

# Problem solving by search II

Tomáš Svoboda, Matěj Hoffmann, and Petr Pošík

Vision for Robots and Autonomous Systems, Center for Machine Perception  
Department of Cybernetics  
Faculty of Electrical Engineering, Czech Technical University in Prague

March 3, 2023

# Outline

- ▶ Graph search
- ▶ Heuristics (how to search faster)
- ▶ Greedy
- ▶ A\*. A-star search.

# A Maze, what could possibly go wrong?

	0	1	2	3	4	
0	0.00	0.00	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0.00	0.00	1
2	0.00	0.00	0.00	0.00	0.00	2
3	0.00	0.00	0.00	0.00	0.00	3
4	0.00	0.00	0.00	0.00	0.00	4
	0	1	2	3	4	

<https://youtu.be/WKSoedfRZQ4>

3 / 26

## Notes

Analyze the demo run (BFS). What happened? Why did it take that long?

Because it is TREE\_SEARCH...

Many loops are created and all nodes with depth < 7 need to be expanded first. Goal is at depth 8.

Notes for teacher:

Working note for demo:

```
python3 easy_search_agents.py
```

'n' for next

's' for skip

code settings:

```
MAP = 'maps/easy/easy2.bmp'
```

```
TREE_SEARCH = True
```

```
node_type = 'BFS'
```

How to decode printout on command line:

- Every iteration ends with: `print('End of while loop: length of the frontier:', len(frontier), 'length of the expanded:', len(expanded.states), frontier, frontier.is_empty())`
- But note that the algo is written in a general way (like UCS), stopping after expanding the goal node – that is why you see also depth 9 in the frontier notes at the end.
- Size of the visualiation can be altered in `./kuimaze/maze.py`, look for `MAX_CELL_SIZE`

# Tree search the maze

**function** TREE\_SEARCH(env) **return** a solution or failure

    initialize the **frontier**

**while** frontier **do**

        node  $\leftarrow$  frontier.pop()

**if** goal in node **then return** node

**end if**

        child\_nodes  $\leftarrow$  env.expand(node.state)

        Add child\_nodes to frontier

**end while**

**end function**

	0	1	2	3	4	
0	0.00	0.00	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0.00	0.00	1
2	0.00	0.00	0.00	0.00	0.00	2
3	0.00	0.00	0.00	0.00	0.00	3
4	0.00	0.00	0.00	0.00	0.00	4
	0	1	2	3	4	

## Notes

Make a **frontier** and **expand** columns on a paper and follow the algorithm by putting and removing (scratching out) nodes from the list.

Note that there are many more nodes than states (*search tree vs. state space*).

*Tree search* seems hugely ineffective. Note that this is (also) because of the state space. It's a maze with undirected edges. If we had directed edges, there would be much much fewer cycles.

# A graph search

```
function GRAPH_SEARCH(env) return a solution or failure
  init frontier by the start state
  initialize the explored set to be empty
  while frontier do
    node ← frontier.pop()
    add node.state to explored
    if goal in node then return node
  end if
  child_nodes ← env.expand(node.state)
  for all child_nodes do
    if child_node.state not in explored then
      add nodes to frontier
    end if
  end for
end while
end function
```

▷ What about frontier?



**Do not forget:** `node` is not the same as `state`!

---

## Notes

Think about what is node and what state. What is main difference? How are they connected? Where do they appear? What is node/state in the maze problem?

The main idea: Do not expand a `state` twice.

What would be a good data structure to implement the *explored* set? Yes, it would be a *set* ;) – where every element is present only once. Unlike *list*.

"What about frontier?" - if you can ensure that the first time you add a node to frontier, the state will be reached by an optimal path from start, you can also check frontier here (e.g., BFS). If you can't guarantee that, you have to be more careful.

# The BFS graph search

**function** BFS\_GRAPH\_SEARCH(env) **return** a solution or failure

node ← env.observe()

frontier ← FIFOqueue(node)

explored ← set()

**while** frontier not empty **do**

node ← frontier.pop()

explored.add(node.state)

▷ Add state, not node!

child\_nodes ← env.expand(node.state)

**for all** child\_nodes **do**

**if** child\_node.state not in explored and not in frontier **then**

**if** child\_node contains Goal **then return** child\_node

**end if**

frontier.insert(child\_node)

**end if**

**end for**

**end while**

**end function**

6 / 26

---

## Notes

Why adding/checking state and not node in explored data structure? Can I do the simple presence check for all kind of graph search algorithms?

Run demo again with BFS graph search.

Notes for teacher:

**TREE\_SEARCH = False**

Working note for demo:

```
python3 easy_search_agents.py
```

'n' for next

's' for skip

code settings:

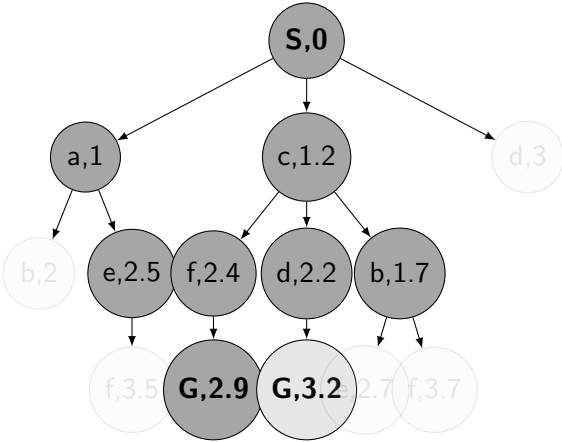
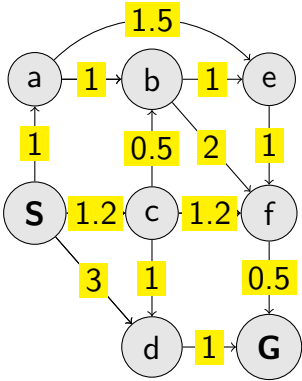
```
MAP = 'maps/easy/easy2.bmp'
```

```
TREE_SEARCH = False
```

```
node_type = 'BFS'
```

Result can be also seen at: [https://youtu.be/4yu\\_nsWZ2ck](https://youtu.be/4yu_nsWZ2ck)

# Checking frontier in uniform costs graph search?



## Notes

When following the algorithm (animation) use the paper list of frontier and explored  
Note the extra features of UCS vs. BFS in action:

1. Update of cost:
  - "b,2" disappears as "b,1.7" appears – update with lower cost.
  - Similarly, "e,2.7" and "f,3.7" appear to immediately disappear again – their cost is higher than already available for those states.
2. Termination only after expanding node with goal state.

# The UCS graph search

**function** UCS\_GRAPH\_SEARCH(env) **return** a solution or failure

node ← env.observe()

frontier ← priority\_queue(node)

explored ← set()

▷ path\_cost for ordering

**while** frontier not empty **do**

node ← frontier.pop()

**if** node contains Goal **then return** node

▷ check here!

**end if**

explored.add(node.state)

child\_nodes ← env.expand(node.state)

**for all** child\_nodes **do**

**if** child\_node.state not in explored and not in frontier **then**

frontier.insert(child\_node)

**else if** child\_node.state in frontier with higher cost **then**

replace that node with the child\_node

**end if**

**end for**

**end while**

**end function**

8 / 26

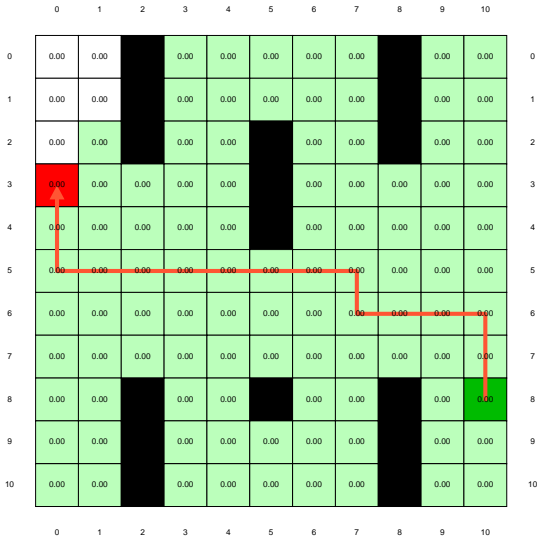
---

## Notes

Does the algorithm always find the best (cheapest) path? Are there any requirements for the path optimality function?



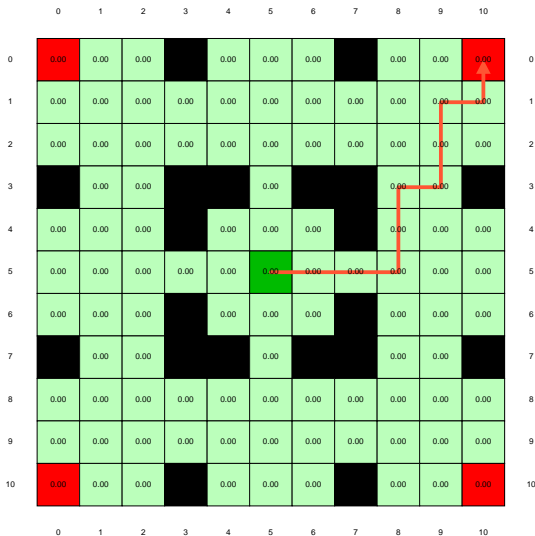
# Few examples of search strategies so far



Run the demos.

Notes

# What is wrong with UCS and other strategies?



Run the demo, or see <https://youtu.be/TT5MY8xCgAg>

---

## Notes

## Node selection, take argmin $f(n)$

Selecting next node to expand/visit (pop operation):

$$\text{node} \leftarrow \underset{n \in \text{frontier}}{\text{argmin}} f(n)$$

What is  $f(n)$  for DFS, BFS, and UCS?

- |        |                                |
|--------|--------------------------------|
| ▶ DFS: | ▶ $f(n) = n.\text{path\_cost}$ |
| ▶ BFS: | ▶ $f(n) = n.\text{depth}$      |
| ▶ UCS: | ▶ $f(n) = -n.\text{depth}$     |

The good: (one) frontier as a priority queue

(I.e., priority queue will work universally. Still, stack (LIFO) and queue (FIFO) are (conceptually) the perfect data structures for DFS and BFS, respectively.)

The bad: All the  $f(n)$  correspond to the cost from  $n$  to the start - only **backward cost**; **cost-to-come** (to  $n$ ).

---

### Notes

- DFS:  $f(n) = -n.\text{depth}$
- BFS:  $f(n) = n.\text{depth}$
- UCS:  $f(n) = n.\text{path\_cost}$

Do humans look back when planing path? Is looking back important at all? If yes, when?

## How far are we from the goal cost-to-go ? – Heuristics

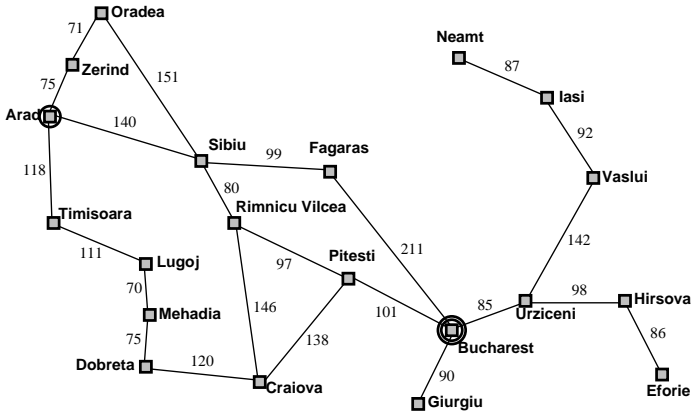
- ▶ A function that estimates how close a *state* is to the goal.
- ▶ Designed for a particular problem.
- ▶  $h(n.state)$  – it is function of the state (attribute of node)
- ▶ It is often shortened as  $h(n)$  – heuristic value of node  $n$ .

---

### Notes

What happens if  $h(n) = \text{true cost}$ ?

# Example of heuristics



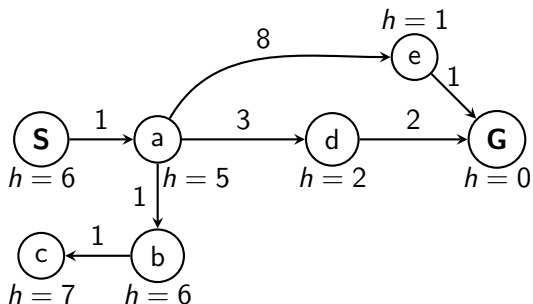
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

## Notes

Straight-line distance to Bucharest.

Illustration of *greedy* failing: Imagine going from Iasi to Fagaras. Neamt will be chosen for expansion. This will add Iasi back. Iasi is closer to Fagaras than Vaslui is and will be expanded again. Infinite loop... (3.5.1. in [2])

Greedy, take the  $n^* = \operatorname{argmin}_{n \in \text{frontier}} h(n)$



What is wrong (and nice) with the Greedy?

---

Notes

Also called “Greedy best-first search” [2].

What will happen in this example:

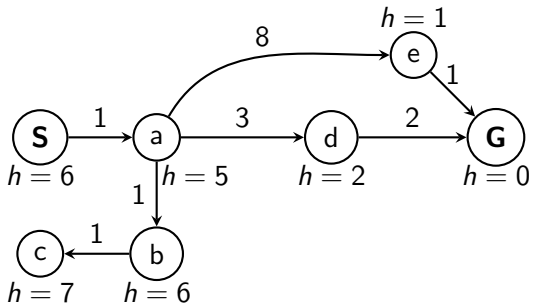
1. Expand “S”. Add “a” to frontier.
2. Expand “a”. Add “b”, “d”, “e”.
3. Expand “e” ( $h = 1$ ). Get “G”.

Wrong:

- not optimal
- not complete (tree search version) (Can be shown on the Romania example – go back.)
- (graph search version is complete only in finite state spaces)

Nice: it is simple.

# A\* combines UCS and Greedy

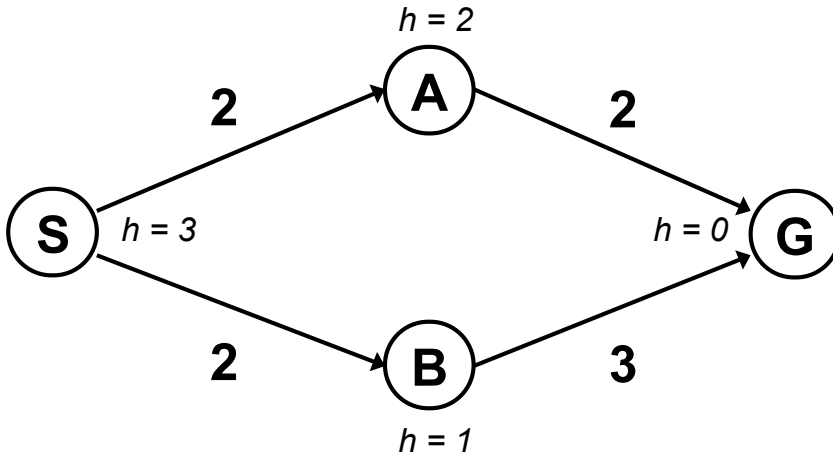


UCS orders by backward (path) cost  $g(n)$

Greedy uses heuristics (goal proximity)  $h(n)$

A\* orders nodes by:  $f(n) = g(n) + h(n)$

# When to stop A\*?



1

When popping  $G$  from frontier.

<sup>1</sup>Graph example: Dan Klein and Pieter Abbeel

16 / 26

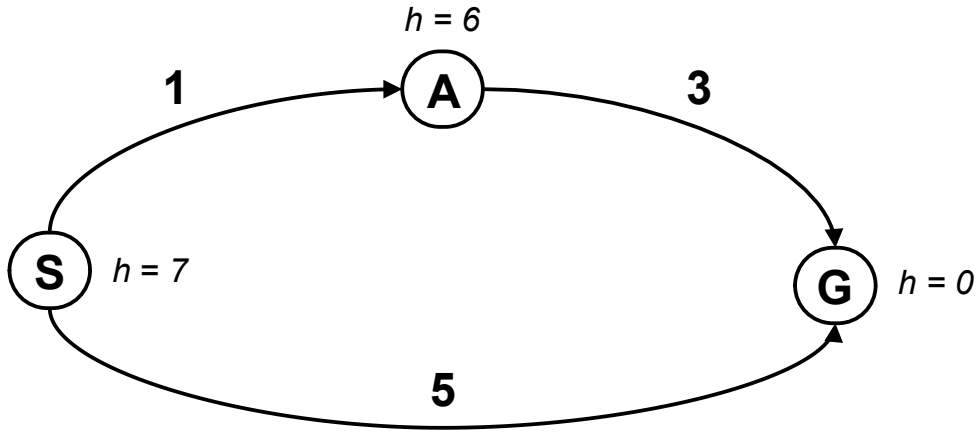
## Notes

1.  $S$ 
  - $f(S) = g(S) + h(S) = 0 + 3 = 3$
  - expanding/popping this one and crossing out (removing from frontier)
2.  $S \rightarrow A$ 
  - $f(A) = g(A) + h(A) = 2 + 2 = 4$
3.  $S \rightarrow B$ 
  - $f(B) = g(B) + h(B) = 2 + 1 = 3$
  - expanding this one and crossing out
4.  $S \rightarrow B \rightarrow G$ 
  - $f(G) = g(G) + h(G) = 5 + 0 = 5$
  - Should I stop now? No. Pop  $S \rightarrow A$  with  $f = 4$ .
5.  $S \rightarrow A \rightarrow G$ 
  - $f(G) = g(G) + h(G) = 4 + 0 = 4$
  - This is now cheapest on the frontier. I pop/expand and I'm done.

Note:  $h$  is a function of the state.  $g$  is a function of a node (the path matters).



# Is A\* optimal?



2

What is the problem?

<sup>2</sup>Graph example: Dan Klein and Pieter Abbeel

17 / 26

## Notes

Try to answer the question before going to the next slide.

1. S

-  $f(S) = g(S) + h(S) = 0 + 7 = 7$

- expanding/popping this one and crossing out (removing from frontier)

2.  $S \rightarrow A$

-  $f(A) = g(A) + h(A) = 1 + 6 = 7$

3.  $S \rightarrow G$

-  $f(G) = g(G) + h(G) = 5 + 0 = 5$

- This is now cheapest on the frontier. I pop/expand and I'm done.

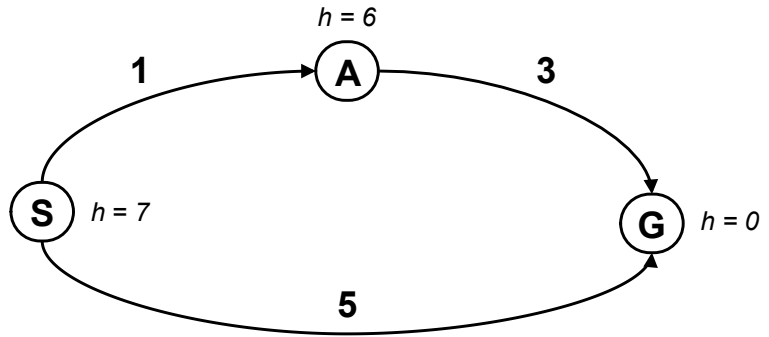
Oops! That's not cheapest! What went wrong?

What follows – keep for next slide. Problem with  $h(A) = 6$ . Overestimating the expense. (Same problem for  $h(S)$ .)

Estimates need to be  $\leq$  actual costs.

# What is the right $h(A)$ ?

- A:  $0 \leq h(A) \leq 4$
- B:  $h(A) \leq 3$
- C:  $0 \leq h(A) \leq 3$
- D:  $0 \leq h(A)$



---

## Notes

$h(A) \leq 3$  it means less than the actual cost of going from  $A$  to goal. Heuristic must not be overly pessimistic. B is correct.

Negative  $h(n)$  does not break the admissibility property but  $h(\text{Goal}) = 0$  must be kept, always. Optimality of *tree-search*  $A^*$  is also kept if the termination waits till taking the Goal node from frontier out. We will discuss the *graph-search* version later.

For a discussion, see, e.g.

<https://stackoverflow.com/questions/30067813/are-heuristic-functions-that-produce-negative-values-inadmissible>

# Admissible heuristics

A heuristic function  $h$  is admissible if:

$$\begin{aligned}h(n) &\leq \text{cost}(n.\text{state}, \text{Goal}_{\text{nearest}}) \\h(\text{Goal}) &= 0\end{aligned}$$

---

## Notes

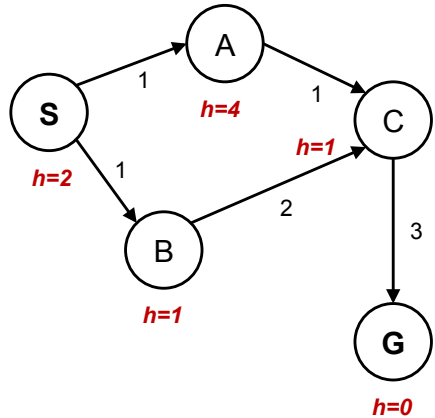
Even if negative heuristic value is allowed on the way to goal, does it make sense? How would you interpret  $h(n) = 0$ ? Is it a meaningful minimum? Why?

# Optimality of A\* tree search

A\* is optimal if  $h(n)$  is admissible.

# A\* graph search

```
function GRAPH_SEARCH(env)
  frontier.insert(startnode)
  explored ← set()
  while frontier do
    node ← frontier.pop()
    if goal in node then return node
    end if
    child_nodes ← env.expand(node.state)
    explored.add(node.state)
    for all child_nodes do
      if child_node.state not in explored then
        frontier.insert(child_node)
      end if
    end for
  end while
end function
```



What went wrong?

Graph example: Dan Klein and Pieter Abbeel.

21 / 26

## Notes

- $f(S) = g(S) + h(S) = 0 + 2 = 2$   
expanding/popping this one and crossing out (removing from frontier); *explored set*: S
- $S \rightarrow A$ ;  $f(A) = g(A) + h(A) = 1 + 4 = 5$
- $S \rightarrow B$ ;  $f(B) = g(B) + h(B) = 1 + 1 = 2$
- B is cheapest on the frontier. Expanding and removing from frontier; *explored set*: S, B
- $B \rightarrow C$ ;  $f(C) = g(C) + h(C) = 3 + 1 = 4$
- C is cheapest on the frontier. Expanding and removing from frontier; *explored set*: S, B, C
- $C \rightarrow G$ ;  $f(G) = g(G) + h(G) = 6 + 0 = 6$
- A is cheapest on the frontier. Expanding and removing from frontier; *explored set*: S, A, B, C
- $A \rightarrow C$ ;  $f(C) = f(C) + h(C) = 2 + 1 = 3$
- C is cheapest on the frontier. But, it's on *explored set*! Can't be expanded.
- Moving on to G, expanding and finishing.

Oops! That's not cheapest!  $cost(S \rightarrow B \rightarrow C \rightarrow G) = 6$ ;  $cost(S \rightarrow A \rightarrow C \rightarrow G) = 5$  What went wrong?

# What would be the proper $h(A)$ ?

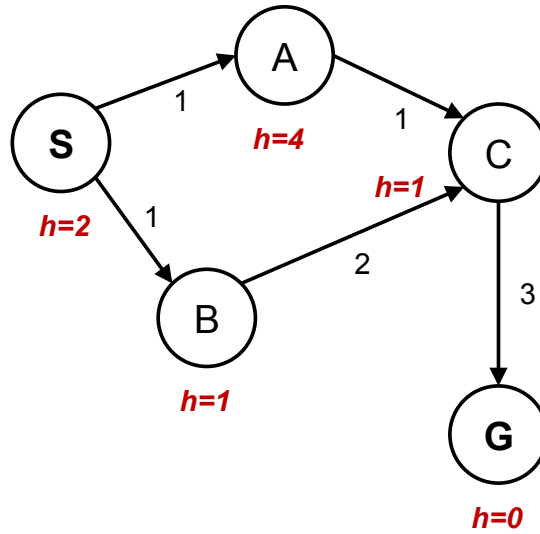
Consider other  $h(s)$  fixed.

A:  $h(A) = 1$

B:  $h(A) = 2$

C:  $1 \leq h(A) \leq 2$

D:  $0 \leq h(A) \leq 1$

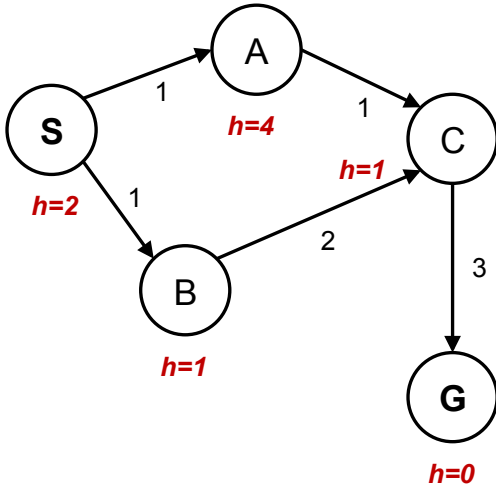


## Notes

As it will be explained in the next slides:  $h(A) \leq c(A, C) + h(C) = 2$

$h(S) \leq c(S, A) + h(A)$  it means  $h(A) \geq h(S) - c(A, S) = 1$

# Consistent heuristics



Admissible  $h$ :  
 $h(A) \leq \text{true cost } A \rightarrow G$

Consistent  $h$ :  
 $h(A) - h(C) \leq \text{true cost } A \rightarrow C$   
in general:  
 $h(n) - h(s) \leq \text{true cost } n \rightarrow s$  for any pair: node  $n$  and its successor  $s$

$f(n) = g(n) + h(n)$  along a path never decreases!

## Notes

Our heuristic was admissible.  
With *tree search* it would have worked. It would have expanded  $C$  and found the alternative, cheaper path.  
For graph search, the problem is the  $A \rightarrow C \rightarrow G$  subgraph where the *consistent* heuristic condition is violated.  
The general condition means we have two constraints for  $(A)$  for this particular graph:

$$h(S) - h(A) \leq c(S, A)$$
$$h(A) - h(C) \leq c(A, C)$$

# Optimality of A\*

- ▶ admissible  $h$  for tree search
- ▶ consistent  $h$  for graph search
- ▶ Are all consistent heuristics also admissible?  
 $h(A) - h(C) \leq \text{cost}(A \rightarrow C)$

---

## Notes

Yes, all consistent heuristics are also admissible. Btw., it is not easy to invent a heuristics that is admissible but not consistent.



# Summary

- ▶ Graph vs Tree search – avoid repeating expansions
- ▶ Effectiveness – adding heuristic estimates of cost-to-go
- ▶ Not all heuristics are equally good (admissibility, consistence, informativeness)

## References, further reading

Some figures from [2]. Chapter 2 in [1] provides a compact/dense intro into search algorithms. (State space) Search algorithms are ubiquitous, explanations in many (text)books about Algorithms.

Nice online course from UC Berkeley (CS 188 Intro to AI):

[http://ai.berkeley.edu/lecture\\_videos.html](http://ai.berkeley.edu/lecture_videos.html) Lecture: Informed Search.

[1] Steven M. LaValle.

*Planning Algorithms.*

Cambridge, 1st edition, 2006.

Online version available at: <http://planning.cs.uiuc.edu>.

[2] Stuart Russell and Peter Norvig.

*Artificial Intelligence: A Modern Approach.*

Prentice Hall, 3rd edition, 2010.

<http://aima.cs.berkeley.edu/>.