

Problem solving by search II

Tomáš Svoboda and Matěj Hoffmann

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

May 18, 2020

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	

Tree search the maze

function TREE_SEARCH(env) **return** a solution or failure

initialize the frontier

while frontier **do**

node = frontier.pop()

if goal in node **then**

break

end if

nodes = env.expand(node.state)

Add 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	

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()

if goal in node **then** break

end if

nodes = env.expand(node.state)

add node.state to explored

for all nodes **do**

if node.state not in explored (or in frontier) then

add nodes to frontier

end if

end for

end while

end function

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()

if goal in node **then** break

end if

nodes = env.expand(node.state)

add node.state to explored

for all nodes do

if node.state not in explored (or in frontier) then

add nodes to frontier

end if

end for

end while

end function

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()

if goal in node **then** break

end if

nodes = env.expand(node.state)

add node.state to explored

for all nodes **do**

if node.state not in explored (or in frontier) then

add nodes to frontier

end if

end for

end while

end function

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()
    if goal in node then break
    end if
    nodes = env.expand(node.state)
    add node.state to explored
    for all nodes do
      if node.state not in explored (or in frontier) then
        add nodes to frontier
      end if
    end for
  end while
end function
```


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()
    if goal in node then break
    end if
    nodes = env.expand(node.state)
    add node.state to explored
    for all nodes do
      if node.state not in explored (or in frontier) then
        add nodes to frontier
      end if
    end for
  end while
end function
```

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()
    if goal in node then break
    end if
    nodes = env.expand(node.state)
    add node.state to explored
    for all nodes do
      if node.state not in explored (or in frontier) then
        add nodes to frontier
      end if
    end for
  end while
end function
```

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)

▷ adding state not node!

child_nodes ← env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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

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)

child_nodes ← env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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.

▷ adding state not node!

The BFS graph search

function BFS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow FIFOqueue(node)

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

explored.add(node.state)

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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.

▷ adding state not node!

The BFS graph search

function BFS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow FIFOqueue(node)

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

explored.add(node.state)

▷ adding state not node!

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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.

The BFS graph search

function BFS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow FIFOqueue(node)

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

explored.add(node.state)

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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.

▷ adding state not node!

The BFS graph search

function BFS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow FIFOqueue(node)

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

explored.add(node.state)

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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

▷ adding state not node!

The BFS graph search

function BFS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow FIFOqueue(node)

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

explored.add(node.state)

▷ adding state not node!

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or in frontier **then**

if child_node contains Goal **then return** child_node

end if

frontier.insert(child_node)

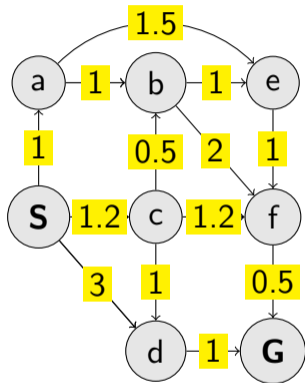
end if

end for

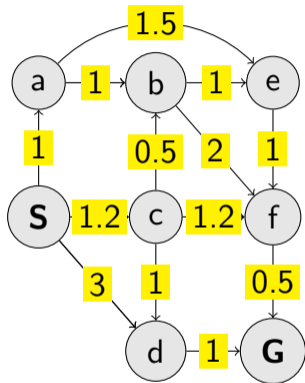
end while

end function

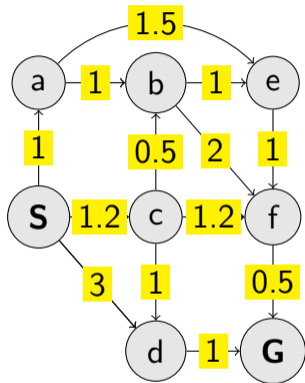
What about actual costs graph search?



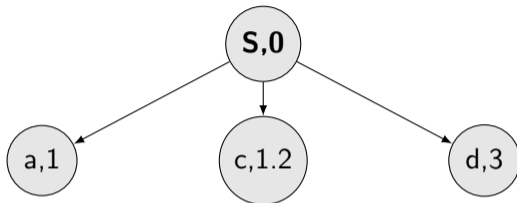
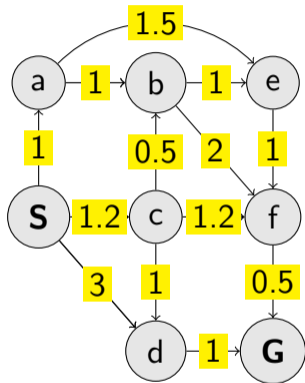
What about actual costs graph search?



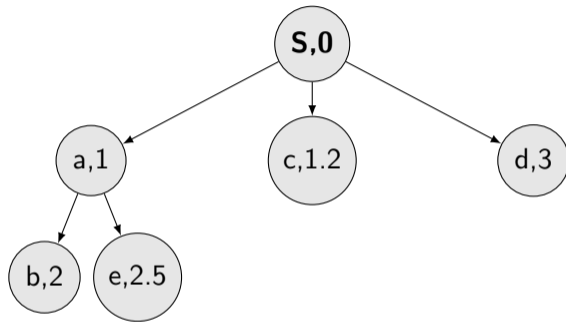
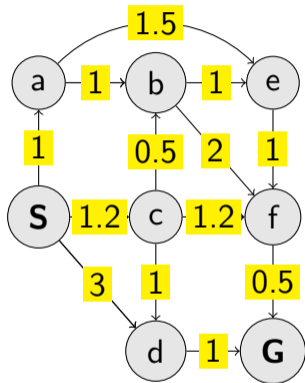
What about actual costs graph search?



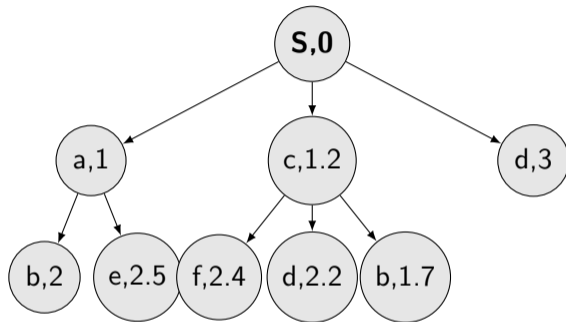
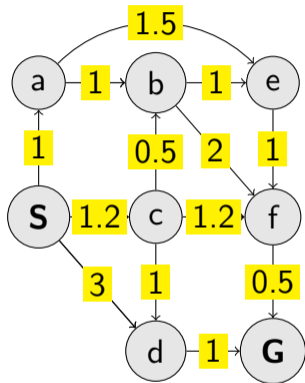
What about actual costs graph search?



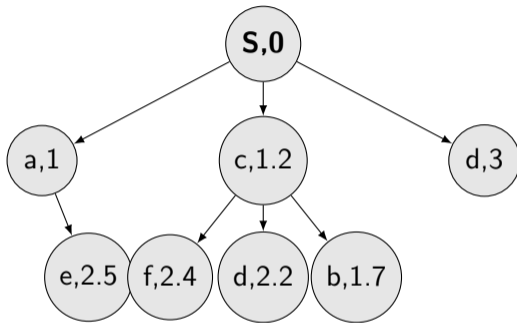
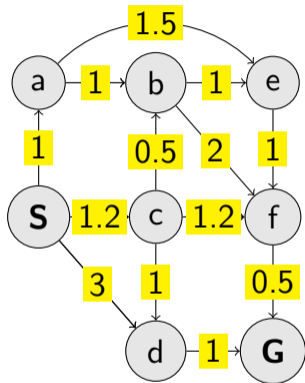
What about actual costs graph search?



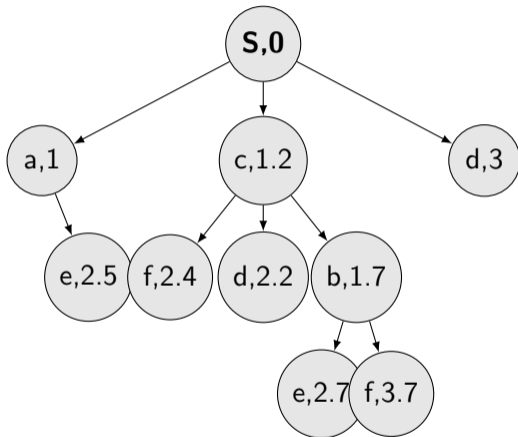
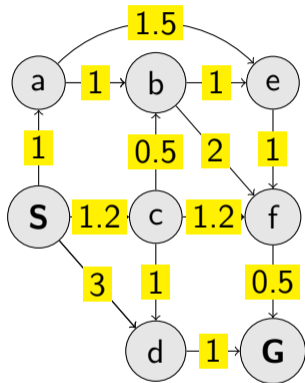
What about actual costs graph search?



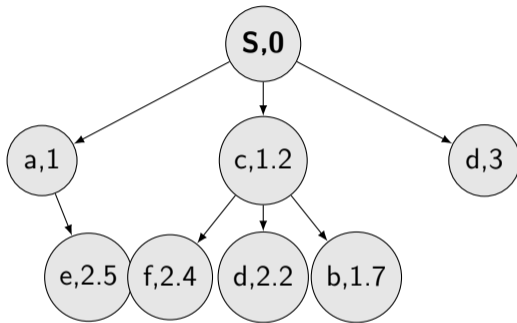
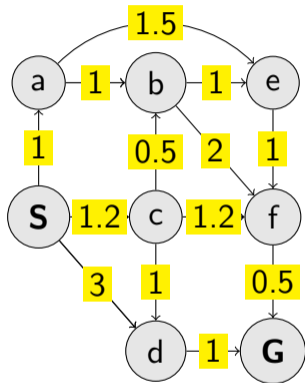
What about actual costs graph search?



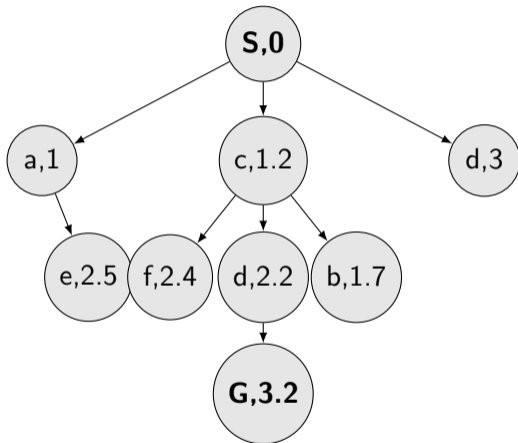
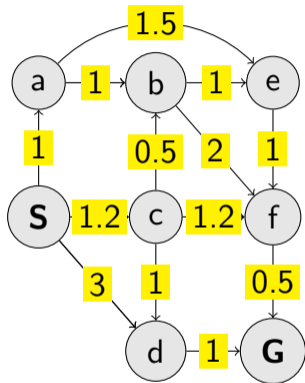
What about actual costs graph search?



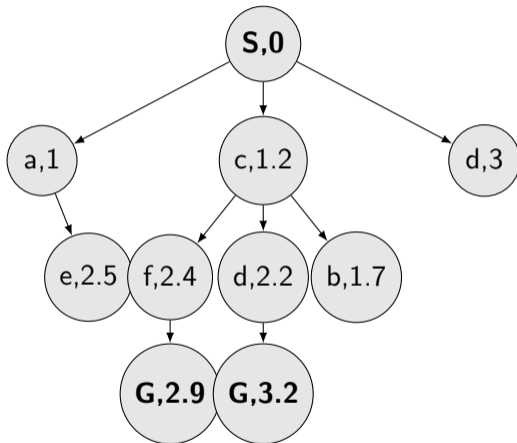
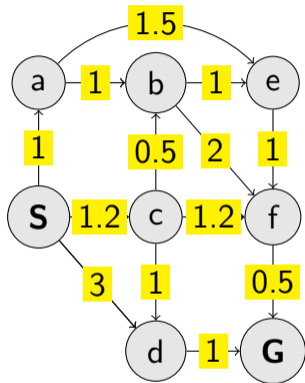
What about actual costs graph search?



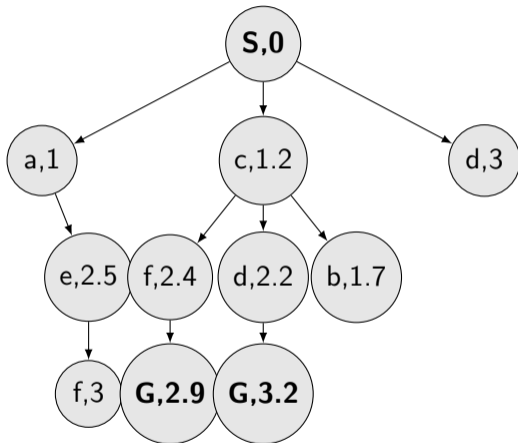
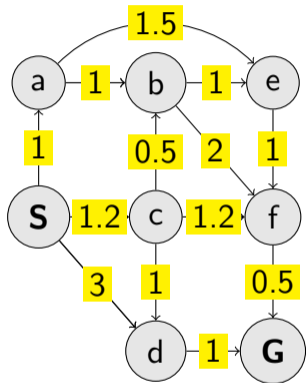
What about actual costs graph search?



What about actual costs graph search?



What about actual costs graph search?



The UCS graph search

function UCS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow priority_queue(node)

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

if node contains Goal **then return** node

end if

explored.add(node.state)

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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

▷ path_cost for ordering

▷ check here!

The UCS graph search

function UCS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow priority_queue(node)

▷ path_cost for ordering

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

if node contains Goal **then return** node

▷ check here!

end if

explored.add(node.state)

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or 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

The UCS graph search

function UCS_GRAPH_SEARCH(env) **return** a solution or failure

node ← env.observe()

frontier ← priority_queue(node)

▷ path_cost for ordering

explored ← set()

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 or 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

The UCS graph search

function UCS_GRAPH_SEARCH(env) **return** a solution or failure

node ← env.observe()

frontier ← priority_queue(node)

▷ path_cost for ordering

explored ← set()

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 or 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

The UCS graph search

function UCS_GRAPH_SEARCH(env) **return** a solution or failure

node \leftarrow env.observe()

frontier \leftarrow priority_queue(node)

▷ path_cost for ordering

explored \leftarrow set()

while frontier not empty **do**

node \leftarrow frontier.pop()

if node contains Goal **then return** node

▷ check here!

end if

explored.add(node.state)

child_nodes \leftarrow env.expand(node.state)

for all child_nodes **do**

if child_node.state not in explored or in frontier **then**

frontier.insert(child_node)

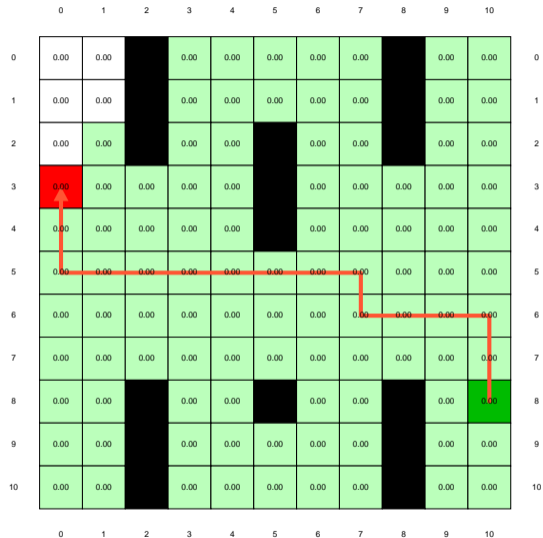
else if child_node.state in frontier with higher cost **then**

replace that node with the child_node

end if

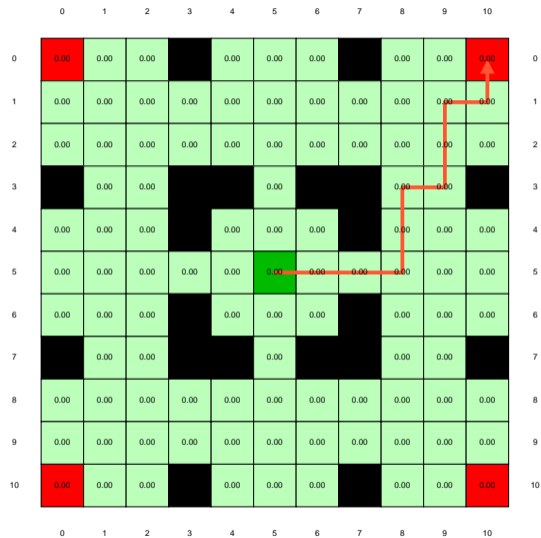
end for

Few examples of search strategies so far



Run the demos.

What is wrong with UCS and other strategies?



Run the demo.

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

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

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

- ▶ DFS: $f(n) = -n.depth$
- ▶ BFS: $f(n) = n.depth$
- ▶ UCS: $f(n) = n.path_cost$

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).

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

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

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

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

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).

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

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

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

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

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).

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

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

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

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

The good: (one) frontier as a priority queue

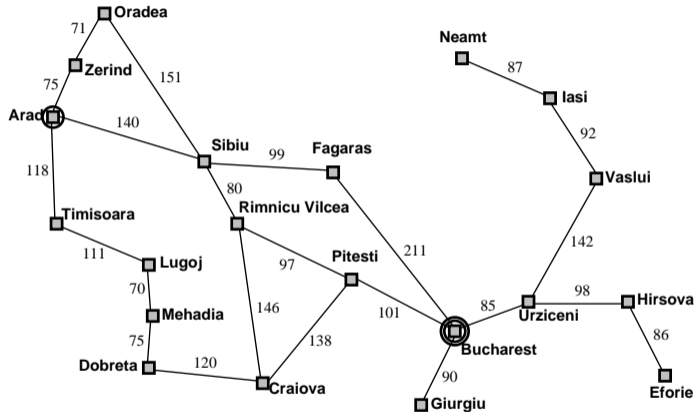
(I.e., priority queue will work universally. Still, stack (LIFO) and queue (FIFO) are (conceptually) the perfect data structures for DFS and BFS, respectively.)

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

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

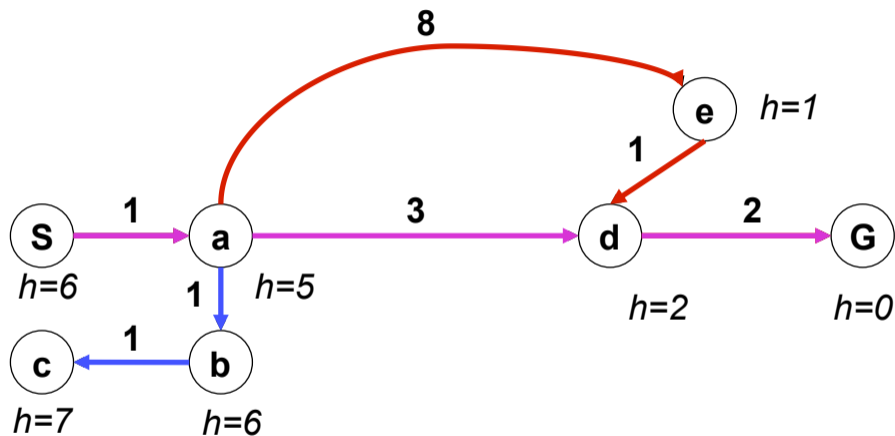
- ▶ A function that estimates how close a state is to the goal.
- ▶ Designed for a particular problem.
- ▶ We will use $h(n)$ – heuristic value of node n .

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

Greedy, take the node argmin $h(n)$

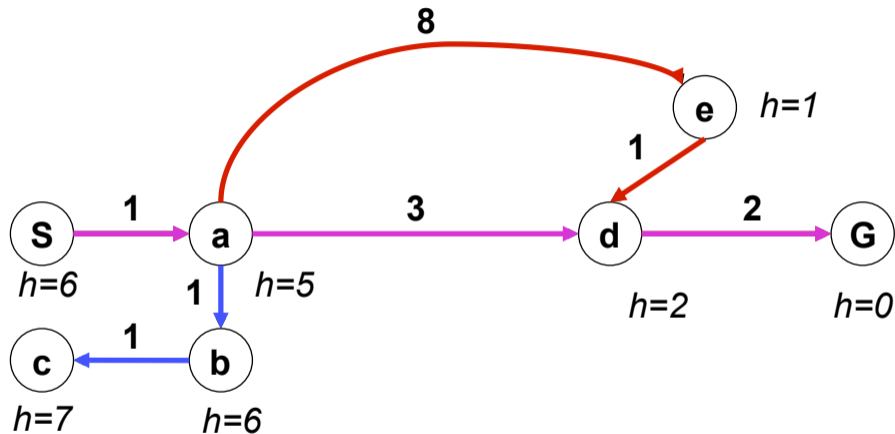


1

What is wrong (and nice) with the Greedy?

¹Graph example: Ted Grenager

Greedy, take the node argmin $h(n)$

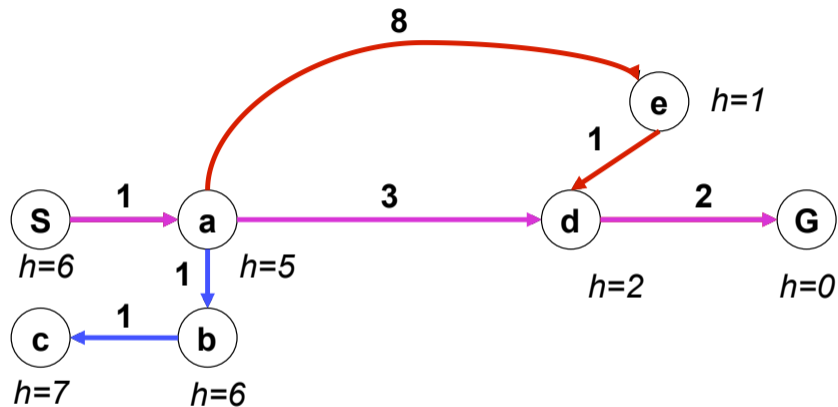


1

What is wrong (and nice) with the Greedy?

¹Graph example: Ted Grenager

A* combines UCS and Greedy

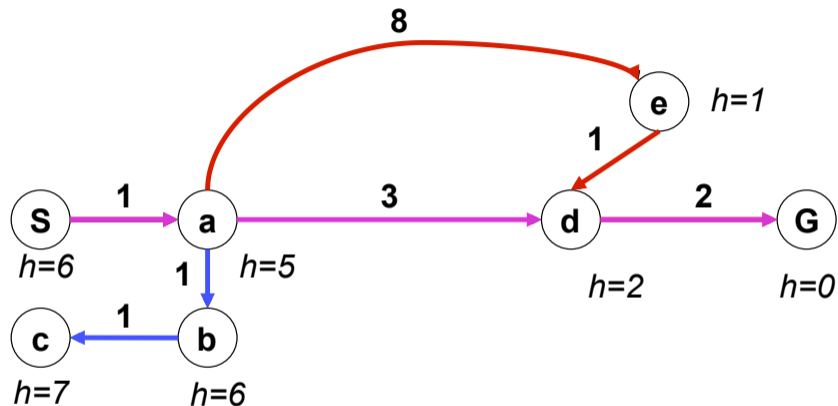


UCS orders by backward (path) cost $g(n)$

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

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

A* combines UCS and Greedy

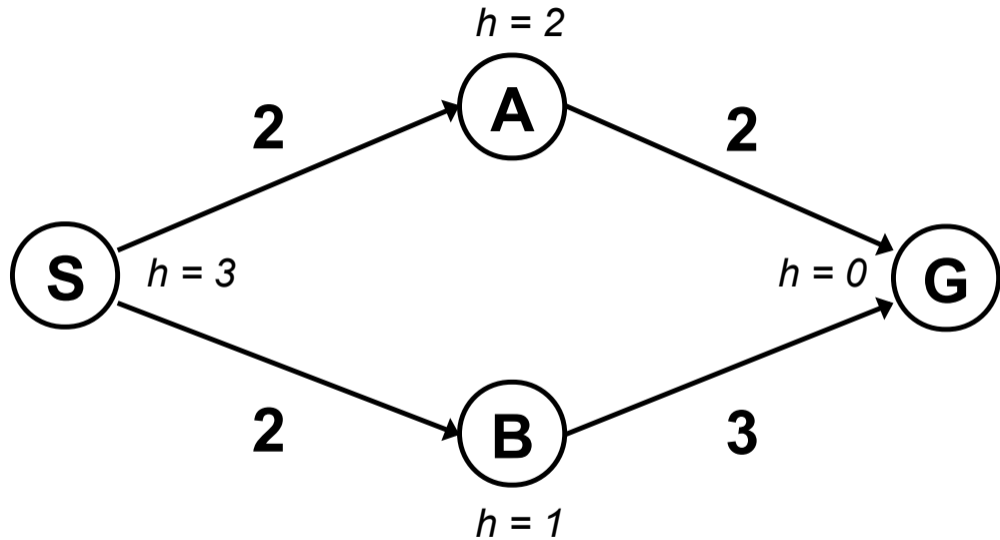


UCS orders by backward (path) cost $g(n)$

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

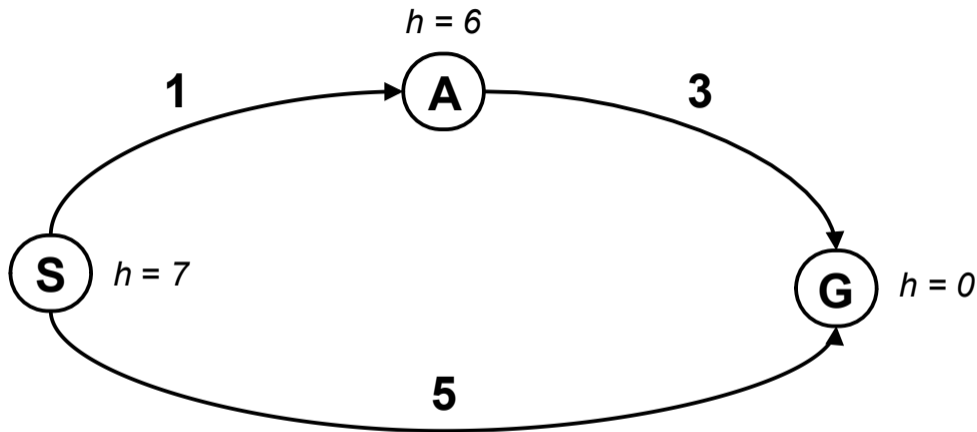
A* orders nodes by: $f(n) = g(n) + h(n)$

When to stop A*?



²Graph example: Dan Klein and Pieter Abbeel

Is A^* optimal?

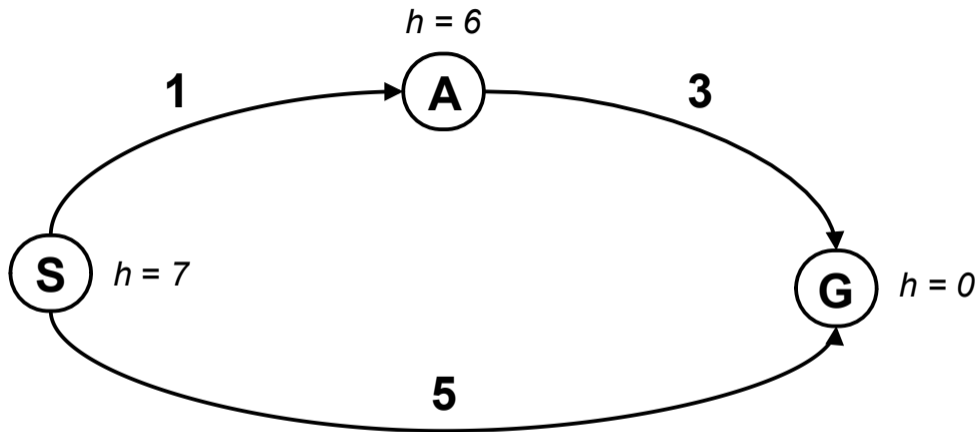


3

What is the problem?

³Graph example: Dan Klein and Pieter Abbeel

Is A^* optimal?



3

What is the problem?

³Graph example: Dan Klein and Pieter Abbeel

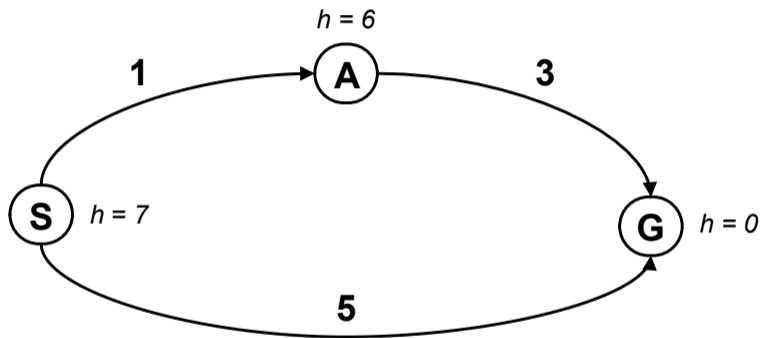
What is the right $h(A)$?

A: $0 \leq h(A) \leq 4$

B: $h(A) \leq 3$

C: $0 \leq h(A) \leq 3$

D: $0 \leq h(A)$



Admissible heuristics

A heuristic function h is admissible if:

$$h(n) \leq h^*(n)$$

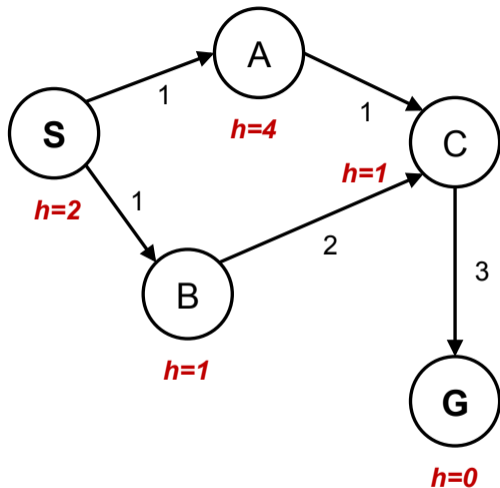
where $h^*(n)$ is the true cost of going from n to the nearest goal.

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 break
    end if
    nodes = env.expand(node.state)
    explored.add(node.state)
    for all nodes do
      if node.state not in explored then
        frontier.insert(node)
      end if
    end for
  end while
end function
```

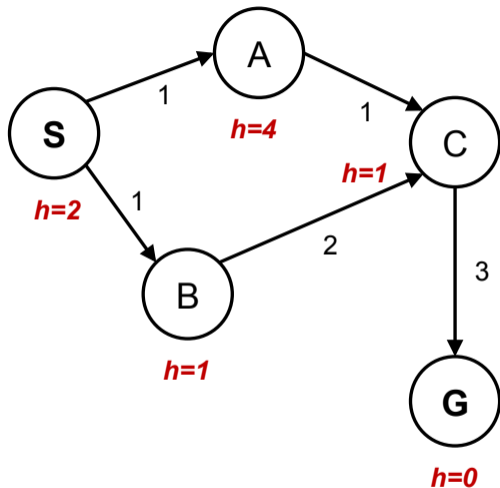


Graph example: Dan Klein and Pieter Abbeel.

What went wrong?

A* graph search

```
function GRAPH_SEARCH(env)
  frontier.insert(startnode)
  explored = set()
  while frontier do
    node = frontier.pop()
    if goal in node then break
    end if
    nodes = env.expand(node.state)
    explored.add(node.state)
    for all nodes do
      if node.state not in explored then
        frontier.insert(node)
      end if
    end for
  end while
end function
```



Graph example: Dan Klein and Pieter Abbeel.

What went wrong?

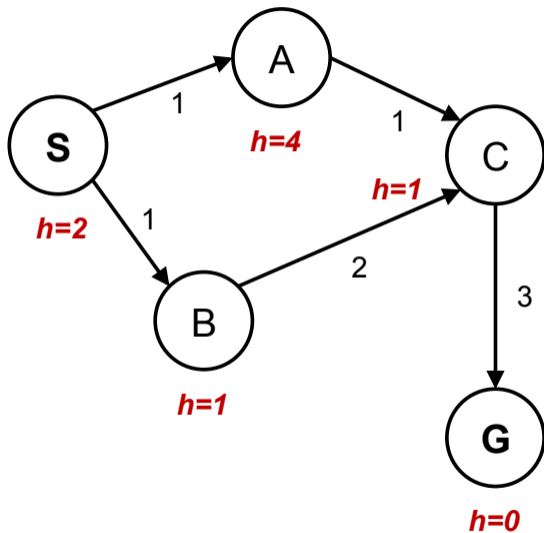
What is the proper $h(A)$?

A: $h(A) = 1$

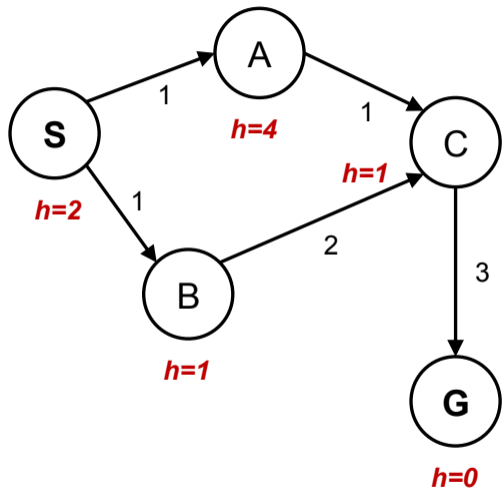
B: $h(A) = 2$

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

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



Consistent heuristics



Admissible h :

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

Consistent h :

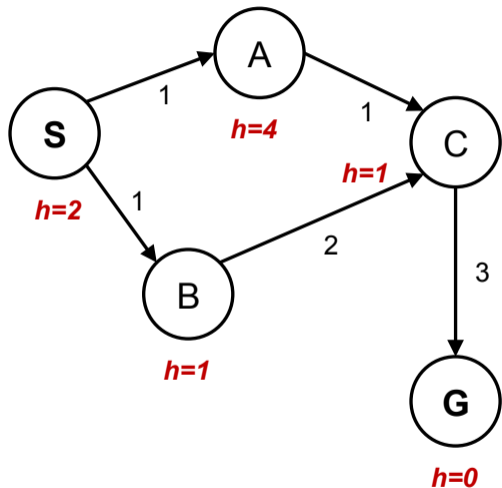
$$h(A) - h(C) \leq \text{true cost } A \rightarrow C$$

in general:

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

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

Consistent heuristics



Admissible h :

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

Consistent h :

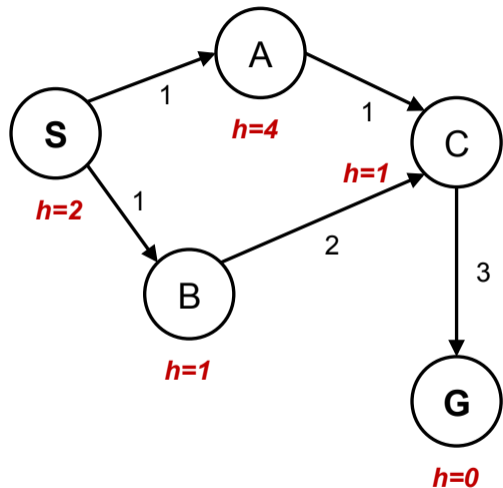
$$h(A) - h(C) \leq \text{true cost } A \rightarrow C$$

in general:

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

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

Consistent heuristics



Admissible h :

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

Consistent h :

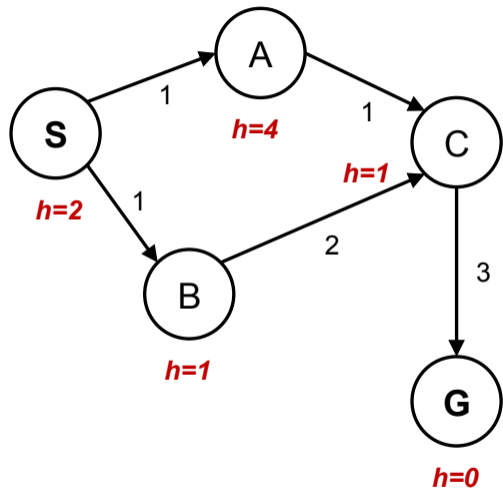
$$h(A) - h(C) \leq \text{true cost } A \rightarrow C$$

in general:

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

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

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:

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

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

Optimality of A^*

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

Optimality of A^*

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

Optimality of A^*

- ▶ admissible h for tree search
- ▶ consistent h for graph search
- ▶ What about UCS?
- ▶ Are all consistent heuristics also admissible?

$$h(A) - h(C) \leq \text{cost}(A \rightarrow C)$$

Optimality of A^*

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

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 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/>.