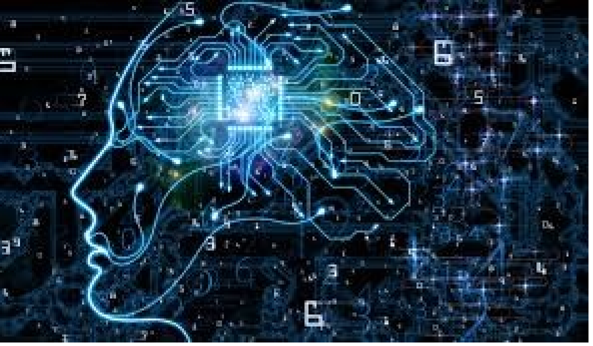
CSE-4101 Artificial Intelligence

Search Strategy

Chapter-2



Today’s class

- Search
- Goal-based agents
- Representing states and operators
- Example problems
- Generic state-space search algorithm
- Specific algorithms
  - Breadth-first search
  - Depth-first search
  - Uniform cost search
  - Depth-first iterative deepening
- Example problems revisited

Search Problem

The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state $g \in G$. A search problem is represented by a 4-tuple $\{S, s^0, A, G\}$.

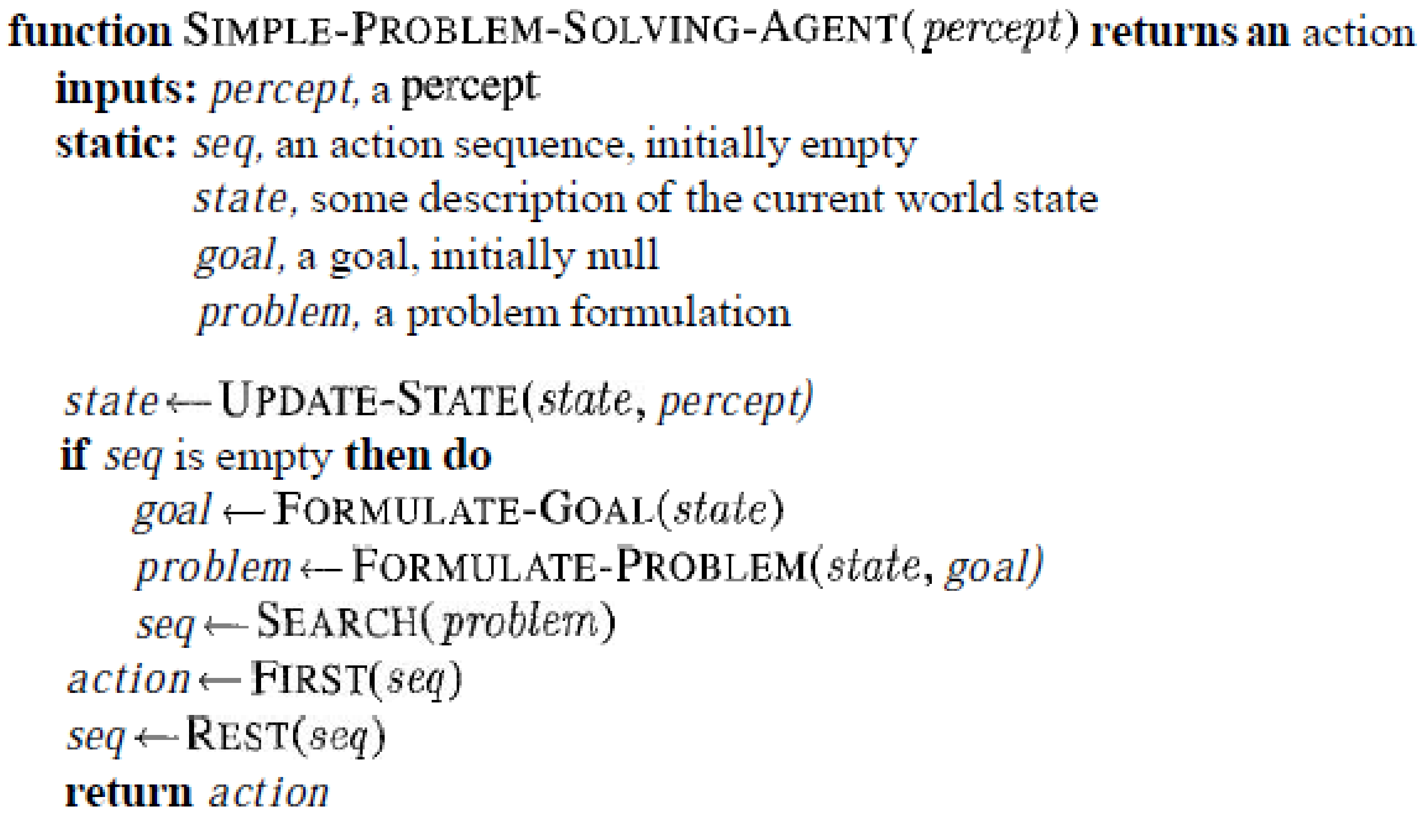
S: set of states

$s^0 \in S$: initial state

A: S S operators/ actions that transform one state to another state

G : goal, a set of states. $G \subseteq S$

A simple problem-solving agent



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    inputs: percept, a percept
    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
```

- It first formulates a goal and a problem,
- Searches for a sequence of actions that would solve the problem, and executes actions
- One at a time. When this is complete, it formulates another goal and starts over.
- When it is executing the sequence it ignores its percepts: it assumes that the solution it has found will always work.

Problem formulation/Searching process

The generic searching process can be very simply described in terms of the following steps:

Do until a solution is found or the state space is exhausted.

1. Check the current state
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state

If it is not, the new state becomes the current state and the process is repeated

Basic concepts

- **<u>State</u>:** finite representation of the world that you want to explore at a given time

.

A problem can be defined formally by four components

- ❑ **An <u>initial state</u>** is the description of the starting configuration of the agent/The problem at the beginning.
- ❑ **<u>Goal state</u>:** desired end state (can be several)/ test to determine if the goal has been reached.
- ❑ **An <u>action</u> or an <u>Operator</u>**: a function that transforms a state into another (also called rule, transition, successor function, production, action).
- ❑ **Path Cost:** The cost of a plan is referred to as the **path cost**

The path cost is a positive number, and a common path cost may be the sum of the costs of the steps in the path.

Search Problem

- The sequence of actions is called a solution plan.
- Solution path is a path from the initial state to a goal state.
- P = {a^0, a^1, … , a^N} which leads to traversing a number of states {s^0, s^1, … , $s^{N+1} \in G$}.
- A sequence of states is called a path.
- The cost of a path is a positive number.
- In many cases the path cost is computed by taking the sum of the costs of each action.

Illustration of a search process

We will now illustrate the searching process with the help of an example. Consider the problem depicted in Figure

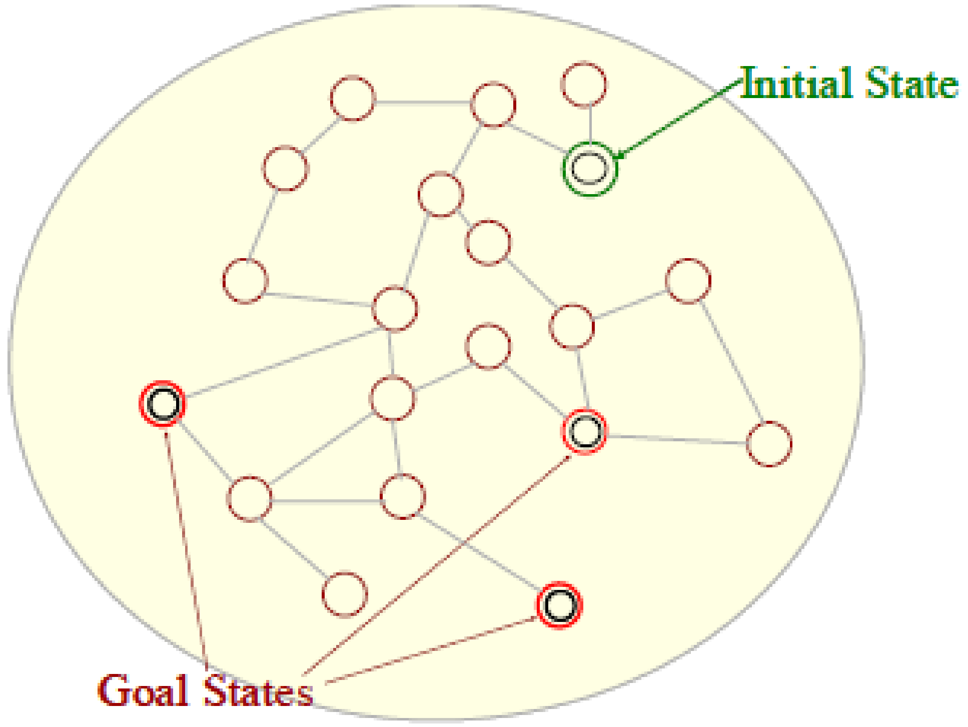


Figure-1

- ❑ s^0 is the initial state.
- ❑ The successor states are the adjacent states in the graph.
- ❑ There are three goal states.

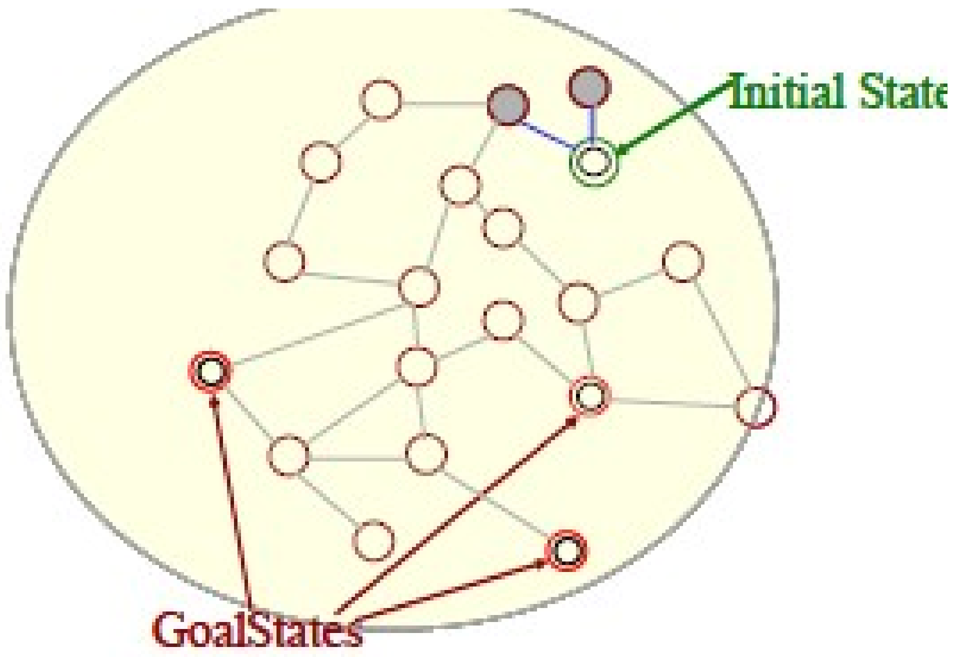


Figure-2

The two successor states of the initial state are generated.

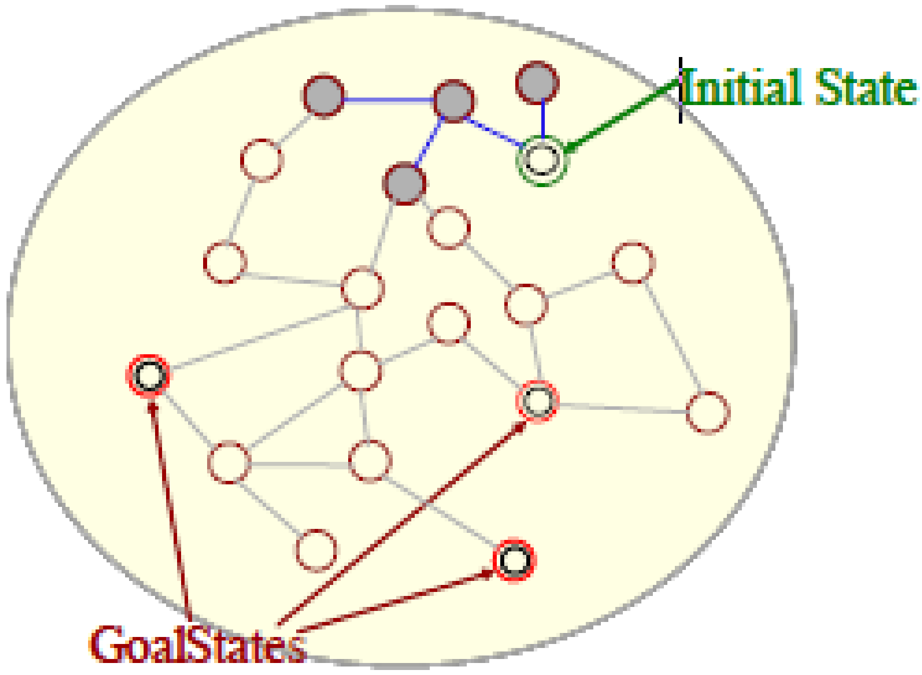


Figure-3

The successors of these states are picked and their successors are generated.

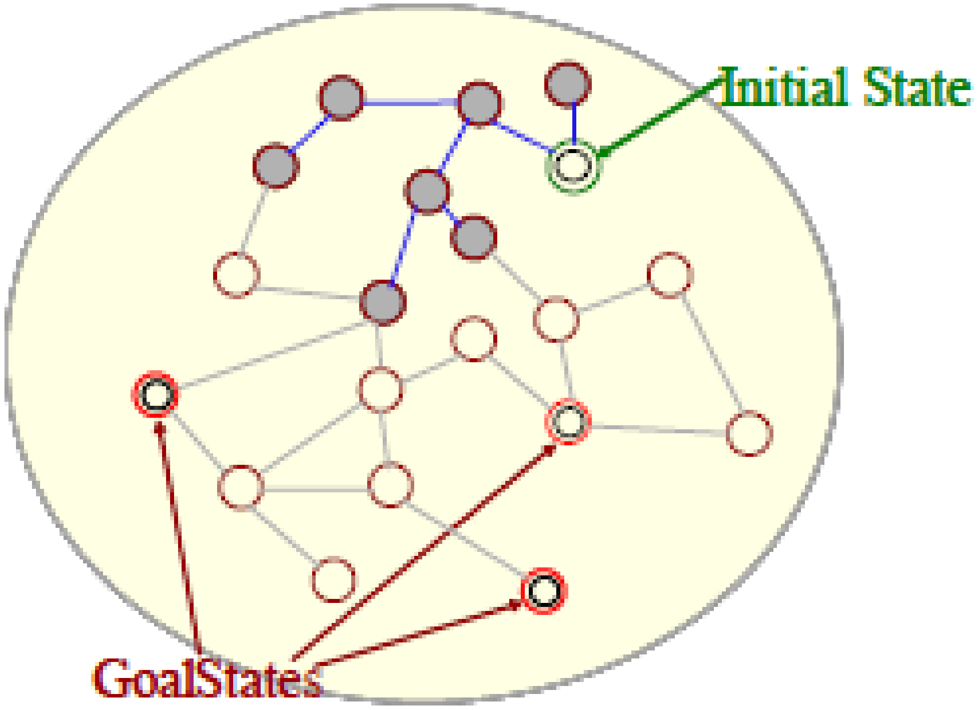


Figure-4

Successors of all these states are generated.

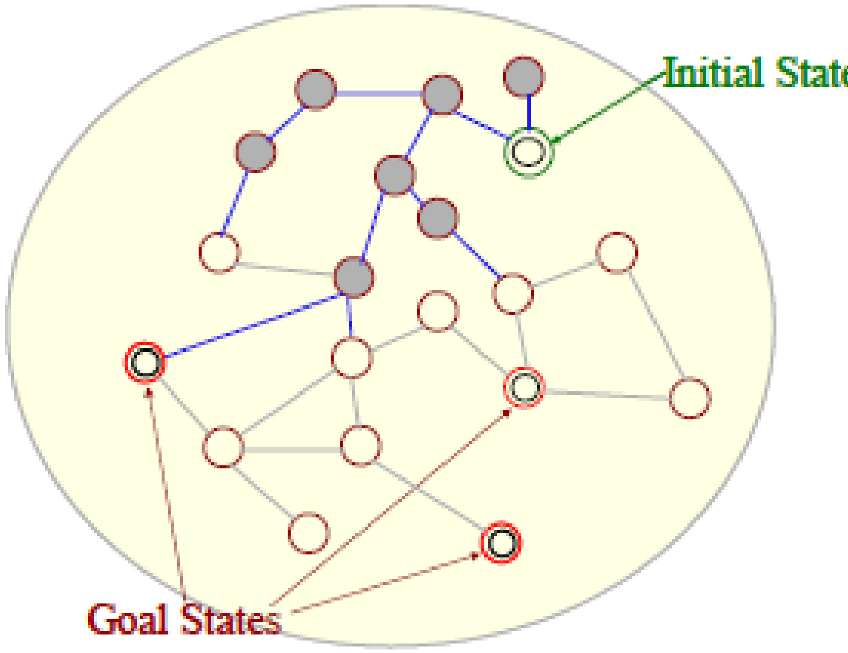
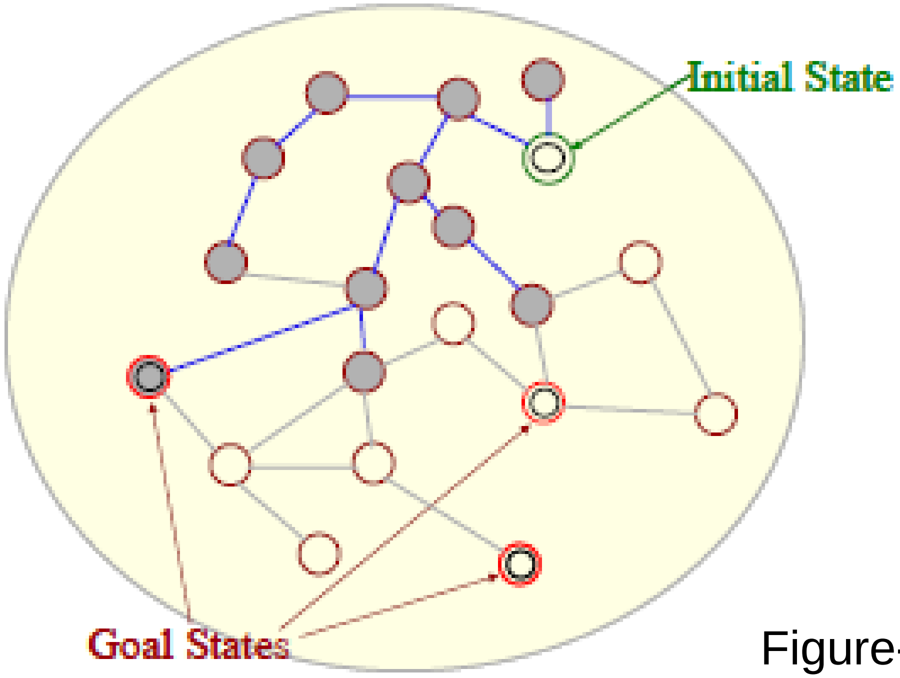


Figure-5



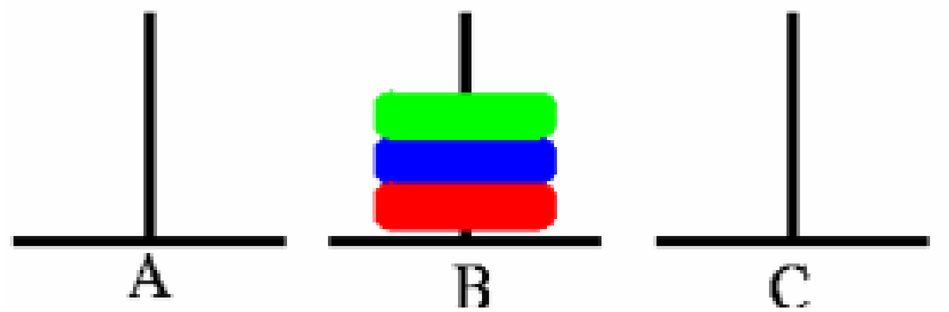
A goal state has been found.

The successors are generated.

Figure-6

Example problem: Pegs and Disks problem

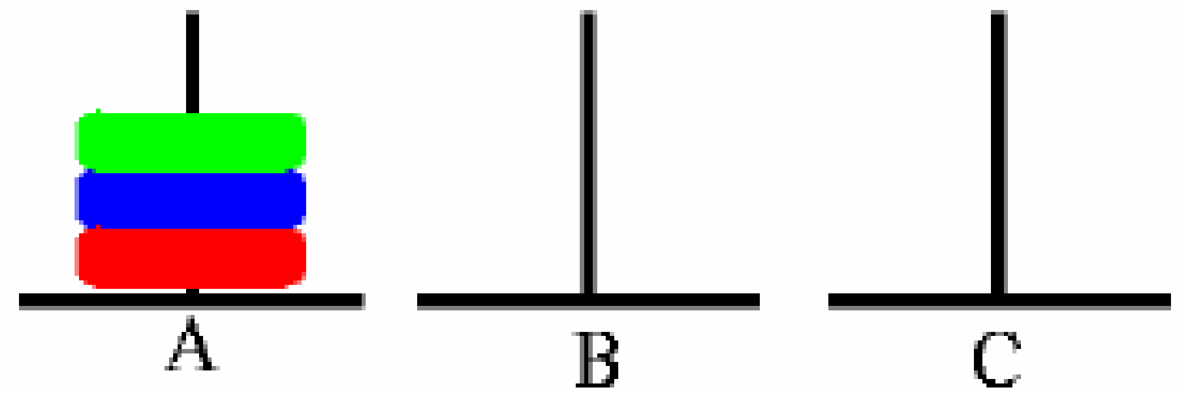
Consider the following problem. We have 3 pegs and 3 disks.



Operators: one may move the topmost disk on any needle to the topmost position to any other needle

In the goal state all the pegs are in the needle B as shown in the figure below..

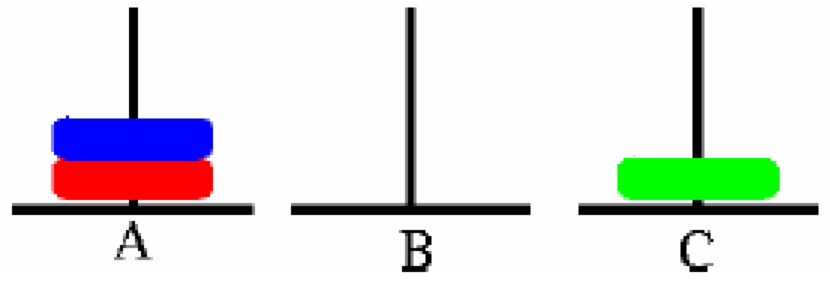
The initial state is illustrated below.

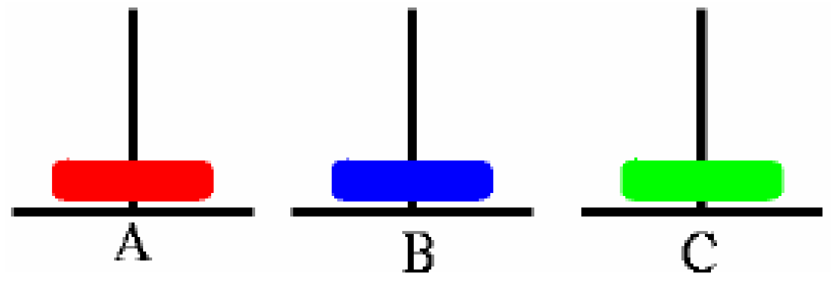


Now we will describe a sequence of actions that can be applied on the initial state.

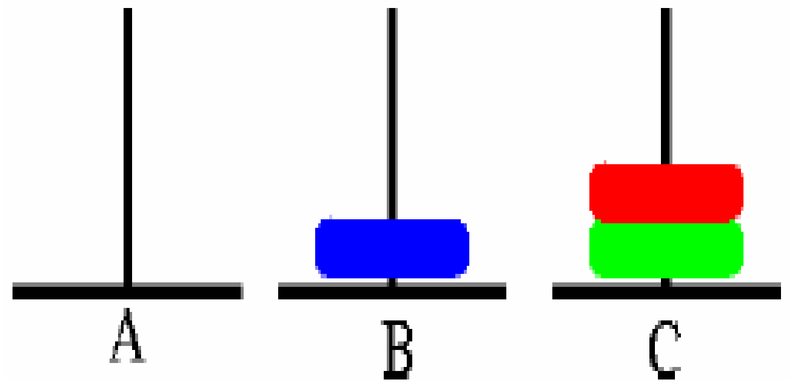
Step 1: Move A → C

Step 2: Move A → B

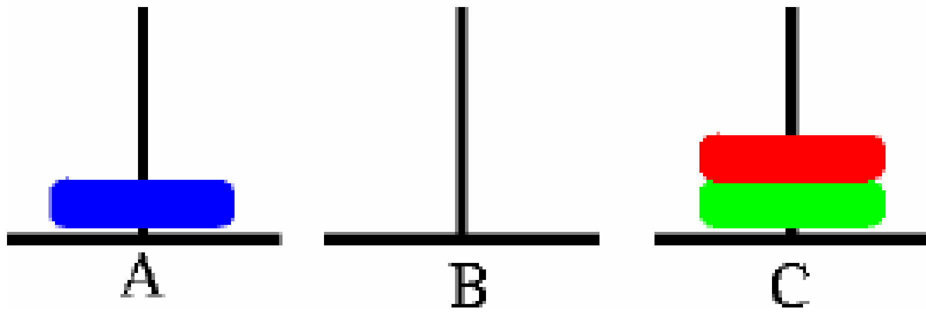




Step 3: Move A → C



Step 4: Move B → A



Step 5: Move C → B

Step 6: Move A → B

Step 7: Move C → B

Another search problem :8 puzzle

- In the 8-puzzle problem we have a 3×3 square board and 8 numbered tiles.
- The board has one blank position.
- Bocks can be slid to adjacent blank positions.
- We can alternatively and equivalently look upon this as the movement of the blank position up, down, left or right.
- The objective of this puzzle is to move the tiles starting from an initial position and arrive at a given goal configuration.

Problem Definition - Example, 8 puzzle

5	4	
6	1	8
7	3	2

Initial State

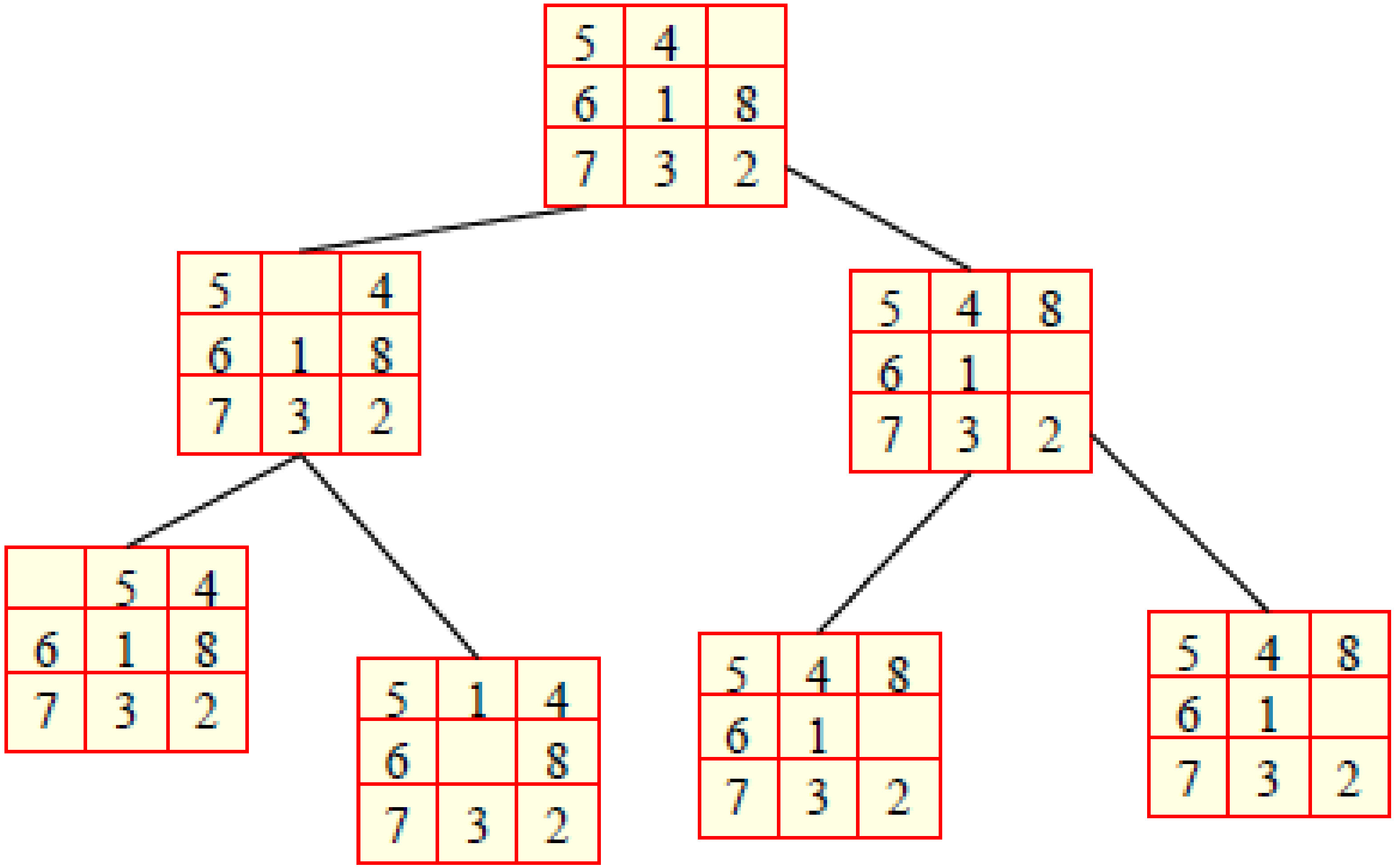
1	4	7
2	5	8
3	6	

Goal State

In the 8-puzzle problem we have a 3×3 square board and 8 numbered tiles

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

Problem Definition - Example, 8 puzzle

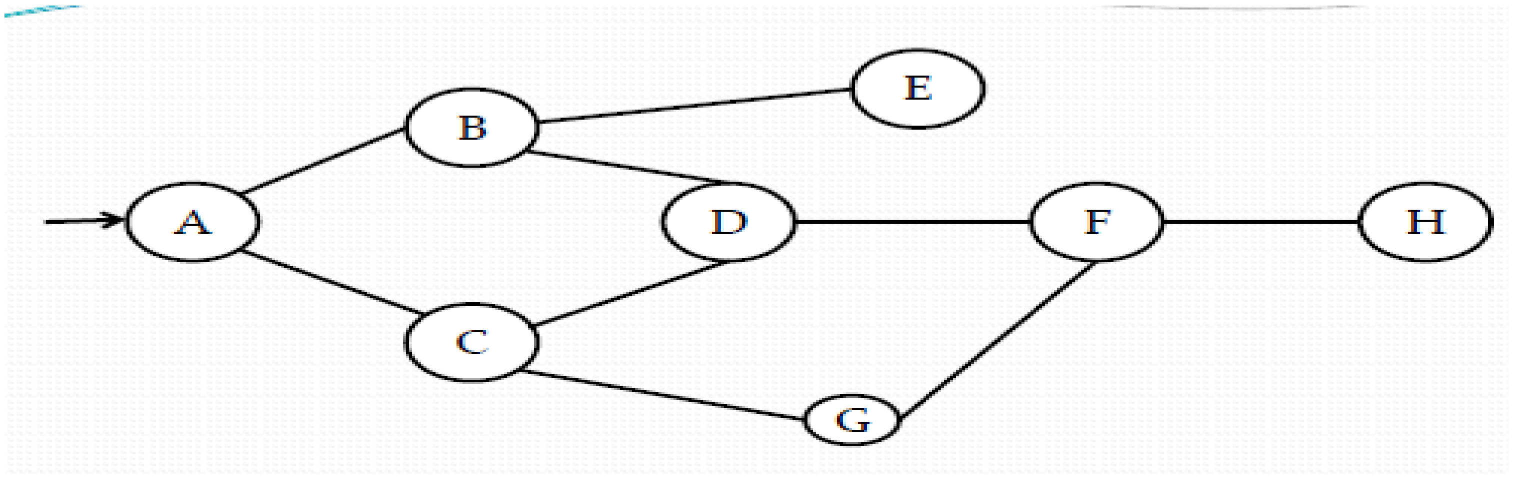


A small portion of the state space of 8-puzzle is shown below. Note that we do not need to generate all the states before the search begins. The states can be generated when required.

Uninformed Search/Blind Search Control Strategy

- Also known as “blind search,” uninformed search strategies use no information about the likely “direction” of the goal node(s)
- Do not have additional info about states beyond problem def.
- Total search space is looked for solution
- No info is used to determine preference of one child over other.
- Example: 1. Breadth First Search(BFS), Depth First Search(DFS), Depth Limited Search (DLS)

Uninformed Search/Blind Search Control Strategy



State Space without any extra information associated with each state

Breadth First Search (BFS)

Algorithm:

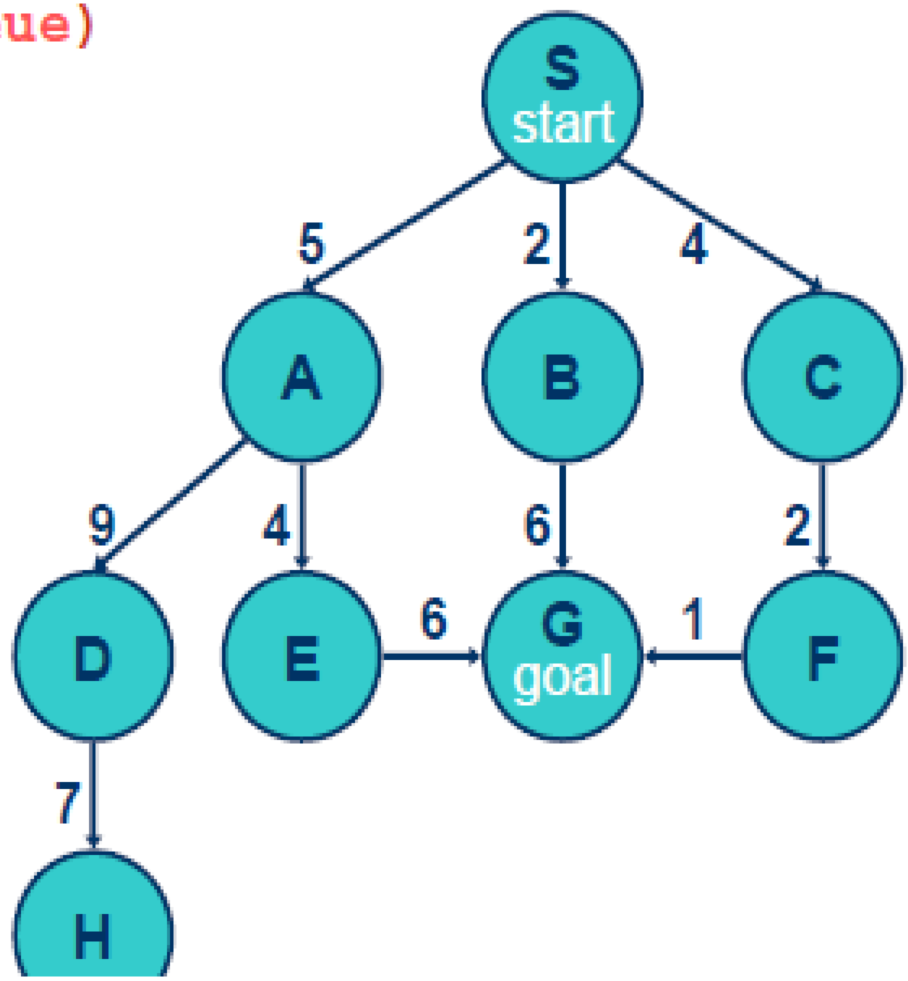
```
 1. Create a variable QUEUE, put the starting node on
                    Queue and set it to initial state.
 2.Loop: Until a Goal State is found or QUEUE  is empty:
        if nodes is empty then return failure
          node := Remove-Front (QUEUE)
         if Goal-Test[problem] applied to State(node) succeeds
            then return node
          new-nodes := Expand (QUEUE,Operators[problem]))
          nodes := Insert-At-End-of-Queue(new-nodes)
        end
```

Breadth First Search

`generalSearch(problem, queue)`

# of nodes tested: 0, expanded: 0

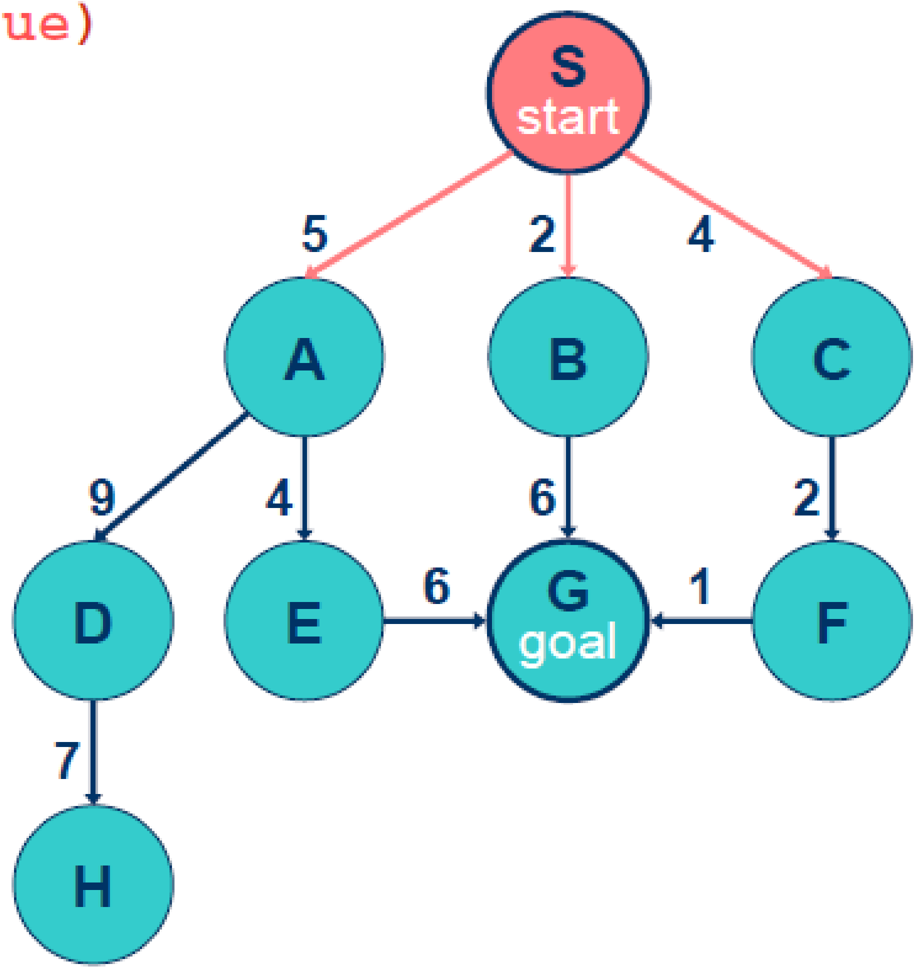
expnd. node	nodes list
	{S}



```
generalSearch(problem, queue)
```

# of nodes tested: 1, expanded: 1

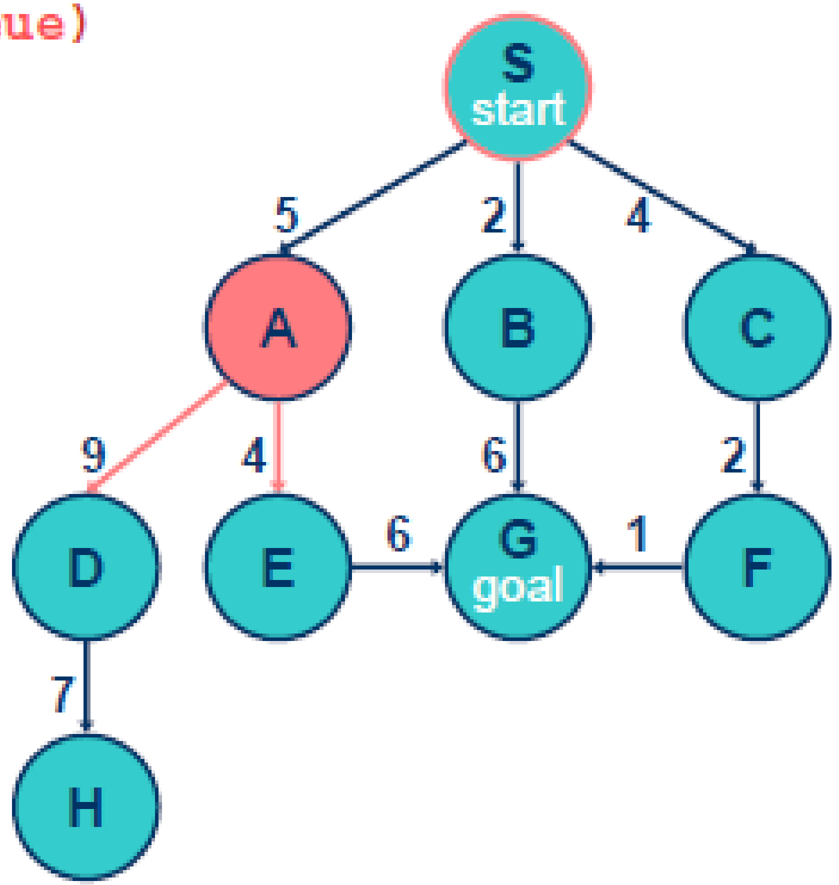
expnd. node	nodes list
	{S}
S not goal	{A,B,C}



`generalSearch(problem, queue)`

# of nodes tested: 2, expanded: 2

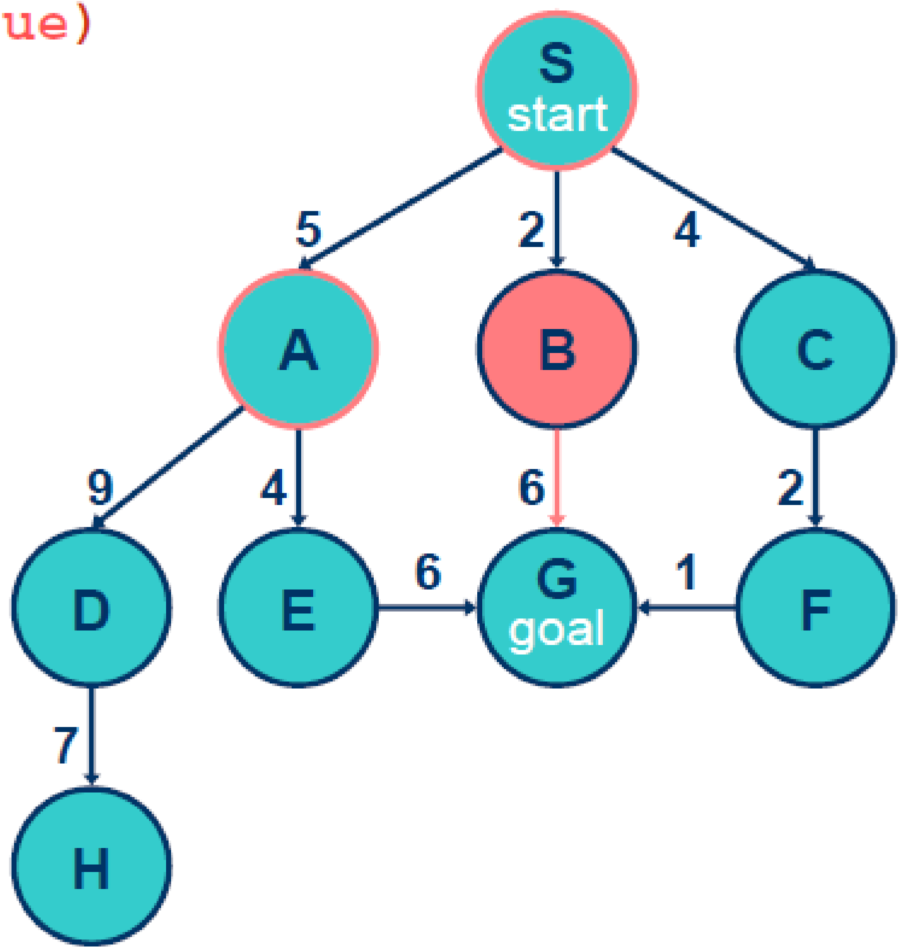
expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{B,C,D,E}



`generalSearch(problem, queue)`

# of nodes tested: 3, expanded: 3

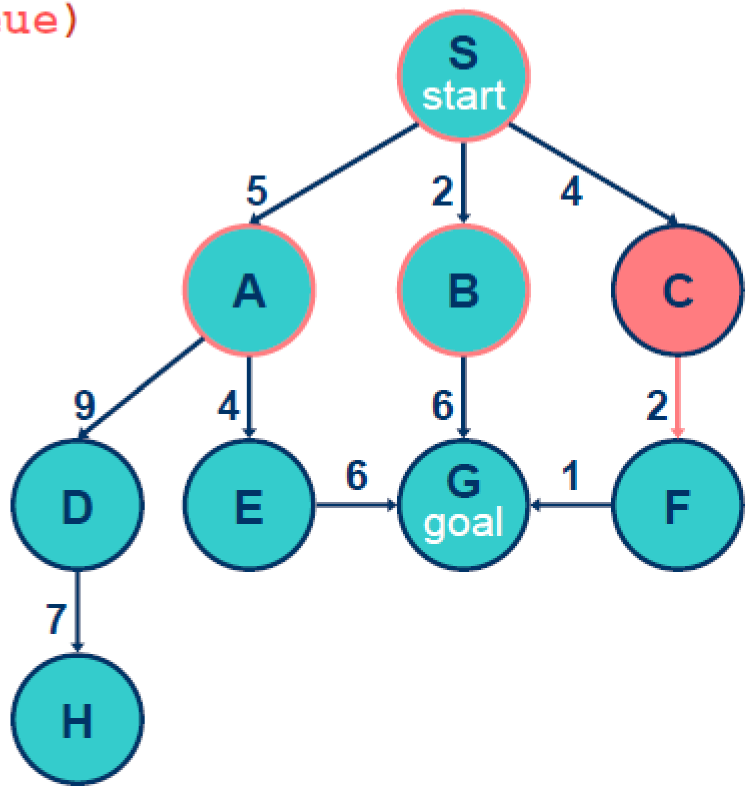
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}



`generalSearch(problem, queue)`

# of nodes tested: 4, expanded: 4

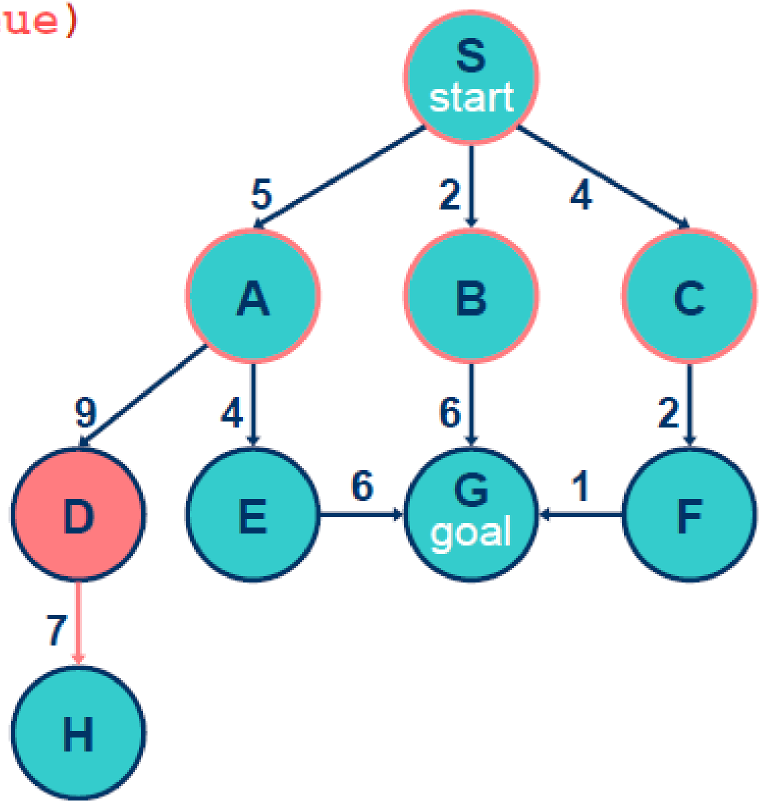
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}



`generalSearch(problem, queue)`

# of nodes tested: 5, expanded: 5

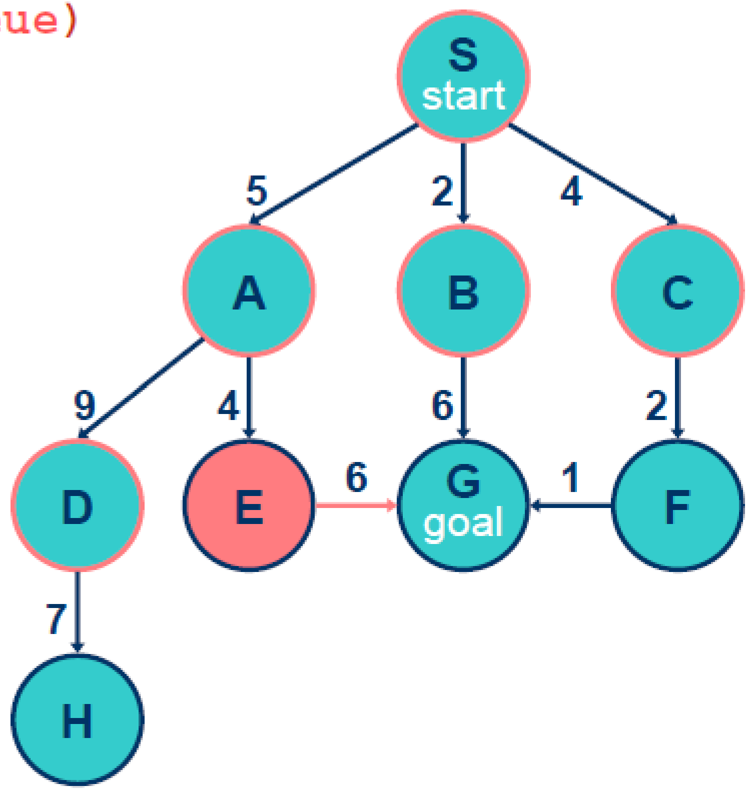
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D not goal	{E,G,F,H}



`generalSearch(problem, queue)`

# of nodes tested: 6, expanded: 6

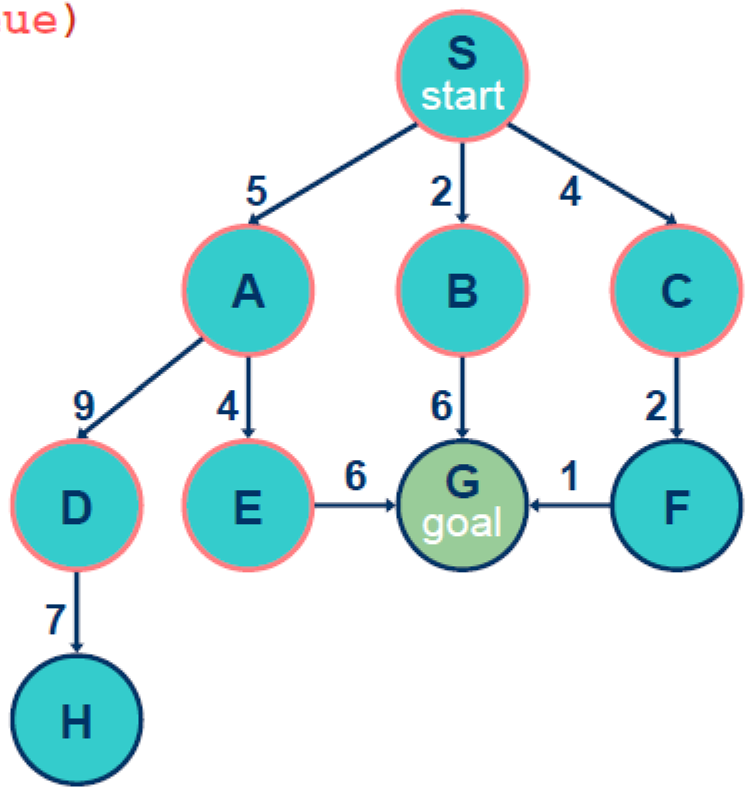
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E not goal	{G,F,H,G}



`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

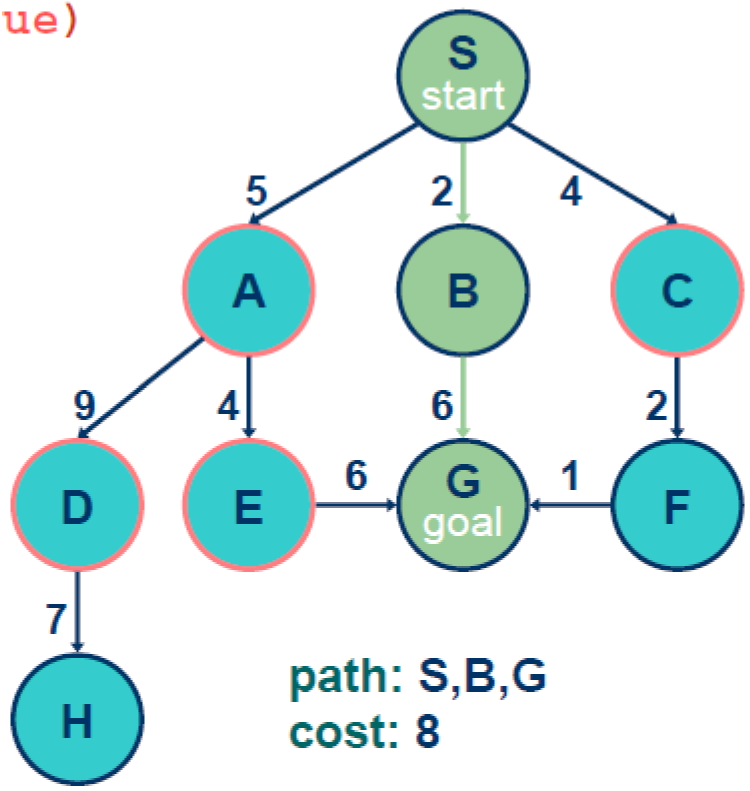
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G goal	{F,H,G} no expand



`generalSearch(problem, queue)`

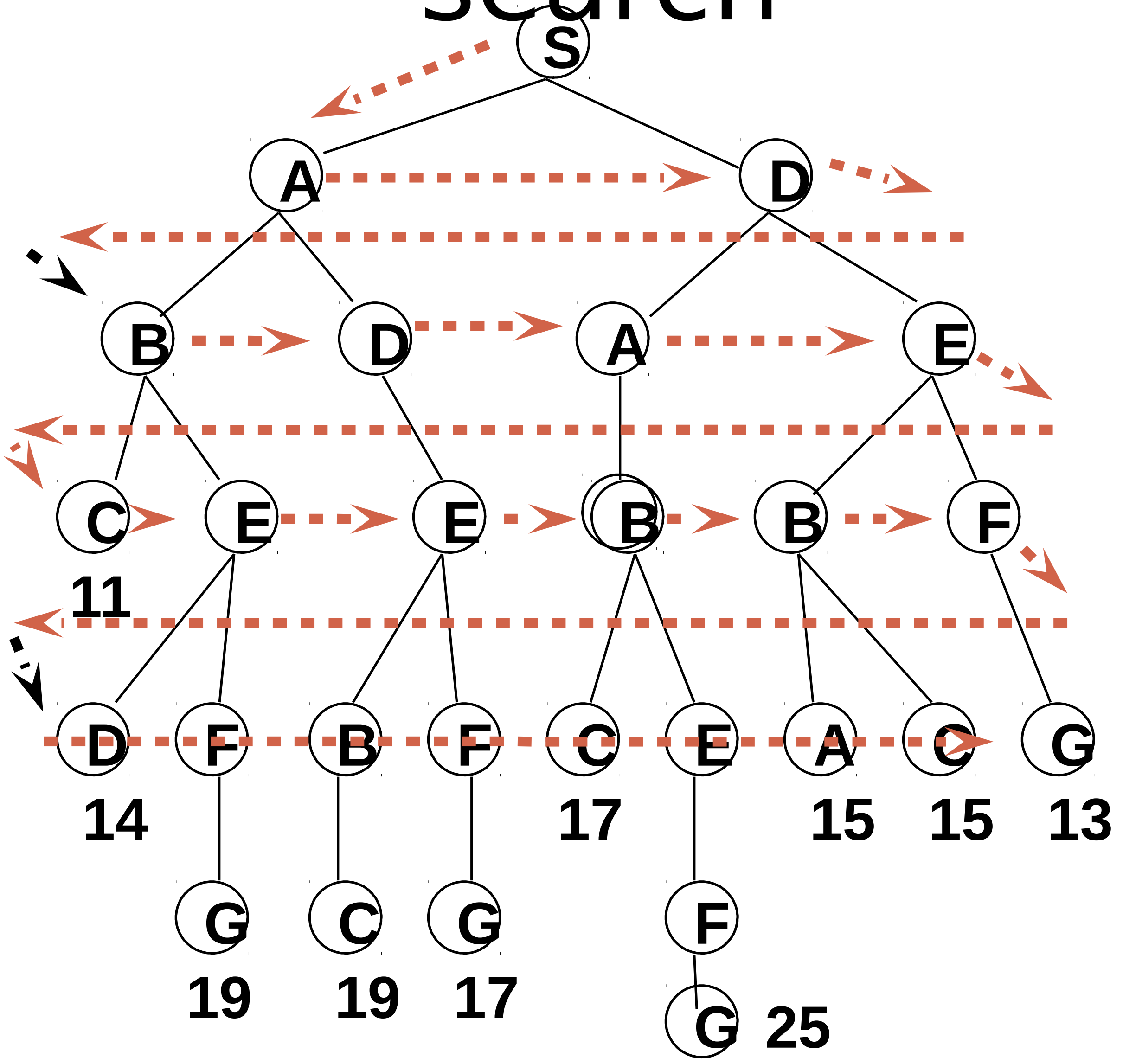
# of nodes tested: 7, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}



path: S,B,G
cost: 8

Another Breath-first search



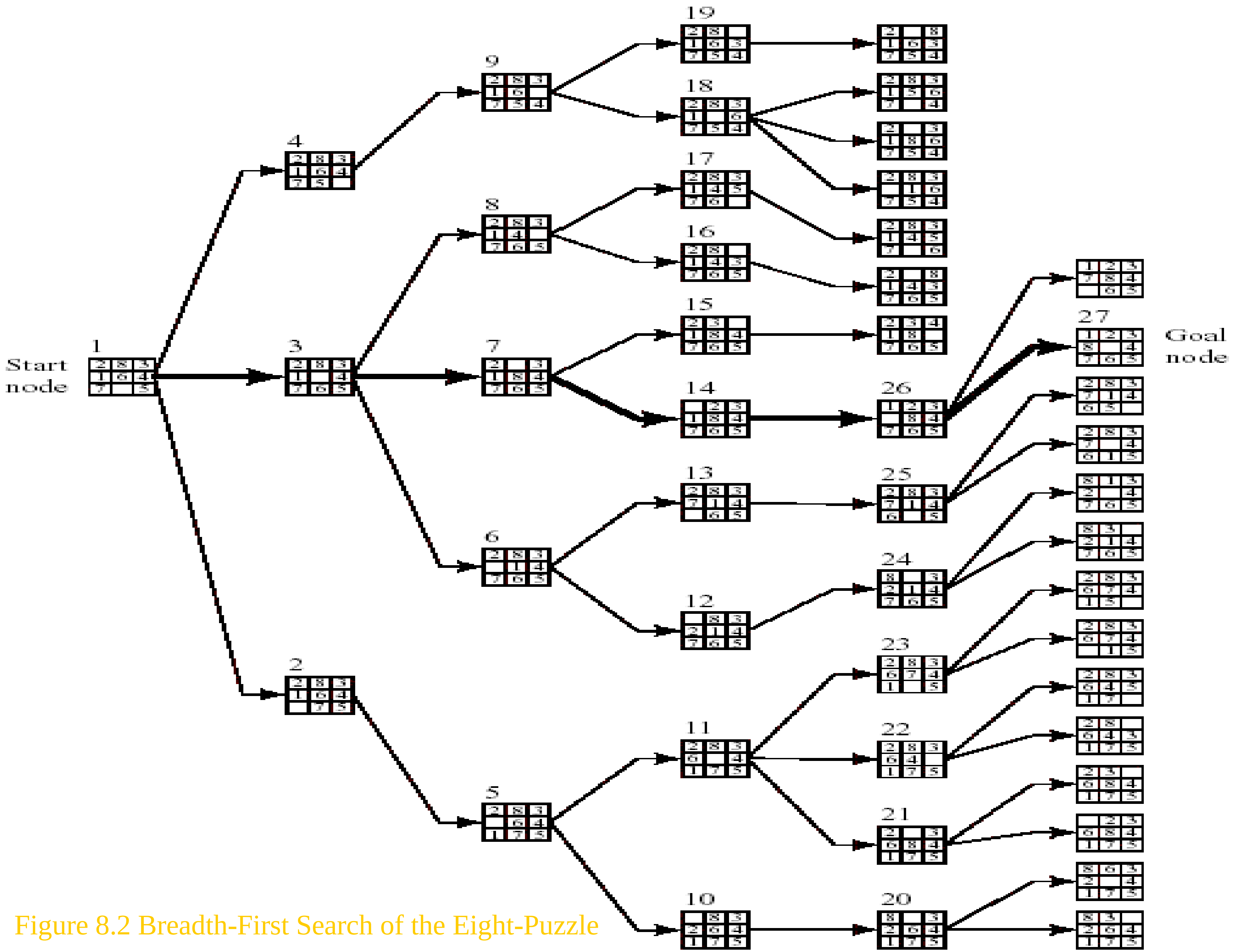


Figure 8.2 Breadth-First Search of the Eight-Puzzle

Advantages of BFS:

1. BFS is a systematic search strategy- all nodes at level n are considered before going to n+1 th level.
2. If any solution exists then BFS guarentees to find it.
3. If there are many solutions , BFS will always find the shortest path solution.

Disadvantages of BFS:

1. All nodes are to be generated at any level. So even unwanted nodes are to be remembered. Memory wastage.
2. Time and space complexity is exponential type- Hurdle.

Depth First Search (DFS)

Algorithm:

1. Create a variable STACK and set it to initial state.
2. Loop: Until a Goal State is found or STACK is empty:

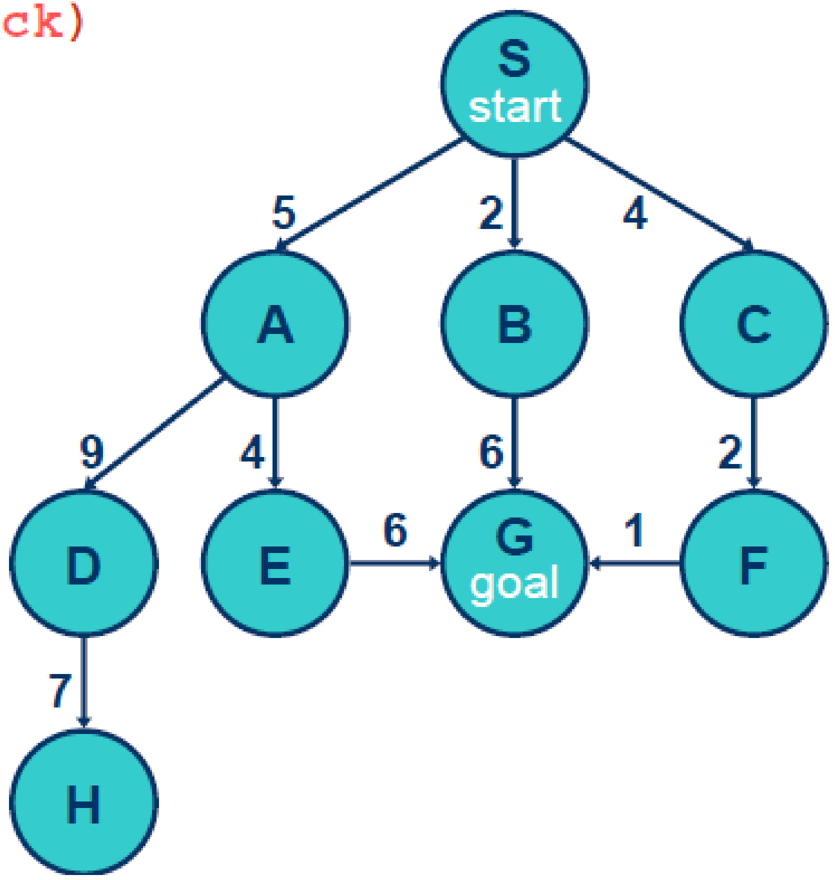
```
if nodes is empty then return failure
   node := Remove-Front (STACK)
  if Goal-Test[problem] applied to State(STACK)
succeeds
      then return node
   new-nodes := Expand (STACK,perators[problem]))
   nodes := Insert-At-Front-of-Stack(new-nodes)
end
```

Depth-first

`generalSearch(problem, stack)`

# of nodes tested: 0, expanded: 0

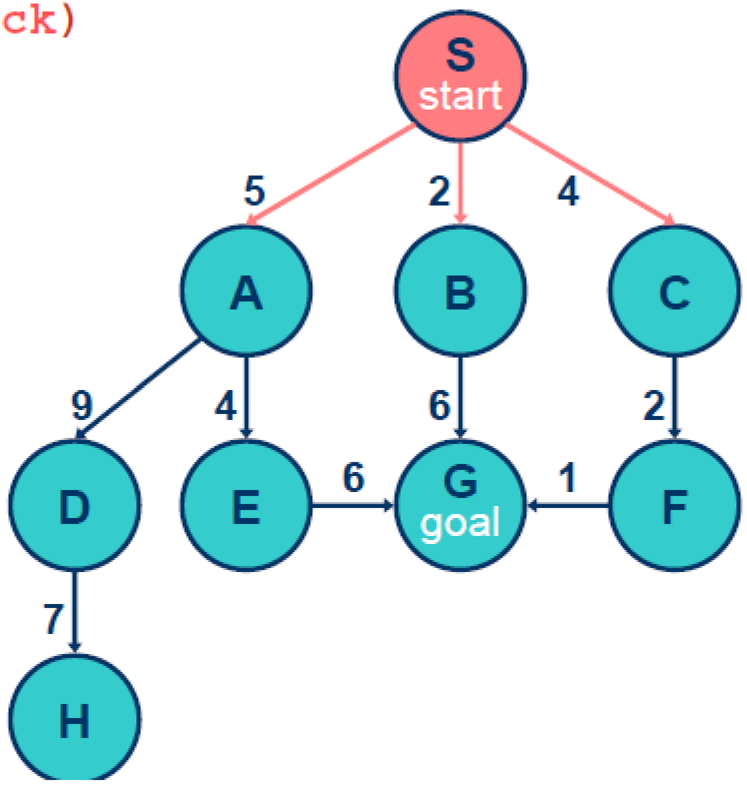
expnd. node	nodes list
	{S}



`generalSearch(problem, stack)`

# of nodes tested: 1, expanded: 1

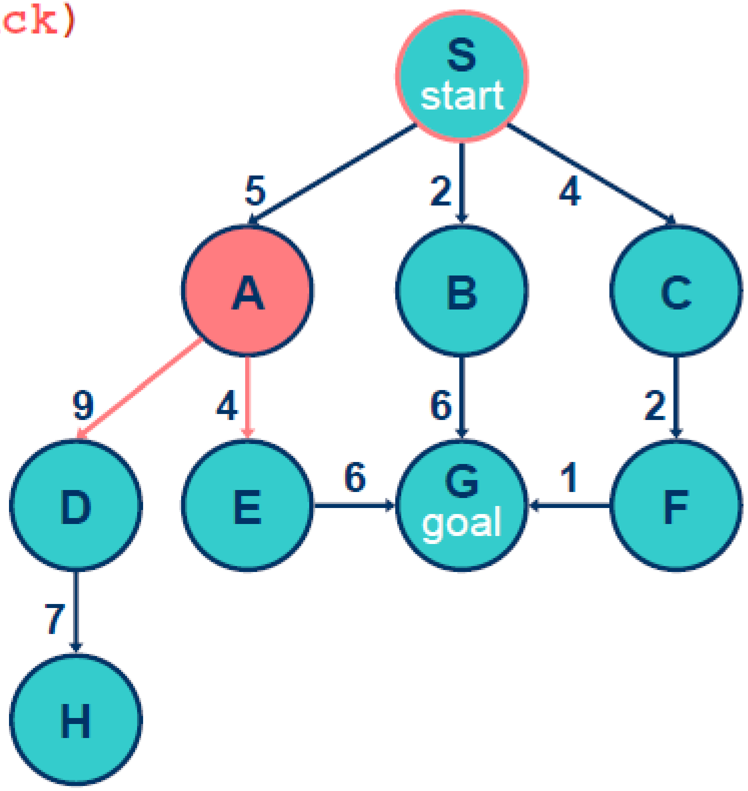
expnd. node	nodes list
	{S}
S not goal	{A,B,C}



`generalSearch(problem, stack)`

# of nodes tested: 2, expanded: 2

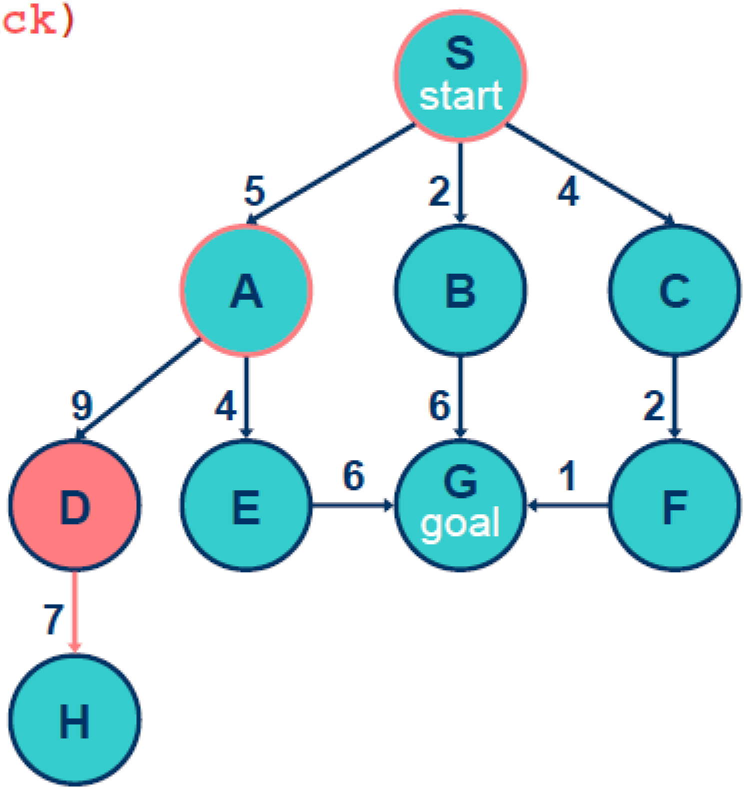
expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{D,E,B,C}



`generalSearch(problem, stack)`

# of nodes tested: 3, expanded: 3

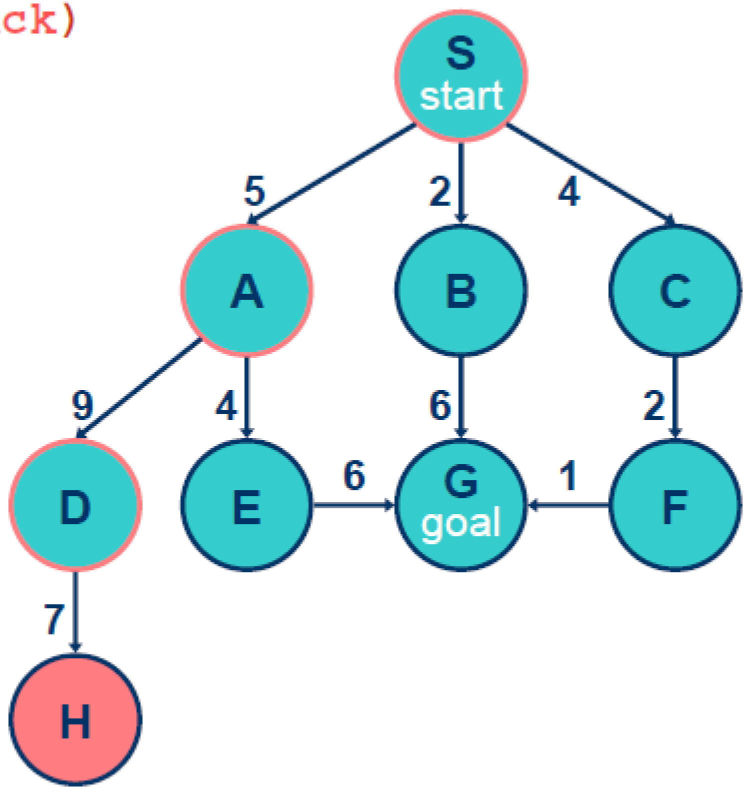
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D not goal	{H,E,B,C}



```
generalSearch(problem, stack)
```

# of nodes tested: 4, expanded: 4

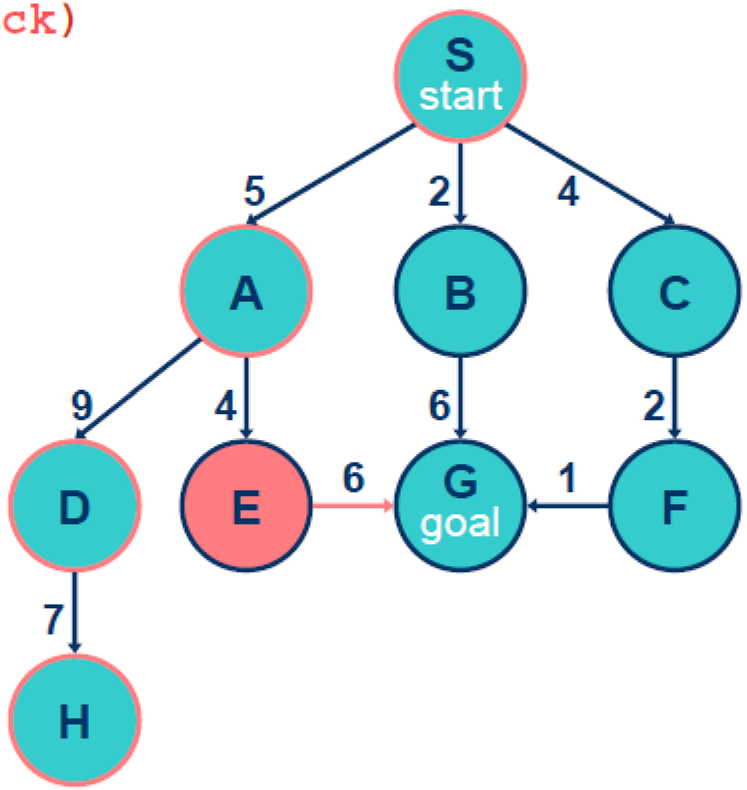
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H not goal	{E,B,C}



generalSearch(problem, stack)

# of nodes tested: 5, expanded: 5

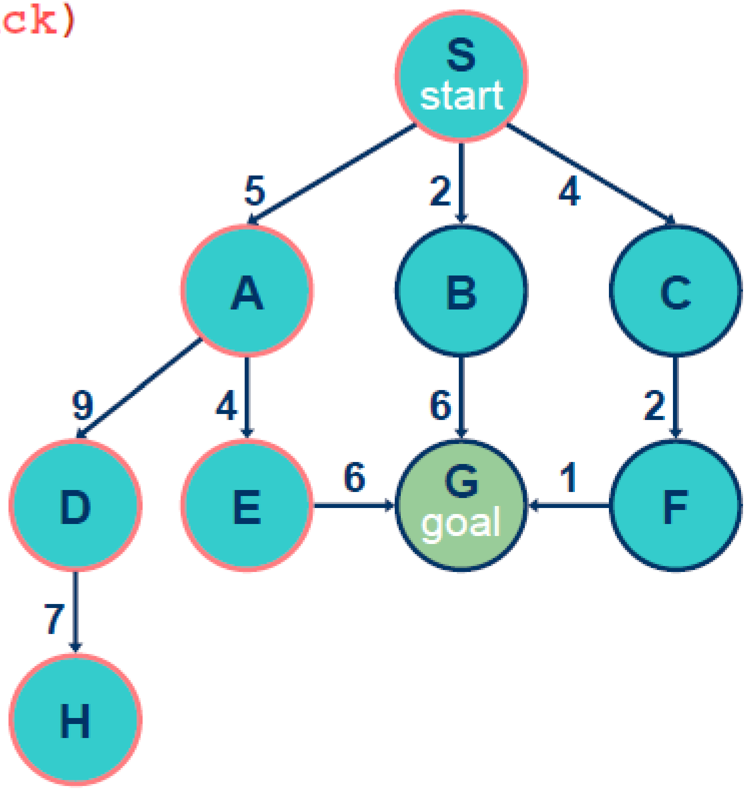
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E not goal	{G,B,C}



```
generalSearch(problem, stack)
```

# of nodes tested: 6, expanded: 5

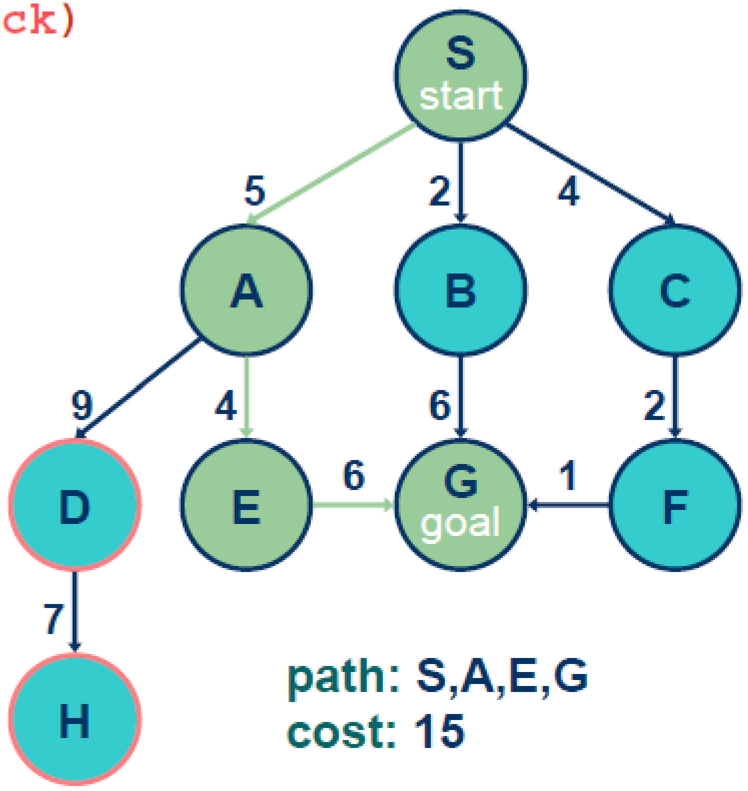
expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G goal	{B,C} no expand



`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G	{B,C}



path: S,A,E,G
cost: 15

4. Depth-First or Backtracking Search (Cont'd)

- 8-puzzle example
  - Depth bound: 5
  - Operator order: left → up → right → down

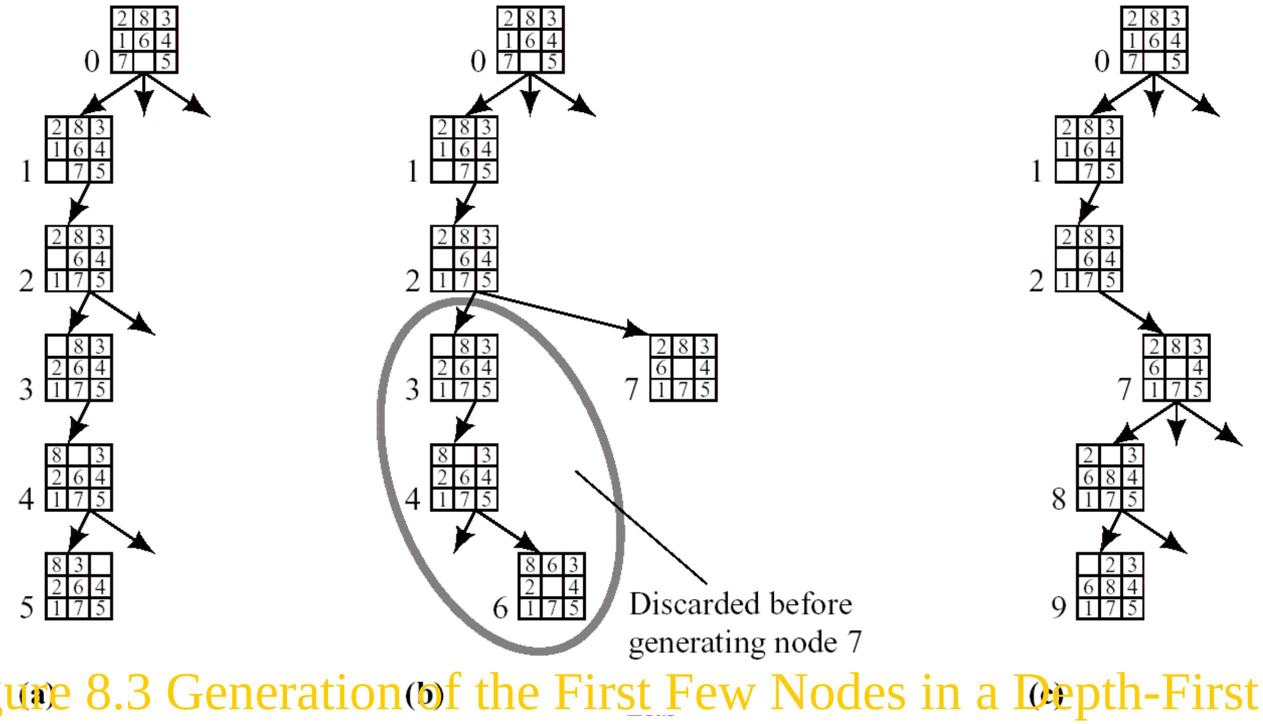
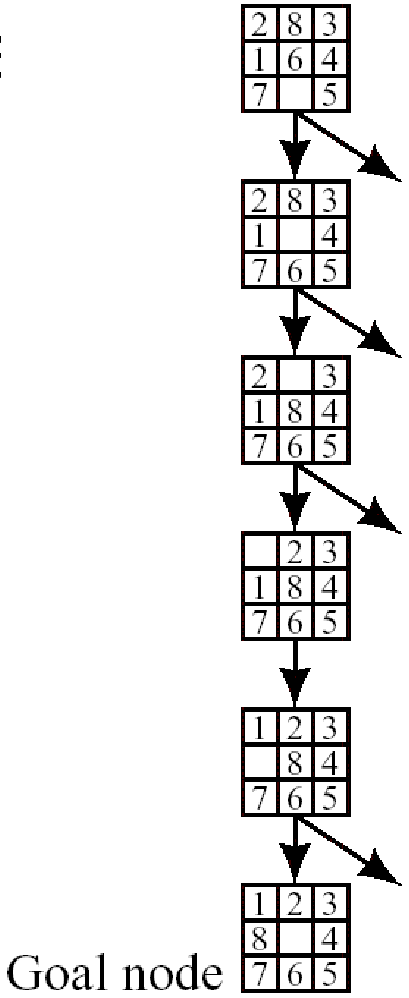


Figure 8.3 Generation of the First Few Nodes in a Depth-First Search

4. Depth-First or Backtracking Search (Cont'd)

– The graph when the goal is reached in depth-first se



Advantages of DFS:

1. Memory requirements in DFS are less compared to BFS as only nodes on the current path are stored.
2. DFS may find a solution without examining much of the search space of all.

Disadvantages of BFS:

1. This search can go on deeper and deeper into the search space and thus can get lost. This is referred to as **blind alley**.!

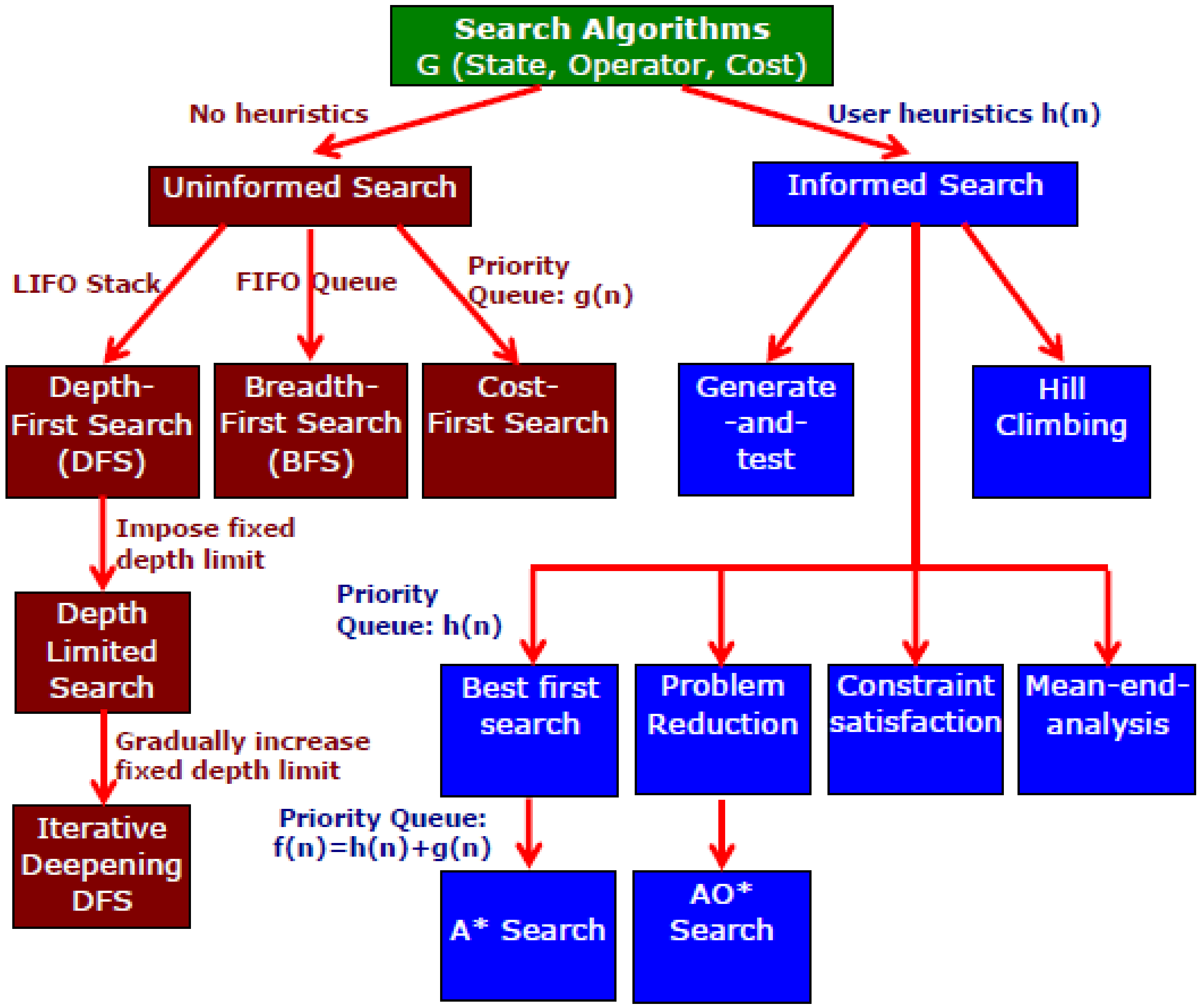


Fig. Different Search Algorithms

Informed Search

- ❑ Also Called heuristic or intelligent search,
- ❑ Uses information about the problem to guide the search,
- ❑ Usually guesses the distance to a goal state and therefore efficient,
- ❑ but the search may not be always possible.
- ❑ The search so guided are called heuristic search and the methods used are called heuristics.

Advantage of Informed Search

- A search strategy which searches the most promising branches of the state-space first can:
  - find a solution more quickly,
  - find solutions even when there is limited time available,
  - Often find a *better* solution, since more profitable parts of the state-space can be examined, while ignoring the unprofitable parts.
- A search strategy which is better than another at identifying the most promising branches of a search-space is said to be more *informed*.

Heuristic Search Compared with other Search

Brute force / Blind search	**Heuristic search**
◇ Only have knowledge about already explored nodes	◇ Estimates “distance” to goal state
◇ No knowledge about how far a node is from goal state	◇ Guides search process toward goal state
	◇ Prefer states (nodes) that lead close to and not away from goal state

Heuristics

Algorithm

- First, generate a possible solution which can either be a point in the problem space Or a path from the initial state.
- Then, test to sea if this possible solution is a real solution by comparing the state
- Reached with the set of goal states.
- If it is real solution, return, else repeat from the first again.

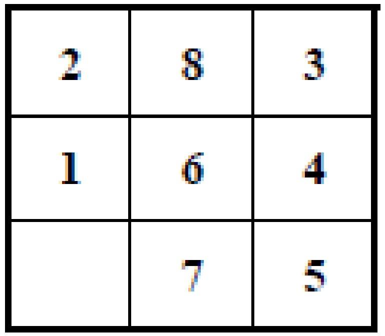
Heuristics Search Technique

Example of Heuristic Function

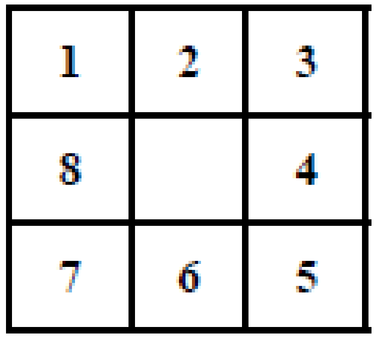
- ❑ A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal. It is denoted by h(n).
- ❑ h(n) = estimated cost of the cheapest path from node n to a goal node

Example 1: We want a path from Kolkata to Guwahati
Heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati
h(Kolkata) = euclideanDistance(Kolkata, Guwahati)

Example 2: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.



Initial State

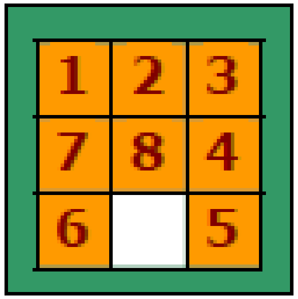


Goal state

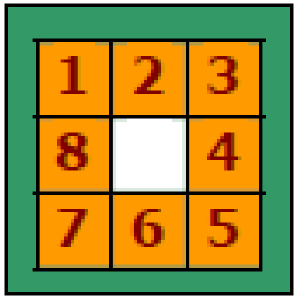
The first picture shows the current state *n*, and the second picture the goal state.*h(n) = 5* because the tiles 2, 8, 1, 6 and 7 are out of place.

Manhattan Distance Heuristic: Another heuristic for 8-puzzle is the Manhattan distance heuristic. This heuristic sums the distance that the tiles are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.
For the above example, using the Manhattan distance heuristic,
$h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6$

Initial: any configuration



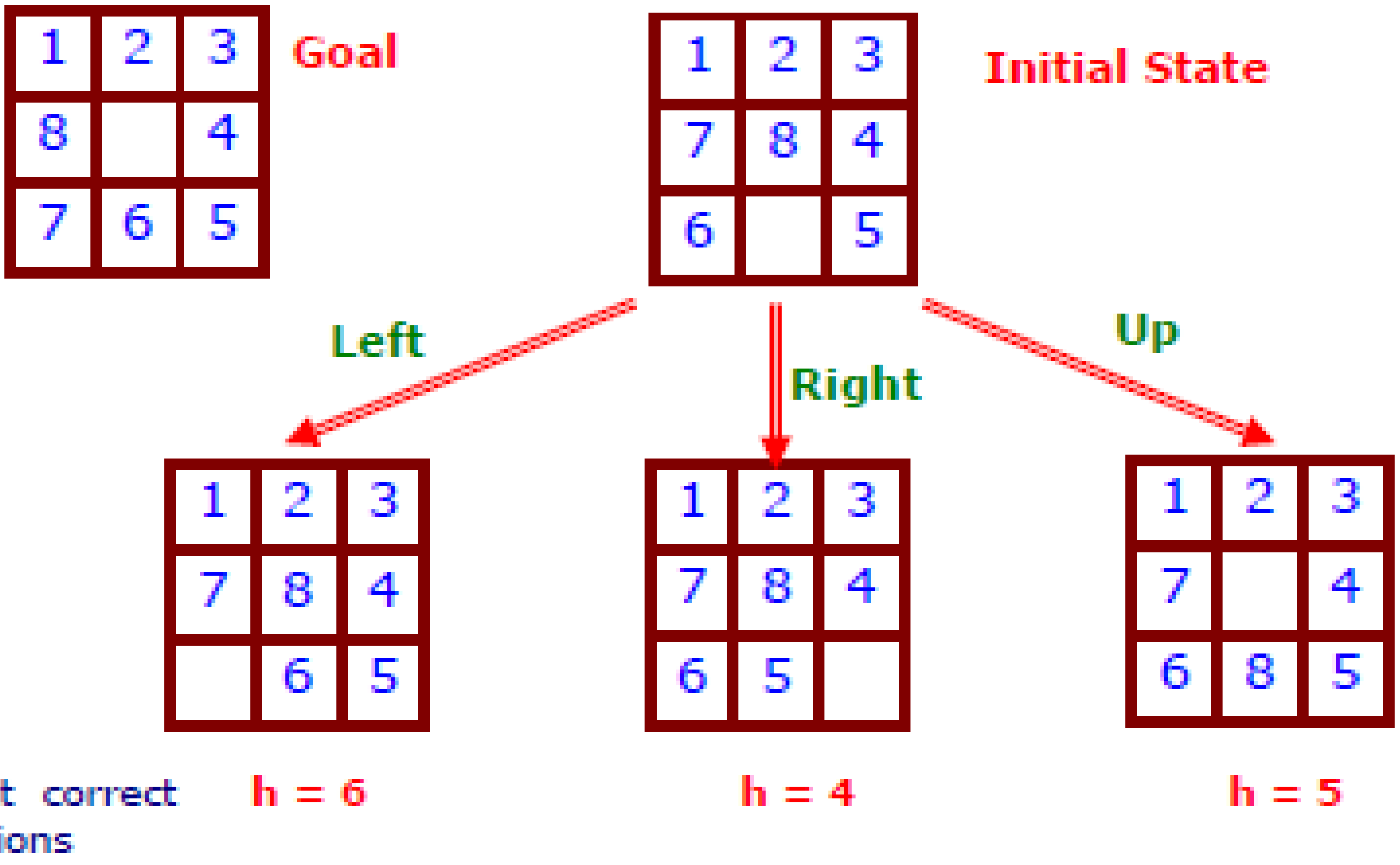
Goal: tiles in a specific order



- Solution: optimal sequence of operators
- Action: "blank moves"
  - Condition: the move is within the board
  - Directions: Left, Right, Up, Dn
- Problem
  - which 8-puzzle move is best?
  - what heuristic(s) can decide?
  - which move is "best" (worth considering first) ?

Action

Three possible moves –left,up,right



Find Which move is best ?

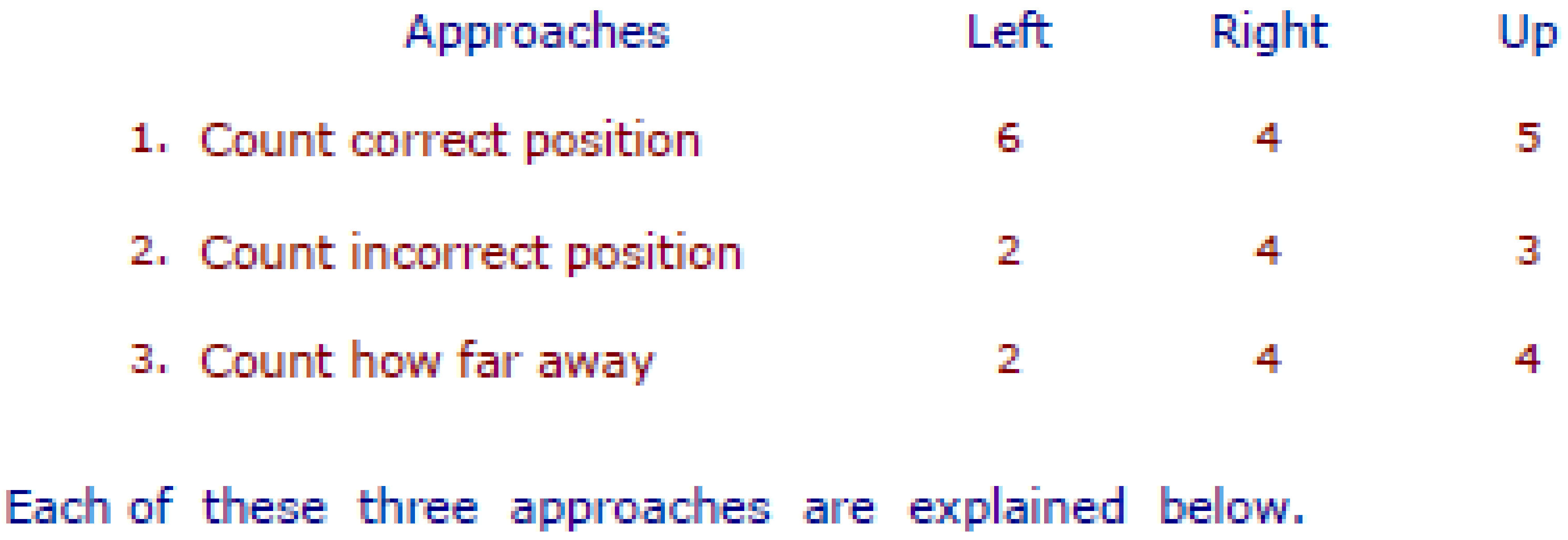
Apply Heuristic

Three different Approaches

-count correct position of each tile, compare to goal state.

-count incorrect position of each tile, compare to goal state.

ct



Approaches	Left	Right	Up
1. Count correct position	6	4	5
2. Count incorrect position	2	4	3
3. Count how far away	2	4	4

Each of these three approaches are explained below.

Three different approaches

- 1st approach :

  Count correct position of each tile, compare to goal state.

  - Higher the number the better it is.
  - Easy to compute (fast and takes little memory).
  - Probably the simplest possible heuristic.

- 2nd approach

  Count incorrect position of each tile, compare to goal state

  - Lower the number the better it is.
  - The "best" move is where lowest number returned by heuristic.

- 3rd approach

  Count how far away each tile is from it's correct position

  - Count how far away (how many tile movements) each tile is from it's correct position.
  - Sum up these count over all the tiles.
  - The "best" move is where lowest number returned by heuristic.

Best First Search

Idea: use an evaluation function *f(n)* for each node
f(n) provides an estimate for the total cost.
→ Expand the node n with smallest f(n).

Implementation:
Order the nodes in fringe increasing order of cost.

- ❑ The algorithm maintains a priority queue of nodes to be explored.
- ❑ A cost function f(n) is applied to each node.
- ❑ The nodes are put in OPEN in the order of their f values.
- ❑ Nodes with smaller f(n) values are expanded earlier. The generic best first search algorithm is outlined below.

Algorithm:

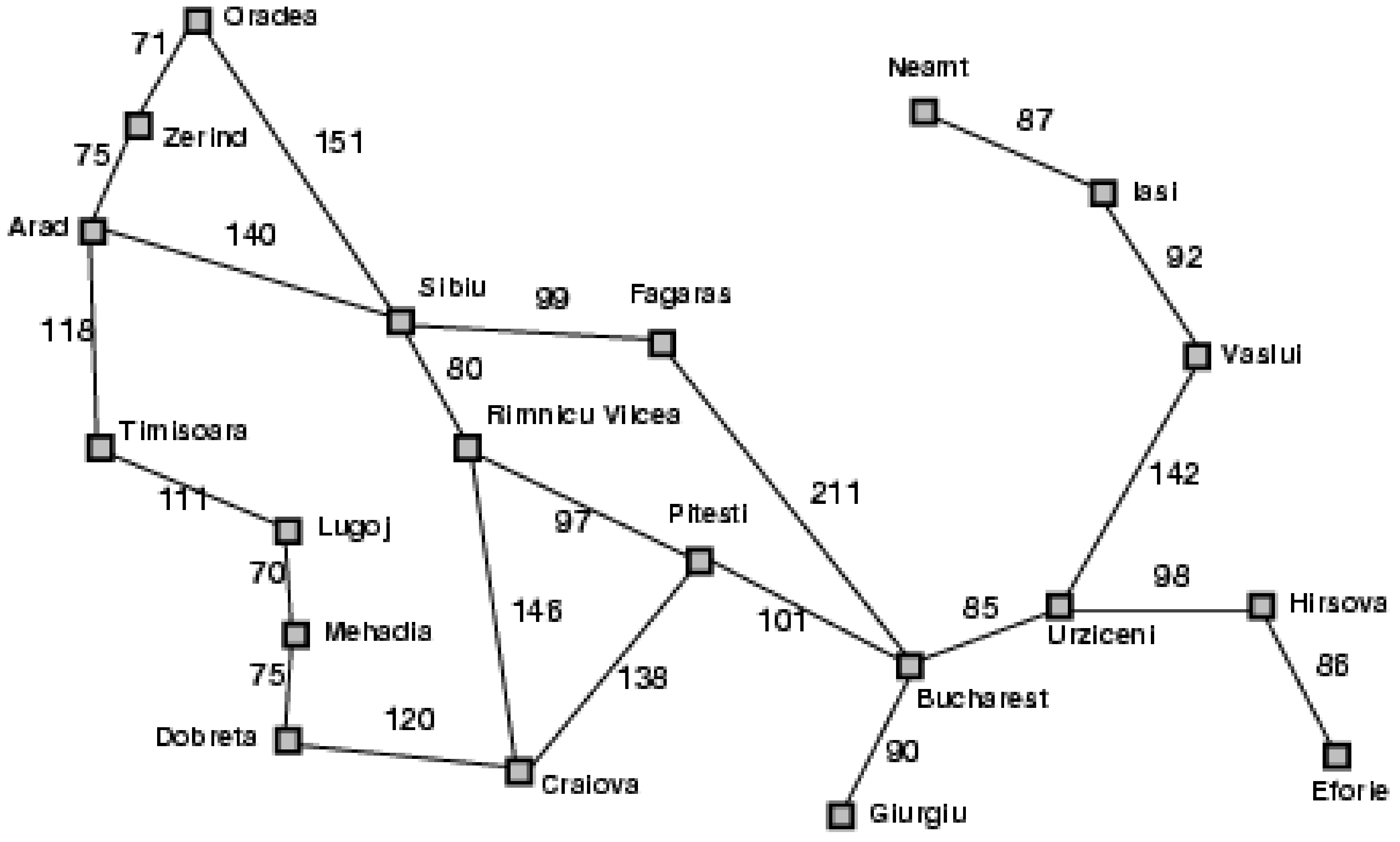
Best First Search

```
Let fringe be a priority queue containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
                        if Node is a goal
                          then return the path from initial state to Node
            else generate all successors of Node, and
              put the newly generated nodes into fringe
              according to their f values
End Loop
```

Greedy Best First Search

- In greedy search, the idea is to expand the node with the smallest estimated cost to reach the goal.
- We use a heuristic function $f(n) = h(n)$
- $h(n)$ estimates the distance remaining to a goal.
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

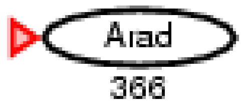
Romania with step costs in km



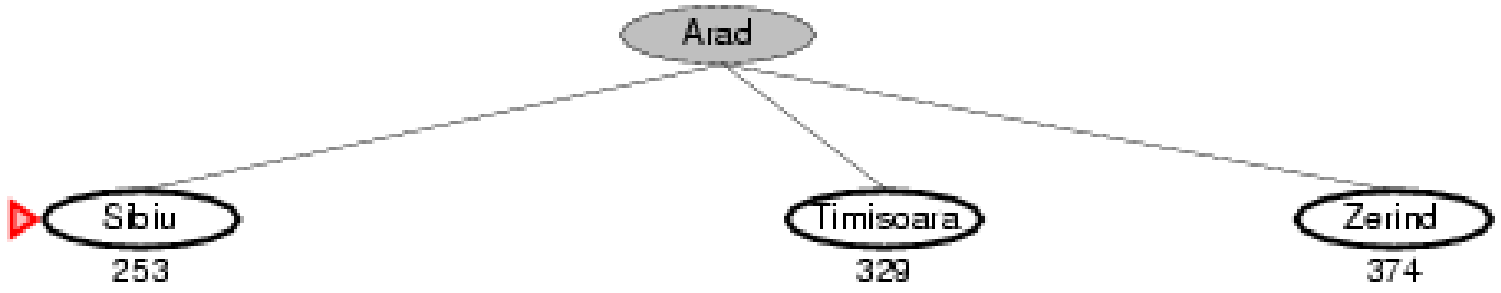
Straight−line distance
to Bucharest

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

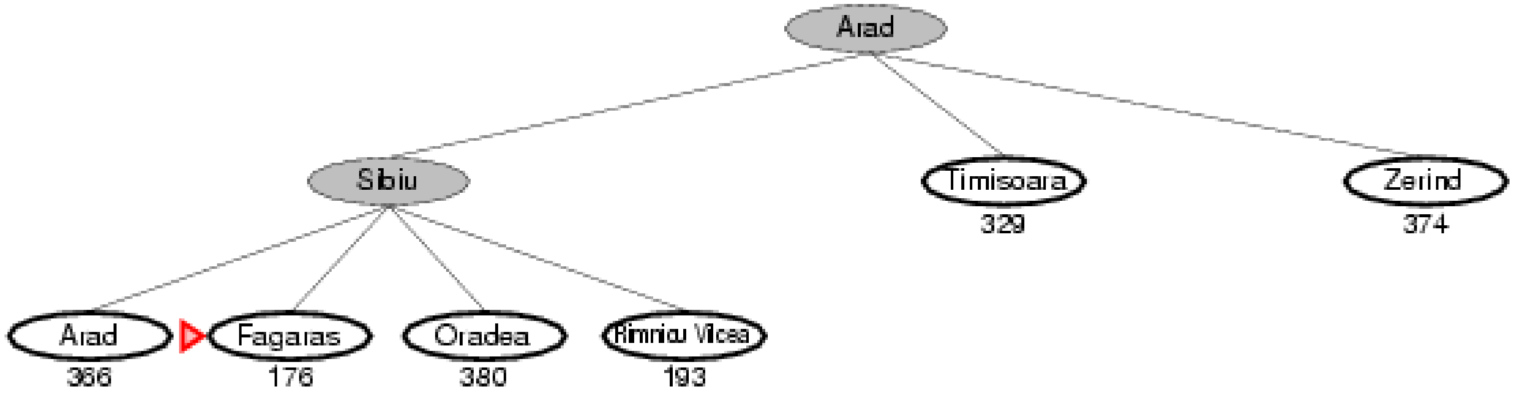
Greedy best-first search example



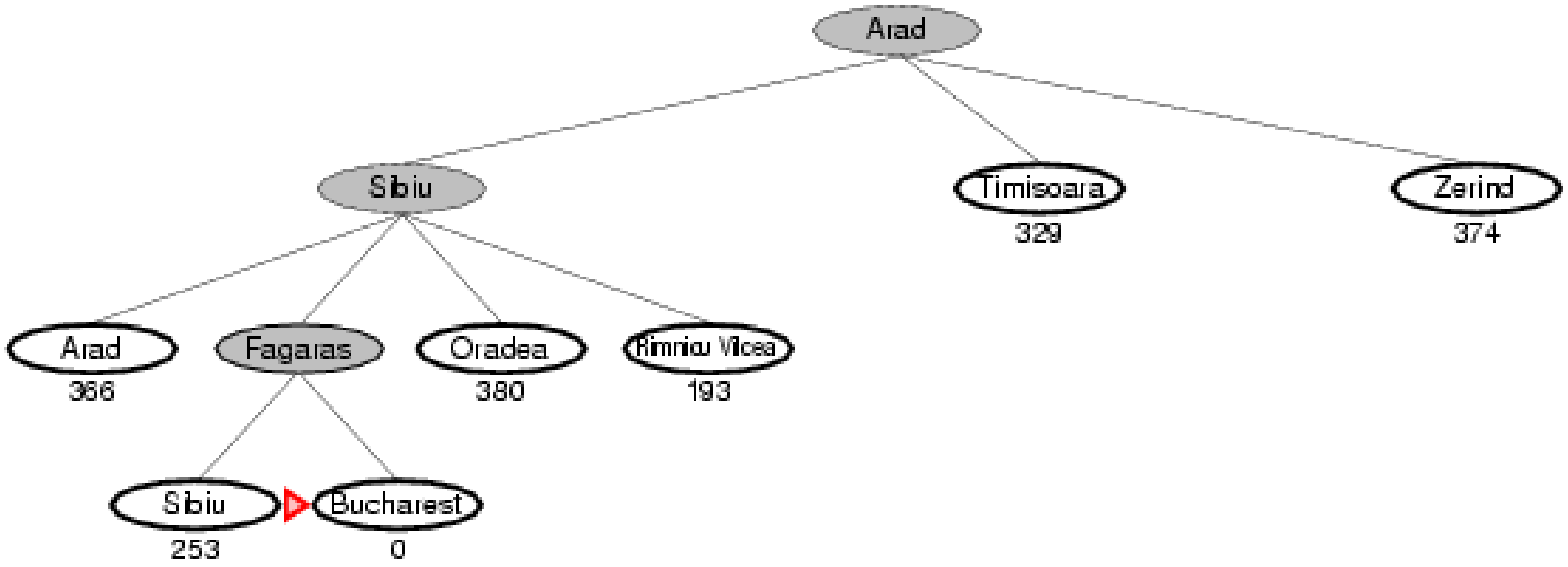
Greedy best-first search example



Greedy best-first search example



Greedy best-first search example



Properties of greedy best-first search

Complete? No – can get stuck in loops, e.g., Iasi Neamt Iasi Neamt

-
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
-
- Space? $O(b^m)$ -- keeps all nodes in memory
-
- Optimal? No

Figure 2 is an example of a route finding problem. S is the starting state, G is the goal state.

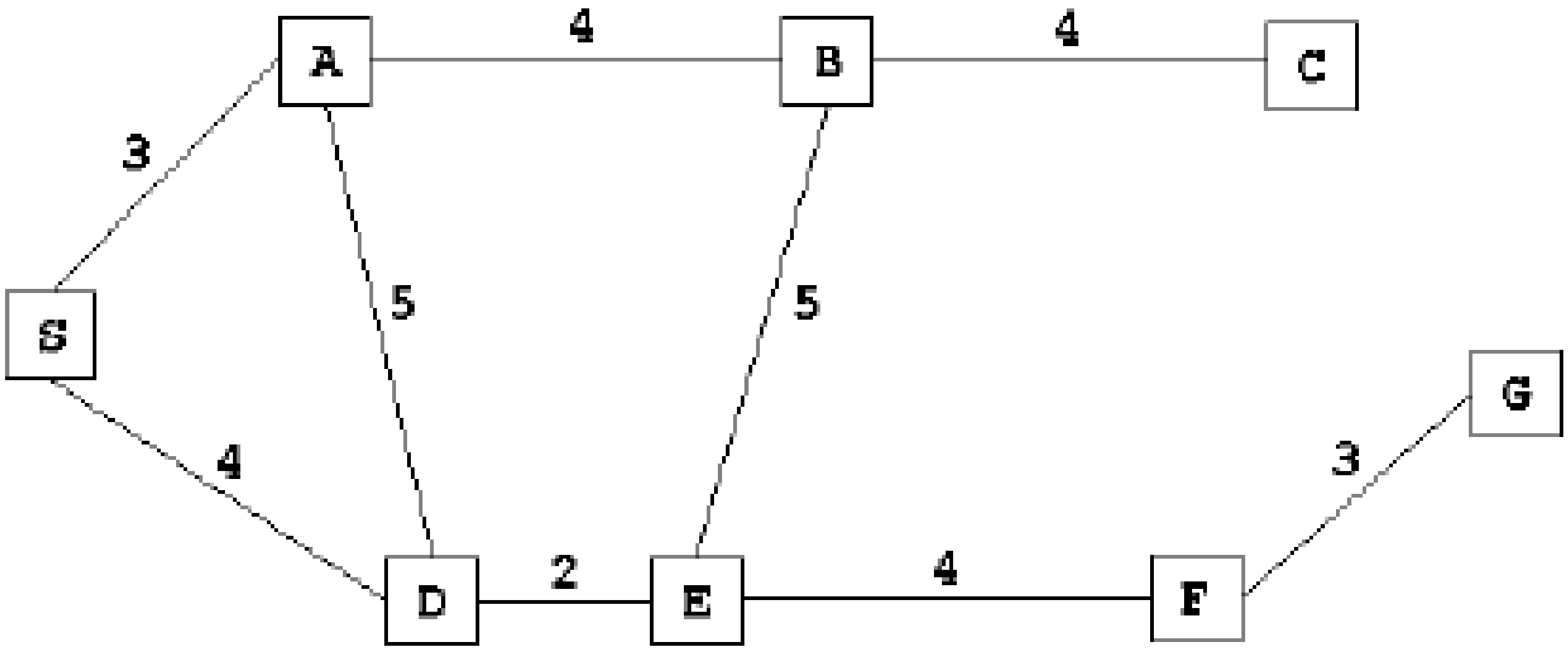


Figure 2

Let us run the greedy search algorithm for the graph given in Figure 2. The straight line distance heuristic estimates for the nodes are shown in Figure

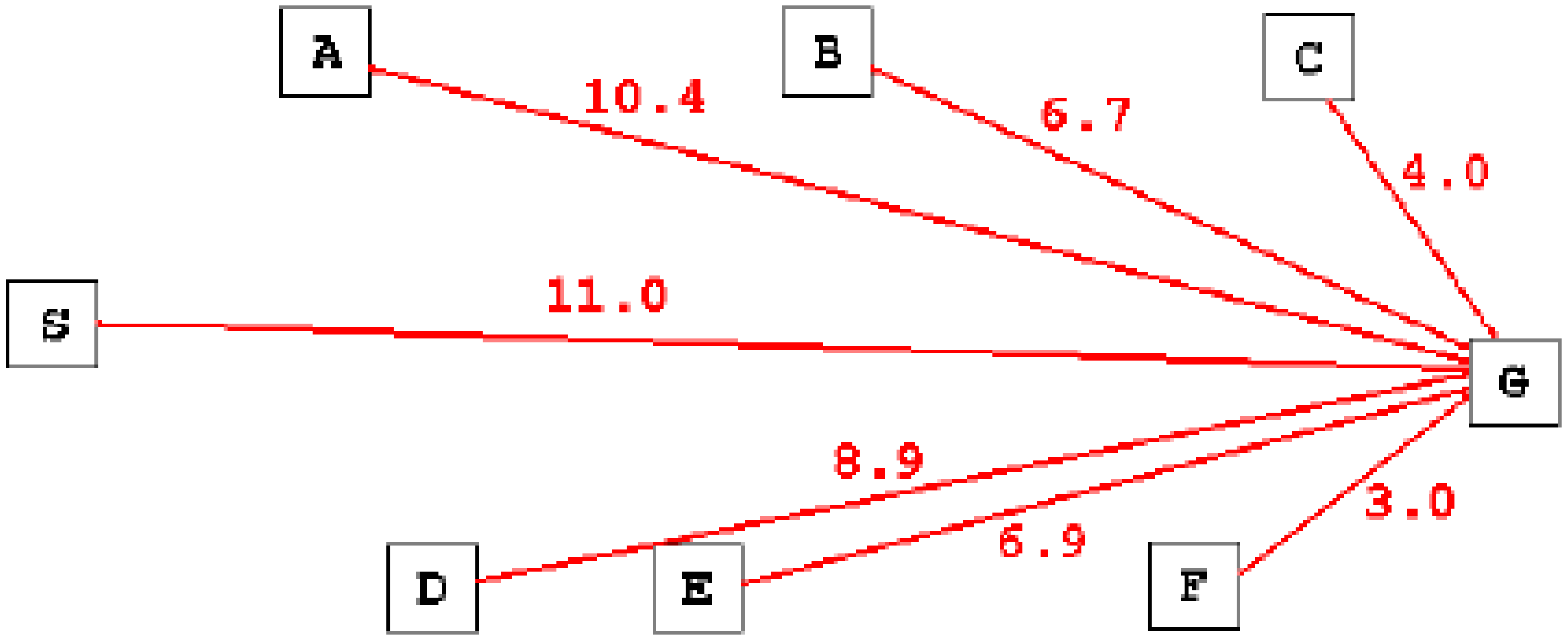
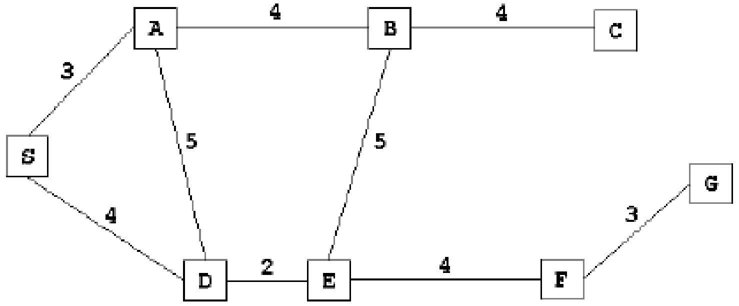
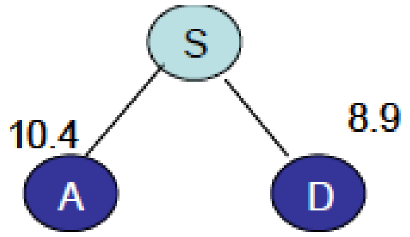


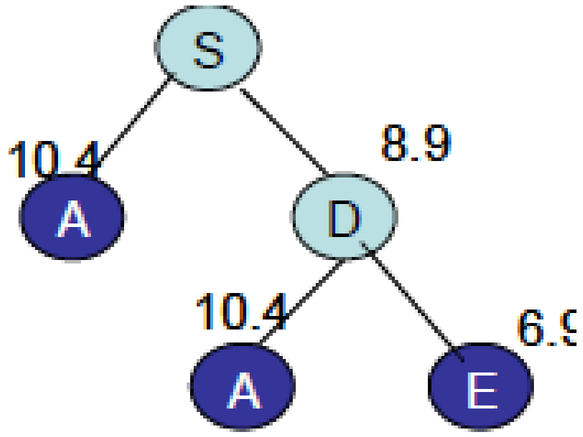
Figure 3

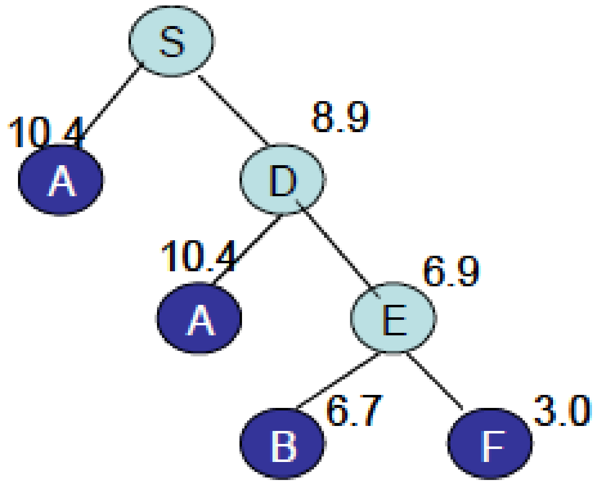


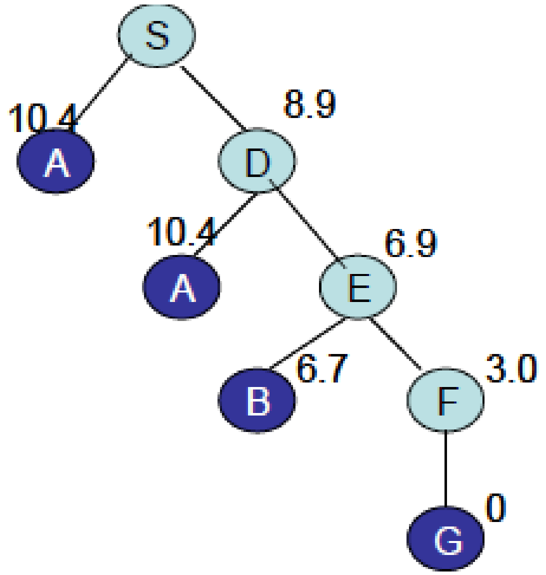
Step 1: S is expanded. Its children are A and D.



Step 2: D has smaller cost and is expanded next.







A* Search

- ❑ Idea: avoid expanding paths that are already expensive
- ❑ A* is a best first search algorithm with

$$f(n)=g(n)+h(n).$$

Where

g(n)=sum of edge costs from start to n (distance current node from **start**)

- ❑ h(n)=estimate of lowest cost path from goal to current node.
- ❑ f(n)=actual distance so far estimated distance remaining.

Best First search has *f(n)=h(n)*

We can prove that if **h(n)** is admissible, then the search will find and optimal solution

The A* Algorithm

- **Input:**
  - **– QUEUE:** Path only containing root
- **Algorithm:**
  - – WHILE (QUEUE not empty && first path not reach goal) DO
    - Remove first path from QUEUE
    - Create paths • to all children
    - Reject paths with loops
    - Add paths and sort QUEUE (by f = cost + heuristic)
- **IF** QUEUE contains paths: P, Q

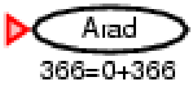
  AND P ends in node Ni && Q contains node Ni

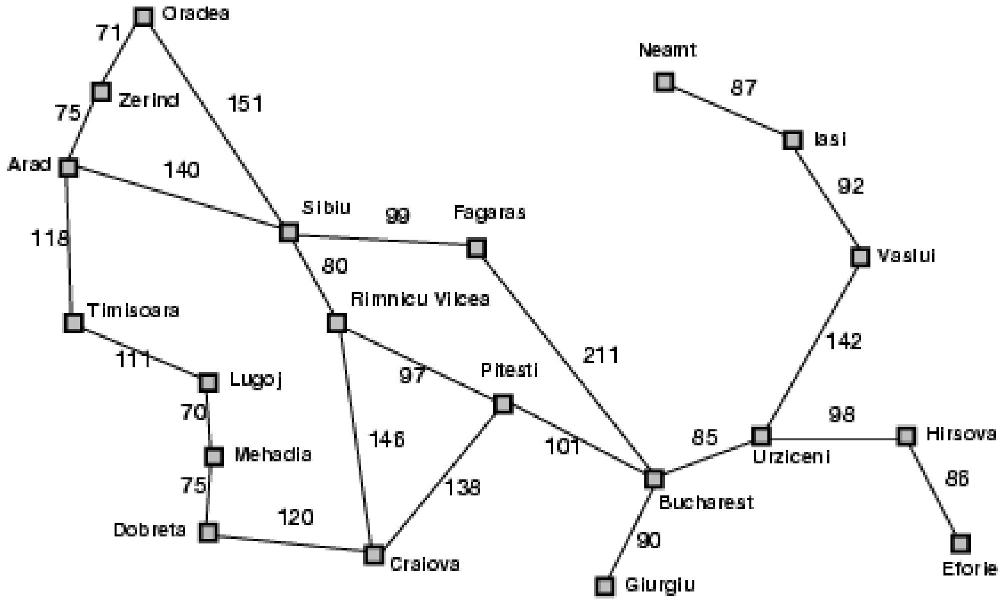
  AND cost_P ≥ cost_Q

  THEN remove P

– **IF** goal reached **THEN** success **ELSE** failure

A* search example

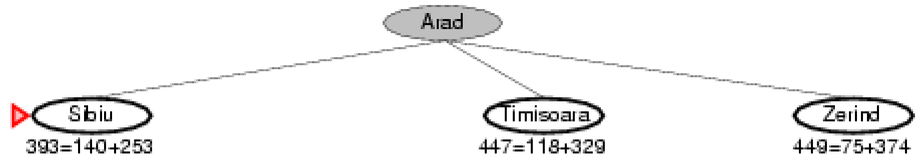


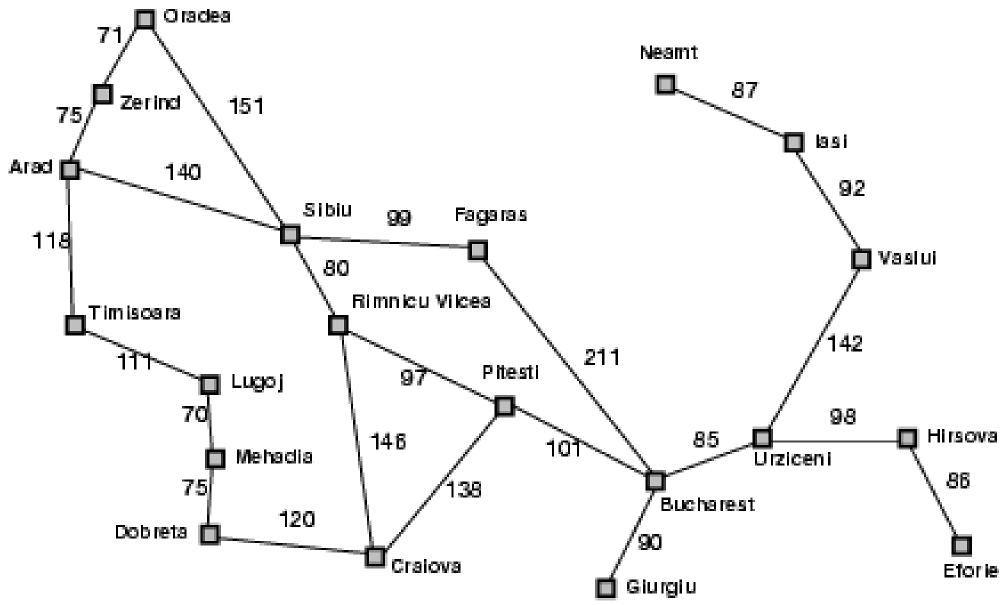


Straight–line distance to Bucharest

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

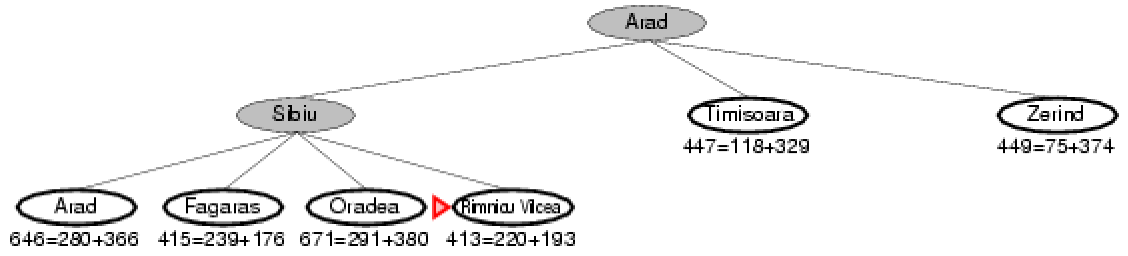


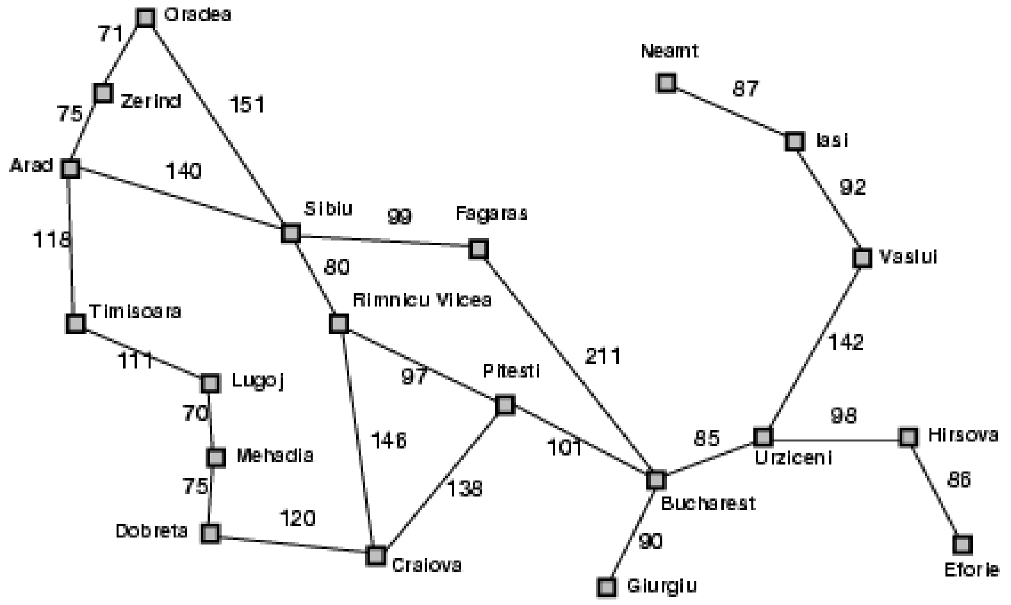


Straight−line distance to Bucharest

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

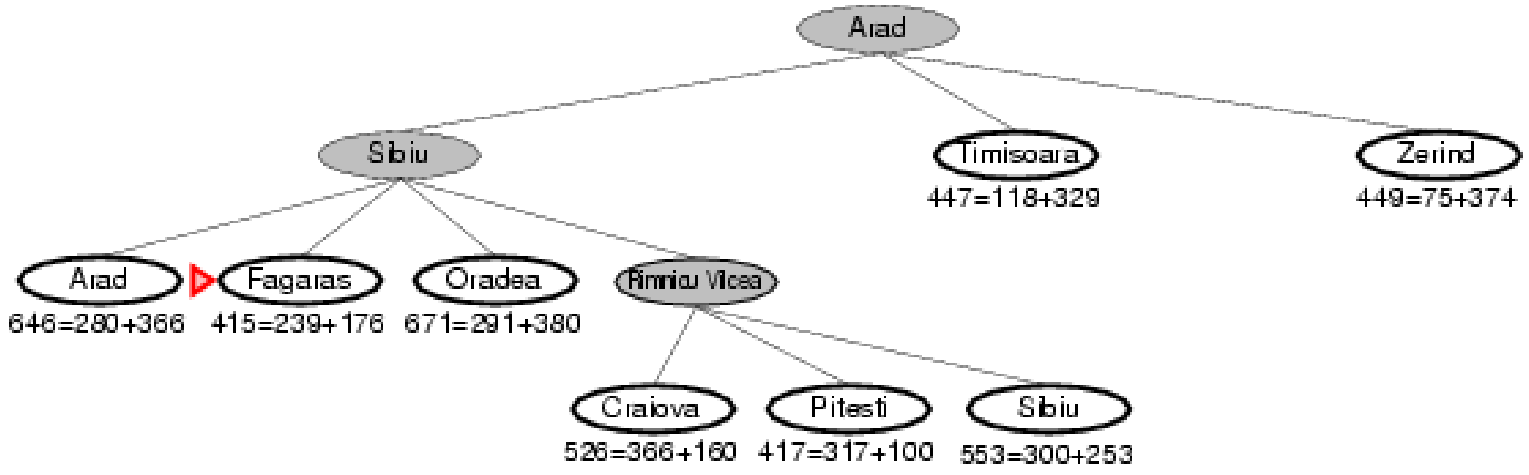


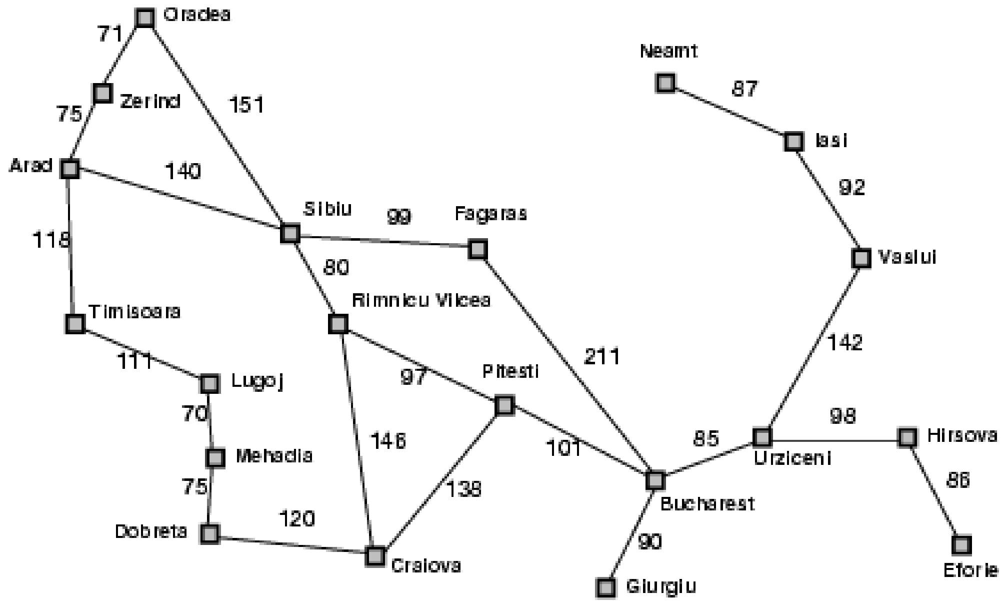


Straight–line distance to Bucharest

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

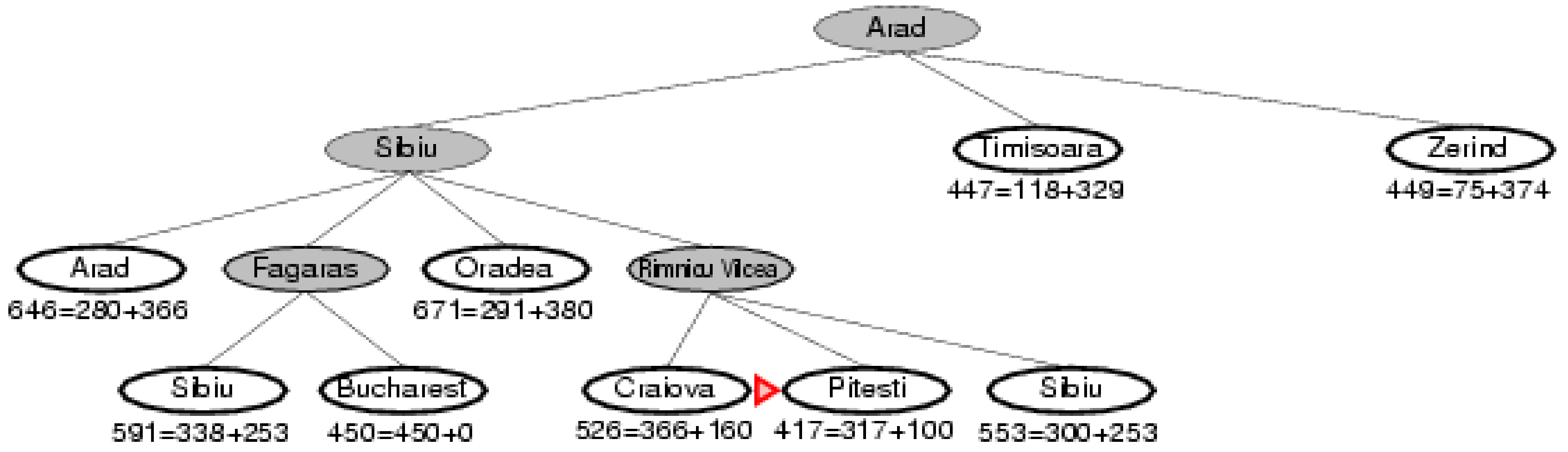


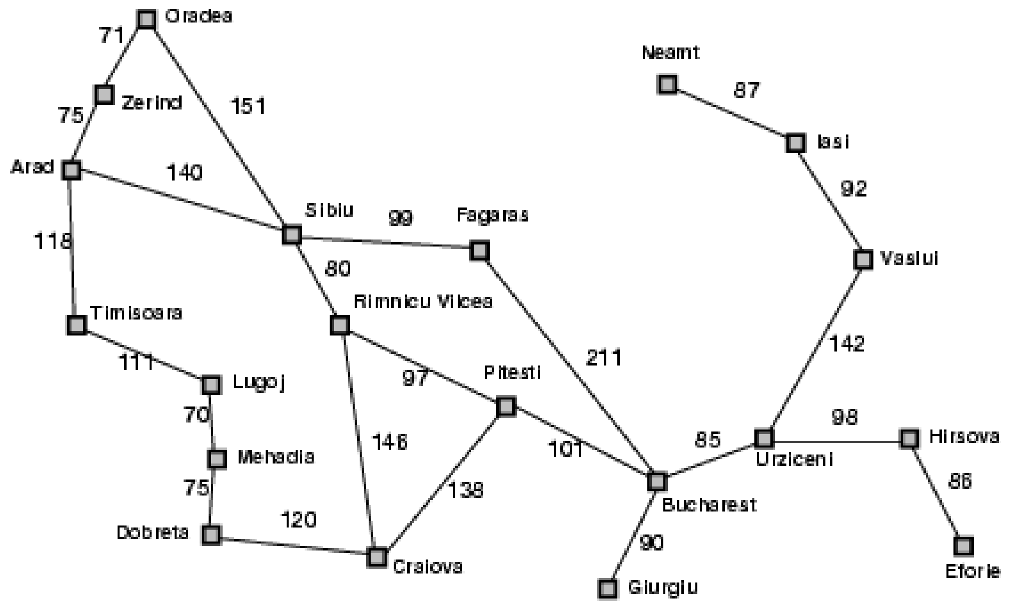


Straight−line distance
to Bucharest

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

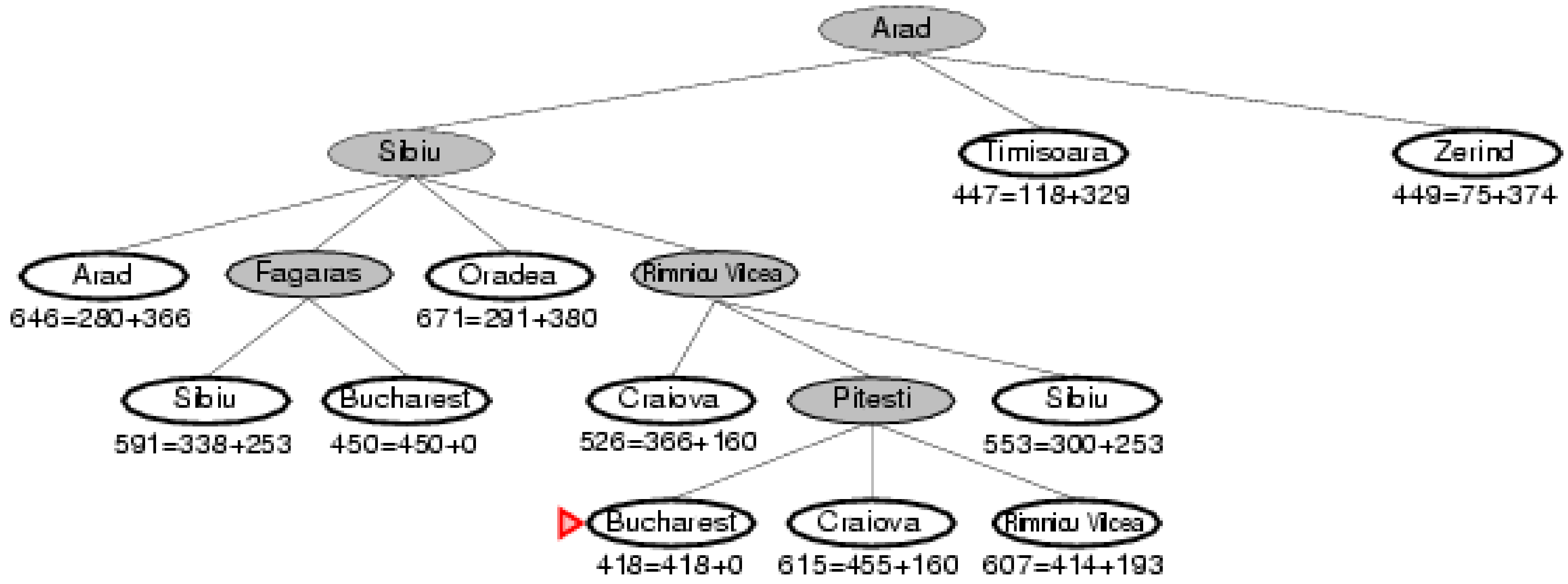


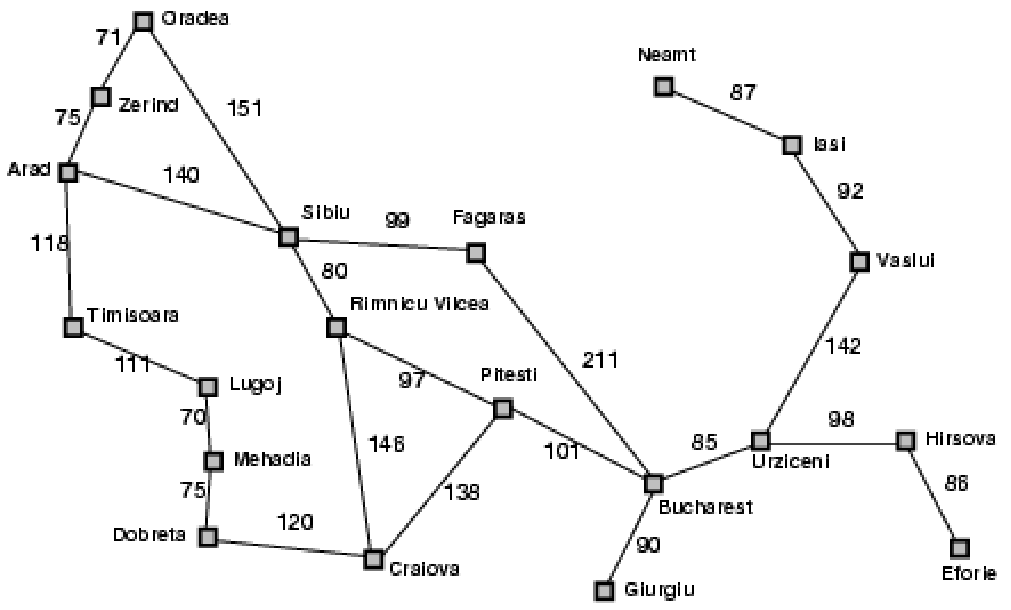


Straight−line distance to Bucharest

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

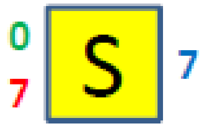




Straight–line distance to Bucharest

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

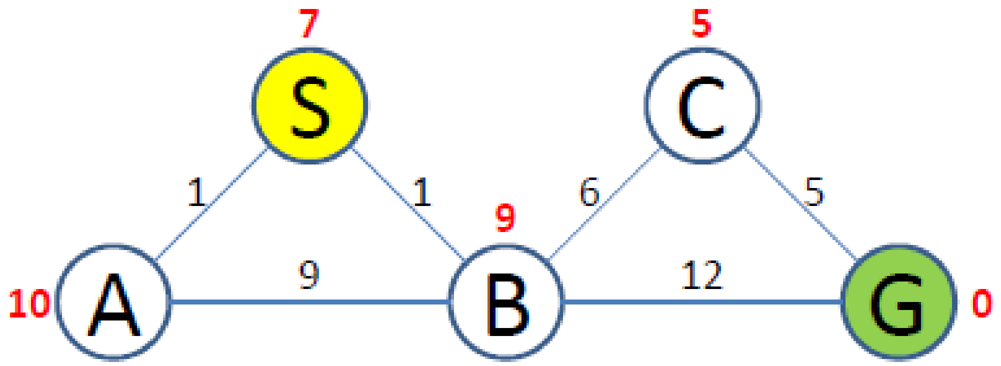
A* search example



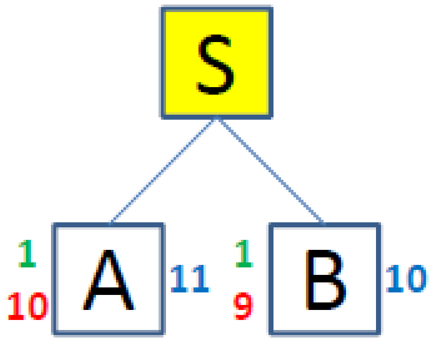
f = accumulated path cost + heuristic

QUEUE = ***path containing root***

QUEUE: <S>



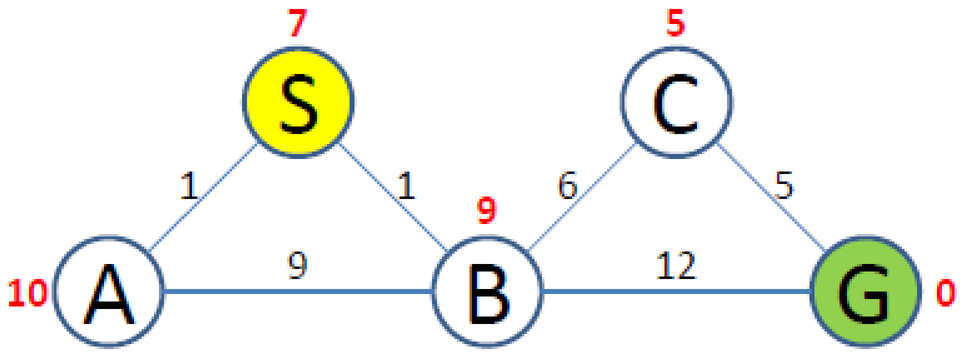
A* search example



f = accumulated path cost + heuristic

Remove ***first path***, Create ***paths to all children***, Reject ***loops*** and ***Add paths***. ***Sort*** QUEUE ***by f***

QUEUE: <SB,SA>

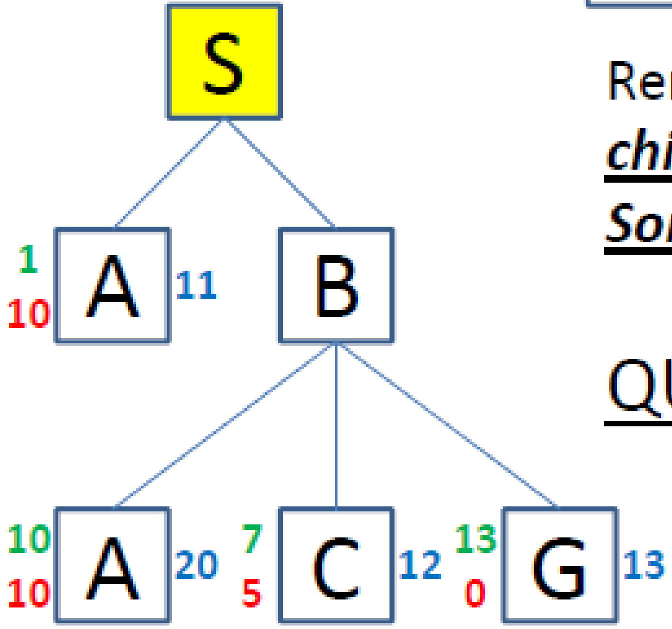


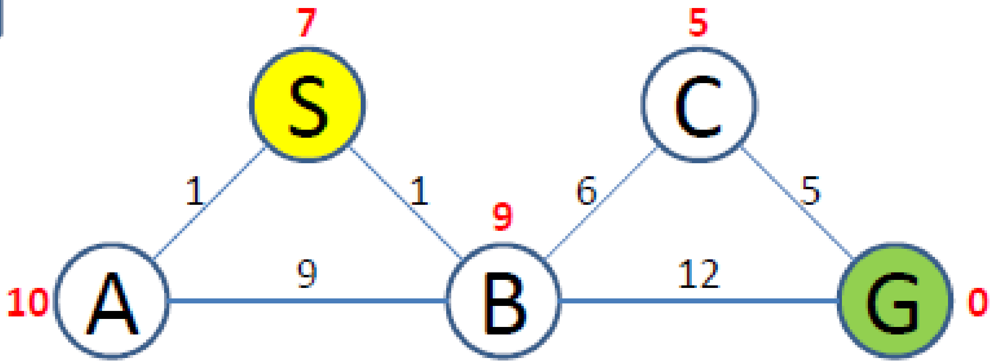
A* search example

f = accumulated path cost + heuristic

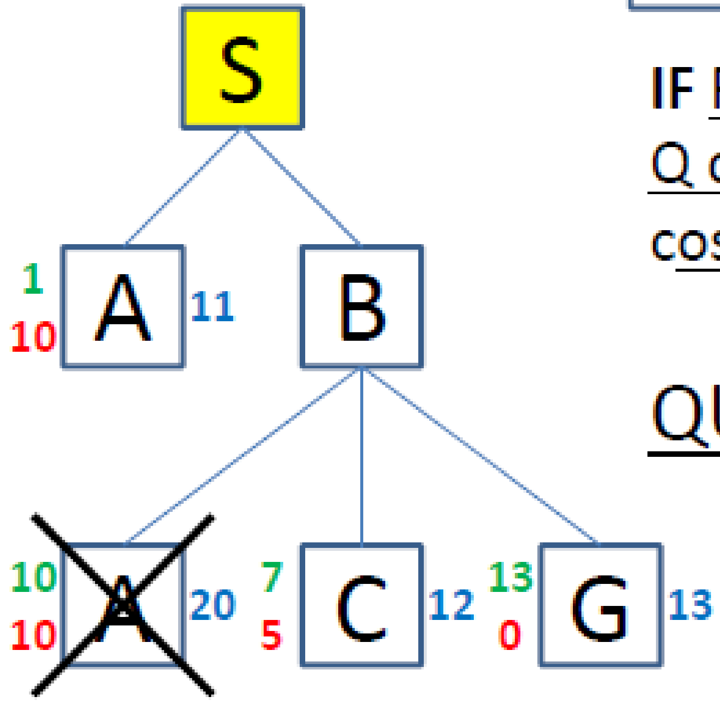
Remove ***first path***, Create ***paths to all children***, Reject ***loops*** and ***Add paths***. ***Sort*** QUEUE ***by f***

QUEUE: <SA,SBC,SBG,SBA>





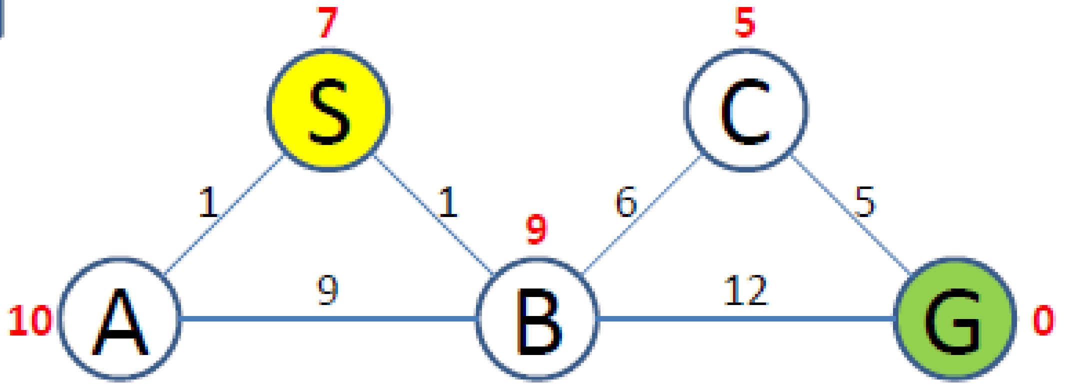
A* search example



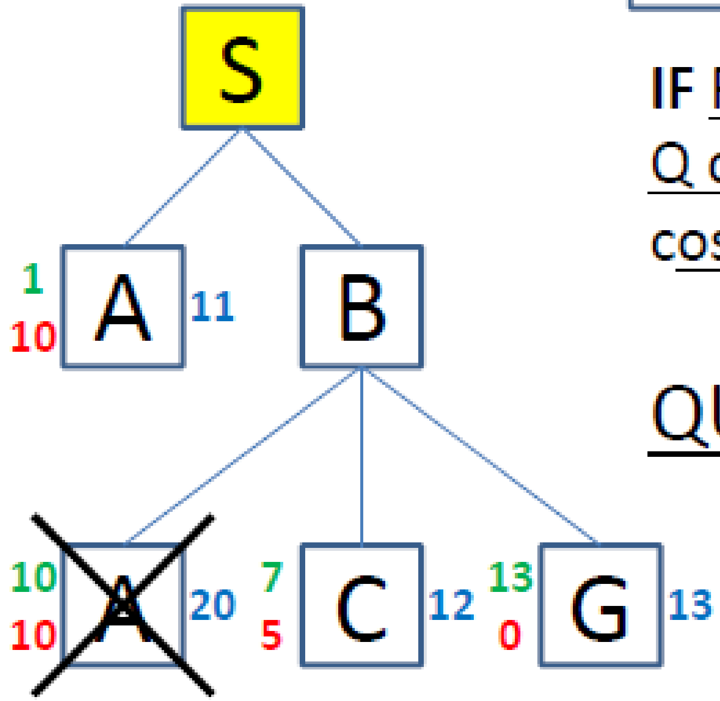
f = accumulated path cost + heuristic

IF P terminating in I with cost_P && Q containing I with cost_Q **AND** cost_P ≥ cost_Q **THEN** remove P

QUEUE: <SA,SBC,SBG,**SBA**>



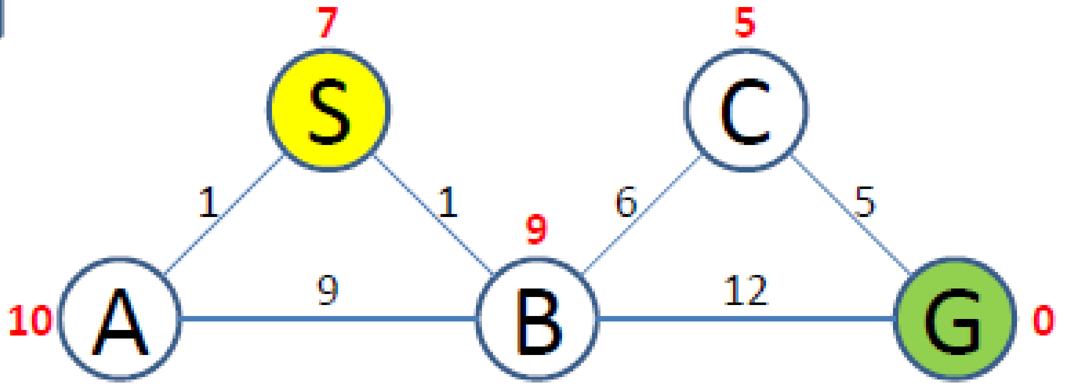
A* search example



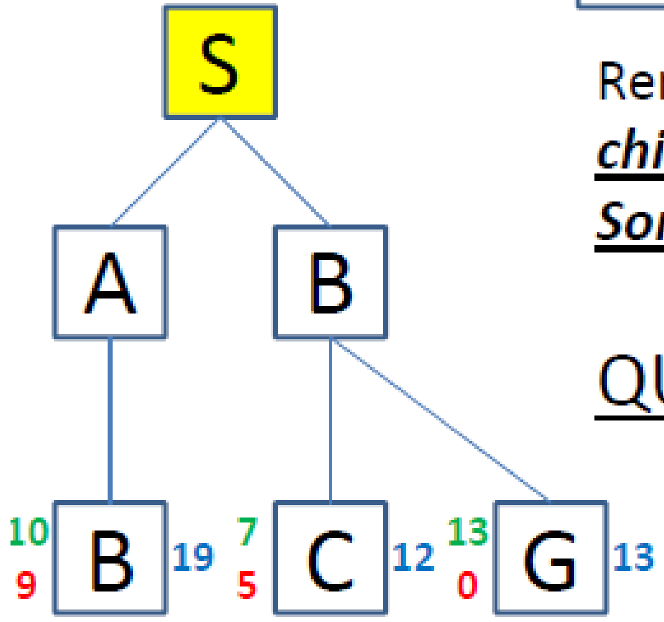
f = accumulated path cost + heuristic

IF P terminating in I with cost_P && Q containing I with cost_Q **AND** cost_P ≥ cost_Q **THEN** remove P

QUEUE: <SA,SBC,SBG,**SBA**>



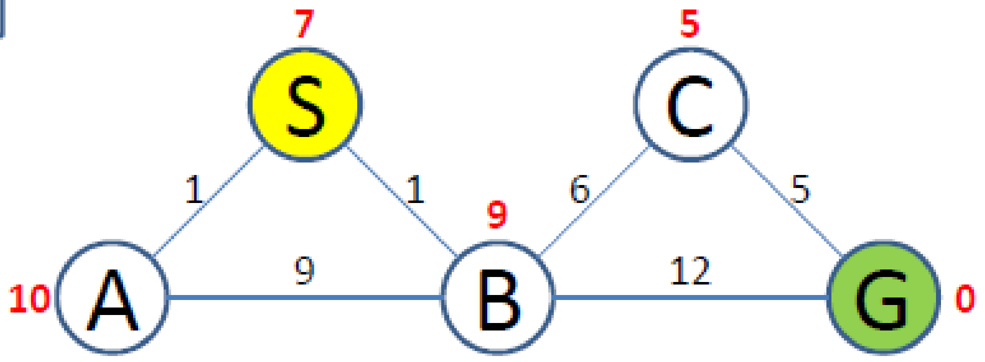
A* search example



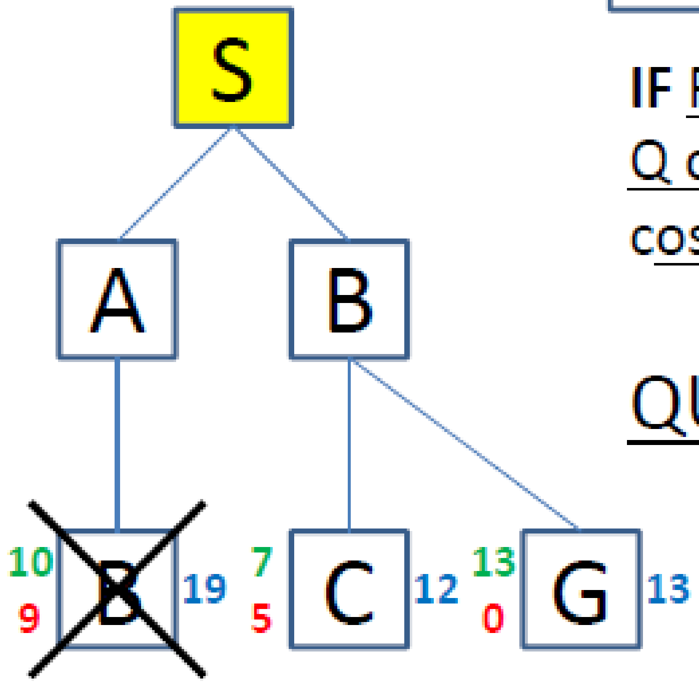
f = accumulated path cost + heuristic

Remove ***first path***, Create ***paths to all children***, Reject ***loops*** and ***Add paths***. ***Sort*** QUEUE ***by f***

QUEUE: <SBC,SBG,SAB>



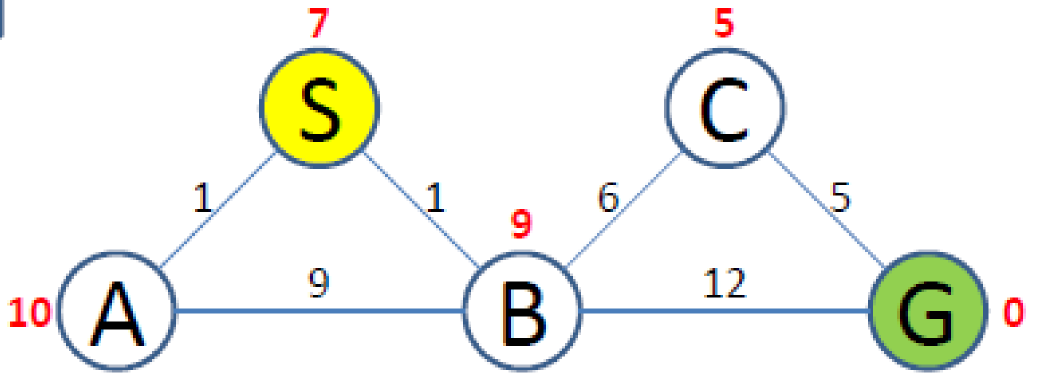
A* search example



f = accumulated path cost + heuristic

IF P terminating in I with cost_P && Q containing I with cost_Q **AND** cost_P ≥ cost_Q **THEN** remove P

QUEUE: <SBC,SBG,**SAB**>

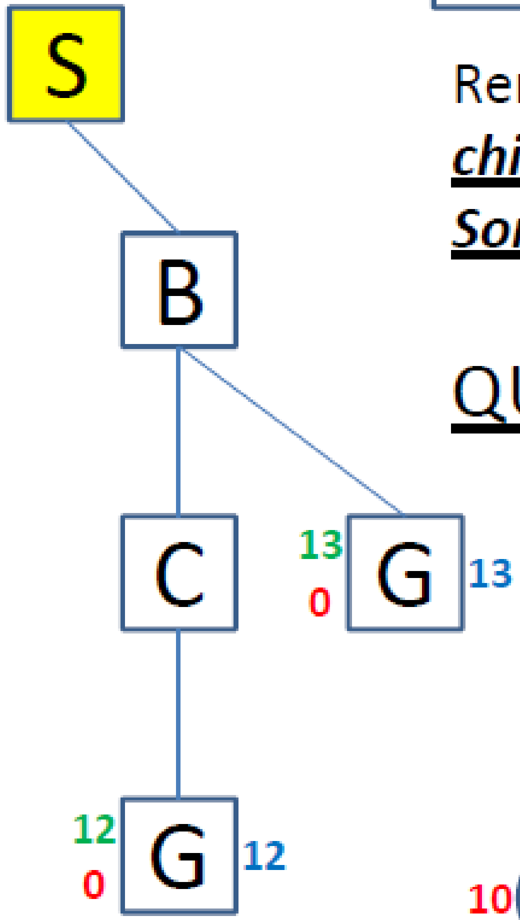


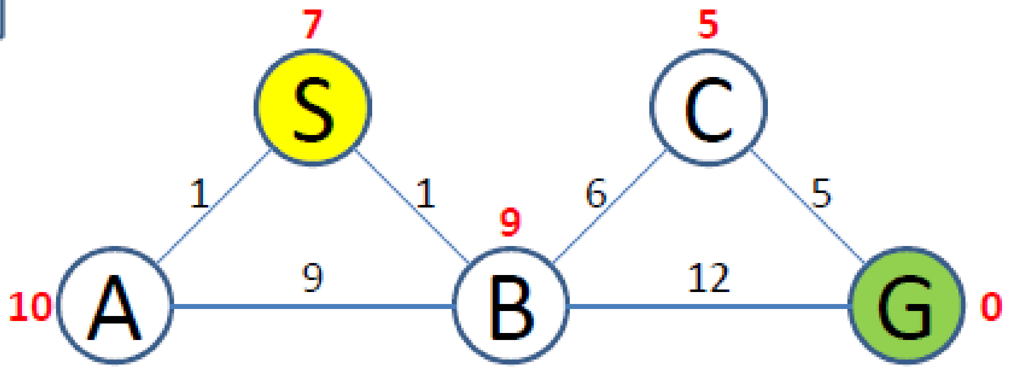
A^* search example

f = accumulated path cost + heuristic

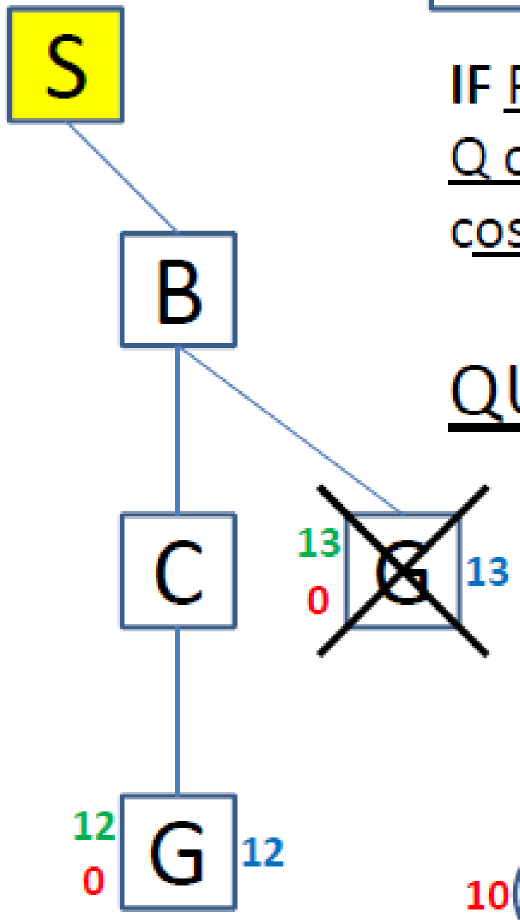
Remove ***first path***, Create ***paths to all children***, Reject ***loops*** and ***Add paths***. ***Sort*** QUEUE ***by f***

QUEUE: <SBCG,SBG>





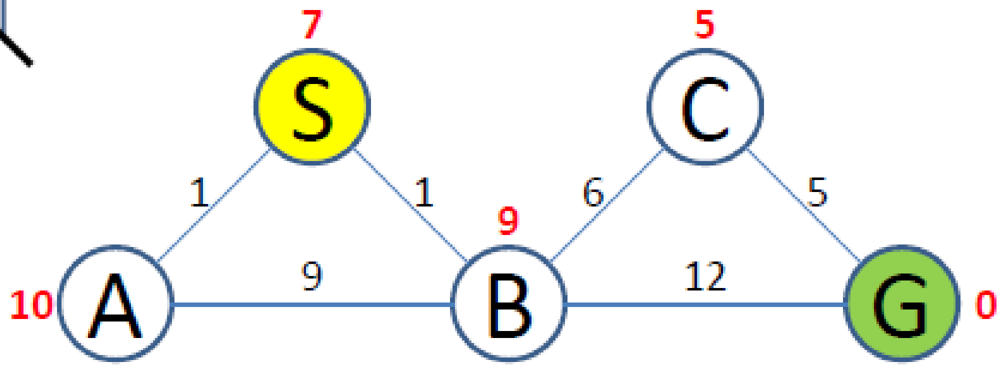
A* search example



f = accumulated path cost + heuristic

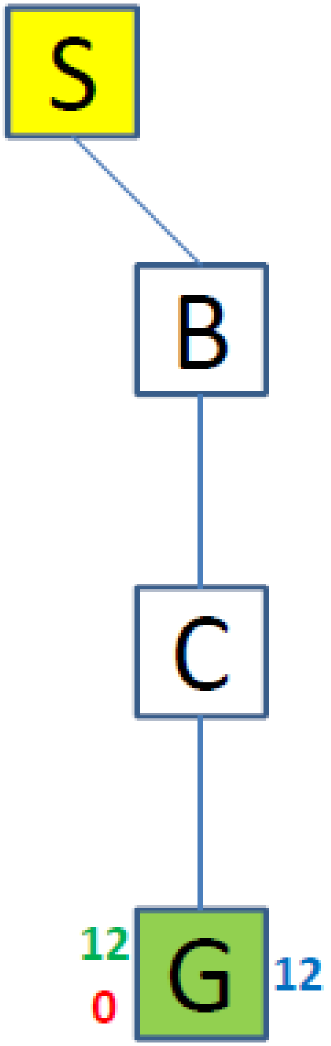
IF P terminating in I with cost_P && Q containing I with cost_Q **AND** cost_P ≥ cost_Q **THEN** remove P

QUEUE: <SBCG,**SBG**>



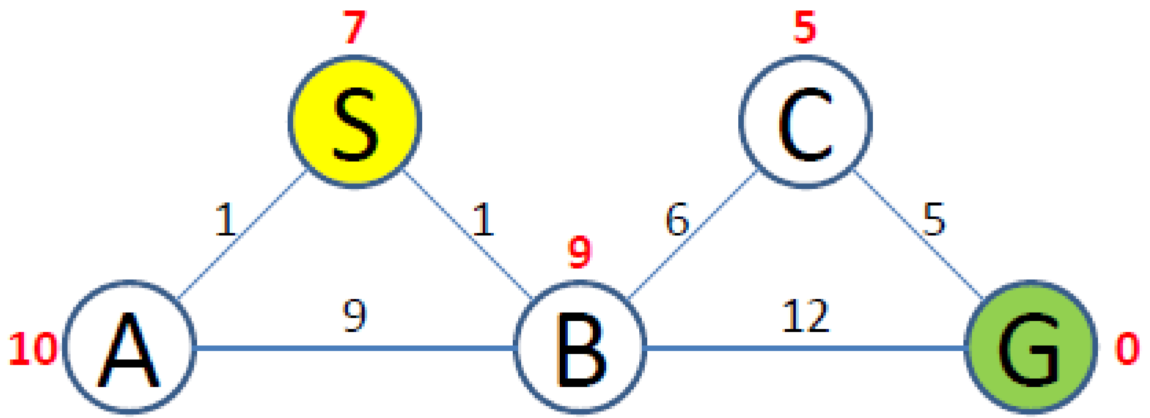
A^* search example

f = accumulated path cost + heuristic



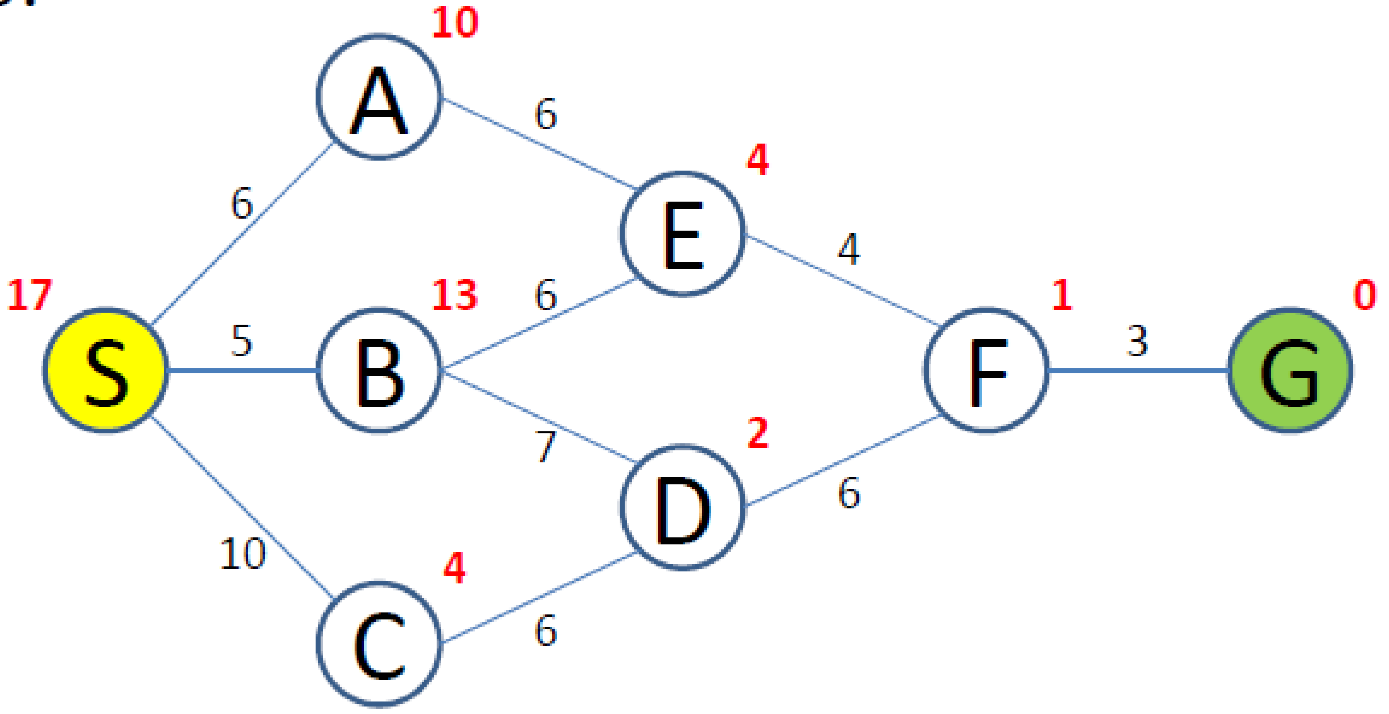
SUCCESS

QUEUE: <SBCG>

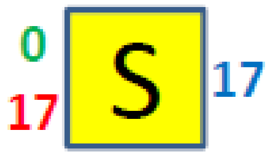


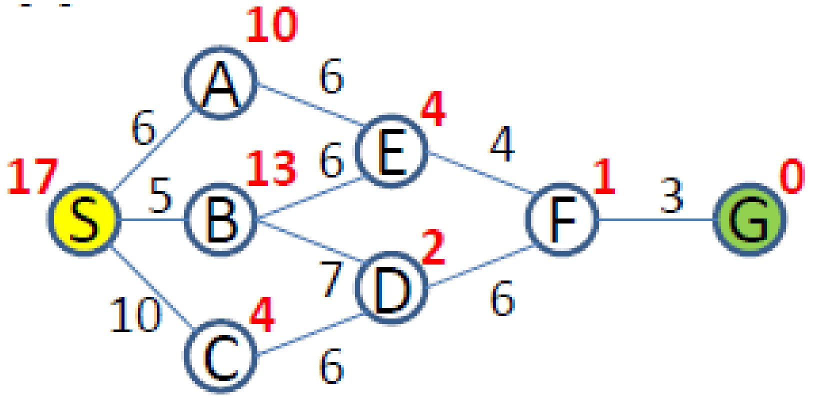
A^* search example

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



A* search example

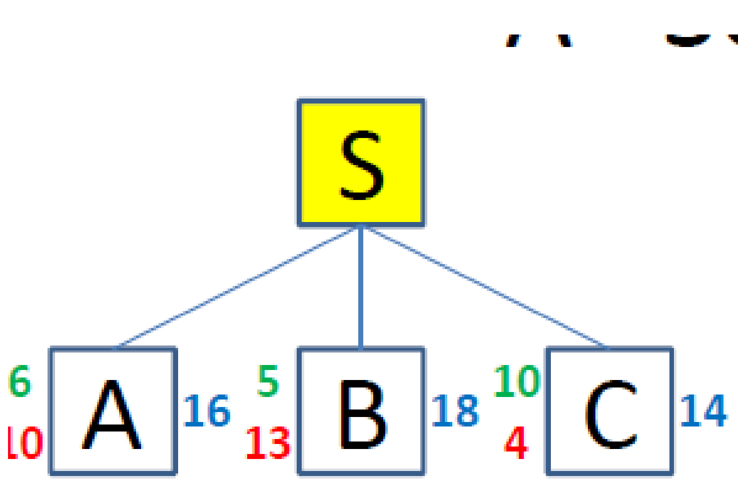


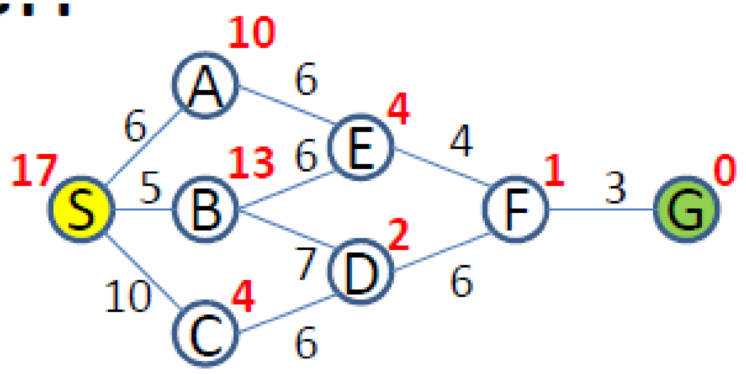


QUEUE:

S

A* search example





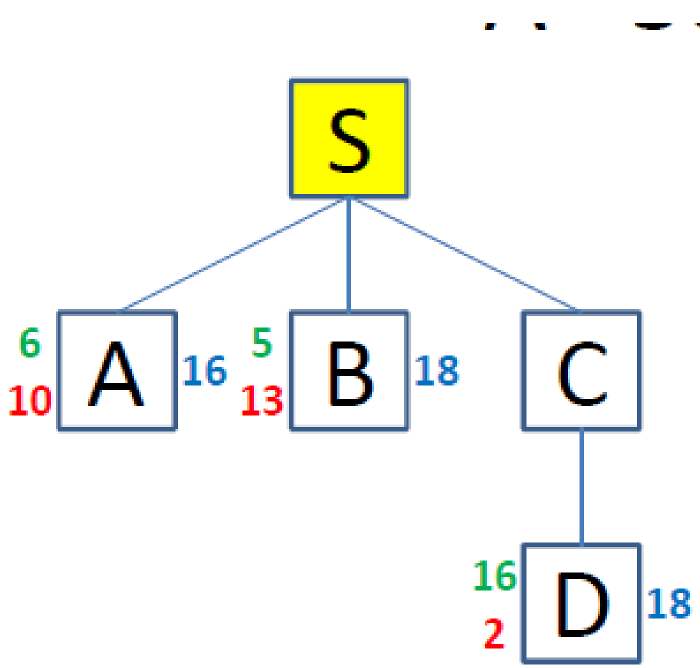
QUEUE:

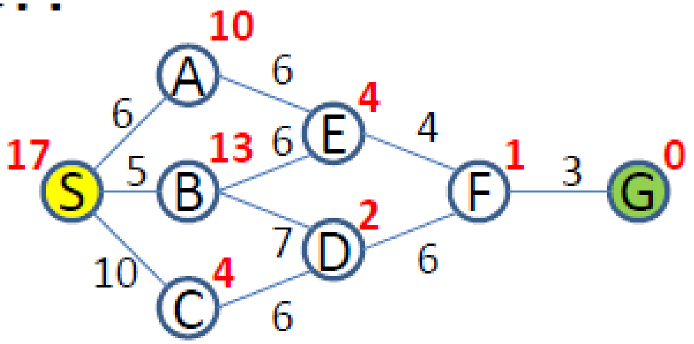
SC

SA

SB

A* search example





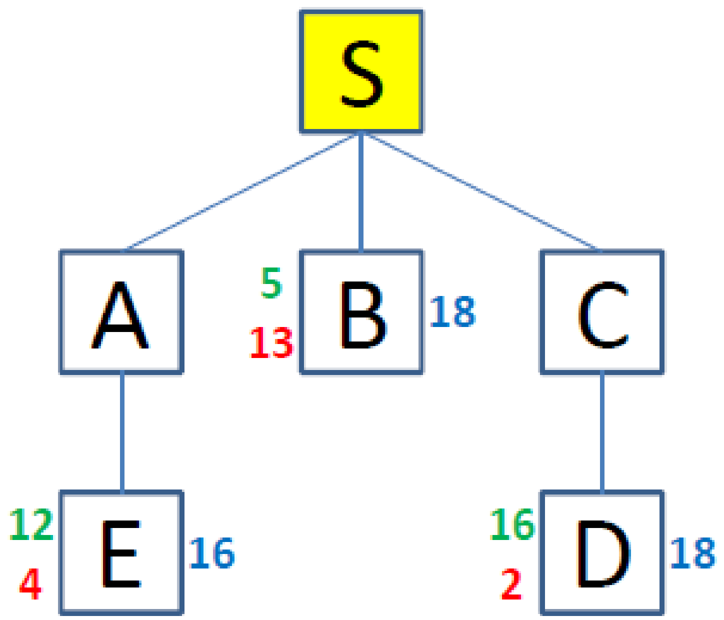
<u>QUEUE:</u>

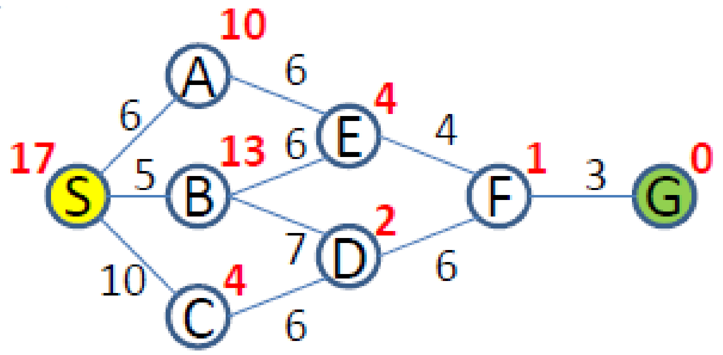
SA

SCD

SB

A* search example





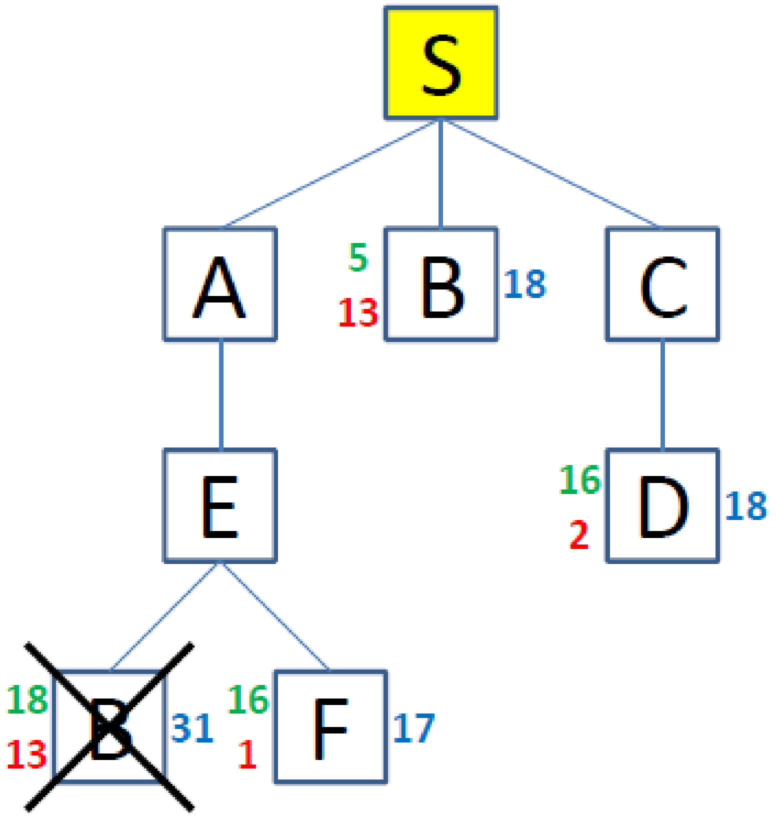
QUEUE:

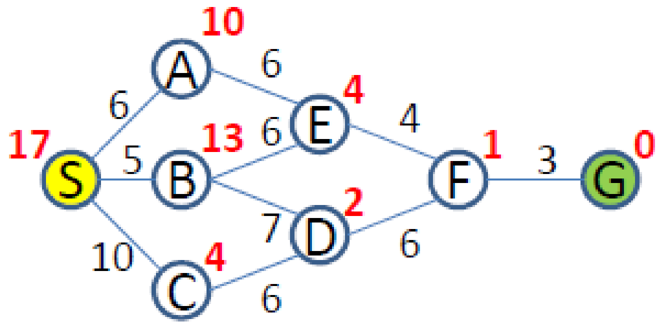
SAE

SCD

SB

A* search example





QUEUE:

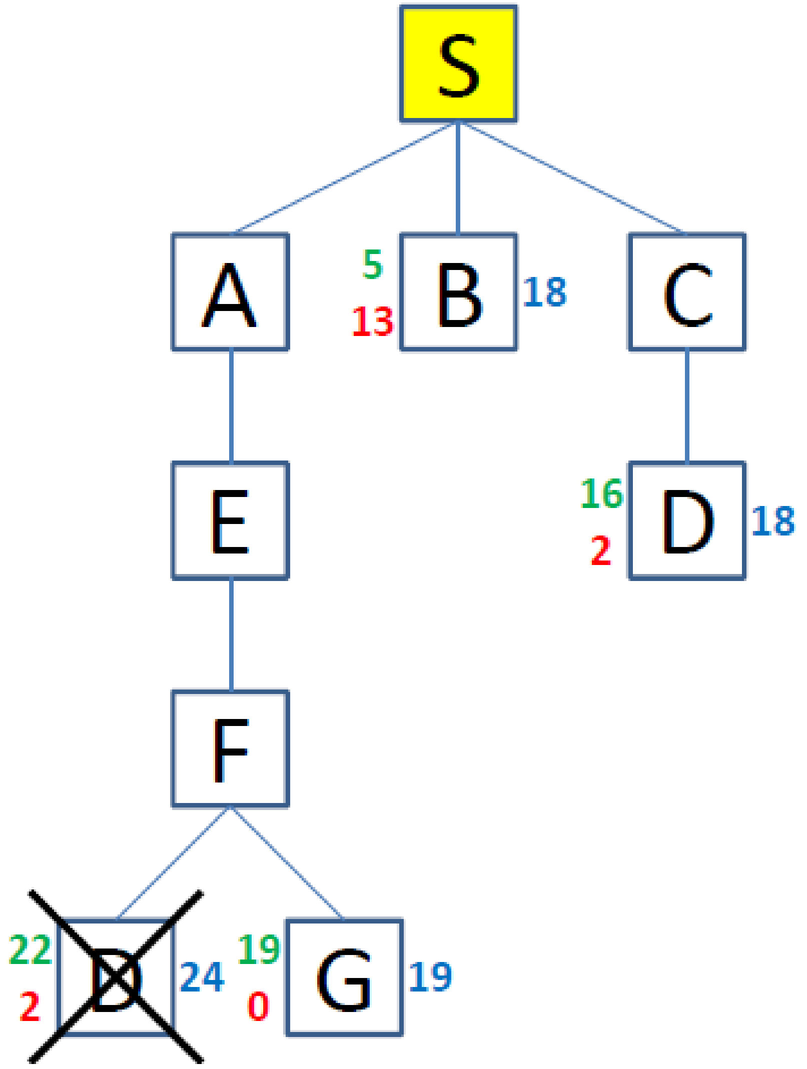
SAEF

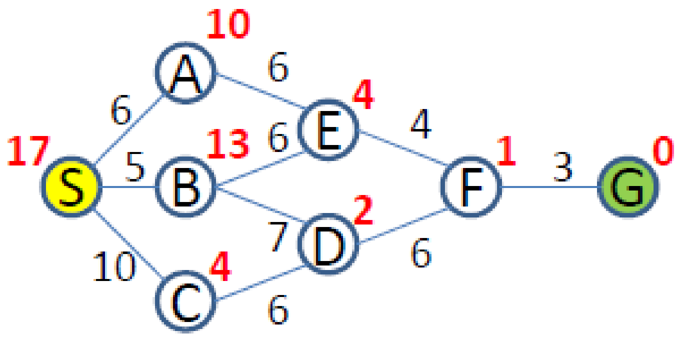
SCD

SB

SAEB

A* search example





<u>QUEUE:</u>

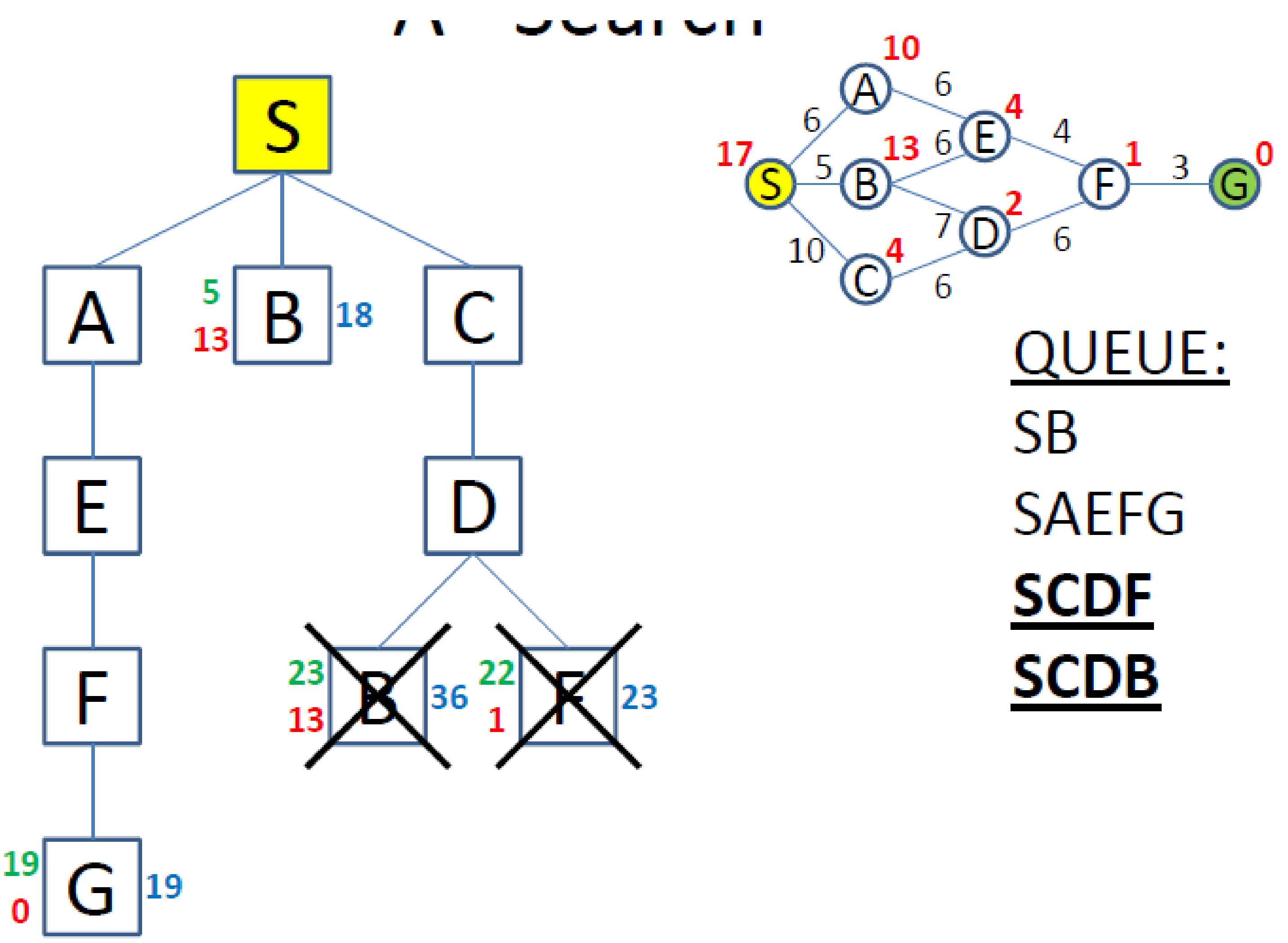
SCD

SB

SAEFG

<u>SAEFD</u>

A* search example



QUEUE:

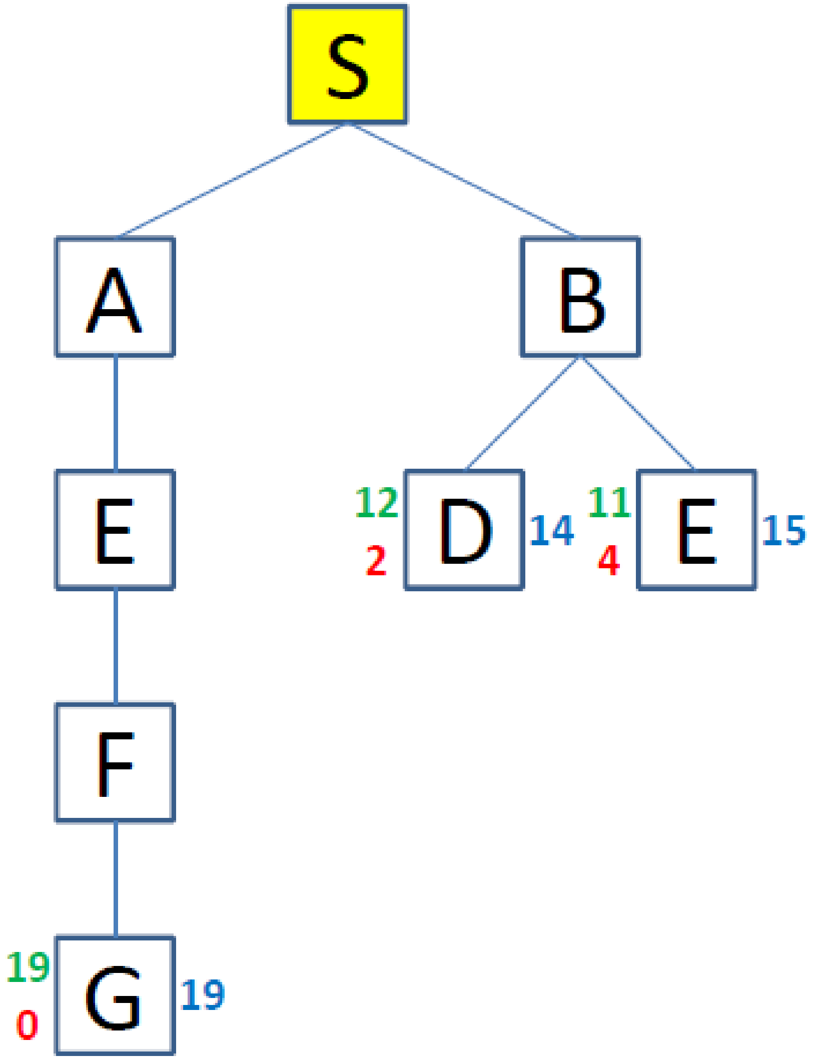
SB

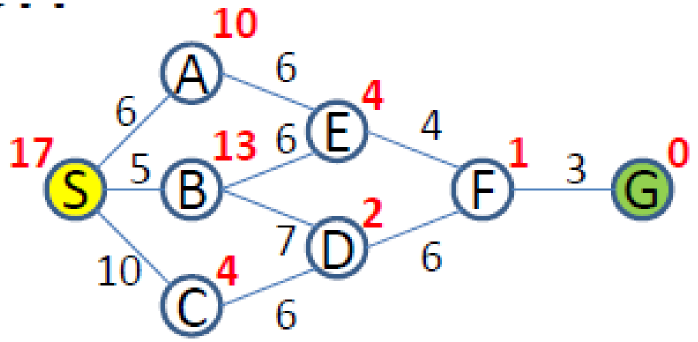
SAEFG

SCDF

SCDB

A* search example





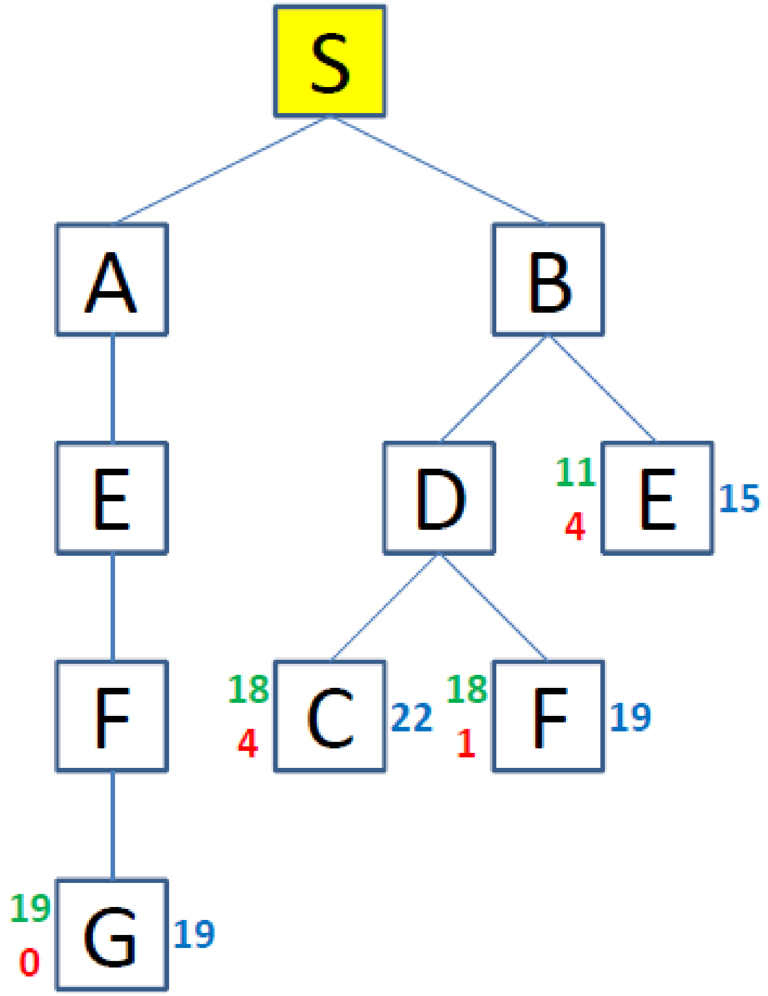
QUEUE:

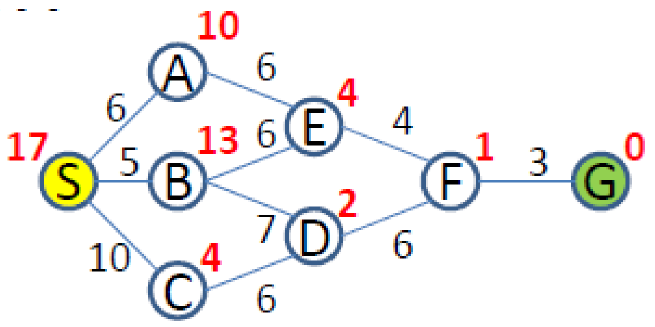
SBD

SBE

SAEFG

A* search example





QUEUE:

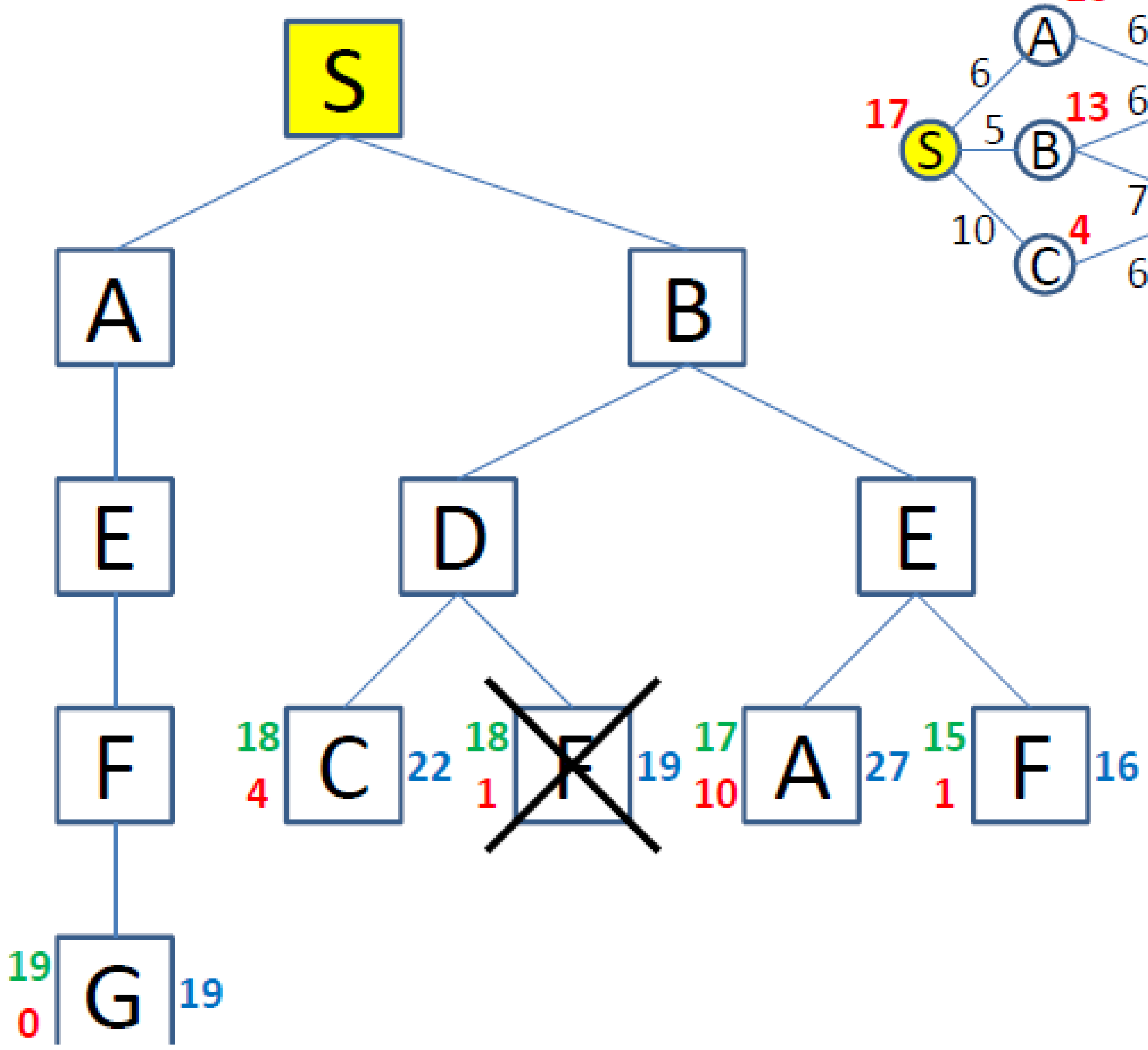
SBE

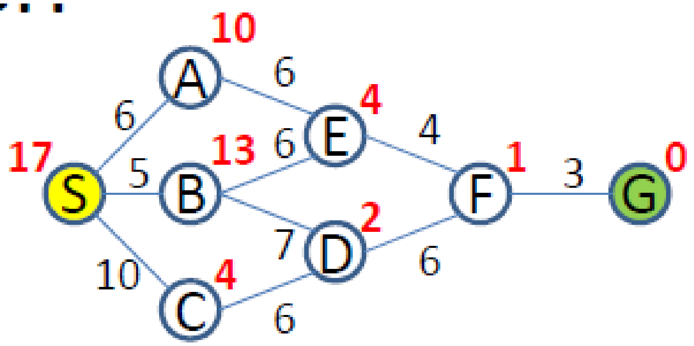
SBDF

SAEFG

SBDC

A^* search example





QUEUE:

SBEF

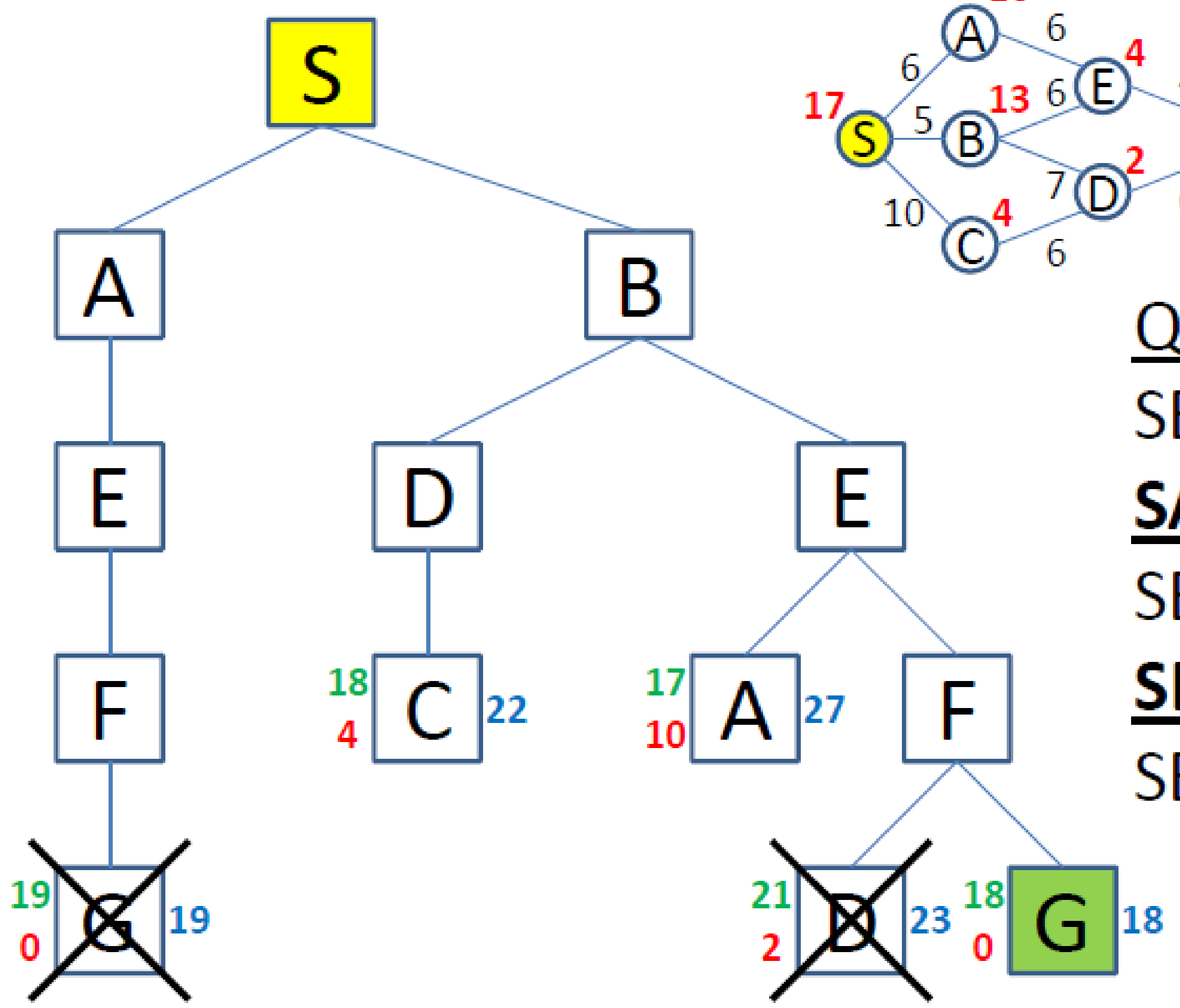
SAEFG

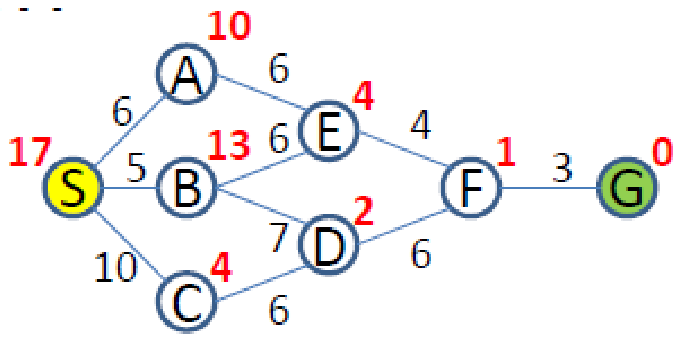
SBDF

SBDC

SBEA

A* search example





QUEUE:

SBEFG

SAEFG

SBDC

SBEFD

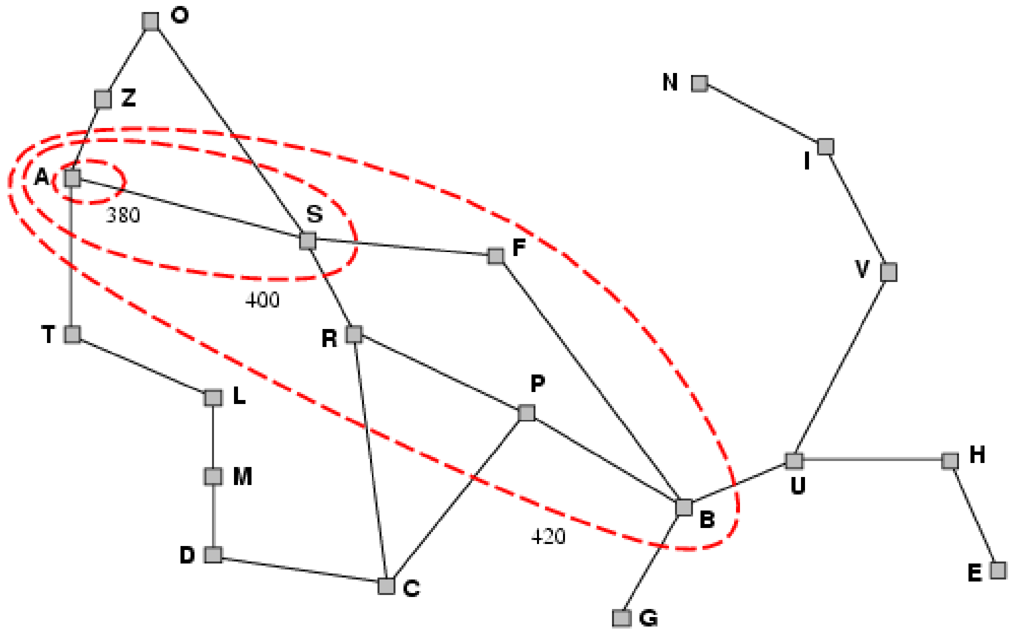
SBEA

9.2.2 Admissibility of A*

- Conditions that guarantee A* always finds minimal cost paths
  - Each node in the graph has a finite number of successors
  - All arcs in the graph have costs greater than some positive amount ε

Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



An alternative: IDA*

- IDA* = Iterative Deepening A*
- Performs a series of depth-first searches, each with a certain cost cut-off.
- **When the f value for a node exceeds this cut-off, then the search must backtrack**.
- If the goal node was not found during the search, then more depth-first searches are conducted with higher cut-offs until the goal is found.

IDA* Algorithm

Iterative deepening A* or IDA* is similar to iterative-deepening depth-first, but with the following modifications:

The depth bound modified to be an f-limit

f-bound ôf(S)

- Algorithm:
  - WHILE (goal is not reached) DO
    - f-bound ðf-limitted_search(f-bound)
      - Perform f-limited search with f-bound

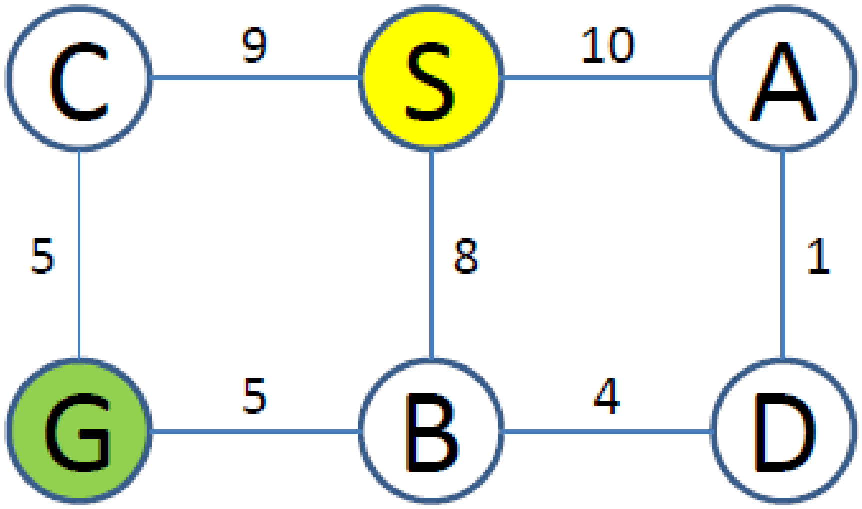
(See next slide)

IDA* Algorithm

- ***Input:***
  - <u>QUEUE</u> ⟵ Path only containing root
  - <u>f-bound</u> ⟵ Natural number
  - <u>f-new</u> ⟵ ∞
- ***Algorithm:***
  - **WHILE** (<u>QUEUE</u> not empty && goal not reached) **DO**
    - Remove **first path** from <u>QUEUE</u>
    - Create paths to children
    - Reject paths with loops
    - Add paths with **f(path) ≤ <u>f-bound</u>** to **front** of <u>QUEUE</u> *(depth-first)*
    - <u>f-new</u> ⟵ minimum({<u>f-new</u>} ∪ {f(P) | P is rejected path})
  - **IF** goal reached **THEN** success **ELSE** report <u>f-new</u>

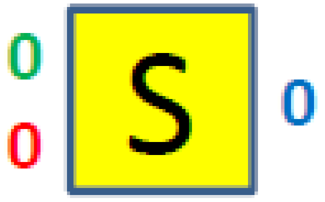
Problem

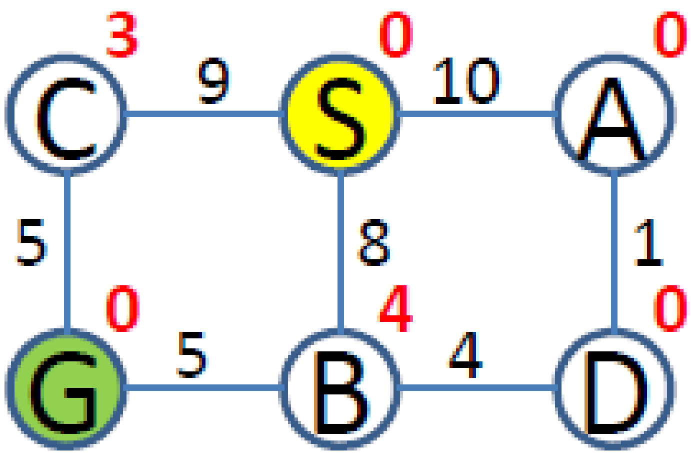
Perform the IDA* Algorithm on the following figure.



	S	A	B	C	D	G
heuristic	0	0	4	3	0	0

IDA* Search

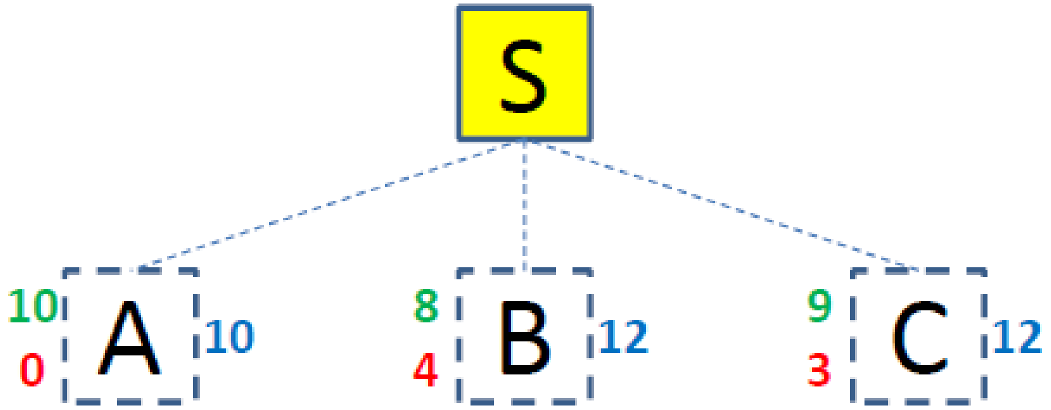




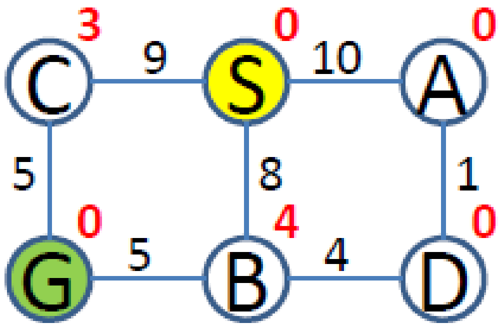
f-bound = 0

f-new = ∞

IDA* Search



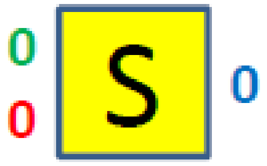
Children are explored **<u>depth-first</u>!**

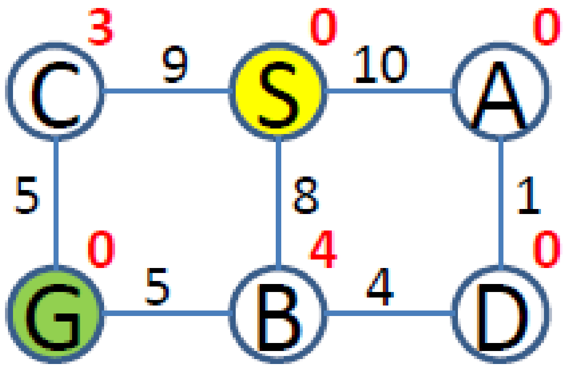


f-bound = 0

f-new = 10

IDA* Search

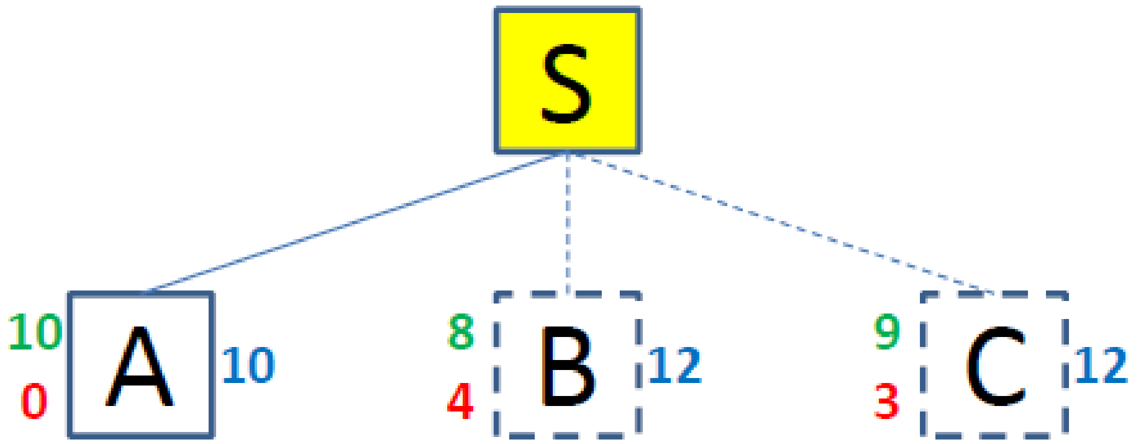


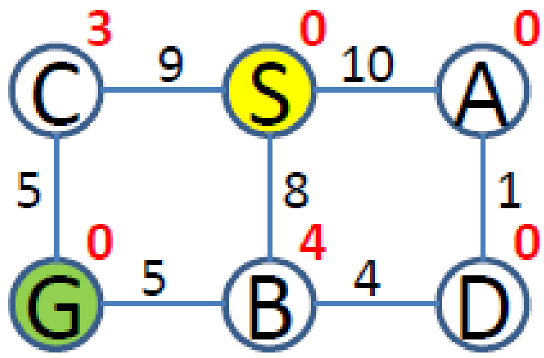


f-bound = 10

f-new = ∞

IDA* Search

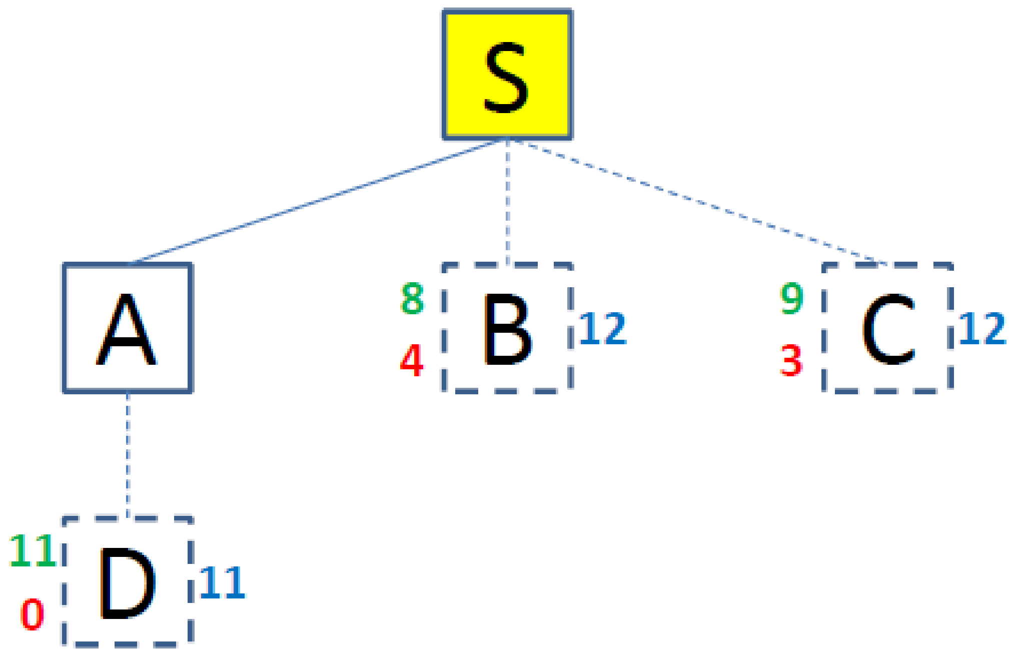


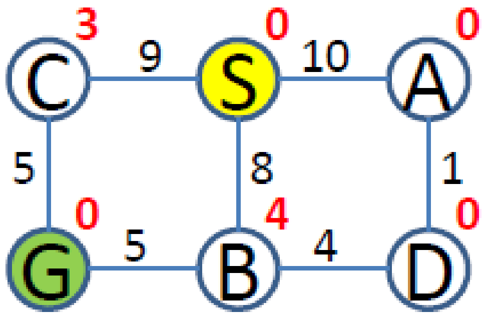


f-bound = 10

f-new = 12

IDA* Search

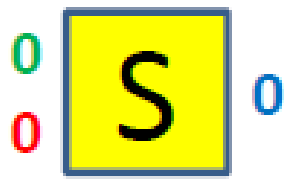


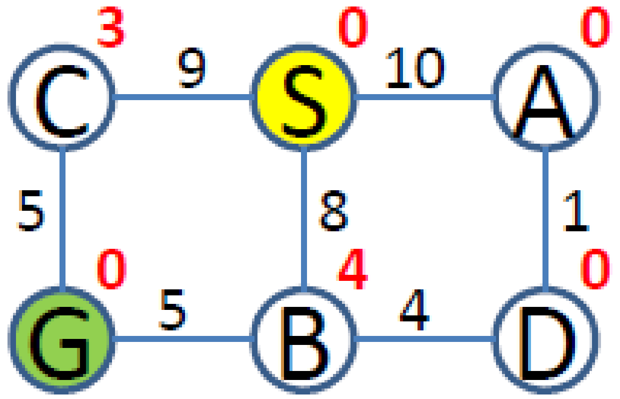


f-bound = 10

f-new = 11

IDA* Search

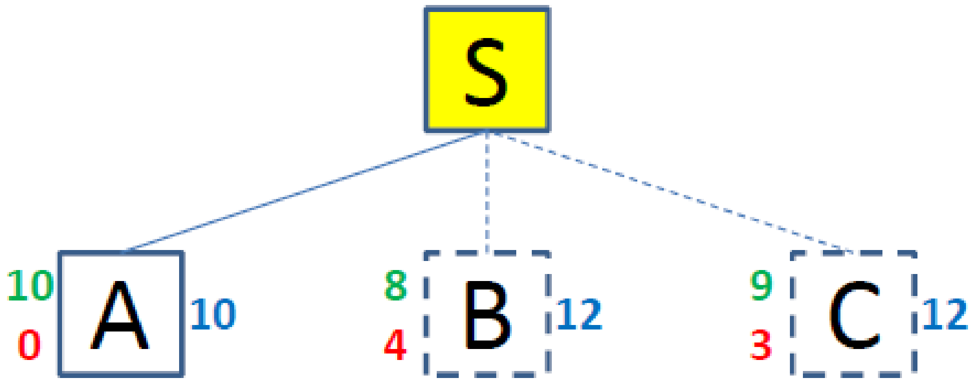


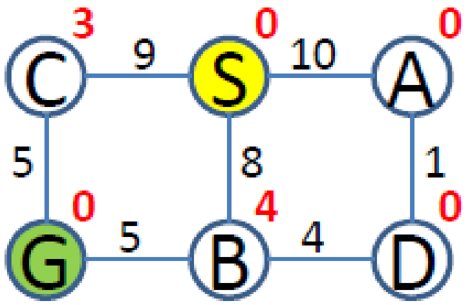


f-bound = 11

f-new = ∞

IDA* Search

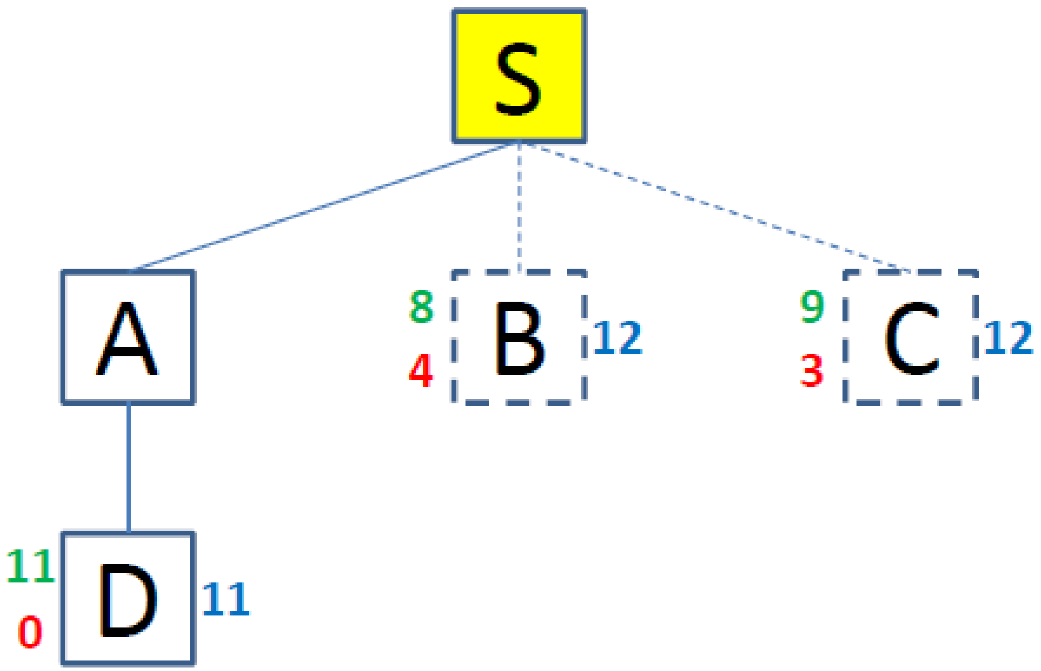


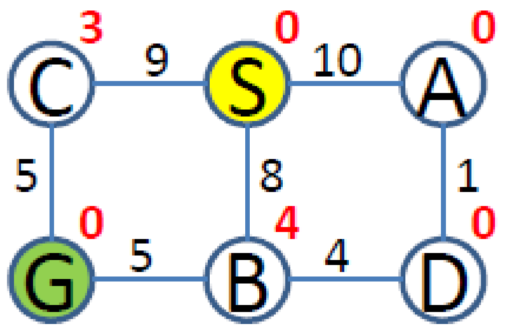


f-bound = 11

f-new = 12

IDA* Search

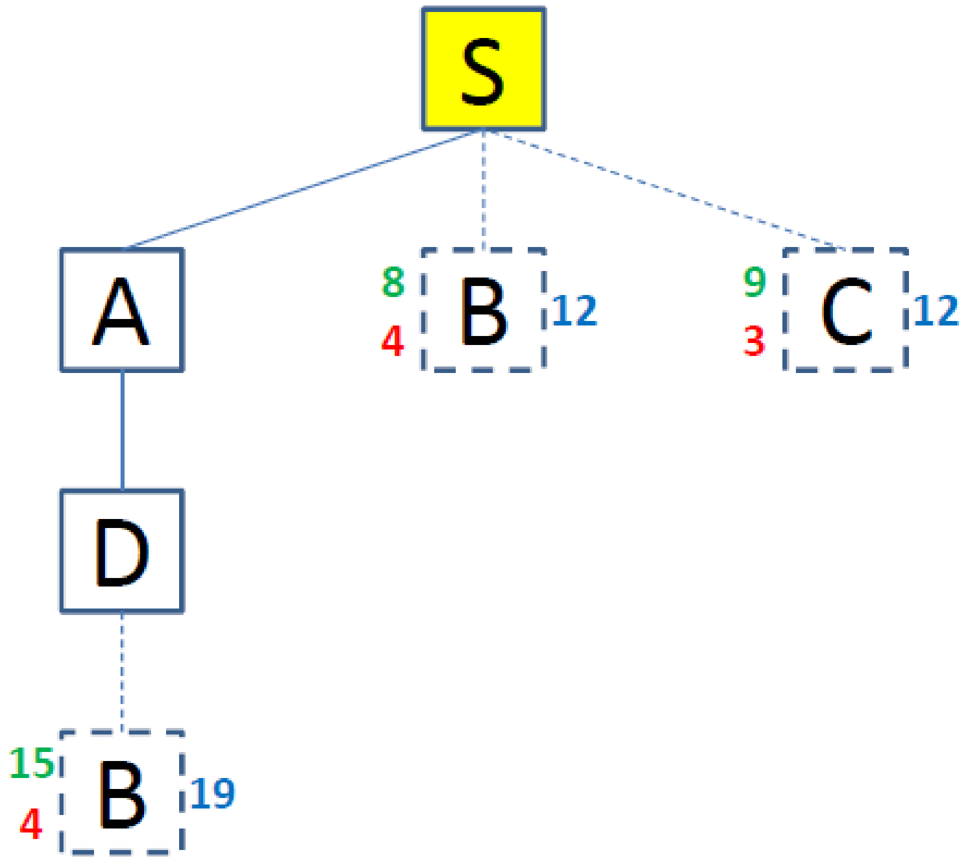


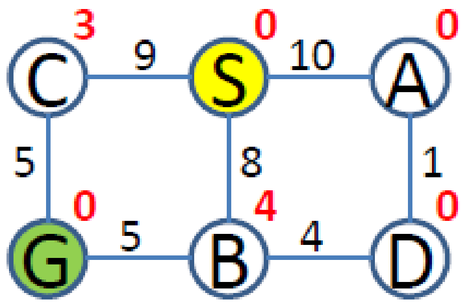


f-bound = 11

f-new = 12

IDA* Search

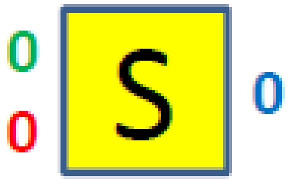


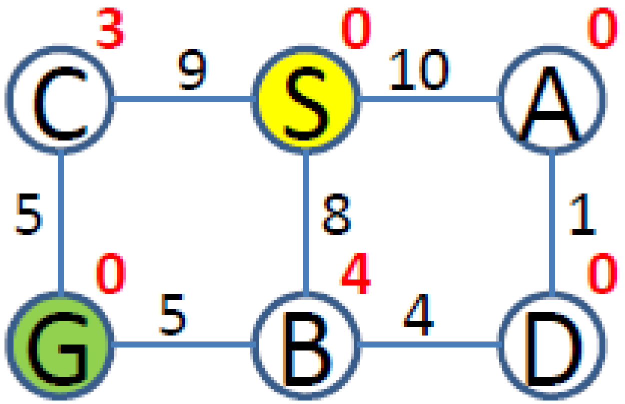


f-bound = 11

f-new = 12

IDA* Search

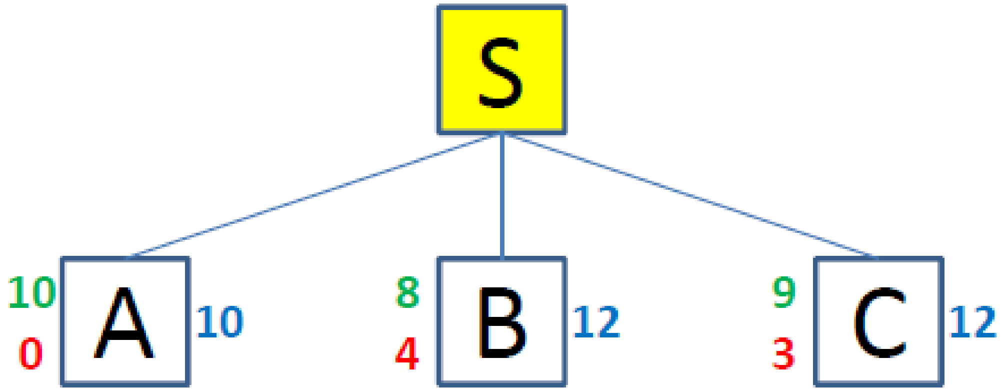


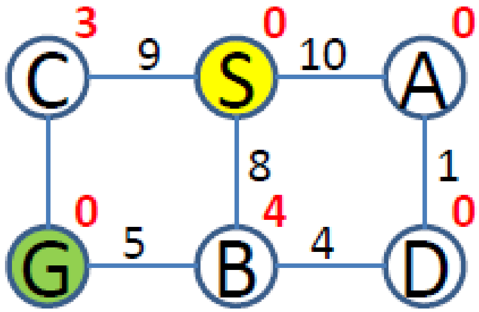


f-bound = 12

f-new = ∞

IDA* Search

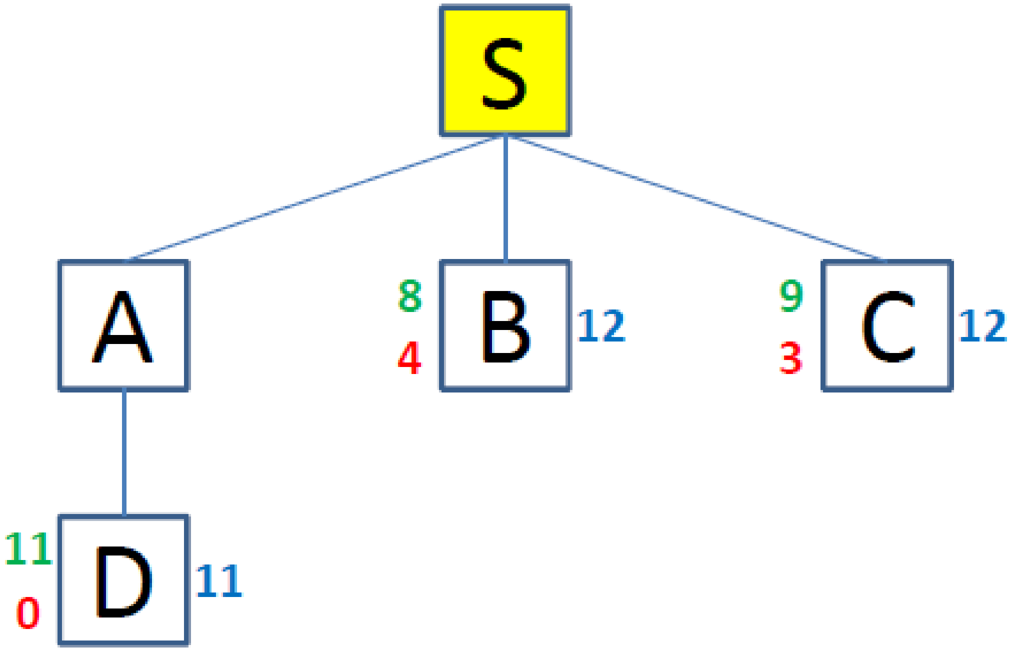


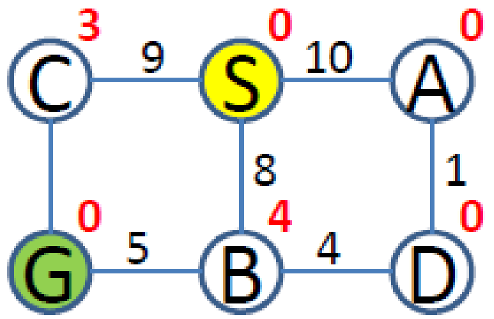


f-bound = 12

f-new = ∞

IDA* Search

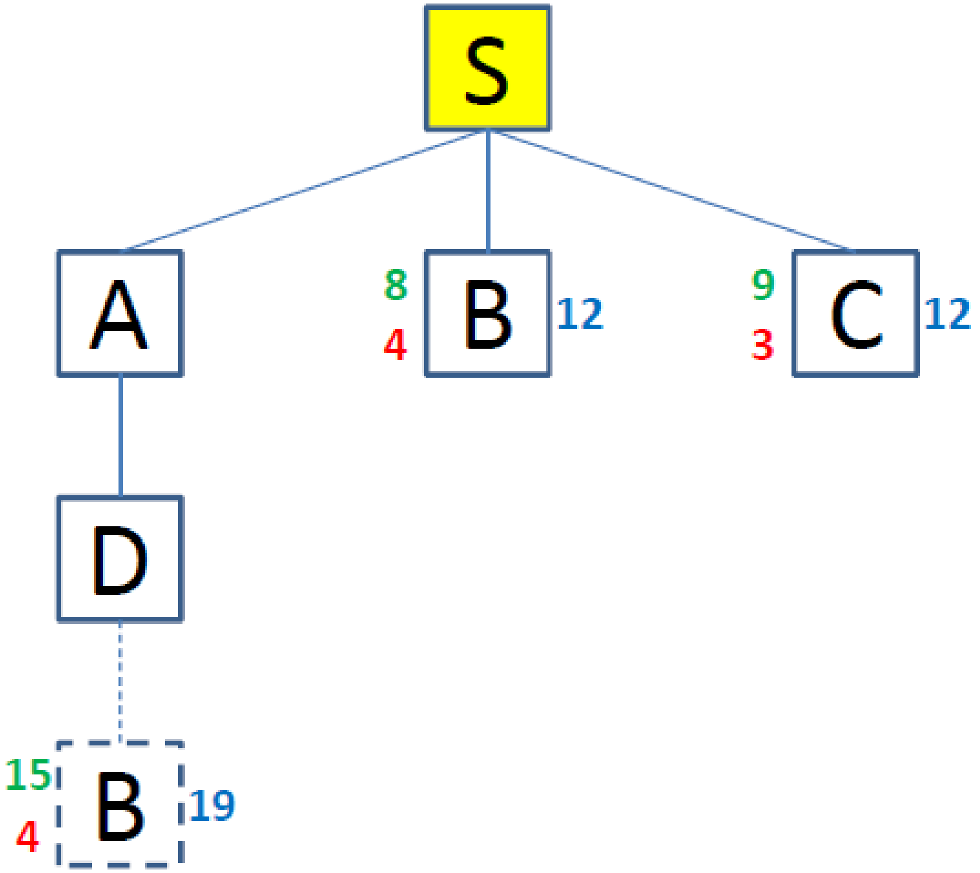


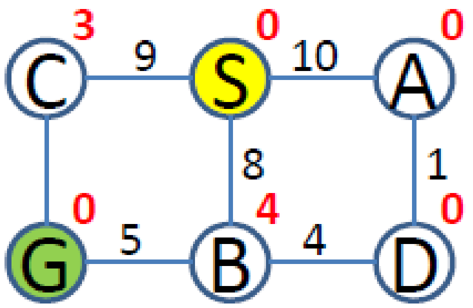


f-bound = 12

f-new = ∞

IDA* Search

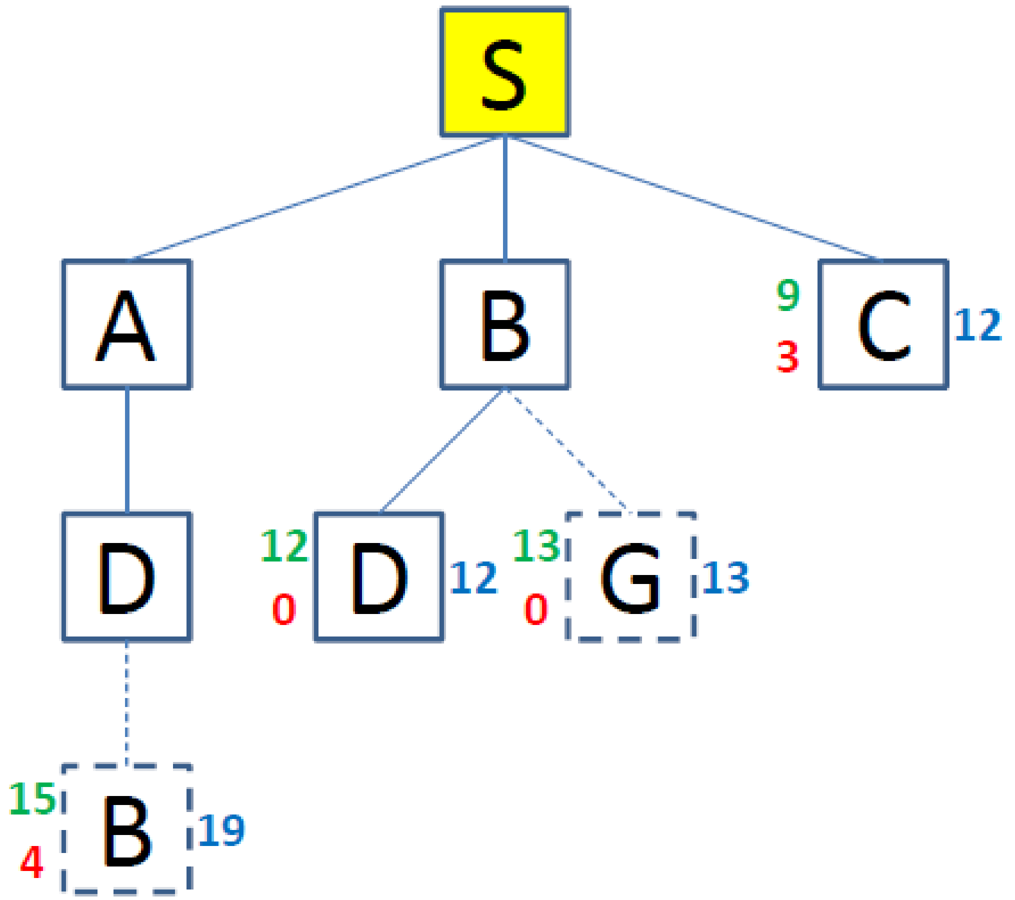


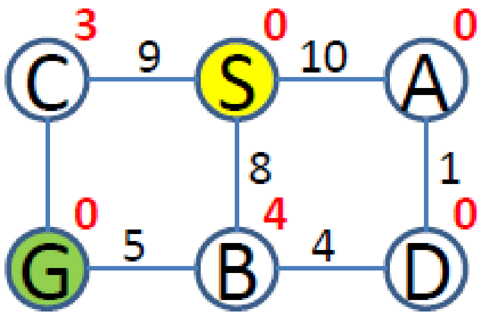


f-bound = 12

f-new = 19

IDA* Search

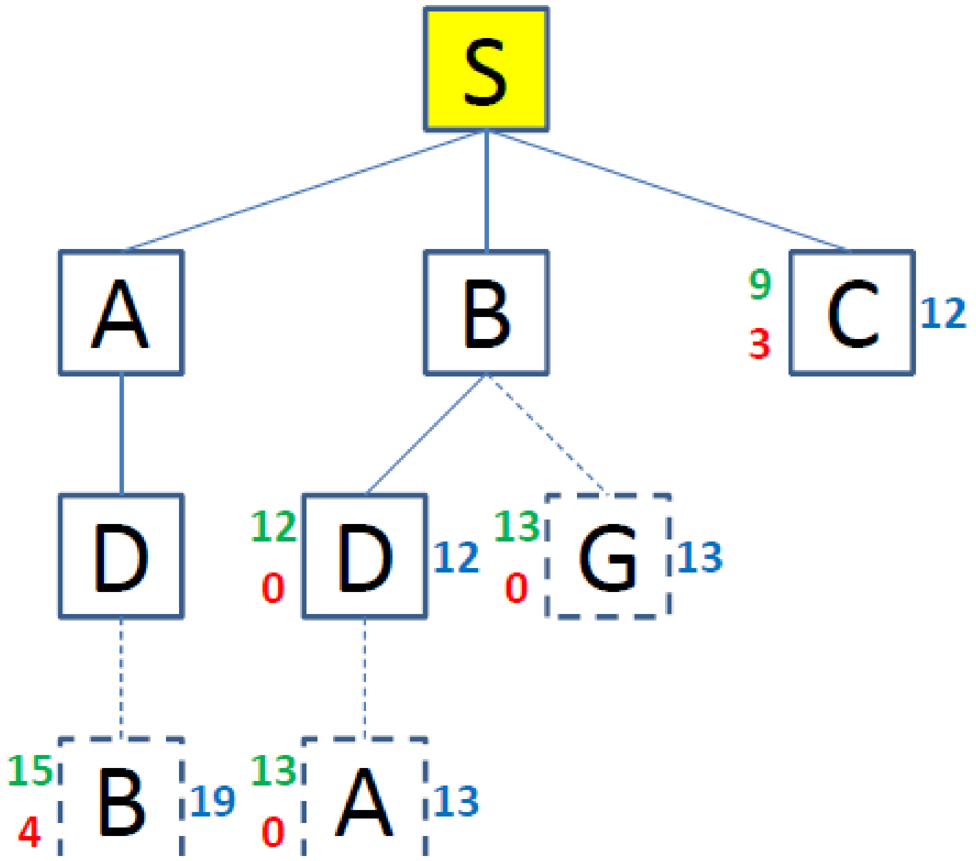


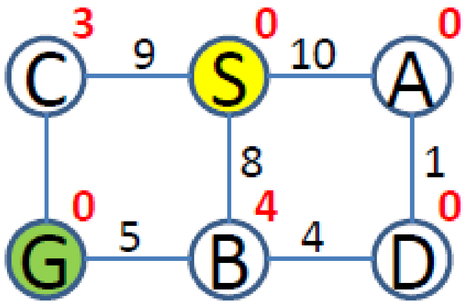


f-bound = 12

f-new = 13

IDA* Search

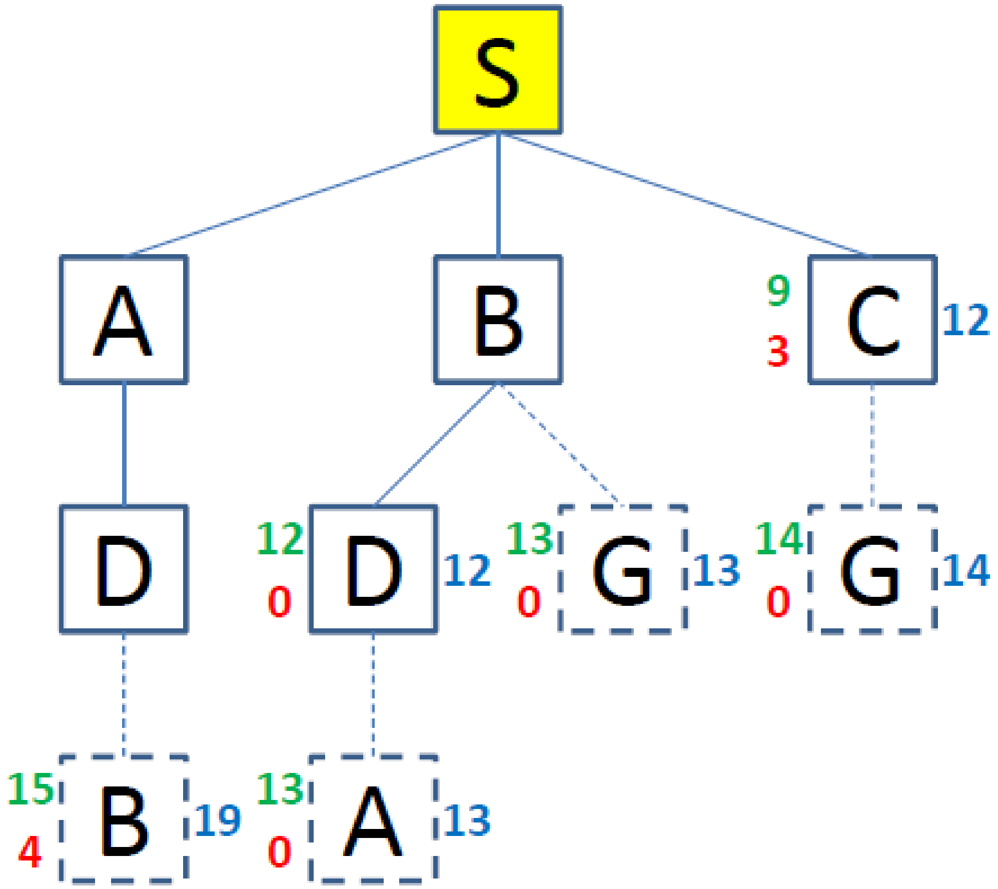


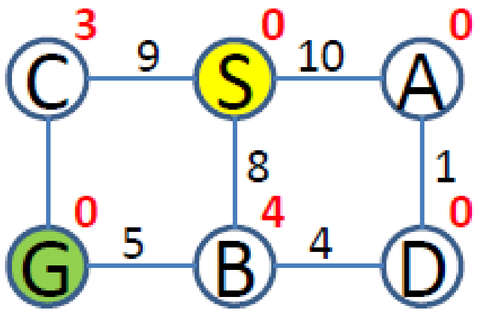


f-bound = 12

f-new = 13

IDA* Search





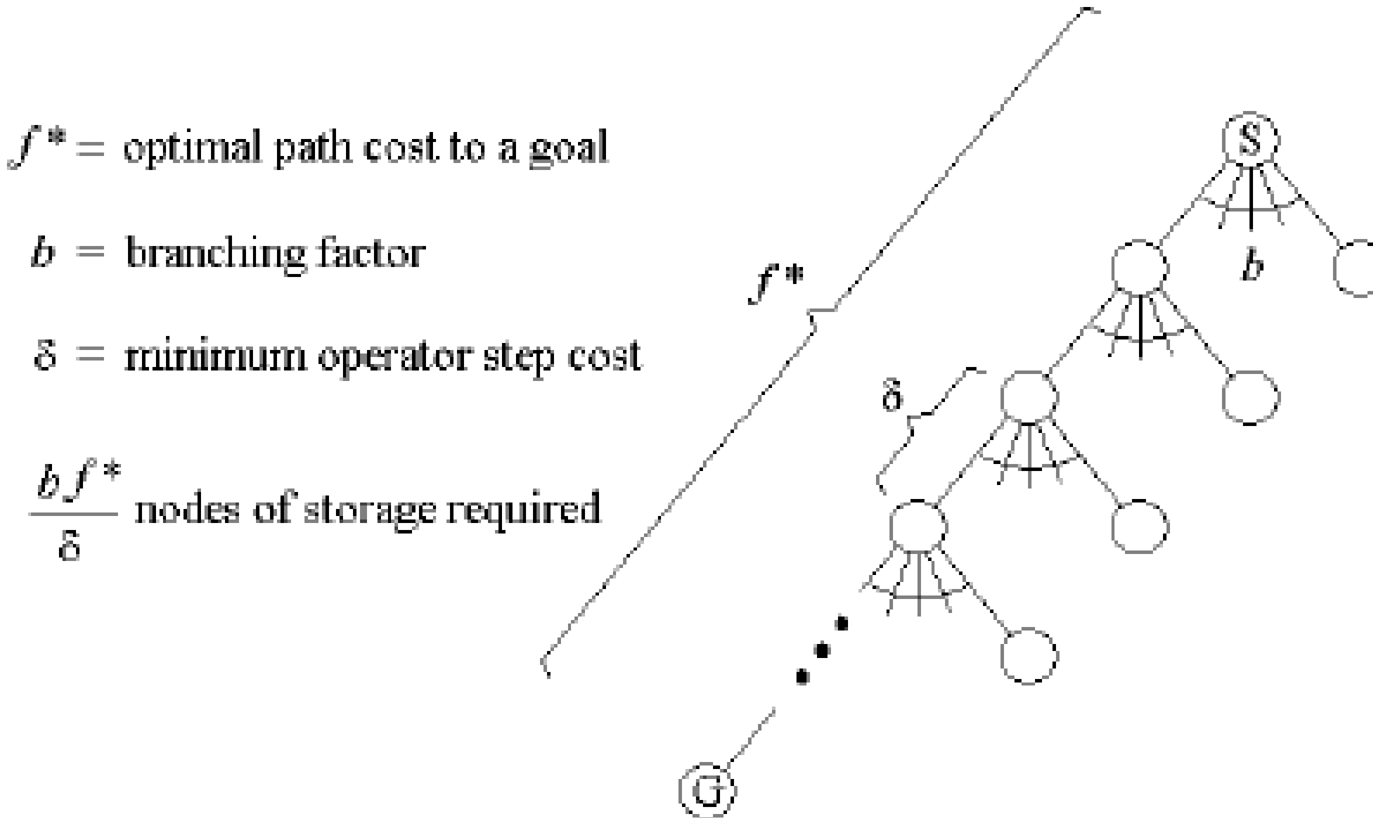
f-bound = 12

f-new = 13

Why do we use IDA*?

IDA* is complete, optimal, and optimally efficient (assuming a consistent, admissible heuristic), and requires only a polynomial amount of storage in the worst case

IDA* uses very little memory



f^* = optimal path cost to a goal

b = branching factor

δ = minimum operator step cost

$\frac{bf^*}{\delta}$ nodes of storage required

SMA*

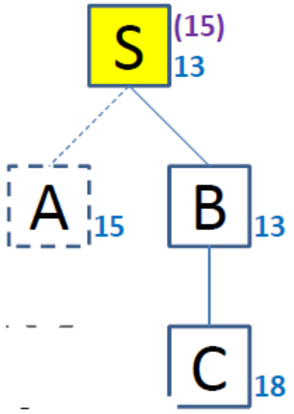
- **Simplified Memory Bounded A*** is a shortest path algorithm based on the A* algorithm.
- The main advantage of SMA* is that it uses a bounded memory, while the A* algorithm might need exponential memory.
- All other characteristics of SMA* are inherited from A*.

SMA* Algorithm

Optimizes A* to work within reduced memory

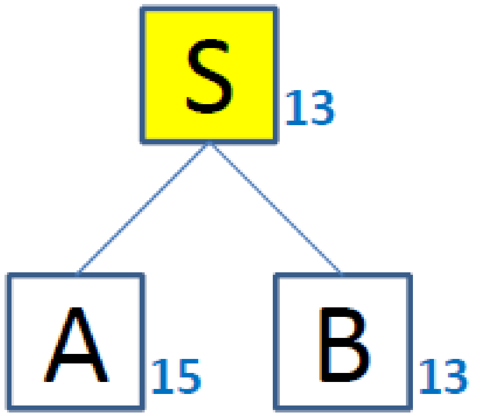
- **Key Idea:**
  - **IF** memory **full** for **extra node (C)**
  - **Remove highest f-value leaf (A)**
  - **Remember best-forgotten child** in each parent node **(15 in** S)

E.g. Memory of 3 nodes only



SMA* Algorithm

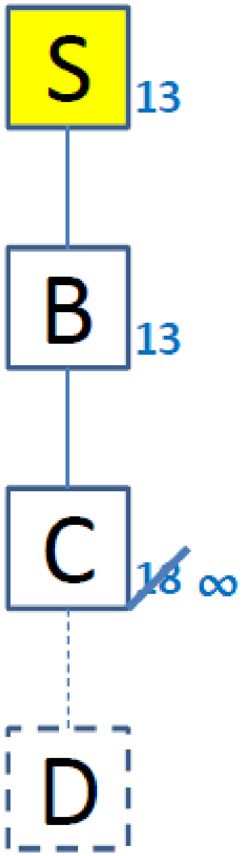
- **Generate Children 1 by 1**
  - **Expanding**: add <u>1 child at the time</u> to QUEUE
  - Avoids **memory overflow**
  - **Allows monitoring** if nodes need deletion



First add A later B

SMA* Algorithm

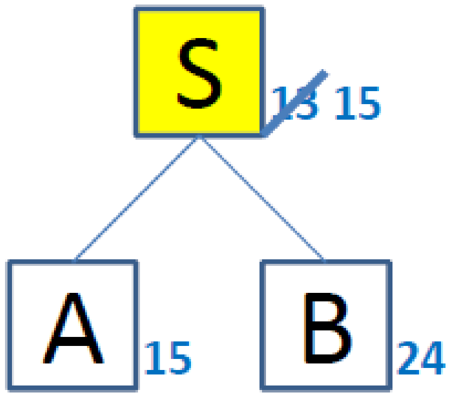
- **Too long paths: Give up**
  - **Extending** path **cannot fit** in **memory**
    - **give up (C)**
  - Set **f-value** node **(C)** to ∞
    - **Remembers:**
      path cannot be found here



E.g. Memory of 3 nodes only

SMA* Algorithm

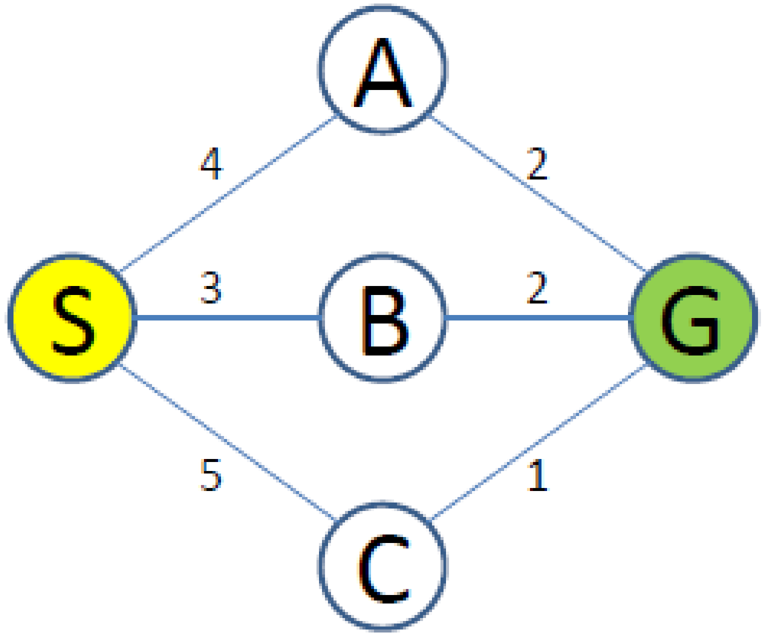
- **Adjust f-values**
  - **IF** all children M_i of node N have been explored
  - **AND** $\forall i$: $f(S \ldots M_i) > f(S \ldots N)$
  - **THEN reset** (through N $\Longrightarrow$ through children)
    - $f(S \ldots N) = \min\{f(S \ldots M_i) \mid M_i \text{ child of } N\}$



Better estimate for f(S)

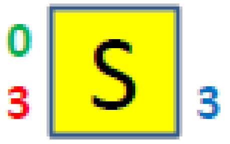
SMA* By Example

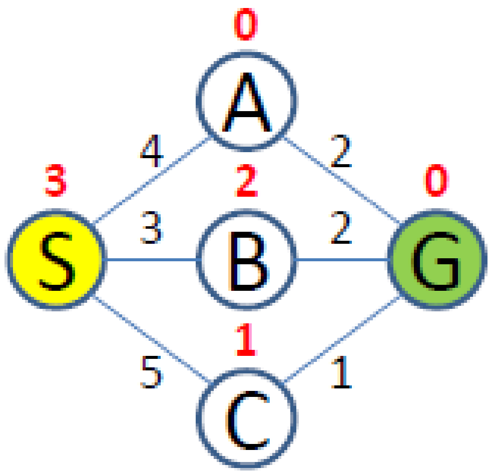
Perform SMA* (memory: 3 nodes) on the following figure.

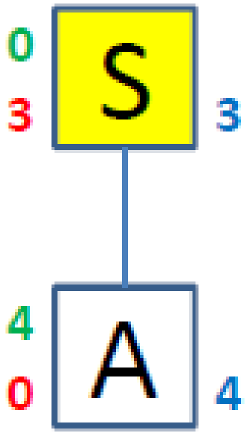


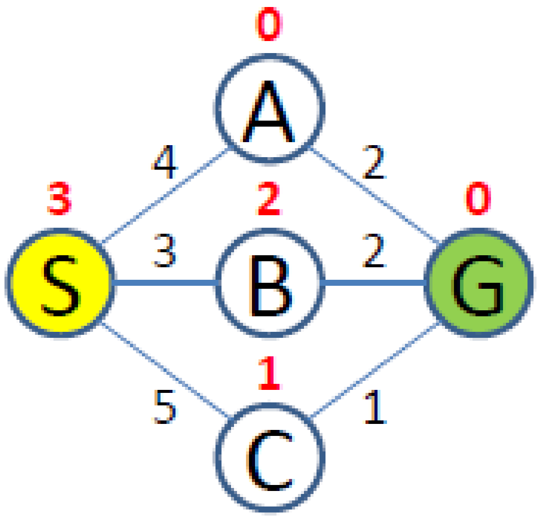
	S	A	B	C	G
heuristic	3	0	2	1	0

SMA* By Example



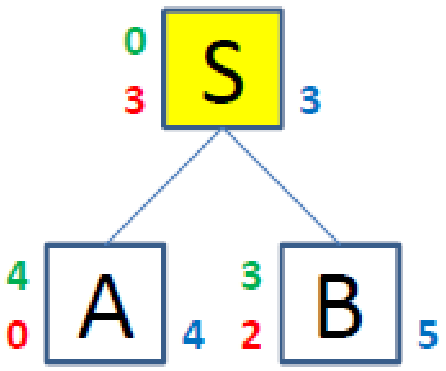


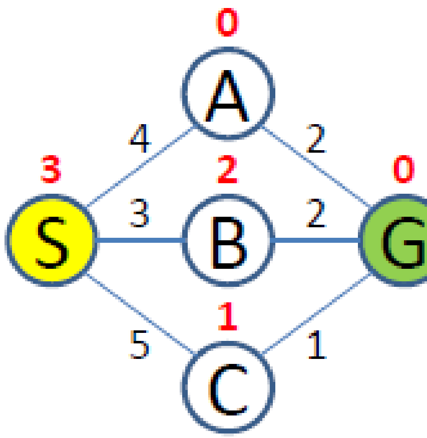




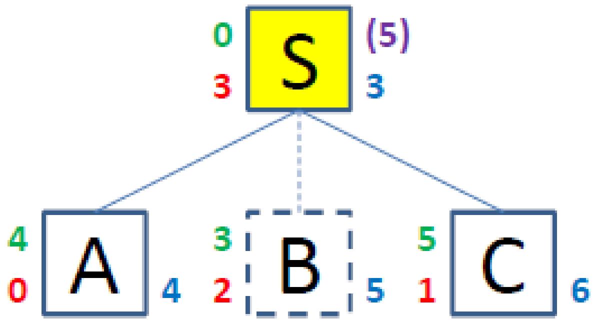
Generate children
(One by one)

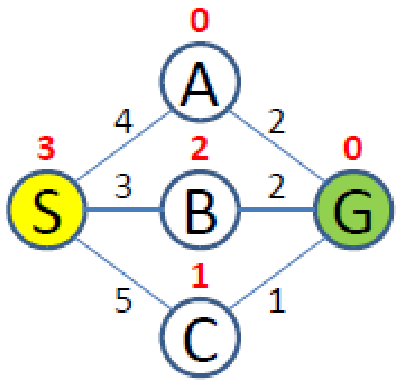
SMA* By Example





Generate childre
(One by one)

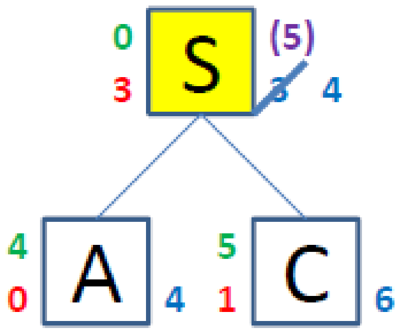


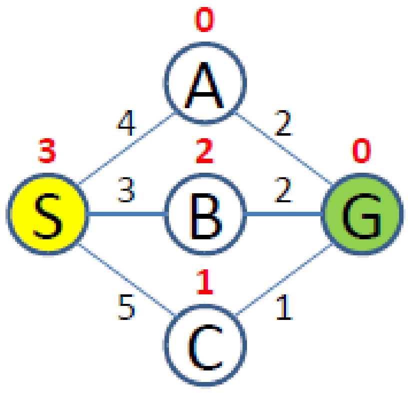


Generate children
(One by one)

Memory full

SMA* By Example

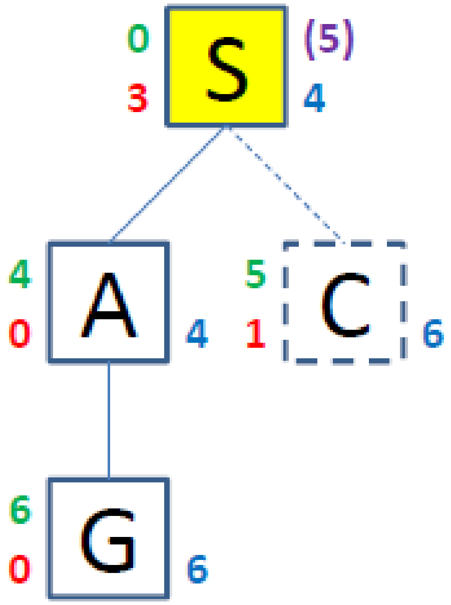


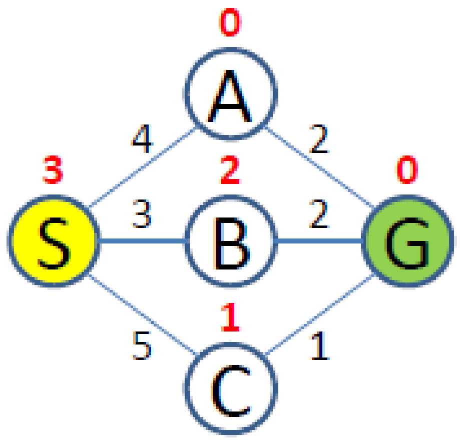


All children are explored

Adjust f-values

SMA* By Example

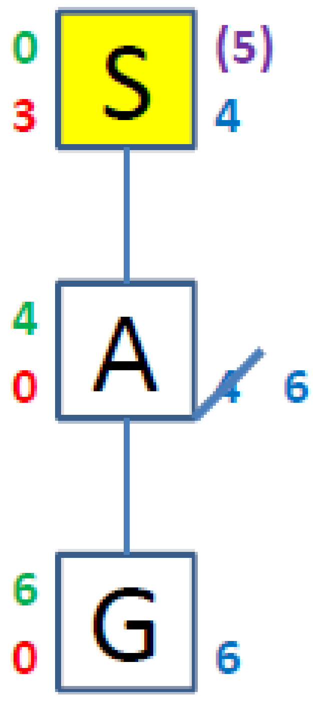


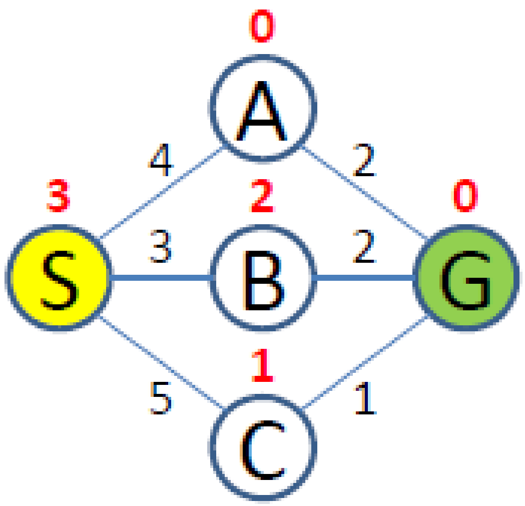


Generate children
(One by one)

Memory full

SMA* By Example

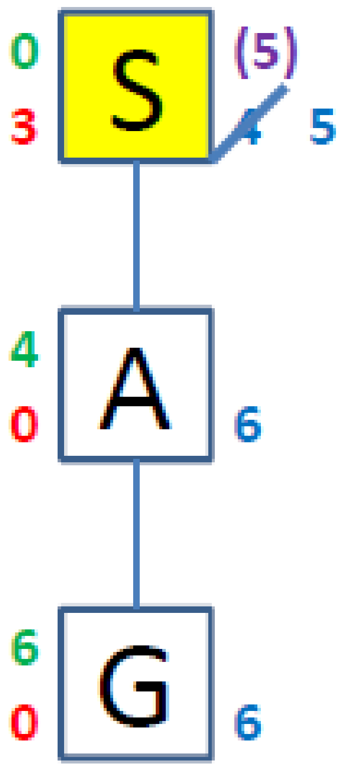


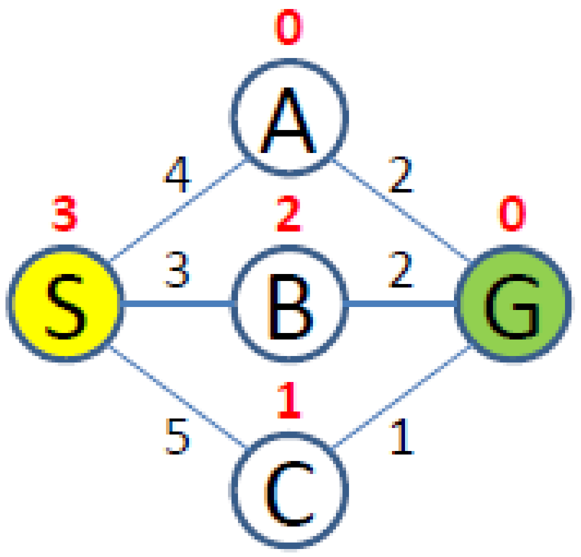


All children are explored

Adjust f-values

SMA* By Example

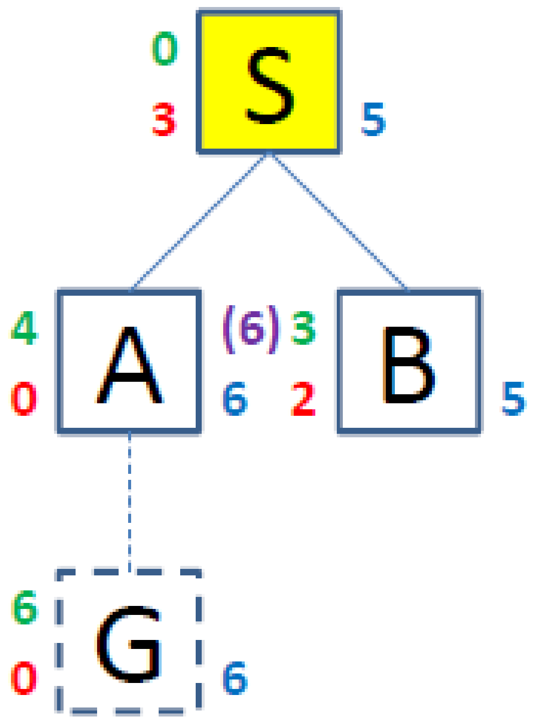


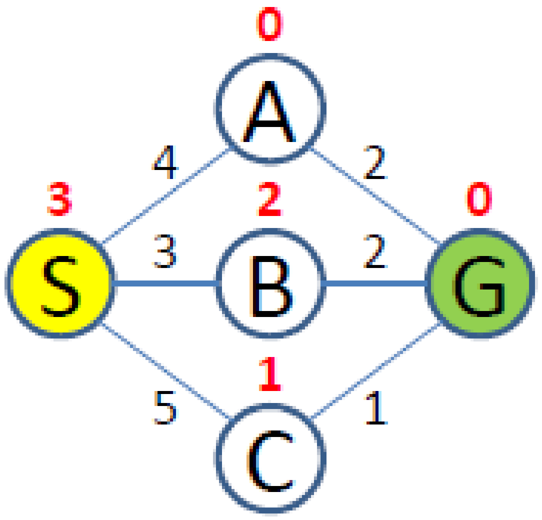


All children are explored (update)

Adjust f-values

SMA* By Example

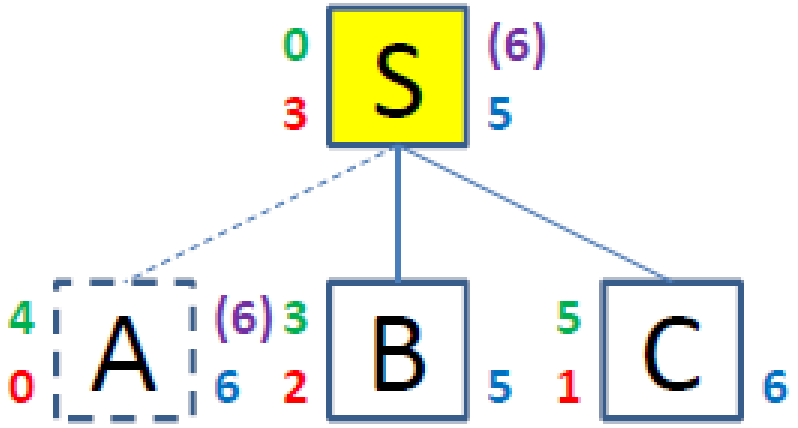


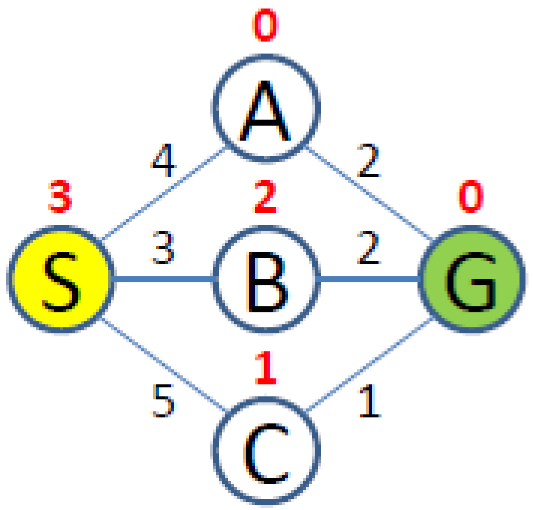


Generate children
(One by one)

Memory full

SMA* By Example

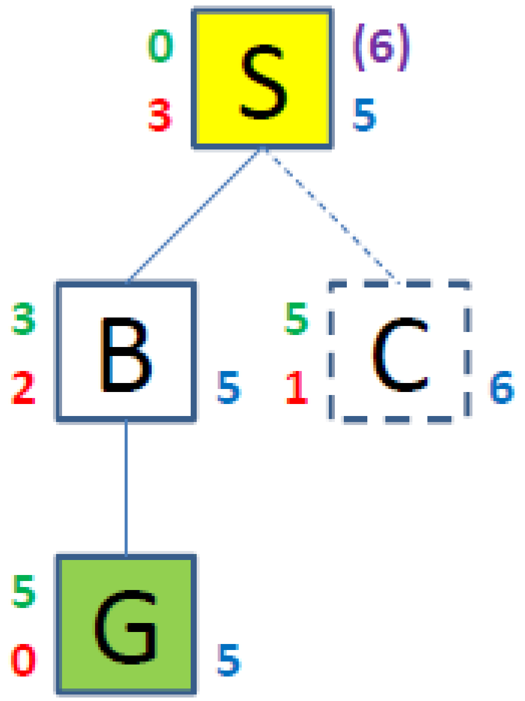


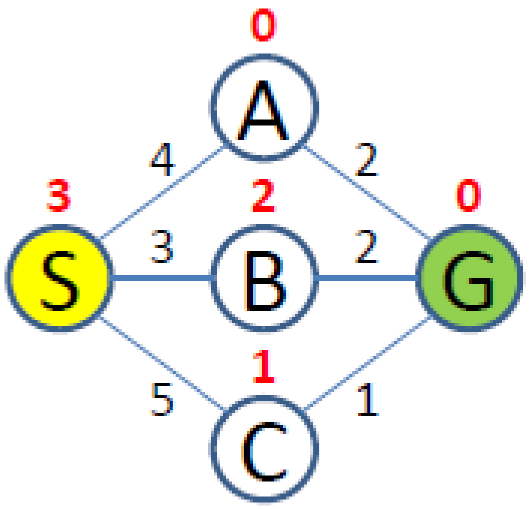


Generate children
(One by one)

Memory full

SMA* By Example





Generate children
(One by one)

Memory full

Confession

- ❑It is possible that some sentences or some information were included in these slides without mentioning exact references. I am sorry for violating rules of intellectual property. When I will have a bit more time, I will try my best to avoid such things.
- ❑These slides are only for students in order to give them very basic concepts about the giant, “Networking”, not for experts.
- ❑Since I am not a network expert, these slides could have wrong/inconsistent information…I am sorry for that.
- ❑Students are requested to check references and Books, or to talk to Network engineers.