Lecture 2
Solving Problems by Searching

TDT4136: Introduction to Artificial Intelligence

Xavier F. C. Sánchez Díaz

Department of Computer Science
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Techno;logy

August 25, 2025

Outline

Why searching?

Problem solving and search

- Some problems have straightforward solutions
  - Solved by applying a formula, or a well-known procedure
  - Example: differential equations
- Other problems require search:
  - no single standardised method
  - alternatives need to be explored to solve the problem
  - the number of alternatives to search among can be very large, even infinite

Why searching?

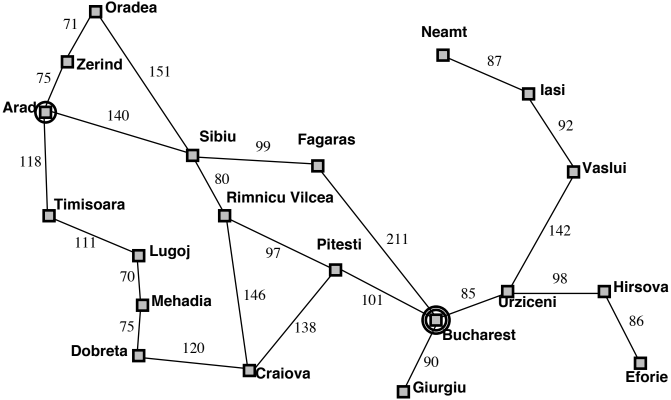
Problem solving and search

- Some problems have straightforward solutions
  - Solved by applying a formula, or a well-known procedure
  - Example: differential equations
- Other problems require search:
  - no single standardised method
  - alternatives need to be explored to solve the problem
  - the number of alternatives to search among can be very large, even infinite

This happens often in the real world, where there is a cost associated with our actions.

An example about search

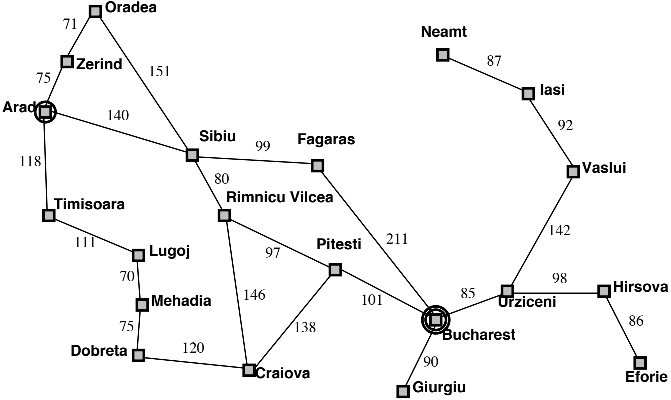
Problem solving and search



A simplified map of part of Romania, with road distances in miles.

An example about search

Problem solving and search



A simplified map of part of Romania, with road distances in miles.

Find a sequence of cities to drive through, from **Arad** to **Bucharest**.

How to solve it?

Problem solving and search

- Formulate the start and goal **states**
- What other **states** are there in the problem? What are the possible **actions** we can take?

How to solve it?

Problem solving and search

- Formulate the start and goal **states**
- What other **states** are there in the problem? What are the possible **actions** we can take?

And now, **search**:

- Simulate sequences of actions in the world to find a sequence that reaches the goal.
  - This sequence (or a subsequence) of actions is **the solution**!

How to solve it?

Problem solving and search

- Formulate the start and goal **states**
- What other **states** are there in the problem? What are the possible **actions** we can take?

And now, **search**:

- Simulate sequences of actions in the world to find a sequence that reaches the goal.
  - This sequence (or a subsequence) of actions is **the solution**!
- Execute: carry out the necessary actions in the solution, one at a time.

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map
- Initial state: *Arad*

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map
- Initial state: *Arad*
- Goal state: *Bucharest*

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map
- Initial state: *Arad*
- Goal state: *Bucharest*
- Actions: *Go from Arad to Sibiu*, and *to Timisoara*, ...

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map
- Initial state: *Arad*
- Goal state: *Bucharest*
- Actions: *Go from Arad to Sibiu*, and *to Timisoara*, ...
  - This can be translated into a transition model that can consider action costs.

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map
- Initial state: *Arad*
- Goal state: *Bucharest*
- Actions: *Go from Arad to Sibiu*, and *to Timisoara*, ...
  - This can be translated into a transition model that can consider action costs.

How to solve it?

Problem solving and search

We need to define the **state space**: all **possible states** of the environment.

- State space: all locations in the Romania map
- Initial state: *Arad*
- Goal state: *Bucharest*
- Actions: *Go from Arad to Sibiu*, and *to Timisoara*, ...
  - This can be translated into a transition model that can consider action costs.

A good problem formulation has the appropriate **level of abstraction**.

Make it formal with mathematics

Problem solving and search

The **state space** is a set:

$$S = \{Oradea, Zerind, Arad, Timisoara, Lugoj, Dobreta, \ldots\}$$

Make it formal with mathematics

Problem solving and search

The **state space** is a set:

$$S = \{Oradea, Zerind, Arad, Timisoara, Lugoj, Dobreta, \ldots\}$$

The **starting** s_0 and **goal** s_g states are elements of the **state space**: $s_0 = Arad$ and $s_g = Bucharest$

Make it formal with mathematics

Problem solving and search

The **state space** is a set:

$$S = \{Oradea, Zerind, Arad, Timisoara, Lugoj, Dobreta, \ldots\}$$

The **starting** s_0 and **goal** s_g states are elements of the **state space**: $s_0 = Arad$ and $s_g = Bucharest$

Actions can be summarised by a function A which maps a **state** (origin) to **another state** (destination): $A : S \to S$. Basically, a table (or a set, really ...)

Make it formal with mathematics

Problem solving and search

The **state space** is a set:

$$S = \{Oradea, Zerind, Arad, Timisoara, Lugoj, Dobreta, \dots\}$$

The **starting** s_0 and **goal** s_g states are elements of the **state space**: $s_0 = Arad$ and

$s_g = Bucharest$

Actions can be summarised by a function A which maps a **state** (origin) to **another state** (destination): $A : S \rightarrow S$. Basically, a table (or a set, really . . .)

The **transition model** can be another function, f, which maps **an action** to a **cost** (a number): $f : A \rightarrow \mathbb{R}$

Make it formal with mathematics

Problem solving and search

The **state space** is a set:

$$S = \{Oradea, Zerind, Arad, Timisoara, Lugoj, Dobreta, \ldots\}$$

The **starting** s_0 and **goal** s_g states are elements of the **state space**: $s_0 = Arad$ and

$s_g = Bucharest$

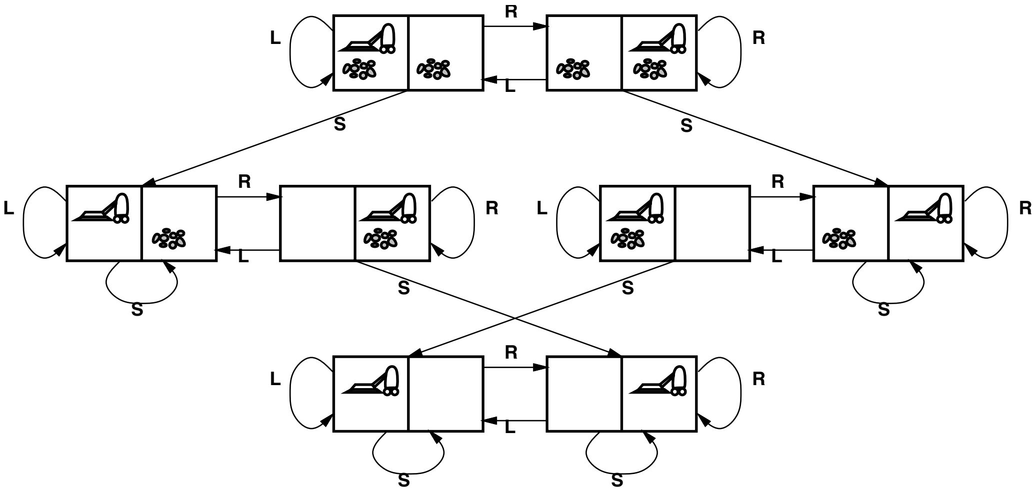
Actions can be summarised by a function A which maps a **state** (origin) to **another state** (destination): $A : S \to S$. Basically, a table (or a set, really ...)

The **transition model** can be another function, f, which maps **an action** to a **cost** (a number): $f : A \to \mathbb{R}$

Having it in mathematical terms makes it **easier to code**!

Another example: Vacuum world

Problem solving and search

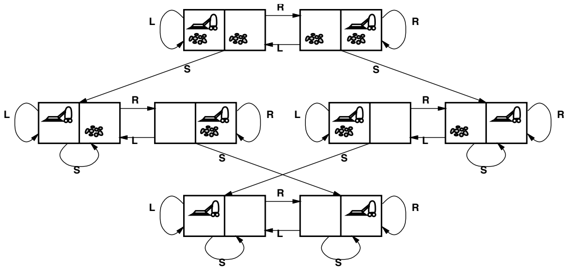


[1]

[1] p. 85 in the textbook.

Another example: Vacuum world

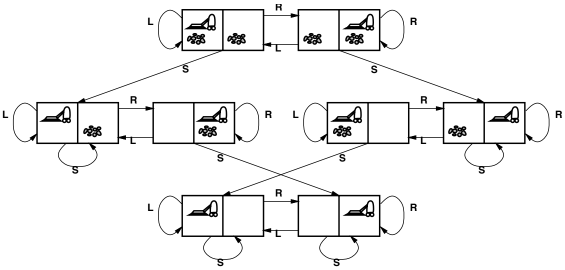
Problem solving and search



- Initial state: any state

Another example: Vacuum world

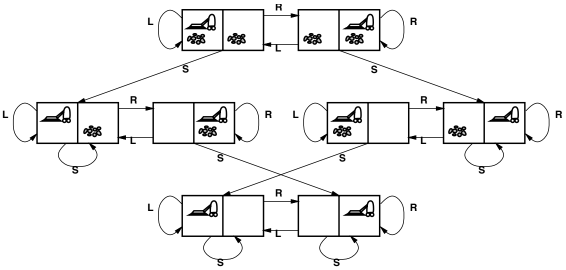
Problem solving and search



- Initial state: any state
- Goal states: no dirt

Another example: Vacuum world

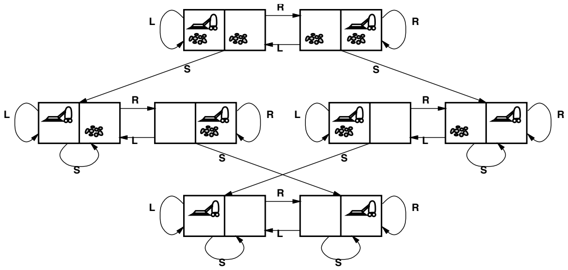
Problem solving and search



- Initial state: any state
- Goal states: no dirt
- Actions: *Suck*, *moveLeft*,*moveRight* …or *Suck*, *Forward*, *Backward*, *TurnLeft*, *TurnRight*, …

Another example: Vacuum world

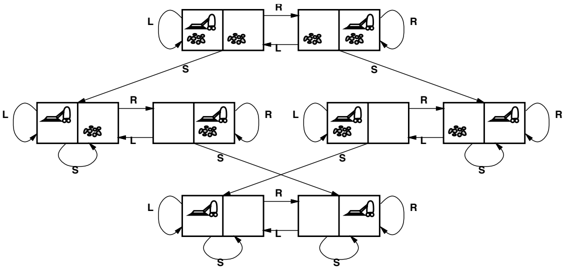
Problem solving and search



- Initial state: any state
- Goal states: no dirt
- Actions: *Suck*, *moveLeft*,*moveRight* ...or *Suck*, *Forward*, *Backward*, *TurnLeft*, *TurnRight*, ...
- Transition model: *Suck* cleans the location, *TurnLeft* changes the direction of agent 90 degrees

Another example: Vacuum world

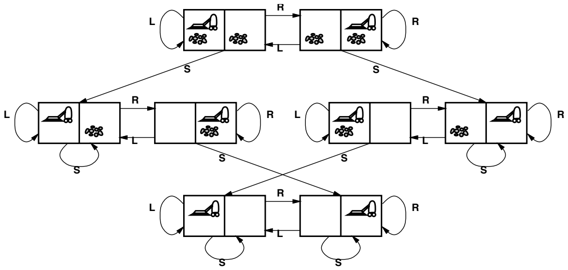
Problem solving and search



- Initial state: any state
- Goal states: no dirt
- Actions: *Suck*, *moveLeft*,*moveRight* …or *Suck*, *Forward*, *Backward*, *TurnLeft*, *TurnRight*, …
- Transition model: *Suck* cleans the location, *TurnLeft* changes the direction of agent 90 degrees
- Action cost: 1 unit

Another example: Vacuum world

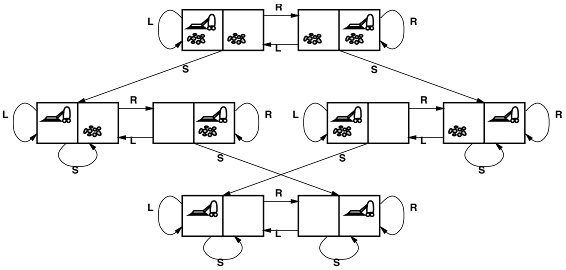
Problem solving and search



- Initial state: any state
- Goal states: no dirt
- Actions: *Suck*, *moveLeft*,*moveRight* …or *Suck*, *Forward*, *Backward*, *TurnLeft*, *TurnRight*, …
- Transition model: *Suck* cleans the location, *TurnLeft* changes the direction of agent 90 degrees
- Action cost: 1 unit

Another example: Vacuum world

Problem solving and search



- Initial state: any state
- Goal states: no dirt
- Actions: *Suck*, *moveLeft*,*moveRight* …or *Suck*, *Forward*, *Backward*, *TurnLeft*, *TurnRight*, …
- Transition model: *Suck* cleans the location, *TurnLeft* changes the direction of agent 90 degrees
- Action cost: 1 unit

Notice how we do not care much about costs here!

These kind of search problems happen all the time!

Applications

Problem solving and search

But the real world is usually **more complex**!

- Resources are limited (and **costs** become important!)

Applications

Problem solving and search

But the real world is usually **more complex**!

- Resources are limited (and **costs** become important!)
- We have constraints and **restrictions**

Applications

Problem solving and search

But the real world is usually **more complex**!

- Resources are limited (and **costs** become important!)
- We have constraints and **restrictions**
- We need to be quick and cannot freely **explore**

Applications

Problem solving and search

The Travelling Salesperson Problem: find shortest route visiting each location once and returns to initial location.

- For example: Delivery services

Applications

Problem solving and search

The Travelling Salesperson Problem: find shortest route visiting each location once and returns to initial location.

- For example: Delivery services
- The Holidays in Romania example

Applications

Problem solving and search

The Travelling Salesperson Problem: find shortest route visiting each location once and returns to initial location.

- For example: Delivery services (and you can always make it more complicated!)
  - Time windows
  - Closed roads
  - Traffic
- The Holidays in Romania example

Applications

Problem solving and search

Assembly problems: find an order for assembling the parts of some object.

- For example: Manufacturing and design (and you can always make it more complicated!)
  - Find the **optimal** order (minimum cost)
  - Reduce idle time on different machines
  - Assembly lines could be **dependent** on each other

Section 2

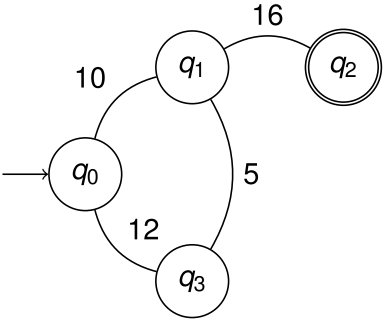
The search algorithm

What is a search algorithm?

Search algorithms

It is a **function** of the form *Search*(*PROBLEM*) that returns either a solution or failure.

- A state is a *representation of* a configuration
- Using a state space graph we can represent all possible states, and the transitions between them.

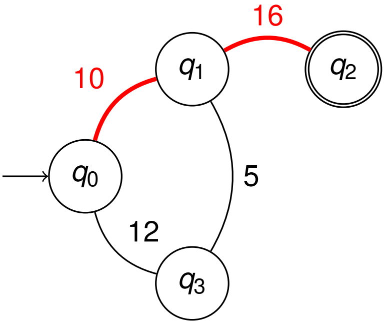


What is a search algorithm?

Search algorithms

It is a **function** of the form *Search*(*PROBLEM*) that returns either a solution or failure.

- A state is a *representation of* a configuration
- Using a state space graph we can represent all possible states, and the transitions between them.
- We can superimpose a search tree on the **space graph** and show a particular algorithm!



Exploring the state space I

Search algorithms

- Most of the time, it is not feasible (or it is too expensive) to build and represent the entire state graph.
- The problem solver agent generates a solution by **incrementally exploring** a small portion of the graph
- We simulate the exploration by generating successors of **already-explored states.**

Exploring the state space II

Search algorithms

The search procedure

1. You are standing on the initial node. What are the states to be explored from here?
2. Is any of the nodes able to be explored, the goal? If not, generate successors of a node: **expand** the node[2]
3. Add the successors nodes into the list of "to be explored".
4. Select (according to certain **criteria**) the next node to expand.

This process will be **repeated** until we either find a solution, or fail (by running out of time, of states, of resources...)

[2] Consider that each algorithm dictates when the 'goal check' is performed!

Exploring the state space

Search algorithms

The search procedure revised

You are standing on the starting node.

1. Check where you are standing: is it the goal?[3]
2. If not, then what are the nodes to be explored here?.
3. **Expand** the node you are in
4. Add the successors nodes into the **frontier**
5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

And then repeat!

[3] Consider that the 'goal check' is dependent on the algorithm!

Exploring the state space

Search algorithms

The search procedure revised

You are standing on the starting node.

1. Check where you are standing: is it the goal?[3]
2. If not, then what are the nodes to be explored here?.
3. **Expand** the node you are in
4. Add the successors nodes into the **frontier**
5. Select (according to certain **criteria**–a function f–) the next node to expand and move.

And then repeat!

[3] Consider that the ‘goal check’ is dependent on the algorithm!

What is a node?

Search algorithms

A **node** is a *representation* of a **state.** It is a data structure constituting a **part of a search tree**:

- The state of the node
- The parent of the node (or which node you came from)
- The action taken to reach the node
- The children of the node (or which states you can go to)
- The path cost of the search (so far)

Notice how a node is not a state, but **a step** in the search!

Terminology and the book I

Search algorithms

If a node is in the frontier, it does not mean it has been expanded! At least not for our book.

- The **frontier** are those nodes *I can expand*
- The *set* of **reached** nodes contains both the frontier AND the **expanded** nodes

So, formally, we know that

- $Frontier \subset Reached$, and
- $Frontier \cup Expanded = Reached$

And so, $Expanded = Reached \setminus Frontier$.

Terminology and the book II

Search algorithms

The book also uses *object-oriented programming* notation to refer to *pertaining* (or *belonging*):

- *node.STATE* is the *STATE* of *node*
- *node.PARENT* is the *PARENT* of *node*...

Operations are usually referred to as **functions**.

- *Search*(*problem*) is the *Search* procedure on the instance *problem*
- *IsEmpty*(*frontier*) is a function which returns true if the *frontier* is empty
- *Pop*(*frontier*) removes the top node of the frontier and returns it, while *Top*(*frontier*) just *peeks* at it (no removal)
- *Add*(*node*, *frontier*)...

You get the idea.

Graph properties

Search algorithms

As many other graphs, search graphs can contain **redundant paths** and **loops.** One can check the chain of parent nodes and make sure not to visit the same node twice. However, keep in mind that **coding** is very different from the theoretical analysis we will do in the course.

The performance of a search algorithm can be measured in different ways:

- **Completeness**: is the algorithm guaranteed to find a solution?
- **Optimality**: the solution quality. Is it optimal? (cheaper, faster, etc.)
- **Time complexity**: how long does the algorithm take? (in seconds, operations, expanded states...)
- **Space complexity**: how much memory do we need, for example, in the *frontier* or *reached* sets?

Section 3

Uninformed search strategies

Searching with no information

Uninformed search strategies

Recall the fifth step in the searching procedure a few slides back:

The searching procedure revised

. . . 5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

Depending on the type selection criteria and storage used, search strategies work differently!

Searching with no information

Uninformed search strategies

Recall the fifth step in the searching procedure a few slides back:

The searching procedure revised

. . . 5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

Depending on the type selection criteria and storage used, search strategies work differently!

- Breadth-first search (BFS)

Searching with no information

Uninformed search strategies

Recall the fifth step in the searching procedure a few slides back:

The searching procedure revised

. . . 5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

Depending on the type selection criteria and storage used, search strategies work differently!

- Breadth-first search (BFS)
- Depth-first search (DFS)

Searching with no information

Uninformed search strategies

Recall the fifth step in the searching procedure a few slides back:

The searching procedure revised

. . . 5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

Depending on the type selection criteria and storage used, search strategies work differently!

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Depth-limited search

Searching with no information

Uninformed search strategies

Recall the fifth step in the searching procedure a few slides back:

The searching procedure revised

. . . 5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

Depending on the type selection criteria and storage used, search strategies work differently!

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Depth-limited search
- Iterative deepening

Searching with no information

Uninformed search strategies

Recall the fifth step in the searching procedure a few slides back:

The searching procedure revised

. . . 5. Select (according to certain **criteria**–a function *f*–) the next node to expand and move.

Depending on the type selection criteria and storage used, search strategies work differently!

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Depth-limited search
- Iterative deepening
- Uniform-cost (Dijkstra)

Things to look out for

Search strategies

Implementation details vary a lot, and can be tricky!

- Is the algorithm checking for redundant paths (**graph search**) or not (**tree search**)?

Things to look out for

Search strategies

Implementation details vary a lot, and can be tricky!

- Is the algorithm checking for redundant paths (**graph search**) or not (**tree search**)?
- Is the goal check performed **early** (when a node is *generated*) or **late** (when a node is *expanded*)?

Things to look out for

Search strategies

Implementation details vary a lot, and can be tricky!

- Is the algorithm checking for redundant paths (**graph search**) or not (**tree search**)?
- Is the goal check performed **early** (when a node is *generated*) or **late** (when a node is *expanded*)?
- Is the algorithm **storing** all **reached states**, or **reconstructing the path** from a chain of parent nodes?

Things to look out for

Search strategies

Implementation details vary a lot, and can be tricky!

- Is the algorithm checking for redundant paths (**graph search**) or not (**tree search**)?
- Is the goal check performed **early** (when a node is *generated*) or **late** (when a node is *expanded*)?
- Is the algorithm **storing** all **reached states**, or **reconstructing the path** from a chain of parent nodes?

Read the book!

To become familiar with the algorithms and their implementations details, you should read the book. These slides are not a replacement for the book; they are a summary of the most important points.

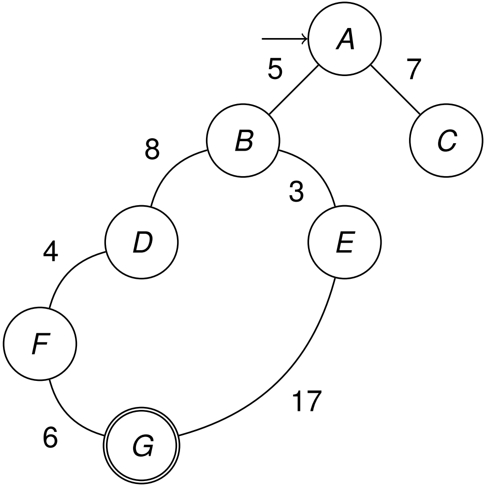
Breadth First Search

Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.



[a] in BFS, we check early for the goal!

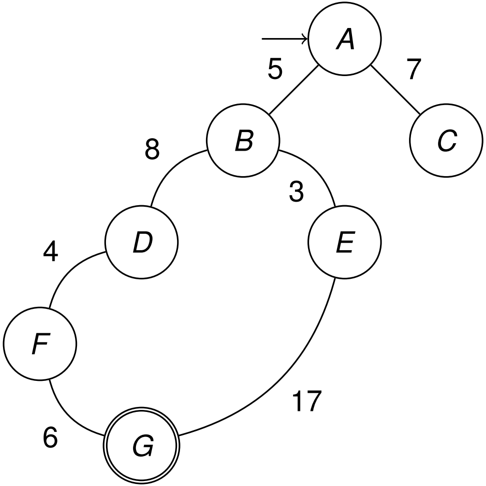
Breadth First Search

Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$



[a] in BFS, we check early for the goal!

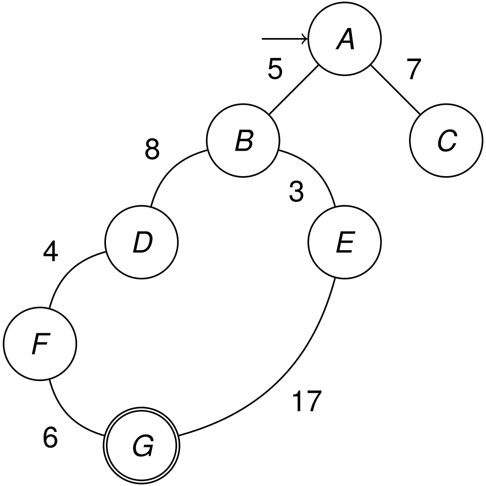
Breadth First Search

Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier*:



[a] in BFS, we check early for the goal!

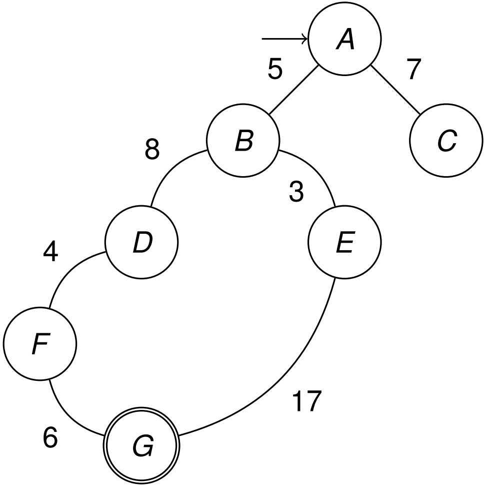
Breadth First Search

Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so *Expand*(*A*)
3. Add successors to *frontier*:
   - Is any of those the goal?[a]



[a] in BFS, we check early for the goal!

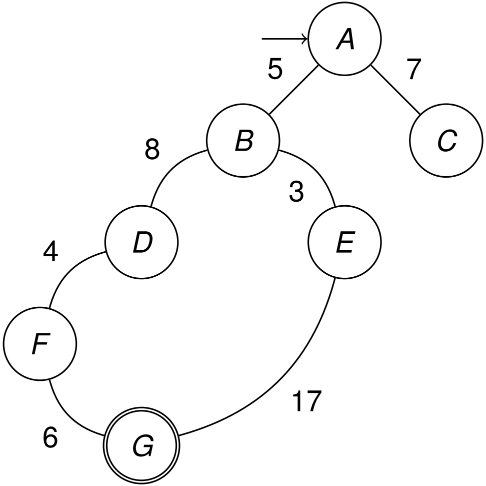
Breadth First Search

Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier*:
   - Is any of those the goal?[a]
   - No, so update *frontier* $= \langle B, C \rangle$



[a] in BFS, we check early for the goal!

Breadth First Search

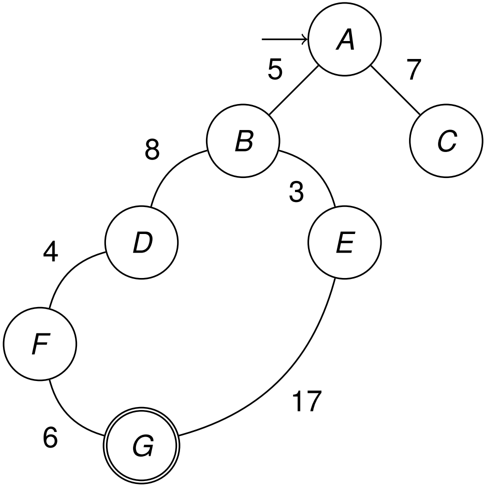
Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier*:
   - Is any of those the goal?[a]
   - No, so update *frontier* $= \langle B, C \rangle$
4. Choose first element in *frontier*.

And repeat...



[a] in BFS, we check early for the goal!

Breadth First Search

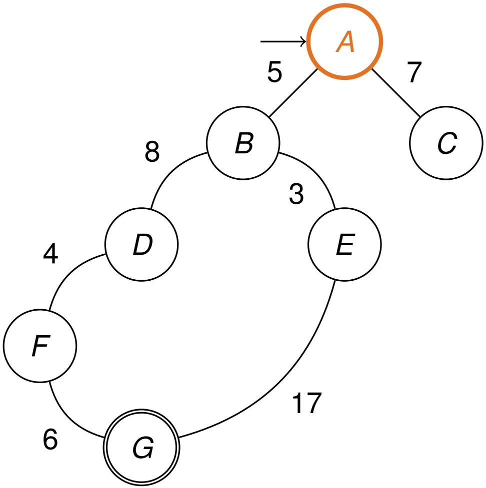
Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier*:
   - $frontier = \langle B, C \rangle$. No goals.
4. Choose first element in *frontier*.

And repeat…



Breadth First Search

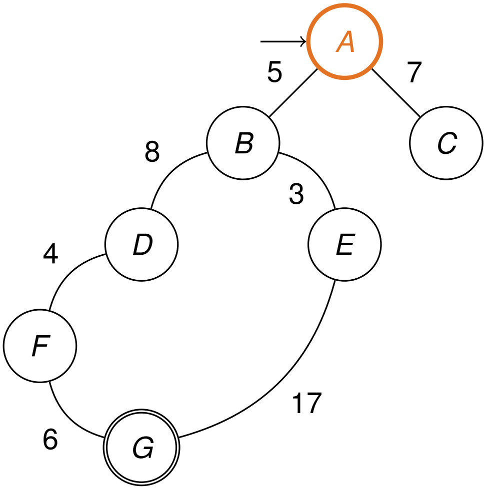
Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so *Expand*(*A*)
3. Add successors to *frontier*:
   - *frontier* $= \langle B, C \rangle$. No goals.
4. Choose first element in *frontier*.

And repeat. . .



Breadth First Search

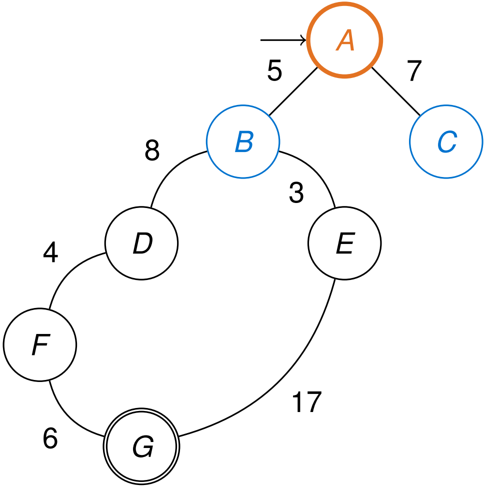
Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier*:
   - $frontier = \langle B, C \rangle$. No goals.
4. Choose first element in *frontier*.

And repeat…



Breadth First Search

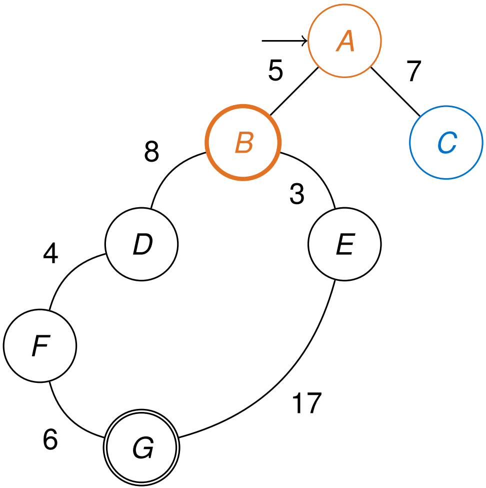
Uninformed search strategies

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by *breadth* first)

- The *frontier* is a queue, i.e., “First In, First Out” (FIFO).
- Start at A and Goal is G.

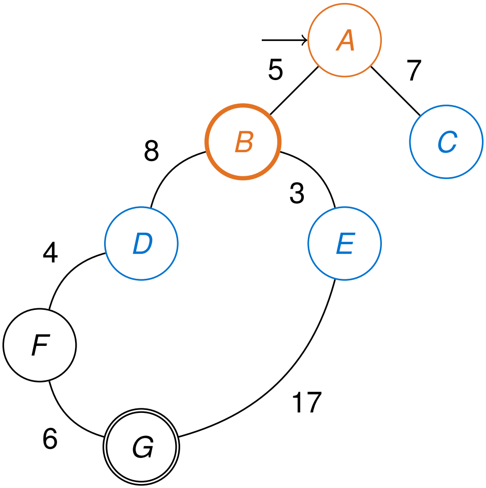
1. Add A to frontier and solution.
2. A is not goal, so *Expand*(*A*)
3. Add successors to *frontier*:
   - *frontier* $= \langle B, C \rangle$. No goals.
4. Choose first element in *frontier*.

And repeat...



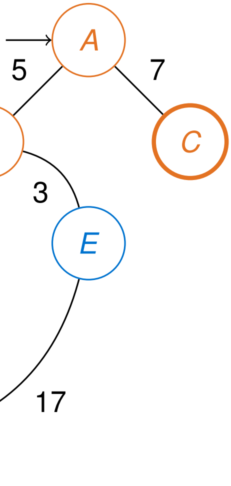
Breadth First Search

Uninformed search strategies



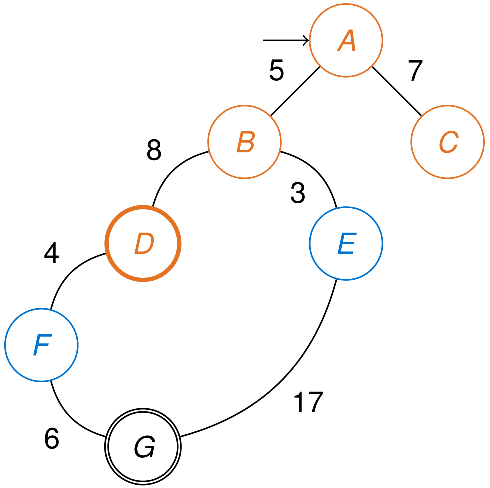
Breadth First Search

Uninformed search strategies



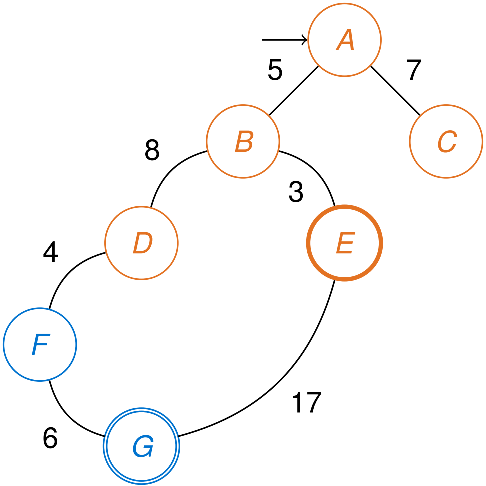
Breadth First Search

Uninformed search strategies



Breadth First Search

Uninformed search strategies

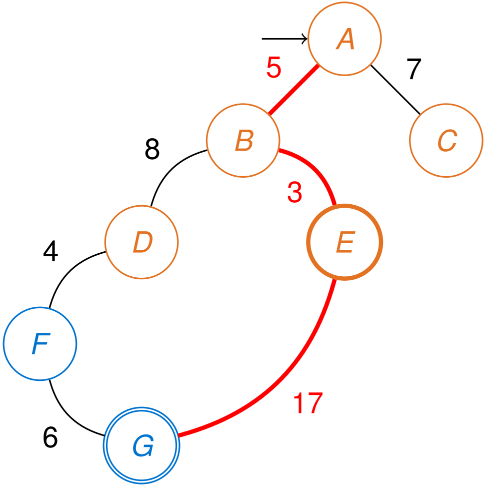


Breadth First Search

Uninformed search strategies

- We have **seen** the goal![a]
- We can reconstruct the solution by creating a *chain of parents* from the **goal**

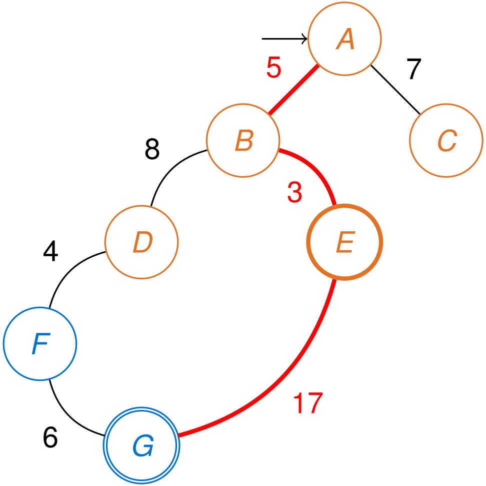
[a] Remember we check for goal when adding to the frontier in BFS!



Breadth First Search

Uninformed search strategies

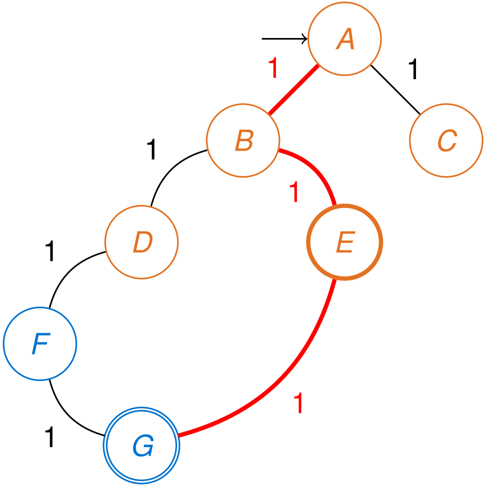
- **Not optimal**, unless all costs were equal!



Breadth First Search

Uninformed search strategies

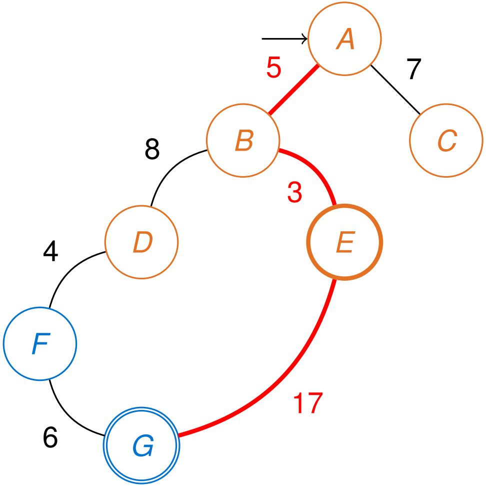
- **Not optimal**, unless all costs were equal!
- Like so!



Breadth First Search

Uninformed search strategies

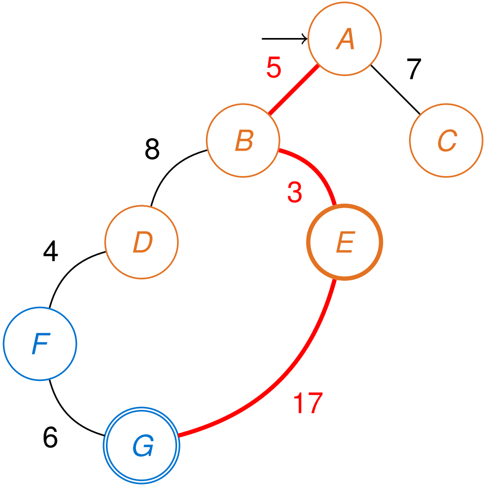
- **Not optimal**, unless all costs were equal!
- Like so!
- **Complete**: always finds a solution if space state is finite



Breadth First Search

Uninformed search strategies

- **Not optimal**, unless all costs were equal!
- Like so!
- **Complete**: always finds a solution if space state is finite
- Time and space complexity is insane $\mathcal{O}(b^d)$ where b is the branching factor (number of successors to consider) and d is the depth of the shallowest solution.



This was a summarised, but still quite detailed explanation. The following algorithms will be **summarised a bit more**, so

check your book for the step by step strategies.

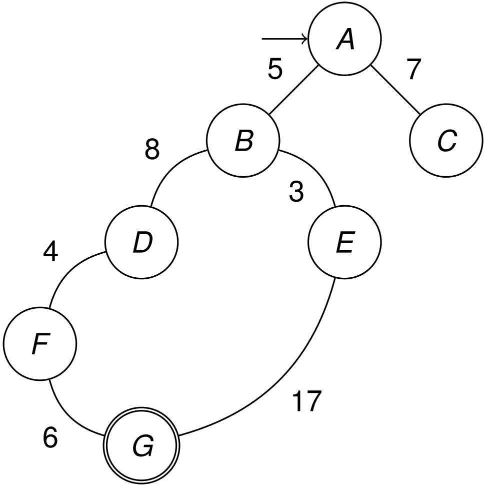
Depth First Search

Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., “Last In, First Out” (LIFO)
- Start at A and Goal is G, as before

1. Add A to frontier and solution.



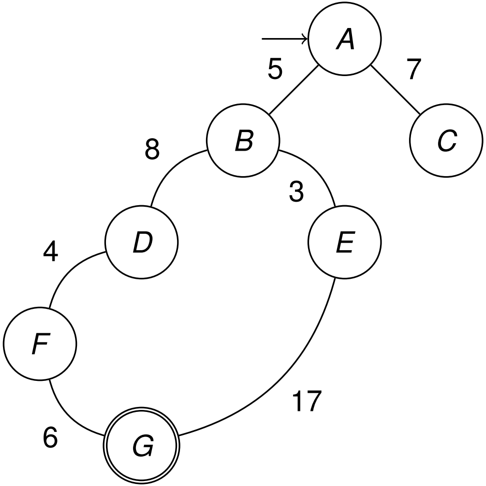
Depth First Search

Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., “Last In, First Out” (LIFO)
- Start at A and Goal is G, as before

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$



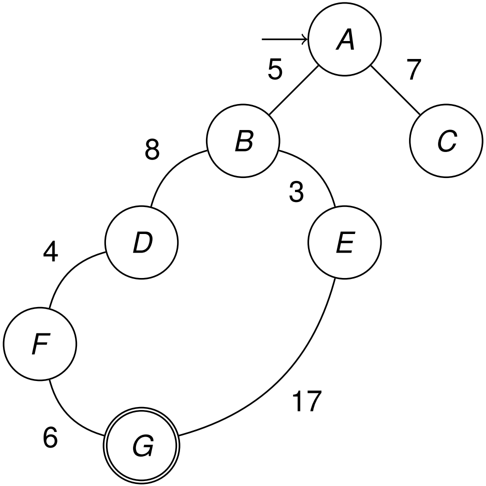
Depth First Search

Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., "Last In, First Out" (LIFO)
- Start at A and Goal is G, as before

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier* (in reverse order):



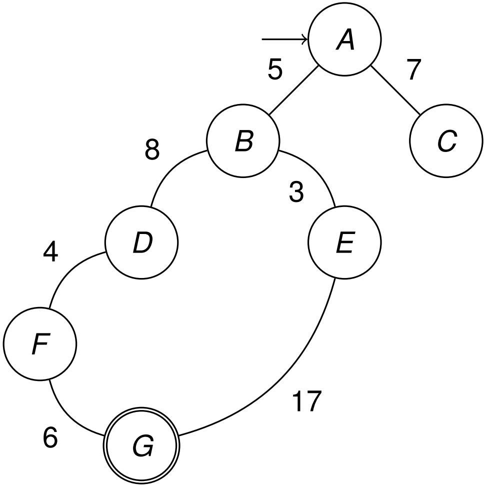
Depth First Search

Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., “Last In, First Out” (LIFO)
- Start at A and Goal is G, as before

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier* (in reverse order):
   - $frontier = \langle B, C \rangle$



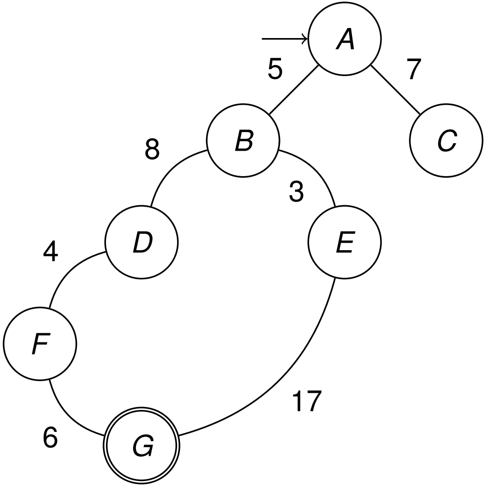
Depth First Search

Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., “Last In, First Out” (LIFO)
- Start at A and Goal is G, as before

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier* (in reverse order):
   - $frontier = \langle B, C \rangle$
   - Frontier now grows towards left!



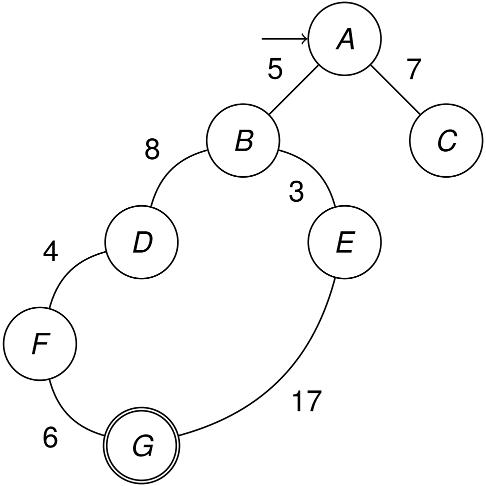
Depth First Search

Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., "Last In, First Out" (LIFO)
- Start at A and Goal is G, as before

1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier* (in reverse order):
   - $frontier = \langle B, C \rangle$
   - Frontier now grows towards left!
4. Choose **first element** in *frontier*.



Depth First Search

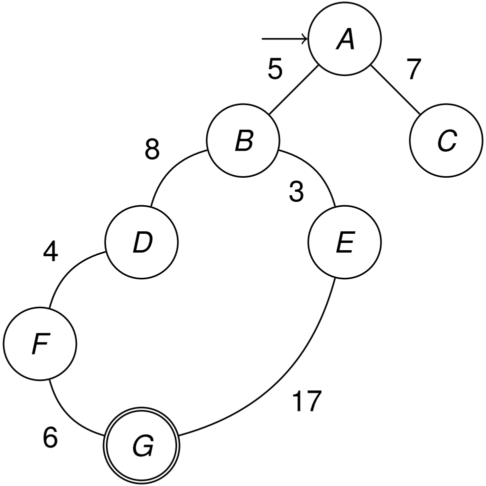
Uninformed search strategies

DFS prioritises **new nodes first**, and previously discovered ones go last (hence the name, as it explores by *depth* first)

- The *frontier* is a **stack**, i.e., “Last In, First Out” (LIFO)
- Start at A and Goal is G, as before

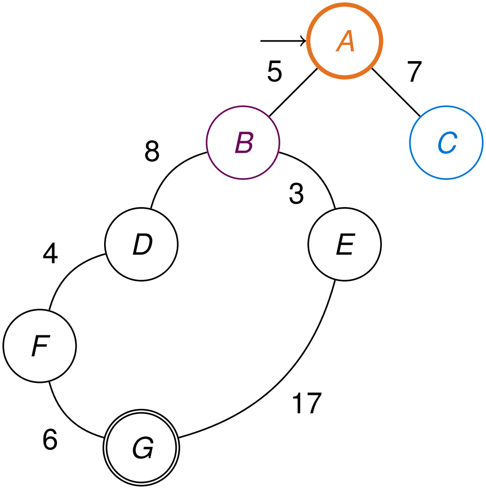
1. Add A to frontier and solution.
2. A is not goal, so $Expand(A)$
3. Add successors to *frontier* (in reverse order):
   - $frontier = \langle B, C \rangle$
   - Frontier now grows towards left!
4. Choose **first element** in *frontier*.

And repeat...



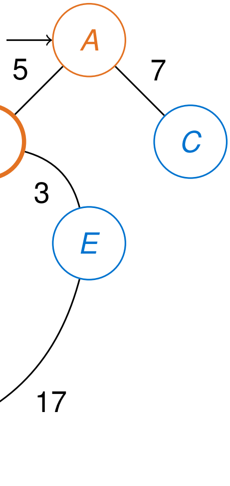
Depth First Search

Uninformed search strategies



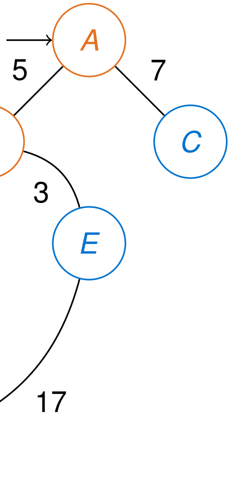
Depth First Search

Uninformed search strategies



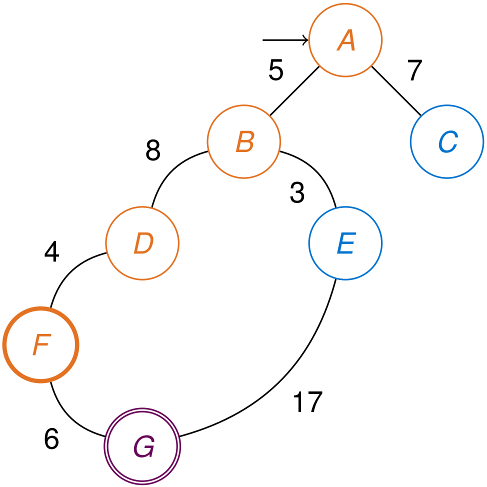
Depth First Search

Uninformed search strategies



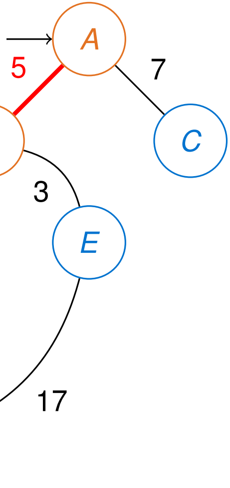
Depth First Search

Uninformed search strategies



Depth First Search

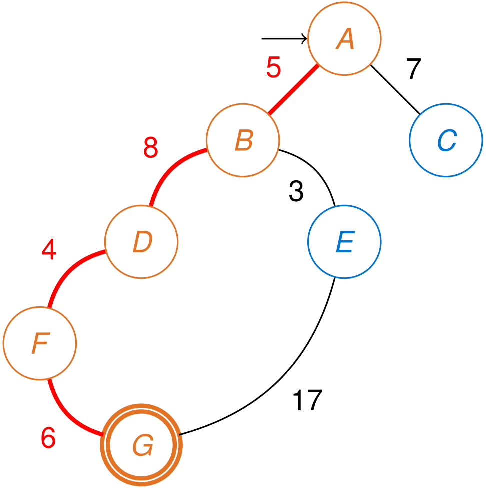
Uninformed search strategies



Depth First Search

Uninformed search strategies

- **Not always optimal**

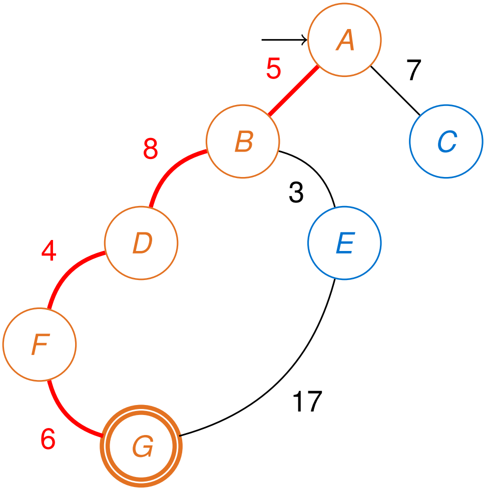


[a]because it is usually implemented as tree search

Depth First Search

Uninformed search strategies

- **Not <u>always</u> optimal**
- It returns the first solution found. We were lucky!

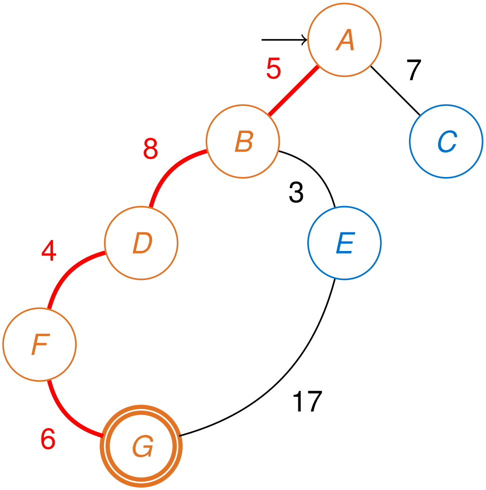


[a]because it is usually implemented as tree search

Depth First Search

Uninformed search strategies

- **Not always optimal**
- It returns the first solution found. We were lucky!
- **Not Complete**: fails in infinitely-deep spaces and spaces with loops[a]

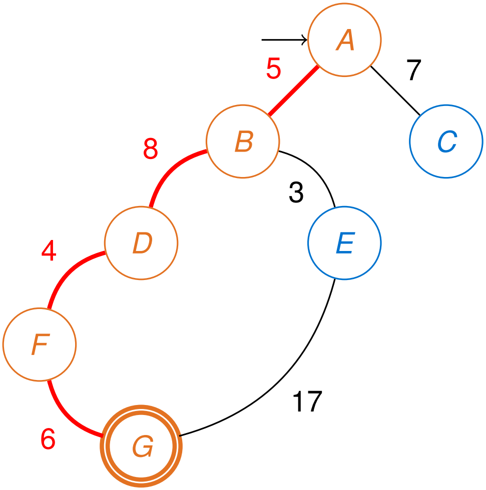


[a]because it is usually implemented as tree search

Depth First Search

Uninformed search strategies

- **Not always optimal**
- It returns the first solution found. We were lucky!
- **Not Complete**: fails in infinitely-deep spaces and spaces with loops[a]
- Time complexity $\mathcal{O}(b^m)$, and space complexity is linear $\mathcal{O}(bm)$ where b is the branching factor and m is the maximum depth in the state space (tree version)

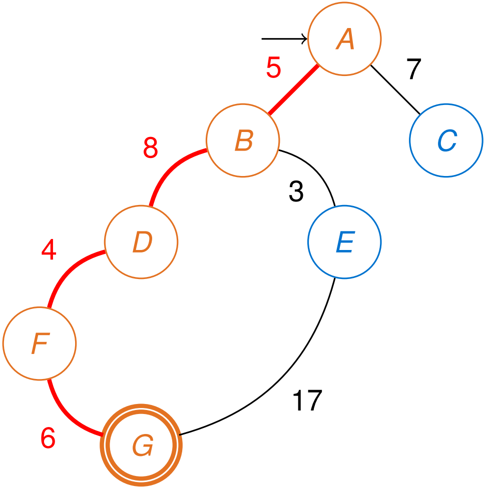


[a]because it is usually implemented as tree search

Depth First Search

Uninformed search strategies

- **Not always optimal**
- It returns the first solution found. We were lucky!
- **Not Complete**: fails in infinitely-deep spaces and spaces with loops[a]
- Time complexity $\mathcal{O}(b^m)$, and space complexity is linear $\mathcal{O}(bm)$ where b is the branching factor and m is the maximum depth in the state space (tree version)
- One can make a smarter version of DFS with graph search (memory). Space complexity grows to exponential, and might still miss if on infinite spaces.



[a]because it is usually implemented as tree search

Depth-limited and Iterative deepening search

Uninformed search strategies

Two other ideas lie on imposing a limit on DFS, both as *tree search strategies*.

- Use DFS with $DepthLimit = 1$
- If no solution found, then try increasing the *DepthLimit iteratively* until a set *cutoff*.

Depth-limited and Iterative deepening search

Uninformed search strategies

Two other ideas lie on imposing a limit on DFS, both as *tree search strategies*.

- Use DFS with $\mathit{DepthLimit} = 1$
- If no solution found, then try increasing the *DepthLimit iteratively* until a set *cutoff*.

- Iterative deepening will try multiple levels and return either a **solution** if it exists, a **failure** if it does not, or a **cutoff**.
- A *cutoff* means the maximum depth we set previously was reached, so a solution might exist deeper than the levels we explored.

Depth-limited and Iterative deepening search

Uninformed search strategies

Two other ideas lie on imposing a limit on DFS, both as *tree search strategies*.

- Use DFS with $\mathit{DepthLimit} = 1$
- If no solution found, then try increasing the *DepthLimit iteratively* until a set *cutoff*.

- Iterative deepening will try multiple levels and return either a **solution** if it exists, a **failure** if it does not, or a **cutoff**.
- A *cutoff* means the maximum depth we set previously was reached, so a solution might exist deeper than the levels we explored.

- **Always complete** if solution exists and state space is finite
- **Not cost optimal** unless costs are the same (like BFS)
- Time complexity: $\mathcal{O}(b^d)$
- Space complexity: $\mathcal{O}(bd)$ (like DFS)

Slightly better than both DFS and BFS!

Uninformed search strategies

- They systematically navigate the search space blindly—not questioning where the goal may be in the space.
- The search space is often very large.

Uninformed search strategies

- They systematically navigate the search space blindly—not questioning where the goal may be in the space.
- The search space is often very large.

Why not being *smarter* about it?

Section 4

Informed search strategies

Heuristic search

Informed search strategies

To take *better informed decisions*, we can use a domain-specific hint about how “desirable” a state can be.

Heuristic search

Informed search strategies

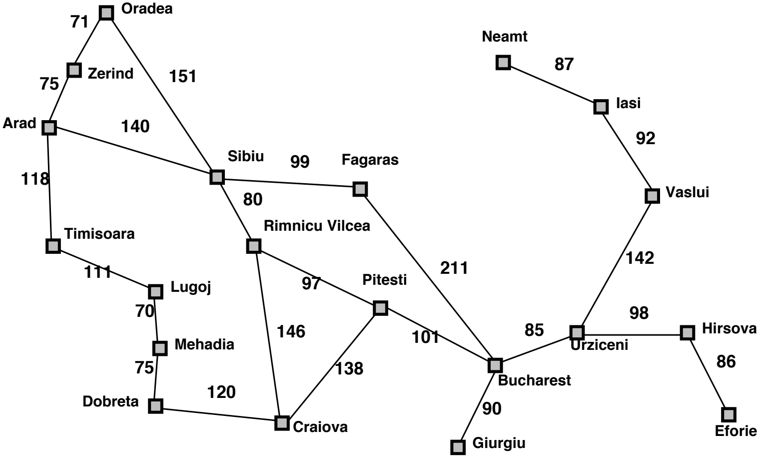
To take *better informed decisions*, we can use a domain-specific hint about how "desirable" a state can be.

This is usually done by using a **heuristic function** $h(n)$, where $h : S \to \mathbb{R}$, i.e., a *guessing function* about an estimated remaining cost to the goal.

Heuristic example: Romania

Informed search strategies

Using *h* as the straight line distance to goal:



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(Greedy) Best First Search

Informed search strategies

Best first

Choose always the best of your expectations (cheapest estimate).

(Greedy) Best First Search

Informed search strategies

Best first

Choose always the best of your expectations (cheapest estimate).

Yet again, same idea:

(Greedy) Best First Search

Informed search strategies

Best first

Choose always the best of your expectations (cheapest estimate).

Yet again, same idea:

1. Start

(Greedy) Best First Search
Informed search strategies

Best first

Choose always the best of your expectations (cheapest estimate).

Yet again, same idea:

1. Start
2. Check for goal

(Greedy) Best First Search

Informed search strategies

Best first

Choose always the best of your expectations (cheapest estimate).

Yet again, same idea:

1. Start
2. Check for goal
3. Expand and update frontier

(Greedy) Best First Search

Informed search strategies

Best first

Choose always the best of your expectations (cheapest estimate).

Yet again, same idea:

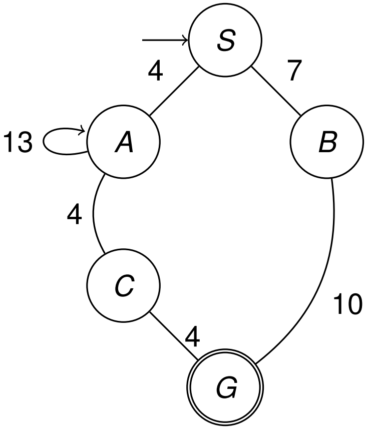
1. Start
2. Check for goal
3. Expand and update frontier
4. Choose the best of the estimates

(Greedy) Best First Search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$



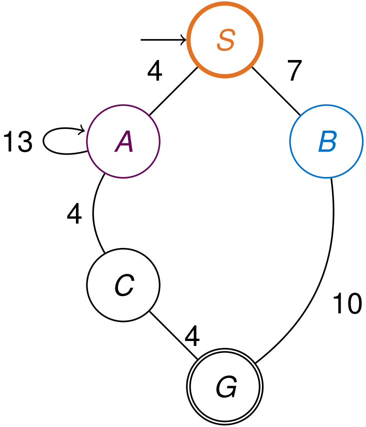
(Greedy) Best First Search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

We choose alphabetically in case of a tie.



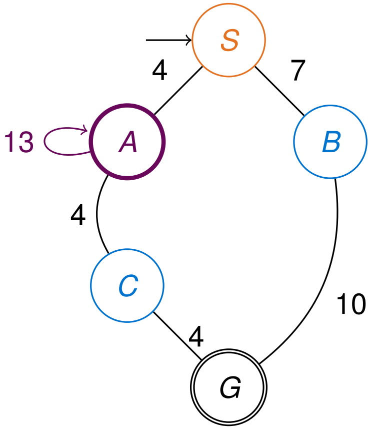
(Greedy) Best First Search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

We choose alphabetically in case of a tie.



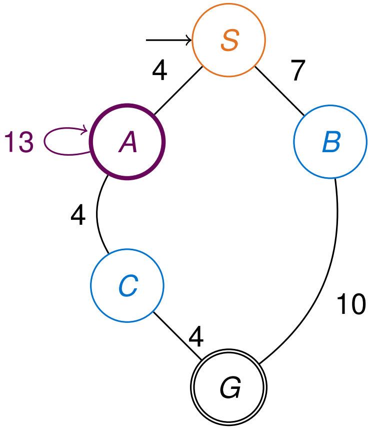
(Greedy) Best First Search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

We choose alphabetically in case of a tie.



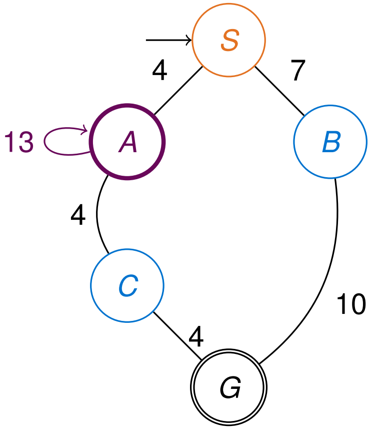
(Greedy) Best First Search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

We choose alphabetically in case of a tie.

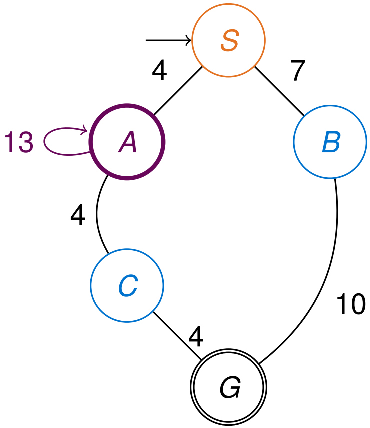


(Greedy) Best First Search

Informed search strategies

With those estimated distances to the goal:

- We have a cycle!
- Tree search would not make it past *A*
- By adding memory we make it smarter. Still, space complexity increases.
- **Always Complete** in finite spaces with no loops (not our case)
- **Might not be optimal** (See Romania example!)



A^* search

Informed search strategies

What if we consider the cost *and* the heuristic?

A^* search

Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **evaluation function** (criterion) will consider both things:

$$f(n) = g(n) + h(n)$$

where

A^* search

Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **evaluation function** (criterion) will consider both things:

$$f(n) = g(n) + h(n)$$

where

- $g(n)$ is the cost we have paid so far to reach n

A^* search

Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **evaluation function** (criterion) will consider both things:

$$f(n) = g(n) + h(n)$$

where

- $g(n)$ is the cost we have paid so far to reach n
- $h(n)$ is the estimated cost of the node (to the goal)

A^* search

Informed search strategies

What if we consider the cost *and* the heuristic?

Our new **evaluation function** (criterion) will consider both things:

$$f(n) = g(n) + h(n)$$

where

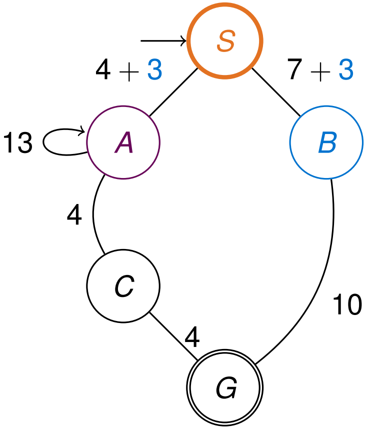
- $g(n)$ is the cost we have paid so far to reach n
- $h(n)$ is the estimated cost of the node (to the goal)
- $f(n)$ is then the estimated cost of the cheapest solution through n to the goal

A^* search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

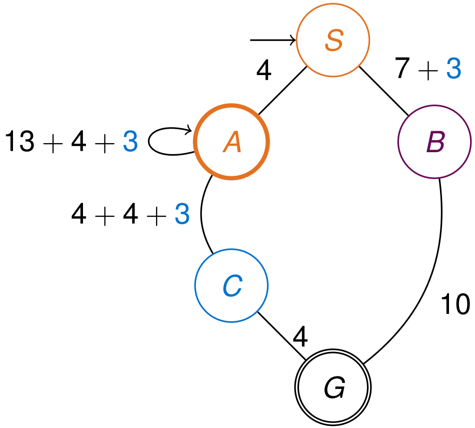


A^* search

Informed search strategies

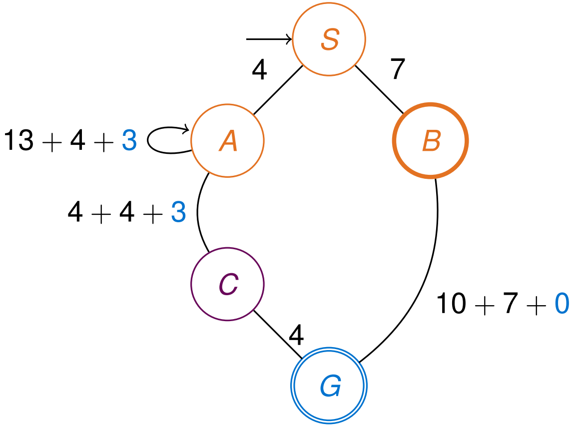
With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$



With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

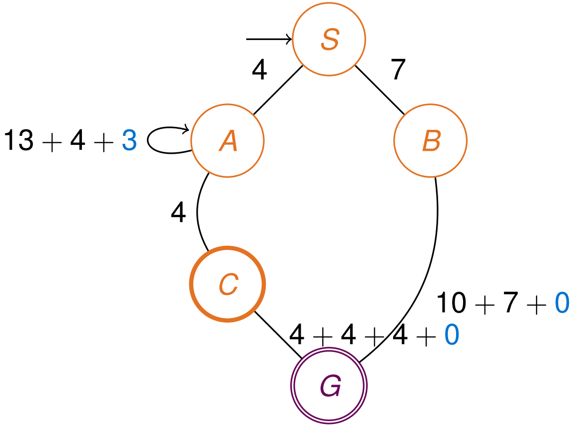


A^* search

Informed search strategies

With the following estimated distances to the goal:

- $h(A) = 3$
- $h(B) = 3$
- $h(C) = 3$
- $h(G) = 0$

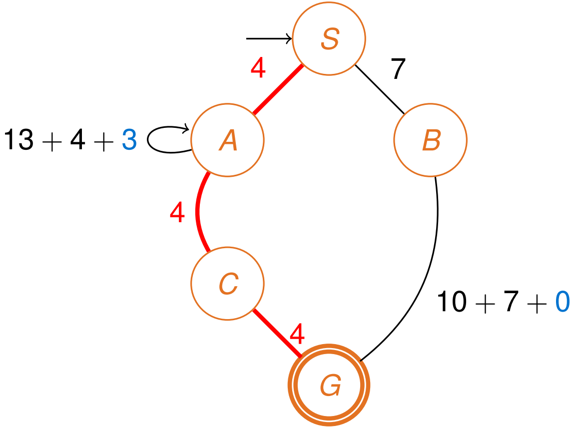


A^* search

Informed search strategies

With the those estimated distances to the goal:

- We have found the goal!

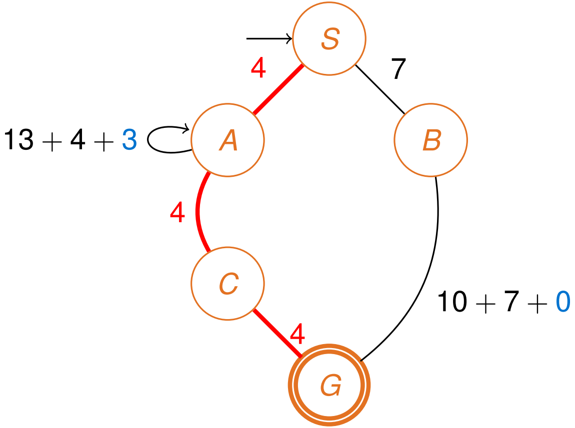


A^* search

Informed search strategies

With the those estimated distances to the goal:

- We have found the goal!
- It is **complete** for positive costs, within a finite state space and an existing solution.

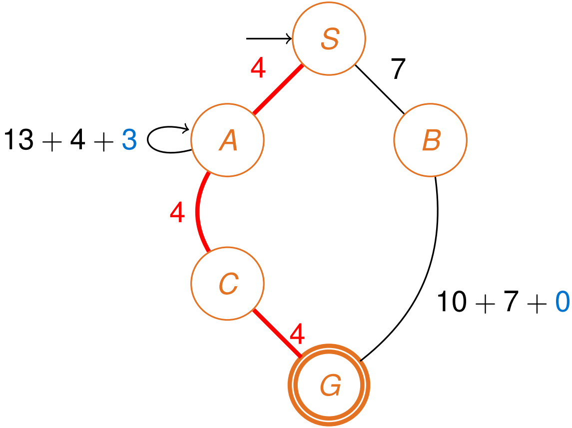


A^* search

Informed search strategies

With the those estimated distances to the goal:

- We have found the goal!
- It is **complete** for positive costs, within a finite state space and an existing solution.
- it is **cost optimal** if **certain conditions are met**



Friendly reminder

Things to look out for

Implementation details vary a lot, and can be tricky!

- Is the algorithm checking for redundant paths (**graph search**) or not (**tree search**)?
- Is the goal check performed **early** (when a node is *generated*) or **late** (when a node is *expanded*)?
- Is the algorithm **storing** all **reached states**, or **reconstructing the path** from a chain of parent nodes?

Read the book!

To become familiar with the algorithms and their implementations details, you should read the book. These slides are not a replacement for the book; they are a summary of the most important points.

Cheatsheet

Things to look out for

Most of the search strategies we cover in this course use the same algorithm to search.[4] It is just `Best-First-Search` with different functions to decide which element will be popped out of the priority queue:

Depth-First Search

$$f(n) = -\text{depth}(n)$$

Uniform-Cost Search (Dijkstra)

$$f(n) = g(n)$$

Greedy Best-First Search

$$f(n) = h(n)$$

A* Search

$$f(n) = g(n) + h(n)$$

[4] Except for BFS that has a separate algorithm.