AI & Robotics

Optimal Search:
The road to A*

PXL AI & ROBOTICS LAB

Goals (1/2)

The **junior-colleague**

- can explain uniform cost search in own words
- can implement uniform cost search
- can describe the link between UCS and breadth-first search
- can explain in own words what accumulated cost means
- can explain how UCS selects the next node
- can explain using an example why UCS does not always find the most optimal path
- can explain in own words and using an example the branch-and-bound principle
- can explain using an example why UCS with branch-and-bound finds the most optimal path
- can implement optimal UCS (using branch-and-bound)
- can describe the time and space complexity of optimal UCS (using branch-and-bound)
- can explain the link and difference with Dijkstra's algorithm

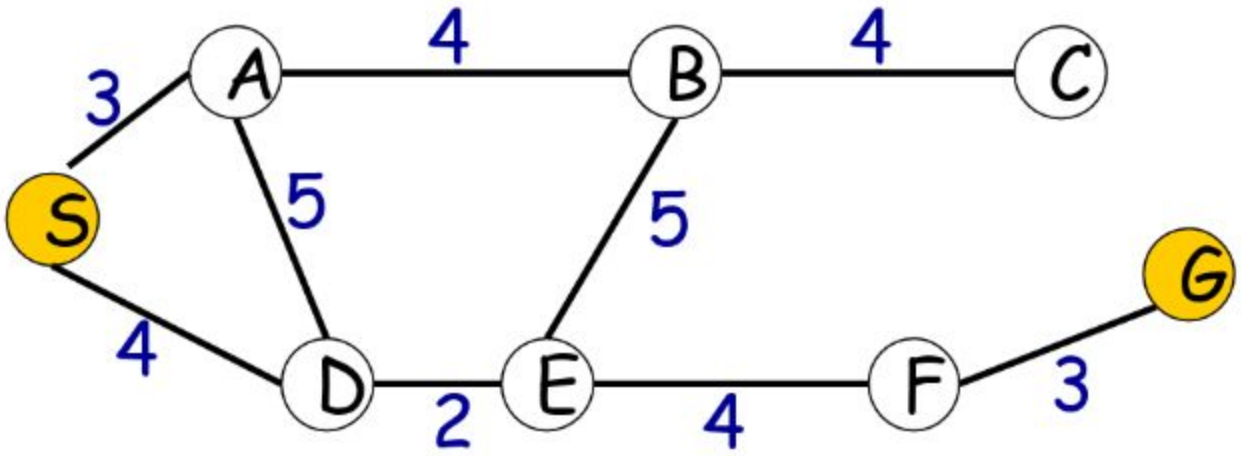
Goals (2/2)

The **junior-colleague**

- can explain in own words what a heuristic is
- can use a given heuristic for a state space
- can explain the heuristics used in the 8-puzzle problem
- can describe in own words what path deletion is and how it works
- can implement path deletion
- can explain A* search in own words
- can implement A* search
- can describe the heuristic function used in A*
- can explain the straight line distance using an example
- can explain the importance of underestimation in the context of heuristics using an example
- can describe completeness, time and space complexity of A*

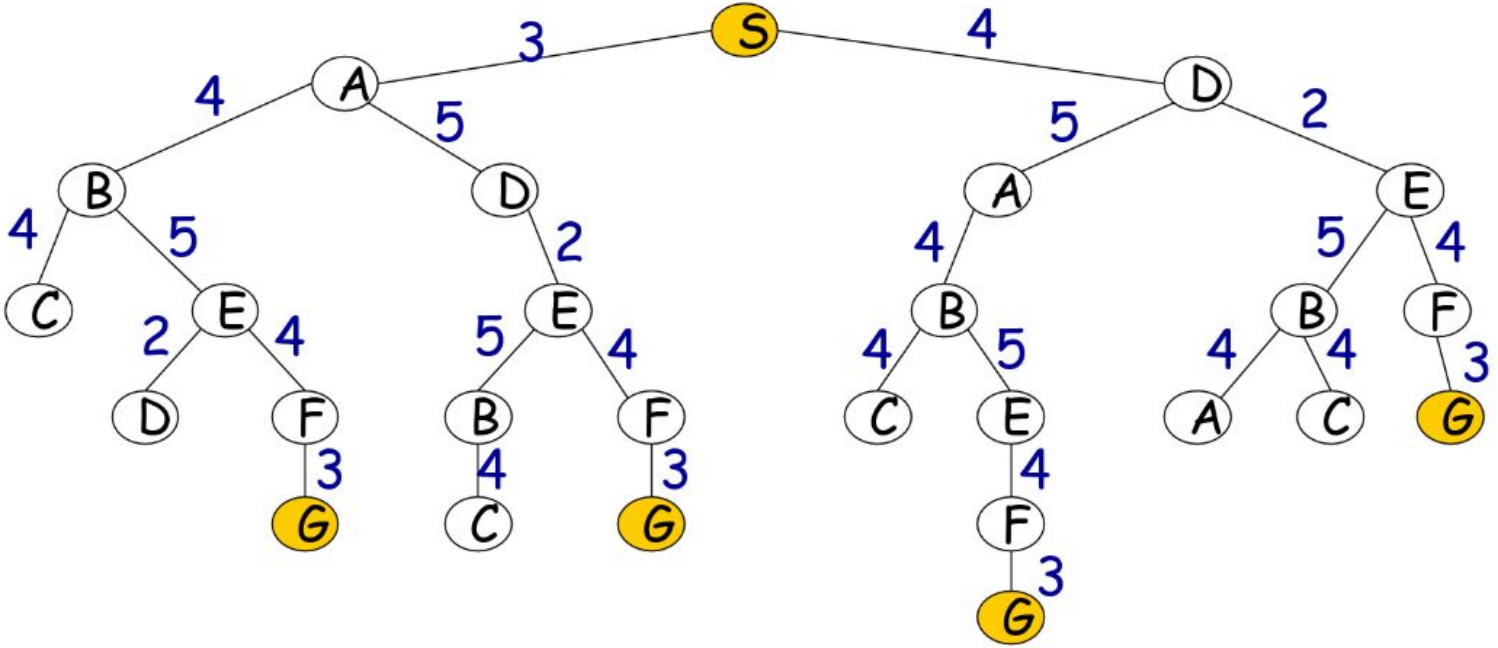
Representation

Running example



Representation: loop-free tree of partial paths

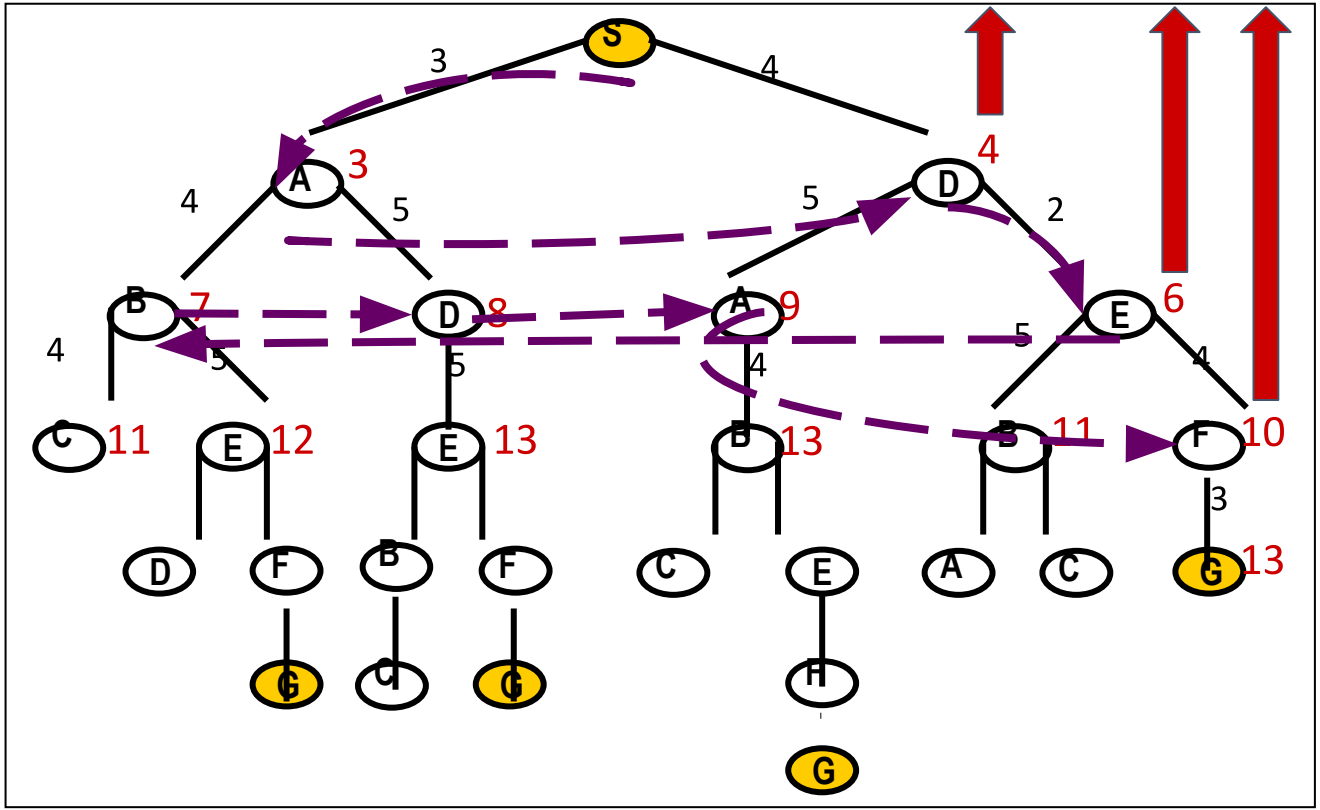
Reintroduce the cost of edges



Uniform cost search

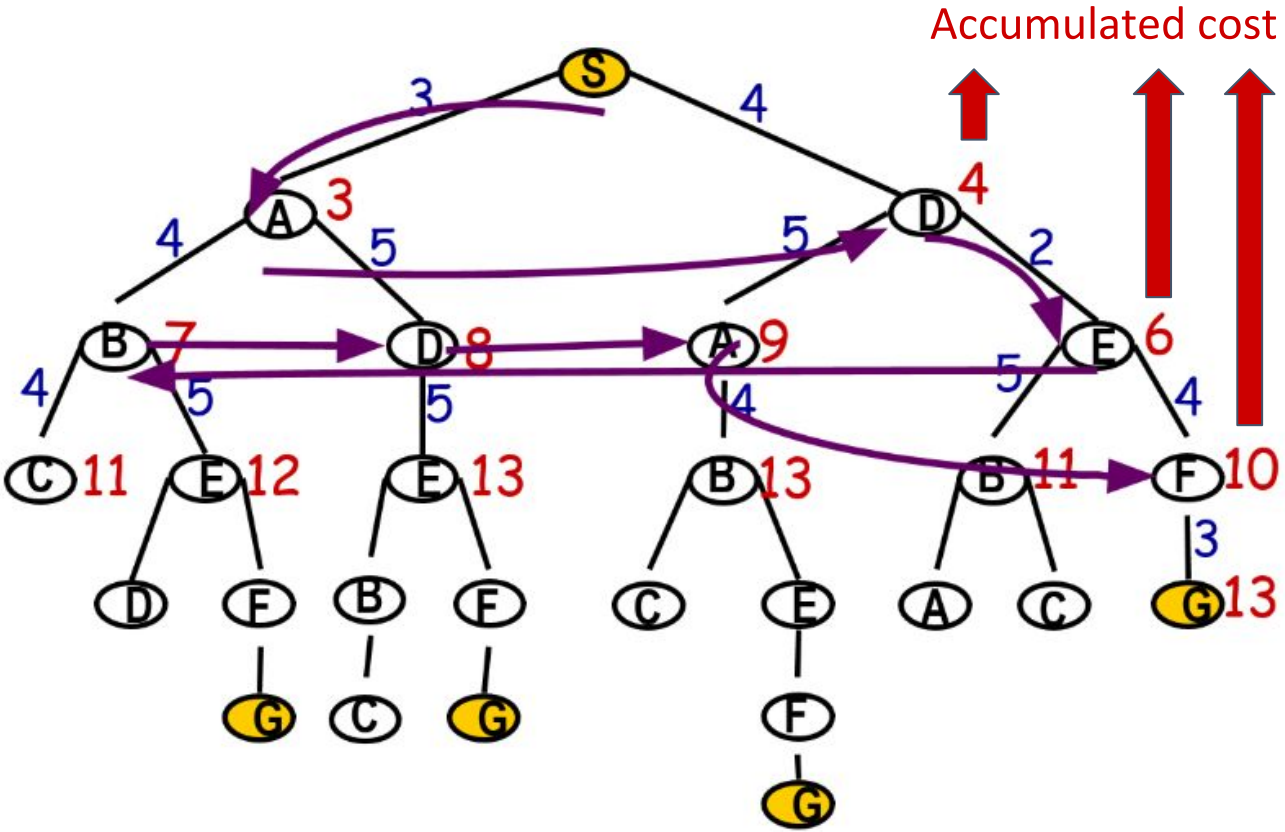
Uniform cost search

- Uniform "best-first" search
- At each step, select the node with the lowest accumulated cost.



Uniform cost search

- Uniform "best-first" search
- At each step, select the node with the lowest accumulated cost.



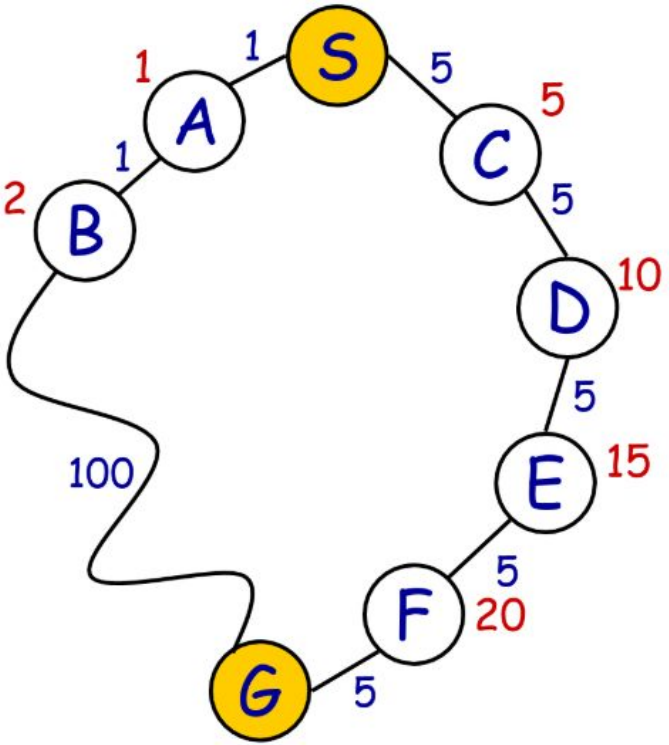
Uniform cost search

[priority: accumulated cost]

```
function uniformCost:
   priorityQueue.insert_with_priority(root)
   while (priorityQueue is not empty and goal is not
 reached)
        v = priorityQueue.pull_highest_priority_element()

   for (all children of v) do
            priorityQueue.insert_with_priority(w)
                [priority: accumulated cost]
```

Uniform cost search

- Problem: **NOT always** optimal
- Uniform cost returns the path with cost *102* while there is a path with cost *25*
- After SAB, G will be selected first, and the goal will be reached

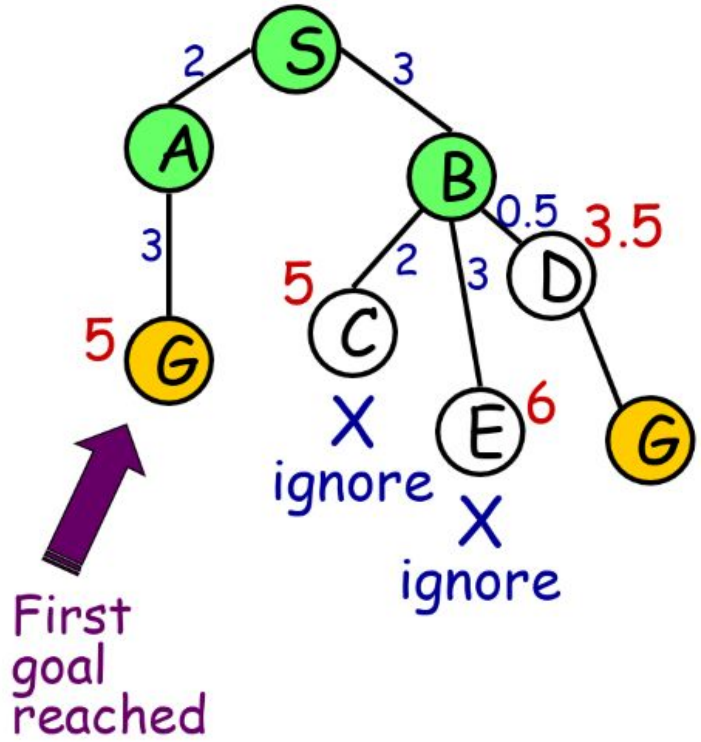


Branch-and-bound principle

- Branch and bound techniques keep track of the solution and its cost and continue searching for a better solution further on.

Branch-and-bound principle

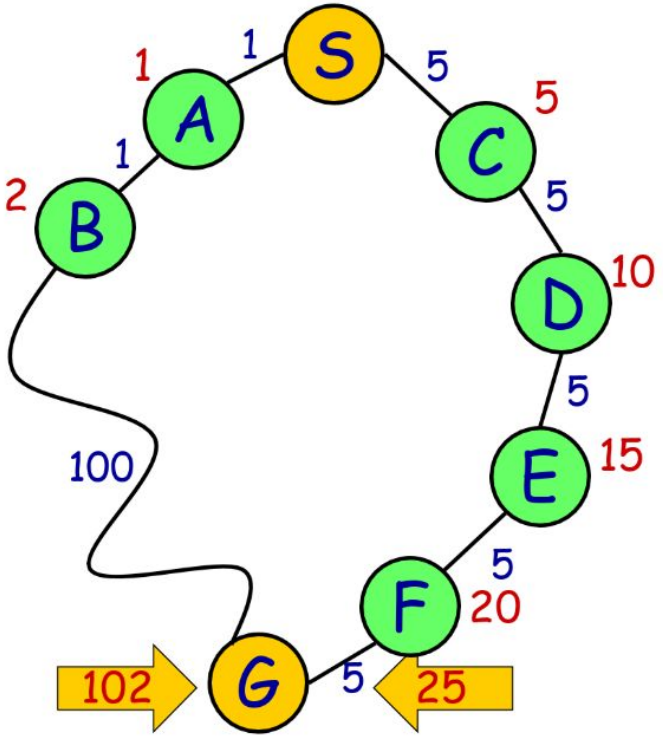
- Use any (complete) search method to find a path
- Remove all partial paths that have an accumulated cost larger or equal than the found path
- Continue search for the next path
- Iterate



Optimal uniform cost search

- Integrate branch-and-bound in uniform cost search
- Change the termination condition:

=> only terminate when a path to a goal node has become the **best** path

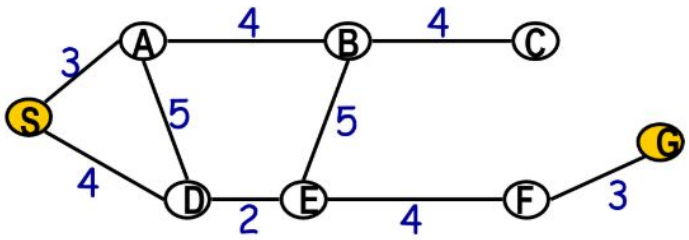


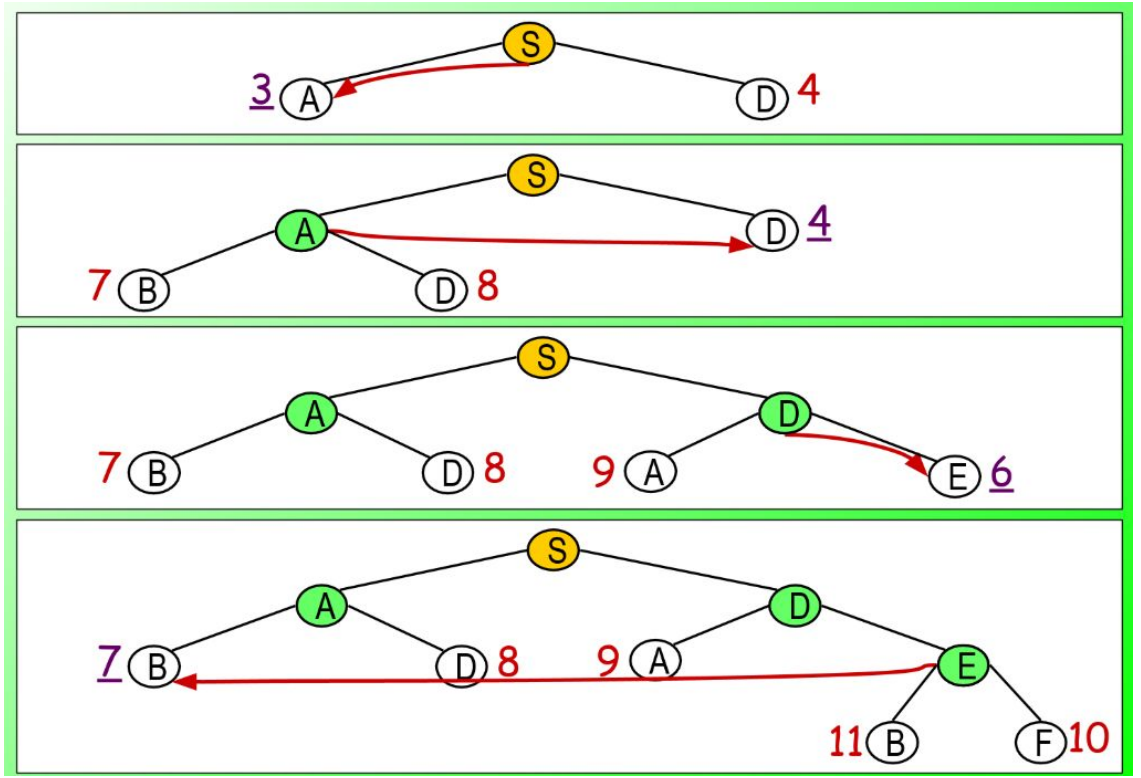
Optimal uniform cost search

[priority: accumulated cost]

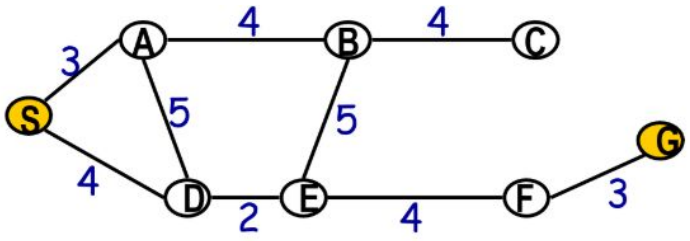
```
function uniformCost:
   priorityQueue.insert_with_priority(root)
   while (priorityQueue is not empty
       and first path does not reach goal)
       v = priorityQueue.pull_highest_priority_element()

   for (all children of v) do
           priorityQueue.insert_with_priority(w)
               [priority: accumulated cost]
```

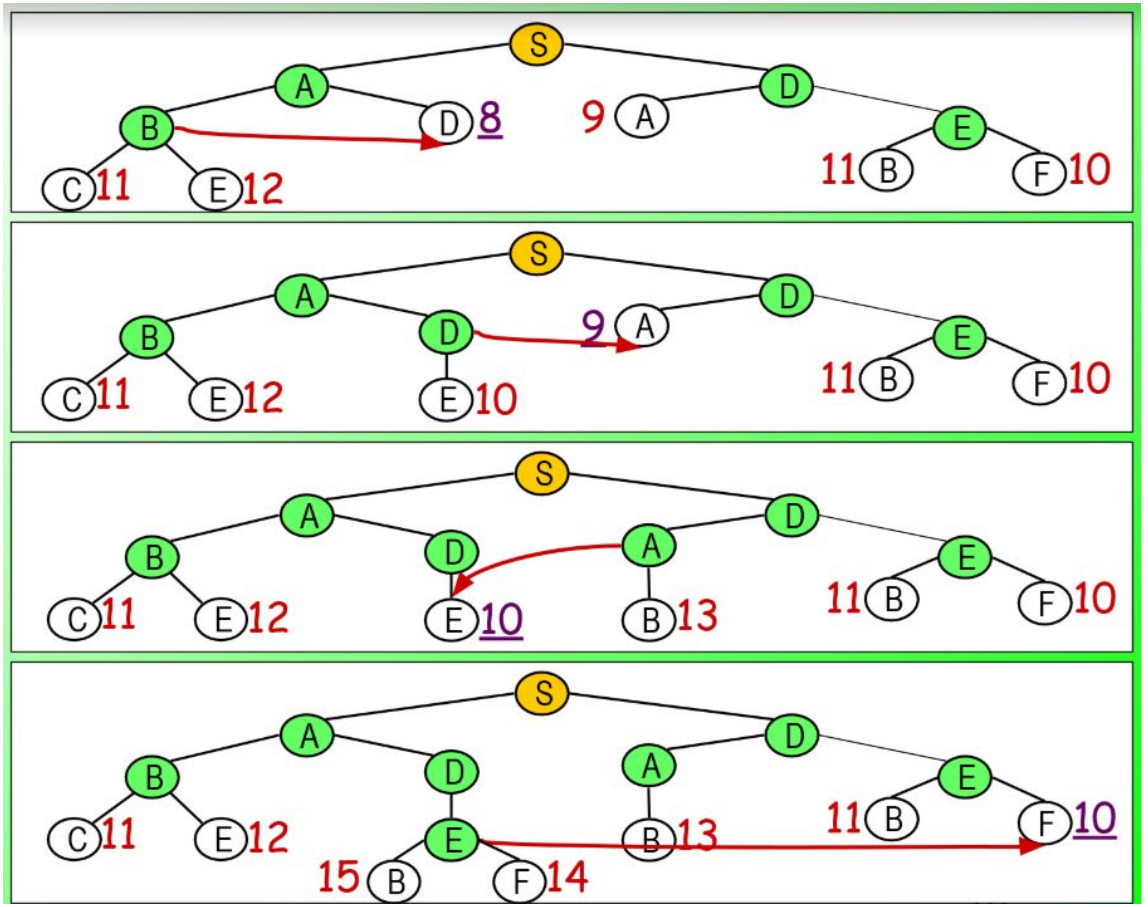
Example


S
A
B
C
D
E
F
G
3
4
4
5
5
4
2
4
3

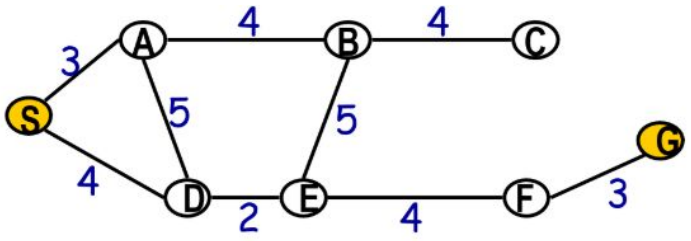

S
3 A
D 4
S
A
D 4
7 B
D 8
S
A
D
7 B
D 8
9 A
E 6
S
A
D
7 B
D 8
9 A
E
11 B
F 10

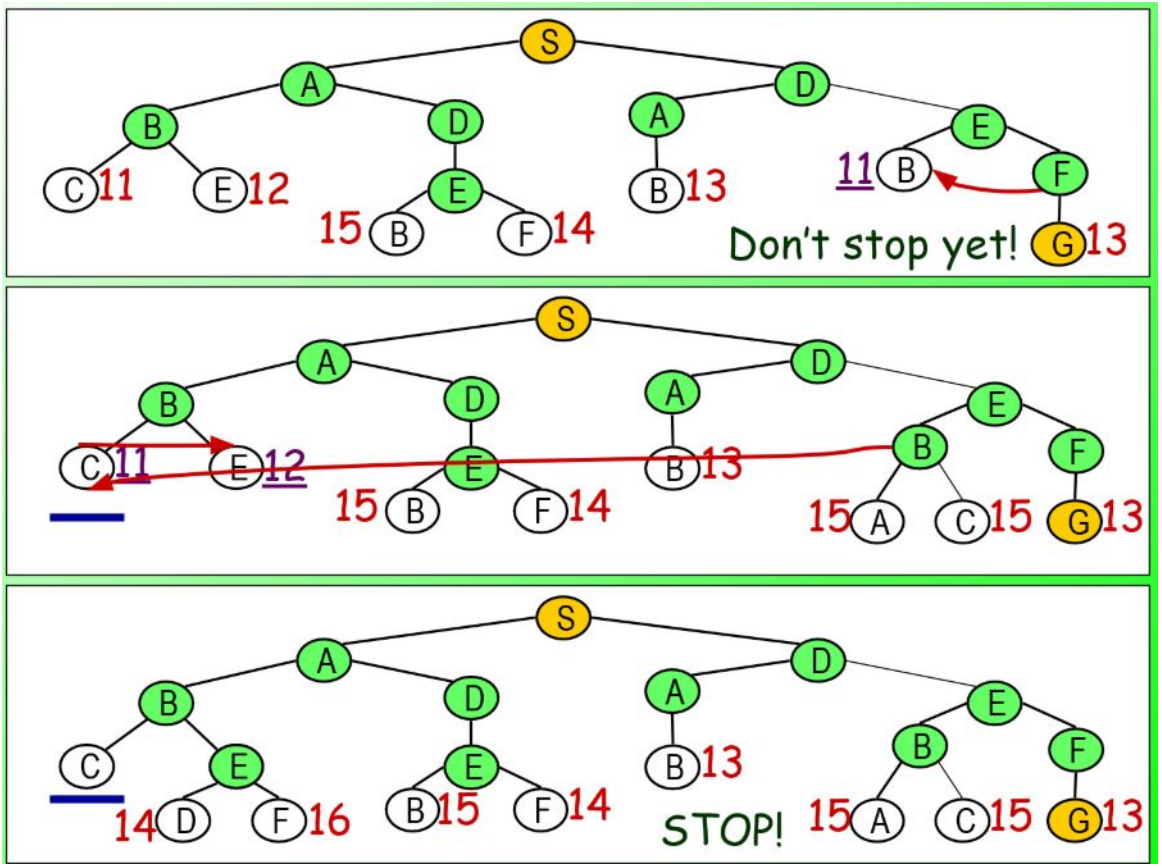
Example


S
A
B
C
D
E
F
G
3
4
4
5
5
4
2
4
3


S
A
D
B
C 11
E 12
D 8
9 A
E
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

Example


S
A
B
C
D
E
F
G
3
4
4
4
5
5
4
2
4
3


S
A
D
B
D
A
E
C 11
E 12
15 B
E
F 14
B 13
11 B
F
Don't stop yet!
G 13
S
A
D
B
D
A
E
C 11
E 12
15 B
E
F 14
B 13
B
F
15 A
C 15
G 13
S
A
D
B
D
A
E
C
E
B
F
14 D
F 16
B 15
E
F 14
B 13
STOP!
15 A
C 15
G 13

Optimal uniform cost search

Optimal path

- If the branching factor is finite, optimal uniform cost search finds the optimal path (if one exists).

Time and Space complexity

- In the worst case, at least as bad as for breadth-first: $O(b^m)$
- + Needs additional sorting step after each path-expansion

Dijkstra’s algorithm

- More general form of Uniform cost search
- Historically significant
- Finds the shortest path from the start node to every other node

  => can also be stopped once the shortest path to the destination node has been determined

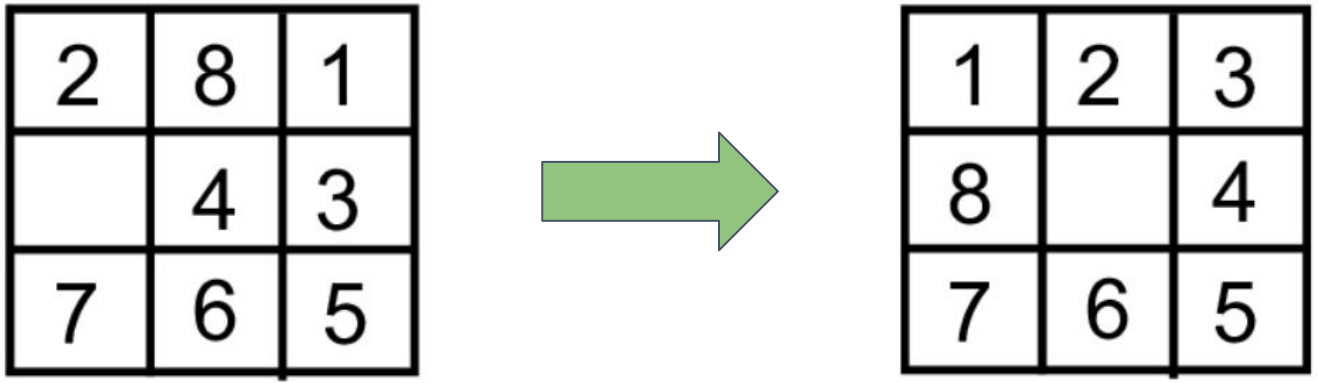
Optimizations

Heuristics

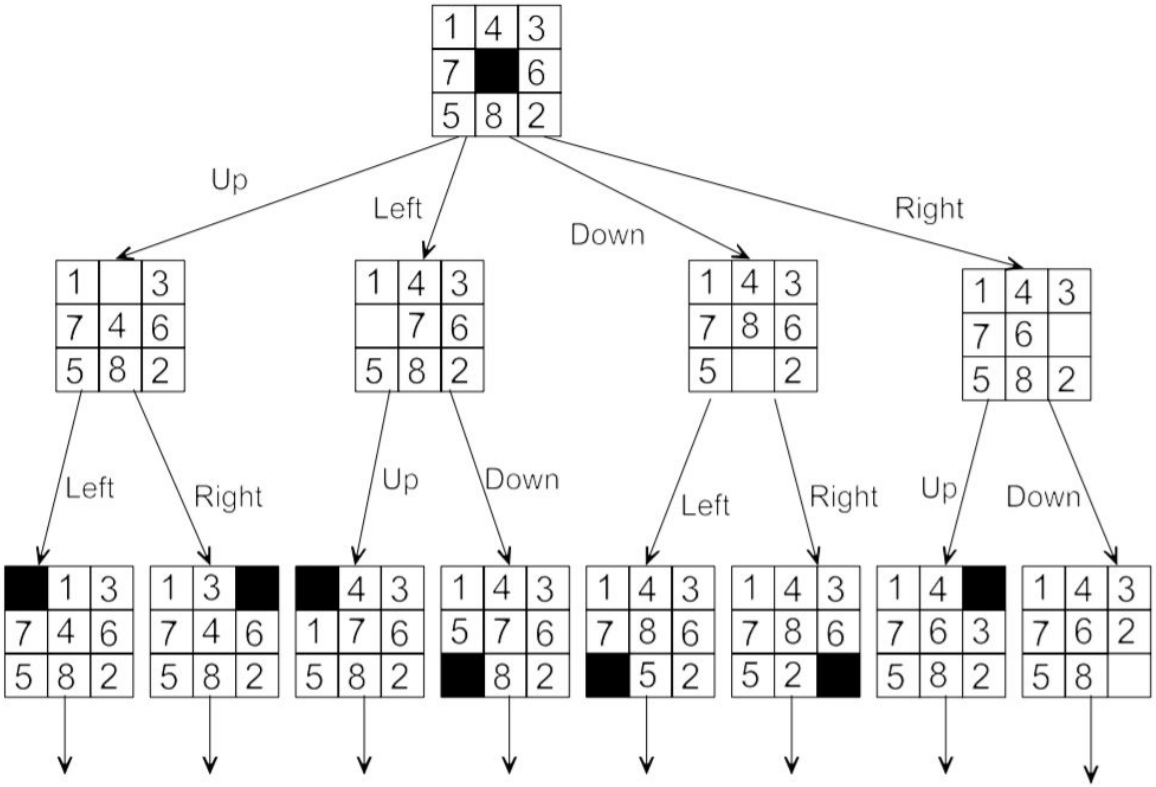
- Guesstimates: approximate cost of finding a solution
- Use problem-specific knowledge
- Not guaranteed to be accurate

  => otherwise we wouldn't need to search
- *f(T)*: expresses the quality of state T
  - *f(T) >= 0* for all nodes
  - *f(T) = 0* for the goal state

Heuristics: Example - The 8-puzzle



Heuristics: Example - The 8-puzzle



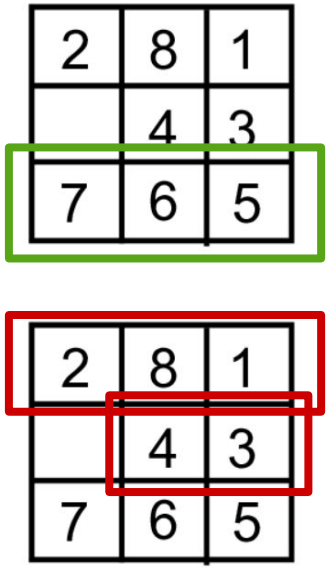
Heuristics: Example - The 8-puzzle

- **Multiple heuristics are possible**
  - *f1(T)*: number of correctly placed tiles

    => 3

  - *f2(T)*: number of incorrectly placed tiles

    => 5



=> In general: ‘distance to goal’ heuristics are more useful

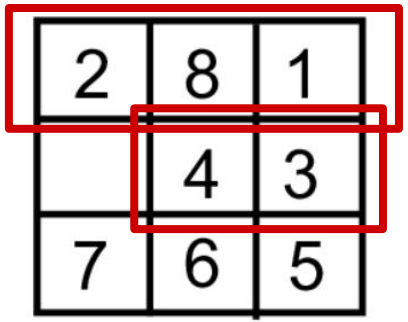
Heuristics: Example - The 8-puzzle

- **Multiple heuristics are possible**
  - *f3(T)*: Manhattan distance:

    the sum of the horizontal + vertical distance that each tile is away from its final destination

    => 1[2] + 2[8] + 2[1] + 1[4] + 1[3]

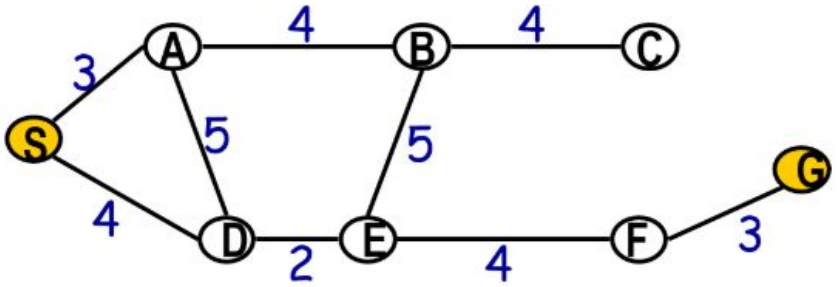
    = 7

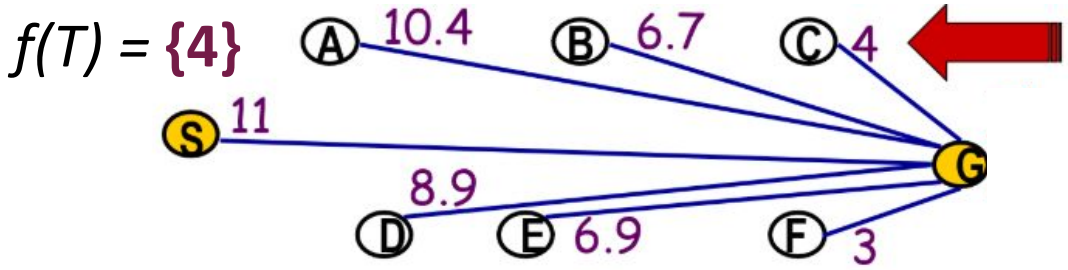


Heuristics: Example

- Straight line distance between a node and the goal

  => Euclidian distance
  https://en.wikipedia.org/wiki/Euclidean_distance



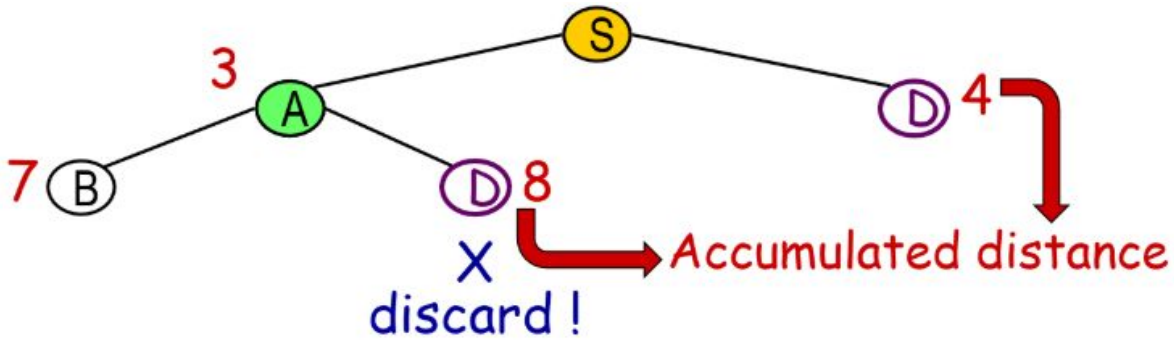


The estimate can be wrong!

Path deletion

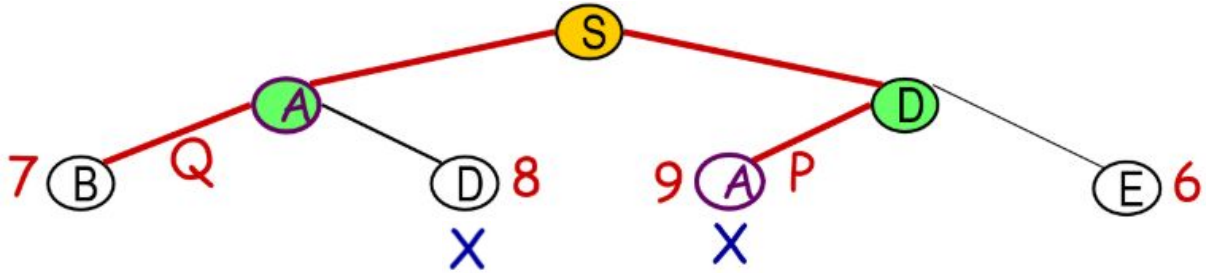
- Discard redundant paths
- Based on the following principle:

min dist. from S to G via N = (min. dist. from S to N) + (min. dist. from N to G)



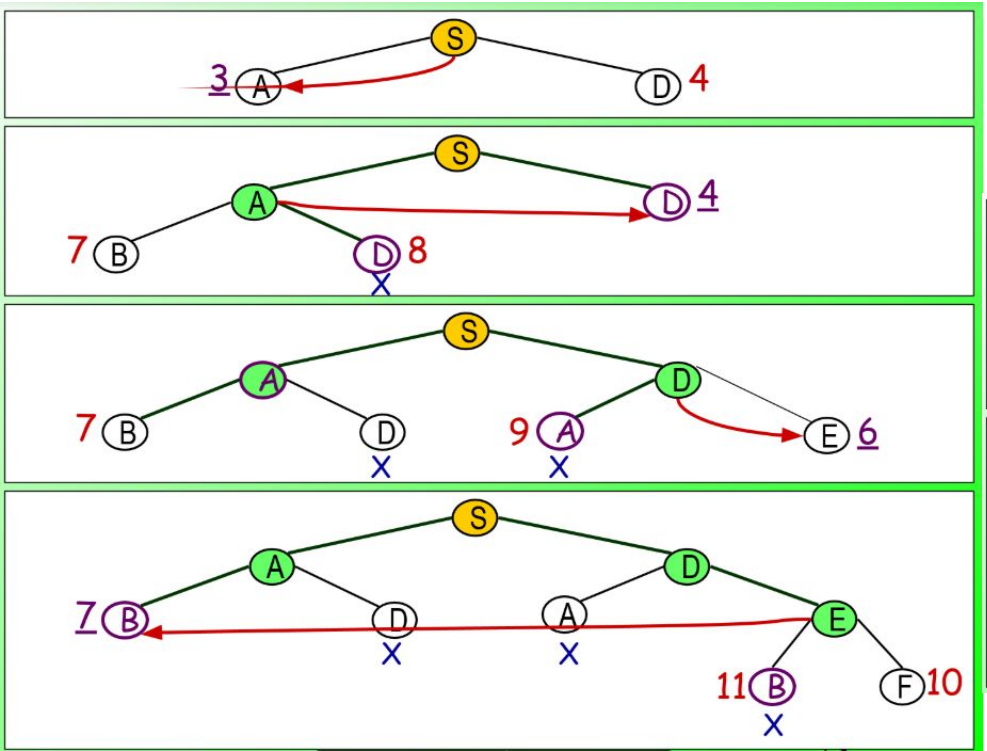
Path deletion

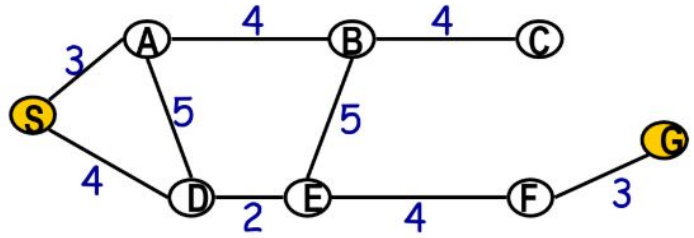
- Formally:

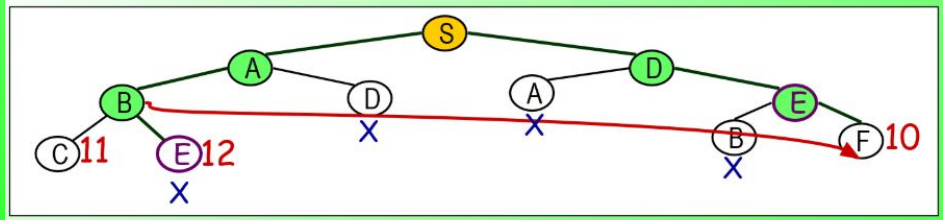


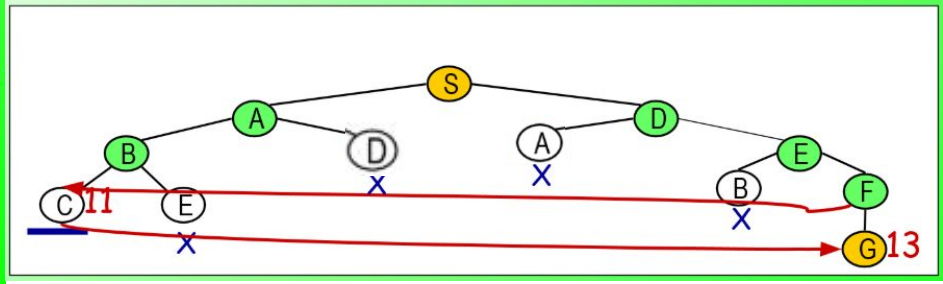

```
if the priorityQueue contains:
    a path P terminating in A, with cost cost_P
    a path Q containing A, with cost cost_Q
    cost_P >= cost_Q
then
    delete P
```

Path deletion: Example









A*

A*

- Branch and bound principle
- Heuristic Underestimate
- Redundant path deletion
- Uniform cost search

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

A*: Heuristic

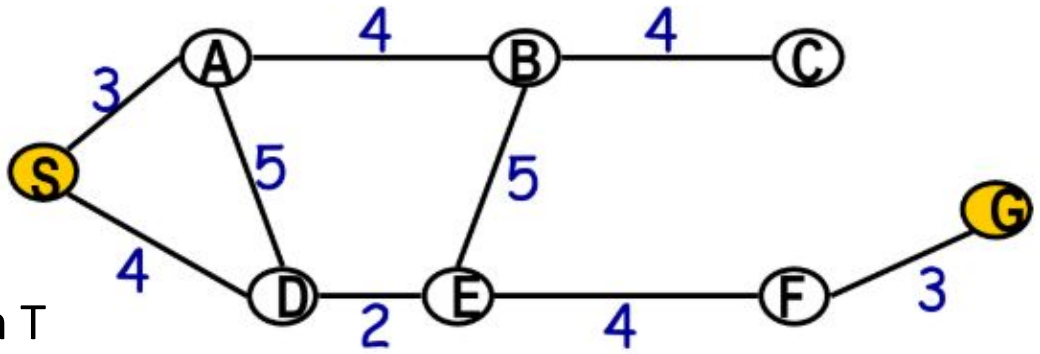
- Replace the ‘accumulated cost’ in the optimal uniform cost algorithm by a function:

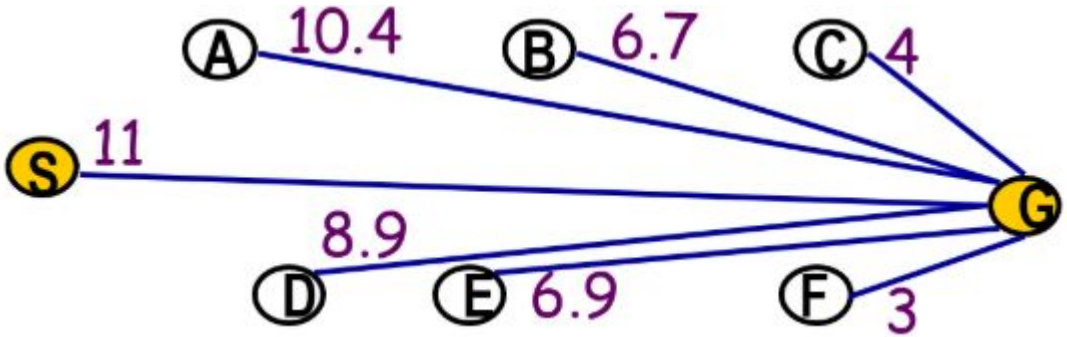
  f(path) = cost(path) + h(endpoint_path)

- *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.

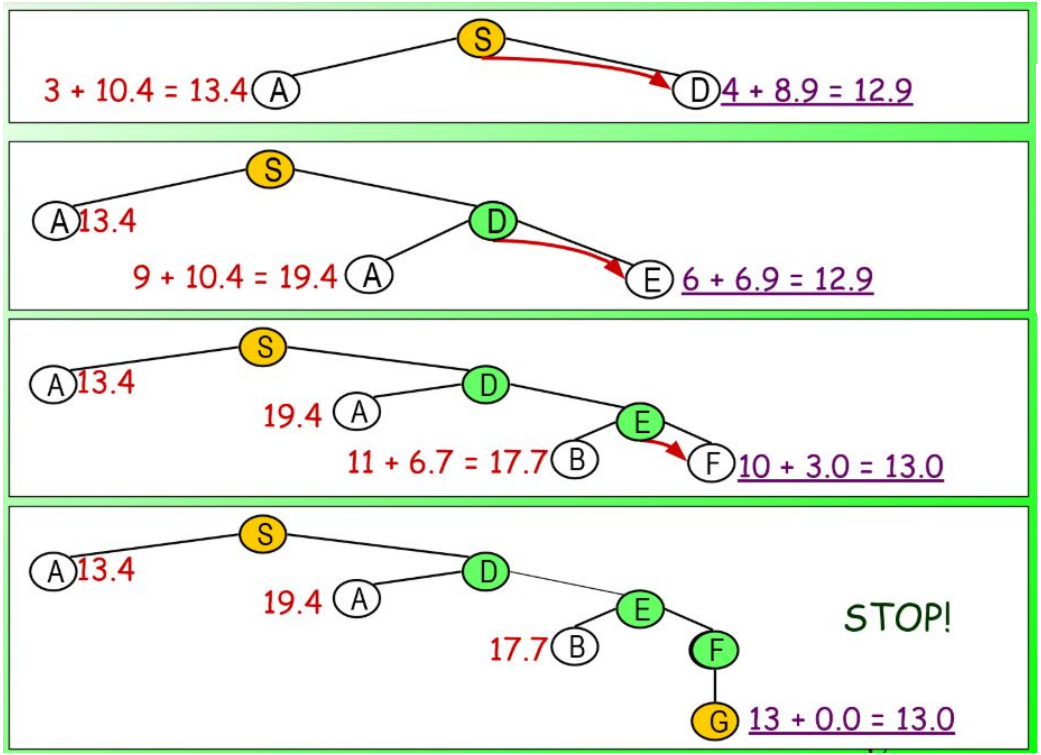
A*: Heuristic

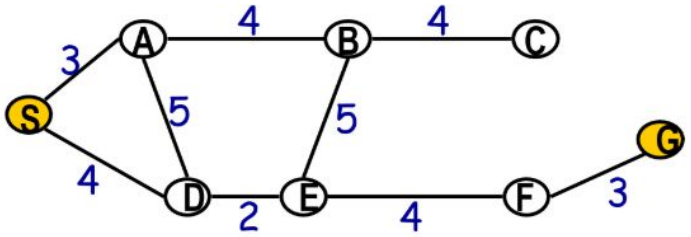
h(T) =
The straight-line distance from T to G

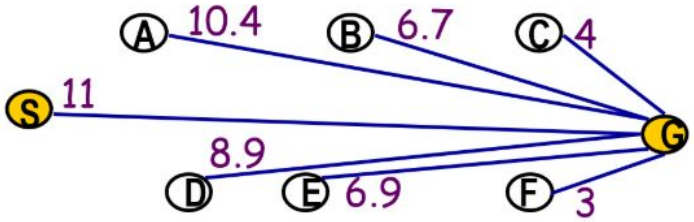




A*: Heuristic

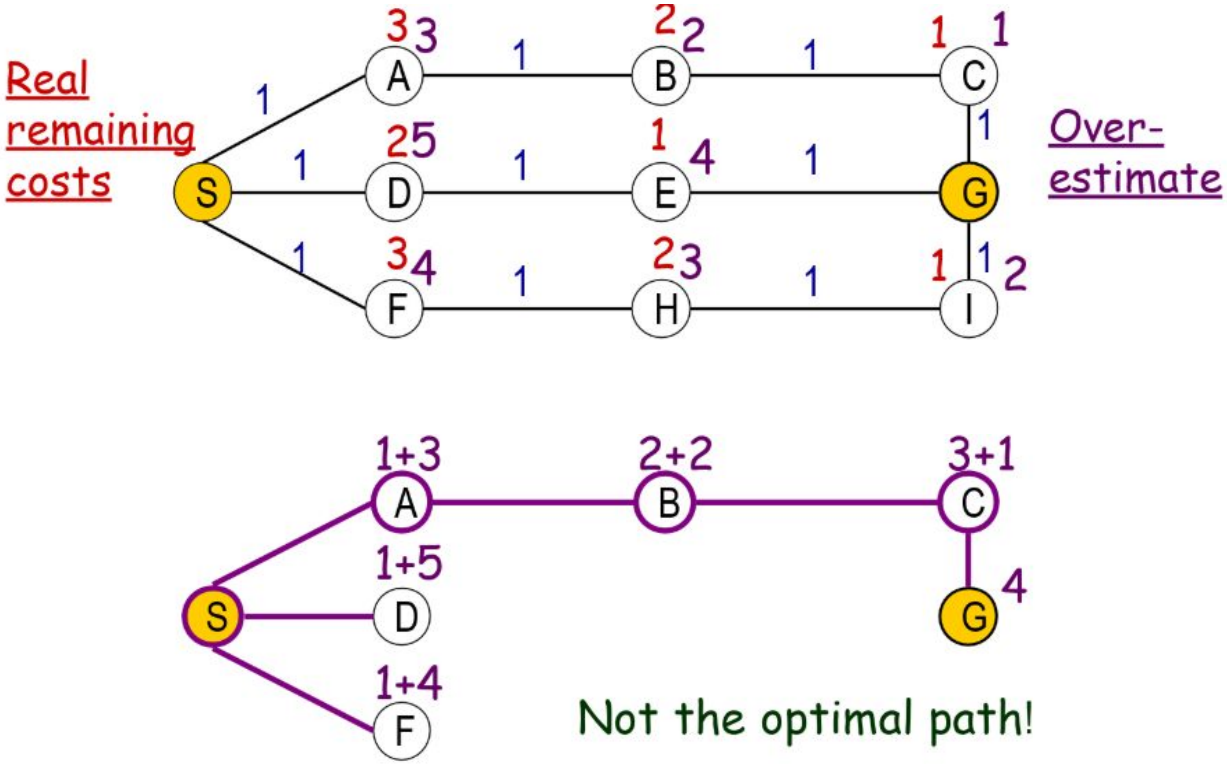

S
3 + 10.4 = 13.4
A
D
4 + 8.9 = 12.9
S
A
13.4
D
9 + 10.4 = 19.4
A
E
6 + 6.9 = 12.9
S
A
13.4
D
19.4
A
E
11 + 6.7 = 17.7
B
F
10 + 3.0 = 13.0
S
A
13.4
D
19.4
A
E
17.7
B
F
G
13 + 0.0 = 13.0
STOP!


A
B
C
S
D
E
F
G
4
4
3
5
5
4
2
4
3


A
10.4
B
6.7
C
4
S
11
G
8.9
D
E
6.9
F
3

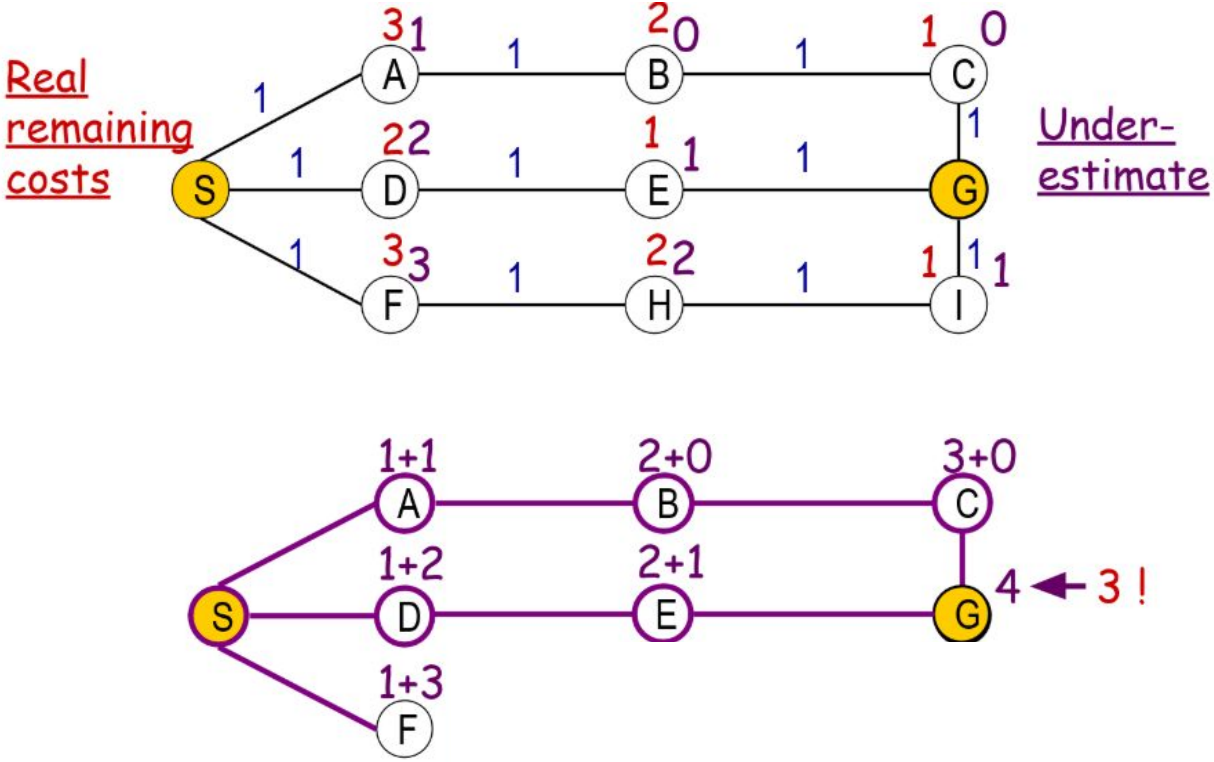
A*: Heuristic

- The importance of heuristic underestimates



A*: Heuristic

- The importance of heuristic underestimates
- Bad underestimates always get corrected by increasing accumulated cost



A*: Heuristic

Time and space complexity

- In the worst case: no improvement over optimal uniform cost
  => Use *h = 0* everywhere.
- For good heuristic functions: search may expand a lot fewer nodes
  => but the cost of computing such functions may be high

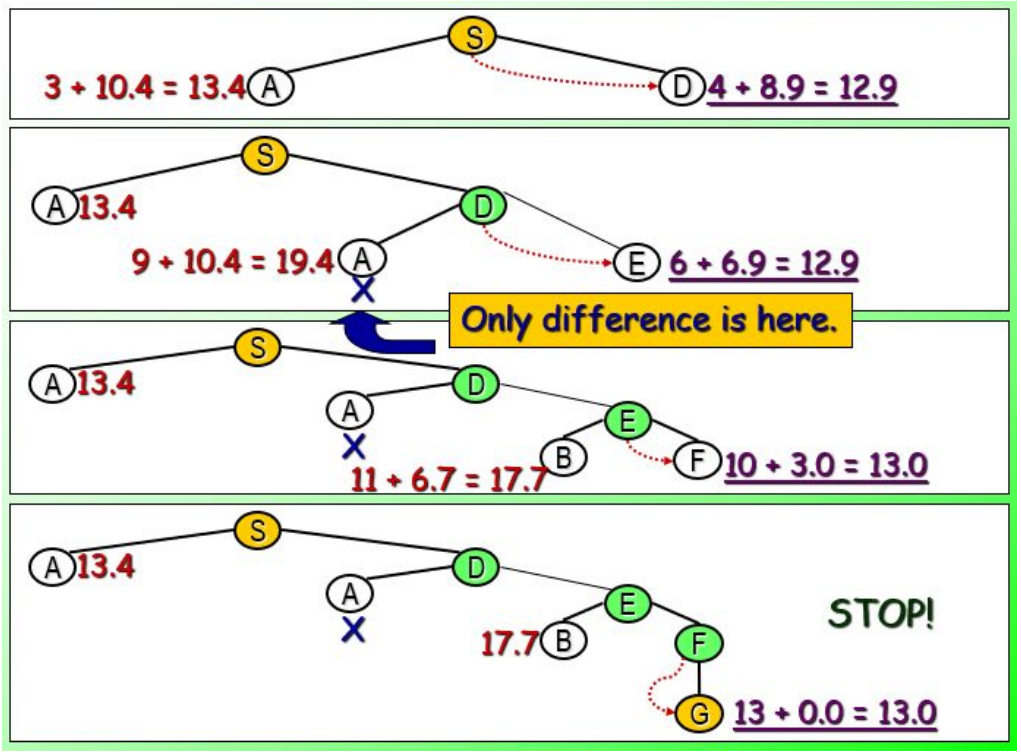
=> Extra improvement in A*: Path deletion

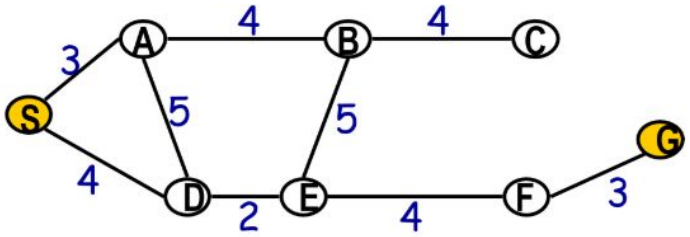
A*

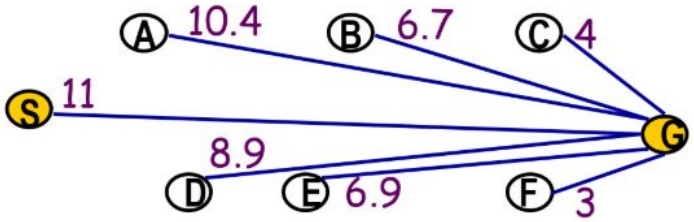
[priority: f = cost + h]

```
function aStarIsBorn:
    priorityQueue.insert_with_priority(root)
    while (priorityQueue is not empty and first path does not reach goal)
         v = priorityQueue.pull_highest_priority_element()

    for (all children of v) do
              priorityQueue.insert_with_priority(w)
                   [priority: f = cost + h]
         if (priorityQueue contains path P terminating in N, with cost cost_P,
                             and path Q containing N, with cost cost_Q
                             and cost_P >= cost_Q)
         then
              delete P
```

A*: Example


S
3 + 10.4 = 13.4
A
D
4 + 8.9 = 12.9
S
A 13.4
D
9 + 10.4 = 19.4
A
E
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
13 + 0.0 = 13.0
STOP!


S
A
B
C
D
E
F
G
3
4
4
5
5
4
2
4
3


A
10.4
B
6.7
C
4
S
11
G
8.9
D
E
6.9
F
3

A*

Completeness

- A* is complete and will always find a solution if one exists

Time complexity (worst case)

- $O(|E|) = O(b^d)$

Space complexity (worst case)

- $O(|V|) = O(b^d)$

=> Extensive analysis falls outside of the scope of this course

=> Research Project!