Introduction to Artificial Intelligence

Ashkan Panahi and Claes Strannegård

Department of Computer Science and Engineering
Chalmers University of Technology

Lecture 3: Classical AI

DO NOT DISTRIBUTE

How do navigators work?

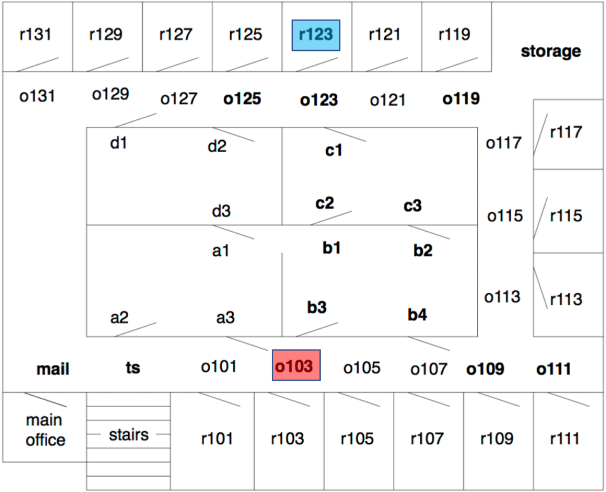
Overview

Sources

- Online book chapter. David L. Poole and Alan K. Mackworth, Cambridge University Press, 2017: Artificial Intelligence: Foundations of Computational Agents. Chapter 3: Searching for Solutions up to and including 3.6.
- Some slides are by Peter Ljunglöf and used with his permission.

Search problems

Delivery robot problem



Suppose a delivery robot wants to go from start (red) to goal (blue).

Graph terminology

- A (directed) *graph* consists of a set of *nodes* and and a binary relation on the nodes, whose elements are called *arcs* or *edges*. Note that the arc relation is generally not symmetric.
- A node n_2 is called a *successor* (or *child*) of the node n_1 if there is an arc from n_1 to n_2.
- A *path* is a sequence of nodes $(n_0, n_1, \ldots, n_k)$ such that (n_i, n_{i+1}) is an arc, for all i such that $0 \leq i \leq k$.
- A *cycle* is a path whose first and last nodes are the same.
- A directed graph without any cycles is called a *directed acyclic graph* (DAG).

Graph problems (today's topic)

Definition

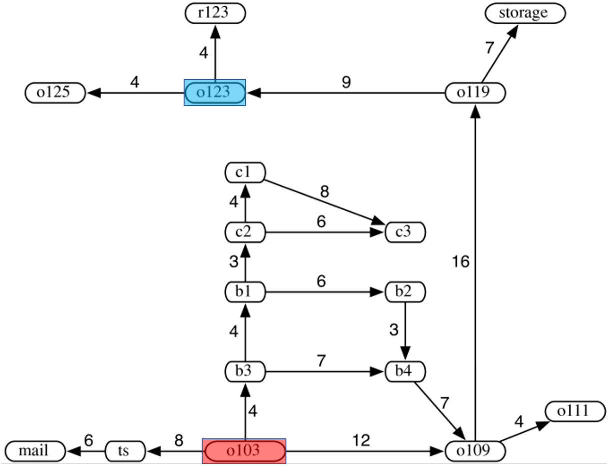
A *graph problem* (or *path-finding problem*) consists of

- a set of *states* (or *nodes*)
- a set of *arcs* between states (that may be labeled with labels representing actions and/or costs)
- a distinguished state called *start state*
- a distinguished set of states called *goal states*.

Definition

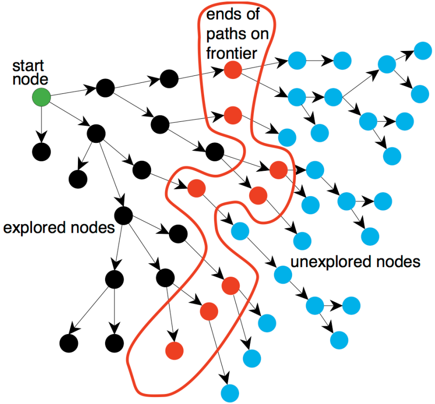
A *solution* to a graph problem is a path leading from the start state to a goal state.

Example of a graph problem



The Delivery robot problem formulated as a graph problem (with exactly 3 solutions). The labels are not relevant here.

Frontier



We will soon define a generic algorithm for solving graph problems. The algorithm maintains a set of paths called the *Frontier* (or *Fringe*). We will make sure that any solution must begin with a path that belongs to the Frontier.

Generic Search Algorithm

Generic Search Algorithm

procedure *Search*(G, S, goal)

    Inputs

        G: graph with nodes N and arcs A

        s: start node

        goal: Boolean function of nodes

    Output

        path from s to a node for which goal is true

        or $\perp$ if there are no solution paths

    Local

        Frontier: set of paths

    Frontier $:= \{\langle s \rangle\}$

    while Frontier $\neq \{\}$ **do**

        select and remove $\langle n_0, \ldots, n_k \rangle$ from Frontier

        if goal (n_k) **then**

            return $\langle n_0, \ldots, n_k \rangle$

        Frontier $:=$ Frontier $\cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$

    return $\perp$

Instances

Different instances of the Generic Search Algorithm can be defined by specifying how the paths are to be selected from the Frontier:

- Breadth-first search: Select the path that was added to the Frontier the longest time ago.
- Depth-first search: Select the path that was added to the Frontier the most recently.
- Best-first search: Use a function that assigns “grades” to paths. Select the path that has the best “grade.”

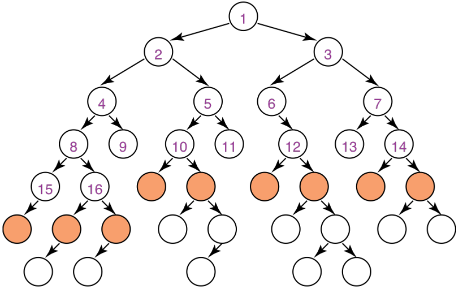
Implementation

One way of implementing the Generic Search Algorithm is to let the Frontier be a list and always select the first path (=head) of the list. Then the above instances can be obtained by different policies for inserting new paths into the list:

- *Breadth-first search*: In this case new paths are inserted at the end of the list. Then the Frontier is an ordinary queue, i.e. a FIFO (First-in First-Out) queue.
- *Depth-first search*: In this case new paths are inserted at the front of the list. Then the Frontier is a stack, i.e. a LIFO (Last-in First-Out) queue.
- *Best-first search*: In this case new paths are inserted into the list based on their "grades." Then the Frontier is sorted by "grades."

Uninformed search

Breadth-first search (BFS)



We can illustrate the order in which paths are checked (and removed from the Frontier) by using a tree whose nodes represent paths. In the case of BFS the paths are checked in the order shown. First the path with end-node 1, then the path with end-node 2, etc. The shaded nodes are the end-nodes of paths that are on the Frontier right after path 16 was checked.

Breadth-first search: analysis

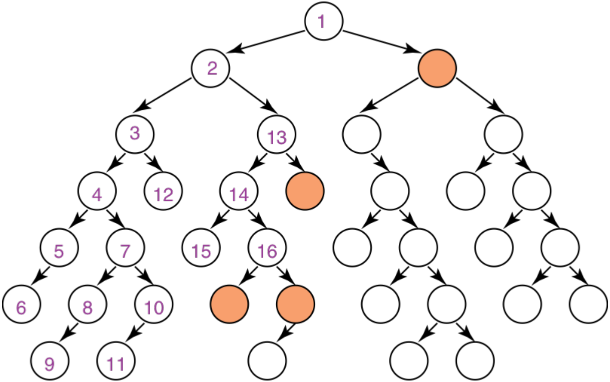
Breadth-first search is useful when:

- the problem is small enough so that the graph can be stored explicitly, or
- there are short solutions.

It is a poor method when:

- the graph is large (and dynamically generated) and
- there are no short solutions.

Depth-first search (DFS)



The shaded nodes are the end-nodes of paths that are on the Frontier right after path number 16 was removed in a search with DFS.

Depth-first search: analysis

Depth-first search is appropriate when:

- the search space is small, or
- many solutions exist

It is poor when:

- it is possible to get caught in infinite paths, which might happen when the graph is infinite or contains loops.

Iterative Deepening DFS

Iterative deepening depth-first search proceeds as follows:

- First do a depth-first search down to depth 1. (So only paths of max length 1 are put into the Frontier.)
- If that does not lead to a solution, do a depth-first search down to depth 2
- If that does not lead to a solution, do a depth-first search down to depth 3
- and so on until a solution is found.

Iterative Deepening DFS

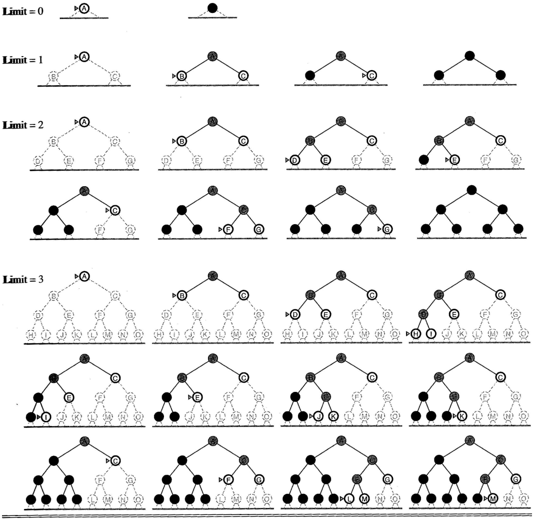


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Iterative Deepening DFS: analysis

Advantages:

- it always finds a solution if there is one (like BFS)
- it always finds the shortest solution (like BFS)
- it is memory efficient (like DFS)

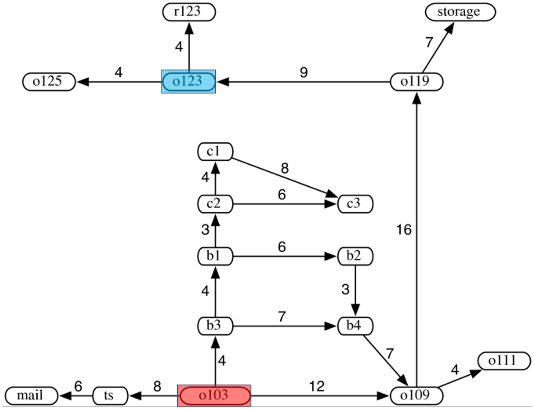
Drawback:

- some nodes are revisited many times.

Adding arc costs

- Sometimes it is useful to put costs on arcs. For example, the costs might represent travel time or travel distance.
- We write the cost of arc (n_i, n_j) as $cost(n_i, n_j)$.
- Given a path $p = (n_0, n_1, \ldots, n_k)$, the cost of p, $cost(p)$ is defined as the sum of the costs of the arcs appearing in p.
- Given a cost function, we may look for an optimal solution to a graph problem, e.g. the shortest path or the fastest path.

Finding an optimal path



Find a path from the red node to the blue node with minimum cost.
This is the kind of problem that navigators need to solve.

Lowest-cost-first search

- Applies when the arcs are labeled with costs.
- A version of the Generic Search Algorithm
- *Lowest-cost-first search*: Let the Frontier be a list sorted by path cost (with the path with the lowest cost first).
- When arc costs are all equal, it coincides with BFS.
- It always finds the cheapest solution, so it is optimal.
- But it has limited scalability (like BFS)...

Informed search

Heuristics

- In everyday language, a *heuristic* is a rule of thumb that indicates where to search primarily.
- The word has the same origin as the Greek "Eureka!" ("I found it!") that Archimedes shouted in his bathtub.

Heuristics

Example

Heuristic principles in everyday life:

- Search for toys at low levels primarily
- Search for blueberries in forests primarily
- Search for translations that use common words primarily
- Search for solutions that are simple primarily

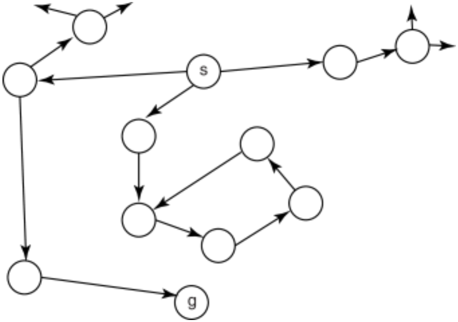
Heuristic functions

Definition

A *heuristic function* is a function h that assigns a non-negative real number $h(p)$ to each path p. Intuitively it is an estimate of the cost of the cheapest path from the end-node of p to a goal node.

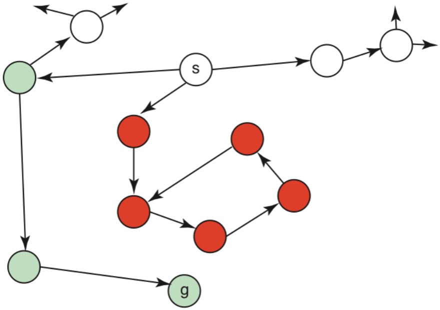
For calculating $h(p)$, the only relevant part of p is its end-node. Some texts define heuristic functions on nodes and costs on paths. Our choice here is to define both heuristic functions and costs on paths.

Example of a heuristic function



- This is a graph problem with arc costs drawn to scale. The cost of each arc is its length. The aim is to find the shortest path from s to g.
- A heuristic function $h(p)$ can be defined as the straight-line distance from the end node of p to g.

Greedy best-first search



- *Greedy best-first search*: Keep the Frontier sorted by heuristic value $h(p)$ (with paths with low values first).
- In the above example, the algorithm will get stuck in the red loop and never terminate!
- So greedy best-first search is not what we want...

The algorithm A*

- Very powerful search algorithm
- Pronounced "A star"
- Invented by Hart, Nilsson and Raphael in 1968.
- A kind of best-first search: the Fringe is sorted by "grades."

A* search

A* search uses both path cost and heuristic values.

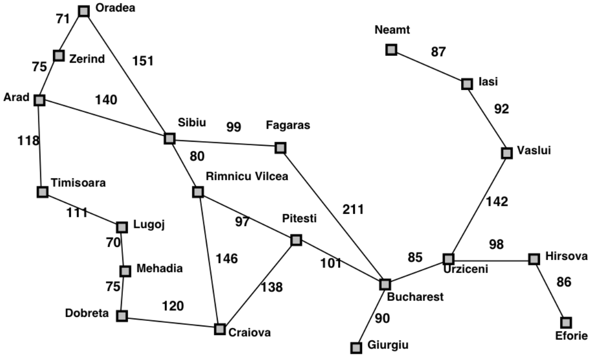
$cost(p)$ is the cost of path p.

$h(p)$ estimates the cost from the end node of p to a goal.

$f(p) = cost(p) + h(p)$, estimates the total path cost of going from the start node, via path p to a goal:

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p}}_{cost(p)}\, n \,\underbrace{\xrightarrow{\text{estimate}}\; goal}_{h(p)}}_{f(p)}$$

Running example: driving in Romania

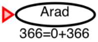


Straight–line distance to Bucharest

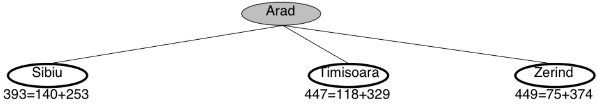
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

We want to find the shortest path from Arad to Bucharest using a map with road distances to neighbors (for computing $cost(p)$) and a table with straight-line distances (for computing $h(p)$). This information is available to a navigator.

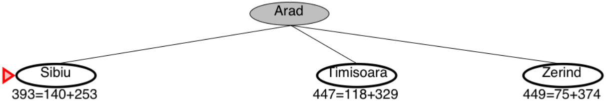
A* at work



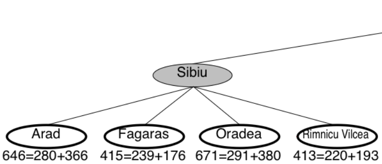
A* at work


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

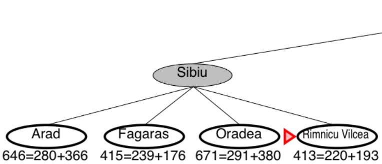
A* at work


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

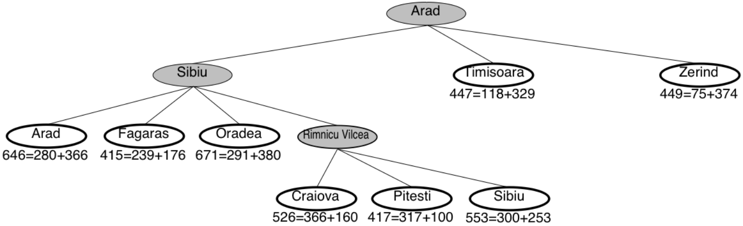
A* at work


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* at work

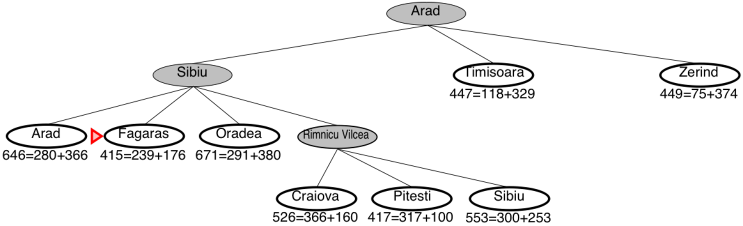

Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* at work

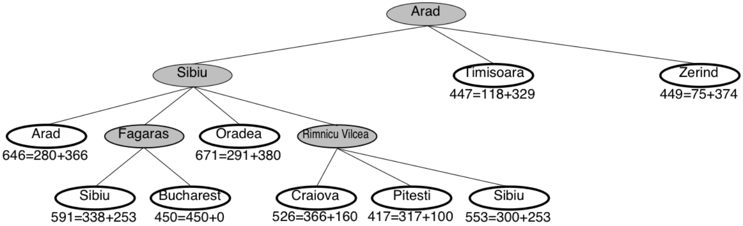


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* at work

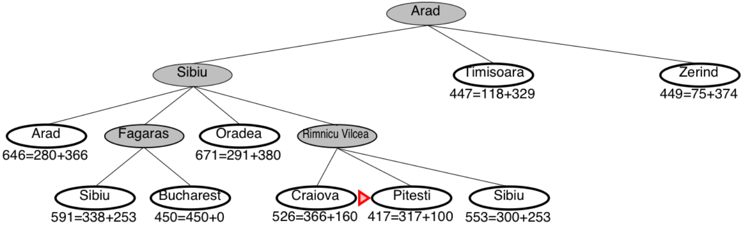

Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* at work



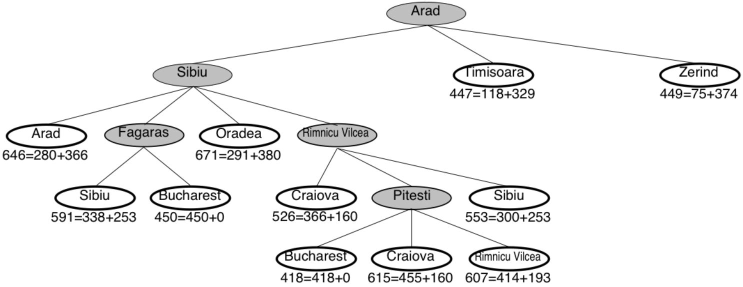
Since we follow the Generic Search Algorithm, we don't stop here just because we added Bucharest (a goal state) to the Frontier.

A* at work

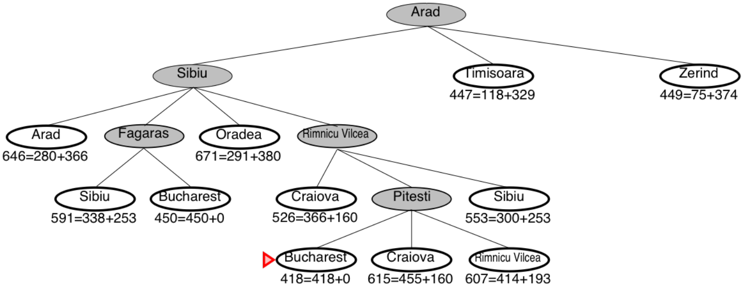


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* at work

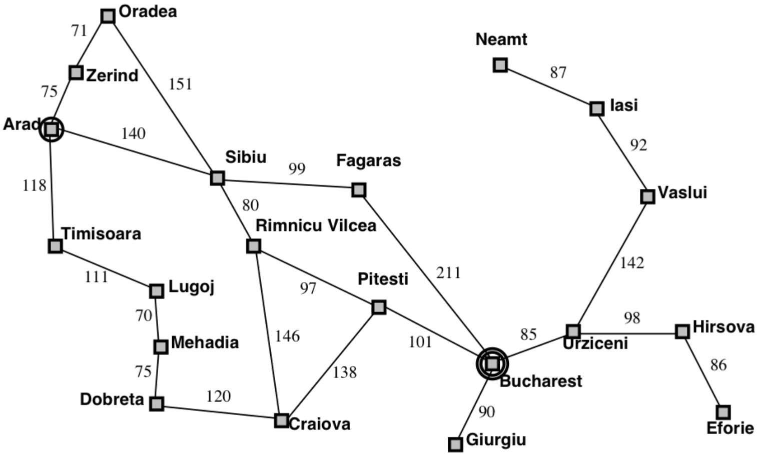

Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

A* at work



Since we follow the Generic Search Algorithm, we stop here. In fact we just removed (and returned) a path ending in a goal state (Bucharest) from the Frontier.

A* at work



So A* found the path Arad-Sibiu-Rimnicu-Pitesti-Bucharest (418km). This is the shortest path. Actually, A* always finds the shortest path from any city to any city!

Admissible heuristics

Definition

The heuristic function h is *admissible* if

$$h(p) \leq cost(p'),$$

whenever p' starts at the end-node of p and ends at a goal node. In words: an admissible heuristic never overestimates the actual cost of reaching a goal node. In other words: The estimate of the remaining cost is never higher than the actual cost.

Example

The straight-line distance heuristic is admissible, since the it is always smaller than or equal to the actual (road) distance.

Optimality of A*

Theorem

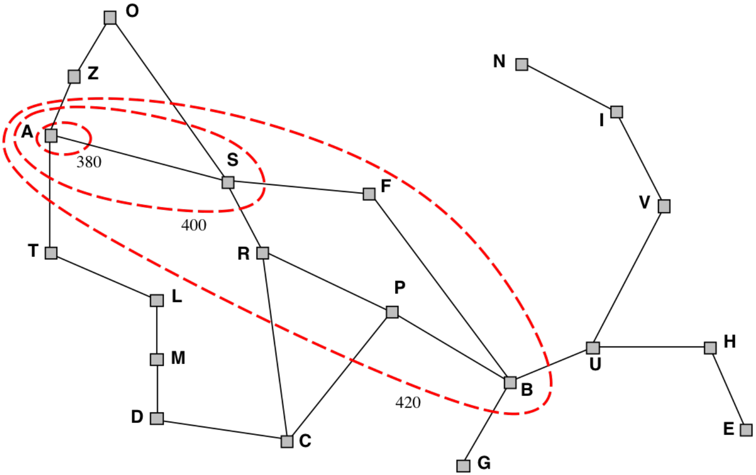
If there is a solution, then A always returns an optimal solution, provided that:*

1. *the branching factor is finite,*
2. *the arc costs are uniformly bounded (i.e., there is an $\varepsilon > 0$ such that all of the arc costs are greater than ε), and*
3. *the heuristic function h is admissible.*

Proof.

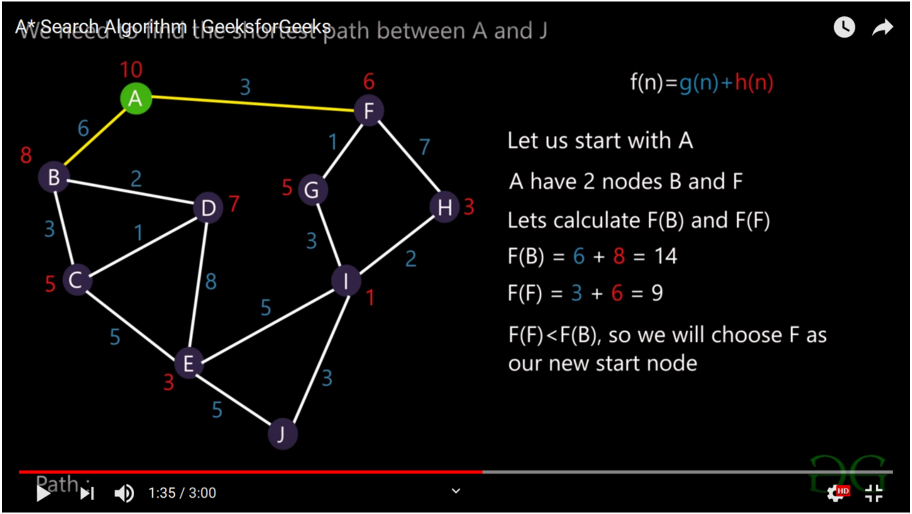
First, suppose there is only one optimal solution, p. Then the first two requirements ensure that p will eventually enter the Frontier. The last requirement ensures that p will be sorted before any other solution. Hence A* will eventually return p. The case when there are several optimal solutions is similar. □

Why is A* optimal?



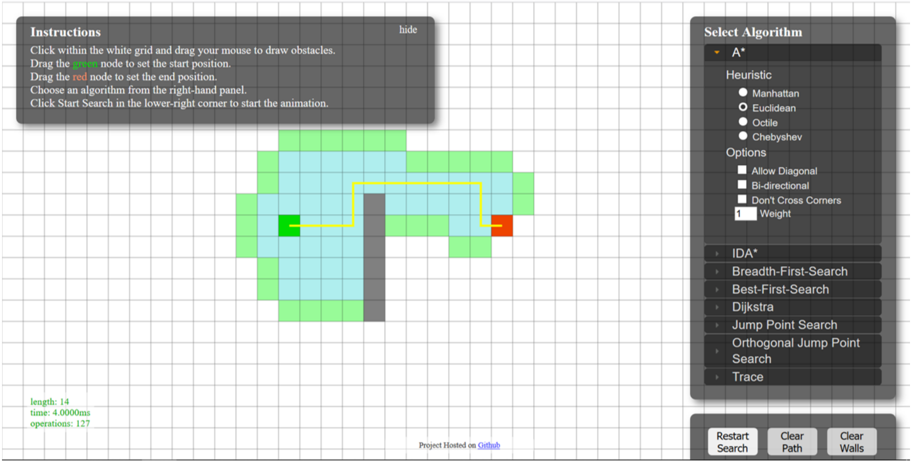
Paths with bigger and bigger f-values will be put on the Fringe.

Video about A*



Video

Play with search algorithms



Animation. Light green: state at the end of some generated path. Blue: state at the end of some selected path. Yellow: returned path.

How do navigators work?

For instance, the TomTom route engine is based on A*. It takes real-time traffic data as input to find the fastest way.