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

April 20, 2021

Outline

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

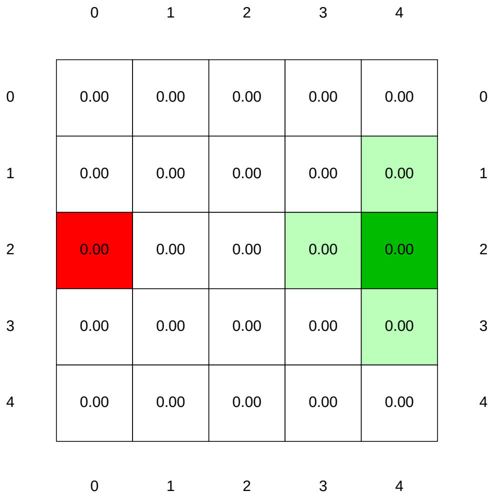
A Maze, what could possibly go wrong?

	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

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
```



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** break
    end if
    nodes = env.expand(node.state)
    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

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

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** break
        end if
        nodes = env.expand(node.state)
        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

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

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 break
        end if
        nodes = env.expand(node.state)
        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
```

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

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 break
        end if
        nodes = env.expand(node.state)
        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
```

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

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 break
        end if
        nodes = env.expand(node.state)
        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
```

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

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 break
        end if
        nodes = env.expand(node.state)
        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
```

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

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 break
        end if
        nodes = env.expand(node.state)
        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
```

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

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)                    ▷ 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)                          ▷ 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)                         ▷ 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)                          ▷ 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
```

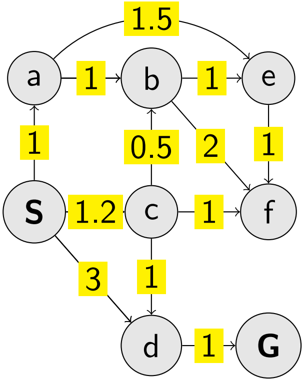
The BFS graph search

```
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)                          ▷ 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
```

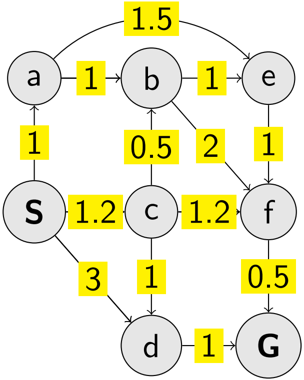
The BFS graph search

```
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)                          ▷ 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
```

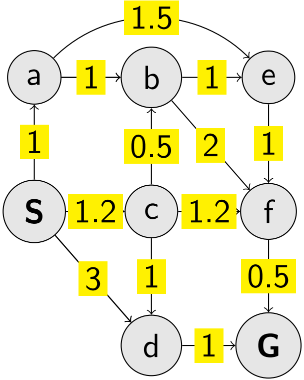
What about uniform costs graph search?



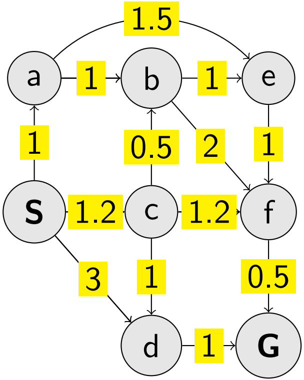
What about uniform costs graph search?

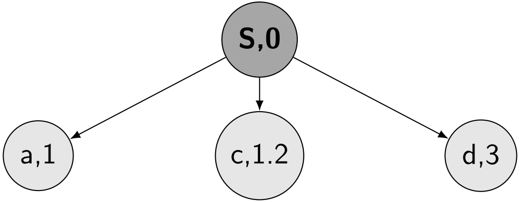


What about uniform costs graph search?

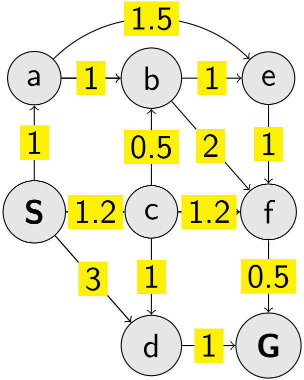


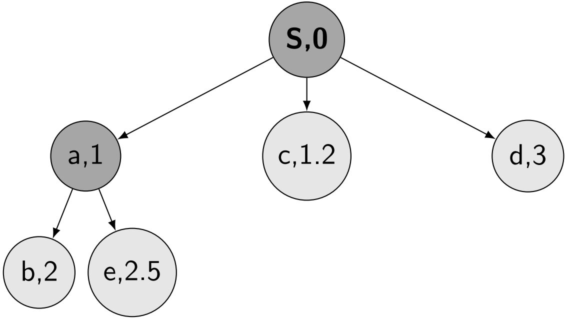
What about uniform costs graph search?



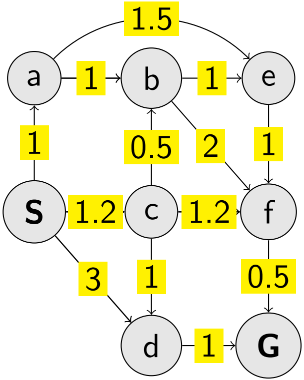


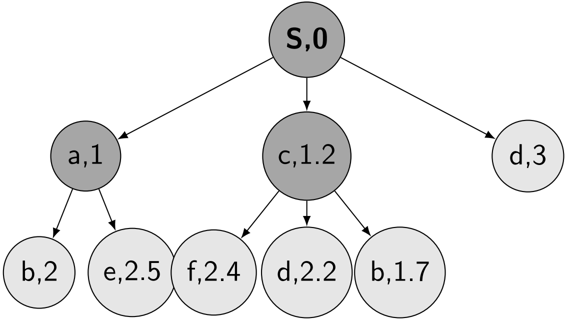
What about uniform costs graph search?



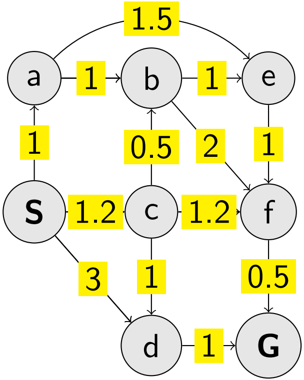


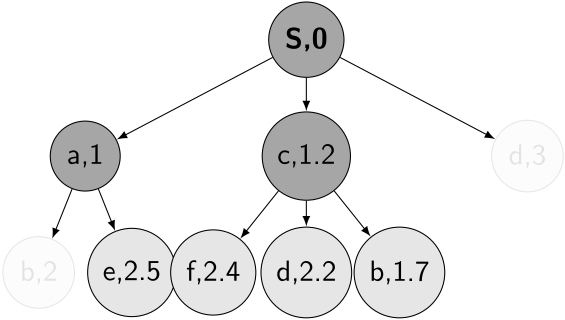
What about uniform costs graph search?



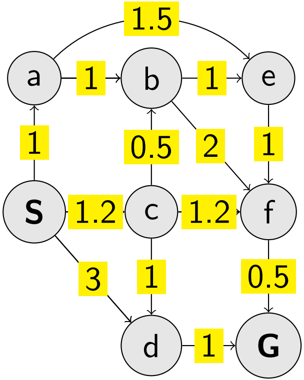


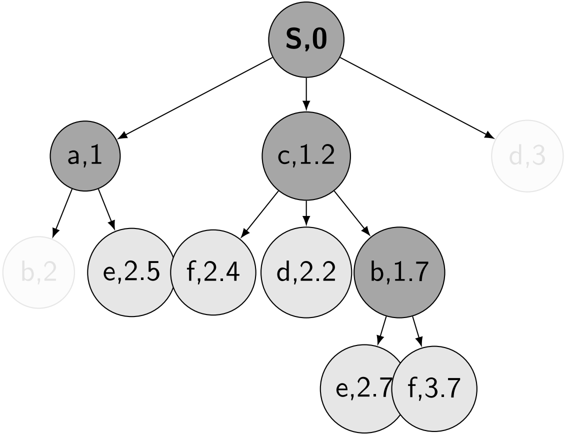
What about uniform costs graph search?



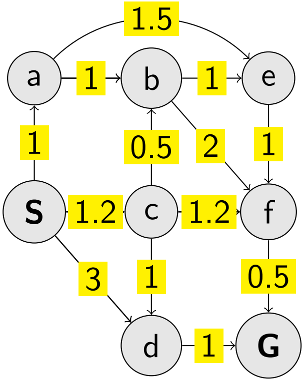


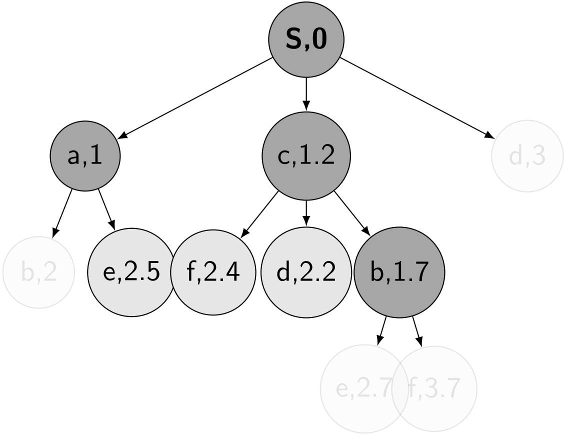
What about uniform costs graph search?



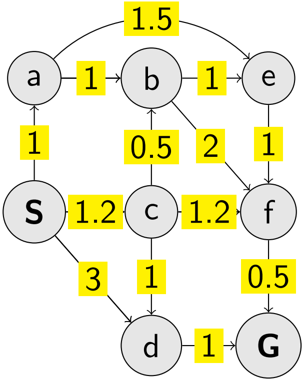


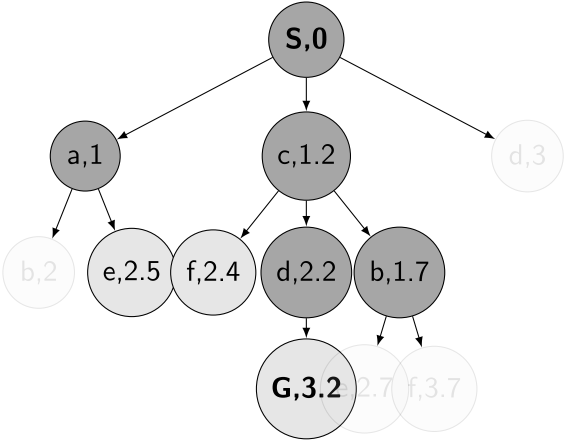
What about uniform costs graph search?



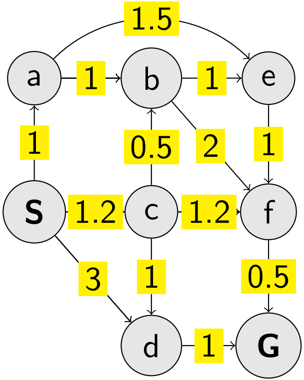


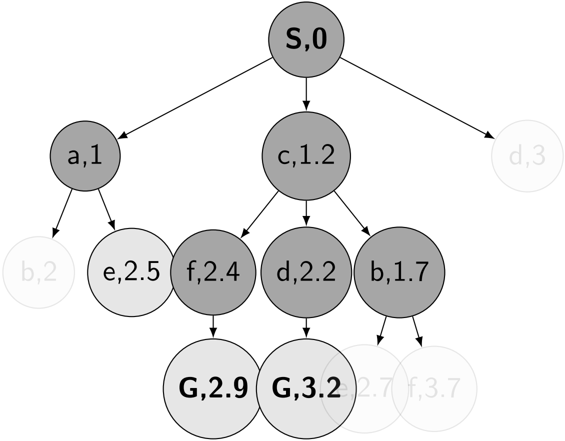
What about uniform costs graph search?



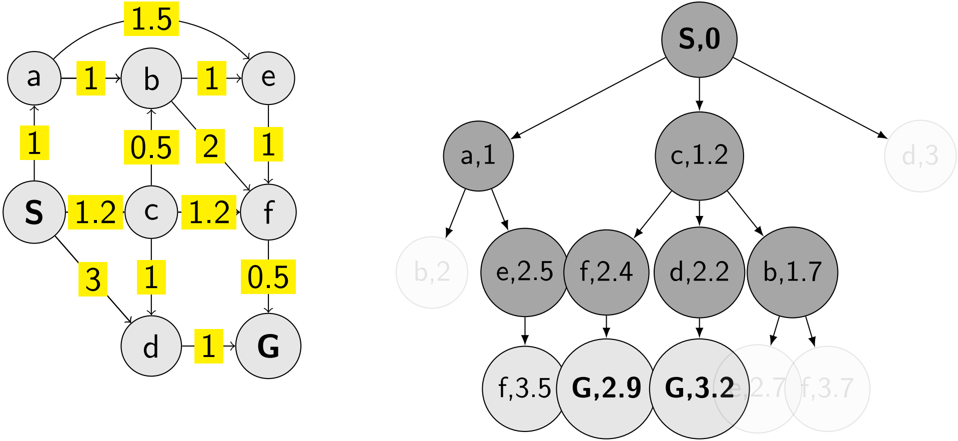


What about uniform costs graph search?

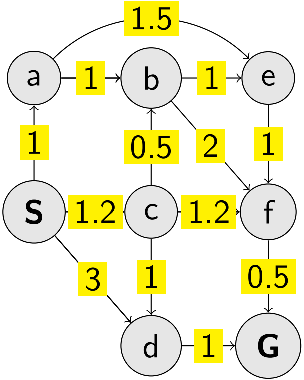


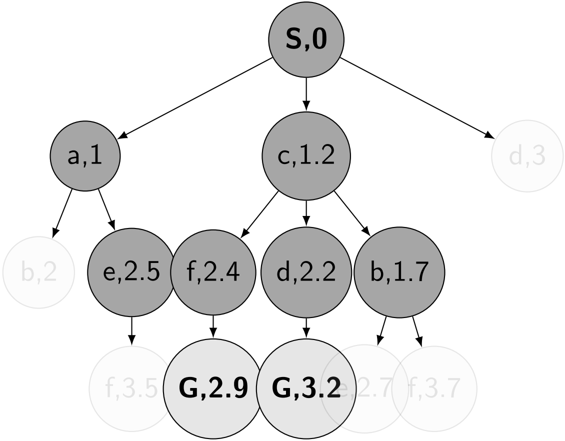


What about uniform costs graph search?

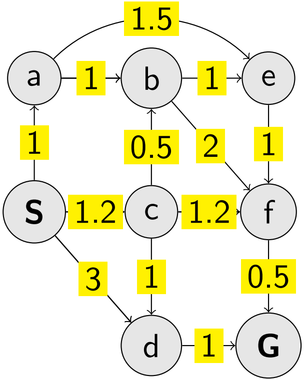


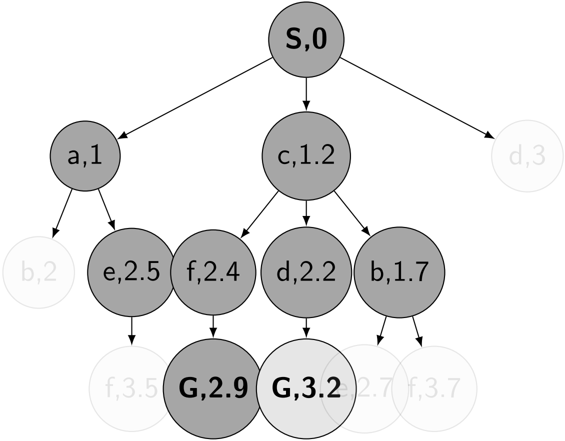
What about uniform costs graph search?





What about uniform costs graph search?





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 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
```

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 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
```

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 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
```

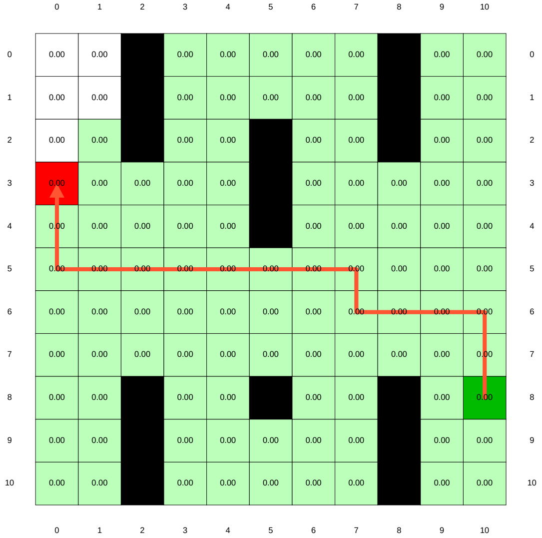
The UCS graph search

```
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 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
```

The UCS graph search

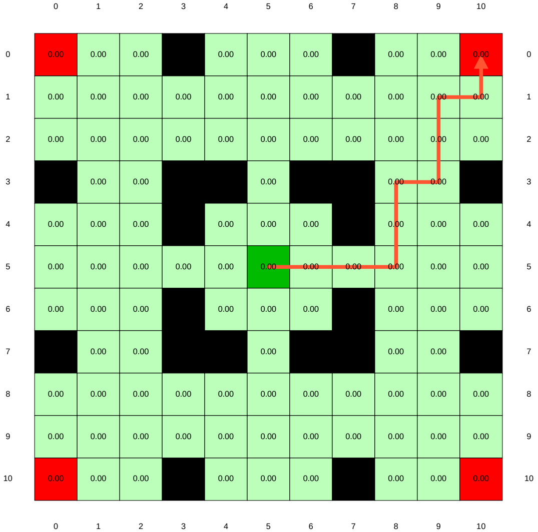
```
function UCS_GRAPH_SEARCH(env) return a solution or failure
    node ← env.observe()
    frontier ← priority_queue(node)                          ▷ path_cost for ordering
    explored ← set()
    while frontier not empty do
        node ← frontier.pop()
        if node contains Goal then return node                         ▷ 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 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
```

Few examples of search strategies so far



Run the demos.

What is wrong with UCS and other strategies?



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

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

$$\texttt{node} \leftarrow \underset{n \in \texttt{frontier}}{\operatorname{argmin}} \; f(n)$$

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

- DFS:
- BFS:
- UCS:

- $f(n) = n.path\_cost$
- $f(n) = n.depth$
- $f(n) = -n.depth$

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).

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

$$\texttt{node} \leftarrow \underset{n \in \texttt{frontier}}{\operatorname{argmin}} \; f(n)$$

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

- DFS:
- BFS:
- UCS:

- $f(n) = n.path\_cost$
- $f(n) = n.depth$
- $f(n) = -n.depth$

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).

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

$$\texttt{node} \leftarrow \underset{n \in \texttt{frontier}}{\operatorname{argmin}} \; f(n)$$

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

- DFS:
- BFS:
- UCS:

- $f(n) = n.path\_cost$
- $f(n) = n.depth$
- $f(n) = -n.depth$

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).

Node selection, take argmin $f(n)$

Selecting next node to expand/visit:

$$\texttt{node} \leftarrow \operatorname*{argmin}_{n \in \texttt{frontier}} f(n)$$

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

- DFS:
- BFS:
- UCS:

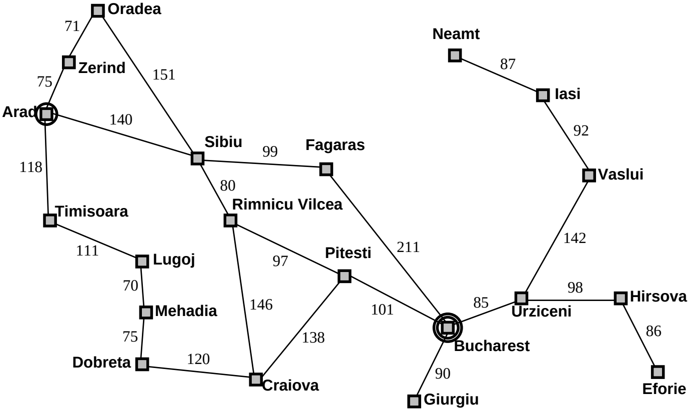
- $f(n) = n.path\_cost$
- $f(n) = n.depth$
- $f(n) = -n.depth$

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).

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

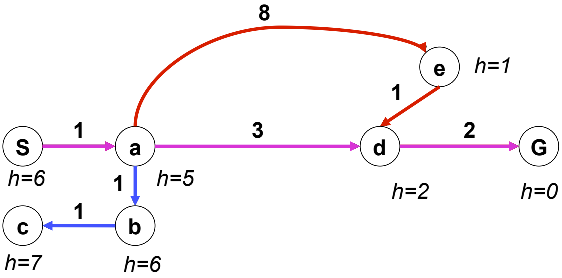
- A function that estimates how close a state is to the goal.
- Designed for a particular problem.
- We will use $h(n)$ – heuristic value of node n.

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 $h(n)$

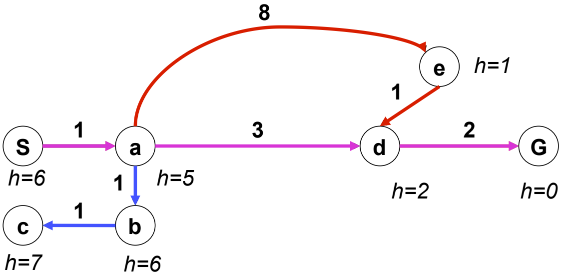


[1]

What is wrong (and nice) with the Greedy?

[1] Graph example: Ted Grenager

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

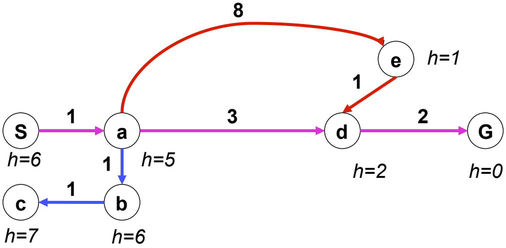


1

What is wrong (and nice) with the Greedy?

[1] Graph example: Ted Grenager

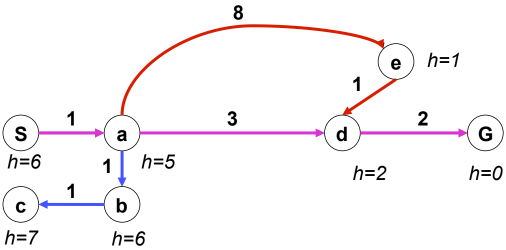
A^* combines UCS and Greedy



UCS orders by backward (path) cost $g(n)$
Greedy uses heuristics (goal proximity) $h(n)$

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

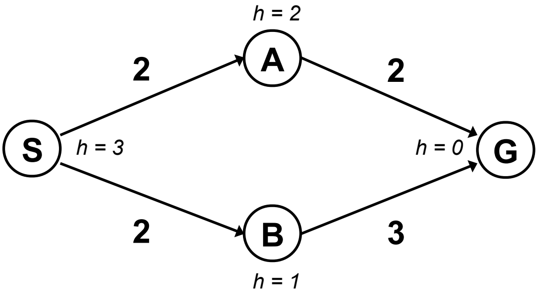
A^* combines UCS and Greedy



UCS orders by backward (path) cost $g(n)$
Greedy uses heuristics (goal proximity) $h(n)$

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

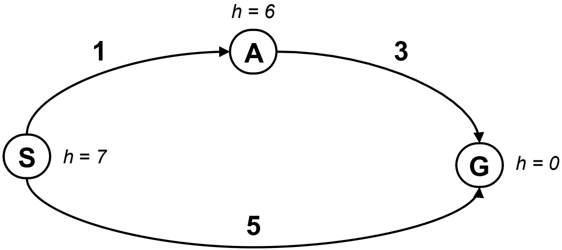
When to stop A*?



2

[2]Graph example: Dan Klein and Pieter Abbeel

Is A* optimal?

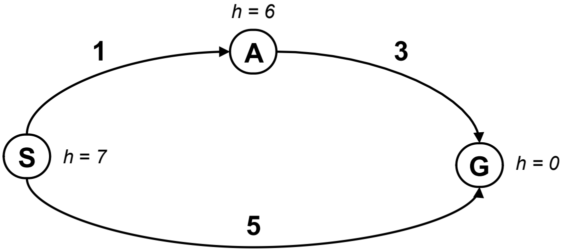


[3]

What is the problem?

[3] Graph example: Dan Klein and Pieter Abbeel

Is A* optimal?



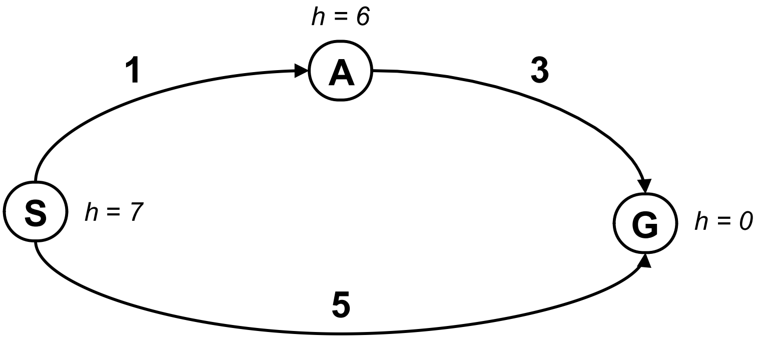
3

What is the problem?

[3]Graph 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:

$$\begin{aligned} h(n) &\leq h^*(n) \\ h(\text{Goal}) &= 0 \end{aligned}$$

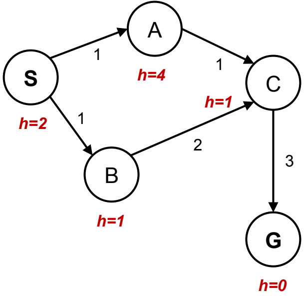
where $h^*(n)$ is the true cost of going from n to the nearest goal.

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
```

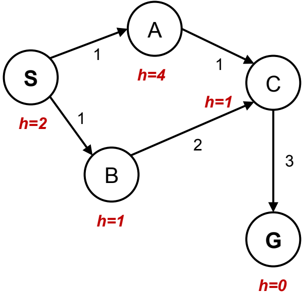


What went wrong?

Graph example: Dan Klein and Pieter Abbeel.

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
```



What went wrong?

Graph example: Dan Klein and Pieter Abbeel.

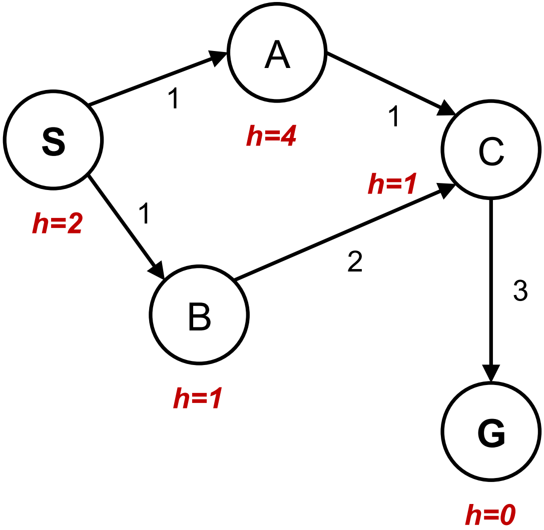
What is the proper $h(A)$?

A: $h(A) = 1$

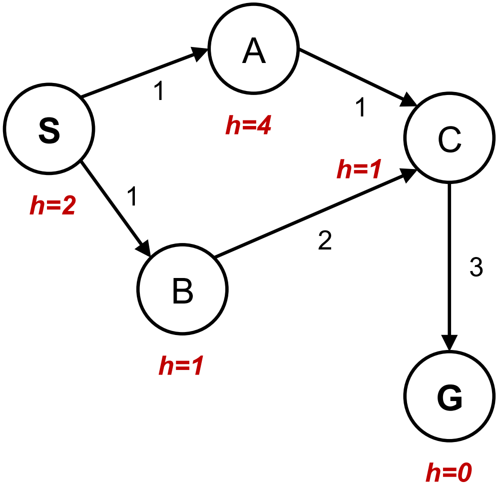
B: $h(A) = 2$

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

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



Consistent heuristics

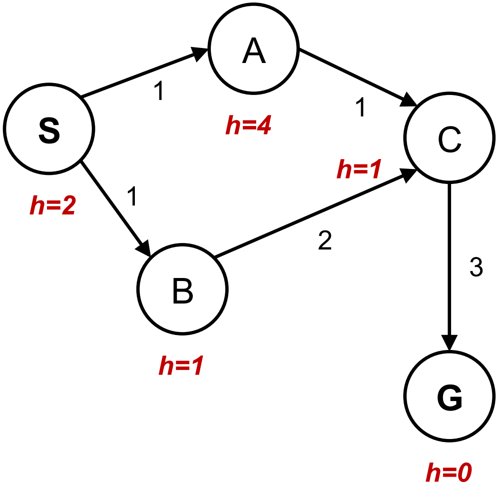


Admissible h:
$h(A) \leq$ true cost $A \rightarrow G$

Consistent h:
$h(A) - h(C) \leq$ true cost $A \rightarrow C$
in general:
$h(n) - h(s) \leq$ 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!

Consistent heuristics

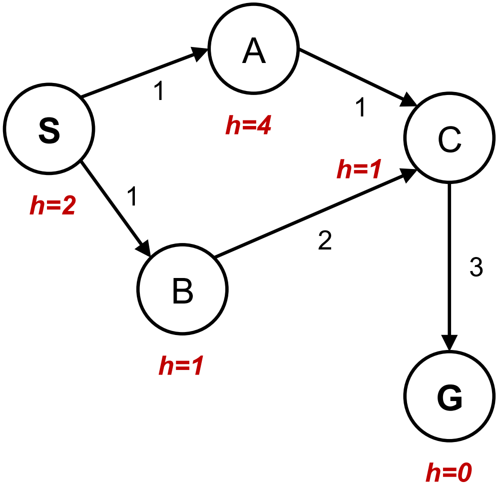


Admissible h:
$h(A) \leq$ true cost $A \rightarrow G$

Consistent h:
$h(A) - h(C) \leq$ true cost $A \rightarrow C$
in general:
$h(n) - h(s) \leq$ 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!

Consistent heuristics

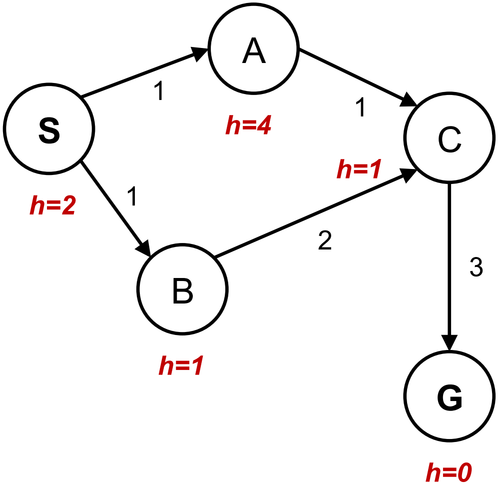


Admissible h:
$h(A) \leq$ true cost $A \rightarrow G$

Consistent h:
$h(A) - h(C) \leq$ true cost $A \rightarrow C$
in general:
$h(n) - h(s) \leq$ 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!

Consistent heuristics



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

Consistent h:
$h(A) - h(C) \leq$ true cost $A \rightarrow C$
in general:
$h(n) - h(s) \leq$ 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!

Optimality of A*

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

Optimality of A*

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

Optimality of A*

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

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

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 algoritmhs are ubiquitous, explanations in many (text)books about Algorithms.
Nice online course from UC Berkeley (CS 188 Into 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/.