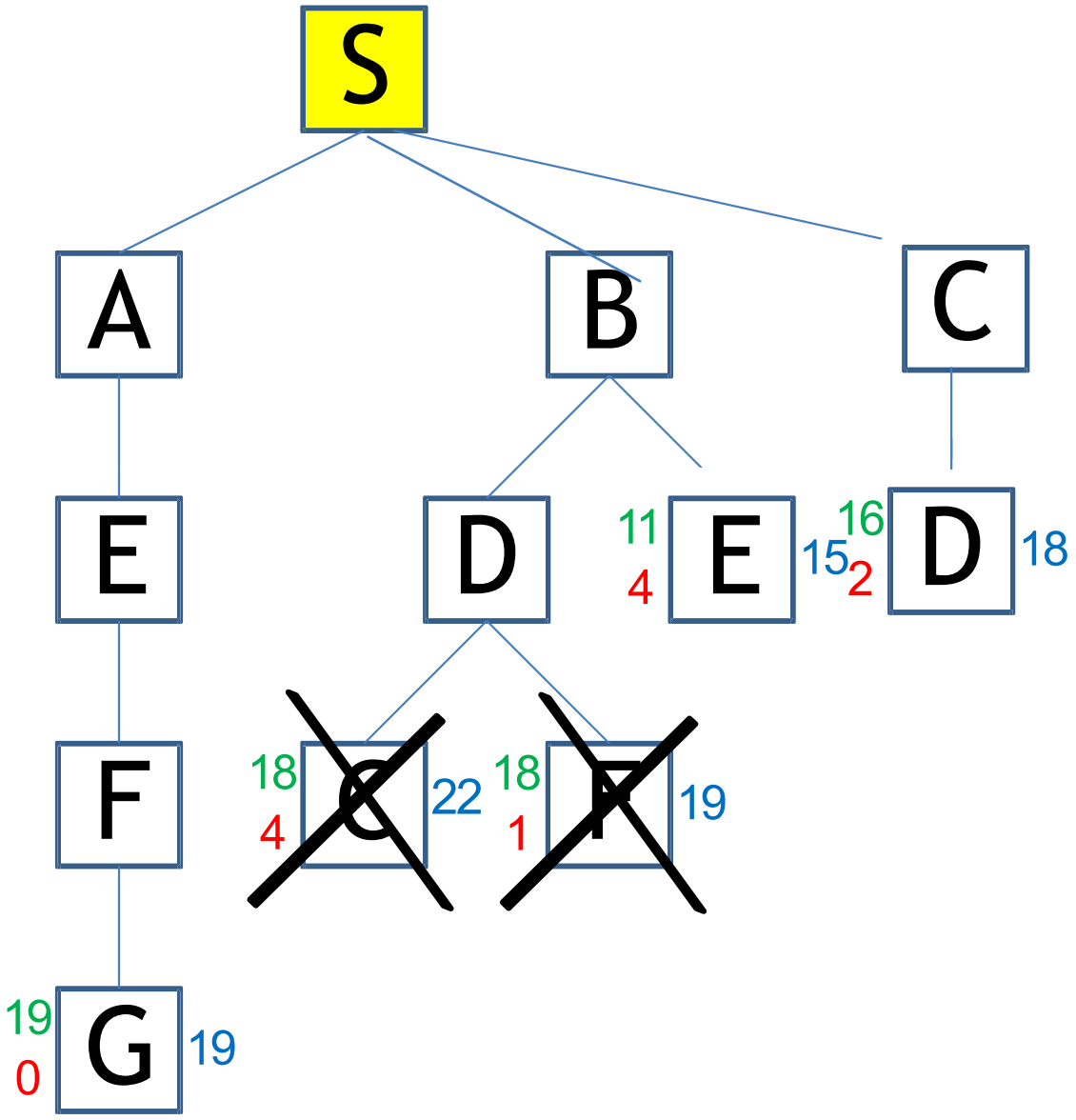


Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$
E^{16}	$F^{17} B^{18} D^{18}$
F^{17}	$B^{18} D^{18} G^{19}$
B^{18}	$D^{14} E^{15} D^{18} G^{19}$

NOTE: Whether B or D is expanded depends on implmentation of Priority queue style(FIFO or FILO)

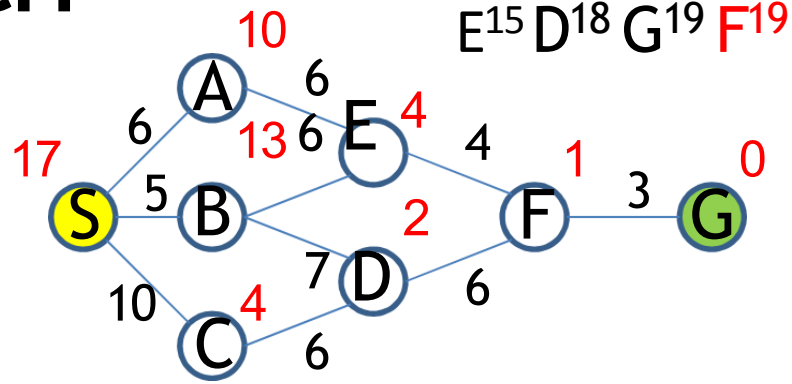
QUEUE: S^{1}
$^{7}C^{14} A^{16} E^{16}$
$F^{17} B^{18} D^{14}$
$E^{15} D^{18} G^{19}$

A* Search



QUEUE:

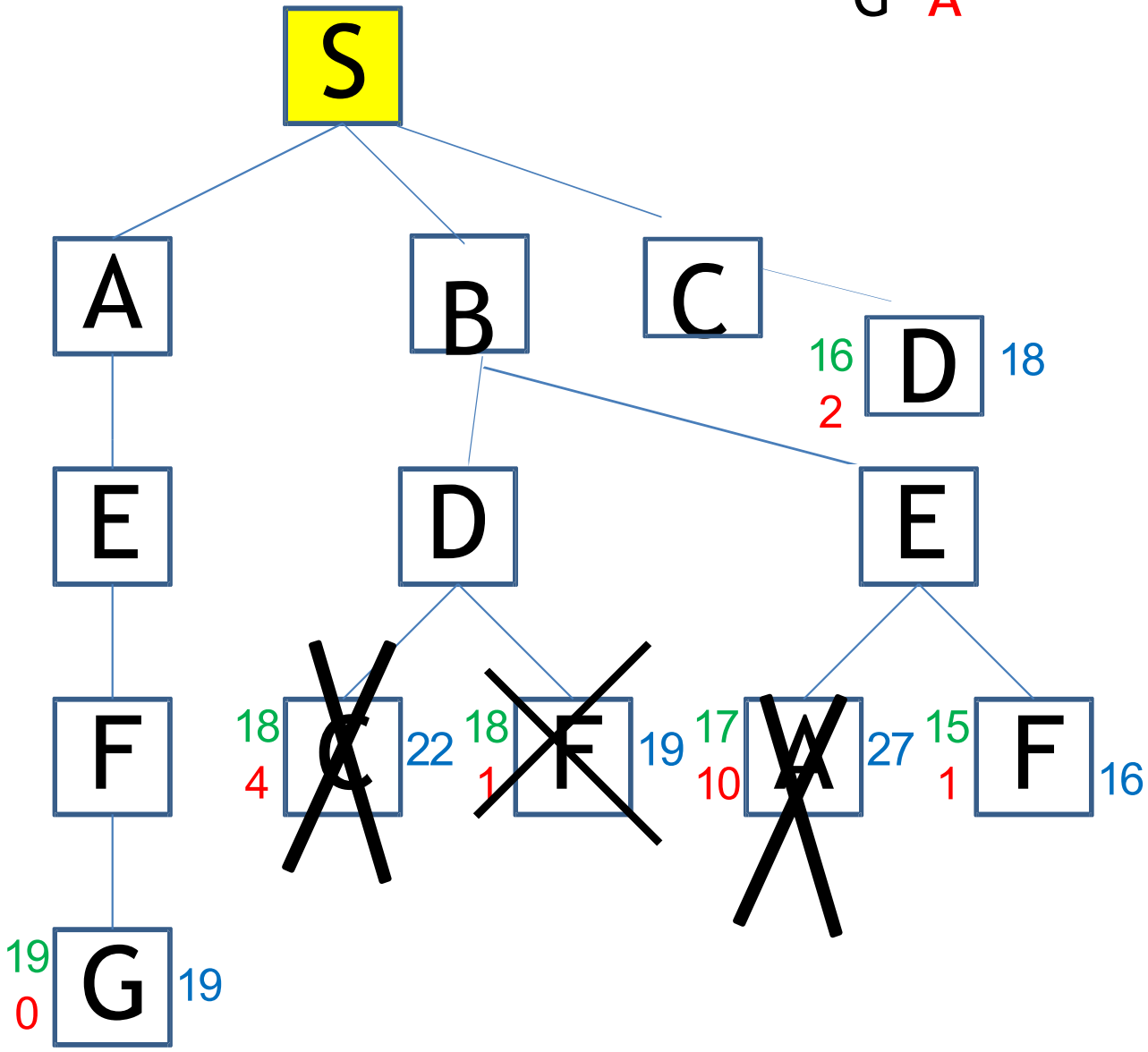
S^{17} C^{14} A^{16} E^{16} F^{17} B^{18} D^{14}
E^{15} D^{18} G^{19} F^{19} C^{22}

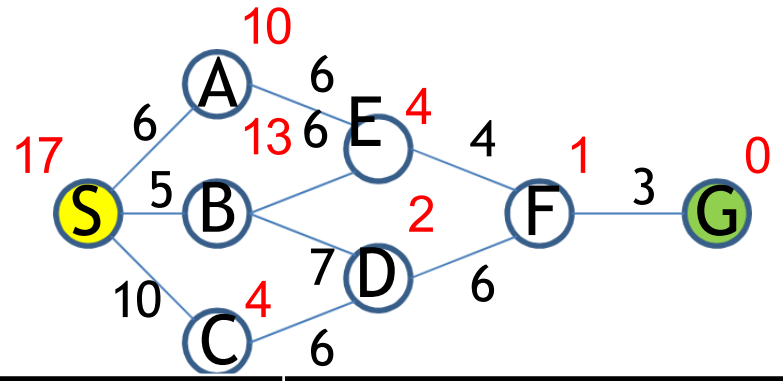


Expanded Node	Fringe Queue
--	S^{17}
S^{17}	C^{14} A^{16} B^{18}
C^{14}	A^{16} B^{18} D^{18}
A^{16}	E^{16} B^{18} D^{18}
E^{16}	F^{17} B^{18} D^{18}
F^{17}	B^{18} D^{18} G^{19}
B^{18}	D^{14} E^{15} D^{18} G^{19}
D^{14}	E^{15} D^{18} G^{19}

A* Search

QUEUE: S^{17} C^{14} A^{16} E^{1}
^{6} F^{17} B^{18} D^{14} E^{15} F^{16} D^{18}
G^{19} A^{27}

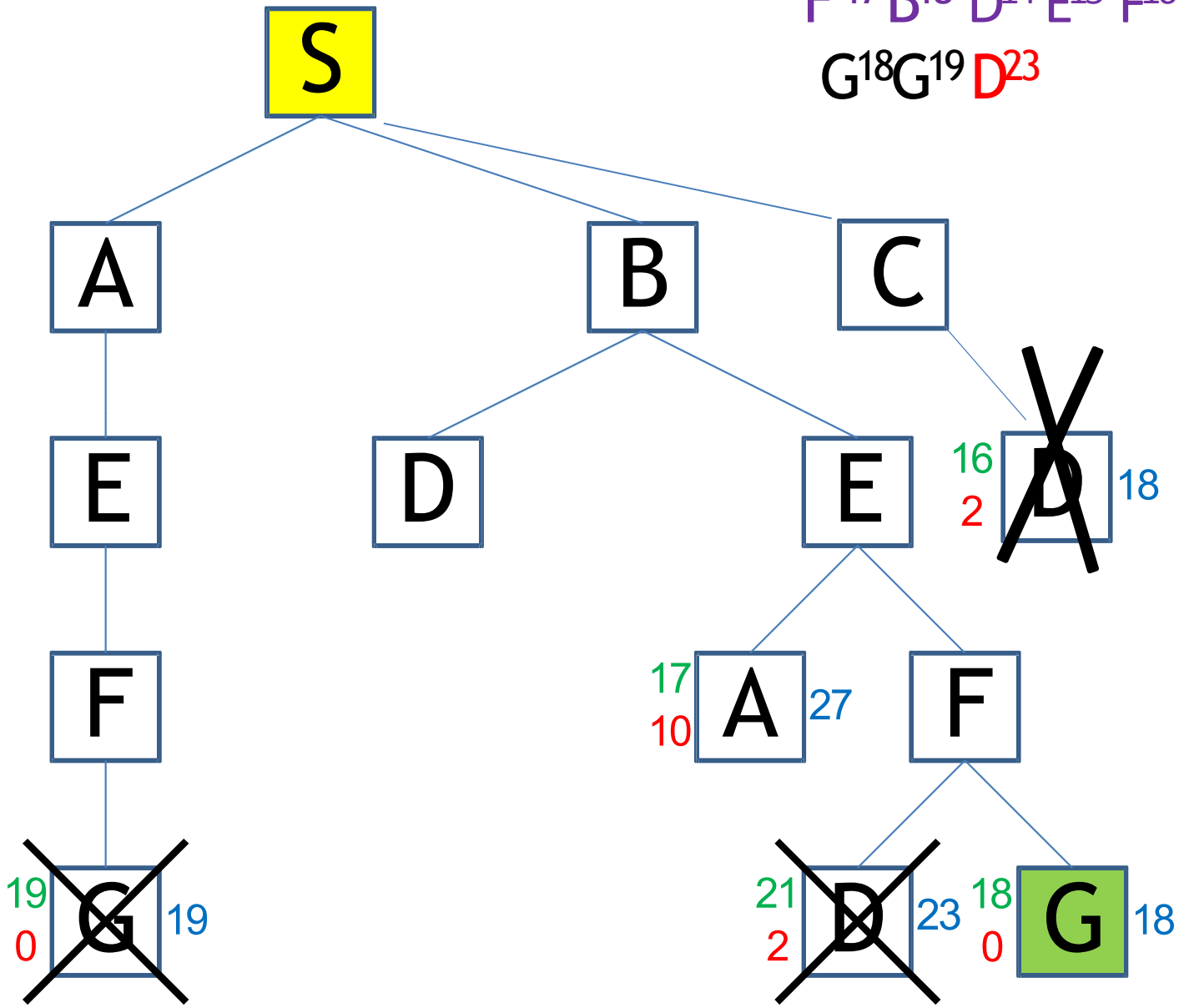


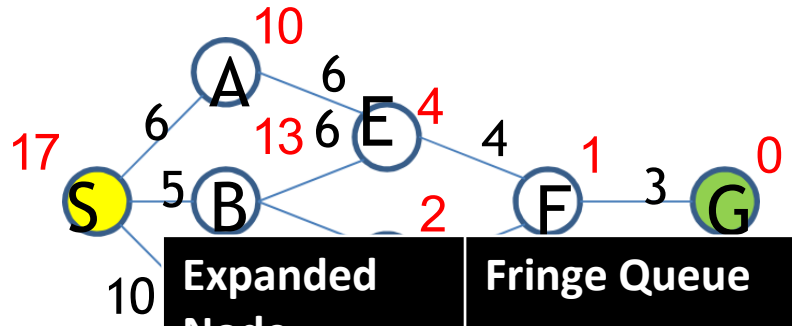


Expanded Node	Fringe Queue
--	S^{17}
S^{17}	C^{14} A^{16} B^{18}
C^{14}	A^{16} B^{18} D^{18}
A^{16}	E^{16} B^{18} D^{18}
E^{16}	F^{17} B^{18} D^{18}
F^{17}	B^{18} D^{18} G^{19}
B^{18}	D^{14} E^{15} D^{18} G^{19}
D^{14}	E^{15} D^{18} G^{19}
E^{15}	F^{16} D^{18} G^{19}

A* Search

QUEUE: S^{17} C^{14} A^{16} E^{16} F^{17} B^{18} D^{14} E^{15} F^{16} D^{18} G^{18} G^{19} D^{23}

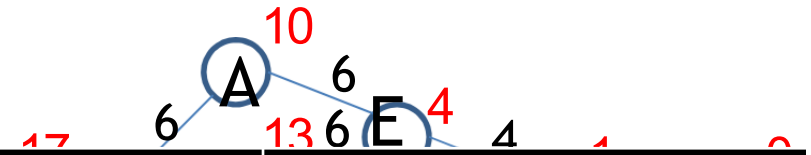


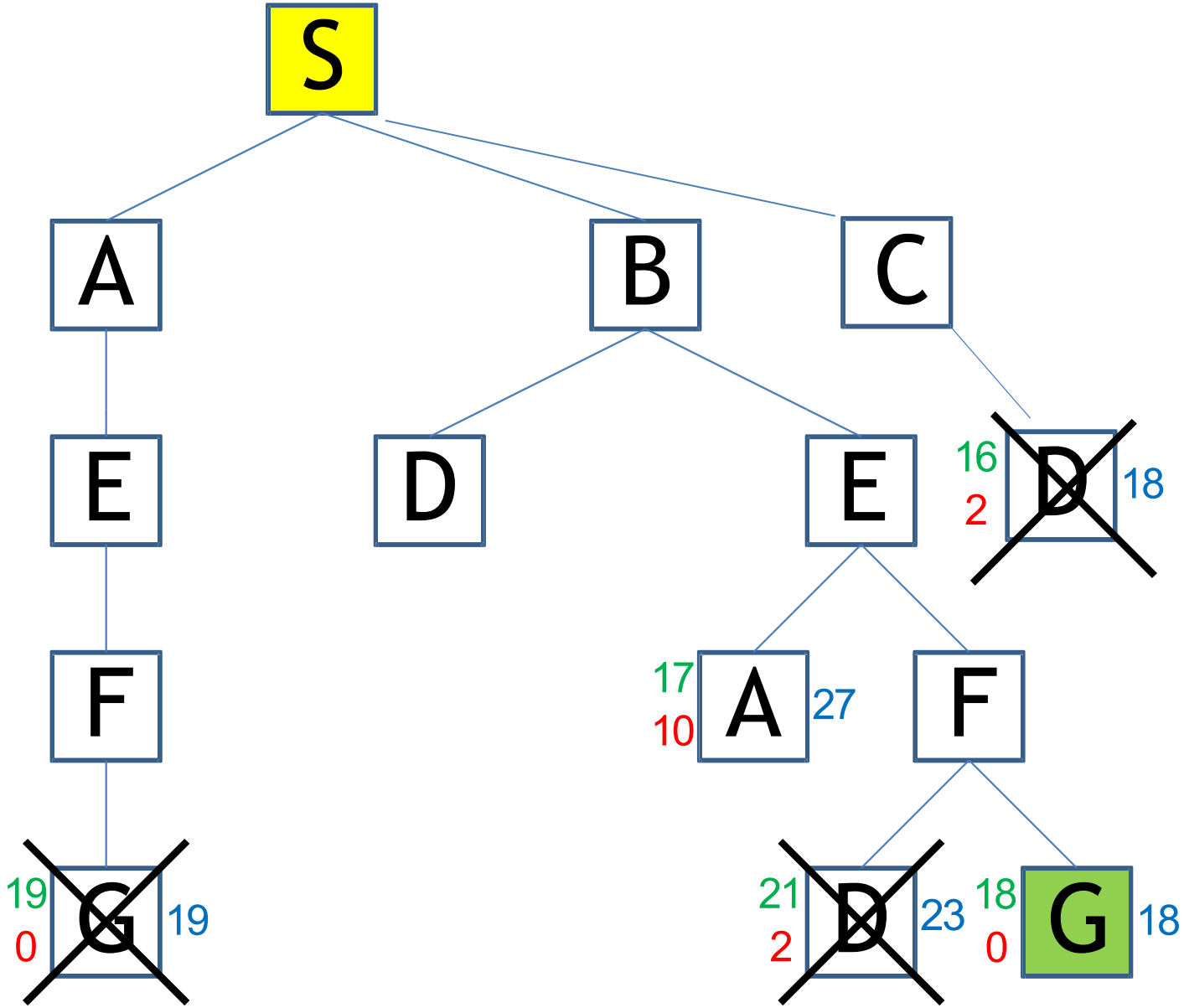


Expanded Node	Fringe Queue
--	S^{17}
S^{17}	C^{14} A^{16} B^{18}
C^{14}	A^{16} B^{18} D^{18}
A^{16}	E^{16} B^{18} D^{18}
E^{16}	F^{17} B^{18} D^{18}
F^{17}	B^{18} D^{18} G^{19}
B^{18}	D^{14} E^{15} D^{18} G^{19}
D^{14}	E^{15} D^{18} G^{19}
E^{15}	F^{16} D^{18} G^{19}
F^{16}	D^{18} G^{18} G^{19}

A* Search

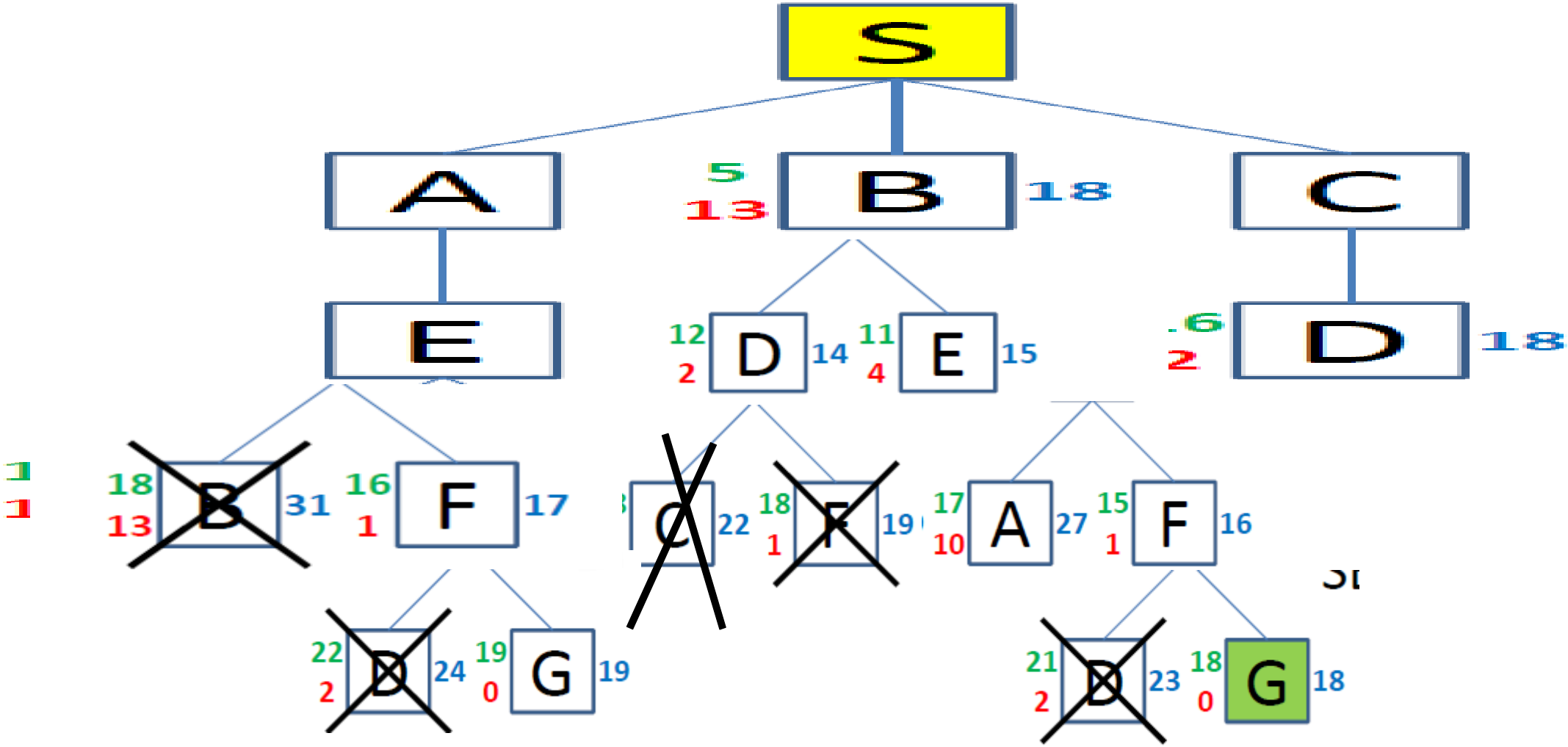
QUEUE: SBEF
G SAEFGSBDC
SBEFDSBEA





Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$
E^{16}	$F^{17} B^{18} D^{18}$
F^{17}	$B^{18} D^{18} G^{19}$
B^{18}	$D^{14} E^{15} D^{18} G^{19}$
D^{14}	$E^{15} D^{18} G^{19}$
E^{15}	$F^{16} D^{18} G^{19}$
F^{16}	$D^{18} G^{18} G^{19}$
G^{18}	

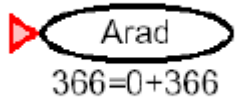
SOLUTION

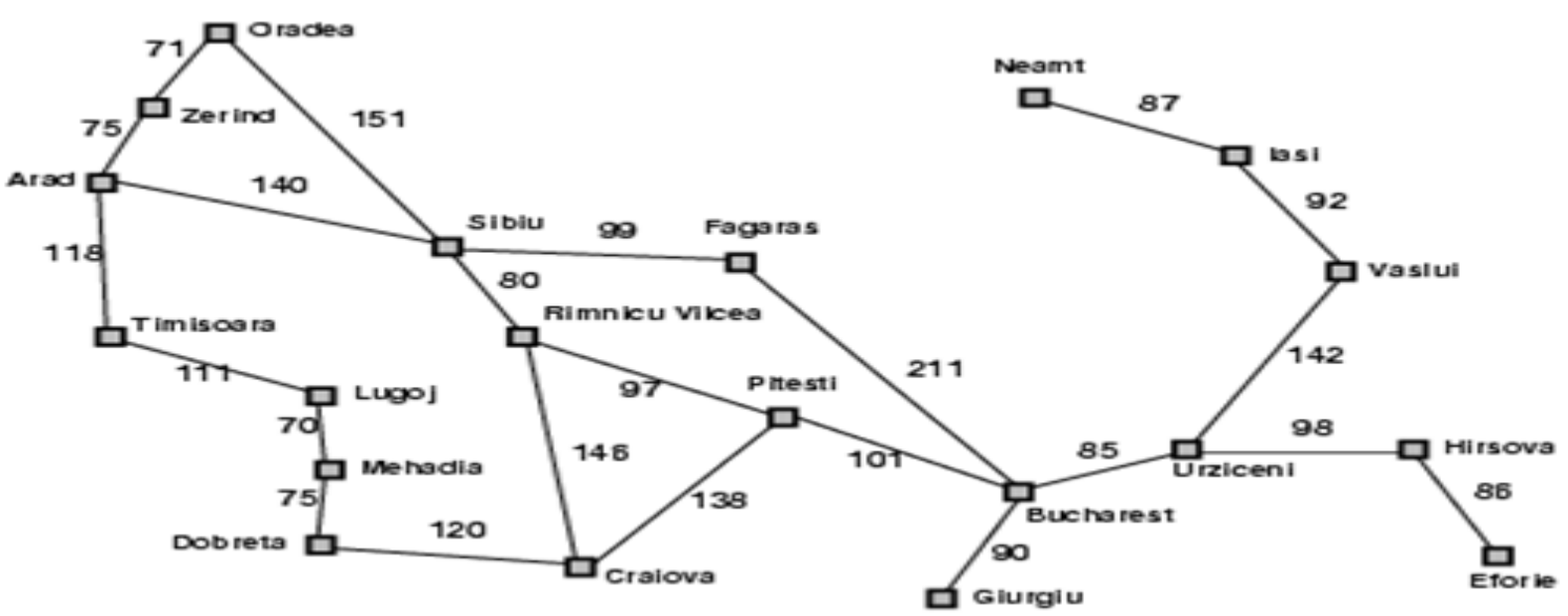

S
A
5
13
B
18
C
E
12
2
D
14
11
4
E
15
6
2
D
18
1
1
18
13
B
31
16
1
F
17
C
22
18
1
F
19
17
10
A
27
15
1
F
16
22
2
D
24
19
0
G
19
21
2
D
23
18
0
G
18

A* SEARCH EXAMPLE

Frontier queue:

Arad 366





Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

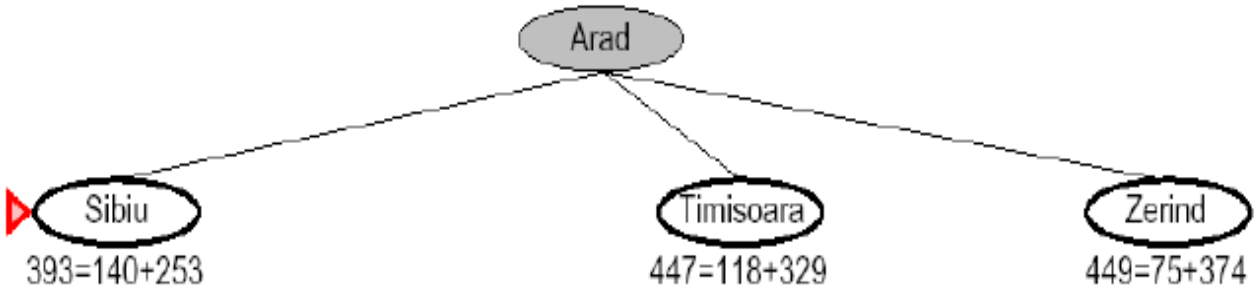
A* SEARCH EXAMPLE

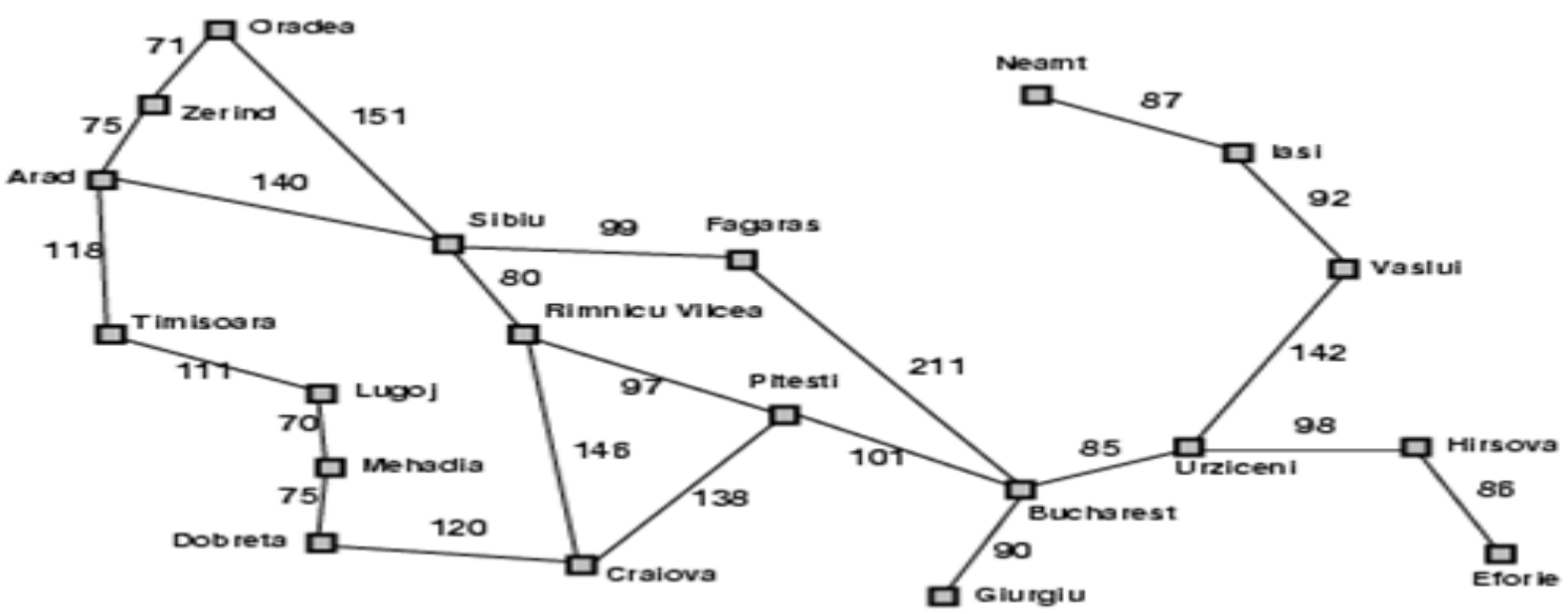
Frontier queue:

Sibiu 393

Timisoara 447

Zerind 449





Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* SEARCH EXAMPLE

Frontier queue:

Rimricu Vicea 413

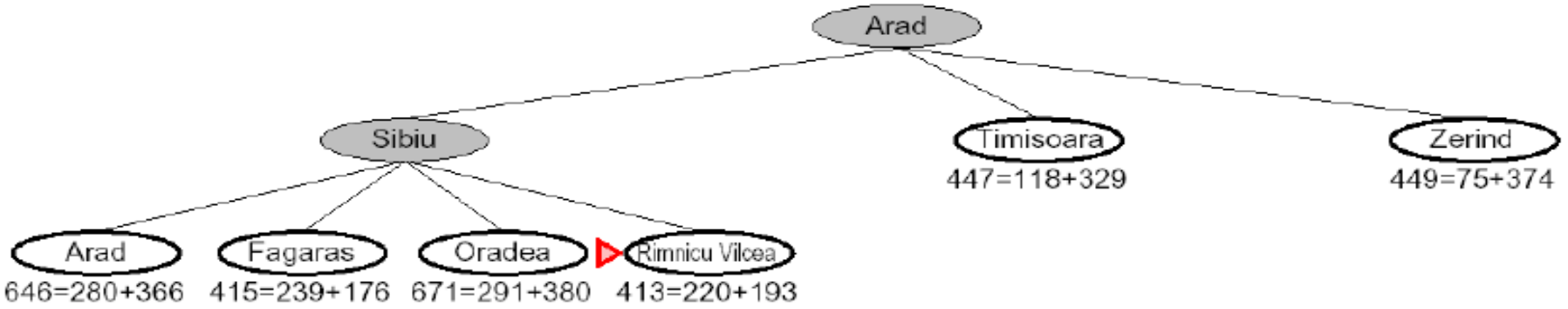
Fagaras 415

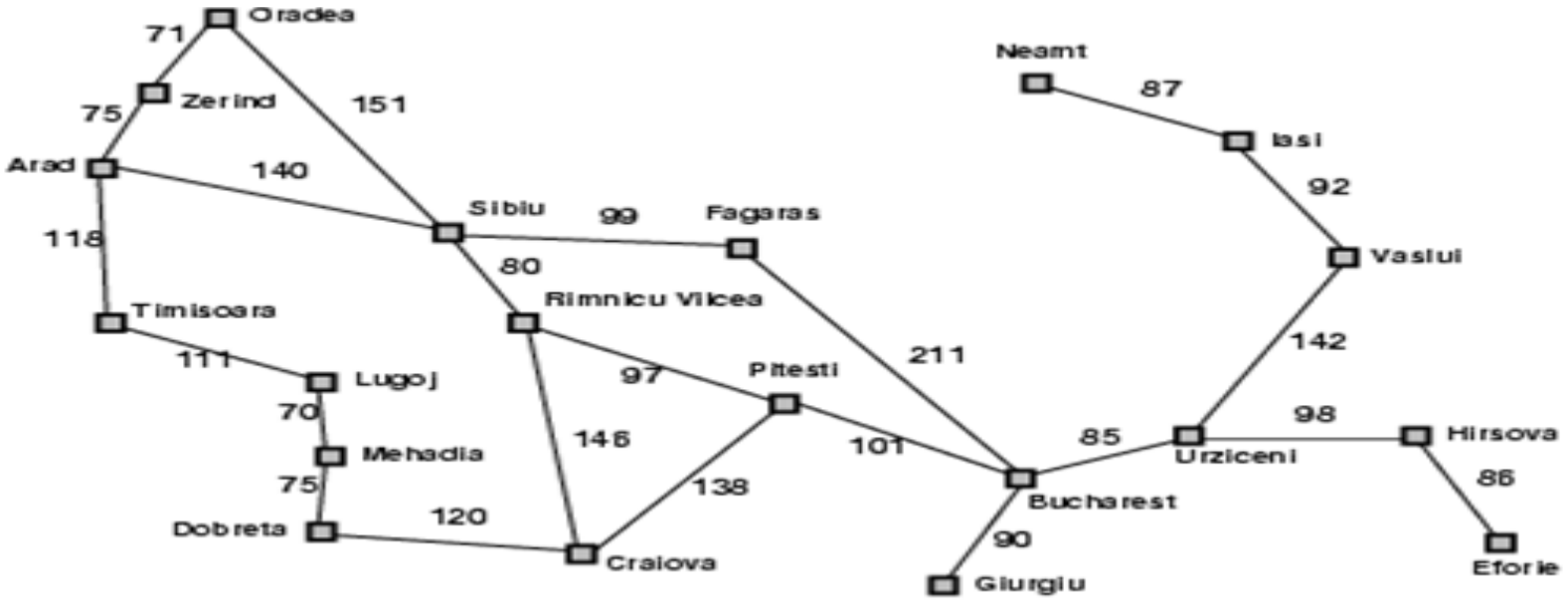
Timisoara 447

Zerind 449

Arad 646

Oradea 671

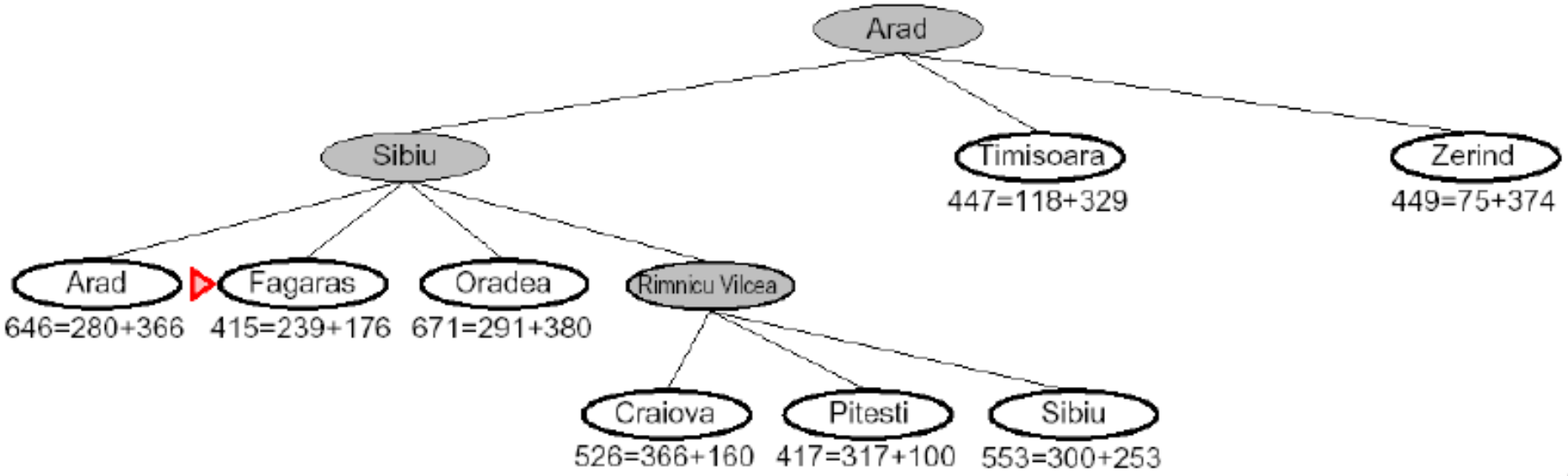


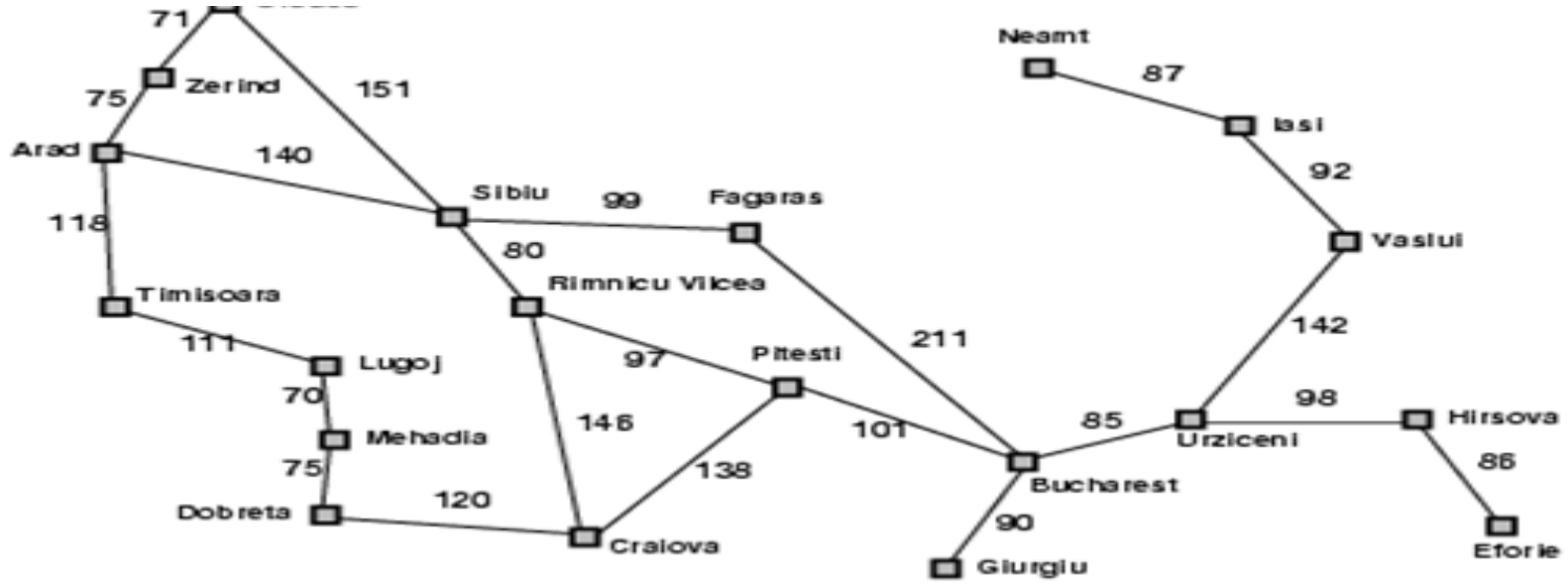


	istance
to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* SEARCH EXAMPLE

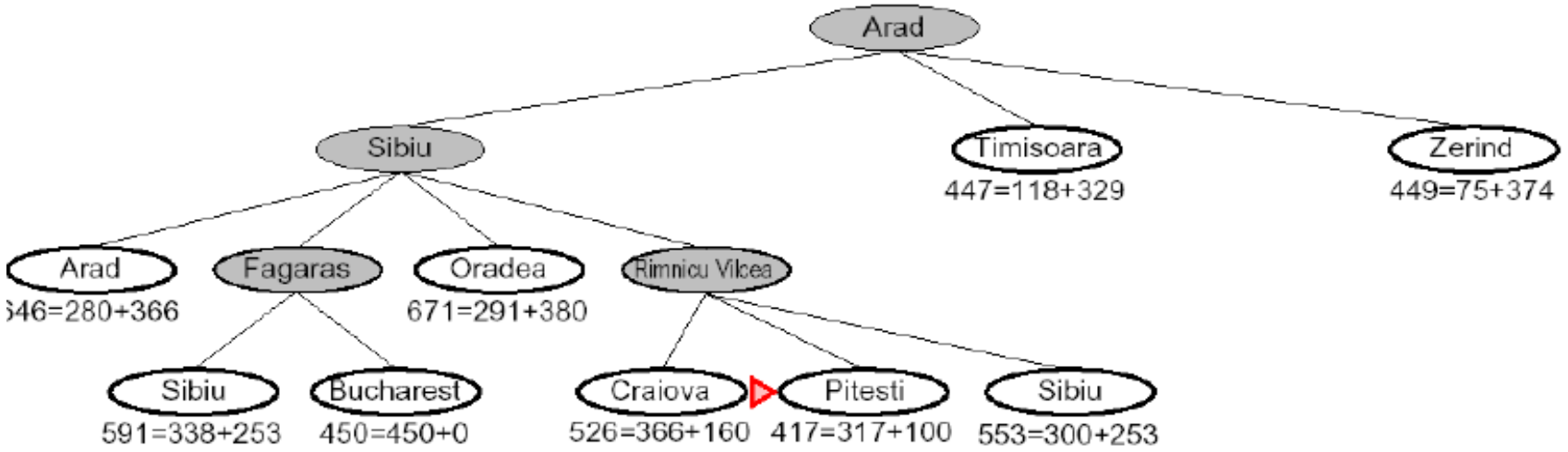
Frontier queue:
Fagaras 415
Pitesti 417
Timisoara 447
Zerind 449
Craiova 526
Sibiu 553
Arad 646
Oradea 671



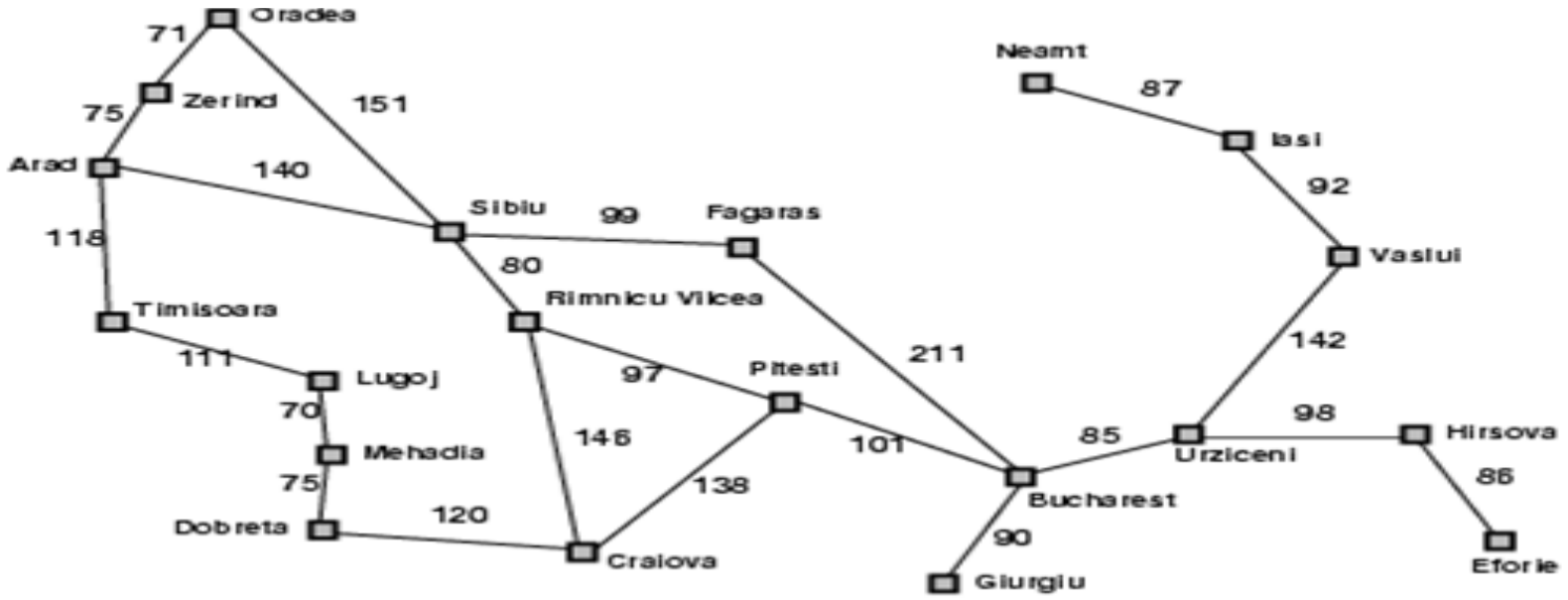


To Bucharest	distance
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* SEARCH EXAMPLE

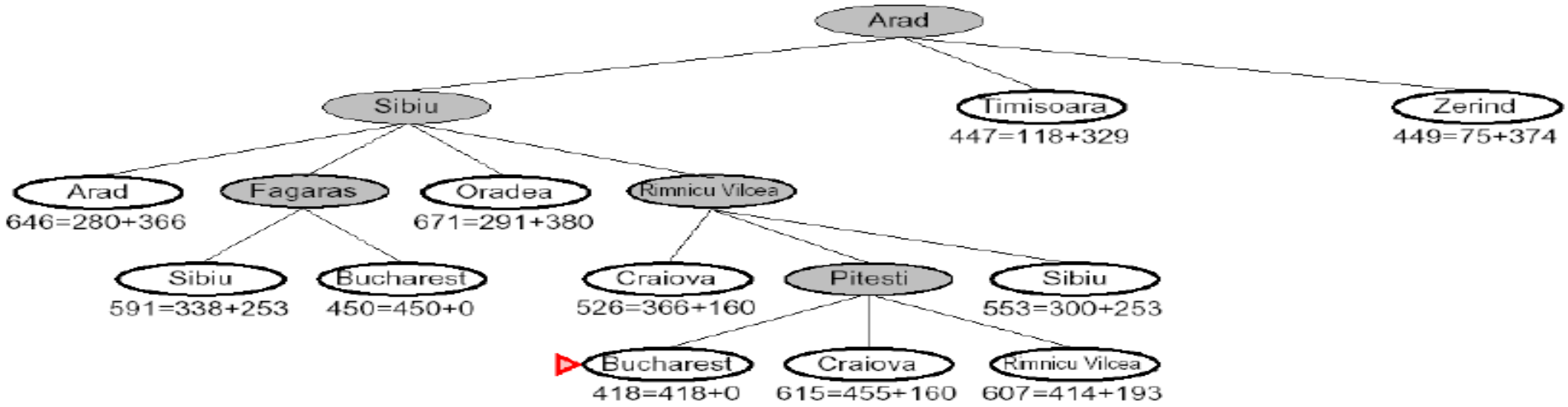


Frontier queue:
Pitesti 417
Timisoara 447
Zerind 449
Bucharest 450
Craiova 526
Sibiu 553
Sibiu 591
Arad 646
Oradea 671



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* SEARCH EXAMPLE



Frontier queue:

Bucharest 418

Timisoara 447

Zerind 449

Bucharest 450

Craiova 526

Sibiu 553

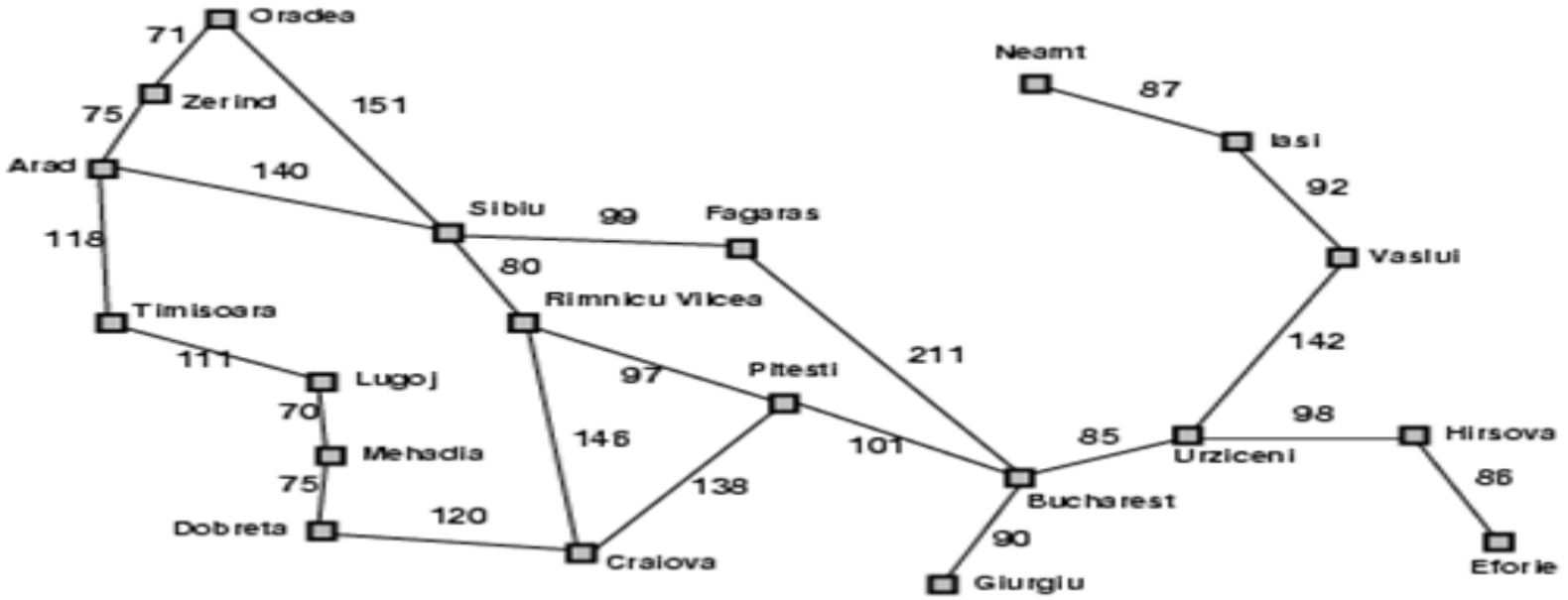
Sibiu 591

Rimricu Vicea 607

Craiova 615

Arad 646

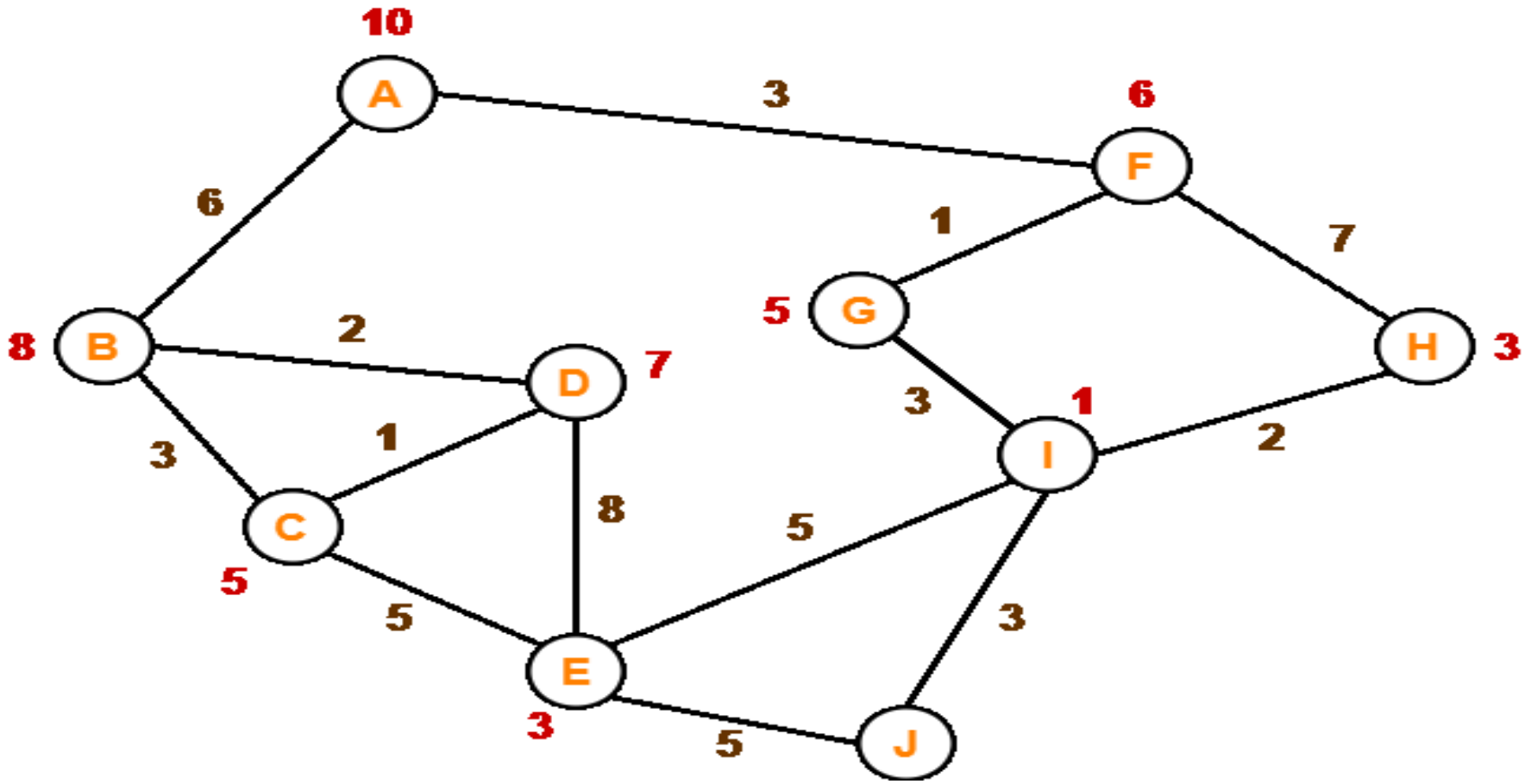
Oradea 671



Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* CHALLENGE



PROPERTIES OF A*

- **Complete?**
- Yes (unless there are infinitely many nodes with f ≤f(G))
- **Time?**
- Exponential
- **Space?**
- Keeps all nodes in memory
- **Optimal?**
- Yes (depending upon search algo and heuristic property)

PROPERTIES OF HEURISTIC FUNCTION

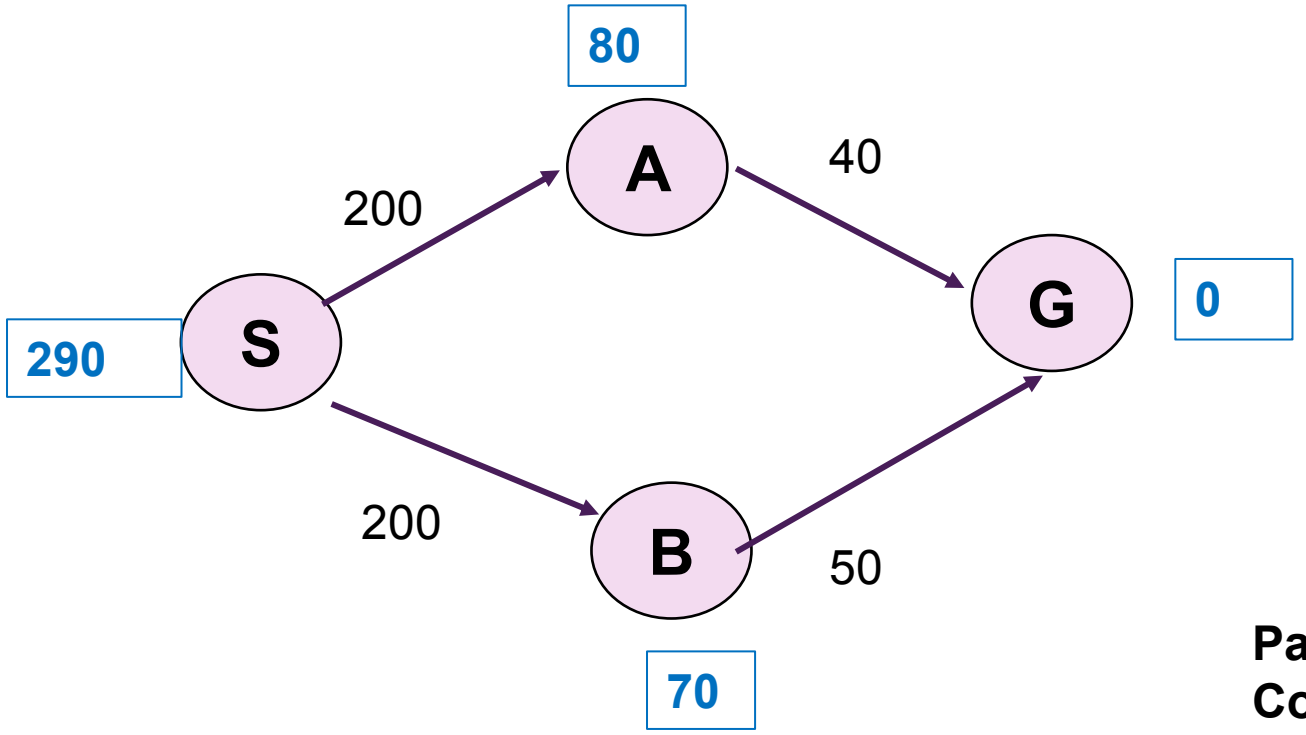
1.OPTIMALITY

"Is it guaranteed to find the shortest path?"

- The **tree search** version of **A***is **optimal** if **h(n) is admissible**, while **graph search version** is **optimal** if **h(n) is consistent**.
- If **h never overestimates the real distance** then h is admissible, A* is guaranteed to find the **optimal solution.** An admissible heuristic is optimistic
- Example admissible heuristic: straight-line distance.
- A* with an admissible heuristic is **optimal** for a tree search, meaning when there is only one path to any given state.
- A search algorithm is **admissible** if it is guaranteed to find a minimal path to a solution whenever such a solution exists. **Breadth first search** is admissible, because it looks at every state at level **n** before considering any state at level **n+1**.
- A* is **complete** if the search graph is finite or when there is no infinite path in the graph having a finite total cost.

ADMISSIBILITY EXAMPLE

Overestimation : will it guarantee optimality?

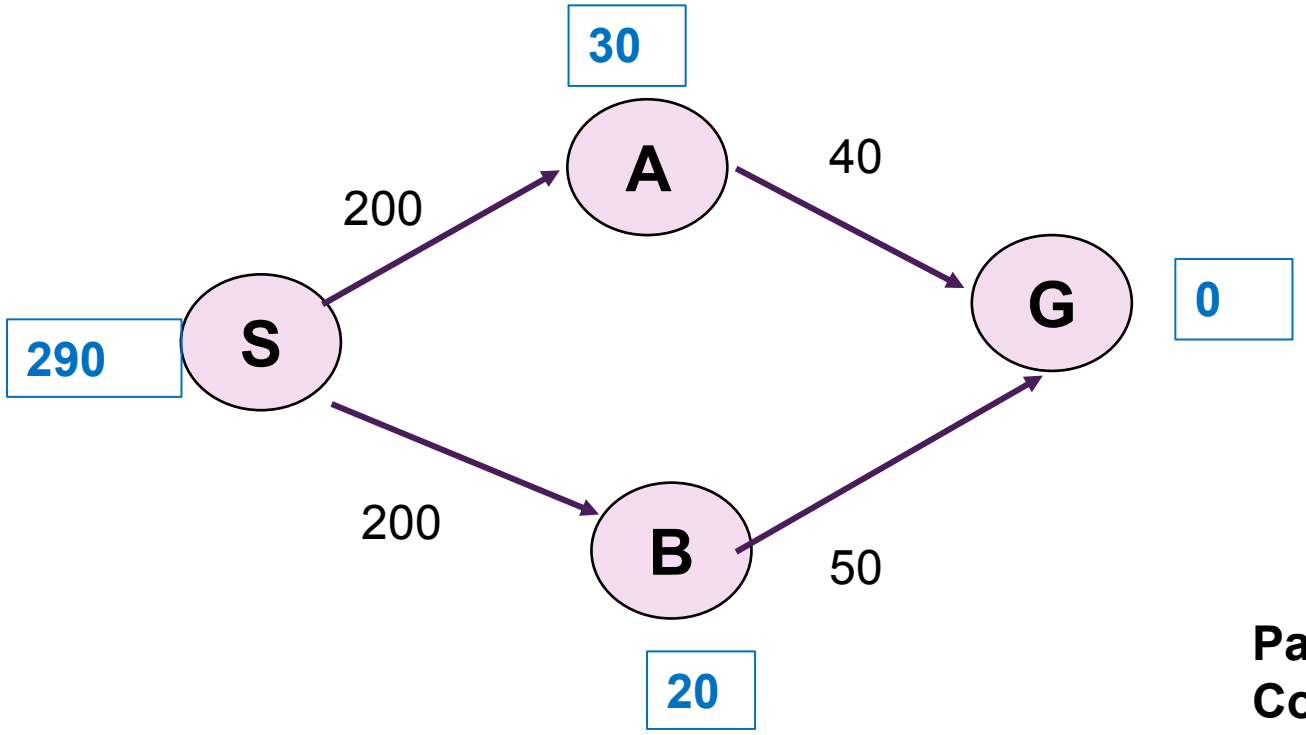


Expanded Node	Fringe Queue
--	S^{290}
S^{290}	A^{280} B^{270}
B^{270}	G^{250} A^{280}
G^{250}	A^{280}

Path: S-B-G
Cost:250
Optimal: NO

ADMISSIBILITY EXAMPLE

Underestimation : will it guarantee optimality?

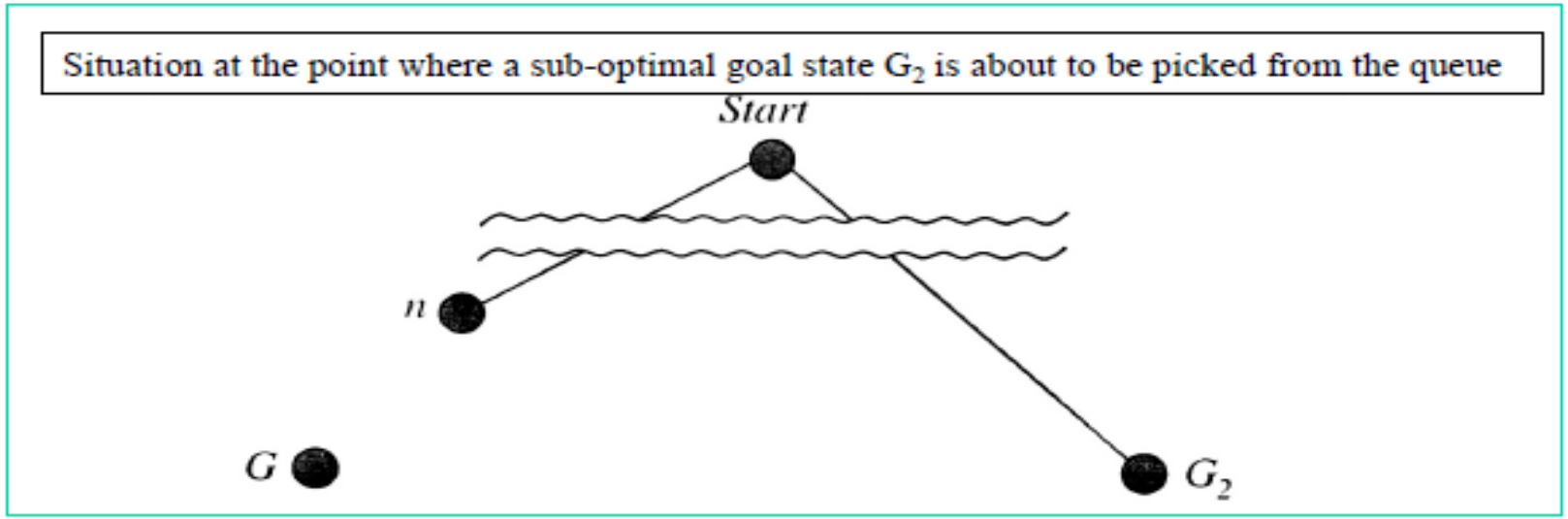


Expanded Node	Fringe Queue
--	S^{290}
S^{290}	A^{230} B^{220}
B^{220}	A^{230} G^{250}
A^{230}	G^{240} G^{250}
G^{240}	

Path: S-A-G
Cost:240
Optimal: YES

PROOF OF OPTIMALITY IN A*

- Let G be an optimal goal state, and f(G) = f* = g(G). Let G2 be a suboptimal goal state, i.e. f(G2) = g(G2) > f*. Suppose for contradiction that A* has selected G2 from the queue. (This would terminate A* with a suboptimal solution) Let n be a node that is currently a leaf node on an optimal path to G.



- Because h is admissible, f* ≥ f(n). If n is not chosen for expansion over G2, we must have f(n) ≥ f(G2) So, f* ≥ f(G2). Because h(G2)=0, we have f* ≥ g(G2), contradiction

2. INFORMEDNESS

"Is it better than another heuristic?"

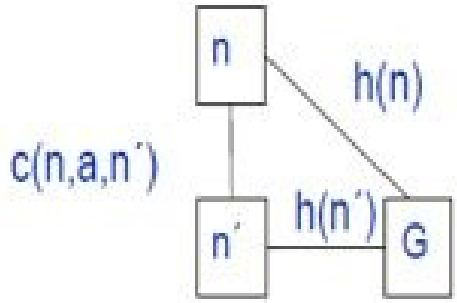
- **Informedness** A search strategy which searches less of the state space in order to find a goal state is more informed.
- Ideally, a search strategy which is both admissible (so it will find us an optimal path to the goal state), and informed (so it will find the optimal path quickly.) is preferred.
- Formally, for two admissible heuristics h1 and h2, if h1(n) <= h2(n) for all states n in the state-space, then heuristic h2 is said to be more informed than h1.
- **Dominance:** If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1
- h_2 is better for search: it is guaranteed to expand less or equal nr of nodes.

3.MONOTONICITY/CONSITENCY

"Does it make steady progress toward a goal?"

- Monotnicity ensures consistency in finding minimal path to each state encountered in the search.
- **If along any path in the search tree of A* the f-cost never decreases, then the heuristic is said to be monotonic.**
- h(n) is monotonic if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n': h(n) ≤ c(n,a,n') +h(n'). This is known as the triangular inequality.

for example if g(n) = 3 andh(n) = 4. Thenf(n) = g(n)+h(n) -that is, we know that the true cost of a solution path through n is at least 7. Suppose also that g(n') = 4 and h(n') =2, so that f(n') = 6. Clearly, this is an example of a nonmonotonic heuristic as cost should be at least 7.



3. MONOTONICITY

- ***f* is *monotonously non-decreasing if* a heuristic function *h* satisfies the *monotonicity restriction:*** Let n' be a descendant of n and if ***h(n) -h(n') ≤ cost(n…n')*** and the heuristic evaluation of the goal state is 0: h(Goal) = 0.
- This property asks if an algorithm is **locally admissible**---that is, it always **underestimates the cost between any two states in the search space**

Consequences of Non-Monotonous Heuristic function

- Montonous Heuristic deletes later-created duplicates
- **A search method that reaches a given node at different depths in the search tree is not monotone.**
- A* with a consistent heuristic is optimal for any kind of search (graph search).
- When a state is discovered by using heuristic search, is there any guarantee that the same state won't be found later in the search at a cheaper cost (with a shorter path from the start state)? This is the property of monotonicity.

ITERATIVE IMPROVEMENT ALGORITHMS

ITERATIVE IMPROVEMENT ALGORITHMS

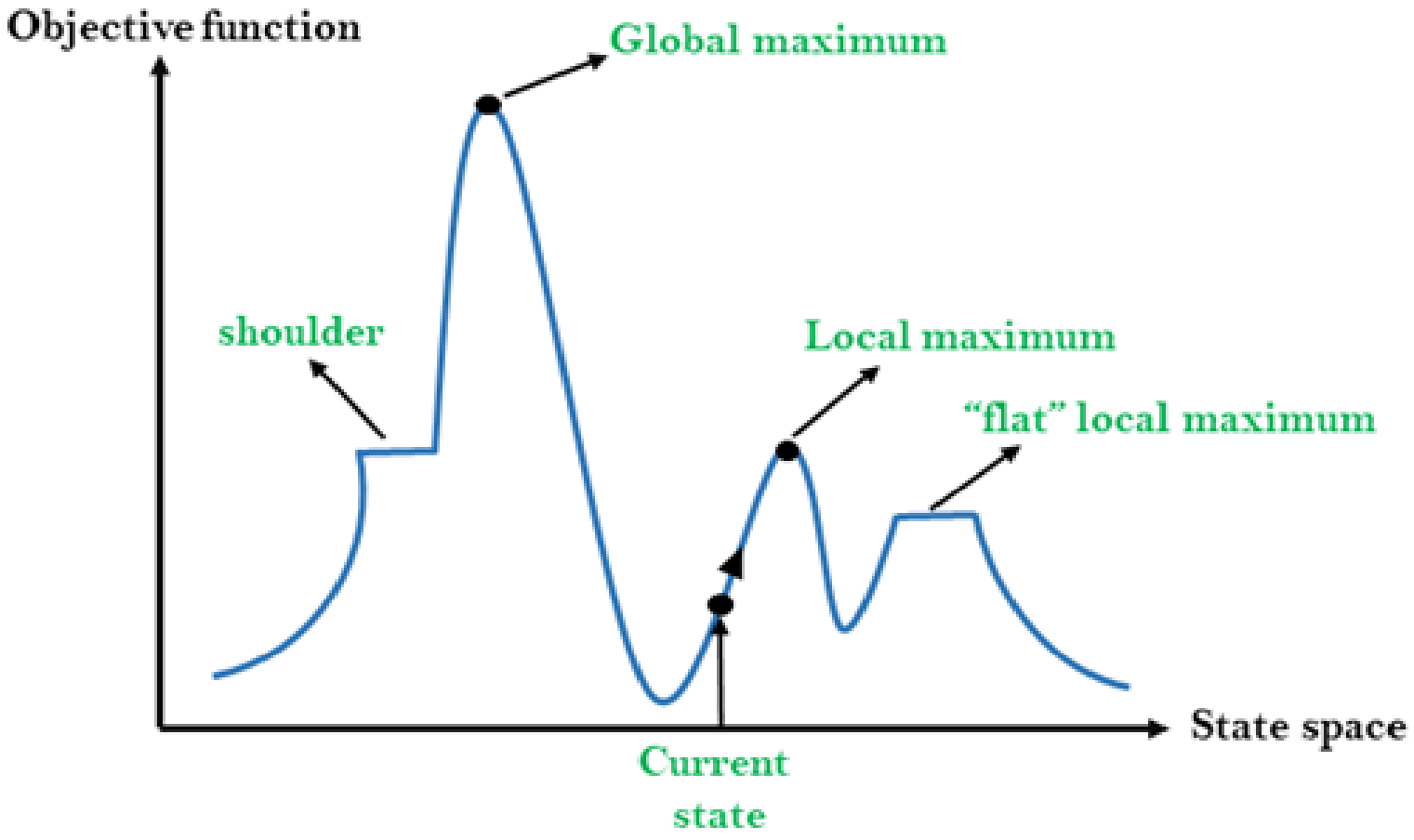
- **Iterative improvement algorithms provide the most practical approach for problems in which all the information needed for a solution are contained in the state description itself but performs purely a local search in state space, evaluating and modifying one or more current states rather than systematically exploring paths from initial state.**
- **The general idea is to start with a complete configuration and to make modifications to improve its quality.**
- The idea of iterative improvement is to **move around the landscape trying to find the highest peaks, which are the optimal solutions**.
- The **height** of any point on the landscape corresponds to the **evaluation function of the state** at that point.
- Iterative improvement algorithms usually **keep track of only the current state**, and do not look ahead **beyond the immediate neighbors of that state** so can **save some memory space**
- **The iterative improvement algorithms divide into two major classes. The first one is called the *Hill-climbing search*, also called *gradient descent*. The second one is called the *Simulated annealing*.**
- **Suitable for problem for which solution state matters rather thanpath cost to reach it.**

LOCAL SEARCH ALGORITHMS & OPTIMIZATION

- Local search algorithm searches in the state space, **evaluates and modifies one or more current states rather than systematically exploring paths from an initial state**.
- Suitable for problems in which all that **matters is the solution state**, not the path cost to reach it.
- **Eg:** 8-Queens problem, integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.
- Local search algorithms operate on a single current node (rather than multiple paths) and generally move only to neighbors of that node. **Path**s followed by the search **are not retained.**
- **Advantages:**

1. Use very **little memory**—usually a constant amount;
2. Often find **reasonable solutions** in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
3. Useful for **solving pure optimization problems**, which the aim to find the best state according to an objective function.

DIFFERENT REGIONS IN THE STATE SPACE LANDSCAPE



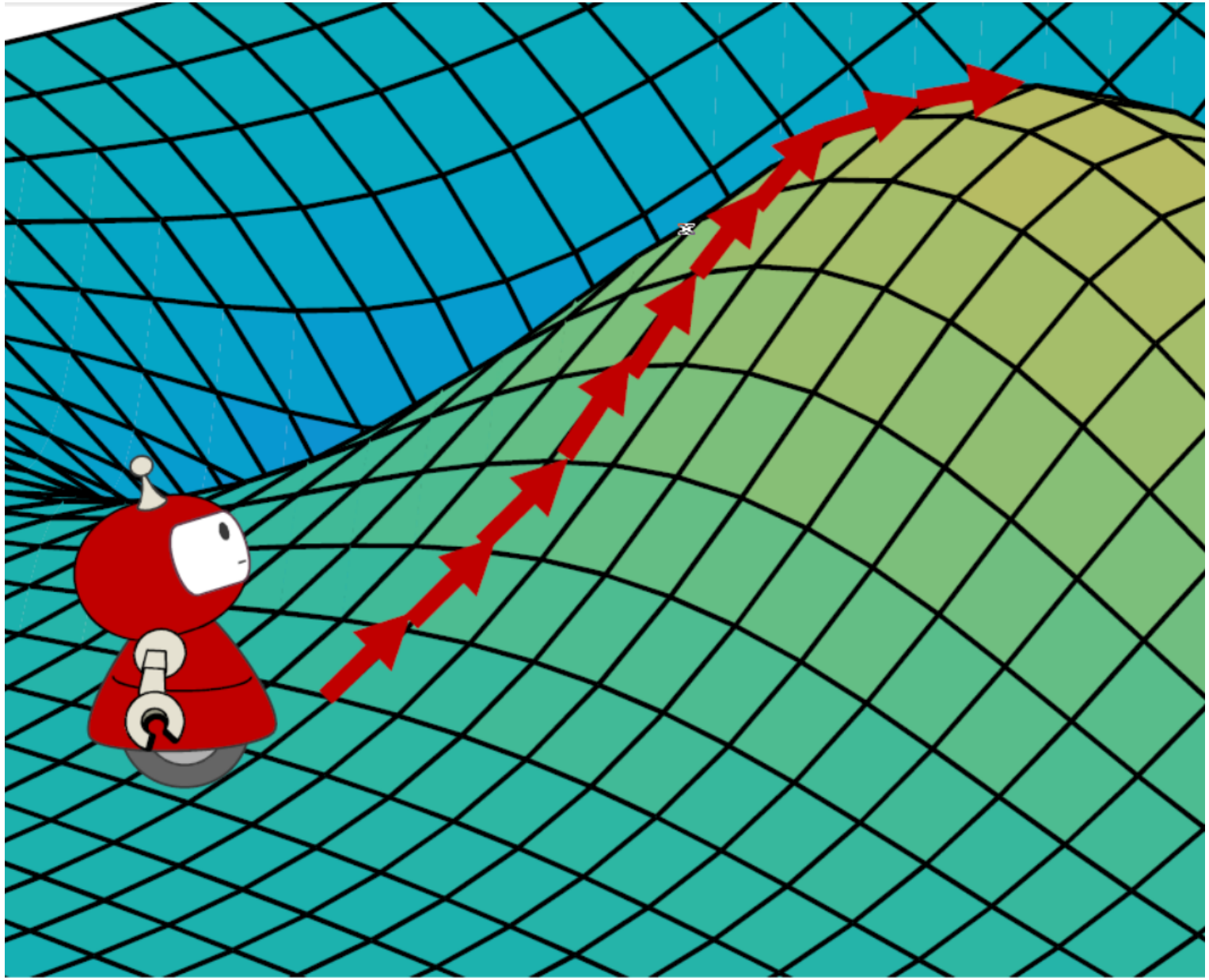
- **Local Maximum:** a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.

TRIVIAL ALGORITHMS

- **Random Sampling**
  - Generate a state randomly
- **Random Walk**
  - Randomly pick a neighbor of the current state
- Both algorithms asymptotically complete.
- Local search can do quite well on optimization problems.

HILL CLIMBING SEARCH

"Like climbing everest in thick fog with amnesia"



HILL CLIMBING SEARCH

- **Analog**
  - Go uphill along the steepest possible path until no farther up
  - A **greedy local search algorithm**
  - 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**
- **Principle**
  - Expand **the current state** of the search and **evaluate** its children
  - Select the **best child, ignore its siblings** and **parent**
  - **No history for backtracking**
  - The basic idea is to **proceed** according to some **heuristic measurement of the remaining distance to the goa**l.
  - attempts to find a better solution by incrementally changing a

HILL CLIMBING SEARCH

- The *Hill-climbing search algorithm* is a loop that continually moves in the direction of increasing value. The algorithm **only records the state** and its **evaluation instead of maintaining a search tree**. It takes a *problem* as an input, and it keeps comparing the values of the *current* and the *next* nodes.
  The ***next* node** is the **highest-valued successor** of the **current node**. If the value of the *current* node is greater than the next node, then the c*urrent* node will be returned. Otherwise, it will go deeper to look at the *next* node of the *next* node.
- **Hill climbing** is a mathematical optimization technique which belongs to the family of **local search**. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.
- It is also called **greedy local search** as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are **state** and **value.**
- Hill Climbing is mostly used **when a good heuristic is available.**

HILL CLIMBING ALGORITHM

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

FEATURES OF HILL CLIMBING

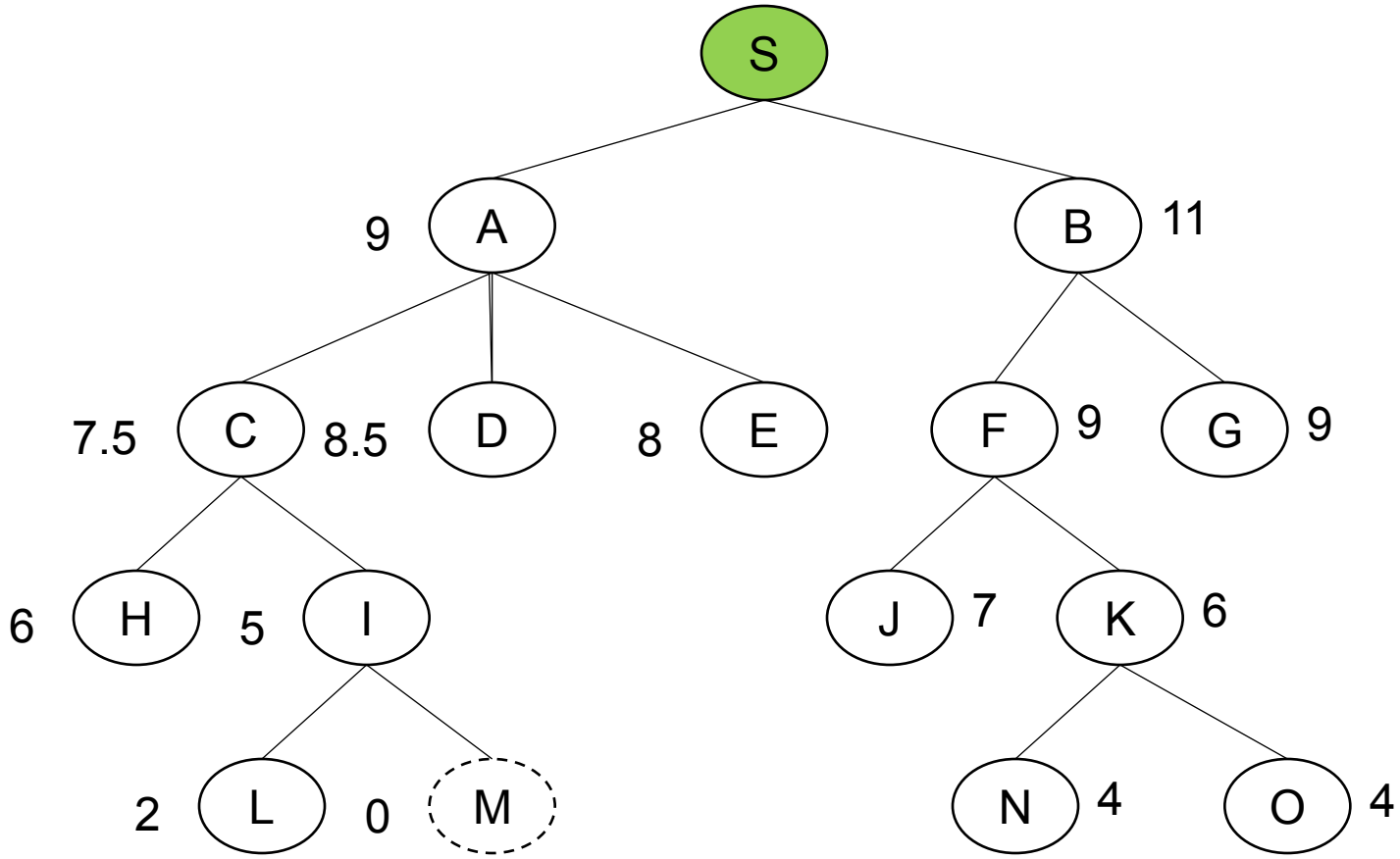
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

STATE SPACE DIAGRAM FOR HILL CLIMBING

- The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

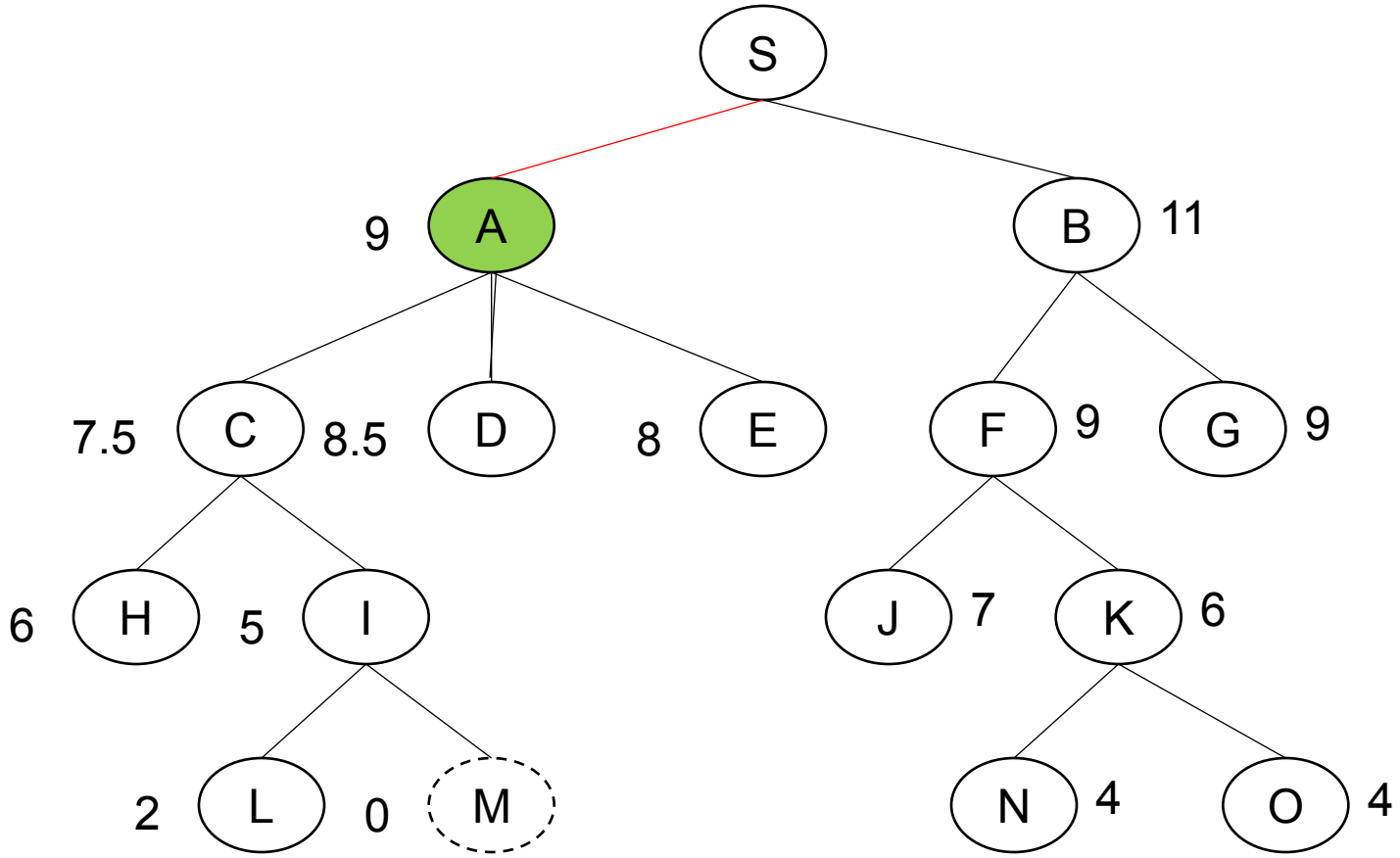
Hill-climbing search example

our aim is to find a path from S to M

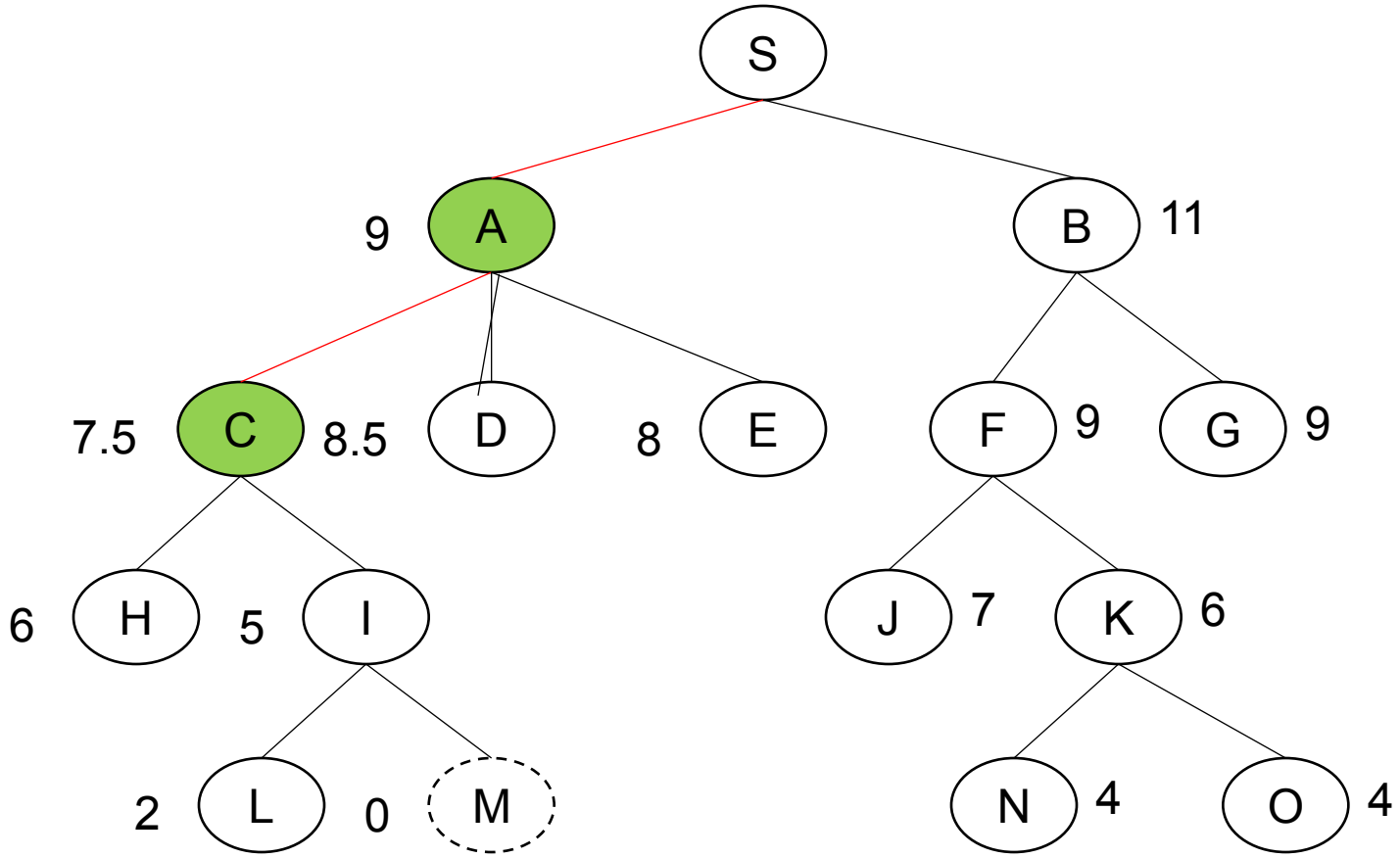


associate heuristics with every node, that is the straight line distance from the path terminating city to the goal city

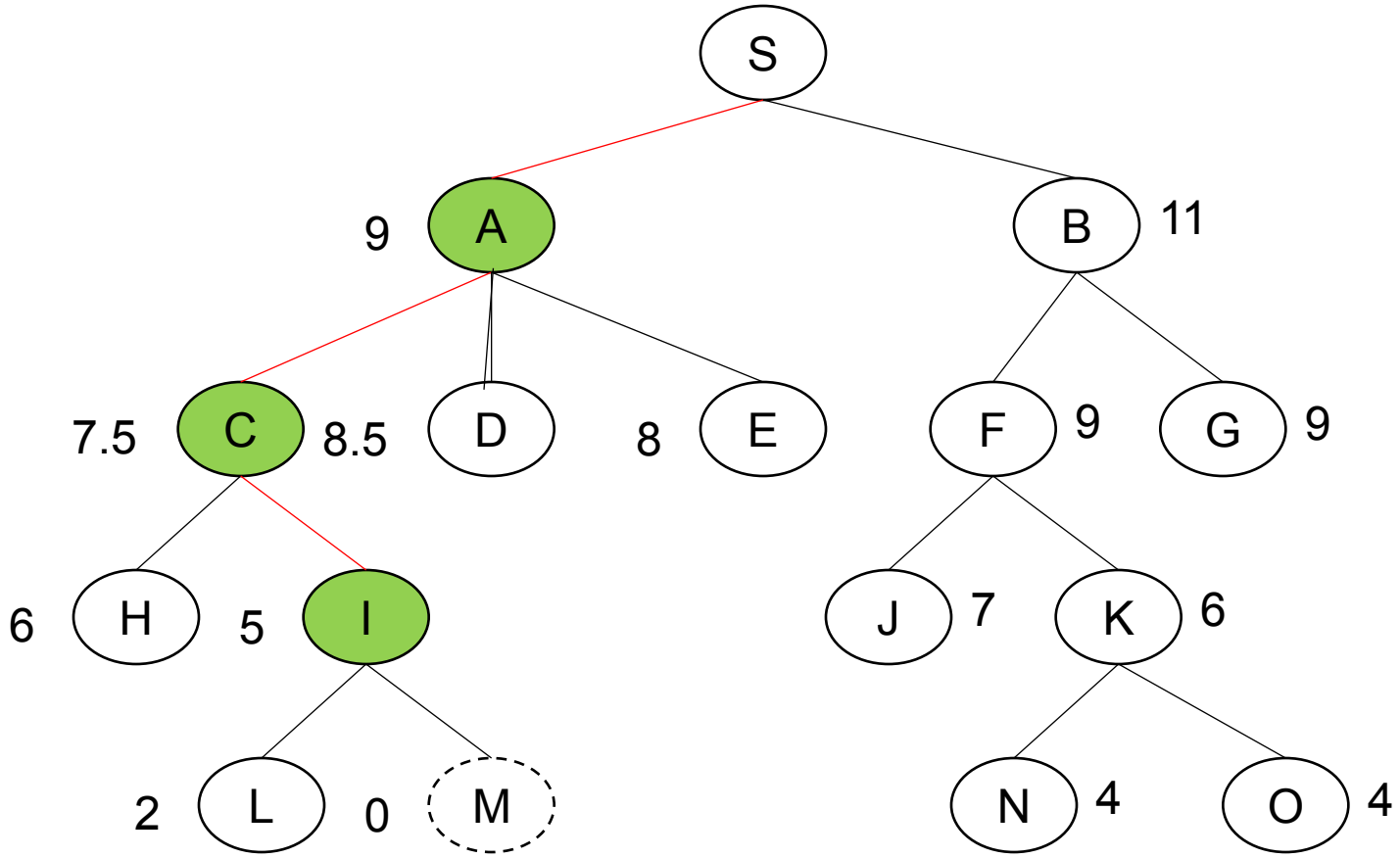
Hill-climbing search example



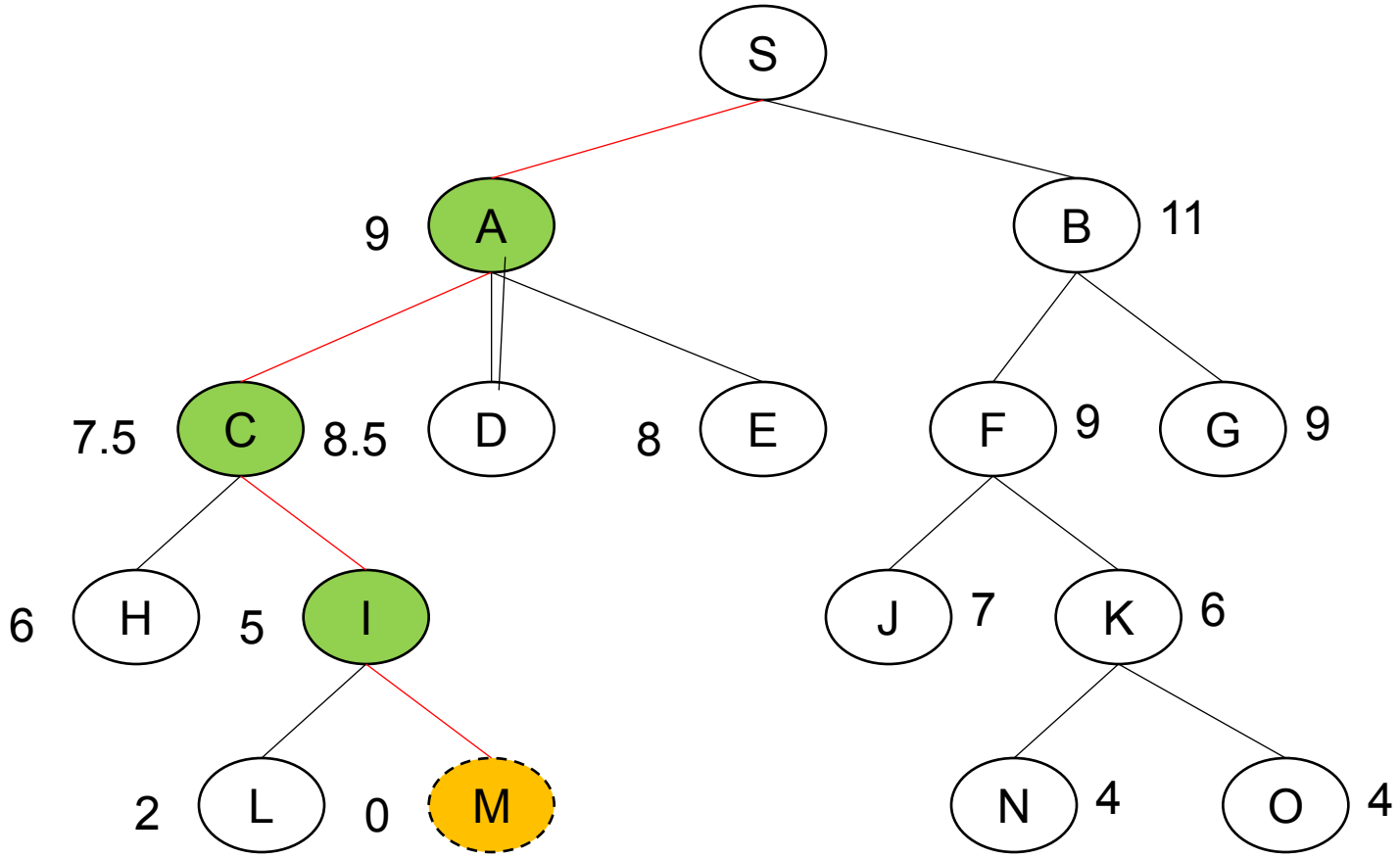
Hill-climbing search example



Hill-climbing search example

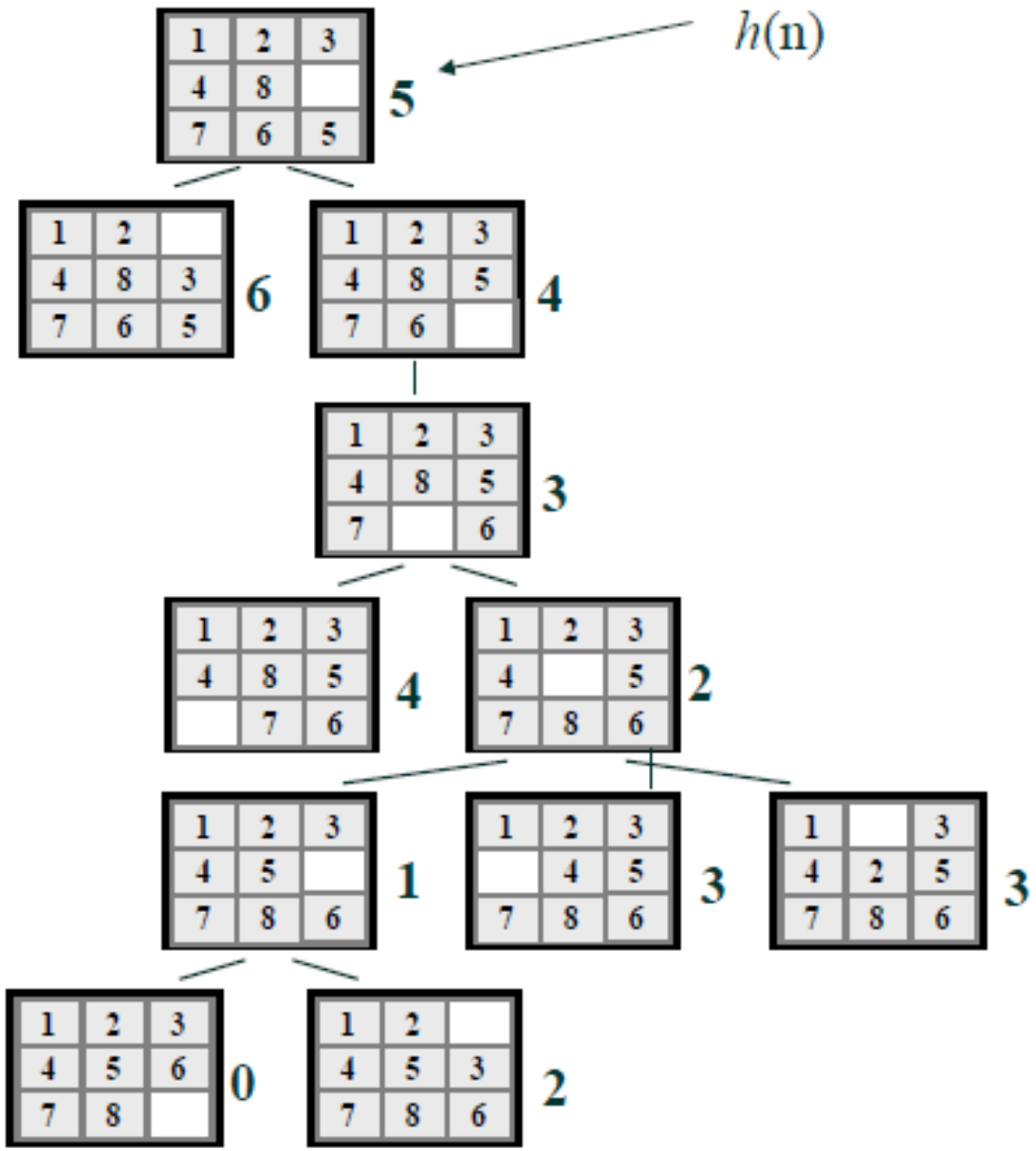


Hill-climbing search example



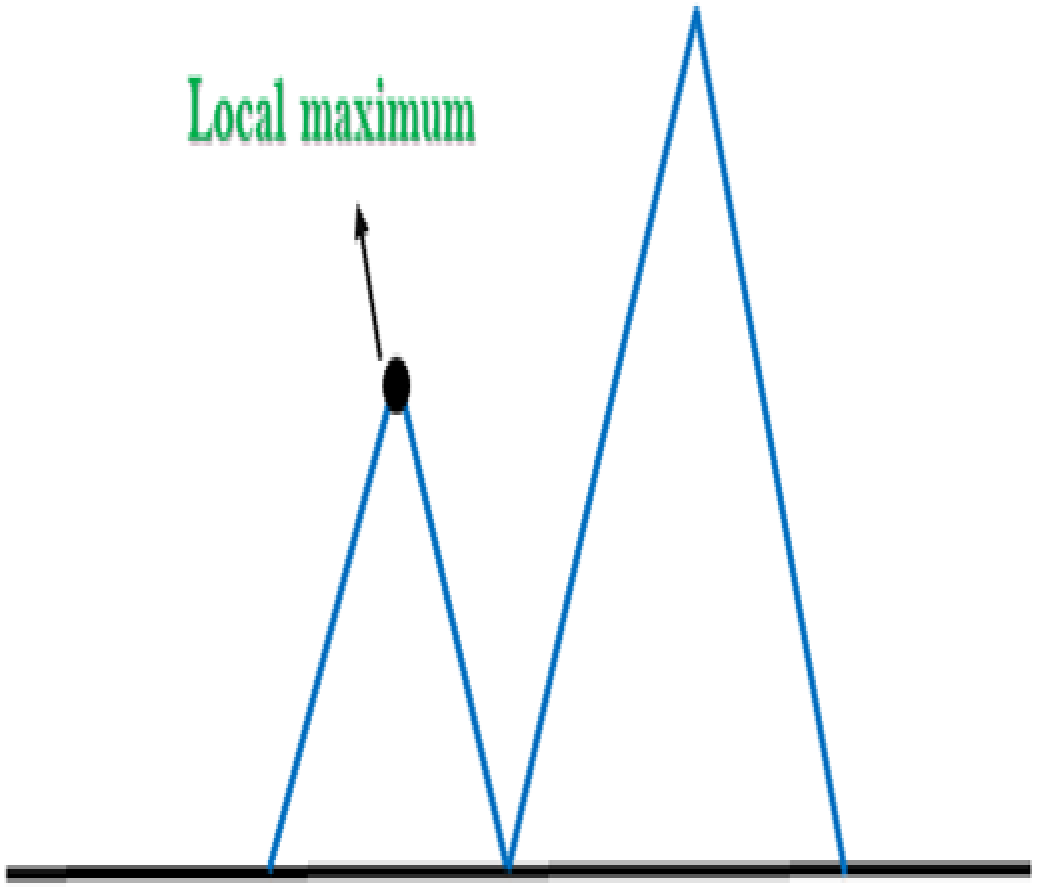
HILL CLIMBING EXAMPLE

MANHATTAN DISTANCE IS USED AS HEURISTIC



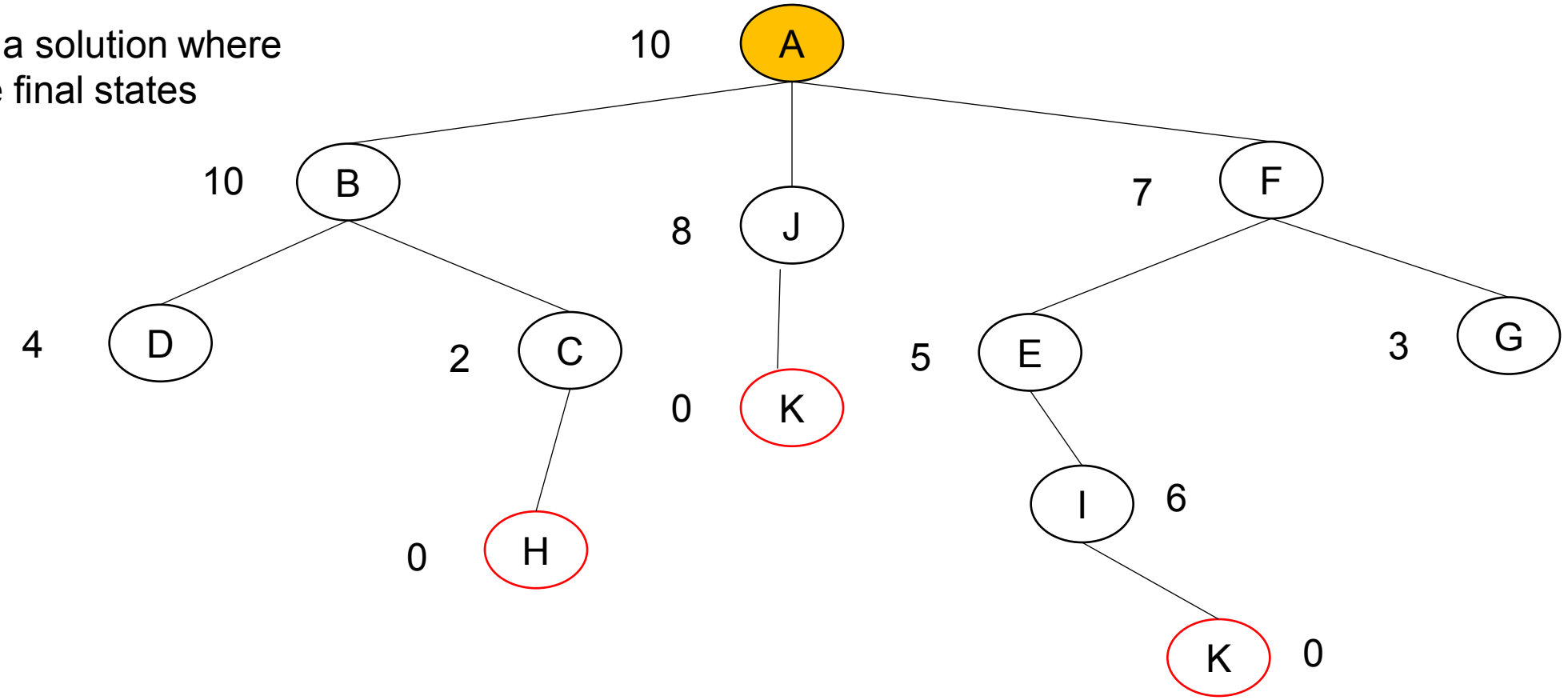
PROBLEMS IN HILL CLIMBING
1. LOCAL MAXIMUM

- A local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. .
- **PROBLEM:** When a local maxima is reached, the algorithm will stuck with no where else to go even a solution has not been reached yet.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

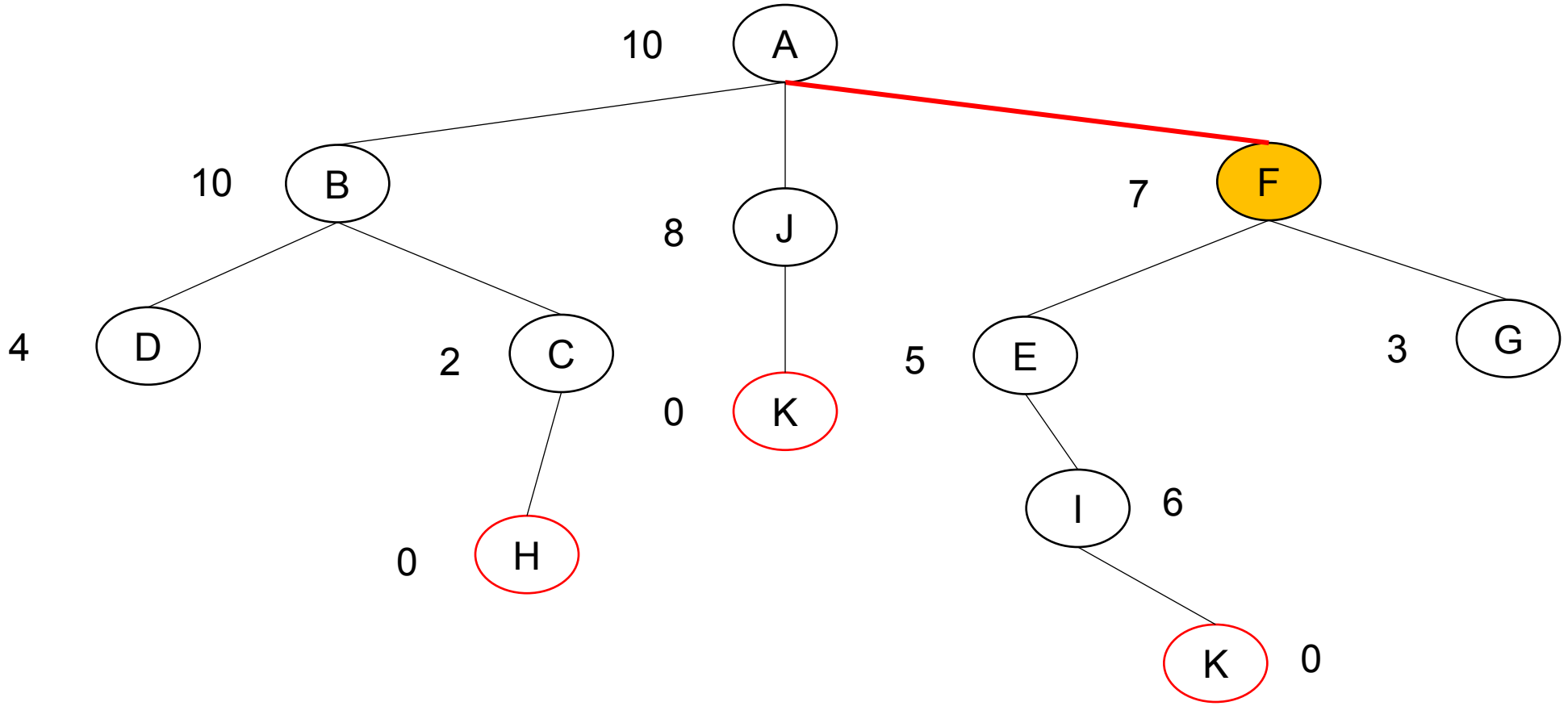


Hill-climbing search example
Local maximum

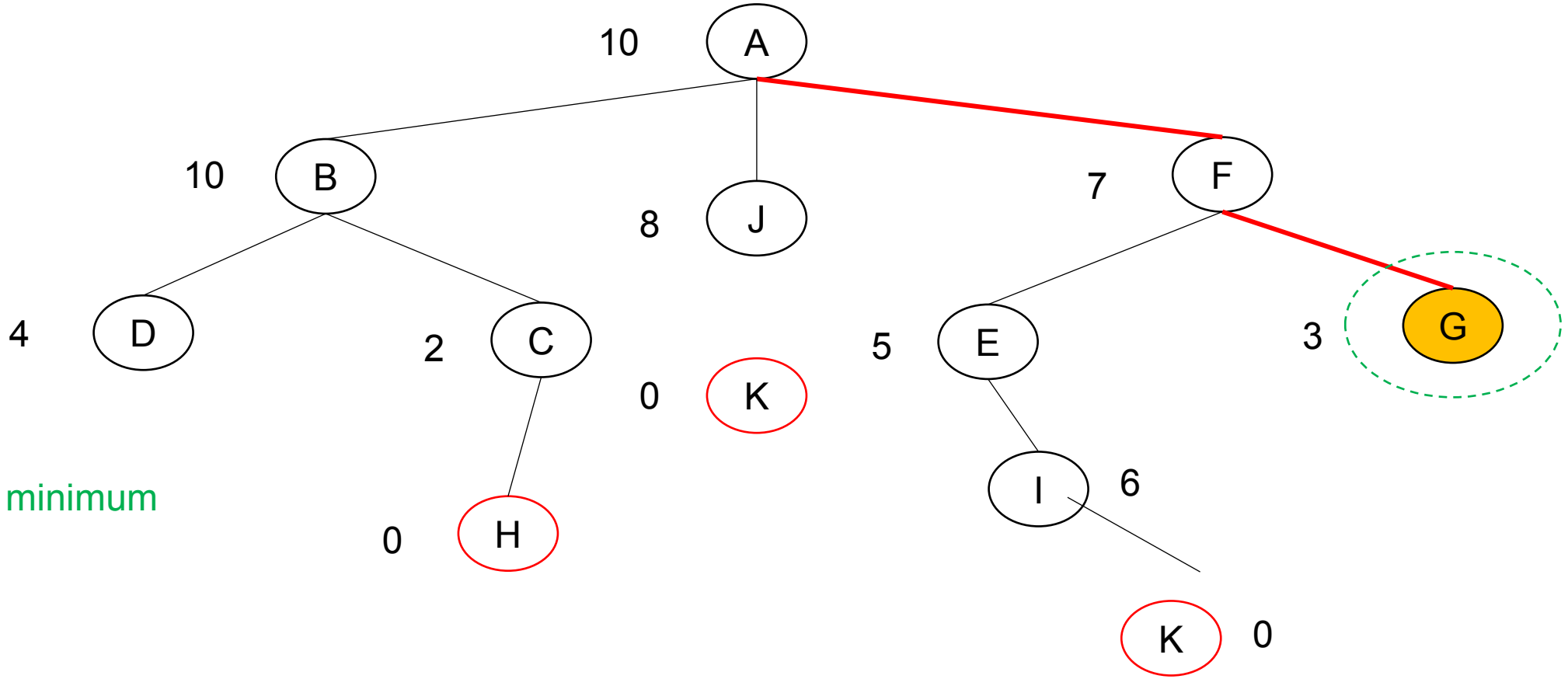
From A find a solution where
H and K are final states



Hill-climbing search example
Local maximum



Hill-climbing search example
Local minimum

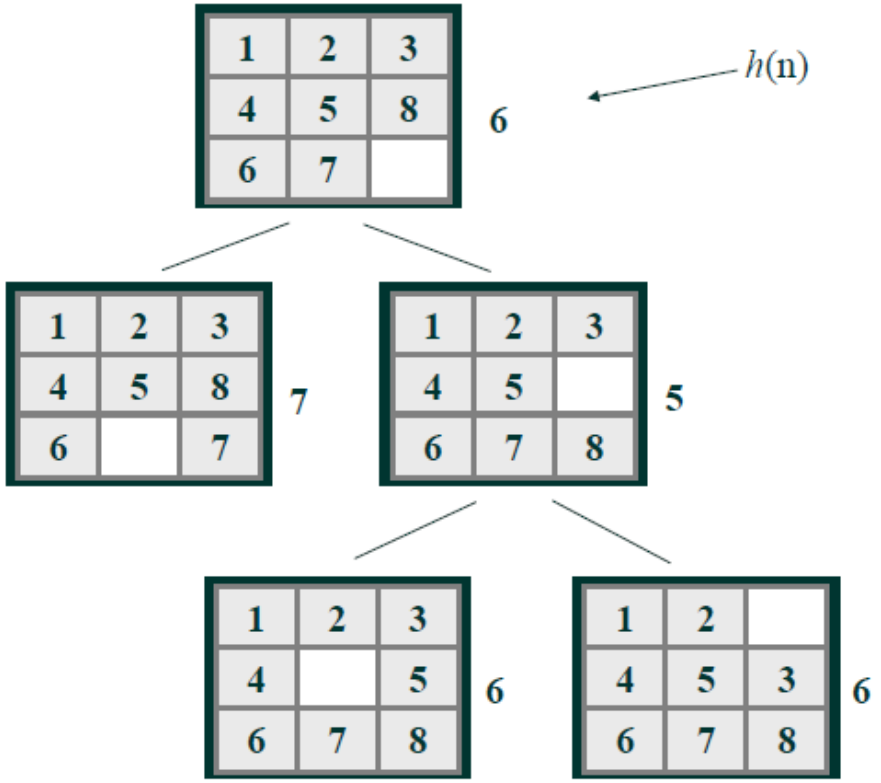


G is local minimum

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

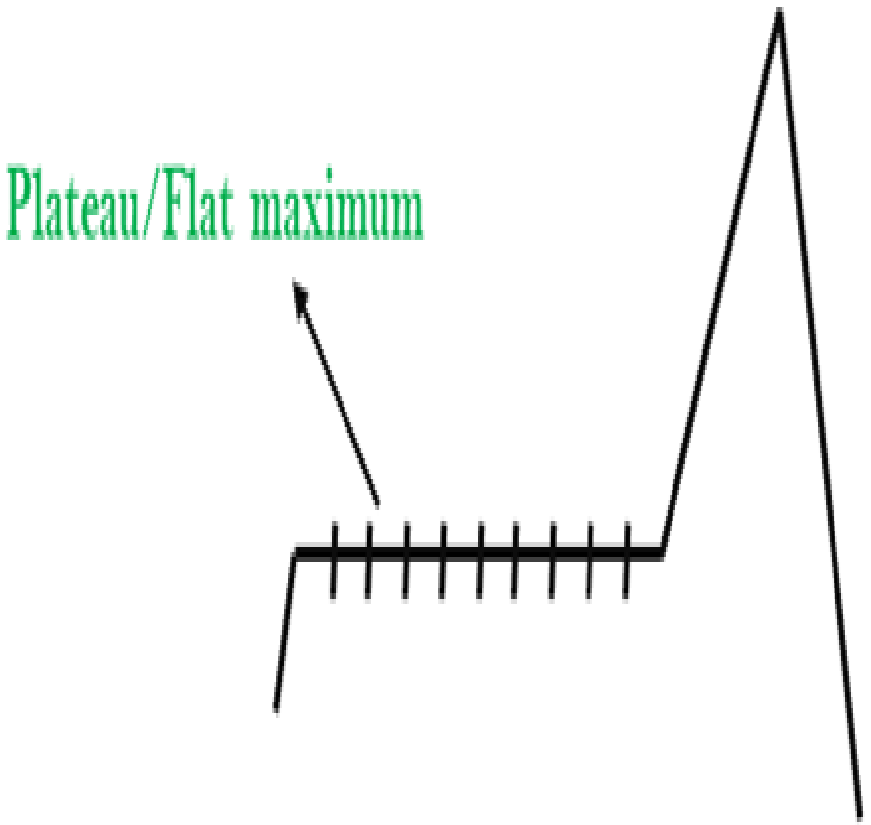
PROBLEMS OF HILL CLIMBING

All the nodes on the fringe are taking a step "backwards" (local minima)



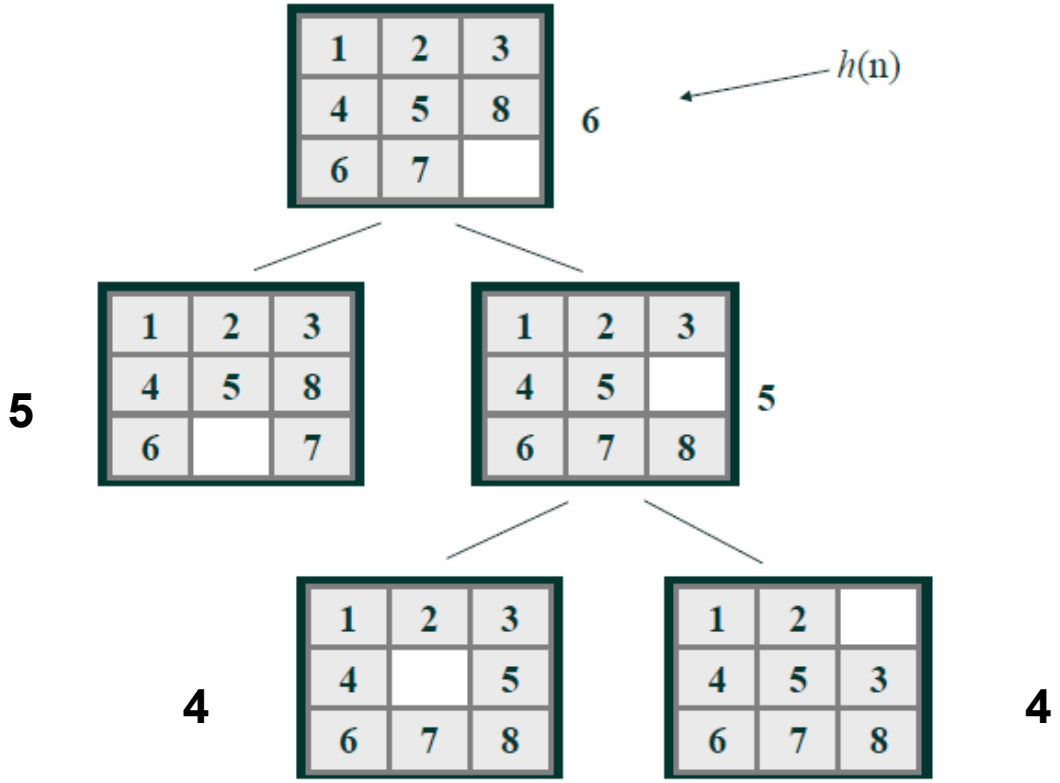
2. PLATEAU

- A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area. : a plateau is an area of the state space landscape where the evaluation function is flat.
- On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.
- A plateau is an area of the state space where the neighbors are about the same height. In such a situation, a random walk will be generated.
- **Solution:** The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-



PROBLEMS OF HILL CLIMBING

All the nodes on the fringe are having same heuristic value 4 (plateau)

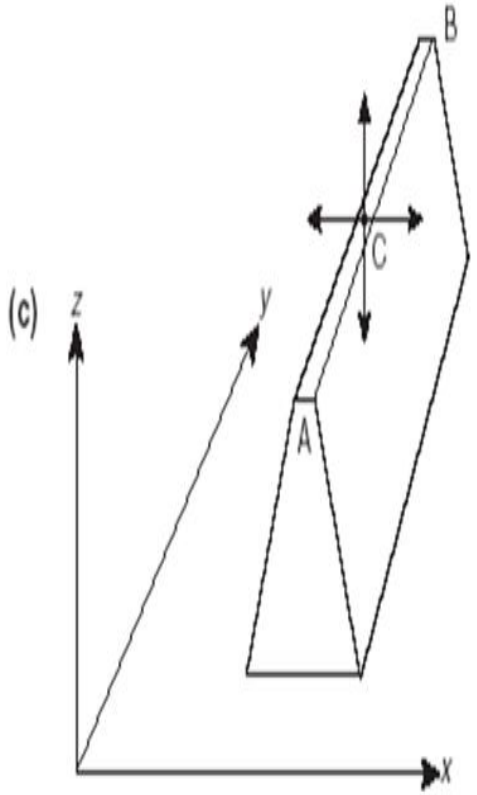


3. RIDGES

- Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate. The search direction is not towards the top but towards the side
- A ridge may have steeply sloping sides towards the top, but the top only slopes gently towards a peak. In this case, the search makes little progress unless the top is directly reached, because it has to go back and forth from side to side.
- A ridge is a special kind of local maximum. It is an area of the search space that is higher that the surrounding areas and that itself has a slope. But theorientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves. Any point on a ridge can look like peak because movement in all probe directions is downward.
- **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.

■B is higher than A.

■ At C, the hill-climber can't find a higher point North, South, East or West, so it stops.



HOW HILL CLIMBING RECOVERS?

- If algorithm encounters a situation that there is **more than one best successor to choose from**, it will randomly select a certain one.
- If algorithm reaches a point at which no progress is being made further from a certain starting point, algorithm start again from a different starting point. It is known as **Random-restart hill-climbing**
- Conducting a series of hill-climbing searches from randomly generated initial states, running each until it halts or makes no discernible progress. It saves the best result found so far from any of the searches. It can use a fixed number of iterations, or can continue until the best saved result has not been improved for a certain number of iterations.
- As a matter of fact, and obviously, the fewer local maxima, the quicker it finds a good solution. And it can eventually find out the optimal solution if enough iterations are allowed. But usually, a reasonably good solution can be found after a small number of iterations.

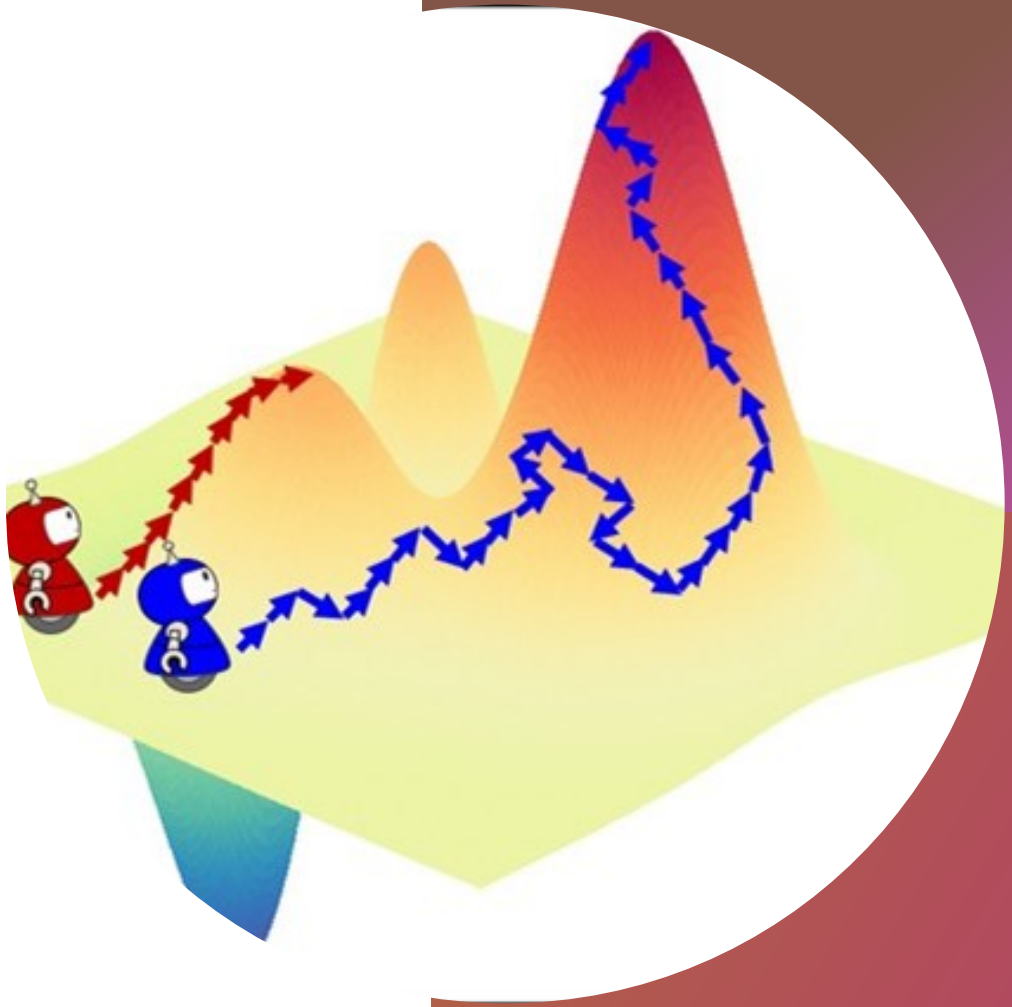
VARIANTS OF HILL CLIMBING ALGORITHMS

1. **Stochastic hill Climbing:** It does not examine all the neighboring nodes before deciding which node to select .It just **selects a neighboring node at random** and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.
2. **Simple hill Climbing:** It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node. **I** It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.
3. **Steepest-Ascent hill-climbing:** It first examines all the neighboring nodes and then selects the node closest to the solution state as of next node. This algorithm consumes more time as it searches for multiple neighbors
4. **First Choice Hill Climbing:** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.
5. **Random-restart Hill Climbing:** It conducts a series of hill-climbing searches from randomly generated initial states,1 until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is 1/p.

APPLICATIONS

- Hill Climbing technique can be used to solve many problems, where the current state allows for an accurate evaluation function, such as Network-Flow, Travelling Salesman problem, 8-Queens problem, Integrated Circuit design, etc.
- Hill Climbing is used in inductive learning methods too. This technique is used in robotics for coordination among multiple robots in a team. There are many other problems where this technique is used.

SIMULATED ANNEALING



SIMULATED ANNEALING

- **Hill Climbing Major drawbacks**
- A **hill-climbing algorithm** never makes “downhill” moves toward states with lower value (or higher cost) turns out to be **incomplete**, since it get stuck on a local maximum. In contrast, **a purely random walk**, moving to a successor chosen uniformly at random from the set of successors is **complete** but **extremely inefficient**.
- **Solution:**
- **Simulated Annealing:** Combine hill climbing with a random walk in some way that yields both efficiency and completeness
- **Analog**
- Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

SIMULATED ANNEALING

- Motivated by the physical annealing process used to **temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.**
- The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.
- When the end the searching is close, it starts behaving like *hill-climbin*g. At the start, make lots of moves and then gradually slow down
- More formally…
  - Instead of picking the best move (as in Hill Climbing), **Generate a random new neighbor** from **current state**, **If it's better take it**,
  - **If it's worse** then **take it with some probability proportional to the temperature ie the delta between the new and old states. Probability gets smaller** as time passes and by the amount of "badness" of the move,
- Compared to hill climbing the main difference is that **SA allows downwards steps**; (moves to higher cost successors).
- SA **can avoid becoming trapped at local maxima**. SA uses a random search that occasionally accepts changes that decrease objective function f. SA uses a control parameter T, which by analogy with the original application is known as the system "temperature." T starts out high and gradually decreases toward 0.

SIMULATED ANNEALING PROBABILITY FACTOR

- By checking the value of Delt-E the algorithm can determine the probability of the next move. If *Delta-E* is greater than zero, then the *next* node will be looked at. Otherwise, the probability for the *next* node to be looked at is *e* to the power *Delta-E* over *T*.
- subtracting the values of the current node from the next node to obtained the difference **Delta-E<-- f (B) – f (A)**
- **$P(move_{A \rightarrow B}) = e^{(f(B) - f(A)) / T}$**
- *Delta-E* is actually the amount by which the evaluation is worsened
- A local variable T which is the temperature controlling the probability of downward steps. The higher T, the more likely a bad move will be made. As T tends to zero, this probability tends to zero, and SA becomes more like hill climbing. If T is lowered slowly enough, SA is complete and admissible.
- **Simulated Annealing** is an algorithm which yields both efficiency and completeness.

SIMULATED ANNEALING STEPS

1.Select a start node(rootnode).

2.Randomly select a child of the current node, calculate a value reflecting how good such child is like v$\boldsymbol{alu}(\boldsymbol{node})=-\boldsymbol{h}eu\boldsymbol{ristic}(\boldsymbol{node})$.

3.Select the child if it is better than the current node. Else try another child.

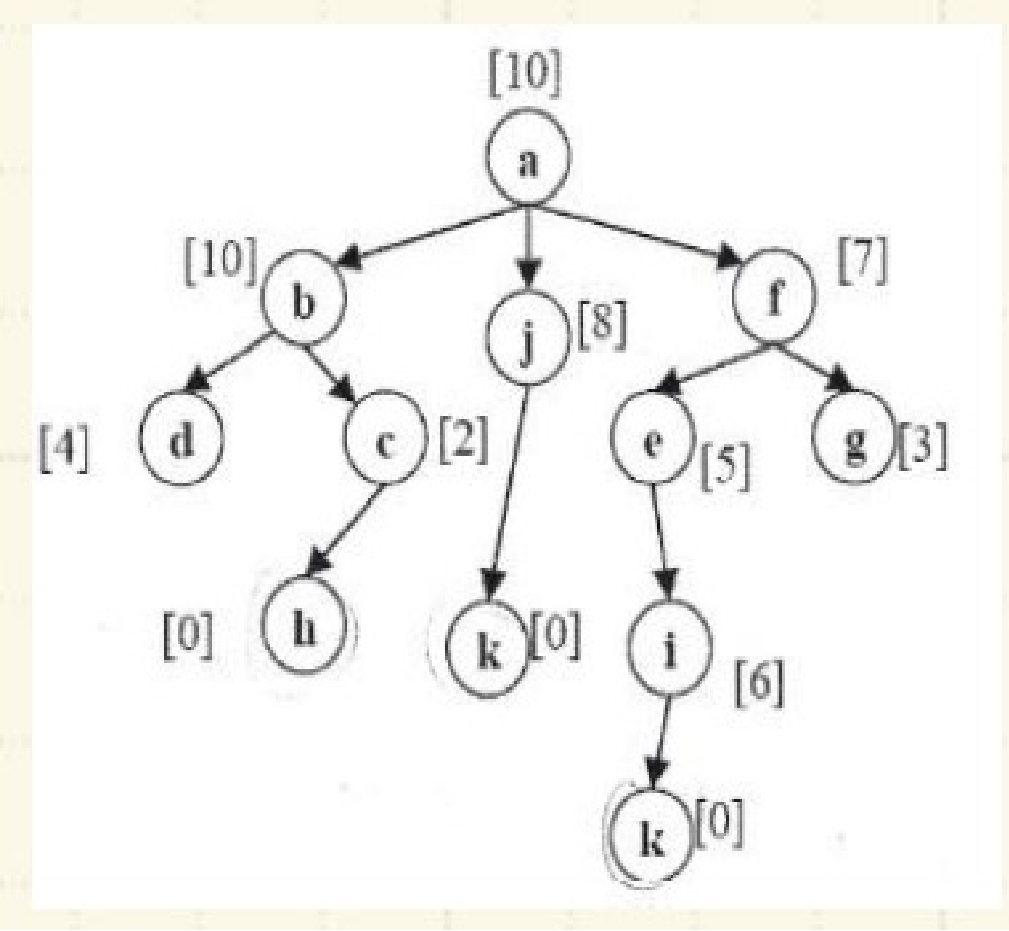
A node is better than the current node if $\boldsymbol{\Delta E=valuenext-valuecurrent>0}$.

Else if $\boldsymbol{\Delta E<0}$, then try to find another child.

4.If the child was not better than the current node then it will be selected with probability equal to $\mathbf{p}=\boldsymbol{e}^{\boldsymbol{\Delta E/T}}$where $\boldsymbol{\Delta E=valuenext-value}[\boldsymbol{current}$ and $\boldsymbol{T}$ is a temperature.

5.Stop if no improvement can be found or after a fixed time.

EXAMPLE



Current	Children
a	---
a	f_7, j_8, b_{10}

Randomly Select a Child

Check if next node f_7 is better than current node

$\Delta E > 0$

$$\Delta E = \text{value}(\text{next}) - \text{value}(\text{current})$$

$$\Delta E = \text{value}(f_7) - \text{value}(a_{10})$$

$$\text{value}(f_7) = -\text{heuristic}(f_7) = -7$$

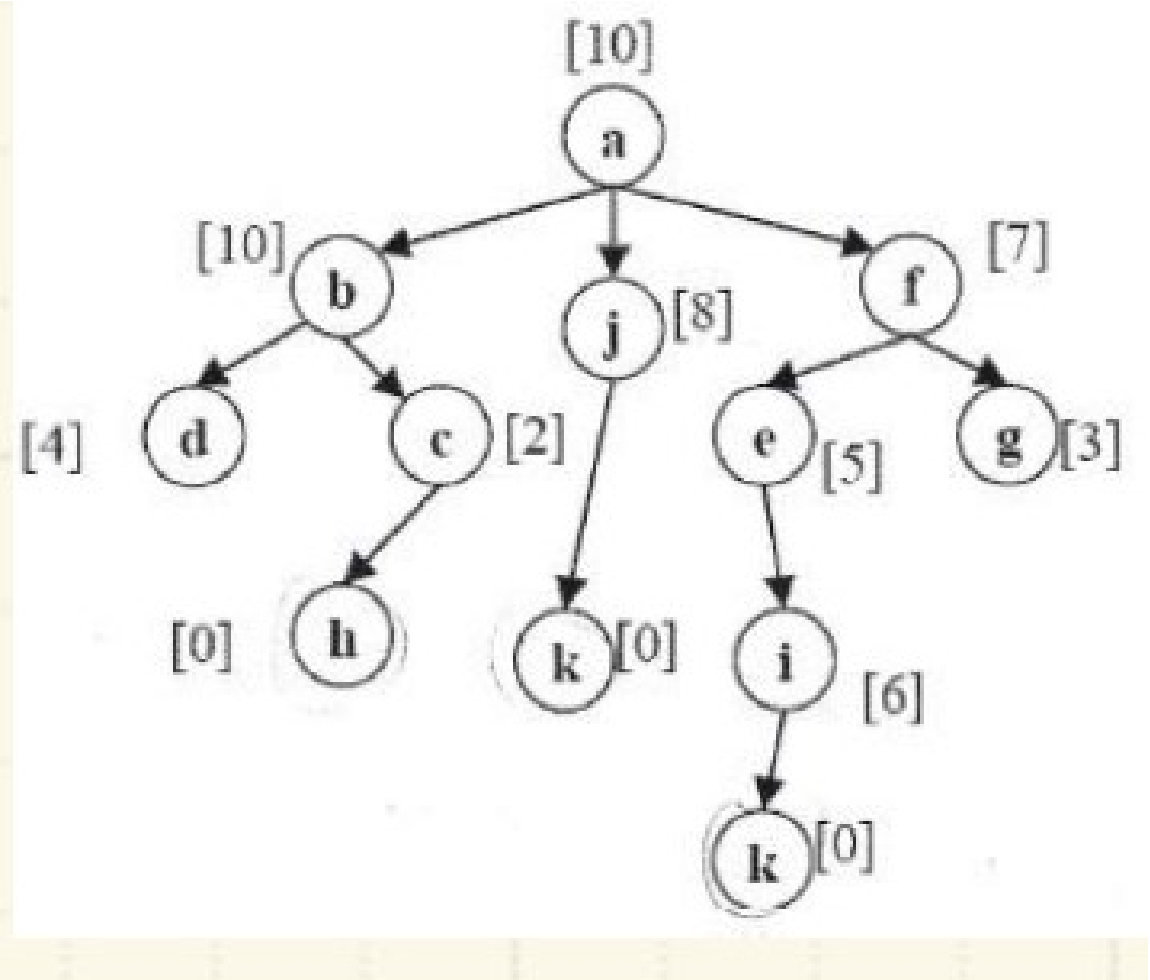
$$\text{value}(a_{10}) = -\text{heuristic}(a_{10}) = -10$$

$$\Delta E = -7 - (-10) = +3$$

$\because \Delta E > 0$

$\therefore f_7$ will be selected with probability 1

EXAMPLE



Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3

Randomly Select a Child

Check if next node e_5 is better than current node

$$\Delta E = \text{value(next)} - \text{value(current)}$$

$$\Delta E = \text{value}(e_5) - \text{value}(f_7)$$

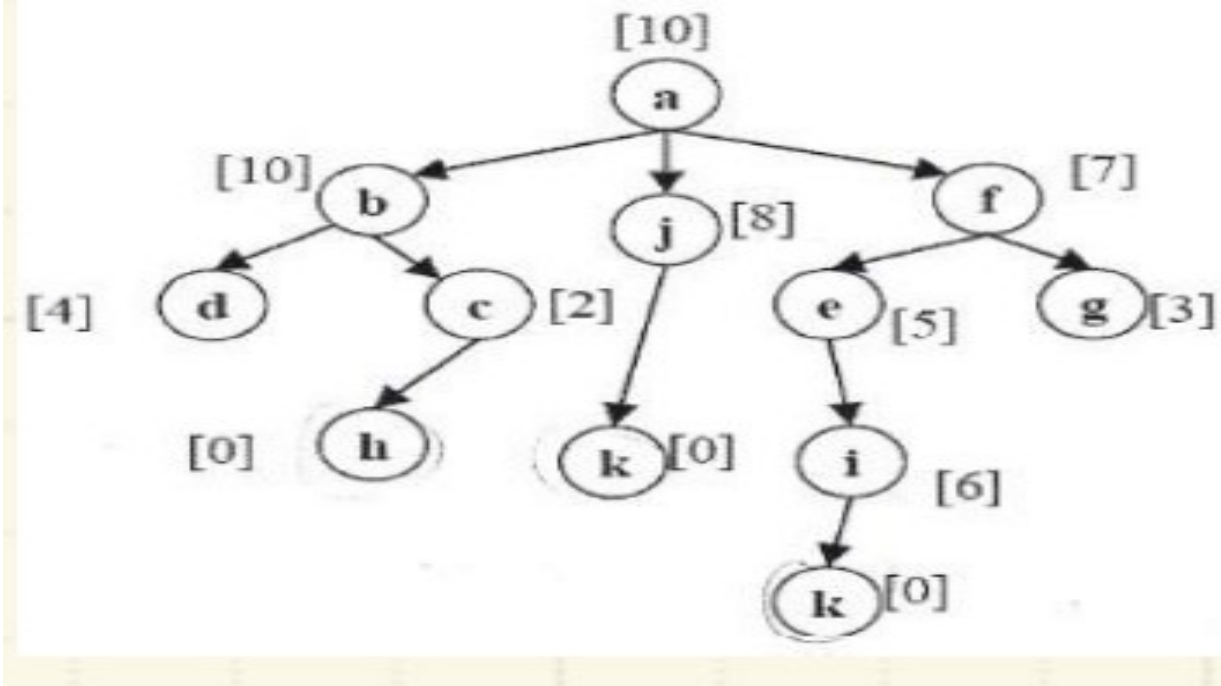
$$\text{value}(e_5) = -\text{heuristic}(e_5) = -5$$

$$\text{value}(f_7) = -\text{heuristic}(f_7) = -7$$

$$\Delta E = -5 - (-7) = +2$$

$\because \Delta E > 0$

$\therefore e_5$ will be selected with probability 1



Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6

Randomly Select a Child

Check if next node i_6 is better than current node

$\Delta E > 0$

$\Delta E = \text{value(next)} - \text{value(current)}$

$\Delta E = \text{value}(i_6) - \text{value}(e_5)$

$\text{value}(i_6) = -\text{heuristic}(i_6) = -6$

$\text{value}(e_5) = -\text{heuristic}(e_5) = -5$

$\Delta E = -6 - (-5) = -1$

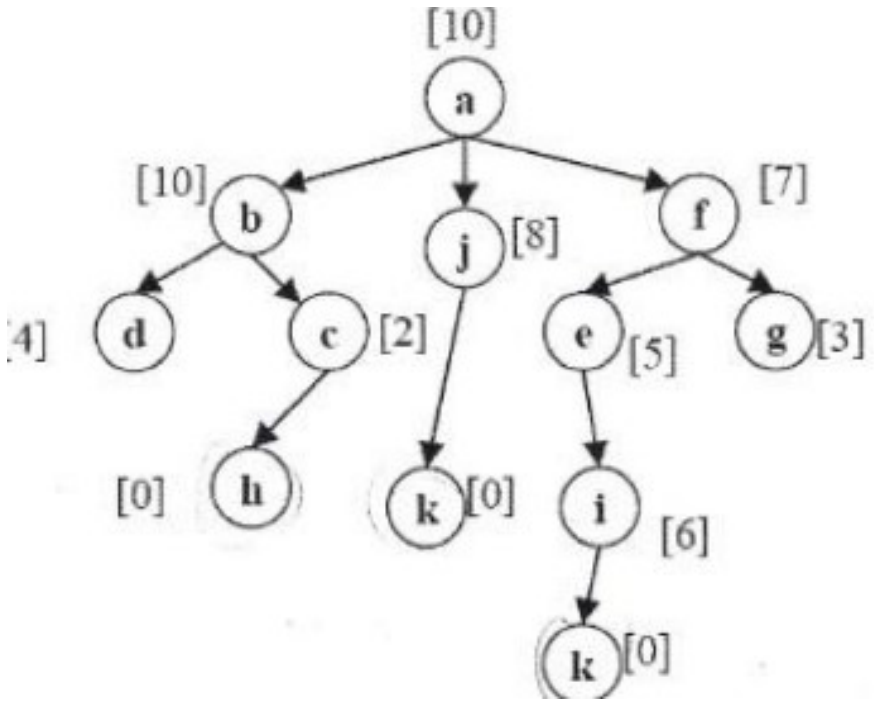
$\because \Delta E < 0$

$\therefore i_6$ can be selected with probability $p = e^{\frac{\Delta E}{T}}$

$$p = e^{\frac{-1}{10}} = e^{\frac{-1}{10}} = .905$$

Because the only child of e_5 is i_6 then it will be selected even if its probability is not 1.

EXAMPLE



Randomly Select a Child

Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6
i	k_0

Check if next node k_0 is better than current node

$\Delta E > 0$

$\Delta E = \text{value}(k_0) - \text{value}(i_6)$

$\text{value}(k_0) = -\text{heuristic}(k_0) = 0$

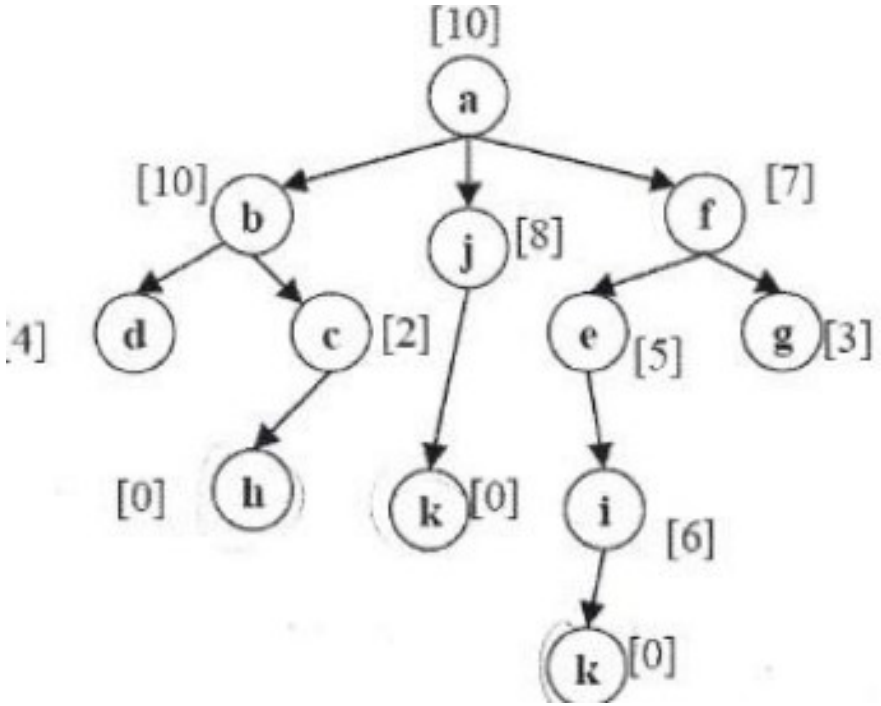
$\text{value}(i_6) = -\text{heuristic}(i_6) = -6$

$\Delta E = 0 - (-6) = +6$

$\because \Delta E > 0$

$\therefore k_0$ will be selected with probability 1

EXAMPLE

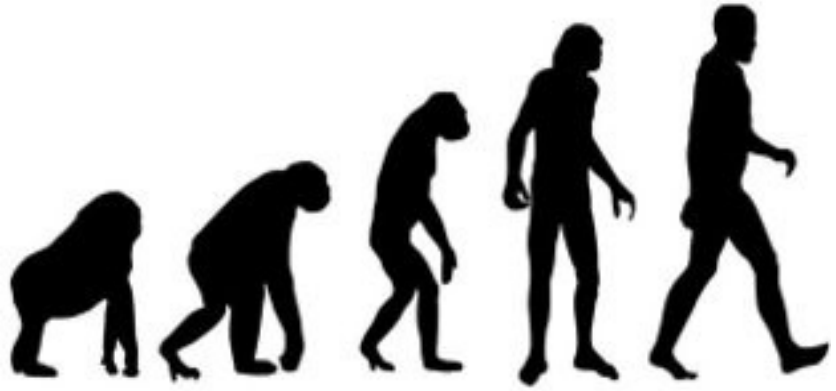


Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6
i	k_0
k	

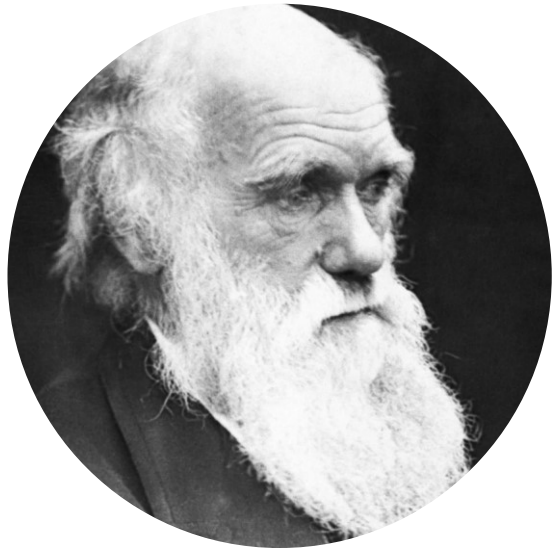
GOAL

GENETIC ALGORITHM

GENETICALGORITHM



- A **genetic algorithm** is an **adaptive heuristic search algorithm** that is inspired by **Charles Darwin's theory of natural evolution** which is **survival of the fittest**.
- **Basic idea of Darwin's theorem:** process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.



"If a population want to thrive, it must improve by itself constantly, it's the survival of the fittest. The best element of the population should inspire the offspring, but the other individuals must not be forgotten in order to maintain some diversity and be able to adapt in case of a variation of the natural environment."-Charles Darwin

Why GA?

GA's do not break easily even if inputs changed slightly or even in the presence of reasonable noise. Offer more robust results than other typical search optimization techniques.

GENETIC ALGORITHM

- Developed by **John Holland(1975).**
- GA's are categorized as **global search heuristics**.
- **GA** are used for **solving optimization problems** generally **maximization or minimization** kind of problems in order to gets best output from a set of input.
- GA's are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as **inheritance, mutation, selection, and crossover**
- GAs have 2 essential components: **1. Survival of the fittest (selection) 2. Variation**

BASICS OF GENETIC ALGORITHM

- GAs begin with a set of k randomly generated states, called the **population.** Each **state, or individual**, is represented as a **string over a finite alphabet—most commonly, a string of 0s and 1s**.
- Each state is rated by the objective function, or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states. In each generation, the fitness of every individual in the population is evaluated.
- A genetic algorithm (or GA) **generates successor states** by **combining two parent states** rather than by modifying a single state.
- The algorithm **terminates** when either a **maximum number of generations has been produced**, or a **satisfactory fitness level has been reached for the population**.
- Produce the next generation of states by “simulated evolution”
  - **Random selection**
  - **Crossover**
  - **Random mutation**

STOCHASTIC OPERATORS OF GENETIC ALGORITHM

A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators

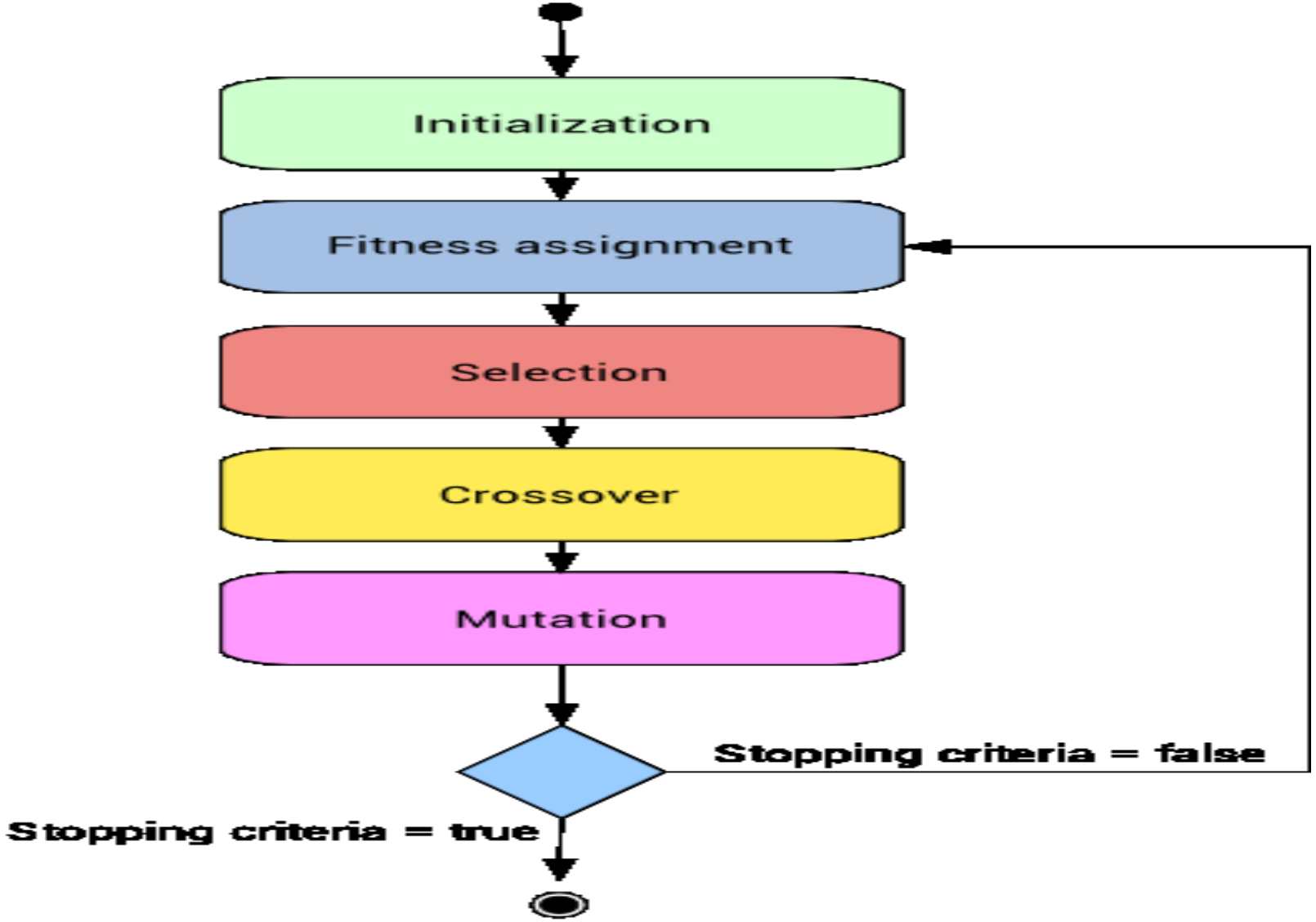
1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass there genes to the successive generations.

2) Crossover Operator(recombination): This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).

3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence.

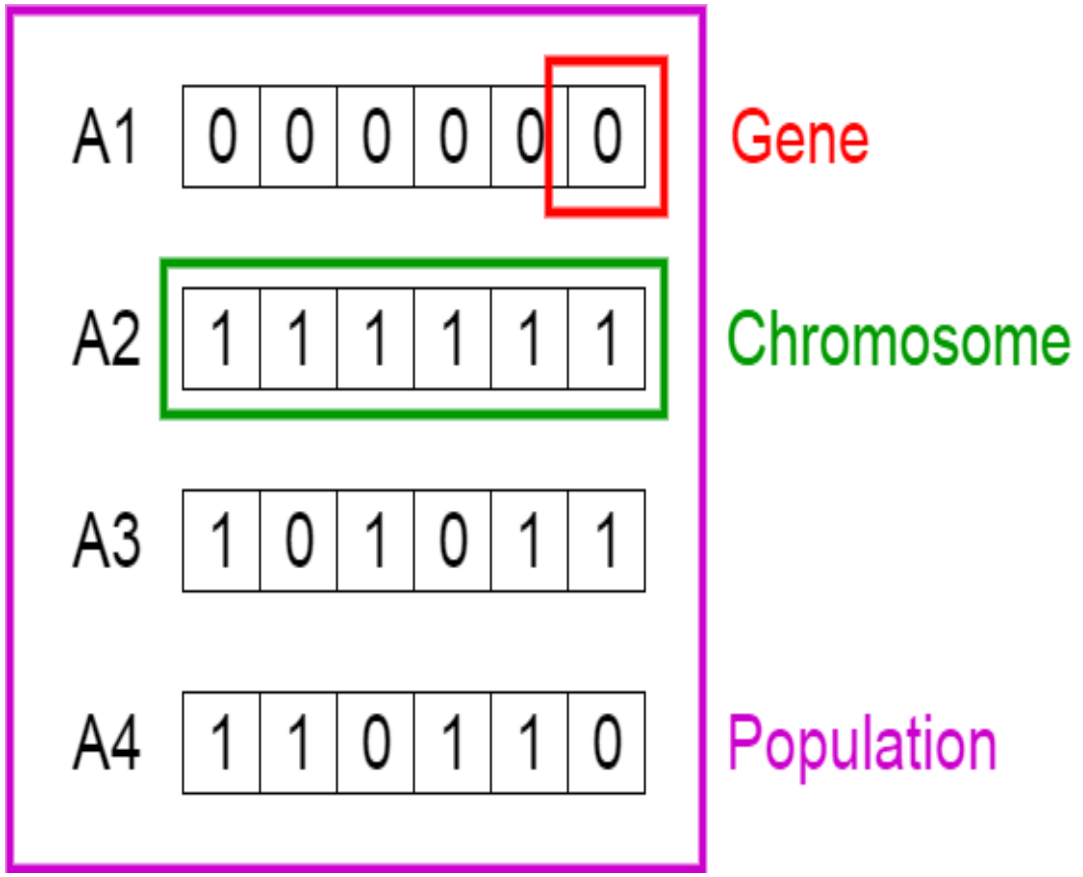
4)Fitness function: Numerical function estimate the quality of each individual.

STAGES OF GENETIC ALGORITHM



INITIAL POPULATION

- Process begins with a set of individuals which is called a **Population**.
- An individual is characterized by a set of parameters (variables/attributes) known as **Genes(solution component)**. Genes are joined into a string to form a **Chromosome** (solution).
- **Individual:** A possible solution of the problem.
- **Allele:** A gene value [eye color=green]
- In a genetic algorithm, the set of genes of an individual is represented using a **string,** in terms of an alphabet. Usually, binary values are used (string of 1s and 0s).



- Initially many individual solutions are **randomly generated** to form an **initial population.**
  The **population size** depends on the **nature of the problem,** but typically contains several hundreds or thousands of possible solutions

CHROMOSOMES REPRESENTATION

- Chromosome representation has great influence in the problem resolution
- Initially represented as binary string
- Nowadays, other possibilities:
  - **Bit strings (0101 ... 1100)**
  - **Real numbers (43.2 -33.1 ... 0.0 89.2)**
  - **Permutations of element (E11 E3 E7 ... E1 E15)**
  - **Lists of rules (R1 R2 R3 ... R22 R23)**
  - **Program elements (genetic programming)**
  - **... any data structure ...**

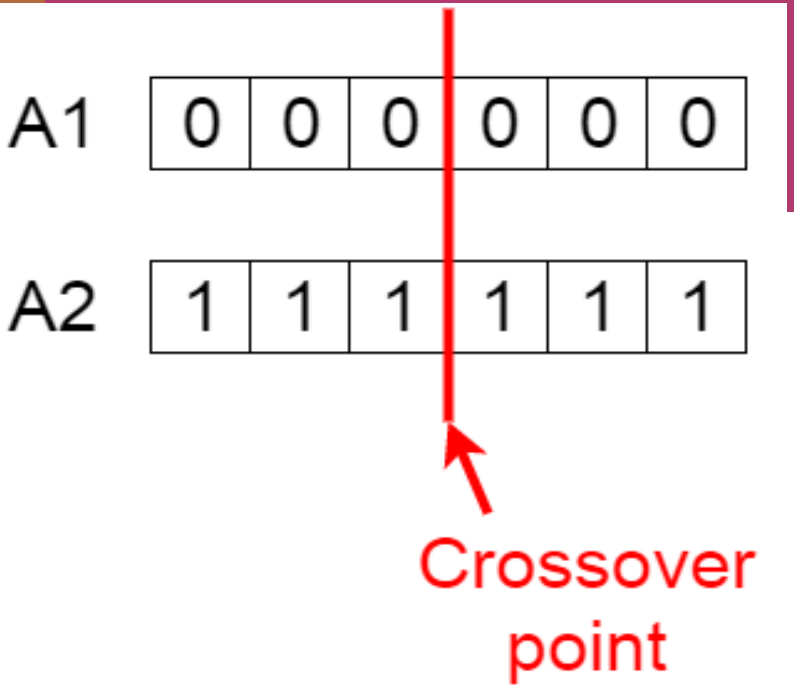
FITNESS FUNCTION

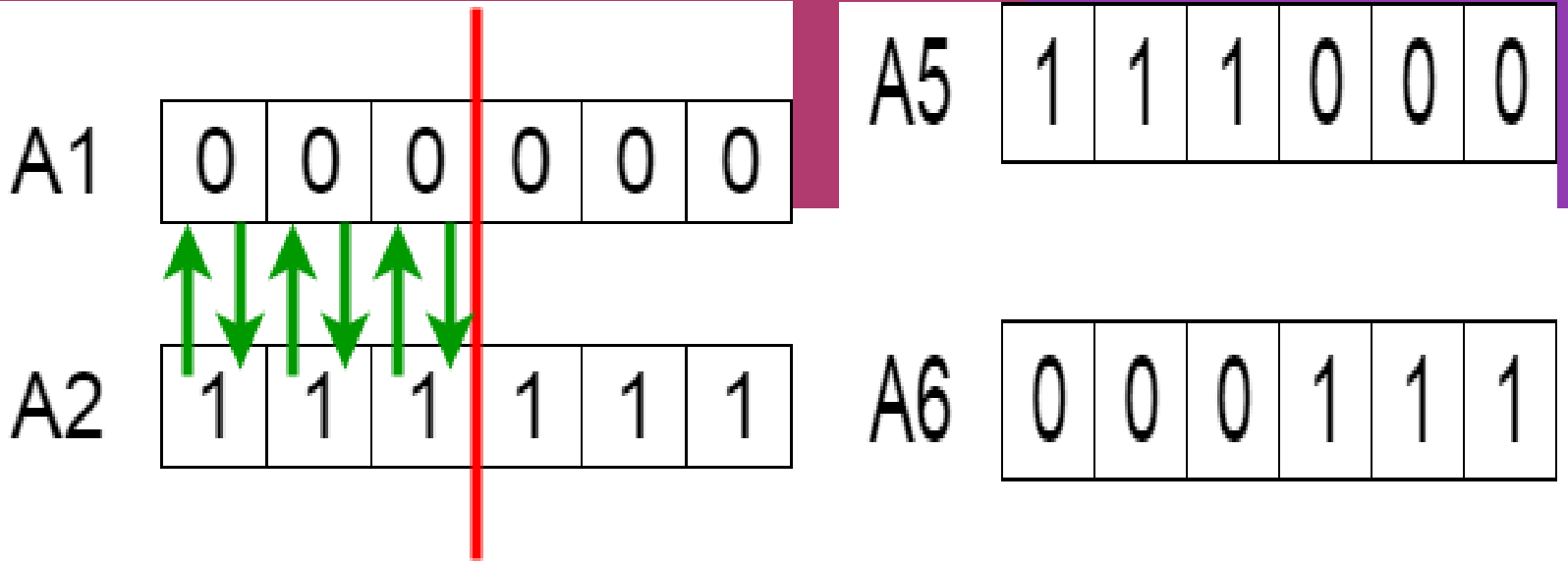
- **Fitness function** determines the fitness of individuals, in general terms, its ability to compete with other individuals and be the final desired output. It gives *fitness scores* to everyone and the probability that an individual will be selected for reproduction is dependent on its fitness score.
- Eg: `fitness score = (number of char correct) / (total number of char)`
- Evaluation function (fitness function) has Higher values for better states. It is opposite to heuristic function, e.g., # non-attacking pairs in 8-queens

SELECTION

- The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.
- During each successive generation, **a proportion of the existing population** is selected to breed a new generation.
- Individual solutions are selected through a ***fitness-based* process**, where fitter solutions (as measured by a fitness function) are typically more likely to be selected.
- There are many variants of selection rule
  - **Culling:** All individuals below a threshold is discarded(converge faster)
- Most s**election functions are stochastic** and designed so that a **small proportion of less fit solutions are selected.** This helps keep the **diversity of the population large, preventing premature convergence** on poor solutions. Popular and well-studied selection methods include **roulette wheel selection** and **tournament selection.**

CROSSOVER

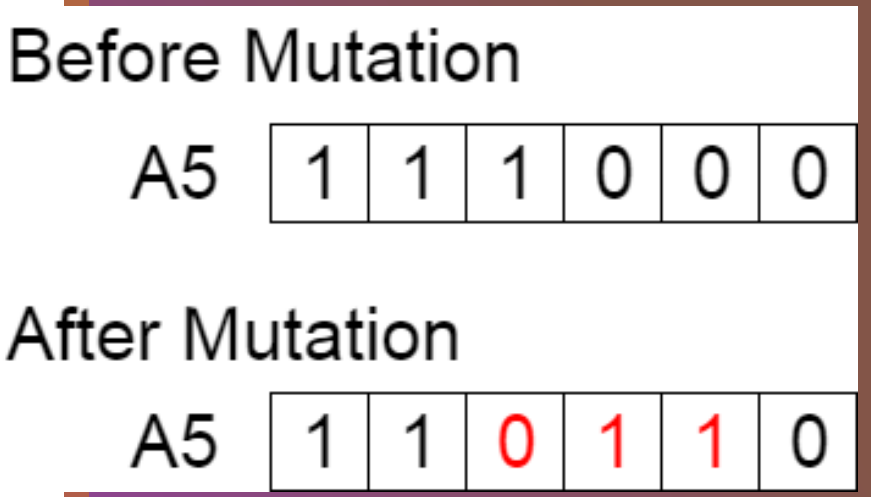




- **Crossover** is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.
- There are different types of cross overs like **single point crossover, two point, uniform, heuristic, arithmetic**
- **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is

MUTATION

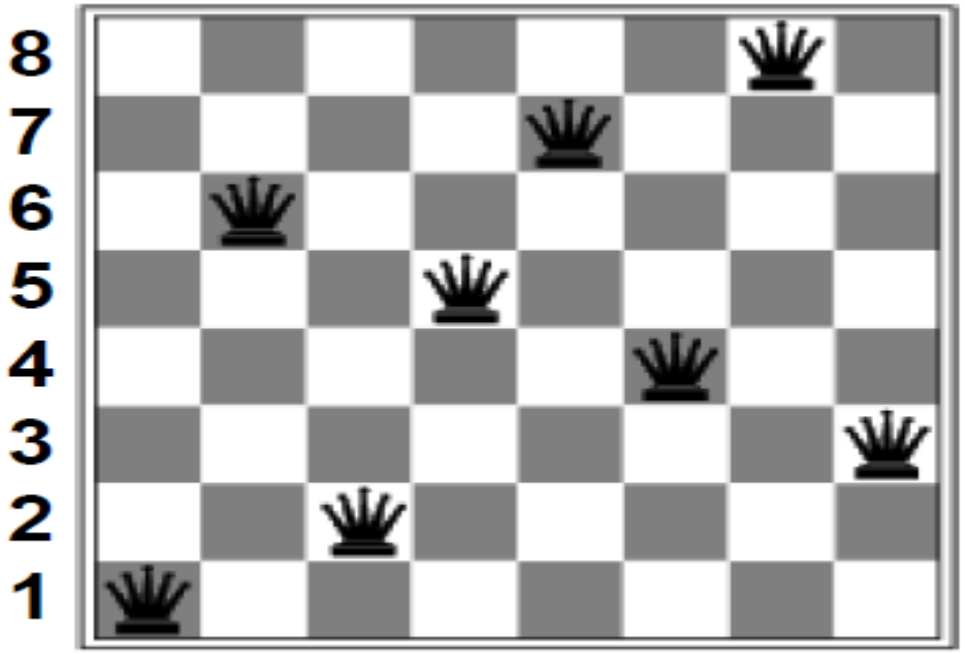
- In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.
- Mutation occurs to **maintain diversity within the population** and **prevent premature convergence.**
- Mutation occurs according to user definable mutation probability which is set to a **fairly lower value that is 0.01.**
- It prevent population from **stagnating at a local optima.**
- **Types:** Flip Bit, Boundary, Uniform, Non-Uniform, Gaussian.



TERMINATION

- The algorithm terminates when either a **maximum number of generations** has been produced, or a **satisfactory fitness level has been reached for the population.**
- The algorithm terminates if the **population has converged** (does not produce offspring which are significantly different from the previous generation).

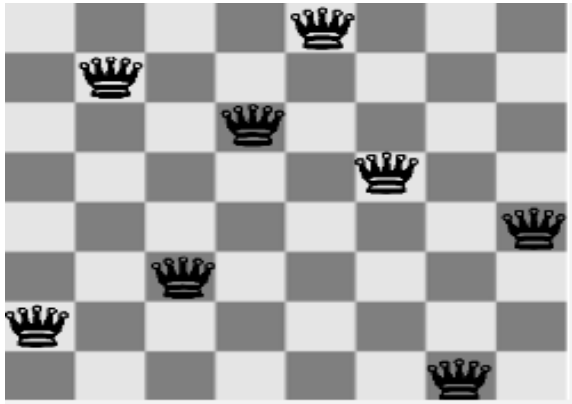
8-QUEENS USING GENETIC ALGORITHMS



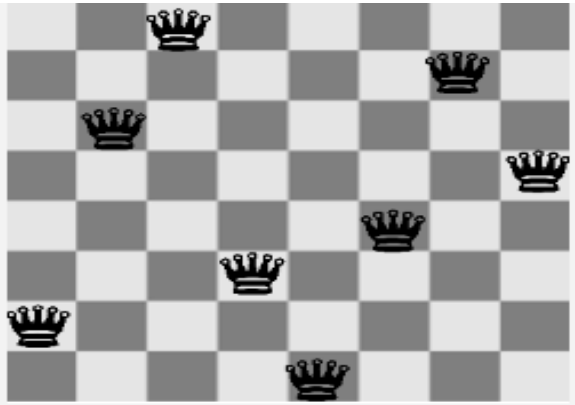
String representation
16257483

States: assume each queen has its own column, represent a state by listing a row where the queen is in each column (digits 1 to 8)

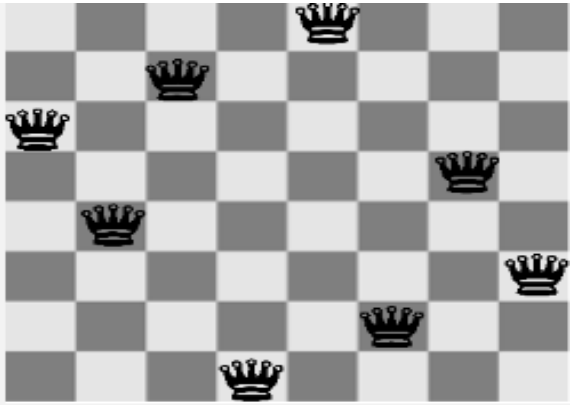
DIFFERENT STATE REPRESENTATION



[16257403]

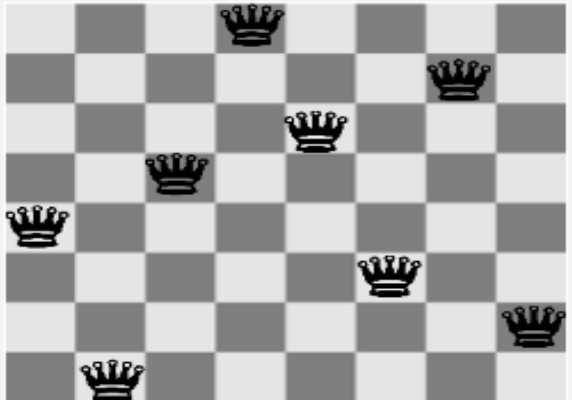


[15720364]

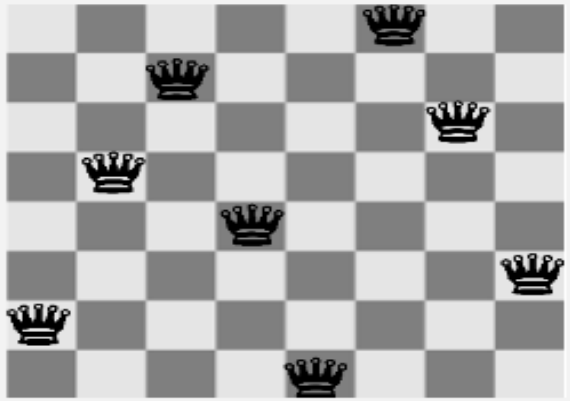


[53607142]

[30475261]



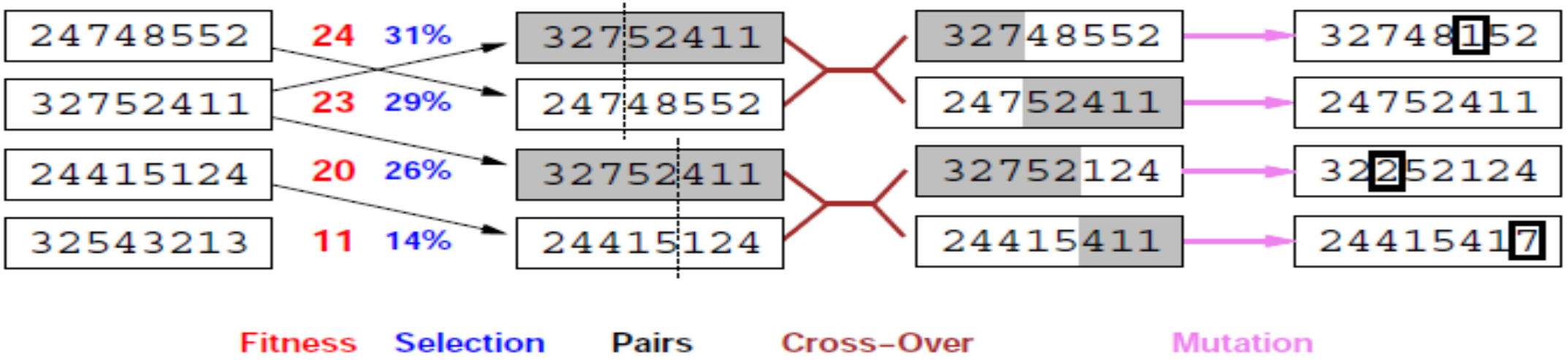
[14630752]



FITNESS FUNCTION OF 8-QUEENS

- **Fitness function:** number of non-attacking pairs of queens (min = 0, max = 8 ×7/2 = 28)
- **Fitness fn is proportional to the number of clashes amongst the queens. There are 28 clashes possible in an 8 x 8 chessboard. Let clashes=number of clashing queens**
- To determine clashes, each queen must be checked for
  - Row clashes
  - Column clashes
  - Diagonal clashes
- **fitness=28−clashes**
- **Therefore, if an individual has high fitness, it will have lower number of clashes. Then, any individual with the maximum fitness will be having least number of clashes.**
- Fitness of the state(16257483) given is 27
- Find fitness of following states
  - 24748552
  - 32752411
  - 24415124
  - 32543213

GENETIC ALGORITHMS



2 pairs of 2 states randomly selected based on fitness. Random crossover points are selected.

FITNESS SCORE PERCENTAGE CALCULATION

24/(24+23+20+11) = 31%

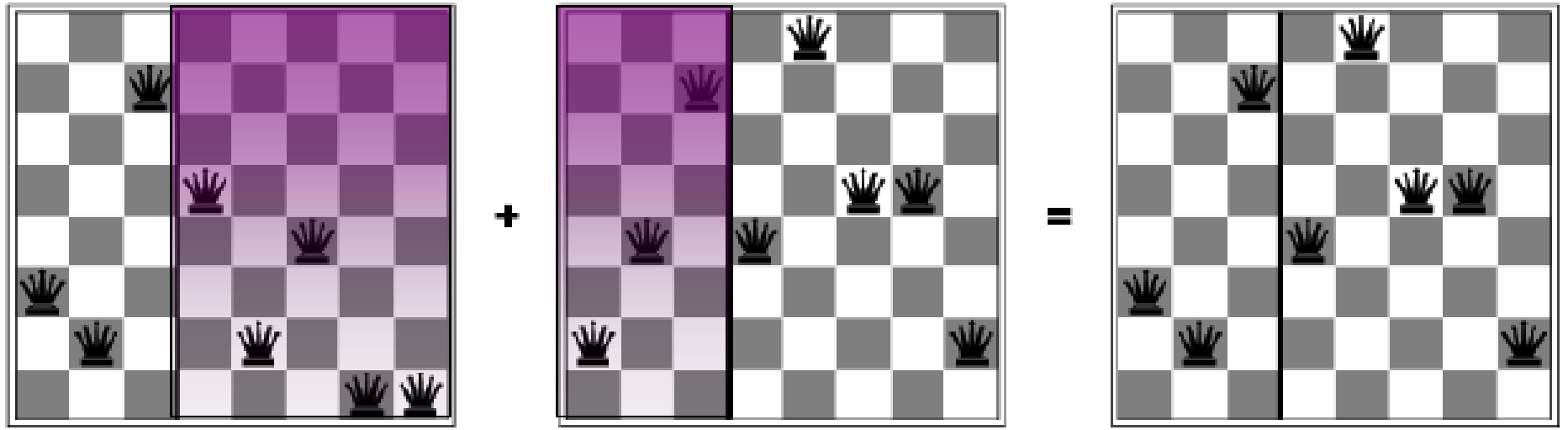
23/(24+23+20+11) = 29% etc

- **2 Pairs randomly selected for reproduction according to probabilities**
- **Crossoverpoint is chosen randomly from positions in string(3rd and 5th)**
- **Mutation corresponds to choosing a queen at random and moving it to a random square in its column.**

TERMINATION

keep a limit of iteration. most optimal solution can be described as the **most fit board position**.

EFFECT OF GA



IMPORTANCE OF REPRESENTATION

- Has the effect of “jumping” to a completely different new part of the search space (quite non-local)
- Parts we swap in crossover should result in a well-formed solution (and in addition better be meaningful)
- consider what would happen with binary representation (where position re- quires 3 digits)
- also, chosen representation reduced search space considerably (compared to representing each square for example)

EXAMPLE 2: MAXONE PROBLEM

- **PROBLEM DEFINITION:** maximize the number of ones in a string of l binary digits
- An individual is encoded (naturally) as a string of *l* binary digits
- The **fitness function *f*** of a candidate solution to the MAXONE problem is the number of ones in its genetic code
- We start with a population of *n* random strings. Suppose that *l* = 10 and *n* = 6

INITIALIZATION

- We toss a fair coin 60 times and get the following initial population:

s_1 = 1111010101 $f(s_1) = 7$

s_2 = 0111000101 $f(s_2) = 5$

s_3 = 1110110101 $f(s_3) = 7$

s_4 = 0100010011 $f(s_4) = 4$

s_5 = 1110111101 $f(s_5) = 8$

s_6 = 0100110000 $f(s_6) = 3$

SELECTION

- we apply fitness proportionate selection with the roulette wheel method:
- Individual *i* will have a $\frac{f(i)}{\sum_i f(i)}$ probability to be chosen
- repeat the extraction as many times as the number of individuals we need to have the same parent population size (6 in our case)
- after performing selection, we get the following population:

s_1` = 1111010101 (s_1)=7/7+5+7+4+8+3=7/34

s_2` = 1110110101 (s_3)=7/34

s_3` = 1110111101 (s_5)=8/34

s_4` = 0111000101 (s_2)=5/34

s_5` = 0100010011 (s_4)=4/34

s_6` = 1110111101 (s_5)=8/34

CROSSOVER

- mate strings for crossover. For each couple we decide according to crossover probability (for instance 0.6) whether to actually perform crossover or not
- Suppose that we decide to actually perform crossover only for couples (s_1`, $s_2$`) and (s_5`, $s_6$`). For each couple, we randomly extract a crossover point, for instance 2 for the first and 5 for the second
- Before crossover:

s1` = 1111010101

s2` = 1110110101

s5` = 0100010011

s6` = 1110111101

- After crossover:

s1`` = 1110110101

s2`` = 1111010101

s5`` = 0100011101

s6`` = 1110110011

MUTATION

- The final step is to apply random mutation: for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1)
- Before applying mutation:

s_1`` = 1110110101

s_2`` = 1111010101

s_3`` = 1110111101

s_4`` = 0111000101

s_5`` = 0100011101

s_6`` = 1110110011

MUTATION

- After applying mutation:

$s_1''' = 1110100101 \quad f(s_1''') = 6$

$s_2''' = 1111110100 \quad f(s_2''') = 7$

$s_3''' = 1110101111 \quad f(s_3''') = 8$

$s_4''' = 0111000101 \quad f(s_4''') = 5$

$s_5''' = 0100011101 \quad f(s_5''') = 5$

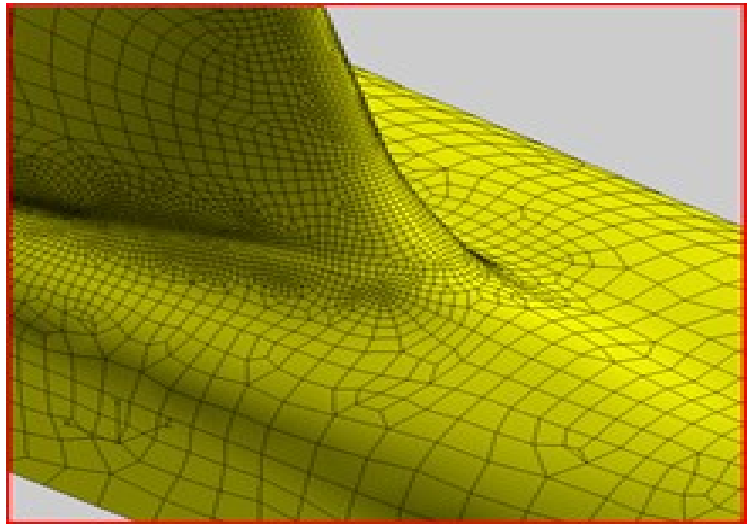
$s_6''' = 1110110001 \quad f(s_6''') = 6$

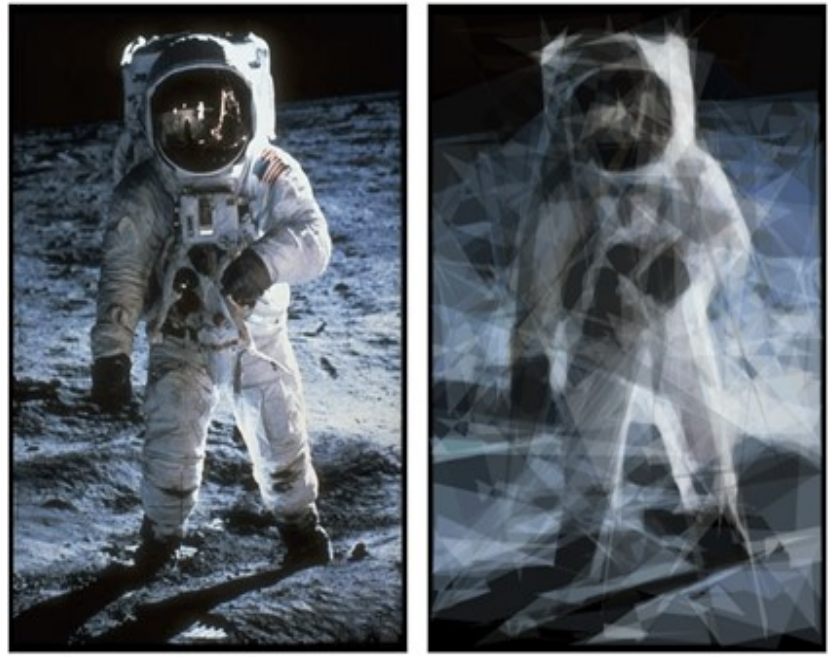
In one generation, the total population fitness changed from 34 to 37, thus improved by ~9%

At this point, we go through the same process all over again, until a stopping criterion is met

APPLICATIONS

- Design jet engines.
- Draw criminals.
- Program computers.
- Criminal-likeness Reconstruction
- Image compression – evolution





THANK YOU