Artificial Intelligence

Heuristic Searches

Is ant search for food is Heuristic *Searches??*

Heuristic Searches - Characteristics

- Has some domain knowledge
- Usually more efficient than blind searches
- Sometimes called an informed search
- Heuristic searches work by deciding which is the next best node to expand (there is no guarantee that it is the best node)

Heuristic Searches - Characteristics

- Heuristic searches estimate the cost to the goal from its current position. It is usual to denote the heuristic evaluation function by $h(n)$

- Compare this with something like Uniform Cost Search which chooses the lowest code node thus far ($g(n)$)

Heuristic Searches - Why Use?

- It may be too resource intensive (both time and space) to use a blind search
- Even if a blind search will work we may want a more efficient search method

Heuristic Searches - Implementation - 1

- Implementation is achieved by sorting the nodes based on the evaluation function; $h(n)$

Heuristic Search Methods

Methods that use a heuristic function to provide specific knowledge about the problem:

Heuristic Functions

Hill climbing

Beam search

Hill climbing (2)

Greedy search

Heuristic Functions

- To further improve the quality of the previous methods, we need to include problem-specific knowledge on the problem.

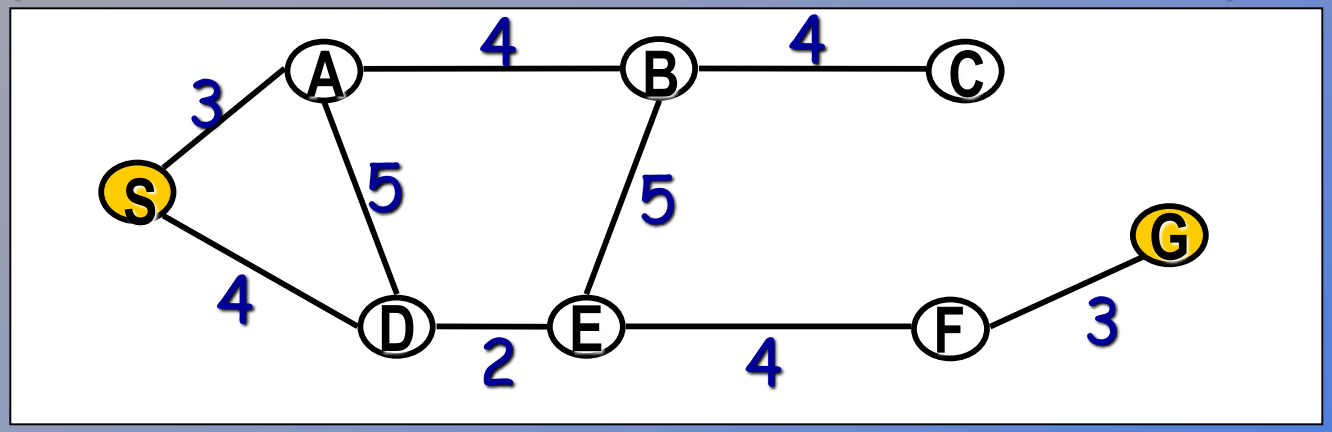
  ➔ How to do this in such a way that the algorithms remain generally applicable ???

➔HEURISTIC FUNCTIONS:

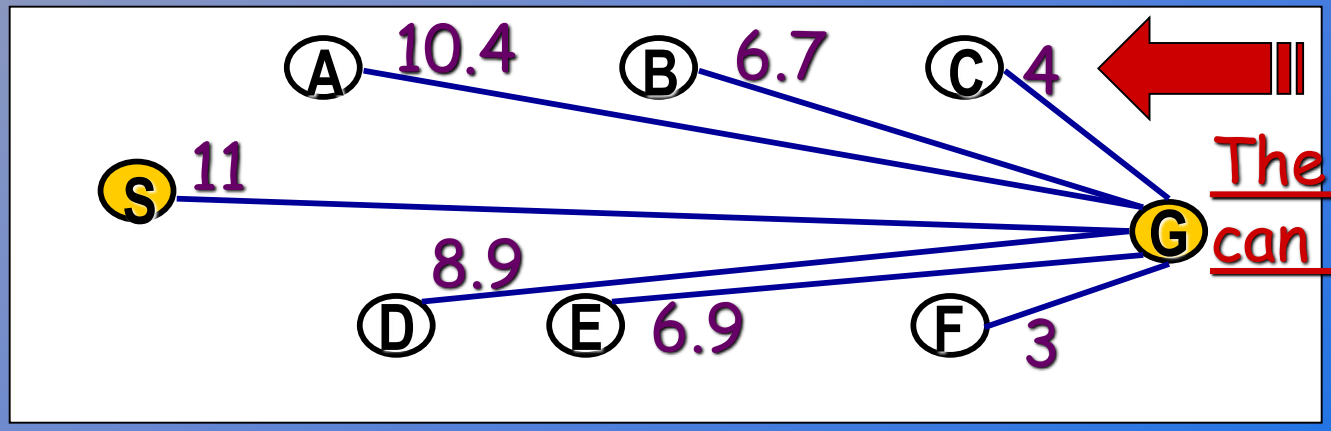
- f: States --> Numbers
- f(T) : expresses the quality of the state T
  - allow to express problem-specific knowledge, <u>BUT</u>: can be imported in a generic way in the algorithms.

Examples (1):

- Imagine the problem of finding a route on a road map and that the NET below is the road map:



- Define f(T) = the straight-line distance from T to G



The estimate can be wrong!

Examples (2): 8-puzzle

- f1(T) = the number correctly placed tiles on the board:

f1 (| 1 | 3 | 2 |
| 8 | | 4 |
| 5 | 6 | 7 |) = 4

- f2(T) = number or incorrectly placed tiles on board:
  - ➔gives (rough!) estimate of how far we are from goal

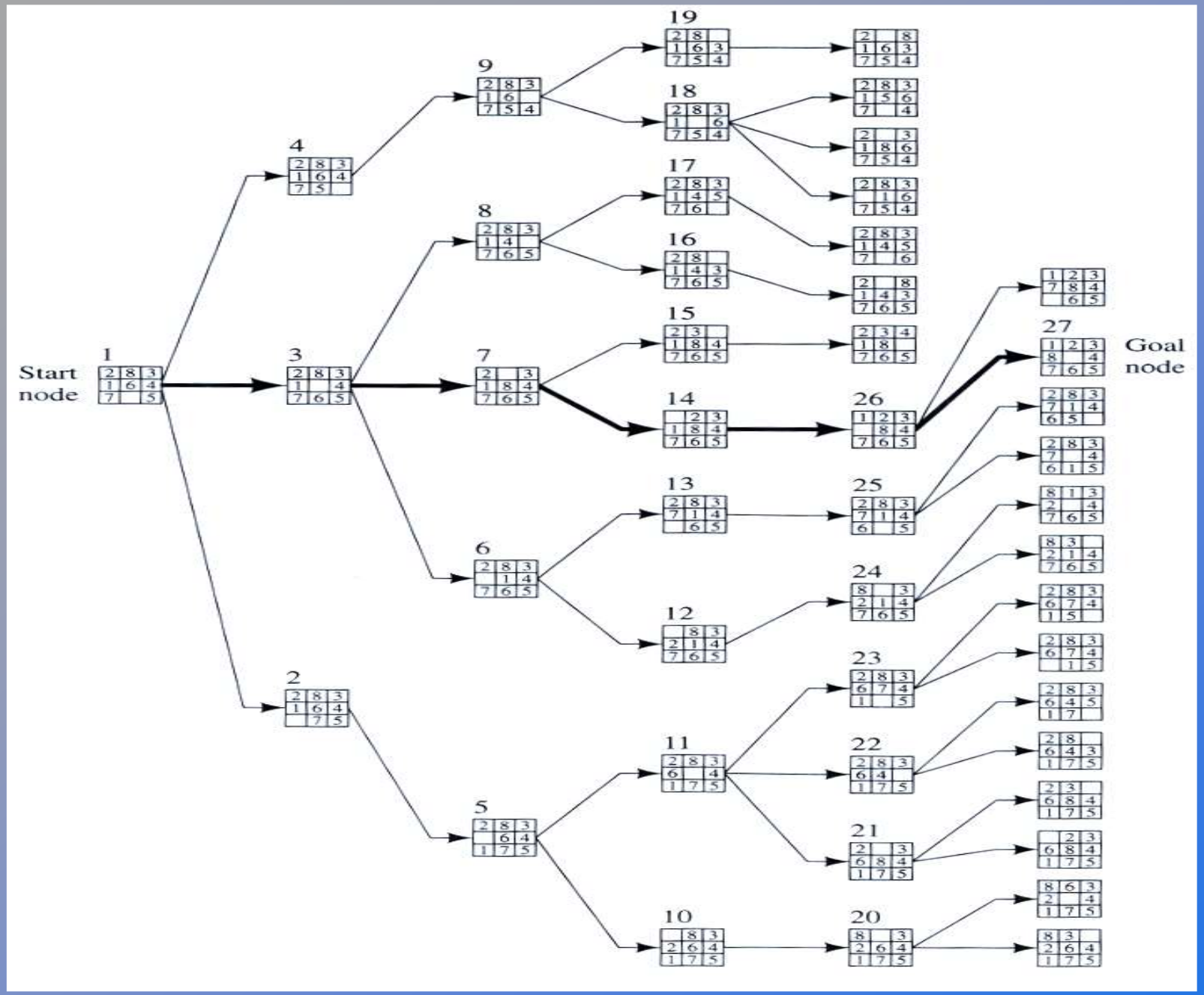
f2 (| 1 | 3 | 2 |
| 8 | | 4 |
| 5 | 6 | 7 |) = 4

Most often, 'distance to goal' heuristics are more useful !

Examples (3): Manhattan distance

- f3(T) = the sum of (the horizontal + vertical distance that each tile is away from its final destination):
  - ➔gives a better estimate of distance from the goal node

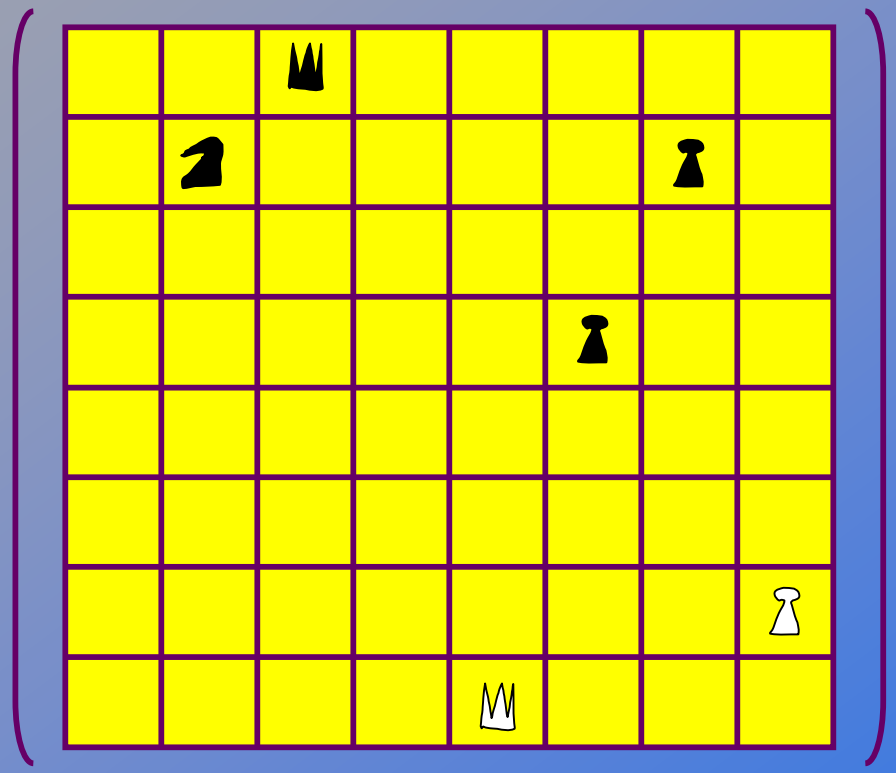
f3 (1 5 2 / 8 4 / 3 6 7) = 1 + 4 + 2 + 3 = 10


Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Examples (4): Chess:

- F(T) = (Value count of black pieces) - (Value count of white pieces)

f



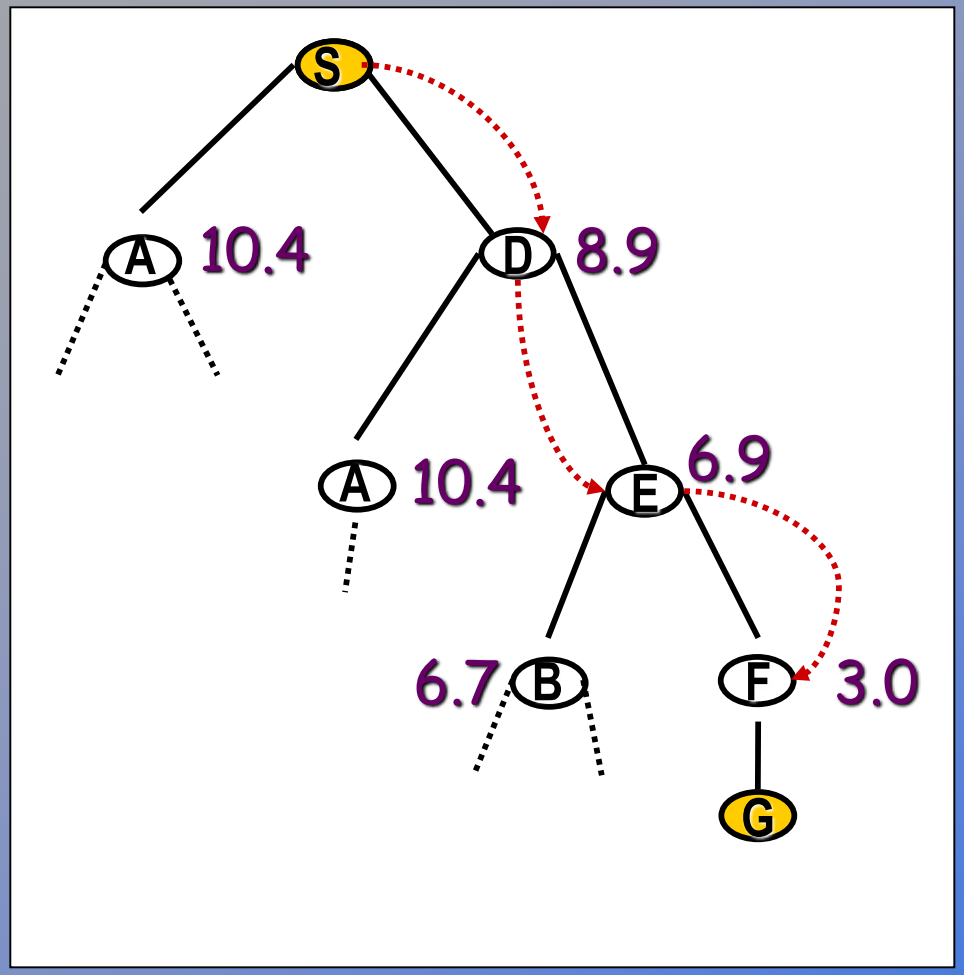
= v(♛) + v(♞)
+ v(♜) + v(♜)
- v(♕) - v(♙)

Hill climbing

A basic heuristic search method:
depth-first + heuristic

Hill climbing_1

- Example: using the straight-line distance:



- Perform depth-first, BUT:
- instead of left-to-right selection,
- FIRST select the child with the best heuristic value

Hill climbing_1 algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
AND goal is not reached

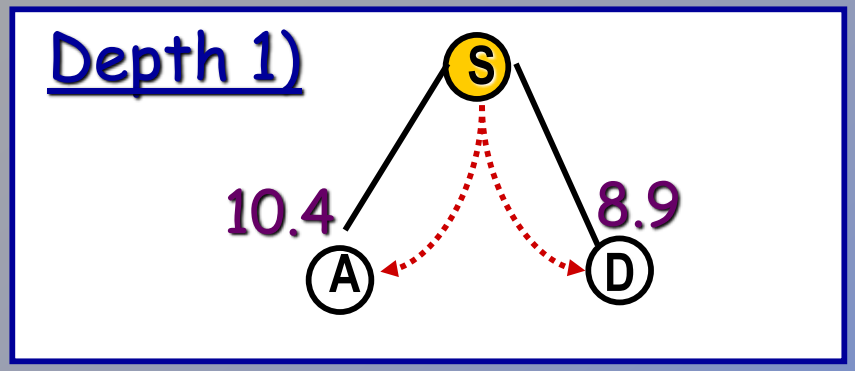
DO { remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
sort new paths (HEURISTIC) ;
add the new paths to front of QUEUE;

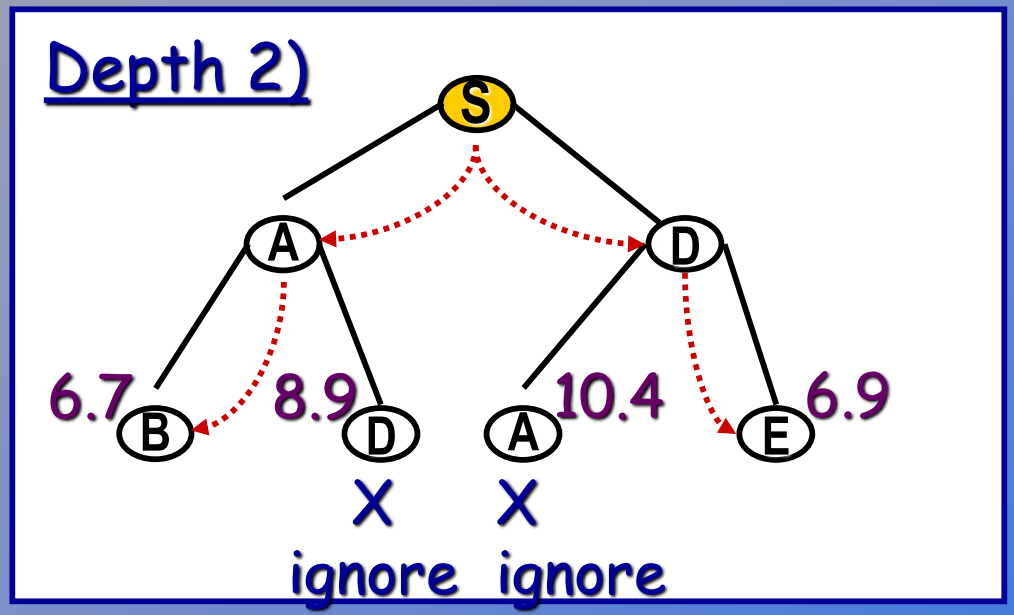
3. IF goal reached
THEN success;
ELSE failure;

Beam search

Narrowing the width of the breadth-first search

Beam search (1):



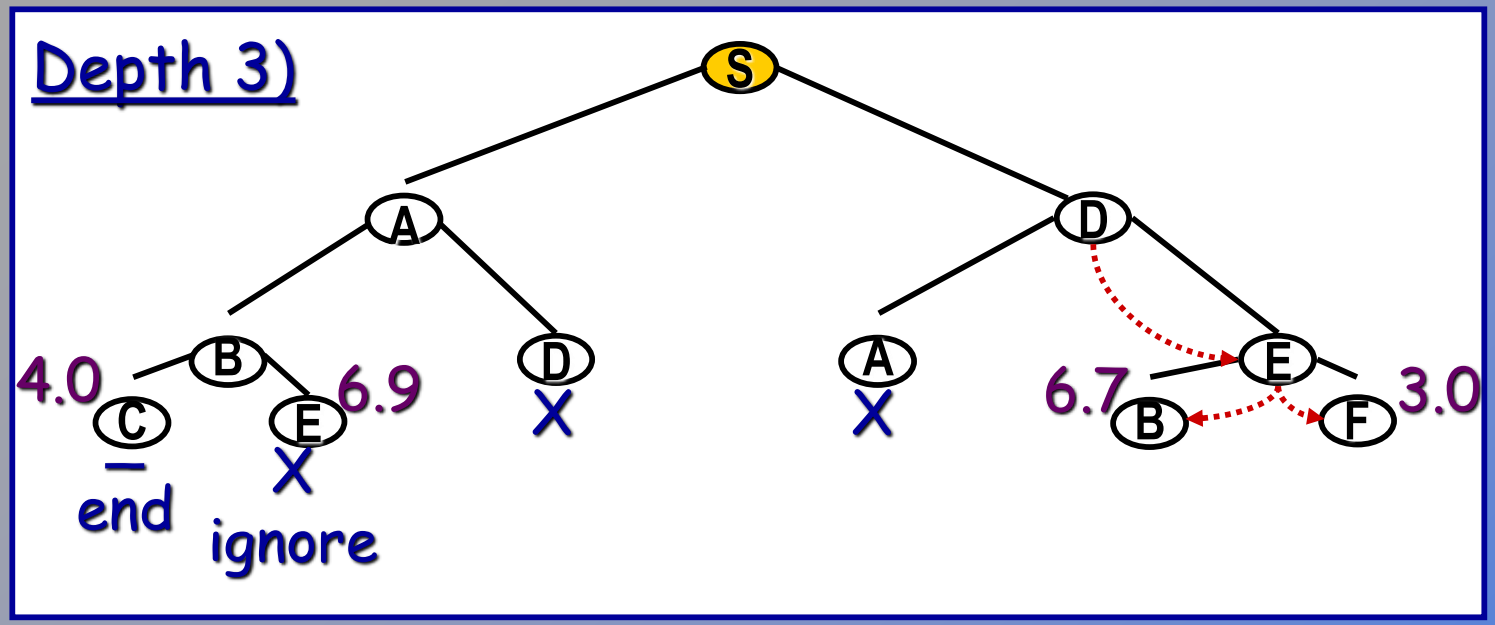


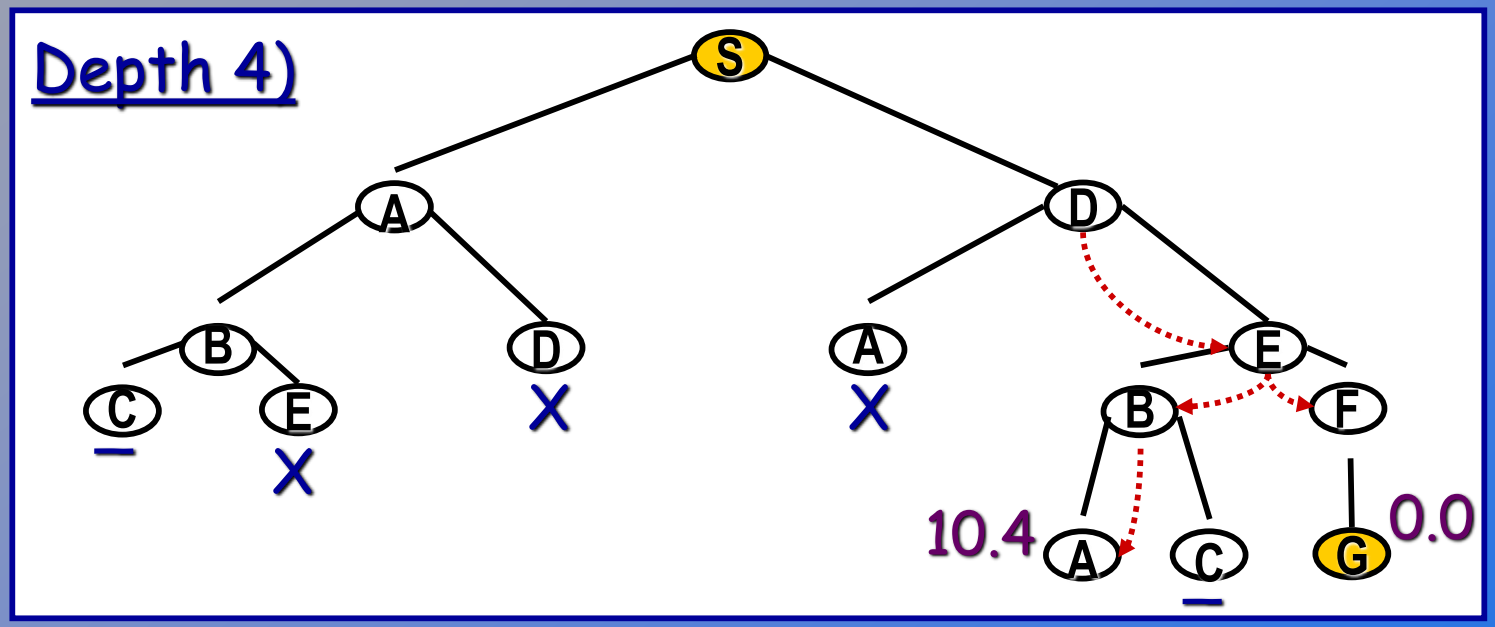
- Assume a pre-fixed WIDTH (example : 2)
- Perform breadth-first, BUT:
- Only keep the WIDTH best new nodes

  depending on heuristic

  at each new level.

Beam search (2):





- Optimi-zation: ignore leafs that are not goal nodes (see C)

Beam search algorithm:

- See exercises!

Properties:

- Completeness:
  - Hill climbing: YES (backtracking), Beam search: NO
- Speed/Memory:
  - Hill climbing:
    - same as Depth-first (in worst case)
  - Beam search:
    - QUEUE always has length WIDTH, so memory usage is constant = WIDTH, time is of the order of WIDTH*m*b or WIDTH*d*b if no solution is found

Hill climbing_2

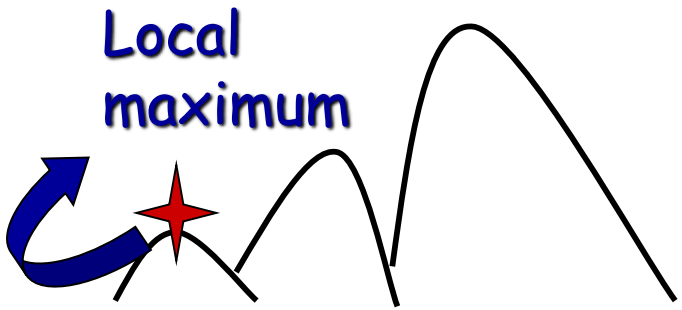
- == Beam search with a width of 1.

- Inspiring Example: climbing a hill in the fog.
  - ➔Heuristic function: check the change in altitude in 4 directions: the strongest increase is the direction in which to move next.

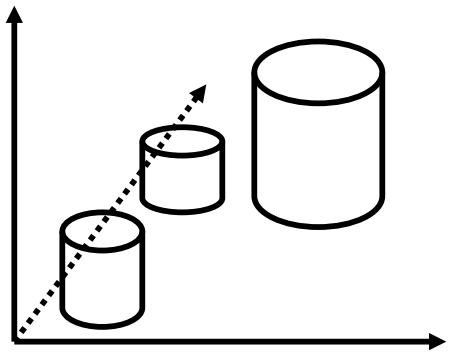
- Is identical to Hill climbing_1, except for dropping the backtracking.
- Produces a number of classical problems:

Problems with Hill climbing_2:

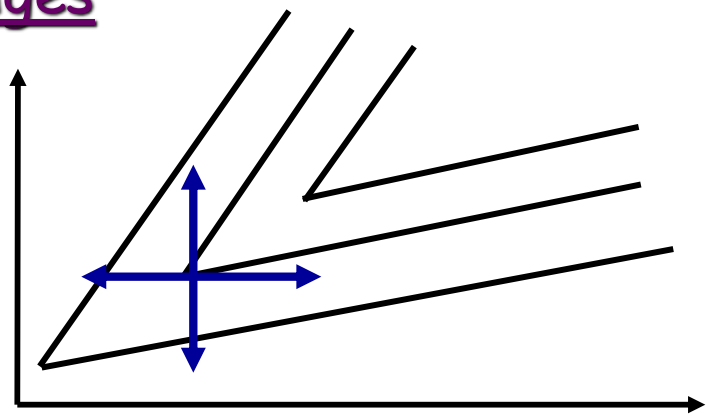
Foothills:



Plateaus



Ridges



Comments:

- Foothills are local minima: hill climbing_2 can't detect the difference.
- Plateaus don't allow you to progress in any direction.

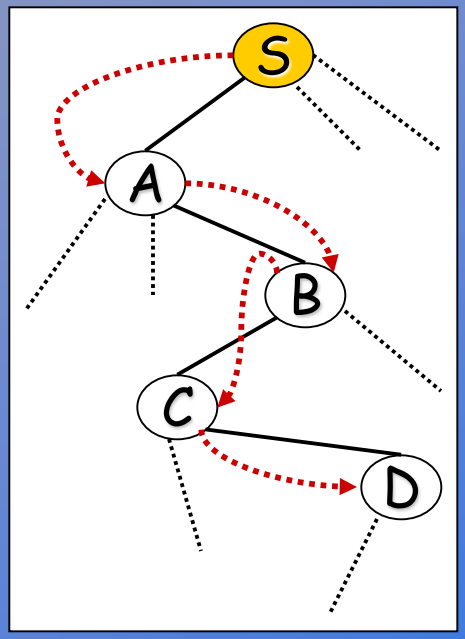
  ➔Foothills and plateaus require random jumps to be combined with the hill climbing algorithm.

- Ridges neither: the directions you have fixed in advance all move downwards for this surface.

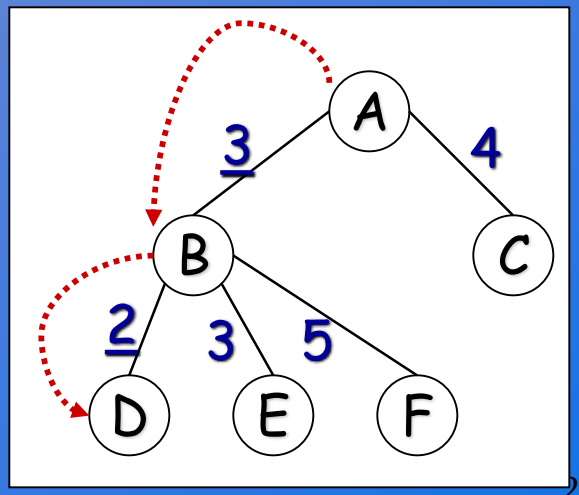
  ➔Ridges require new rules, more directly targeted to the goal, to be introduced (new directions to move) .

Local search

- Hill climbing_2 is an example of local search.
- In local search, we only keep track of 1 path and use it to compute a new path in the next step.
  - ➔ QUEUE is always of the form:
    - ◆ (p)



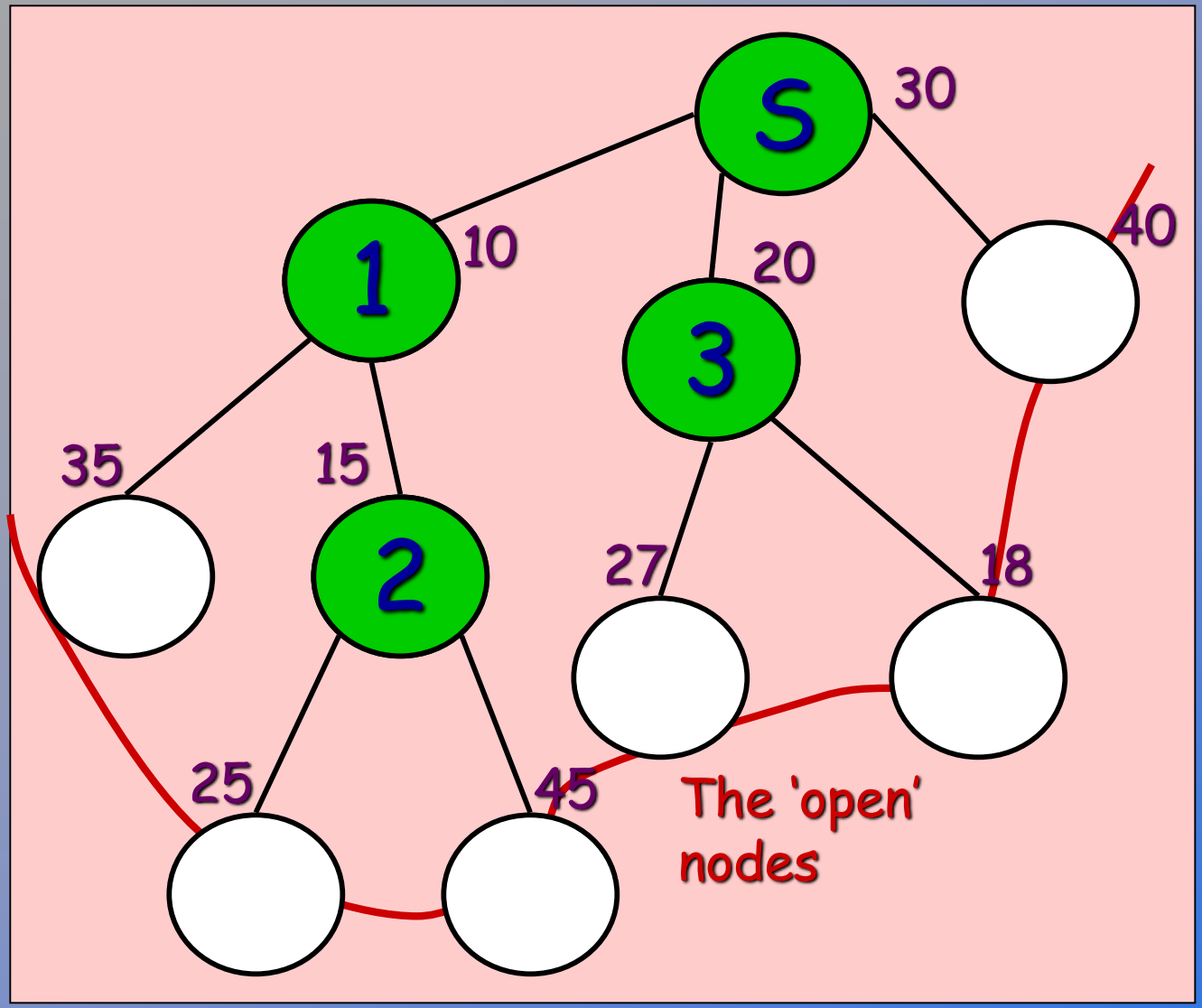
- Another example:
  - ➔ MINIMAL COST search:
- If p is the current path:
  - ➔ the next path extends p by adding the node with the smallest cost from the endpoint of p



Greedy search

Always expand the heuristically best nodes first.

Greedy search, or Heuristic best-first search:



- At each step, select the node with the best (in this case: lowest) heuristic value.

Greedy search algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
AND goal is not reached

DO { remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths and sort the entire QUEUE;

(HEURISTIC)

3. IF goal reached
THEN success;
ELSE failure;

OPTIMAL Search

When cost of TRAVERSING the path should be minimized (even at expense of more complicated SEARCHING) :

Uniform Cost

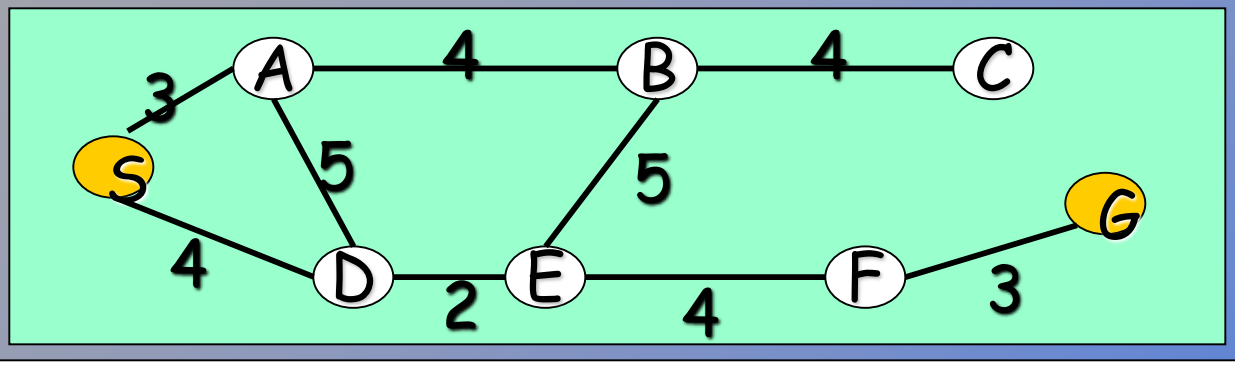
Branch and Bound

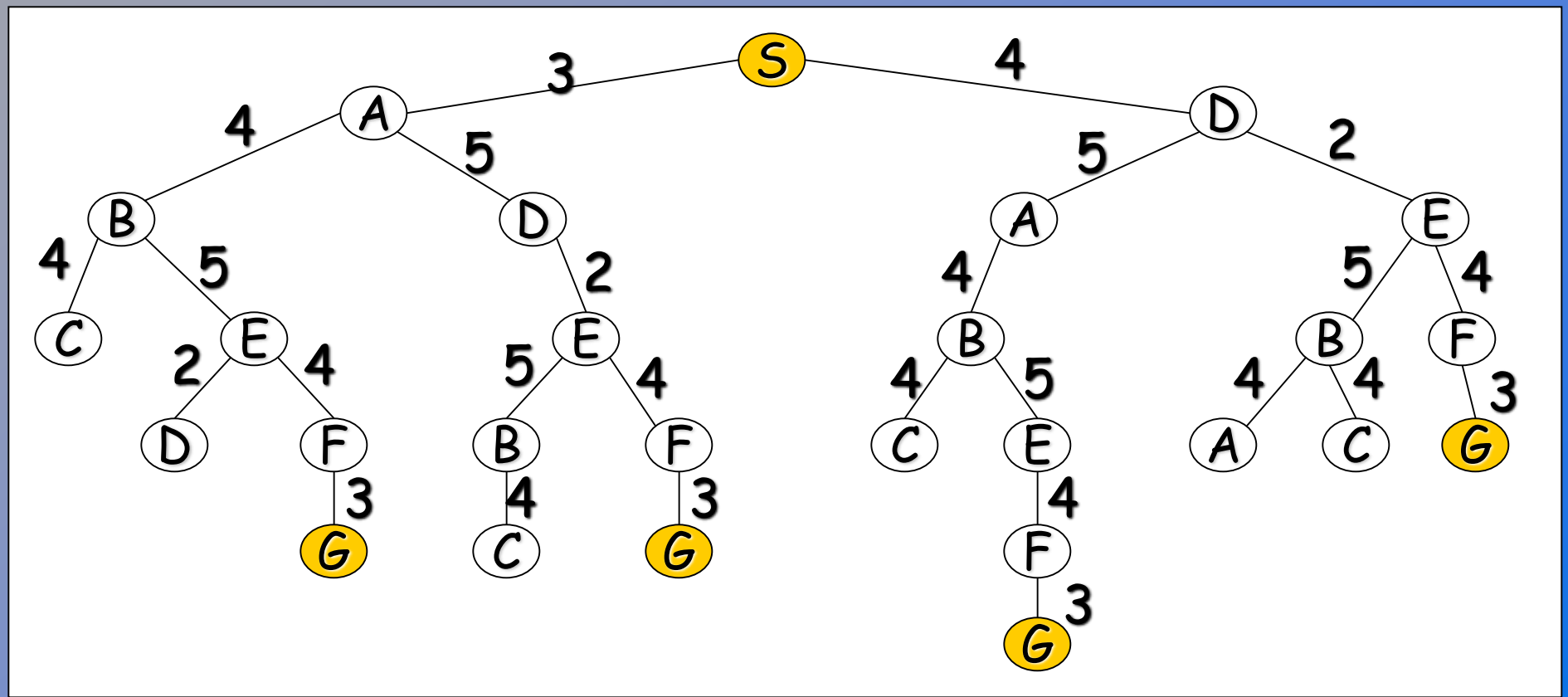
Introducing Underestimates

Path Deletion

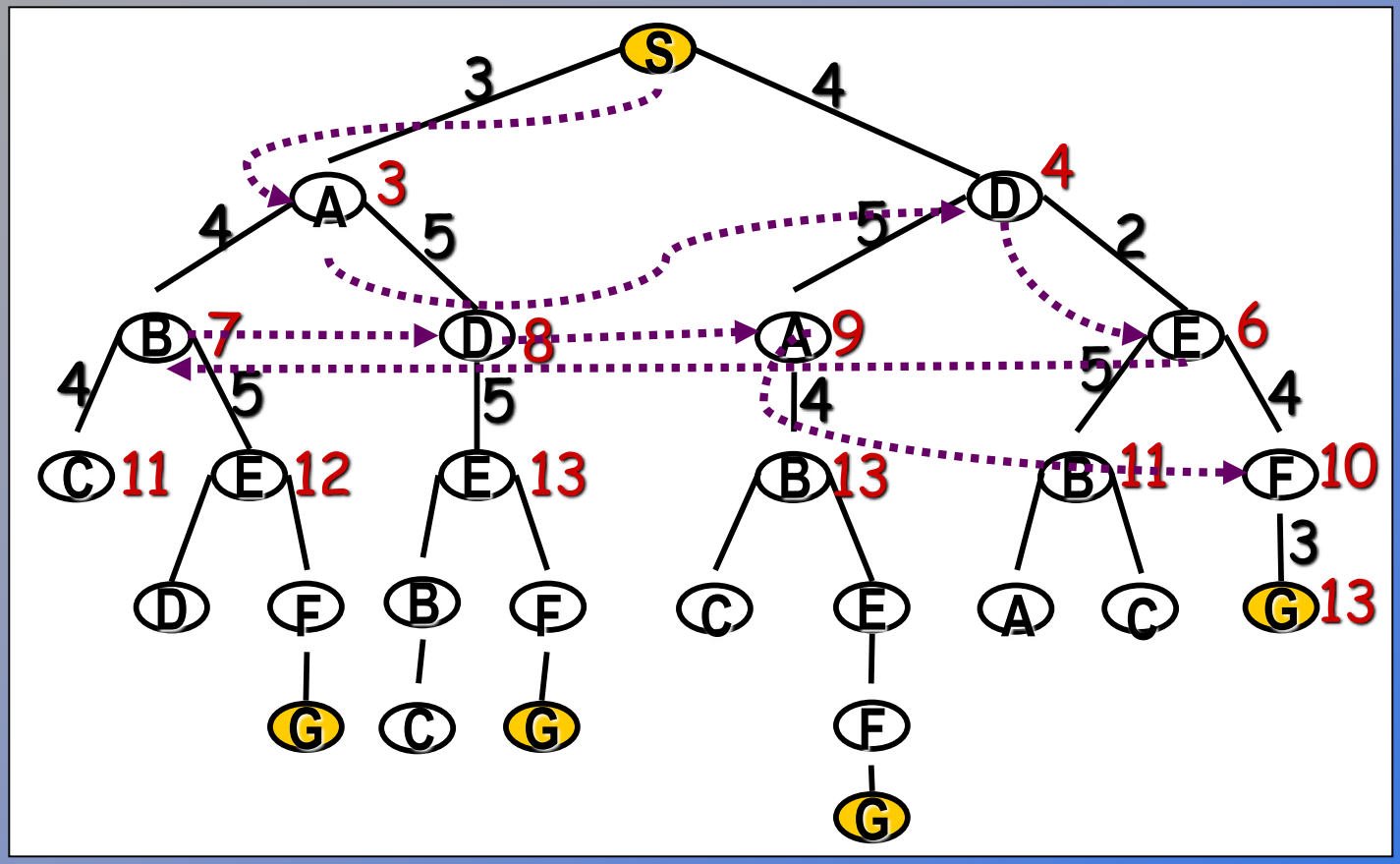
A*

Re-introduce the costs of arcs in the NET





Uniform cost algorithm = uniformed best-first



- At each step, select the node with the lowest accumul-ated cost.

Uniform cost algorithm:

1. **QUEUE** <-- path only containing the root;

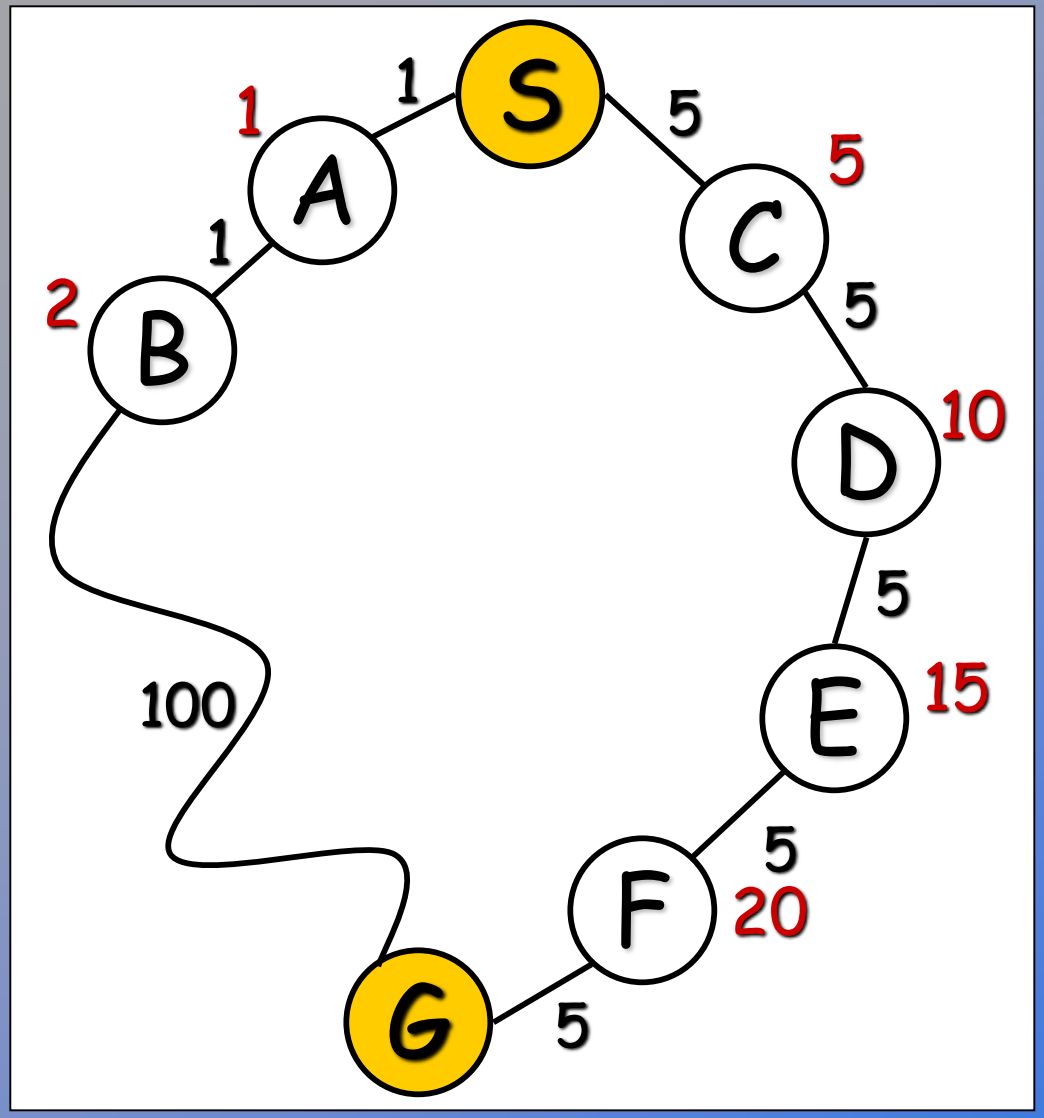
2. WHILE { **QUEUE** is not empty
   AND goal is not reached

   DO { remove the first path from the **QUEUE**;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths and sort the entire **QUEUE**;

   (BY ACCUMULATED COST)

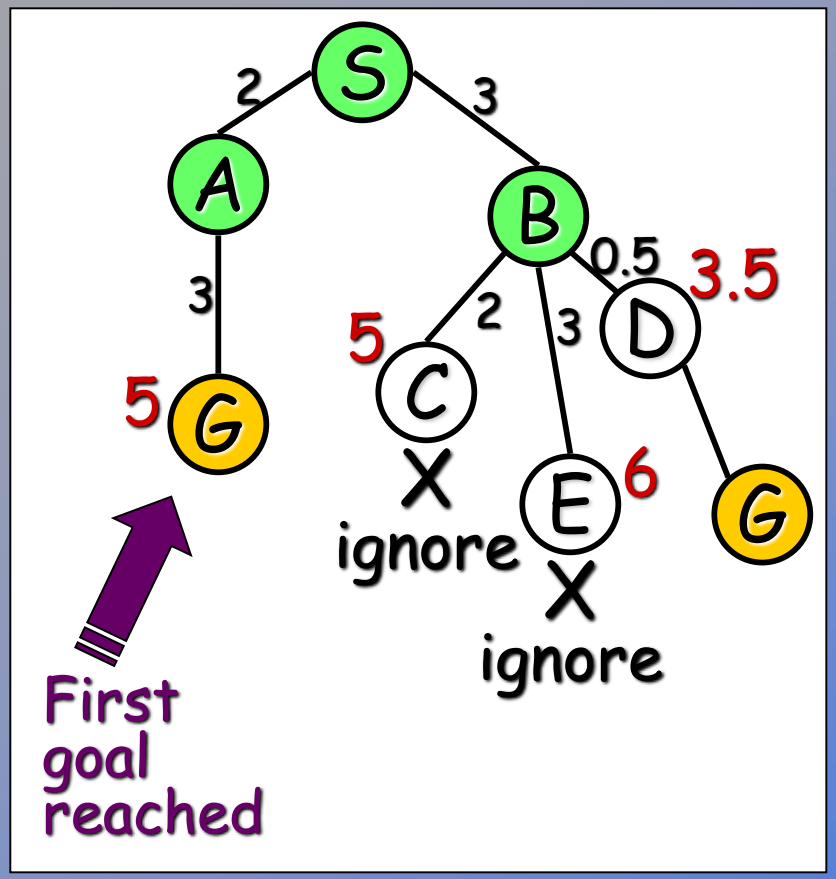
3. IF goal reached
   THEN success;
   ELSE failure;

Problem: NOT always optimal!



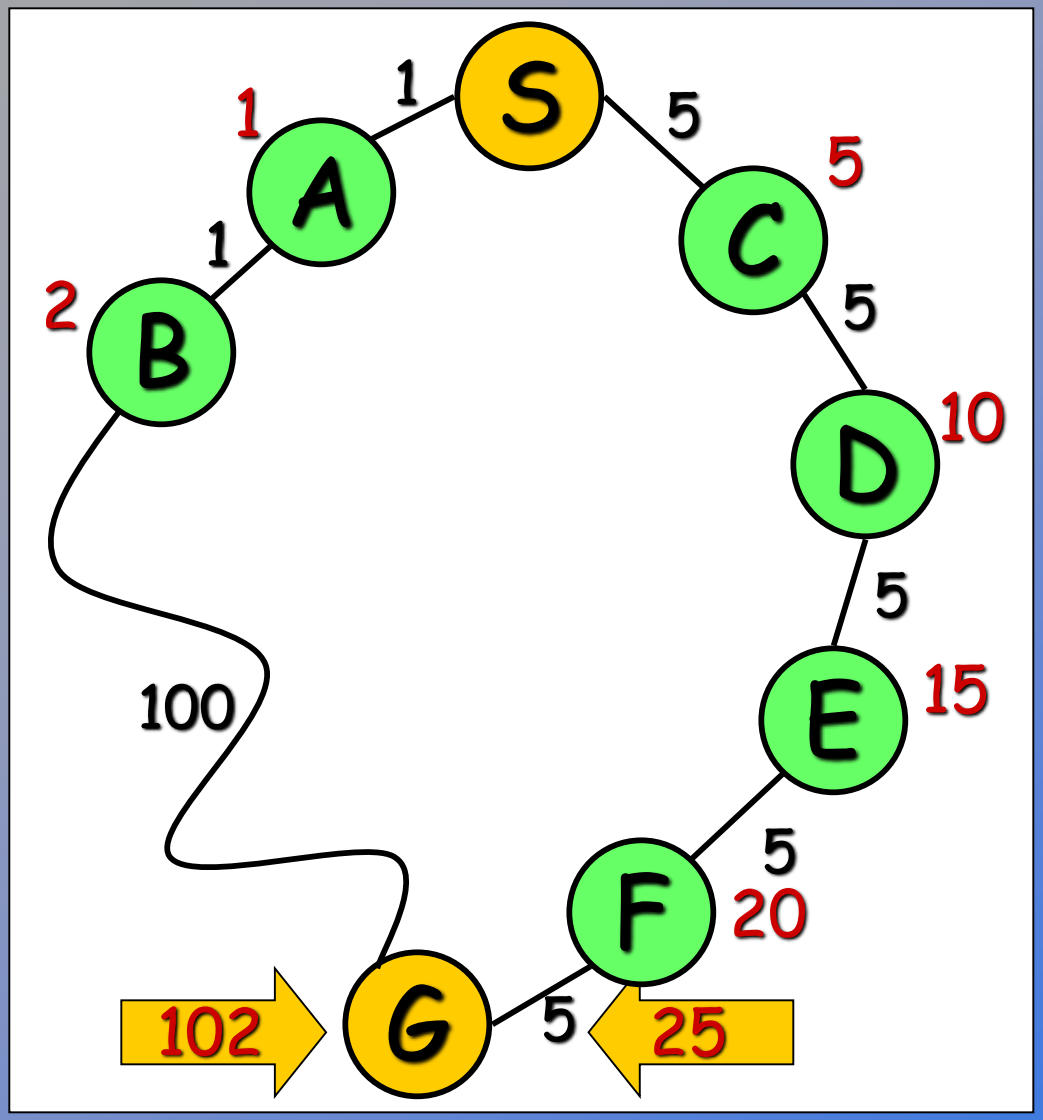
- Uniform cost returns the path with cost 102, while there is a path with cost 25.

The Branch-and-Bound principle



- Use any (complete) search method to find a path.
- Remove all partial paths that have an accumulated cost larger or equal than the found path.
- Continue search for the next path.
- Iterate.

A weak integration of branch and bound in uniform cost:



- Change the termination condition:
  - ➔ only terminate when a path to a goal node HAS BECOME THE BEST PATH.

Optimal Uniform cost version:

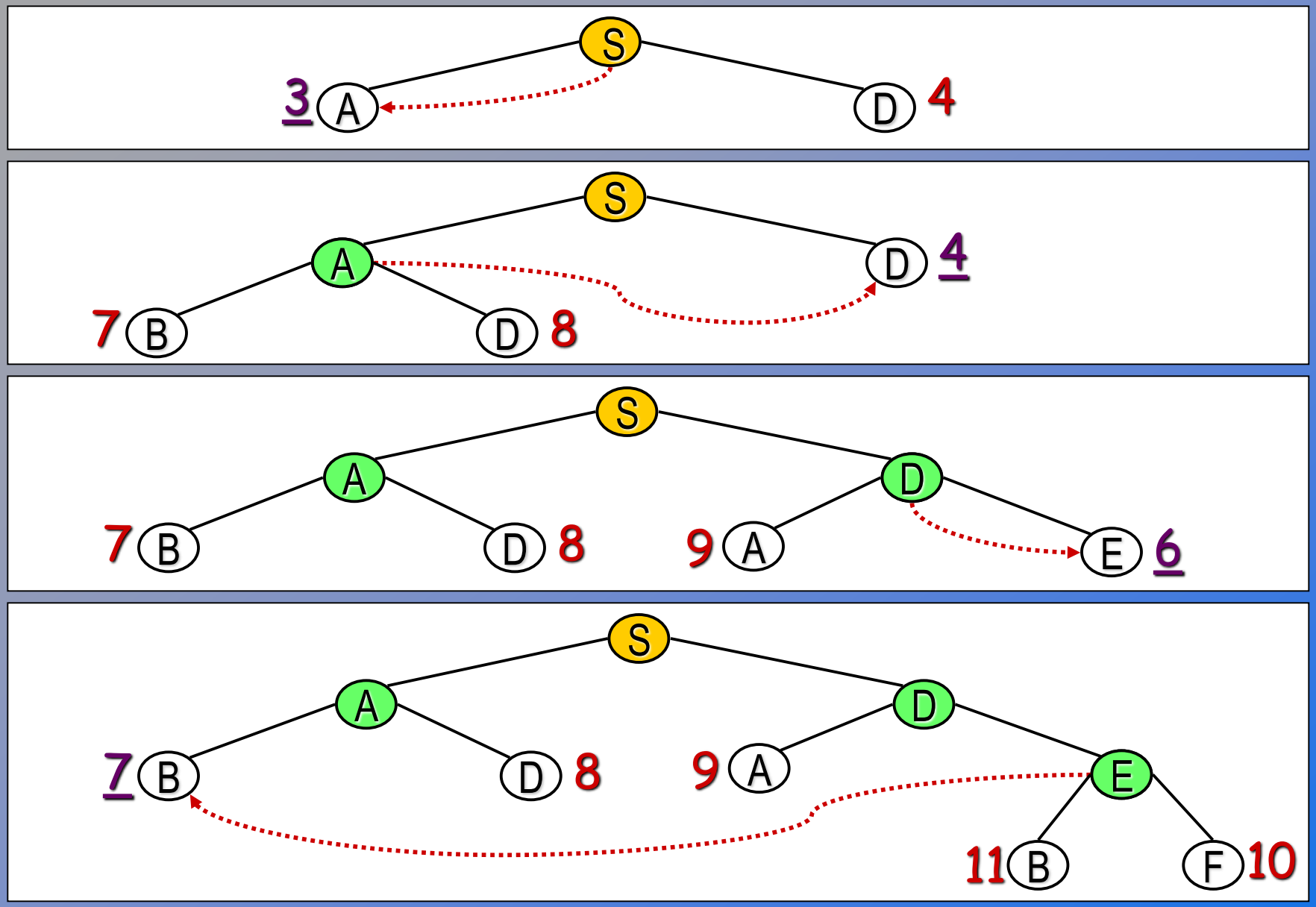
1. QUEUE <-- path only containing the root;

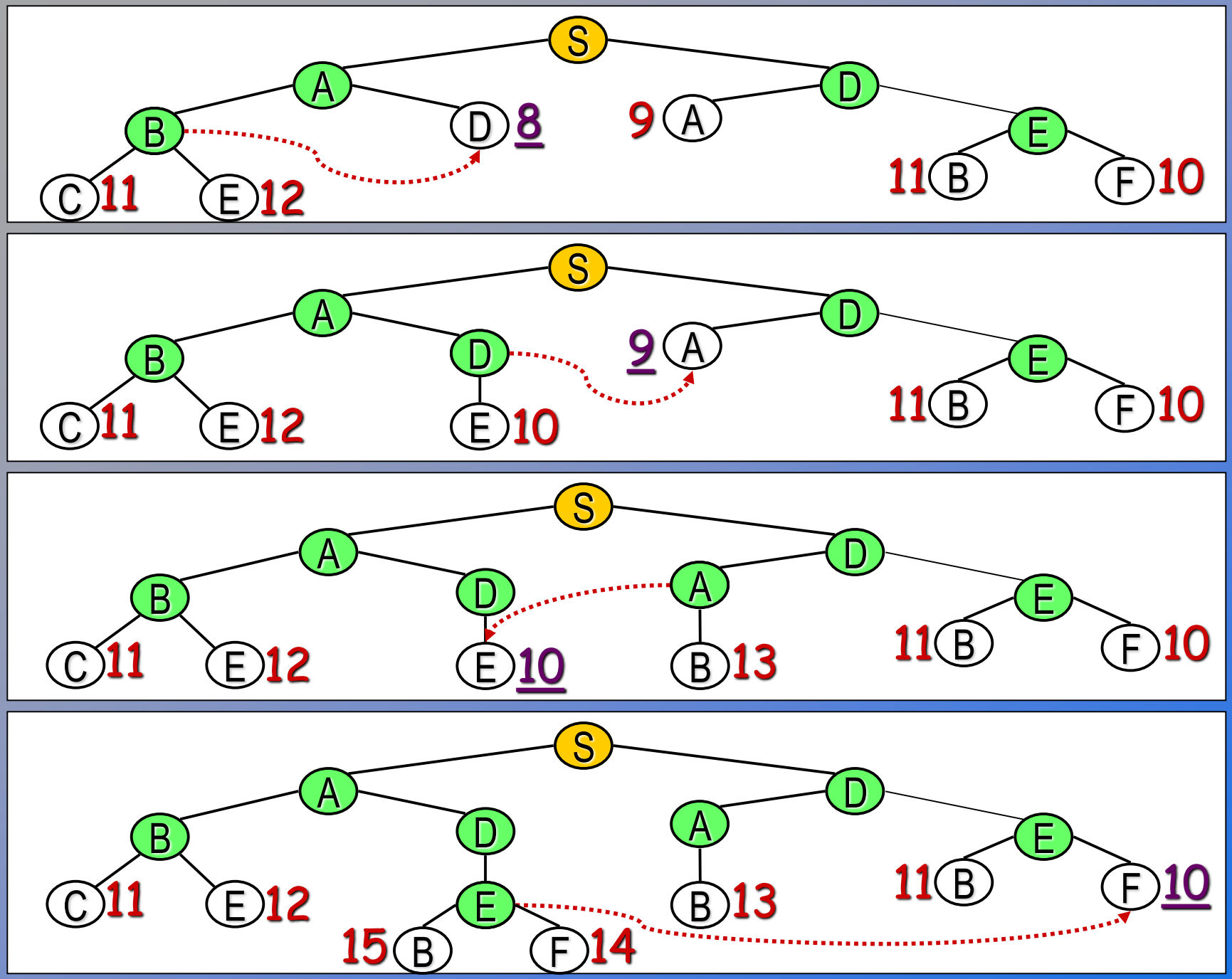
2. WHILE { QUEUE is not empty
   AND first path does not reach goal

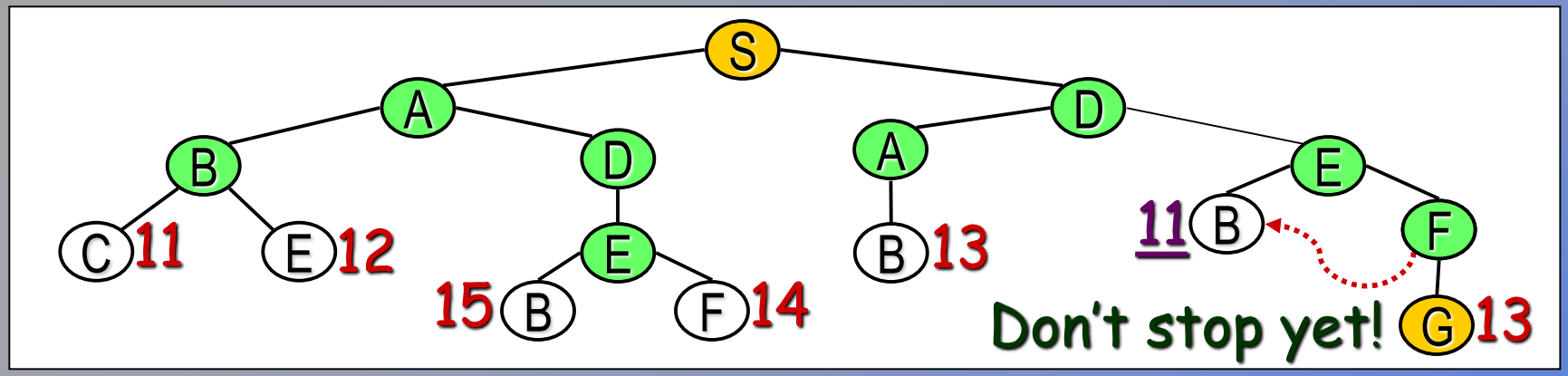
   { remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths and sort the entire QUEUE;
   (by accumulated cost)

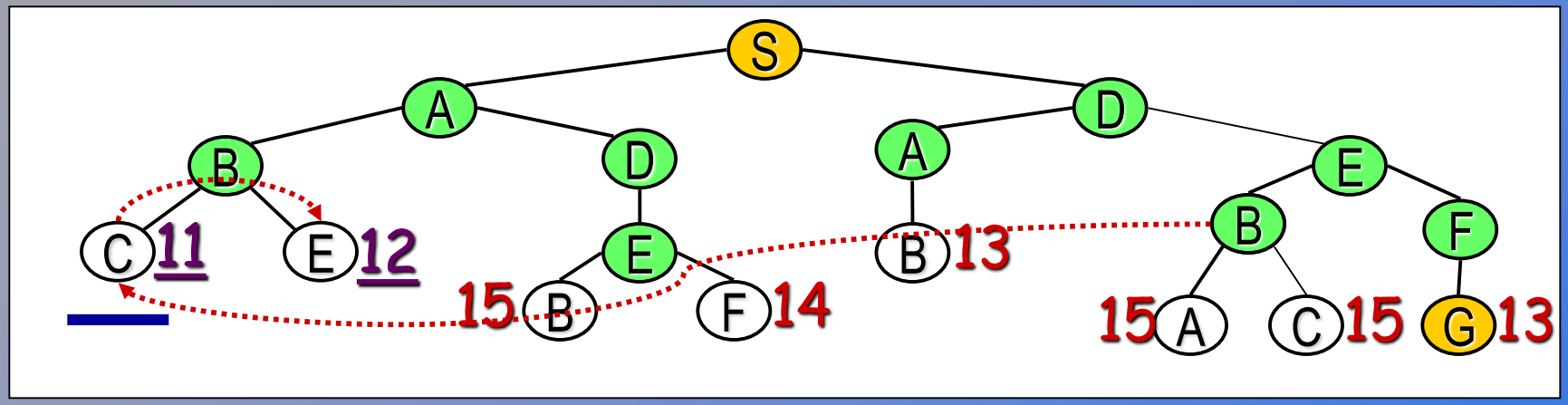
3. IF goal reached
   THEN success;
   ELSE failure;

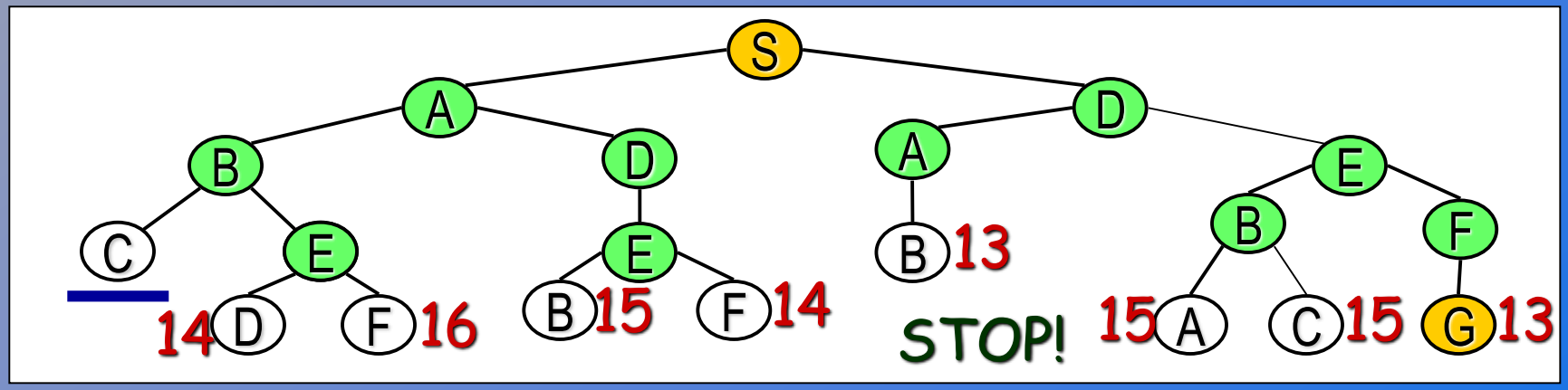
Example:




S
A
D
B
D 8
9 A
E
C 11
E 12
11 B
F 10
S
A
D
B
D
9 A
E
C 11
E 12
E 10
11 B
F 10
S
A
D
B
D
A
E
C 11
E 12
E 10
B 13
11 B
F 10
S
A
D
B
D
A
E
C 11
E 12
E
B 13
11 B
F 10
15 B
F 14


S
A
D
B
D
A
E
C 11
E 12
E
B 13
11 B
F
15 B
F 14
Don't stop yet!
G 13


S
A
D
B
D
A
E
C 11
E 12
E
B 13
B
F
15 B
F 14
15 A
C 15
G 13


S
A
D
B
D
A
E
C
E
E
B 13
B
F
14 D
F 16
B 15
F 14
STOP!
15 A
C 15
G 13

Properties of extended uniform cost :

- Optimal path:

  If there exists a number $\delta > 0$, such that every arc has **cost** $\geq \delta$, and if the branching factor is finite, **Then** extended uniform cost finds the optimal path (if one exists).

- Memory and speed:
  - ➔ In the worst case, at least as bad as for breadth-first:
    - ◆ needs additional sorting step after each path-expansion !
- How to improve ?

Extension with heuristic estimates:

- Replace the 'accumulated cost' in the 'extended uniform cost' algorithm by a function:

$$f(path) = cost(path) + h(endpoint_path)$$

- where:

cost(path) = the accumulated cost of the partial path

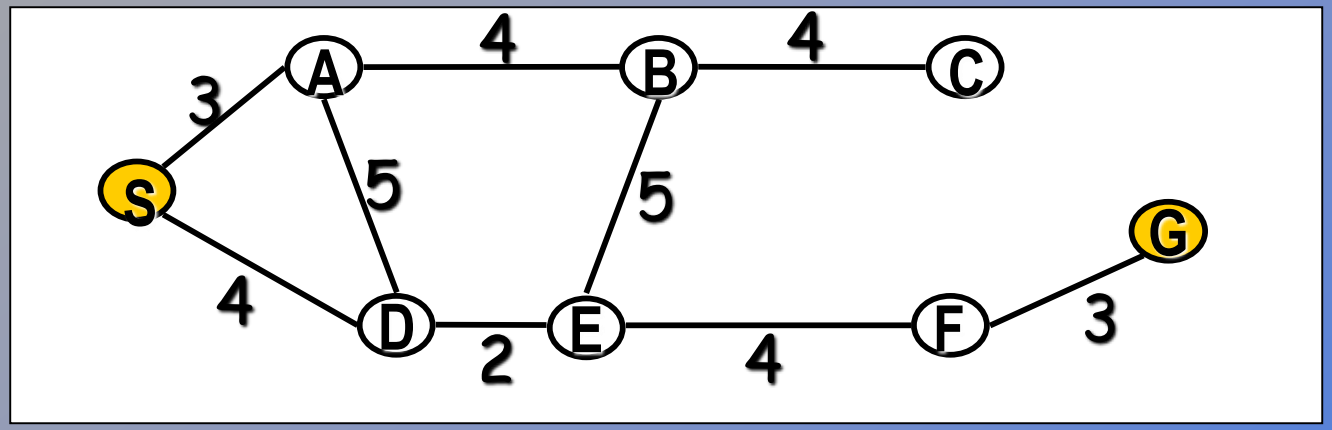
h(T) = a heuristic estimate of the cost remaining from T to a goal node

+

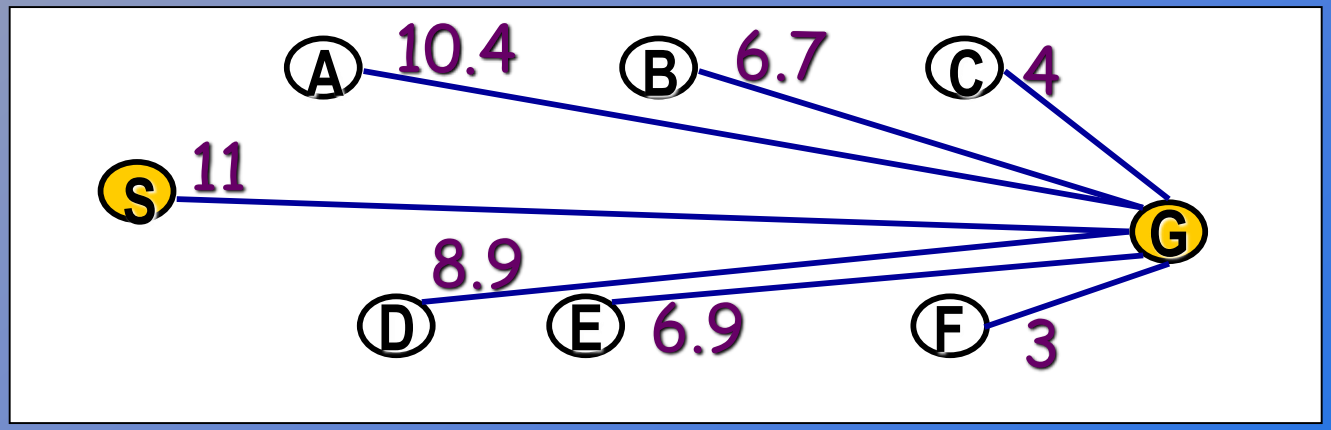
---

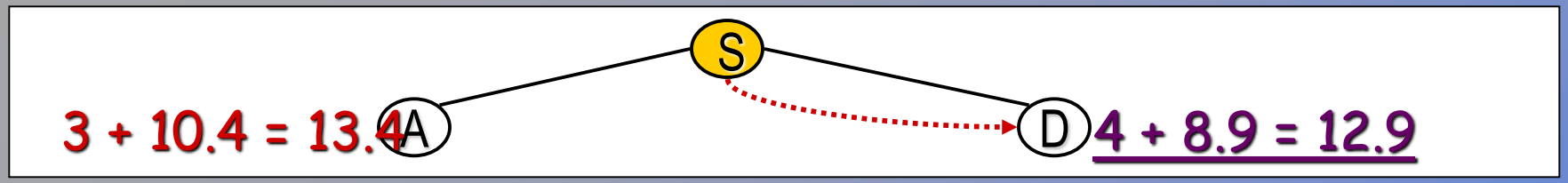
f(path) = an estimate of the cost of a path extending the current path to reach a goal.

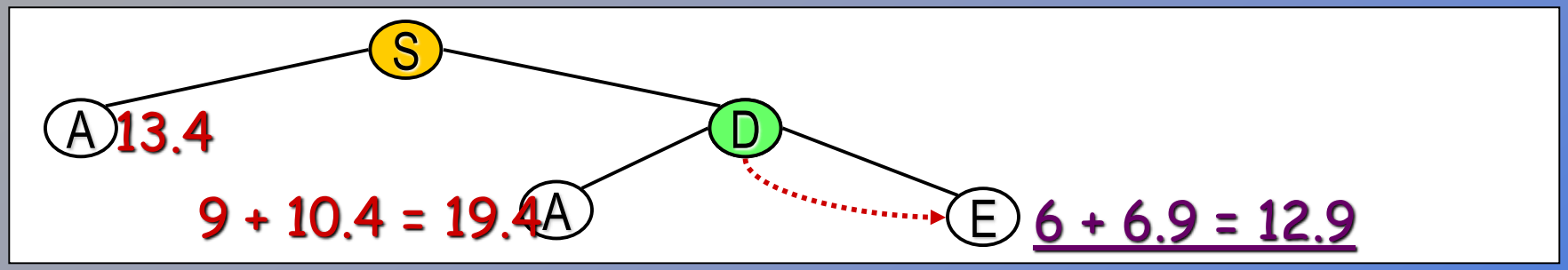
Example: Reconsider the straight-line distance:

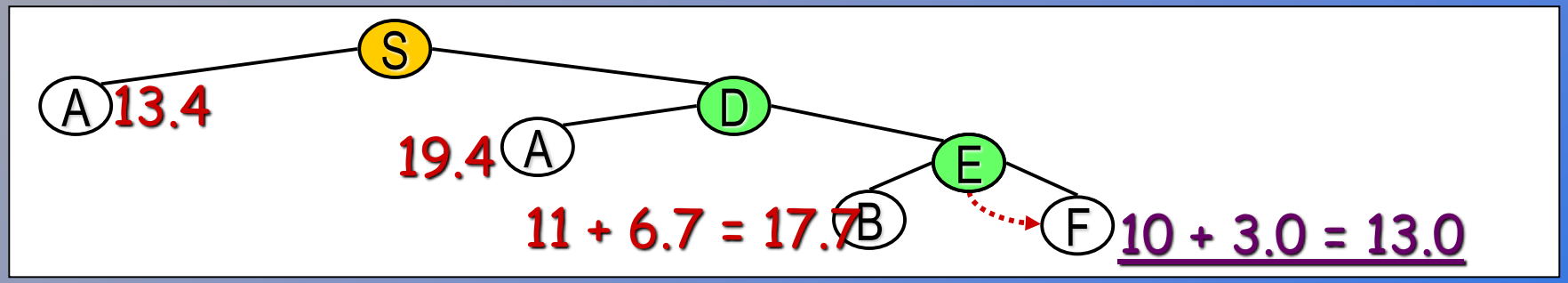


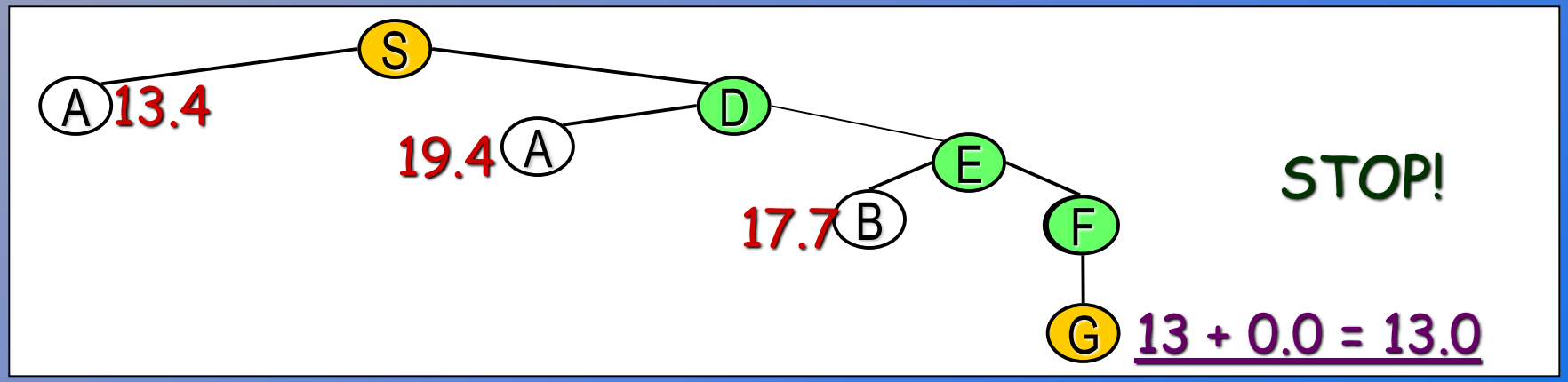
- h(T) = the straight-line distance from T to G




S
A
D
3 + 10.4 = 13.4
4 + 8.9 = 12.9


S
A
13.4
D
9 + 10.4 = 19.4
A
E
6 + 6.9 = 12.9


S
A
13.4
D
19.4
A
E
11 + 6.7 = 17.7
B
F
10 + 3.0 = 13.0


S
A
13.4
D
19.4
A
E
17.7
B
F
G
13 + 0.0 = 13.0
STOP!

Estimate-extended Uniform cost algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
   AND first path does not reach goal

   { remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths and sort the entire QUEUE;
   (by f = cost + h)

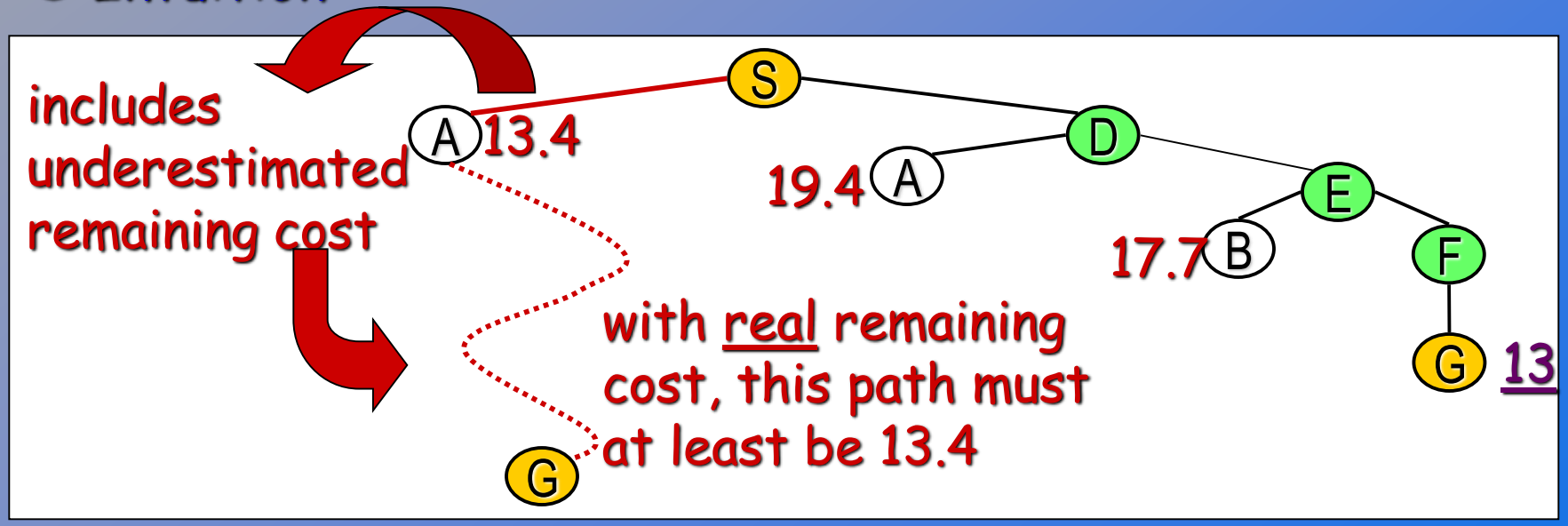
3. IF goal reached
   THEN success;
   ELSE failure;

Optimality

- With the same condition on the arcs-costs and on the branching factor:

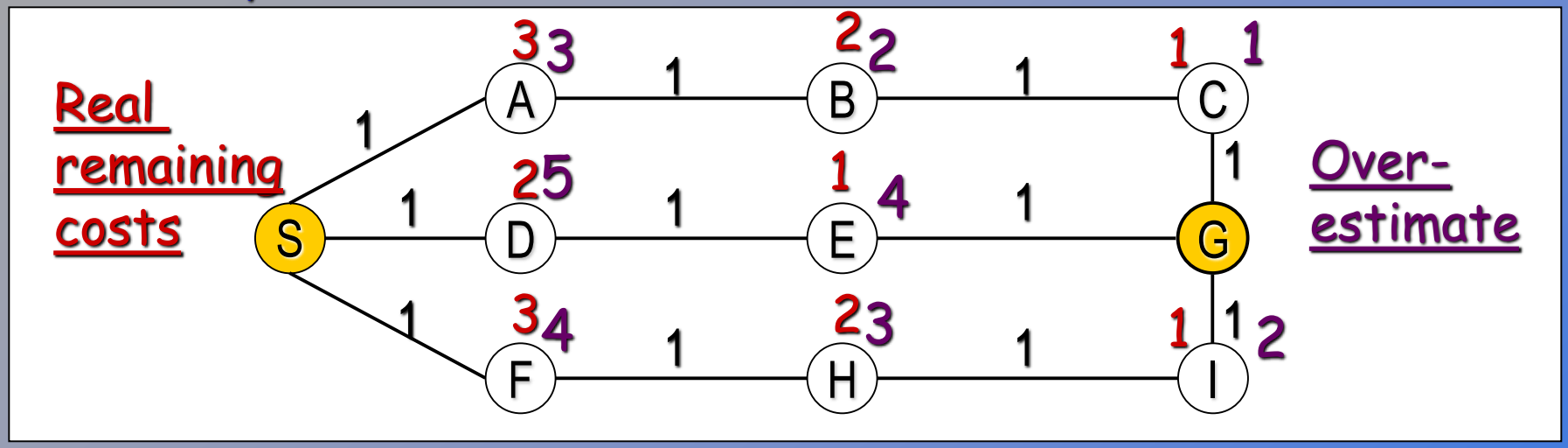
> IF for all T: h(T) is an UNDERestimate of the remaining cost to a goal node
>
> THEN estimate-extended uniform cost is optimal.

- Intuition:

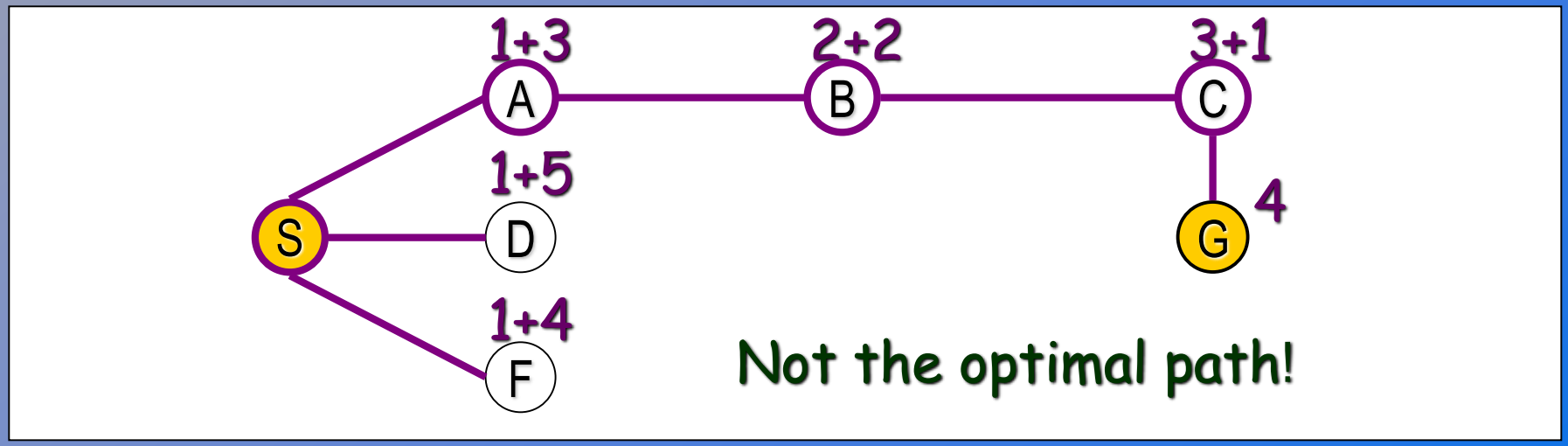


More on underestimates:

- Example:

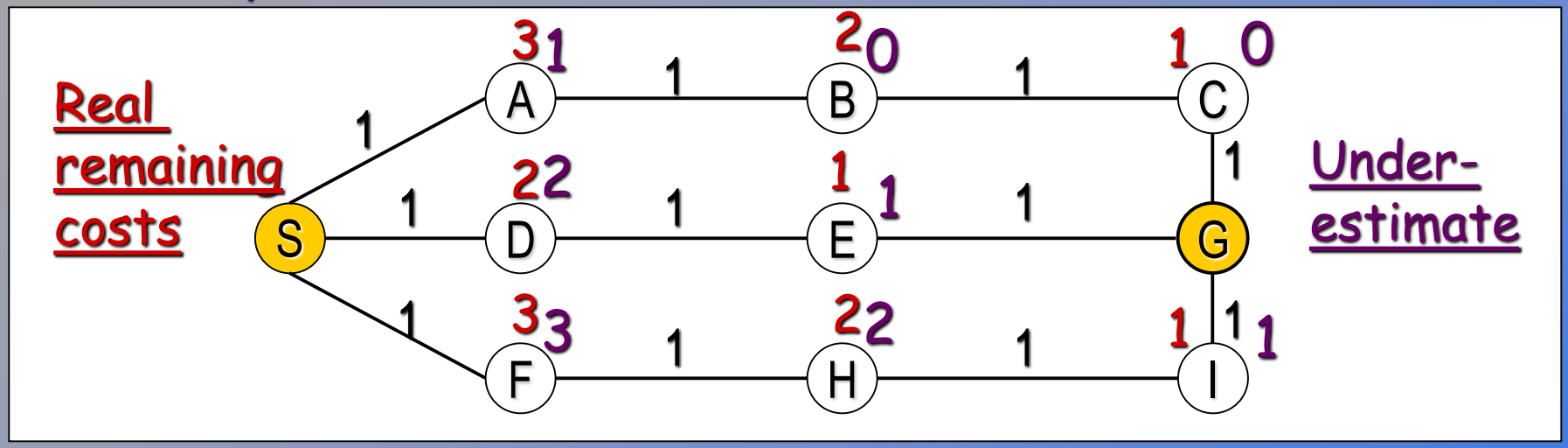


- If h is NOT an underestimate:

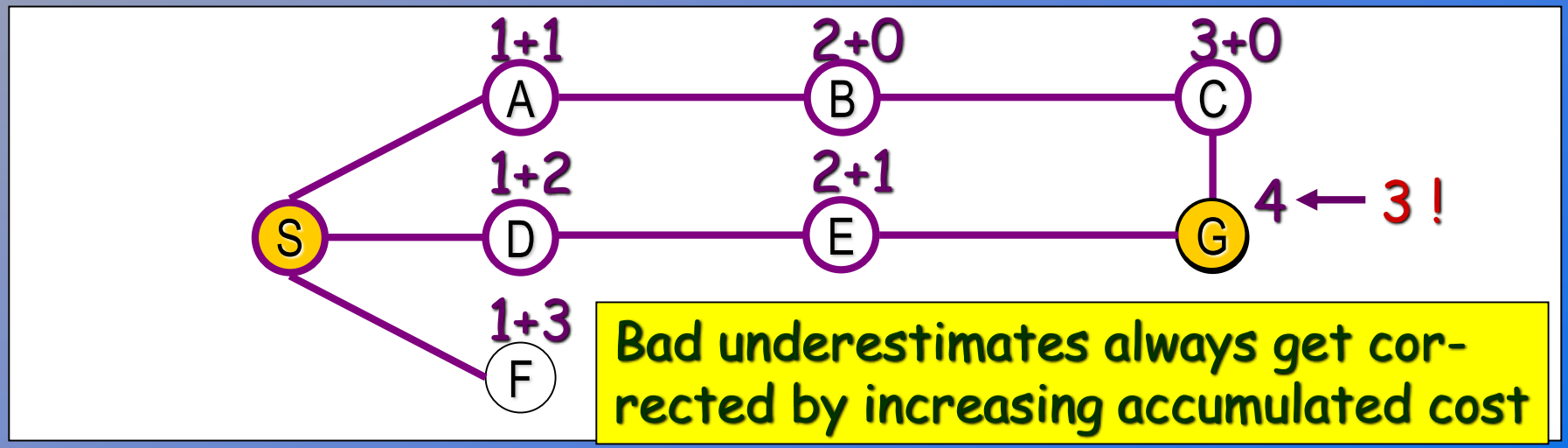


More on underestimates:

- Example:



- If h is an underestimate:

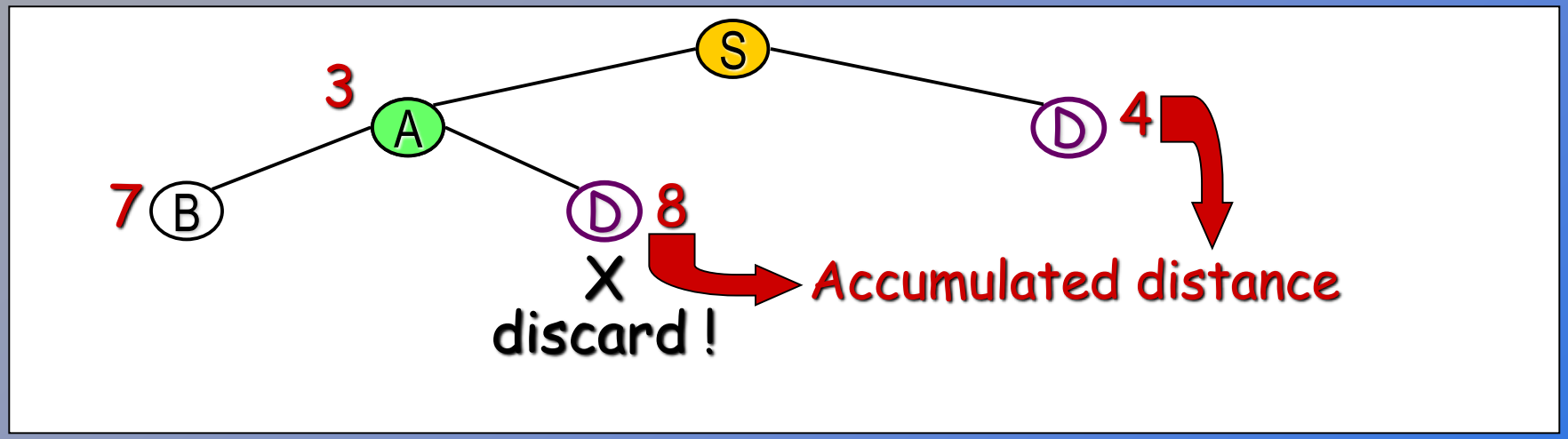


Speed and memory

- In the worst case: no improvement over 'branch and bounded extended uniform cost'
  - ➔ Just take h = 0 everywhere.
- For good heuristic functions: search may expand much less nodes !
  - ➔ See our running example.
- BUT: the cost of computing such functions may be high
  - ➔ Trade-off

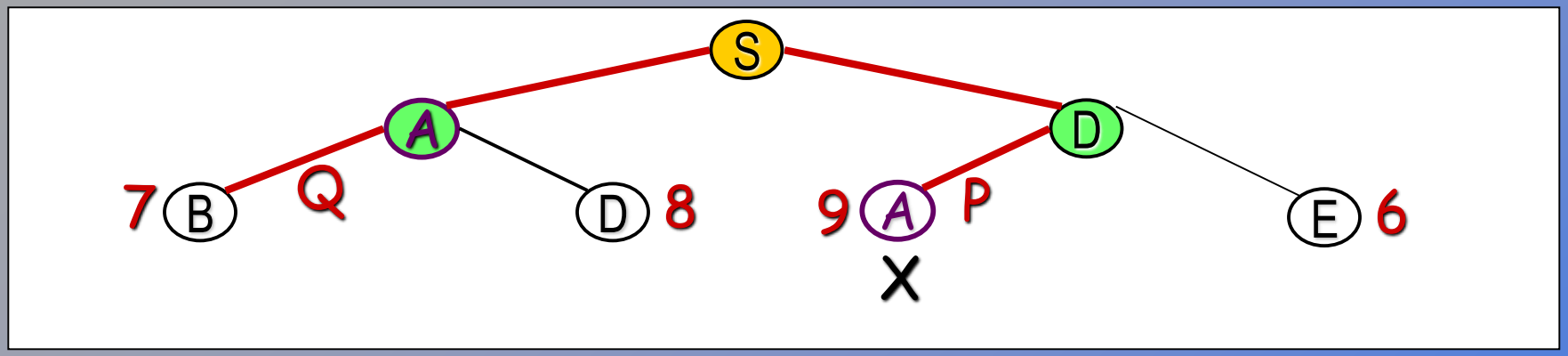
An orthogonal extension: path deletion

- Discard redundant paths:
  - ➔ in the 'branch and bound extended uniform cost' :



- Principle:
  - ➔ the minimum distance from S to G via I = (min. dist. from S to I) + (min. dist. from I to G)

More precisely:

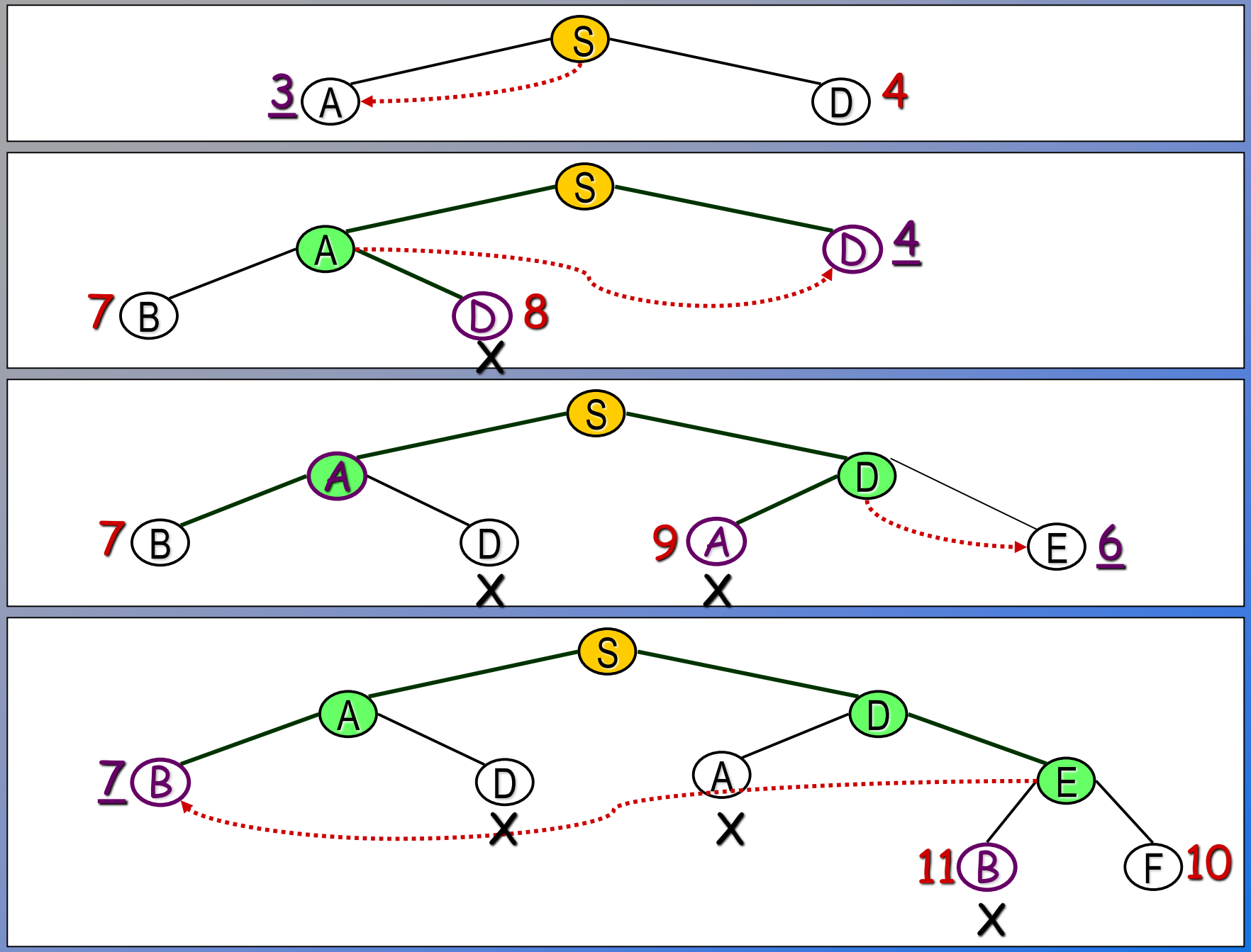


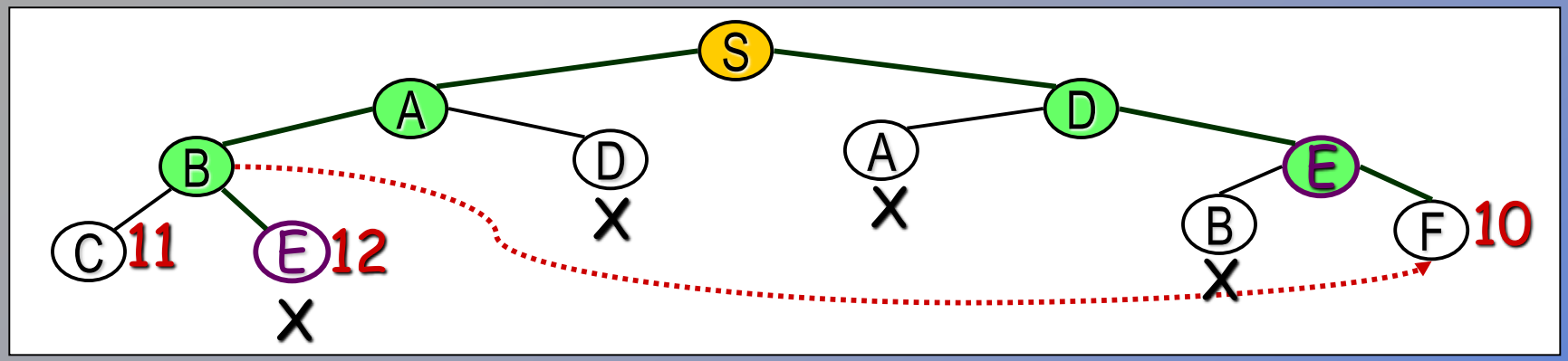
IF the QUEUE contains:

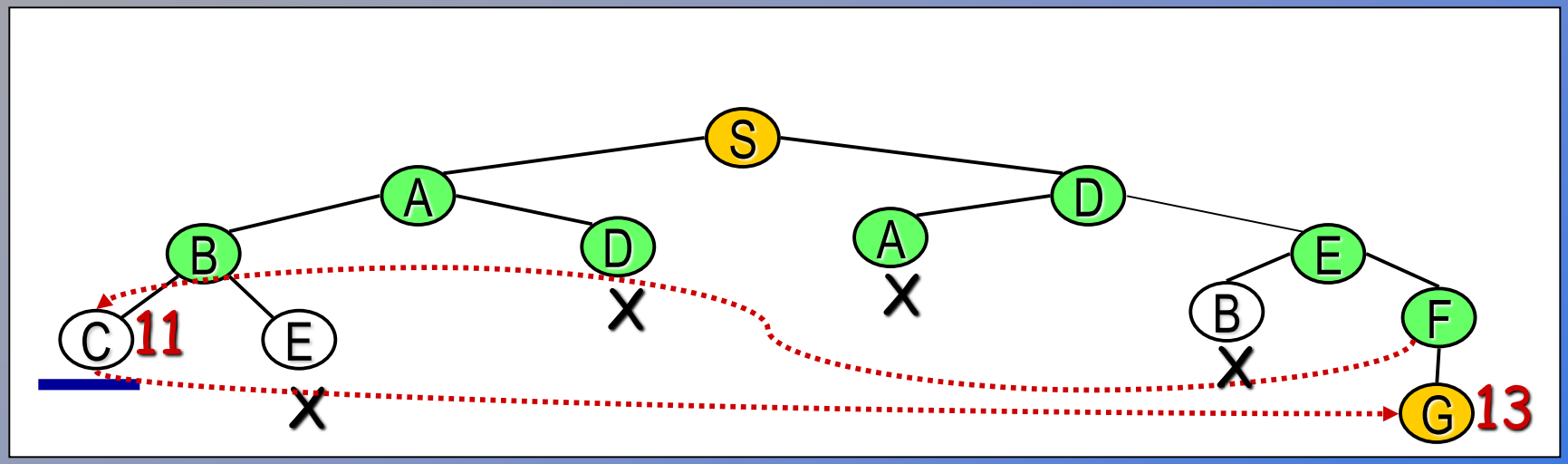
- a path P terminating in I, with cost cost_P
- a path Q containing I, with cost cost_Q
- cost_P ≥ cost_Q

THEN

delete P


S
3
A
D
4
S
A
D
4
7
B
D
8
X
S
A
D
7
B
D
X
9
A
X
E
6
S
A
D
7
B
D
X
A
X
E
11
B
X
F
10





Note how this optimization reduces the number of expansions VERY much, compared to 'branch and bound extended uniform cost'.

5 expansions less !

A* search

- IS:
  - Branch and bound extended,
  - Heuristic Underestimate extended,
  - Redundant path deletion extended,
  - Uniform Cost Search.

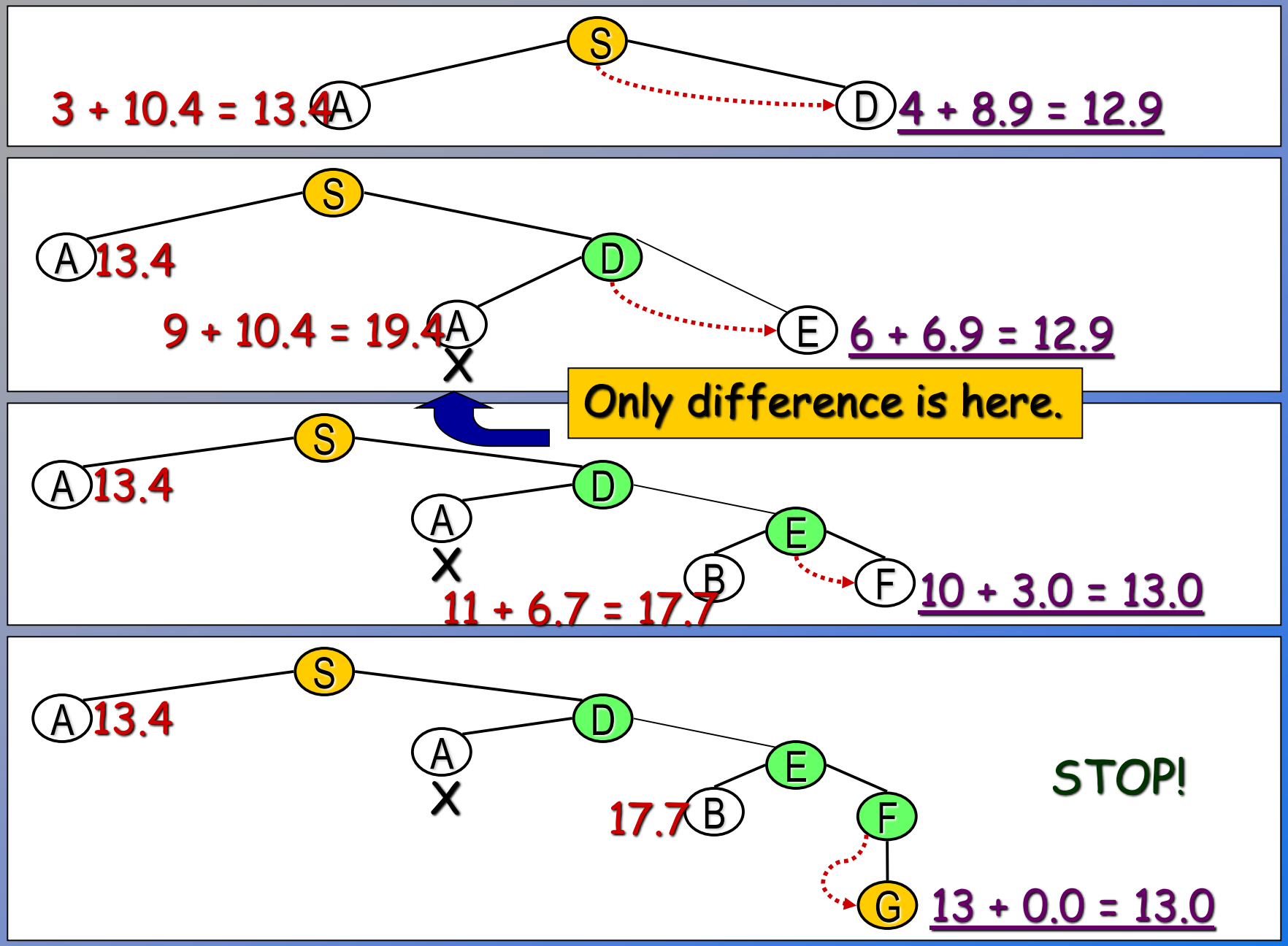
- Note that redundant path deletion is based on the accumulated costs only, so that there is no problem combining it with heuristic underestimates.

A* algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
   AND first path does not reach goal
   remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths and sort the entire QUEUE;
   (by f = cost + h)
   IF QUEUE contains path P terminating in I,
   with cost cost_P, and path Q containing I,
   with cost cost_Q AND cost_P ≥ cost_Q
   THEN delete P

3. IF goal reached THEN success;
   ELSE failure;


S
A
D
3 + 10.4 = 13.4
4 + 8.9 = 12.9
S
A
13.4
D
A
E
9 + 10.4 = 19.4
6 + 6.9 = 12.9
Only difference is here.
S
A
13.4
D
A
E
B
F
11 + 6.7 = 17.7
10 + 3.0 = 13.0
S
A
13.4
D
A
E
17.7
B
F
G
STOP!
13 + 0.0 = 13.0