Shortest paths algorithms in weighted graphs

Lluís Alsedà

Departament de Matemàtiques
Universitat Autònoma de Barcelona
http://www.mat.uab.cat/~alseda

Version 3.0 (May, 2023)

DEPARTAMENT DE MATEMÀTIQUES

Table of Contents

Shortest paths in weighted graphs

Contents

Reminder of basic graph definitions

[1] A *(combinatorial) graph* is a pair $G = (V, E)$ consisting of a *set of vertices* or *nodes* V, and a subset $E \subset V \times V$ of the Cartesian product $V \times V$.

In the case of an *undirected graph* the elements of E are called *edges* and the pairs $(a, b) \in E$ are considered unordered (that is, there is an edge between $a \in V$ and $b \in V$ when $(a, b) \in E$ or $(b, a) \in E$ — i.e., the pairs (a, b) and (b, a) are identified).

In the case of a *directed* or *oriented graph* the elements of E are called *arrows* and the pairs $(a, b) \in E$ are considered with order (that is, there is an arrow from $a \in V$ to $b \in V$ if and only if $(a, b) \in E$, and the pairs (a, b) and (b, a) are *not* identified).

[1] http://en.wikipedia.org/wiki/Graph_theory

Reminder of basic graph definitions

- The *order* of a graph is the number of vertices, i.e. the *cardinal* of the set V: $|V|$.
- The *size* of a graph is the number of edges or arrows, i.e. the *cardinal* of the set E: $|E|$.
- The *degree* or *valence* of a vertex is the number of edges reaching or leaving the vertex (if an edge connects a vertex with itself it counts twice). For directed graphs,
  - the *in-degree* of a vertex is the number of edges that arrive to the vertex, and
  - the *out-degree* of a vertex is the number of edges coming out of the vertex.
- The vertices that belong to a single edge (i.e. the vertices of valence 1) are called *terminal* or *leaf* vertices.
- A vertex with valence larger than 2 is called *branching*.

Reminder of basic graph definitions: paths and loops

- A *path* is a linear sequence of connecting edges. When the graph is oriented, the end of an arrow must be the beginning of the next one.
- The *length* of a path is the number of its edges or arrows.
- A *loop* or *circuit* is a closed path. That is, the end of the last edge coincides with the beginning of the first one.
- A path is called *acyclic* if it does not contain any circuit or loop. Observe that a path is cyclic if and only if it has repeated vertices. Equivalently, a path is acyclic if and only if every vertex appears at most once in the path.

Basic graph definitions: Concatenation of paths

Given two paths

$$\alpha = \left(a_0 \longrightarrow a_1 \longrightarrow \cdots \longrightarrow a_n\right) \text{ of length } n, \text{ and}$$

$$\beta = \left(b_0 \longrightarrow b_1 \longrightarrow \cdots \longrightarrow b_m\right) \text{ of length } m,$$

such that $a_n = b_0$, we define the *concatenation of α and β*, denoted by $\alpha\beta$, as the path

$$\alpha\beta := \left(a_0 \to a_1 \to \cdots \to a_n \to b_1 \to \cdots \to b_m\right).$$

Observation: The length of $\alpha\beta$ is $n+m$, i.e. the addition of lengths of α and β.

Assume that α is a loop (i.e. $a_n = a_0$). In what follows we will use the following notation:

$$\begin{aligned}
\alpha^1 &:= \alpha,\\
\alpha^2 &:= \alpha\alpha,\\
\alpha^3 &:= \alpha^2\alpha = \alpha\alpha\alpha,\\
\cdots & \quad \cdots,\\
\alpha^n &:= \left(\alpha^{n-1}\right)\alpha = \overbrace{\alpha\alpha \cdots \alpha}^{n \text{ times}} \text{ for every } n \geq 2.
\end{aligned}$$

Weighted graphs

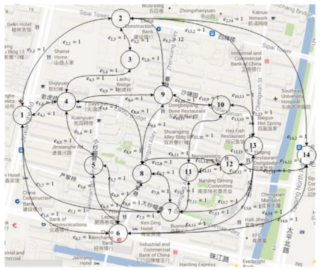
A *weighted graph*[2] or a *network* is a graph in which a number (the *weight*) is assigned to each edge (see the examples in Page 7). Such weights might represent for example costs, lengths or capacities, depending on the problem at hand.

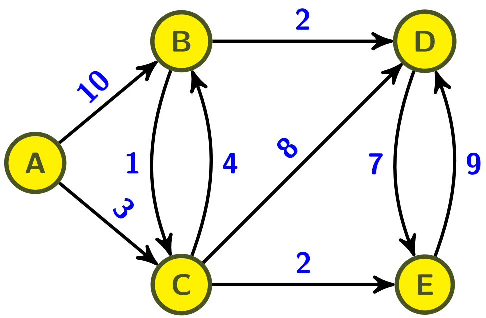
Notationally the weight associated to and edge or arrow is usually written above the edge or the arrow.

Also, we can encompass all the weights of a graph in a single *edge-weight function*:

$$\begin{array}{rccc}
\omega: & E & \longrightarrow & \mathbb{R}\\
 & a & \longmapsto & \omega(a)\\
 & (x,y) & \longmapsto & \omega((x,y))
\end{array}$$

[2] A weighted graph can be both directed and undirected.





Example on the edge-weight function: $\omega\big((C, D)\big) = 8$.

In a weighted graph, the *weight of a path* $\alpha = v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_n$ is defined to be

$$\omega(\alpha) := \sum_{i=1}^{n} \omega\big((v_{i-1}, v_i)\big).$$

Example (on the weighted graph at the right of Page 7)

Consider the following (weighted) path in the graph:

$$\alpha = A \xrightarrow{\ 10\ } B \xrightarrow{\ 1\ } C \xrightarrow{\ 4\ } B \xrightarrow{\ 2\ } D \xrightarrow{\ 7\ } E.$$

Then $\qquad \omega(\alpha) = 10 + 1 + 4 + 2 + 7 = 24.$

Observation

If $\alpha\beta$ is a concatenated path then, clearly,

$$\omega(\alpha\beta) = \omega(\alpha) + \omega(\beta).$$

The *minimum* or *optimum weight* of a path from a to b is defined as

$$\sigma(u, v) := \min\{\omega(\alpha) : \alpha \text{ is a path from } u \text{ to } v\}.$$

Convention: $\sigma(u, v) = \infty$ if no path from u to v exists.

Important observation (see the example in the next page)

The *minimum weight $\sigma(u, v)$ of a path* may not exist. However, when it exists it is uniquely defined.

A *minimal path from $u \in V$ to $v \in V$* is any path from u to v with weight $\sigma(u, v)$ (i.e. with minimum weight), whenever the minimum weight $\sigma(u, v)$ exists.

Observation: non-unicity of minimal paths

In general, there might be several minimal paths between a given pair of vertices.

Consider the weighted graph at the right of Page 7 with $\omega\big((C, B)\big) = 4$ replaced by $\omega\big((C, B)\big) = -4$. Consider also a family of paths

$$\alpha_n = \big(A \longrightarrow B\big)\big(B \longrightarrow C \longrightarrow B\big)^n\big(B \longrightarrow D \longrightarrow E\big)$$

with $n \geq 1$, similar to the ones from the previous example. Then,

$$\begin{aligned}\omega\big(\alpha_n\big) &= \omega\big(A \longrightarrow B\big) + \omega\Big(\big(B \longrightarrow C \longrightarrow B\big)^n\Big) + \omega\big(B \longrightarrow D \longrightarrow E\big)\\ &= \omega\big(A \longrightarrow B \longrightarrow D \longrightarrow E\big) + n\omega\Big(B \longrightarrow C \longrightarrow B\Big)\\ &= 19 - 3n.\end{aligned}$$

The *minimum weight $\sigma\big(A, E\big)$* of a path from A to E is not defined since in the graph there are such paths of arbitrarily small (negative) weight, because

$$\lim_{n\to\infty} \omega\big(\alpha_n\big) = \lim_{n\to\infty} 19 - 3n = -\infty.$$

Conclusion

All edge weights must be non-negative or, equivalently, the *edge-weight function* ω is a function from E to $\mathbb{R}^+$: $\qquad \omega\colon E \longrightarrow \mathbb{R}^+$.

More on weighted graphs

In the spirit of the previous page, a weighted graph (V, E, ω) will be called

- *non-negative* whenever $\omega(a) \geq 0$;
- *positive* if $\omega(a) > 0$; and
- *strongly positive* if there exists $\tau > 0$ such that $\omega(a) \geq \tau$

for every edge $a \in E$. Observe that a positive weighted graph is strongly positive whenever the graph has finite size.

The conclusion of the previous page is that the minimum weight (and hence the notion of optimal path) is only defined for non-negative weighted graphs. However, to assure the convergence of routing algorithms, for the single-source shortest paths problem, we will require that the graph is strongly positive.

Basic properties of shortest paths: Optimality Principle

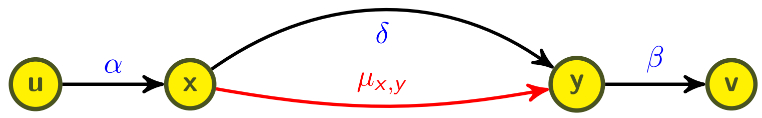
Theorem (Optimality principle)

Any sub-path of a minimal path is minimal.

Proof

Let $\alpha\delta\beta$ be a minimal (concatenated) path from u to v, where δ is a sub-path from x to y.

Assume by way of contradiction that δ is not a minimal path from x to y. Then there exists a path $\mu_{x,y}$ from x to y, such that $\omega(\mu_{x,y}) < \omega(\delta)$ (in particular, $\mu_{x,y} \neq \delta$). So, $\alpha\mu_{x,y}\beta$ is another path from u to v such that
$$\omega(\alpha\mu_{x,y}\beta) = \omega(\alpha) + \omega(\mu_{x,y}) + \omega(\beta) < \omega(\alpha) + \omega(\delta) + \omega(\beta) = \omega(\alpha\delta\beta);$$
which contradicts the assumption that $\alpha\delta\beta$ is a path from u to v of minimal weight.



Basic properties of shortest paths: Triangle Inequality

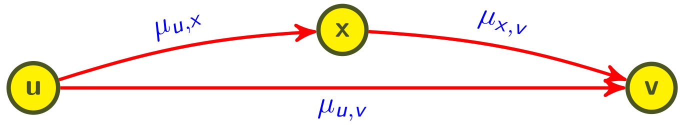
Theorem (Triangle Inequality)

For all $u, v, x \in V$, we have $\sigma(u, v) \leq \sigma(u, x) + \sigma(x, v)$.

Proof

Observe that if either does not exist path from u to x or from x to v, then $\sigma(u, x) + \sigma(x, v) = \infty$, and the lemma holds. Otherwise, let $\mu_{u,x}$ be a minimal path from u to x (i.e. $\omega(\mu_{u,x}) = \sigma(u, x)$), and let $\mu_{x,v}$ be a minimal path from x to v (i.e. $\omega(\mu_{x,v}) = \sigma(x, v)$).

The concatenated path $\mu_{u,x}\mu_{x,v}$ is clearly a path from u to v, and
$$\omega(\mu_{u,x}\mu_{x,v}) = \omega(\mu_{u,x}) + \omega(\mu_{x,v}) = \sigma(u, x) + \sigma(x, v).$$
Hence (by the definition of minimum weight)
$$\sigma(u, v) \leq \omega(\mu_{u,x}\mu_{x,v}) = \sigma(u, x) + \sigma(x, v).$$



The routing problem statement: Single-source shortest paths

The single-source shortest paths problem

Let (V, E, ω) be a strongly positive weighted graph. Given a *source* vertex $\xi \in V$, find a minimal path and the optimum path weight from ξ to every node from V.

The routing problem

Let (V, E, ω) be a strongly positive weighted graph. Given a *source* vertex $\xi \in V$ and a *goal* node[3] $\gamma \in V$, find a minimal path and the optimum path weight from ξ to γ.

The single-source shortest paths problem for standard (unweighted) graphs is usually formulated in a rooted graph, being the root the *source vertex*.

[3] The notation $\xi \in V$ to denote the *source vertex*, and $\gamma \in V$ for the *goal node* will be kept throughout the rest of the presentation.

The single-source shortest paths problem for unweighted graphs: Breadth-first search

The single-source shortest paths problem for unweighted graphs

Let (V, E) be an unweighted graph or, equivalently, let (V, E, ω) be a weighted graph with constant weight function ω; i.e. $\omega(a) = 1$ for every $a \in E$.

Given a *source* vertex $\xi \in V$, find a minimal path and the optimum path weight from ξ to every node from V.

As it is well known, this is equivalent to the computation of the *depths* of all nodes from a graph, with the source node as *root*.

This problem can be solved in time $\mathcal{O}(|V| + |E|)$ by the *Breadth-first search algorithm* (by means of a FIFO queue). The BFS algorithm computes a *minimal spanning tree* of the graph.

Graphs: Definitions and Basic Algorithms, Pages 50 to 70, `http://mat.uab.cat/~alseda/MatDoc/GrafsDefimovs-en.pdf`

Dijkstra's Algorithm

Contents

Introduction to Dijkstra's Algorithm

Dijkstra's algorithm is designed to solve the *single-source shortest paths problem* by computing a minimal spanning tree.

It can also solve the *routing problem* by stopping the algorithm once the shortest path to the destination node has been determined.

Dijkstra's algorithm is based on a (controlled) *greedy strategy*; that is, it makes a local optimal choice at every stage[4].

[4] A greedy strategy does not usually produce an optimal solution by itself.

Dijkstra's Algorithm in pseudocode

Dijkstra's Algorithm for graphs, using an efficient priority queue

```
procedure DIJKSTRA(graph G, source)
    Pq ← EmptyPriorityQueue                          ▷ Declaration and initial assignment:
    expanded[G.order] ← initialized to false           expanded[v] = true ⟺ v is extract_min-
    dist[G.order] ← initialized to ∞                   taken-out from the list and expanded
    parent[G.order] ← uninitialized                    dist: distances vector from source to every node
                                                       parent: previous vertices in an optimal path
    dist[source] ← 0                                 ▷ Initialization: source has distance 0 to
    parent[source] ← ∞                                 itself, has no parent and is enqueued
    Pq.add_with_priority(source, dist[source])
    while (not Pq.IsEmpty) do                                                 ▷ The main loop
        node ← Pq.extract_min()               ▷ extract_min removes a node with minimal dist from Pq
        expanded[node] ← true   ▷ node has been removed from the priority queue and will be expanded
        for each adj ∈ node.neighbours and not expanded[adj] do   ▷ New cost from source to
            dist_aux ← dist[node] + ω(node, adj)                    adj through node
            if (dist[adj] > dist_aux) then                                    ▷ Relaxation step
                if (dist[adj] = ∞) then Pq.add_with_priority(adj, dist_aux)
                else Pq.decrease_priority(adj, dist_aux)
                end if
                dist[adj] ← dist_aux
                parent[adj] ← node
            end if
        end for
    end while
    return dist, parent
end procedure
```

Comments on Dijkstra's Algorithm

dist[v] $= \infty$ for some vertex v

This will happen at termination whenever the vertex v is unreachable form the source. This may indicate that the graph is not connected or that it is directed and there is no (direct) path from the source vertex to v.

How the minimal spanning tree is specified?

Through the vectors `dist` and `parent`.

- `dist[v]` gives the computed optimal distance from source to the vertex `v`.
- `parent[v]` specifies the predecessor of the node `v` in a shortest path.

Thanks to the vector `parent` we can backwards construct the computed optimal paths to all vertices, thus building a minimal spanning tree.

Comments on Dijkstra's Algorithm

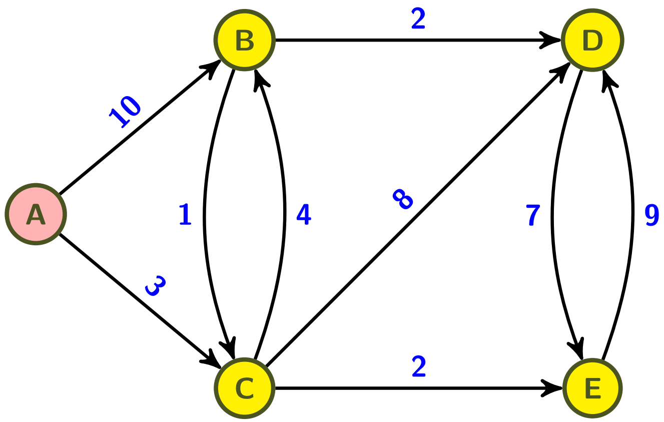
Consequences of the necessity of the `extract_min` function

The most important operation to be performed with the queue is the `extract_min` function.

As a consequence, the queue management completely determines the efficiency of the algorithm (see the *Analysis of Dijkstra's Algorithm efficiency* starting in Slide 52, and specially Slide 53). This analysis shows that a plain FIFO queue (as in the Breadth First Search Algorithm) is not the best option here, and that we rather have to use a priority queue.

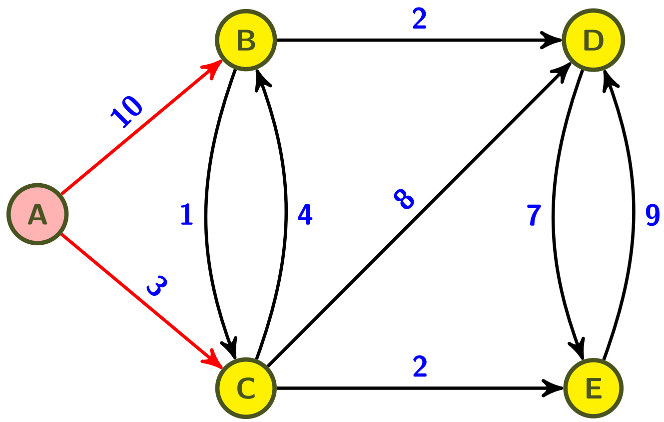
Different strategies of queue management will be discussed in the part *Queue management strategies* starting in Slide 28.

An example of the Dijkstra's Algorithm



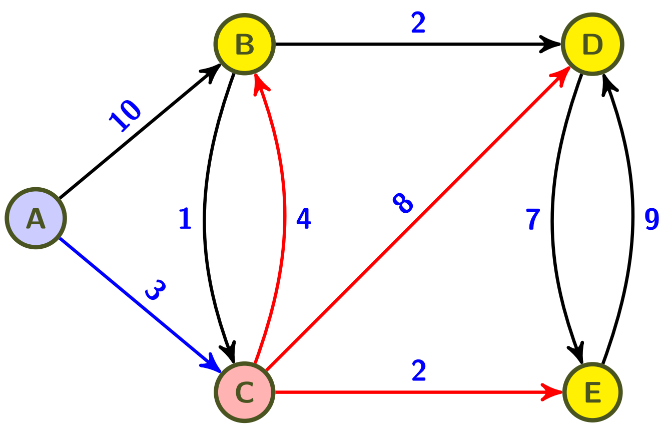
PriQueue	A
dist	0
parent	nil

An example of the Dijkstra's Algorithm



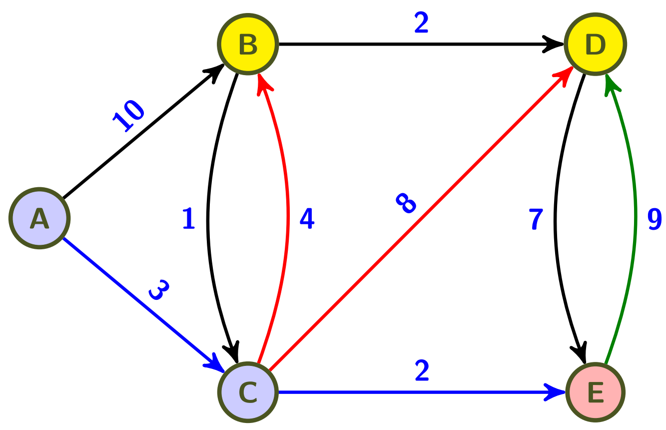
expanded	A
dist	0
parent	nil

PriQueue	C	B
dist	3	10
parent	A	A



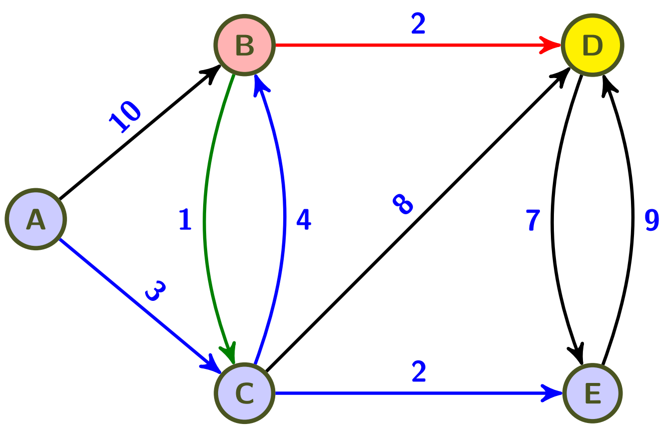
expanded	A	C
dist	0	3
parent	nil	A

PriQueue	E	B	D
dist	5	7	11
parent	C	C	C



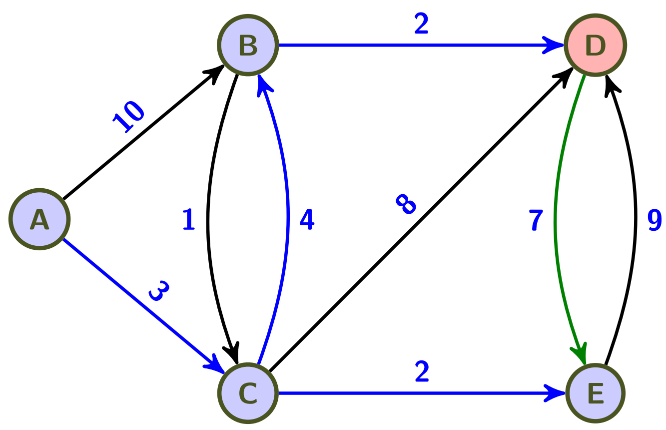
expanded	A	C	E
dist	0	3	5
parent	nil	A	C

PriQueue	B	D
dist	7	11
parent	C	C



expanded	A	C	E	B
dist	0	3	5	7
parent	nil	A	C	C

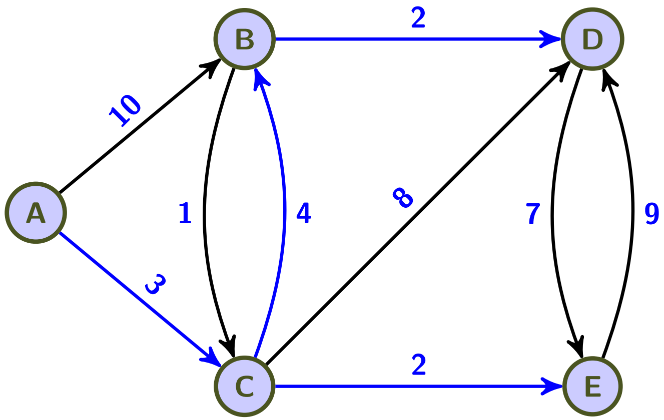
PriQueue	D
dist	9
parent	B



expanded	A	C	E	B	D
dist	0	3	5	7	9
parent	nil	A	C	C	B

PriQueue
dist
parent

An example of the Dijkstra's Algorithm



expanded	A	C	E	B	D
dist	0	3	5	7	9
parent	nil	A	C	C	B

Convergence of Dijkstra's Algorithm

The convergence of Dijkstra's Algorithm is assured by the next

Theorem

The equality `dist[v]` $= \sigma($`source`$, v)$ holds whenever a vertex $v \in V$ is dequeued (with the function `extract_min`) and expanded, and it is maintained during the rest of the algorithm. In particular, Dijkstra's algorithm terminates with `dist[v]` $= \sigma($`source`$, v)$ for every vertex $v \in V$.

To prove this theorem we will use the following two lemmas:

DA–Lemma 1

The inequality `dist[v]` $\geq \sigma($`source`$, v)$ holds at every iteration of the algorithm, for every vertex $v \in V$.

DA–Lemma 2

Let α be a minimal path from `source` to a vertex $v \in V$. Let u be the predecessor of v in α, and assume that `dist[u]` $= \sigma($`source`$, u)$. Then, if the edge (u, v) is relaxed we have `dist[v]` $= \sigma($`source`$, v)$ after the relaxation.

Convergence of Dijkstra's Algorithm (II)

DA–Lemma 1

The inequality `dist[v]` $\geq \sigma($`source`$, v)$ holds at every iteration of the algorithm, for every vertex $v \in V$.

Proof of DA–Lemma 1

The initial assignment

dist[] ← **initialized to** ∞
dist[source] ← 0

guarantees that `dist[v]` $\geq \sigma($`source`$, v)$ holds for every vertex $v \in V$ when the algorithm starts (before the `while` loop).

Now we will prove that these inequalities are maintained during the whole algorithm. Assume by way of contradiction that there exists a first vertex v for which `dist[v]` $< \sigma($`source`$, v)$. Let u be the vertex that caused `dist[v]` to change (by setting `dist[v]` = `dist[u]` $+ \omega(u, v)$ at a relaxation step). We have,

$$
\begin{aligned}
\texttt{dist[v]} &< \sigma(\texttt{source}, v) && \triangleright \textit{ assumption} \\
&\leq \sigma(\texttt{source}, u) + \sigma(u, v) && \triangleright \textit{ triangle inequality} \\
&\leq \sigma(\texttt{source}, u) + \omega(u, v) && \triangleright \textit{ optimal path has weight smaller than or equal to the weight of a specific path} \\
&\leq \texttt{dist[u]} + \omega(u, v) = \texttt{dist[v]}; && \triangleright \textit{ v is the first vertex for which } \texttt{dist[v]} < \sigma(\texttt{source}, v)
\end{aligned}
$$

a contradiction.

Convergence of Dijkstra's Algorithm (III)

DA–Lemma 2

Let α be a minimal path from `source` to a vertex $v \in V$. Let u be the predecessor of v in α, and assume that `dist[u]` $= \sigma($`source`$, u)$. Then, if the edge (u, v) is relaxed we have `dist[v]` $= \sigma($`source`$, v)$ after the relaxation.

Proof of DA–Lemma 2

The minimality of α and the Optimality Principle imply that

$$\sigma(\texttt{source}, v) = \omega(\alpha) = \sigma(\texttt{source}, u) + \omega(u, v).$$

Observe that when the value of `dist[v]` is modified by the algorithm, it decreases strictly. Assume that, at some step of the algorithm, `dist[v]` $\leq \sigma($`source`$, v)$. By DA–Lemma 1 we have that `dist[v]` $= \sigma($`source`$, v)$ until the end of the algorithm. Thus, the lemma holds in this case.

Suppose now that `dist[v]` $> \sigma($`source`$, v)$ before the relaxation. We have,

$$\texttt{dist[v]} > \sigma(\texttt{source}, v) = \sigma(\texttt{source}, u) + \omega(u, v) = \texttt{dist[u]} + \omega(u, v).$$

Then, during the relaxation step the algorithm sets

$$\texttt{dist[v]} = \texttt{dist[u]} + \omega(u, v) = \sigma(\texttt{source}, v).$$

Convergence of Dijkstra's Algorithm (IV)

Theorem (Convergence of Dijkstra's Algorithm)

The equality `dist[v]` $= \sigma($`source`$, v)$ holds whenever a vertex $v \in V$ is dequeued (with the function `extract_min`) and expanded, and it is maintained during the rest of the algorithm. In particular, Dijkstra's algorithm terminates with `dist[v]` $= \sigma($`source`$, v)$ for every vertex $v \in V$.

Proof of Theorem

If `dist[v]` $= \sigma($`source`$, v)$ holds whenever a vertex $v \in V$ is dequeued, then this equality is maintained during the rest of the algorithm because of DA–Lemma 1 and the fact that the values `dist[v]` cannot increase during the computation.

So, we only need to prove the first statement of the theorem. Assume that $v \in V$ is the first vertex for which the inequality `dist[v]` $\neq \sigma($`source`$, v)$ holds at the moment of dequeueing it with the function `extract_min`. Note that, by DA–Lemma 1, in fact we have `dist[v]` $> \sigma($`source`$, v)$.

Let us denote by S the set of vertices $u \in V$ that have been already dequeued with the function `extract_min` and expanded. Clearly,

- `source` $\in S$,
- $v \notin S$ because the algorithm is just going to dequeue v, and
- since v is the first vertex that will be dequeued with `dist[v]` $> \sigma($`source`$, v)$, the equality `dist[u]` $= \sigma($`source`$, u)$ holds for every vertex $u \in S$ whenever it is dequeued, and it is maintained during the rest of the algorithm.

Convergence of Dijkstra's Algorithm (V)
Proof of the Theorem

Proof of Theorem (continued)

Let β be a minimal path from `source` to v. Since `source` $\in S$, there exist vertices $x, y \in V$ such that:

1. (x, y) is an edge of β,
2. $y \notin S$, and
3. every vertex lying in the sub-path of β from `source` to x (including x) belongs to S.

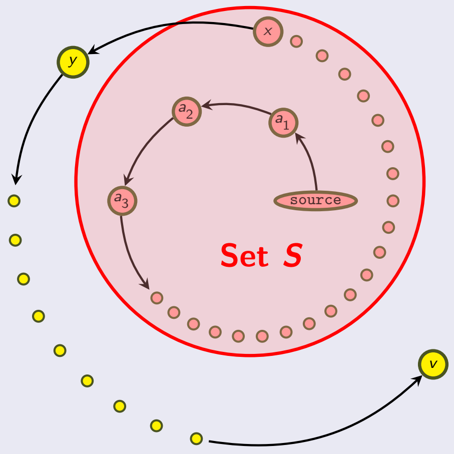
When the vertex x was dequeued and added to S, we had

`dist[x]` $= \sigma($`source`$, x)$,

and the edge (x, y) was relaxed. By DA–Lemma 2 with v replaced by y, u replaced by x, and α replaced by the sub-path of β from `source` to y (notice that α is a minimal path by the Optimality Principle), we get

`dist[y]` $= \sigma($`source`$, y)$

after the relaxation of (x, y).



Convergence of Dijkstra's Algorithm (VI)
Proof of the Theorem

Proof of Theorem (end)

Since $y \notin S$, then either `dist[y]` $= \infty >$ `dist[v]` (recall that every node in the queue has finite `dist` value), or y is in the queue and `dist[v]` $\leq$ `dist[y]` because v is being dequeued with `extract_min`.

On the other hand, since v is farther from `source` than y in the minimal path β, we have $\sigma($`source`$, y) \leq \sigma($`source`$, v)$.

Then, summarizing,

$$\boxed{\texttt{dist[v]} \leq \texttt{dist[y]} = \sigma(\texttt{source}, y) \leq \sigma(\texttt{source}, v) < \texttt{dist[v]};}$$

a contradiction.

Queue management strategies

Here we will describe and comment four alternative queue management strategies towards the efficient use (and implementation) of the `extract_min` function.

Boolean State Vector

In this strategy the list is implemented as a vector of bolean type (to store `true` or `false` values) of fixed size `order` (such as `IsNodeInQueue[order]`), which works as follows: A node v is in the queue if and only if `IsNodeInQueue[v] = true`.

Comments: This strategy wastes a lot of memory (uses during the whole algorithm the same amount of memory of a queue having *all nodes* in it), and really gives a "worst case scenario" for the search of the queue element with minimum cost. Indeed, the whole boolean state vector has to be checked to detect which nodes belong to the queue and, for each of them, its costs has to be compared with the current minimum cost candidate.

Queue management strategies

Plain linked list (not sorted)

The indices of the vertices in the queue are stored as a plain linked list (see the document below).

Comments: The memory use of this strategy is minimal: just one integer per node in the queue (to store the index), and the memory used by the pointers in the list maintenance. Moreover, the queue automatically resizes itself to have length equals to the number of enqueued nodes. However, the function `extract_min` is very inefficient: first one has to run the whole queue to determine the node with minimum cost; second one has to run again thee queue to travel to that node to dequeue it.

Queues implementation with a plain linked list can be seen at:

- *Tipus de Dades, Estructures i Llistes en in **C**: Stacks i Cues*, `http://mat.uab.cat/~alseda/MatDoc/DadesEstructuresLlistes-StacksICues.pdf`

Queue management strategies

Linked list sorted by priority (cost)

The indices of the vertices in the queue are stored as a linked list, but the list must be sorted according to cost (being the first list element the one with a vertex with a smaller cost, and the last list element the one with a vertex with a larger cost) *at every step of the algorithm*.

This has the following consequences:

- The function `extract_min` is trivial: One has to systematically dequeue the first element in the list.
- The function `enqueue` must choose the right place to insert a new element in the list according to its cost, to maintain the assumption that the list is sorted according to cost at every step of the algorithm.

Queue management strategies

Linked list sorted by priority (cost) — continued

- The function `decrease_priority` or `requeue` (that had nothing to do in the previous two strategies and was, in fact, useless), now has to keep the ordering of the list. This must be done by (perhaps) moving the "relaxed vertex" (which has decreased its cost) to a new specific place closer to the beginning of the list.

Comments: As for a plain linked list, the memory usage of this strategy is minimal for the same reasons. The function `extract_min` is trivial but the management of the list (`enqueue` and `requeue` functions) is a bit more involved.

In the following slides we will discuss the *Binary Heap* priority queue strategy. We will forget about *Fibonacci Heap* priority queue strategy, which seems to be the most efficient one but rather difficult to implement.

Queue management strategies: Binary Heap priority queues

The Binary Heap priority queues strategy is exactly the same as with the priority queues that use a linked list (sorted by cost) but using a binary arrangement (binary tree) as storage data type for the queue instead of using a linear one.

In particular, the same comments apply as for the case of linked lists sorted by priority: The memory usage is minimal but the management of the list is a bit more involved. However, the function `extract_min` is a bit more complicate than the one for linked lists sorted by priority because it destroys the binary structure that must be reconstructed.

Queue management strategies: Binary Heap priority queues

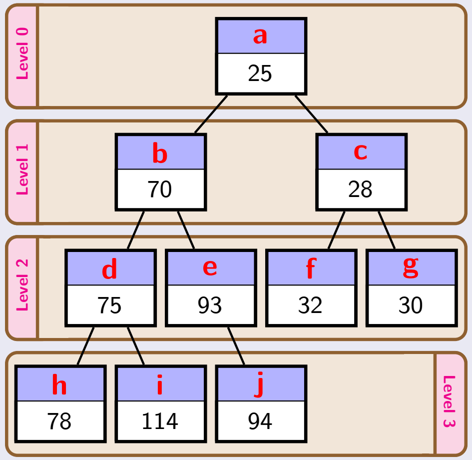
Definition

A *binary heap* is a binary tree with a *value* stored at every node which verifies the following two basic properties:

Shape Property (Completeness): all levels of the tree except possibly the last (deepest) one are completely filled (with two children per node) and, if the last level of the tree is not complete, the nodes at this level are filled from left to right.

Heap Property: the *value* stored in each node is less than or equal to the values stored at the node's children, according to some total order.

A Binary Heap Example



The *depth* of a node is also called its *level* in binary heap notation.

Binary Heap priority queues

Comments on the item with lowest value

On the item with lowest value and the function `extract_min`

The iterative use of the Heap Property tells us that the value of a given node N is smaller than or equal to the values of all nodes in the subtree that has the node N as root.

In particular, the root of the whole binary heap (the only node that is at level 0) is the item with the lowest value (see the example in the previous page).

Consequently, when the priority queue uses a binary heap, the function `extract_min` does not need to perform any search to do its task.

Binary Heap priority queues

Comments on the shape and the number of elements of a binary heap

The shape of a binary heap can be characterized by the *number of levels* ℓ, and the *number of nodes* r *in the last level*.
By convention ℓ is zero if and only if the binary heap is empty.

Observations

- When the binary heap is non-empty (that is, when $\ell > 0$), the levels are numbered $0, 1, \dots, \ell - 1$, according to their *depth* in the tree.
- Every level $n \in \{0, 1, \dots, \ell - 1\}$ has at most 2^n nodes. By the completeness property, the levels $n \in \{0, 1, \dots, \ell - 2\}$ have exactly 2^n nodes, and the last level $\ell - 1$ has $1 \le r \le 2^{\ell-1}$ elements. Consequently, when $\ell = 1$ the last level (which is level 0) is necessarily full.
- The total number of nodes in a non-empty binary heap is:
$$T := \begin{cases} 1 & \text{when } \ell = 1, \text{ and} \\ 2^0 + 2^1 + \dots + 2^{\ell-2} + r = \left(2^{\ell-1} - 1\right) + r & \text{when } \ell \ge 2. \end{cases}$$
Moreover, the maximum number of nodes that can be stored in a binary heap with ℓ levels is $2^\ell - 1$.
Example: The binary heap in the previous figure has $\ell = 4$ levels, and at the last level there are $r = 3$ nodes. So, in total it has $T = \left(2^{4-1} - 1\right) + 3 = 10$ nodes.

Binary Heap priority queues

Basic operation procedures

`IsEmpty`

Tells whether a binary heap is empty. Equivalently it checks whether the number of levels of the binary heap is zero or positive.

`dequeue` or, equivalently, `extract_min`

Reads and deletes the root node (see Slide 34). This leaves a broken heap that must be repaired to a new "legal" one.

`enqueue`

Adds a new node to a binary heap so that the new binary heap is "legal".

`requeue`

Used when a node that is already in the binary heap has changed its value to a lower one. In this case the shape property is still maintained but not necessarily the heap property since the node to be re-queued may have a new value smaller than the one of its parent.

Basic Operation: indexing a binary heap and auxiliary low-level procedures

Indexing a binary heap

A node in a binary heap is indexed (located) by a pair ($\mathtt{d},\mathtt{p}$) where:

- $\mathtt{d}$ is the depth (level) at which the node is located in the binary heap. Of course $\mathtt{d} \in \{0, 1, \ldots, \ell - 1\}$.
- $\mathtt{p}$ is the position (from left to right) occupied by the node in the level $\mathtt{d}$. Then,

$$\mathtt{p} \in \begin{cases} \{0, 1, \ldots, 2^{\mathtt{d}} - 1\} & \text{if } \mathtt{d} \le \ell - 2, \text{ and} \\ \{0, 1, \ldots, r - 1\} & \text{if } \mathtt{d} = \ell - 1. \end{cases}$$

Observation: When $\mathtt{d} = 0$, ($\mathtt{d},\mathtt{p}$) must be $(0, 0)$, which is the root node.

$\mathtt{parentOf}(\mathtt{d},\mathtt{p})$ is defined[5] only for $\mathtt{d} > 0$

$$\mathtt{parentOf}(\mathtt{d},\mathtt{p}) = \left(\mathtt{d} - 1, \left\lfloor \tfrac{\mathtt{p}}{2} \right\rfloor\right),$$

where for $x \in \mathbb{R}$, $\lfloor x \rfloor$ denotes the *integer part function* or *floor function* which gives, by definition, the greatest integer less than or equal to x.

$\mathtt{leftchildOf}(\mathtt{d},\mathtt{p})$ and $\mathtt{rightchildOf}(\mathtt{d},\mathtt{p})$ are only defined when $\mathtt{d} < \ell - 1$

$\mathtt{leftchildOf}(\mathtt{d},\mathtt{p}) = (\mathtt{d} + 1, 2\mathtt{p})$ and $\mathtt{rightchildOf}(\mathtt{d},\mathtt{p}) = (\mathtt{d} + 1, 2\mathtt{p} + 1)$.

[5] When $\mathtt{d} = 0$, $(\mathtt{d},\mathtt{p}) = (0, 0)$ is the root node whose parent is undefined.

Super Basic Operations: the `heapify_up` low-level standard procedure

Heapify Up Assumptions

We have a non-empty binary heap which verifies the shape property, and there exists a unique node ($\mathtt{d},\mathtt{p}$) such that $\mathtt{valueOf}(\mathtt{d},\mathtt{p}) < \mathtt{valueOf}(\mathtt{parentOf}(\mathtt{d},\mathtt{p}))$ (that is, the heap property is satisfied for all nodes except for ($\mathtt{d},\mathtt{p}$)).

Observation: The node ($\mathtt{d},\mathtt{p}$) cannot be the root since it has no parent. In particular $\ell \ge 2$ (i.e. the number of levels is larger than one).

Algorithm: The `heapify_up` repairing procedure

```
procedure HEAPIFY_UP(d,p)            ▷ The input is the only node that breaks the heap property
  while d > 0 and valueOf(d,p) < valueOf(parentOf(d,p)) do   ▷ When d = 0 no repair is needed
    node_aux ← node(d,p)                     }
    node(d,p) ← node(parentOf(d,p))          }  ▷ Swapping the nodes
    node(parentOf(d,p)) ← node_aux           }    (d,p) and parentOf(d,p)
    (d,p) ← parentOf(d,p)    ▷ Now parentOf(d,p) is the node that perhaps breaks the heap
                               property, and is going to be checked in the next iteration
  end while
end procedure
```

Observation

In the whole procedure above the shape property is maintained. So, we only have to take care of repairing the heap property.

Super Basic Operations: the `heapify_down` low-level standard procedure

Heapify Down Assumptions

We have a non-empty binary heap which verifies the shape property, and there exists a unique node ($\mathtt{d},\mathtt{p}$) which has at least one child and such that either

$\mathtt{valueOf}(\mathtt{d},\mathtt{p}) > \mathtt{valueOf}(\mathtt{leftchildOf}(\mathtt{d},\mathtt{p}))$, or

$\mathtt{valueOf}(\mathtt{d},\mathtt{p}) > \mathtt{valueOf}(\mathtt{rightchildOf}(\mathtt{d},\mathtt{p}))$ (if $\mathtt{rightchildOf}(\mathtt{d},\mathtt{p})$ exists).

In particular, the heap property is satisfied for all nodes except for ($\mathtt{d},\mathtt{p}$) (if a node has no children, then it automatically satisfies the heap property).

Observation: When a node has a unique child, it must compulsory have the left child by the shape property. The assumption above that the node ($\mathtt{d},\mathtt{p}$) has at least one child implies that $\mathtt{d} < \ell - 1$ (i.e., the node ($\mathtt{d},\mathtt{p}$) cannot belong to the last level), and if $\mathtt{d} = \ell - 2$, then $\mathtt{p} \le \left\lfloor \frac{r-1}{2} \right\rfloor$. In particular, again, $\ell \ge 2$ (i.e. the number of levels is larger than one).

Super Basic Operations: the `heapify_down` low-level standard procedure

Algorithm: The `heapify_down` repairing procedure

```
procedure HEAPIFY_DOWN(d,p)          ▷ The input is the only node that breaks the heap property
  while d < ℓ − 1 and Exist(leftchildOf(d,p)) do   ▷ Otherwise (d,p) has no children,
                                                     and hence it does not break the
                                                     heap property; no repair is needed
    ◁ Determining smallson: the child with lower cost
    { smallson ← leftchildOf(d,p)
    { if Exist(rightchildOf(d,p)) and valueOf(rightchildOf(d,p)) < valueOf(smallson) then
    {     smallson ← rightchildOf(d,p)
    { end if
    if valueOf(d,p) ≤ valueOf(smallson) then
        return          ▷ The heap property is already verified by (d,p); no repair is needed
    end if
    node_aux ← node(d,p)               }
    node(d,p) ← node(smallson)         }  ▷ Swapping the nodes (d,p) and smallson
    node(smallson) ← node_aux          }
    (d,p) ← smallson    ▷ Now smallson is the node that perhaps breaks the heap property, and is
                          set to be checked in the next iteration
  end while
end procedure
```

Observation

In the whole procedure above the shape property is maintained. So, we only have to take care of repairing the heap property.

Binary Heap priority queues
Basic Operation: the `requeue` procedure

`requeue` (or, equivalently, `heapify_up`):
A node already in the binary heap has changed its value to a lower one

In this case the shape property is still maintained but not necessarily the heap property, since the node to be re-queued may have a new value smaller than the one of its parent (provided that it has parent; i.e., is not the root node).

In this case the node to be re-queued verifies the assumptions of the Heapify Up procedure, and to repair the heap we only need to use the function `heapify_up`.

In other words, `requeue` and `heapify_up` are the same functions.

Binary Heap priority queues
Basic Operation: strategy for the `enqueue` procedure

Step 0: If the binary heap is empty the node is added as the unique level 0 element. This heap fulfils both the shape and heap properties.

Step 1: **`Adding the node without breaking the shape property.`** The new node is added to the last level, consecutively to the existing nodes (leaving no holes). If the last level is already full (it is level `d` and has 2^d elements), a new level `d+1` is created with the new node as the only element.

Step 2: **`Repairing the heap to fulfil the heap property.`** In this case either the binary heap already verifies the heap property, or the added node verifies the assumptions of the Heapify Up procedure. In this second case, to repair the heap we only need to use the `heapify_up` function.

Binary Heap priority queues
Basic Operation: the `extract_min` or `dequeue` procedure

Step 1: Read the root node (by the heap property it is the one we are looking for), and eliminate it. Observe that the root node cannot be deleted since the remaining object, if it is not-empty, does not verify the shape property (it is not a tree any more; it has become two disconnected binary trees).

Step 2: **`Replace the root node by the last node of the binary heap (the rightmost one of the last level), and delete this last node.`** This removes the old root node from the queue and reduces the size of the binary heap by 1 (because we delete the last node) *without breaking the shape property*.

Step 3: **`Repairing the heap to fulfil the heap property.`** The result of Step 2, with very high probability, does not verify the heap property. Specifically, all nodes will verify the heap property except, perhaps, the new root node which will have a cost too large.

However, observe that the new root node verifies the assumptions of the Heapify Down operation. Thus, the repairing of the heap is achieved simply by using the `heapify_down` function starting with the root node.

Binary Heap priority queues
Binary Heap: Comments on implementation and data types

In the implementation of almost every algorithm it is crucial to choose the right abstract data type (for efficiency and programming easiness).

In the case of a binary heap, it seems reasonable to use a binary tree. However this is not recommended because it creates some programming complications and subtleties. For instance the binary tree must be bi-directional to allow going from children to parent and from parent to children, and this must be dealt appropriately in the code.

Surprisingly enough, in most applications, a binary heap is implemented as a (linear) *consecutive levels vector* of length $2^{\nu}-1$, where ν is the maximum number of levels allowed. The idea is that all levels are stored consecutively in the vector: first the unique element of level 0; second the two elements from level 1; third the four elements from level 2; etc.

This has a certain level of inefficiency due to three serious

Drawbacks:

1. The management of the binary heap through this data structure needs to map the 2-dimensional coordinates (`d`,`p`) to 1-dimensional vector indexes (in fact the parent and children functions are programmed directly in linear coordinates for efficiency). More concretely, the element (`d`,`p`) of the binary heap is stored at the position $(2^{\mathtt{d}}-1)+\mathtt{p}$ in the consecutive levels vector.

2. The maximum number of elements (and levels) that the queue may have is fixed a priori. So, unless the consecutive levels vector is enormous, we can easily run out of space for the queue provoking an undesirable error (or having to use the horrible `realloc` function).
3. The memory consumption does not adapt to the queue size at any moment of the algorithm. The queue uses $2^{\nu}-1$ memory positions all the time, independently on the queue size.

The abstract data model we propose for a binary heap implementation is a "triangular matrix" where every row of the matrix is a level. Thus, the row $\mathtt{d}=0,1,2,\dots$ has size $2^{\mathtt{d}}$.

This can be done by means of the following declarations:

```
#define MAXNumlevels 32
typedef struct {
    short ell; // Initially set to zero
    unsigned long r;
    unsigned int * level[MAXNumlevels];
} Binary_Heap_Priority_Queue;
```

Observe that this data type does not assign memory to any level. To do it, one must use the `malloc` function.

Example: The initialization of a new (last) level of a queue `Binary_Heap_Priority_Queue Q` can be done as follows:

```
Q.level[Q.ell] = (unsigned *) malloc((1LU << Q.ell)*sizeof(unsigned));
ell = ell + 1;
r = 1; // The new level must be non-empty.
```


- `No need of mapping 2-dim to 1-dim coordinates`
  Every element of this arrangement can be directly indexed by a pair (`d`,`p`), and can be accessed by the simple expression
  `Q.level[d][p]`
- `On the memory consumption of this data type:`
  1. No level is assigned memory a priori.
  2. A level is initialized and assigned memory only when it is needed (to store queue elements).
  3. When a level becomes empty it is freed to save memory.

  Therefore the memory use in this data type is adapted to the number of elements in the queue except for:

  1. the vector `unsigned * level[MAXNumlevels]` which uses `MAXNumlevels * sizeof(unsigned *)` bytes during the whole algorithm as a management fixed cost, and
  2. the $2^{\ell}-r$ positions in the last level, that are not used. Observe that if ℓ is large this can temporally waste a lot of memory.

We assume that the routing graph is stored by using the *adjacency list memory model* as a vector of structures (of size the `order` of the graph), and each of these structures contains the information corresponding to one of the vertices.

To add a vertex to the queue it is customary to store its index in the graph vector (an unsigned integer). In 64 bit systems, the `unsigned int` variables and constants use 4 bytes of memory, and can store numbers up to $2^{32} - 1$.

The minimal necessary information relative to a node is: the *path cost* and *parent* computed by the Dijkstra's Algorithm, the *number of adjacent vertices*, and the *index* and *edge-weight* of each adjacent vertex. Assuming that the number of adjacent vertices is of type `unsigned short` (2 bytes), that the rest of variables use 4 bytes each, and that there is a unique vertex adjacent to each vertex, we get that the bare minimum of memory to store a vertex is 18 bytes.

If the graph has 2^{32} vertices (indexed from 0 to $2^{32} - 1$), the total amount of memory used by the graph is:

$$2^{32}{}_{\text{vertices}} \cdot 18 \tfrac{\text{bytes}}{\text{vertex}} \cdot 2^{-30} \tfrac{\text{Giga bytes}}{\text{byte}} = 72_{\text{Giga bytes}}.$$

This is already absolutely prohibitive for real applications.

So, for us, there do not exist graphs with more than 2^{32} vertices. Consequently:

- The indices of the graph vector fit in `unsigned int` variables, and the queue elements (i.e., the elements of the `level[d]` vectors) can be declared of type `unsigned int`.
- Since the maximum number of nodes that can be stored in a binary heap with ℓ levels is $2^{\ell} - 1$, there is no reason to set `MAXNumlevels` larger than 32. In such case we would not have enough node indexes to fill the binary heap priority queue.

- **`The binary heap priority queue is of size both limited and virtually infinite: A proper explanation.`** We already know that we will not have a routing graph with more than $2^{32} - 1$ vertices. So, we can put all vertices to the queue without problems. In other words, we will never see the error: *heap full: Unable to add more elements to the heap. Aborting . . .* .

  In the limit case when the queue is full, all levels in the binary heap have been initialized and assigned memory. So, the total amount of memory used by the queue in this case is a little bit more than

$$\left(2^{32} - 1\right)_{\text{queue positions}} \cdot 4 \tfrac{\text{bytes}}{\text{queue position}} \cdot 2^{-30} \tfrac{\text{Giga bytes}}{\text{byte}} \approx 16_{\text{Giga bytes}},$$

  which is really affordable.

  What is not affordable is to have a graph with more than $2^{32} - 1$ vertices, which gives $88_{\text{Giga bytes}}$ as an unrealistically low estimate of the necessary memory for this exercise.

Dijkstra's Algorithm for graphs, using a priority queue
Repetitive part — omitting initialization

```
while (not Pq.IsEmpty) do
    node ← Pq.extract_min()
    expanded[node] ← true
    for each adj ∈ node.neighbours and not expanded[adj] do
        dist_aux ← dist[node] + ω(node, adj)
        if (dist[adj] > dist_aux) then
            if (dist[adj] = ∞) then Pq.add_with_priority(adj, dist_aux)
            else Pq.decrease_priority(adj, dist_aux)
            end if
            dist[adj] ← dist_aux
            parent[adj] ← node
        end if
    end for
end while
```

▷ • *Average time taken by the function* `extract_min`*:* T_{EM}
• `node` *runs among all possible graph nodes* $\Longrightarrow$ *The* `while` *loop runs for* $|V|$ *repetitions*

▷ *Loop iterating over all possible graph edges* (`node, adj`) $\Longrightarrow$ *The loop runs for at most* $|E|$ *repetitions*

- Average time taken by the function `add_with_priority`: T_{AwP}
- `add_with_priority` is run $|V|$ times since every node must be added to the queue, and it enters to it exactly once

- Average time taken by the function `decrease_priority`: T_{DP}
- `decrease_priority` is run $|E| - |V|$ times

Estimated average execution time

$$\boxed{|V|\left(T_{\text{EM}} + T_{\text{AwP}}\right) + \left(|E| - |V|\right) T_{\text{DP}}}$$

Analysis of Dijkstra's Algorithm efficiency

Table of estimated average run times for the three main functions of the Dijkstra's Algorithm in terms of the queue management strategy

Queue management	T_{EM}	T_{AwP}	T_{DP}	Total	Order
State Vector boolean	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert^2+\lvert E\rvert)$	$\mathcal{O}(\lvert V\rvert^2)$
Plain linked list not sorted	$\mathcal{O}(\overline{Q})$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert\overline{Q}+\lvert E\rvert)$	$\mathcal{O}(\lvert V\rvert^2)$
Linked list sorted by priority	$\mathcal{O}(1)$	$\mathcal{O}(\overline{Q})$	$\mathcal{O}(\overline{Q})$	$\mathcal{O}(\lvert V\rvert+\lvert E\rvert\overline{Q})$	$\mathcal{O}(\lvert E\rvert\overline{Q})$
Binary Heap priority queue	$\mathcal{O}(\log_2(\overline{Q}))$	$\mathcal{O}(\log_2(\overline{Q}))$	$\mathcal{O}(\log_2(\overline{Q}))$	$\mathcal{O}((\lvert V\rvert+\lvert E\rvert)\log_2(\overline{Q}))$	$\mathcal{O}(\lvert E\rvert\log_2(\overline{Q}))$

Notation

- $\overline{Q}$ denotes the average number of elements in the queue during the whole algorithm.
- For the computation of the the estimates for the worst case scenarios we can use: $\overline{Q}\le\lvert V\rvert$ and $\lvert E\rvert\in\mathcal{O}(\lvert V\rvert^2)$.

Justification of the Dijkstra's Efficiency Table
Notation for the estimated average run times

In the next computations we set $n=\lvert V\rvert$ and we denote by Q_i the number of elements in the queue for the repetition i of the `while` loop, with $i=1,2,\dots,n$.

We denote by a_i the total number of times that the function `add_with_priority` is run at the repetition i of the `while` loop.

We denote by d_i the total number of times that the function `decrease_priority` is run at the repetition i of the `while` loop.

As already said, the function `extract_min` is run once at every repetition i of the `while` loop.

Remark: $\sum_{i=1}^n a_i=n$ and $\sum_{i=1}^n d_i=\lvert E\rvert-n$.

Justification of the Dijkstra's Efficiency Table
Estimated average run times for a plain linked list not sorted

Average run time of `extract_min`:

The function `extract_min` has to perform a sequential search through the list to find the element with minimum cost. Then the expected run time T_{EM} *at the repetition i of the `while` loop* is $\mathcal{O}\left(\frac{Q_i}{2}\right)\approx K_i^{\text{EM}}\frac{Q_i}{2}$.

Thus, the total run time average is:

$$\frac{1}{n}\sum_{i=1}^n K_i^{\text{EM}}\frac{Q_i}{2}\le\frac{\max\left\{K_1^{\text{EM}},K_2^{\text{EM}},\dots,K_n^{\text{EM}}\right\}}{2}\frac{1}{n}\sum_{i=1}^n Q_i=$$
$$\max\left\{K_1^{\text{EM}},K_2^{\text{EM}},\dots,K_n^{\text{EM}}\right\}\frac{\overline{Q}}{2}=\mathcal{O}\left(\frac{\overline{Q}}{2}\right)=\mathcal{O}(\overline{Q}).$$

The average run time of `decrease_priority` is $\mathcal{O}(1)$:

Here it is assumed that *the cost data is not included in the list*. More specifically the list must include only the indices of the nodes that belong to it, while the information about the node's costs should be stored either in the node's vector or in a separate auxiliary vector.

If done like this, the operation `decrease_priority` only has to update the costs in the node's vector or the auxiliary vector and the list does not need to be modified.

Justification of the Dijkstra's Efficiency Table
Estimated average run times for a linked list sorted by priority **in increasing order**

Average run time of `add_with_priority`:

The function `add_with_priority` has to perform a sequential search through the list to find the place where to insert the new element according to its cost. The expected run time T_{AwP} *at the repetition i of the `while` loop* is $a_i\mathcal{O}\left(\frac{Q_i}{2}\right)\approx a_iK_i^{\text{AwP}}\frac{Q_i}{2}$. The total run time average is:

$$\frac{1}{n}\sum_{i=1}^n a_iK_i^{\text{AwP}}\frac{Q_i}{2}\le\frac{\max\left\{a_1K_1^{\text{AwP}},a_2K_2^{\text{AwP}},\dots,a_nK_n^{\text{AwP}}\right\}}{2}\frac{1}{n}\sum_{i=1}^n Q_i=$$
$$\max\left\{a_1K_1^{\text{AwP}},a_2K_2^{\text{AwP}},\dots,a_nK_n^{\text{AwP}}\right\}\frac{\overline{Q}}{2}=\mathcal{O}\left(\frac{\overline{Q}}{2}\right)=\mathcal{O}(\overline{Q}).$$

Average run time of `decrease_priority`:

`decrease_priority` has to perform a sequential search through the list to find the place where to re-insert the node according to its new cost, and remove it from the initial place. Since the list is sorted in cost-ascending order, the initial place of the node whose cost is being modified it bigger than the new place where it has to be inserted. So, we only need to do a *single* sequential search.

The expected run time T_{DP} *at the repetition i of the `while` loop* is $d_i\mathcal{O}\left(\frac{Q_i}{2}\right)\approx d_iK_i^{\text{DP}}\frac{Q_i}{2}$.

The total run time average is:

$$\frac{1}{\lvert E\rvert-n}\sum_{i=1}^n d_iK_i^{\text{DP}}\frac{Q_i}{2}\le\frac{\max\left\{d_1K_1^{\text{DP}},d_2K_2^{\text{DP}},\dots,d_nK_n^{\text{DP}}\right\}}{2}\frac{n}{\lvert E\rvert-n}\frac{1}{n}\sum_{i=1}^n Q_i=$$
$$\frac{n\max\left\{d_1K_1^{\text{DP}},d_2K_2^{\text{DP}},\dots,d_nK_n^{\text{DP}}\right\}}{\lvert E\rvert-n}\frac{\overline{Q}}{2}=\mathcal{O}\left(\frac{\overline{Q}}{2}\right)=\mathcal{O}(\overline{Q}).$$

Justification of the Dijkstra's Efficiency Table
Estimated average run times for a binary heap sorted by priority

Average run time of `extract_min`:

Here the complexity comes neither from the extraction nor from the deletion of the node. It rather comes from the *heapify operation* to rebuild the binary heap after removing the first node.

The run time T_{EM} *at the repetition i of the* `while` *loop* is $\mathcal{O}\left(\log_2(Q_i)\right) = K_i^{\text{EM\_BH}} \log_2(Q_i)$.
Since the $\log_2$ function is concave, by *Jensen's Inequality*, the total run time average is:

$$\frac{1}{n}\sum_{i=1}^{n} K_i^{\text{EM\_BH}} \log_2(Q_i) \le \max\left\{K_1^{\text{EM\_BH}}, K_2^{\text{EM\_BH}}, \dots, K_n^{\text{EM\_BH}}\right\} \frac{1}{n}\sum_{i=1}^{n} \log_2(Q_i) \overset{\text{Jensen Ineq.}}{\le}$$
$$\max\left\{K_1^{\text{EM\_BH}}, K_2^{\text{EM\_BH}}, \dots, K_n^{\text{EM\_BH}}\right\} \log_2\left(\overline{Q}\right) = \mathcal{O}\left(\log_2\left(\overline{Q}\right)\right).$$

Average run time of `add_with_priority`:

The run time T_{AwP} *at the repetition i of the* `while` *loop* is $\mathcal{O}\left(\log_2(Q_i)\right) = a_i K_i^{\text{AwP\_BH}} \log_2(Q_i)$.
Again by *Jensen's Inequality* the total run time average is:

$$\frac{1}{n}\sum_{i=1}^{n} a_i K_i^{\text{AwP\_BH}} \log_2(Q_i) \le \max\left\{a_1 K_1^{\text{AwP\_BH}}, a_2 K_2^{\text{AwP\_BH}}, \dots, a_n K_n^{\text{AwP\_BH}}\right\} \frac{1}{n}\sum_{i=1}^{n} \log_2(Q_i) \overset{\text{Jensen Ineq.}}{\le}$$
$$\max\left\{a_1 K_1^{\text{AwP\_BH}}, a_2 K_2^{\text{AwP\_BH}}, \dots, a_n K_n^{\text{AwP\_BH}}\right\} \log_2(\overline{Q}) = \mathcal{O}\left(\log_2\left(\overline{Q}\right)\right).$$

Justification of the Dijkstra's Efficiency Table
Estimated average run times for a binary heap sorted by priority

Average run time of `decrease_priority`:

The run time T_{DP} *at the repetition i of the* `while` *loop* is $d_i\mathcal{O}\left(\log_2(Q_i)\right) \approx d_i K_i^{\text{DP\_BH}} \log_2(Q_i)$.
The total run time average is:

$$\frac{1}{|E|-n}\sum_{i=1}^{n} d_i K_i^{\text{DP\_BH}} \log_2(Q_i) \le$$
$$\frac{n\max\left\{d_1 K_1^{\text{DP\_BH}}, d_2 K_2^{\text{DP\_BH}}, \dots, d_n K_n^{\text{DP\_BH}}\right\}}{|E|-n} \frac{1}{n}\sum_{i=1}^{n} \log_2(Q_i) \overset{\text{Jensen Ineq.}}{\le}$$
$$\frac{n\max\left\{d_1 K_1^{\text{DP\_BH}}, d_2 K_2^{\text{DP\_BH}}, \dots, d_n K_n^{\text{DP\_BH}}\right\}}{|E|-n} \log_2\left(\overline{Q}\right) = \mathcal{O}\left(\log_2\left(\overline{Q}\right)\right).$$

An implementation of the *Dijkstra's Algorithm* in **C**
Initializations and main

```c
#include <stdio.h>
#include <stdlib.h>
#include <values.h> // For MAXFLOAT = \infty and UINT_MAX = \infty

typedef struct{ unsigned vertexto; float weight; } weighted_arrow;
typedef struct{ char name;
                unsigned arrows_num; weighted_arrow arrow[5];
                float dist; unsigned parent;
} graph_vertex;

#define ORDER 5

int main() { register unsigned i;
    graph_vertex Graph[ORDER] = {
        { 'A', 2, {{1, 10}, {2, 3}}, MAXFLOAT, UINT_MAX }, // vertex 0
        { 'B', 2, {{2, 1},  {3, 2}}, MAXFLOAT, UINT_MAX }, // vertex 1
        { 'C', 3, {{1, 4},  {3, 8}, {4, 2}}, MAXFLOAT, UINT_MAX }, // vertex 2
        { 'D', 1, {{4,7}}, MAXFLOAT, UINT_MAX }, // vertex 3
        { 'E', 1, {{3,9}}, MAXFLOAT, UINT_MAX }, // vertex 4
    };

   Dijkstra(Graph, 0U);

   fprintf(stdout, "Vertex |  Cost | Parent\n-------|-------|-------\n");
   fprintf(stdout, " %c (%u) |%6.1f |\n", Graph[0].name, 0U, Graph[0].dist);
   for(i=1; i < ORDER; i++)
       fprintf(stdout, " %c (%u) |%6.1f | %c (%u)\n",
               Graph[i].name, i, Graph[i].dist, Graph[Graph[i].parent].name, Graph[i].parent);
}
```

Output: the minimal spanning tree

```
 Vertex |  Cost | Parent
 -------|-------|-------
  A (0) |   0.0 |
  B (1) |   7.0 | C (2)
  C (2) |   3.0 | A (0)
  D (3) |   9.0 | B (1)
  E (4) |   5.0 | C (2)
```


An implementation of the *Dijkstra's Algorithm* in **C**
Priority queue declarations and the Dijkstra function code

```c
typedef struct QueueElementstructure {
   unsigned v;
   struct QueueElementstructure *seg;
} QueueElement;
typedef QueueElement * PriorityQueue;

int IsEmpty( PriorityQueue Pq ){ return ( Pq == NULL ); }

void Dijkstra(graph_vertex * Graph, unsigned source){
    PriorityQueue Pq = NULL;
    char expanded[ORDER] = {[0 ... ORDER-1] = 0};

    Graph[source].dist = 0.0;
    add_with_priority(source, &Pq, Graph);

    while(!IsEmpty(Pq)){ register unsigned i;
        unsigned node = extract_min(&Pq);
        expanded[node] = 1;
        for(i=0; i < Graph[node].arrows_num; i++){
            unsigned adj = Graph[node].arrow[i].vertexto;
            if(expanded[adj]) continue;
            float dist_aux = Graph[node].dist + Graph[node].arrow[i].weight;
            if(Graph[adj].dist > dist_aux){
                char Is_adj_In_Pq = Graph[adj].dist < MAXFLOAT;
                Graph[adj].dist = dist_aux;
                Graph[adj].parent = node;
                if(Is_adj_In_Pq) decrease_priority(adj, &Pq, Graph);
                else add_with_priority(adj, &Pq, Graph);
}   }   }   }
```

An implementation of a priority queue as a linked list in C

The priority queue functions code: `extract_min`

Notation and the definition of a *Priority Queue*

Given pointers `QueueElement *a, *b`, we will write $a < b$ to denote that the queue element `*b` is a descendant (in the queue) of the element `*a` (that is, `b = a->seg->seg···->seg`).

In these notes a *Priority Queue* verifies

$a < b \iff$ `Graph[a->v].dist` $\leq$ `Graph[b->v].dist`

for every pair of valid pointers `QueueElement *a, *b`.

Then the function `extract_min` has to deal (without any search) with the first element of the queue.

The `extract_min` function code

```
unsigned extract_min(PriorityQueue *Pq){
  PriorityQueue first = *Pq;
  unsigned v = first->v;

  *Pq = (*Pq)->seg;
  free(first);
  return v;
}
```


An implementation of a priority queue as a linked list in C

The priority queue functions code: `add_with_priority`

The `add_with_priority` function code

```
void add_with_priority( unsigned v,
                        PriorityQueue *Pq, graph_vertex * Graph )
{   QueueElement *aux = (QueueElement *) malloc(sizeof(QueueElement));
    if(aux == NULL) exit(66);

    aux->v = v;

    float costv = Graph[v].dist;
    if( *Pq == NULL || !(costv > Graph[(*Pq)->v].dist) ) {
        aux->seg = *Pq; *Pq = aux;
        return;
    }

    register QueueElement * q;
    for(q = *Pq; q->seg && Graph[q->seg->v].dist < costv; q = q->seg );
    aux->seg = q->seg; q->seg = aux;
    return;
}
```

Exit with error code the Devil's number

Standard creation of a new queue element

`*Pq = NULL` is equivalent to queue empty: The queue is initialized with `v`. Then `aux->seg = *Pq = NULL` correctly marks that `aus` is the end of the queue.

The check `!(costv > Graph[(*Pq)->v].dist)` occurs when `*Pq != NULL`. Then the queue `*Pq` is not empty, and the new element `aux` containing `v` must be the first element of the queue.

At this point `*Pq != NULL` and `Graph[(*Pq)->v].dist < costv`. This `for` loop computes *the largest* `QueueElement *q` with `q` $\geq$ `*Pq` such that `Graph[q->v].dist < costv` (the *insertion point* of `aux`). The loop ends either with:

- `q->seg = NULL`: then, `*q` is the last element of the queue (equivalently `costv` is greater than all costs in the queue) and `aux` must be placed at the end of the queue (i.e. after `*q` — `q->seg = aux`), or
- `Graph[q->v].dist < costv <= Graph[q->seg->v].dist`: then, `*q` is *not* the last element of the queue (`q->seg != NULL`), and `aux` must be placed between `*q` and `*(q->seg)`.

An implementation of a priority queue as a linked list in C

The function `requeue_with_priority` code: a simple but inefficient approach to `decrease_priority`

Notation and Strategy

- `pv` denotes the pointer `QueueElement * pv` to the element of the queue which contains `v`. In particular, `pv->v = v`.
- `prepv` denotes the pointer `QueueElement * prepv` to the element of the queue which is *before* `*pv`. That is, `prepv->seg = pv`, and `prepv->seg->v = pv->v = v`.

Strategy: Remove `*pv` from the queue and re-enqueue `v` with the new decreased cost.

The `requeue_with_priority` function code

```
void requeue_with_priority( unsigned v,
                            PriorityQueue *Pq, graph_vertex * Graph ){
    if((*Pq)->v == v) return;

    register QueueElement * prepv;
    for(prepv = *Pq; prepv->seg->v != v; prepv = prepv->seg);
    QueueElement * pv = prepv->seg;
    prepv->seg = pv->seg;
    free(pv);

    add_with_priority(v, Pq, Graph);
}
```

`for` loop to sequentially compute `prepv`: It is not necessary to check `prepv->seg != NULL` since `prepv` is initialized as `*Pq`, `(*Pq)->seg <= pv` and `v = prepv->seg->v` *is* in the queue. Then, in the loop, `prepv->seg` will run through the queue element containing `v`.

Nothing to do: The first element of the queue is `v`. Since the new `Graph[v].dist` is smaller, it is not necessary to re-order the queue. In the rest of the function, `(*Pq)->v != v` $\iff$ `*Pq < pv` $\iff$ `*Pq <= prepv < prepv->seg = pv`.

An implementation of a priority queue as a linked list in C

The function `decrease_priority` code (with detailed comments in the next pages)

The `decrease_priority` function code

```
void decrease_priority( unsigned v,
                        PriorityQueue *Pq, graph_vertex * Graph ){
    if((*Pq)->v == v) return;

    float costv = Graph[v].dist;
    if(!(costv > Graph[(*Pq)->v].dist)){ register QueueElement *prepv;
        for(prepv = *Pq; prepv->seg->v != v; prepv = prepv->seg);
        QueueElement * swap = *Pq;
        *Pq=prepv->seg; prepv->seg=prepv->seg->seg; (*Pq)->seg=swap;
        return;
    }

    register QueueElement *q, *prepv;
    for(q = *Pq; Graph[q->seg->v].dist < costv; q = q->seg );
    if(q->seg->v == v) return;

    for(prepv = q->seg; prepv->seg->v != v; prepv = prepv->seg);
    QueueElement *pv = prepv->seg;
    prepv->seg = pv->seg; pv->seg = q->seg; q->seg = pv;
    return;
}
```

Nothing to do: The first element of the queue is `v`. Since the new `Graph[v].dist` is smaller, it is not necessary to re-order the queue. In the rest of the function, `(*Pq)->v != v` $\iff$ `*Pq < pv` $\iff$ `*Pq <= prepv < prepv->seg = pv`.

An implementation of a priority queue as a linked list in C

Comments to the `decrease_priority` function code

The special case `costv <= Graph[(*Pq)->v].dist`

The *new cost* `costv` of `*pv` is smaller than or equal to the cost of `*Pq`.

Strategy: `*pv` has to be moved to the beginning of the queue

Consequently, we need to compute `prepv` and

connect `*prepv` with `*(pv->seg) = *(prepv->seg->seg)`

Remark: This justifies why we need to compute `prepv` instead of the (apparently more natural) computation of `pv`.

Computation of `prepv` (`pv = prepv->seg`)

As we have seen, here we have `(*Pq)->v!=v`, which is equivalent to

`*Pq <= prepv < prepv->seg = pv`.

We can compute `prepv` with this `for` loop — see the "callout" note at page 63.

Case: `!(costv > Graph[(*Pq)->v].dist)`

`!(costv > Graph[(*Pq)->v].dist)` ⟺ `costv <= Graph[(*Pq)->v].dist`

```
float costv = Graph[v].dist;
if(!(costv > Graph[(*Pq)->v].dist)){ register QueueElement *prepv;
    for(prepv = *Pq; prepv->seg->v != v; prepv = prepv->seg);
    QueueElement * swap = *Pq;
    *Pq=prepv->seg; prepv->seg=prepv->seg->seg; (*Pq)->seg=swap;
    return;
}
```


An implementation of a priority queue as a linked list in C

Comments to the `decrease_priority` function code

The general case `costv > Graph[(*Pq)->v].dist`

The *new cost* `costv` of `*pv` is larger than the cost of `*Pq`.

Notation

In the general case, when the loop below stops, we have `q >= *Pq` and

`Graph[a->v].dist < costv <= Graph[q->seg->v].dist`

for every `QueueElement *a` such that `*Pq <= a <= q` (see the corresponding "callout" note at page 62).

Strategy

Compute `q` and `pv` (in fact, `prepv`), and re-allocate `*pv = *(prepv->seg)` between `*q` and `*(q->seg)`.

Computation of `q` and exit if `q->seg = pv`

```
register QueueElement *q, *prepv;
for(q = *Pq; Graph[q->seg->v].dist < costv; q = q->seg );
if(q->seg->v == v) return;
```

Exercise: `if(q->seg->v == v)` there is nothing to do

When `q->seg->v = v` ⟺ `q->seg = pv` it is not difficult to see that the queue is still sorted after decreasing `Graph[v].dist`.

From now on `q->seg->v != v` ⟺ `q->seg != pv` which implies `q->seg < pv`.

An implementation of a priority queue as a linked list in C

Final comments to the `decrease_priority` function code

Strategy recalled

Compute `q` (already done) and `prepv`, and re-allocate `*pv = *(prepv->seg)` between `*q` and `*(q->seg)`.

Computation of `prepv`

As we have seen, here we have `q->seg < pv`, which is equivalent to

`q->seg <= prepv < prepv->seg = pv`.

Then the `for` loop below sequentially computes `prepv`.

It is not necessary to check the condition `prepv->seg != NULL` (see the vertical "callout" note at page 63) because `prepv` is initialized as `q->seg <= prepv` and `v = prepv->seg->v` *is* in the queue. Then, in the loop, `prepv->seg` must run through the queue element containing `v`.

Computation of `prepv` and re-allocation of `*pv = *(prepv->seg)`

```
for(prepv = q->seg; prepv->seg->v != v; prepv = prepv->seg);
QueueElement *pv = prepv->seg;
prepv->seg = pv->seg; pv->seg = q->seg; q->seg = pv;
return;
```

Re-allocation of `*pv = *(prepv->seg)` between `*q` and `*(q->seg)`

We also need to connect `*prepv` with `*(pv->seg) = *(prepv->seg->seg)`.

Optional Exercise

Implement the Dijkstra's Algorithm in C with a binary heap priority queue. That is, implement a binary heap priority queue in C and use it in Dijkstra's Algorithm.

A* Algorithm

Contents

Introduction to A* Algorithm[6]

A* is a graph traversal and path search algorithm for solving the *routing problem*. It is *complete*, *optimal* and *computationally efficient*. It is the best solution in many cases (despite of the major practical drawback that it stores all generated nodes in memory).

A* is an *informed search algorithm*, or a *best-first search*. It maintains a tree of paths originating at the start node and extending one edge at a time until its termination criterion is satisfied. A* can be seen as an extension of Dijkstra's Algorithm. It achieves better performance by using heuristics to guide its search.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes $f(v) = g(v) + h(v)$ where v is the next node on the path, $g(v)$ is the cost of the path from the start node to v, and $h(v)$ *is a heuristic function that estimates the cost of the cheapest path from v to the goal node*.

A* terminates when the path it chooses to extend, is a path from start to goal or if there are no eligible paths to be extended.

[6] Inspired in https://en.wikipedia.org/wiki/A*_search_algorithm

Introduction to A* Algorithm

The heuristic function[7] is problem-specific. When it is *admissible*, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

Typical implementations of A* use a *priority queue* to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the *Open Queue* (or *Open Set*). At each step of the algorithm, the node with the lowest f value is removed from the queue, the f and g values of its neighbours are updated accordingly, and these neighbours are added to the queue. The algorithm continues until a removed node (thus the node with lowest f value out of all open nodes) is a goal node. The f value of that goal is then also the cost of the shortest path, since h at the goal is zero in an admissible heuristic.

To find the actual sequence of steps that constitute a shortest path, as in Dijkstra's Algorithm, one has to keep track of the predecessor of each node on the computed shortest path. At A* termination, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

[7] As an example, when searching for the shortest route on a map, $h(v)$ might represent the straight-line distance from v to the goal, since that is physically the smallest possible distance between any two points.

A* Algorithm pseudocode

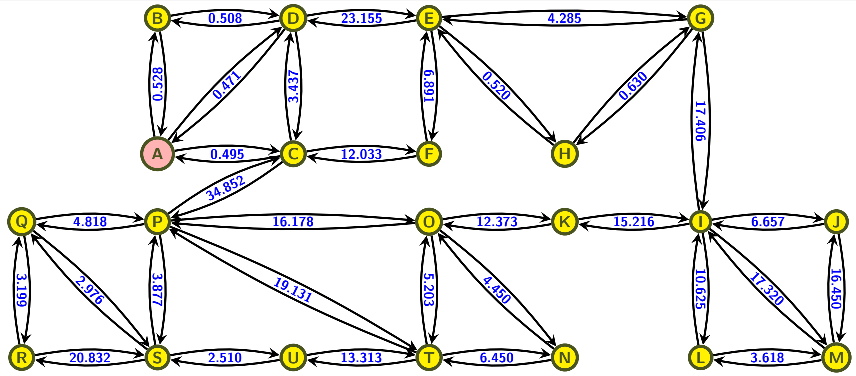
```
procedure ASTAR(graph G, start, goal, h)
    Open ← EmptyPriorityQueue                    }
    parent[G.order] ← uninitialized              } ▷ General initialization
    g[G.order] ← initialized to ∞                } ▷ Important to detect the non-visited (and non-enqueued) nodes
    g[start] ← 0                                 }
    parent[start] ← ∞                            } ▷ Open Set initialization: start has distance
    Open.add_with_priority(start, g, h)          }   0 to itself, has no parent and is enqueued
    while not Open.IsEmpty do                                            ▷ The main loop
        current ← Open.extract_min(g, h)                                 ▷
        if (current is goal) then return g, parent                       ▷ We have found the solution
        for each adj ∈ current.neighbours do
            adj_new_try_gScore ← g[current] + ω(current, adj)  ▷ New cost from start to adj through current
            if adj_new_try_gScore < g[adj] then
                parent[adj] ← current
                g[adj] ← adj_new_try_gScore
                if not Open.BelongsTo(adj) then Open.add_with_priority(adj, g, h)
                else Open.requeue_with_priority(adj, g, h)
                end if
            end if
        end for
    end while
    return failure                                    ▷ goal is not accessible from start
end procedure
```

Relaxation step ◁

extract_min removes a node current with minimal cost $f(\text{current}) = g(\text{current}) + h(\text{current})$ from the **Open Queue**. Subsequently, the node current will be expanded.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)

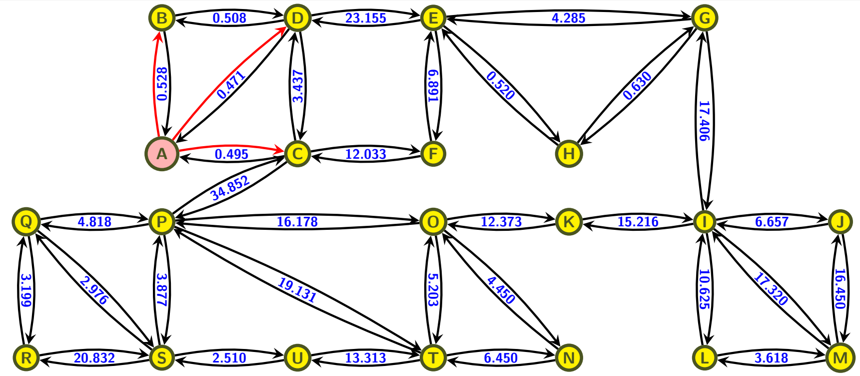


Open Queue	A
g	0
f	0.471
parent	nil

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



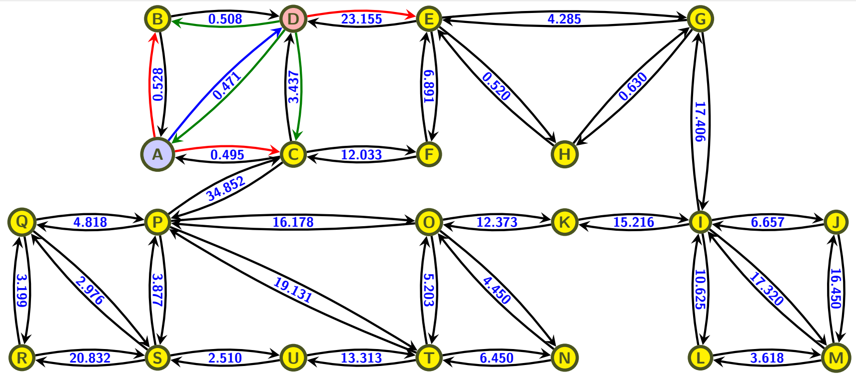
Open Queue	D	C	B
g	0.471	0.495	0.528
f	0.942	0.99	1.036
parent	A	A	A

expanded	A
g	0
f	0.471
parent	nil

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



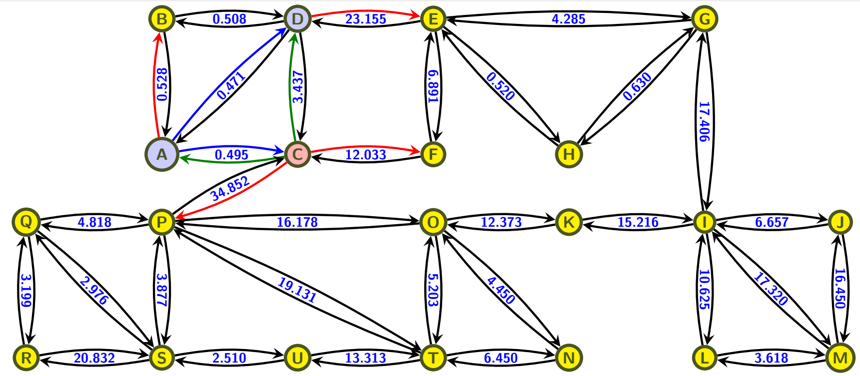
Open Queue	C	B	E
g	0.495	0.528	23.626
f	0.99	1.036	24.146
parent	A	A	D

expanded	A	D
g	0	0.471
f	0.471	0.942
parent	nil	A

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



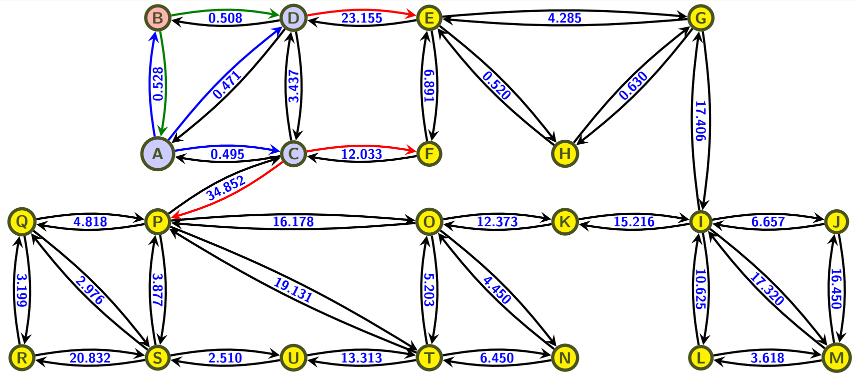
Open Queue	B	F	E	P
g	0.528	12.528	23.626	35.347
f	1.036	19.419	24.146	39.224
parent	A	C	D	C

expanded	A	D	C
g	0	0.471	0.495
f	0.471	0.942	0.99
parent	nil	A	A

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A⋆ Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



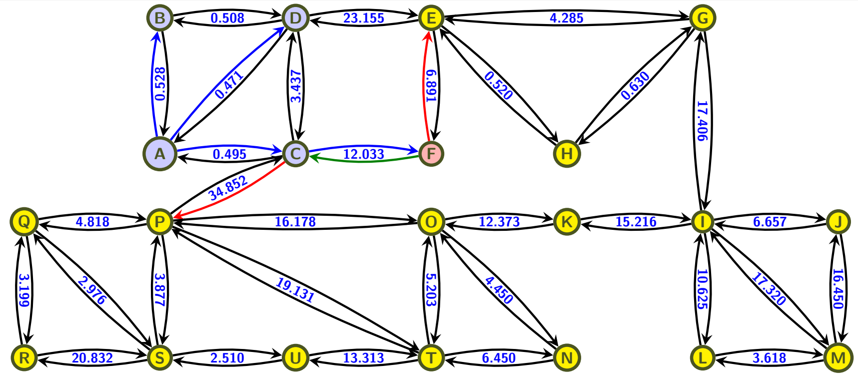
Open Queue	F	E	P
g	12.528	23.626	35.347
f	19.419	24.146	39.224
parent	C	D	C

expanded	A	D	C	B
g	0	0.471	0.495	0.528
f	0.471	0.942	0.99	1.036
parent	nil	A	A	A

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A⋆ Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



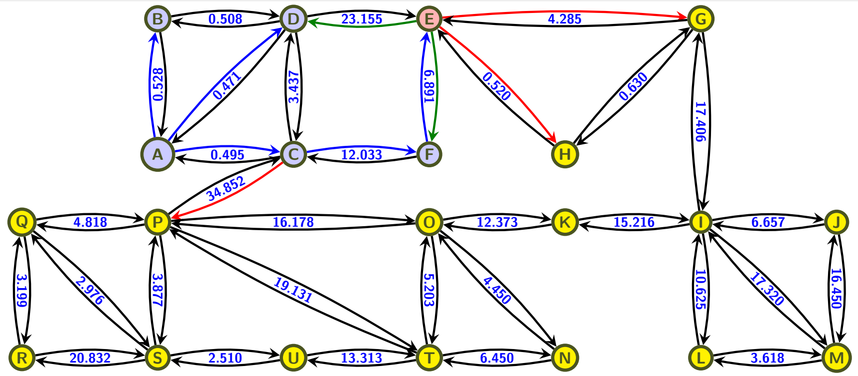
Open Queue	E	P
g	19.419	35.347
f	19.939	39.224
parent	F	C

expanded	A	D	C	B	F
g	0	0.471	0.495	0.528	12.528
f	0.471	0.942	0.99	1.036	19.419
parent	nil	A	A	A	C

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A⋆ Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



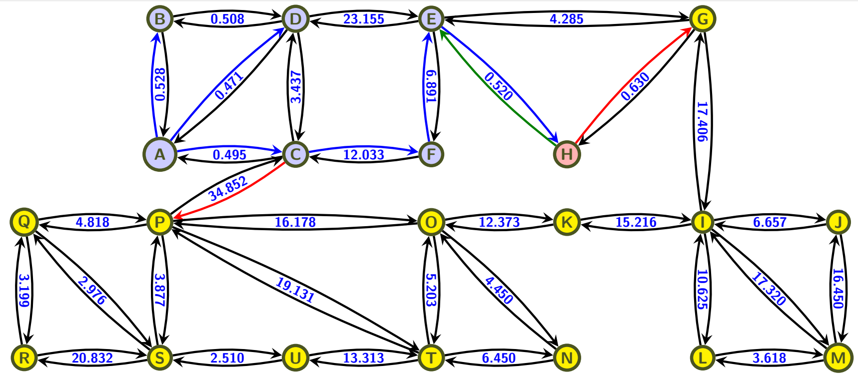
Open Queue	H	G	P
g	19.939	23.704	35.347
f	20.459	24.334	39.224
parent	E	E	C

expanded	A	D	C	B	F	E
g	0	0.471	0.495	0.528	12.528	19.419
f	0.471	0.942	0.99	1.036	19.419	19.939
parent	nil	A	A	A	C	F

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A⋆ Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



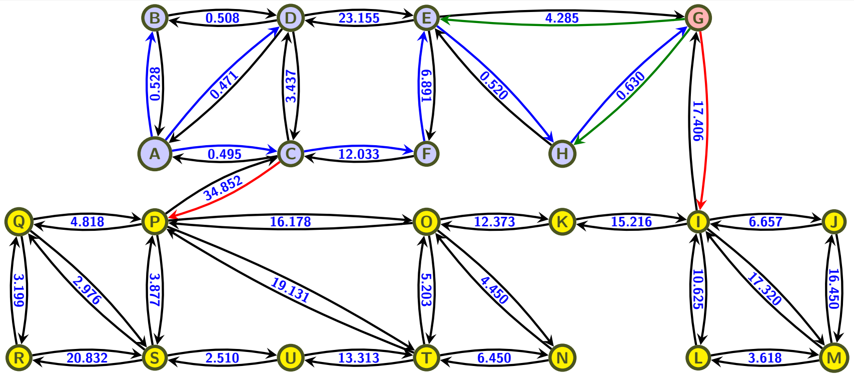
Open Queue	G	P
g	20.569	35.347
f	21.199	39.224
parent	H	C

expanded	A	D	C	B	F	E	H
g	0	0.471	0.495	0.528	12.528	19.419	19.939
f	0.471	0.942	0.99	1.036	19.419	19.939	20.459
parent	nil	A	A	A	C	F	E

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



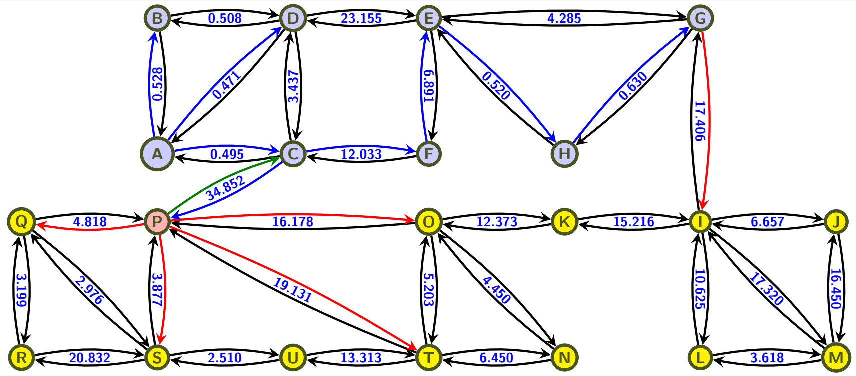
Open Queue	P	I
g	35.347	37.975
f	39.224	44.632
parent	C	G

expanded	A	D	C	B	F	E	H	G
g	0	0.471	0.495	0.528	12.528	19.419	19.939	20.569
f	0.471	0.942	0.99	1.036	19.419	19.939	20.459	21.199
parent	nil	A	A	A	C	F	E	H

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



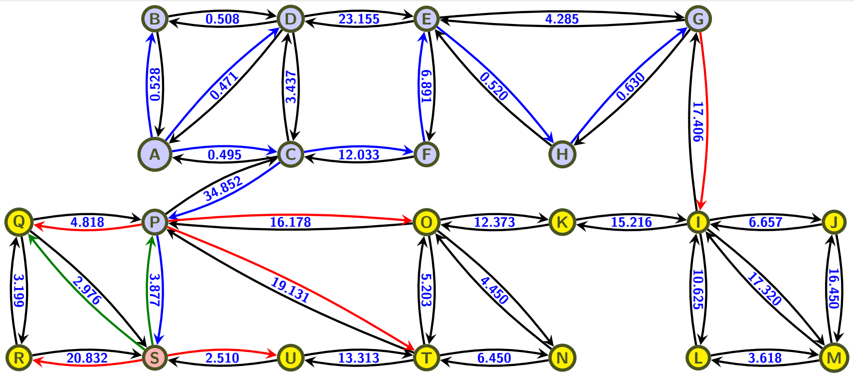
Open Queue	S	Q	I	O	T
g	39.224	40.165	37.975	51.525	54.478
f	41.734	43.141	44.632	55.975	59.681
parent	P	P	G	P	P

expanded	A	D	C	B	F	E	H	G	P
g	0	0.471	0.495	0.528	12.528	19.419	19.939	20.569	35.347
f	0.471	0.942	0.99	1.036	19.419	19.939	20.459	21.199	39.224
parent	nil	A	A	A	C	F	E	H	C

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



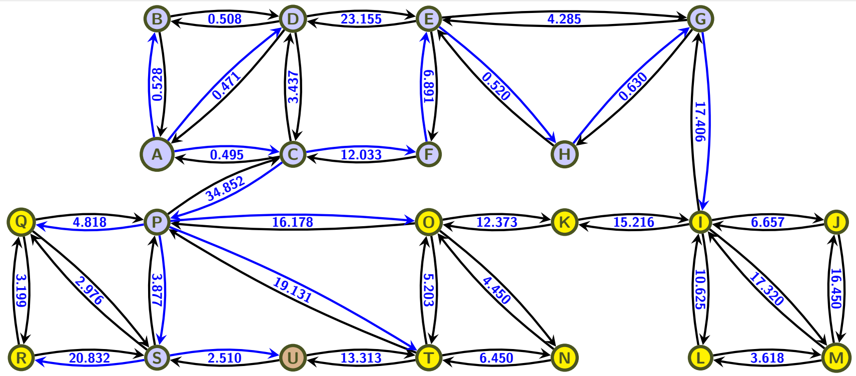
Open Queue	U	Q	I	O	T	R
g	41.734	40.165	37.975	51.525	54.478	60.056
f	41.734	43.141	44.632	55.975	59.681	63.255
parent	S	P	G	P	P	S

expanded	A	D	C	B	F	E	H	G	P	S
g	0	0.471	0.495	0.528	12.528	19.419	19.939	20.569	35.347	39.224
f	0.471	0.942	0.99	1.036	19.419	19.939	20.459	21.199	39.224	41.734
parent	nil	A	A	A	C	F	E	H	C	P

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

An example of the A* Algorithm

Finding the optimal path from source node *A* to node goal *U* (heuristics to be discussed)



Open Queue	Q	I	O	T	R
g	40.165	37.975	51.525	54.478	60.056
f	43.141	44.632	55.975	59.681	63.255
parent	P	G	P	P	S

expanded	A	D	C	B	F	E	H	G	P	S	U
g	0	0.471	0.495	0.528	12.528	19.419	19.939	20.569	35.347	39.224	41.734
f	0.471	0.942	0.99	1.036	19.419	19.939	20.459	21.199	39.224	41.734	41.734
parent	nil	A	A	A	C	F	E	H	C	P	S

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again). As we will see this is due to the fact that the heuristic function is monotone.

On the heuristic function

The whole algorithm and, in particular, its efficiency depend on the heuristic function

As we will see, the best heuristic function estimates (but never overestimates) the actual cost to get to the goal from any node.

Simple general example for weighted graphs
(does not use latitude and longitude)

Let $G = (V, E, \omega)$ be a weighted graph and let γ denote the goal node. For every vertex $v \in V$ we can set

$$h(v) := \begin{cases} \min\{\omega(v,u) : (v,u) \in E\} & \text{if } v \neq \gamma, \\ 0 & \text{if } v = \gamma. \end{cases}$$

We will show that this heuristic function is *admissible* and *monotone*. However, it is very bad since it is far from correctly estimating $\sigma(v, \gamma)$.

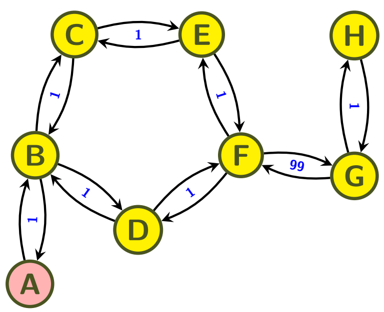
In the example contained in the next slides we will see the enormous importance of choosing the best possible heuristic function.

An example of the A* Algorithm: Comparing two heuristics

Finding the optimal path from source node ***A*** to node goal ***G***

Heuristic function

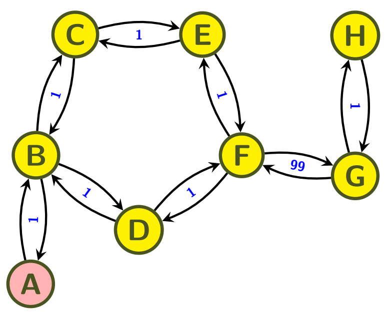
	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0



Open Queue	A
g	0
f	0
parent	nil

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



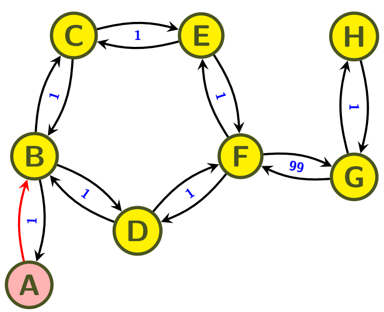
Open Queue	A
g	0
f	0
parent	nil

An example of the A* Algorithm: Comparing two heuristics

Finding the optimal path from source node ***A*** to node goal ***G***

Heuristic function

	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0

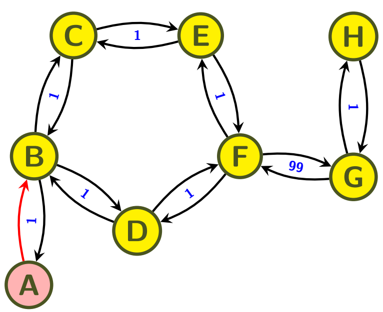


expanded	A
g	0
f	0
parent	nil

Open Queue	B
g	1
f	1
parent	A

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



expanded	A
g	0
f	0
parent	nil

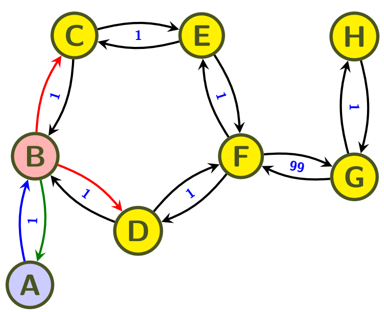
Open Queue	B
g	1
f	102
parent	A

An example of the A* Algorithm: Comparing two heuristics

Finding the optimal path from source node ***A*** to node goal ***G***

Heuristic function

	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0

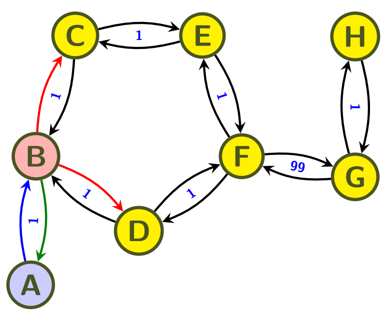


expanded	A	B
g	0	1
f	0	1
parent	nil	A

Open Queue	C	D
g	2	2
f	2	102
parent	B	B

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1

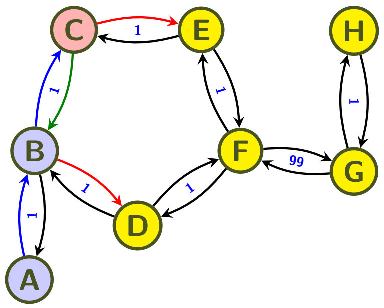


expanded	A	B
g	0	1
f	0	102
parent	nil	A

Open Queue	D	C
g	2	2
f	102	103
parent	B	B

Heuristic function

	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0

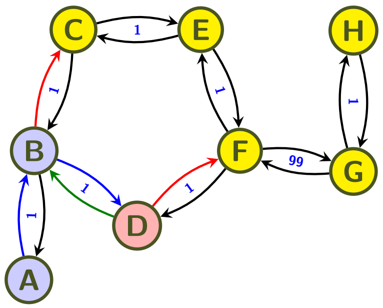


expanded	A	B	C
g	0	1	2
f	0	1	2
parent	nil	A	B

Open Queue	**E**	**D**
g	3	2
f	3	102
parent	C	B

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



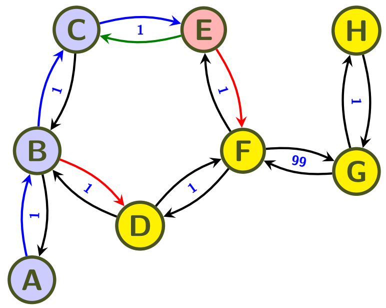
expanded	A	B	D
g	0	1	2
f	0	102	102
parent	nil	A	B

Open Queue	**F**	**C**
g	3	2
f	102	103
parent	D	B

An example of the A* Algorithm: Comparing two heuristics
Finding the optimal path from source node ***A*** to node goal ***G***

Heuristic function

	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0

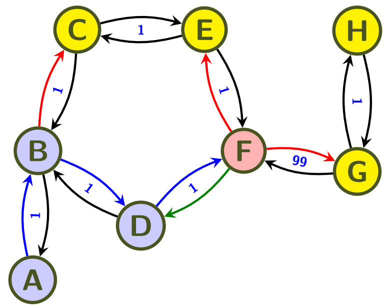


expanded	A	B	C	E
g	0	1	2	3
f	0	1	2	3
parent	nil	A	B	C

Open Queue	**F**	**D**
g	4	2
f	4	102
parent	E	B

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



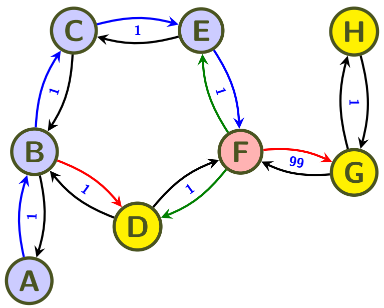
expanded	A	B	D	F
g	0	1	2	3
f	0	102	102	102
parent	nil	A	B	D

Open Queue	**G**	**C**	**E**
g	102	2	4
f	102	103	104
parent	F	B	F

An example of the A* Algorithm: Comparing two heuristics
Finding the optimal path from source node ***A*** to node goal ***G***

Heuristic function

	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0

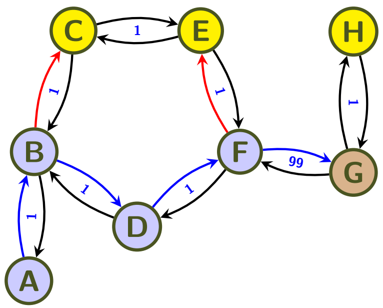


expanded	A	B	C	E	F
g	0	1	2	3	4
f	0	1	2	3	4
parent	nil	A	B	C	E

Open Queue	**D**	**G**
g	2	103
f	102	103
parent	B	F

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



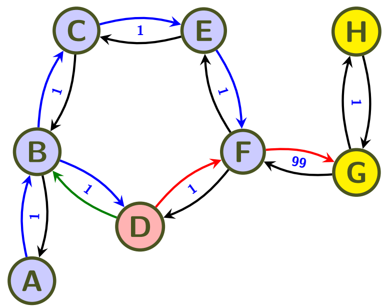
expanded	A	B	D	F	G
g	0	1	2	3	102
f	0	102	102	102	102
parent	nil	A	B	D	F

Open Queue	**C**	**E**
g	2	4
f	103	104
parent	B	F

An example of the A* Algorithm: Comparing two heuristics
Finding the optimal path from source node ***A*** to node goal ***G***

Heuristic function

	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0



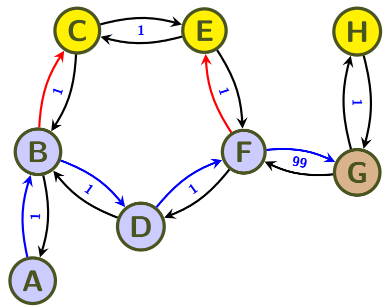
F 4 4 E

expanded	A	B	C	E	D
g	0	1	2	3	2
f	0	1	2	3	102
parent	nil	A	B	C	B

Open Queue	**F**	**G**
g	3	103
f	3	103
parent	D	F

Heuristic function

	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



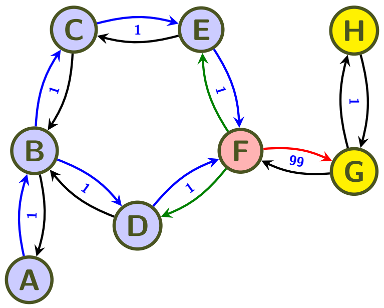
expanded	A	B	D	F	G
g	0	1	2	3	102
f	0	102	102	102	102
parent	nil	A	B	D	F

Open Queue	**C**	**E**
g	2	4
f	103	104
parent	B	F

An example of the A* Algorithm: Comparing two heuristics

Finding the optimal path from source node ***A*** to node goal ***G***

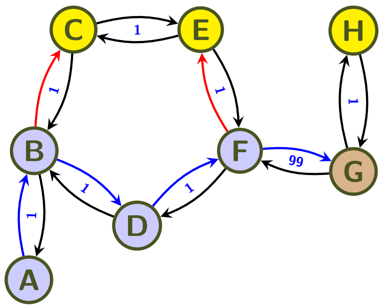
Heuristic function								
	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0



expanded	A	B	C	E	D	F
g	0	1	2	3	2	3
f	0	1	2	3	102	3
parent	nil	A	B	C	B	D

Open Queue	G
g	102
f	102
parent	F

Heuristic function								
	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



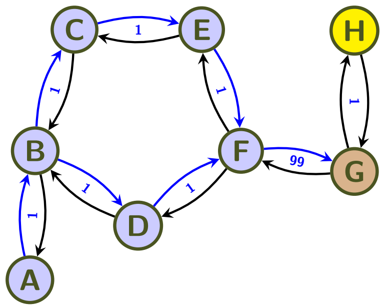
expanded	A	B	D	F	G
g	0	1	2	3	102
f	0	102	102	102	102
parent	nil	A	B	D	F

Open Queue	C	E
g	2	4
f	103	104
parent	B	F

An example of the A* Algorithm: Comparing two heuristics

Finding the optimal path from source node ***A*** to node goal ***G***

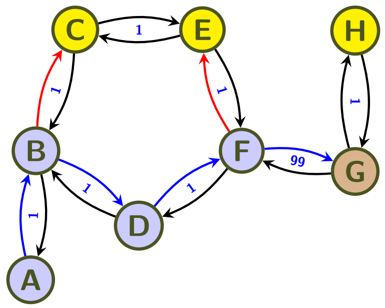
Heuristic function								
	A	B	C	D	E	F	G	H
h	0	0	0	100	0	0	0	0



expanded	A	B	C	E	D	F	G
g	0	1	2	3	2	3	102
f	0	1	2	3	102	3	102
parent	nil	A	B	C	B	D	F

Open Queue
g
f
parent

Heuristic function								
	A	B	C	D	E	F	G	H
h	0	101	101	100	100	99	0	1



expanded	A	B	D	F	G
g	0	1	2	3	102
f	0	102	102	102	102
parent	nil	A	B	D	F

Open Queue	C	E
g	2	4
f	103	104
parent	B	F

The A* Basic Step

Let $v \in V$ be a vertex of G for which there exists a node $u \in V \setminus \{\gamma\}$ such that:

1. $(u, v) \in E$ is an edge of the graph,
2. u is removed from the **Open Queue** by the function `extract_min`, and
3. $g(v) > g(u) + \omega(u, v)$.

Then, the **if** clause of the relaxation step holds true for `adj = v`, and

- $g(v)$ is set to the lower value $g(u) + \omega(u, v) < \infty$,
- u is set to be `parent[v]`, and
- v is set to belong to the **Open Queue** with the new $g(v)$ value.

Moreover, this is the only way that v can enter to the **Open Queue** and `parent[v]` can be modified.

Definition

The operation described in the above remark will be called *relaxing the node v after expanding the node u*.

The A* Basic Operation

Based on the construction (or exploration) of paths

Definition: Path constructed by the A* Algorithm at a relaxation

Let $\alpha := (\xi{=}v_0 \to v_1 \to \cdots \to v_{n-1} \to v_n)$ be a path in a graph G.

We say that *α has been constructed by the A* Algorithm at a relaxation of v_n* (or, equivalently, *at a v_n opening*) whenever v_n is relaxed after the expansion of v_{n-1} (and thus added to the **Open Queue**), and `parent[`v_j`]` $= v_{j-1}$ for $j = n-1, n-2, \ldots, 2, 1$ at the relaxation of v_n (in particular, $g(v_i) < \infty$ for $i = 0, 1, 2, \ldots, n-1$).

After relaxing v_n, we also have

$$g(v_n) < \infty \quad \text{and} \quad v_{n-1} = \texttt{parent[}v_n\texttt{]}.$$

Remark

Every relaxation of a vertex v generates a new path from ξ to v that is strictly cheaper than all other paths from ξ to v constructed so far for the algorithm.

The A* Basic Operation
Acyclic paths

Definition: Acyclic Path

A path is called *acyclic* if it does not contain a loop. Equivalently, a path is *acyclic* if and only if every node appearing in the path is not repeated (i.e., it appears exactly once in the whole path).

An A* Basic Lemma — How A* works

All paths constructed by the A* Algorithm are acyclic.

Proof of A* Basic Lemma

Assume by way of contradiction that A* has just constructed a cyclic path

$$\xi = v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_m \longrightarrow v_{m+1} \longrightarrow \cdots \longrightarrow v_{m+k} \longrightarrow v_m,$$

with $v_0, v_1, \ldots, v_m, v_{m+1}, \ldots, v_{m+k}$ pairwise different. Then, prior (i.e., just before) the relaxation of v_m after the expansion of v_{m+k} we have the following situation:

1. The nodes
$$v_0 = \texttt{parent}[v_1], v_1 = \texttt{parent}[v_2], \ldots, v_m = \texttt{parent}[v_{m+1}],$$
$$v_{m+1} = \texttt{parent}[v_{m+2}], \ldots, v_{m+k-1} = \texttt{parent}[v_{m+k}],$$
have been previously relaxed,
2. (v_{m+k}, v_m) is an edge of the graph,
3. v_{m+k} is removed from the **Open Queue** by the function `extract_min`,

and finally (since v_m has been already expanded, and 1 and 3 from Slide 76 hold),

$$\sum_{i=0}^{m-1} \omega(v_i, v_{i+1}) = g(v_m) > g(v_{m+k}) + \omega(v_{m+k}, v_m) =$$
$$\sum_{i=0}^{m+k-1} \omega(v_i, v_{i+1}) + \omega(v_{m+k}, v_m) > \sum_{i=0}^{m-1} \omega(v_i, v_{i+1});$$

a contradiction.

Algorithmic properties of A*: Termination

A* always terminates on finite graphs.

Proof

Let C be the maximal subgraph of G that contains ξ and is connected. Clearly, C is finite because G is finite and every path of G starting at ξ is contained in C.

On the other hand, the number of acyclic paths starting at ξ (and thus contained in C) is finite.

By the A* Basic Lemma, the A* Algorithm constructs a subset of the acyclic paths starting at ξ, and traverse the subgraph of C formed by the union of these (finitely many) acyclic paths in finite time. Recall that reopened nodes correspond to new acyclic paths from ξ to the node being reopened, since A* only reopens a node when constructing a path strictly cheaper than (and thus different from) the ones constructed previously to the node being reopened.

So, either

- $\gamma \in C$ and A* will stop after finding the goal node (when extracting γ from the **Open Queue** with the function `extract_min`) and ending the construction of an acyclic path from the source ξ to γ; or
- $\gamma \notin C$ and, in this case, A* will traverse the whole graph C (searching for the inexistent γ) by sequentially constructing all (finitely many) acyclic paths contained in C starting at ξ. After finishing the construction of all these paths the algorithm stops with an error message.

Algorithmic properties of A*: Completeness

Completeness

An algorithm is said to be *complete* if it terminates with a (non necessarily optimal) solution when one exists.

Completeness Theorem

A* is complete.

Proof

By using the notation from the proof the termination property we get that $\gamma \in C$ by assumption. So, by the proof the termination property, in this case A* constructs an acyclic path from the source ξ to γ, thus giving a solution path.

Algorithmic properties of A*: Admissibility

Admissibility

An algorithm is *admissible* if it is guaranteed to return an optimal solution whenever a solution exists.

Definition

An heuristic function h is said to be *admissible* if for every vertex $v \in V$,

$$\boxed{h(v) \leq \sigma(v, \gamma)}$$

where γ is the goal node.

Admissibility Theorem

If h is admissible, then A* is admissible.

Example (the heuristic function from Page 74 is admissible)

If $v = \gamma$ we have: $h(v) = h(\gamma) = 0 = \sigma(\gamma, \gamma)$.
If $v \neq \gamma$, let α be an optimal path from v to the node goal γ and let $u \in V$ be such that $(v, u) \in E$ and α starts with (v, u). We have

$$h(v) = \min\{\omega(v, x) : (v, x) \in E\} \leq \omega(v, u) \leq \omega(\alpha) = \sigma(v, \gamma).$$

Algorithmic properties of A*: Dominance and Optimality

Dominance

An algorithm $A_1^\star$ is said to *dominate* $A_2^\star$ if every node expanded by $A_1^\star$ is also expanded by $A_2^\star$. Similarly, $A_1^\star$ *strictly dominates* $A_2^\star$ if $A_1^\star$ dominates $A_2^\star$ and $A_2^\star$ does not dominate $A_1^\star$. We will also use the phrase "more efficient than" interchangeably with dominates.

Optimality

An algorithm is said to be *optimal* over a class of algorithms if it dominates all members of that class.

Definition

An heuristic function h_2 is *more informed than* h_1 if both are admissible and $h_2(v) > h_1(v)$ for every non-goal vertex $v \in V$. Similarly, an A* algorithm using h_2 is said to be *more informed than* that using h_1.

Theorem

If $A_2^\star$ is more informed than $A_1^\star$, then $A_2^\star$ dominates $A_1^\star$.

Algorithmic properties of A*: Monotone Heuristics

By the triangle inequality we have $\sigma(u, \gamma) \leq \sigma(u, v) + \sigma(v, \gamma)$ for every $u, v \in V$, where $\gamma \in V$ denotes the goal node. Since, by admissibility $h(\cdot)$ is an estimate of $\sigma(\cdot, \gamma)$, it is now reasonable to expect that if the process of estimating $h(\cdot)$ is consistent, it should inherit the above inequality and satisfy $h(u) \leq \sigma(u, v) + h(v)$ for every $u, v \in V$.

Definition (Consistency and Monotonicity)

An heuristic function h is said to be *consistent* if

$$h(u) \leq \sigma(u, v) + h(v)$$

is satisfied for all pairs of nodes $u, v \in V$.

An heuristic function h is said to be *monotone* if it satisfies

$$h(u) \leq \omega(u, v) + h(v)$$

for every $u, v \in V$ such that $(u, v) \in E$ is an edge of the graph.

Algorithmic properties of A*: Monotone Heuristics

Monotonicity may seem, at first glance, to be less restrictive than consistency, because it only relates the heuristic of a node to the heuristics of its immediate successors. However, a simple proof by induction on the depth of the descendants of u shows the following

Theorem

An heuristic function is monotone if and only if it is consistent.

It is also simple to relate consistency to admissibility.

Theorem

Every consistent heuristic is admissible.

Example (the heuristic function from Page 74 is monotone and admissible)

Let $u, v \in V$ be such that $(u, v) \in E$ is an edge of the graph. Then,

$$h(u) = \min\{\omega(u, x) : (u, x) \in E\} \leq \omega(u, v) \leq \omega(u, v) + h(v)$$

because h is non-negative.

Algorithmic properties of A*: Monotone Heuristics

Theorem (All discovered paths are optimal)

An A* algorithm guided by a monotone heuristic finds optimal paths to all expanded vertices $v \in V$. That is, as in Dijkstra's Algorithm,

$$\boxed{g(v) = \sigma(\xi, v)}$$

for every expanded vertex $v \in V$.

Theorem (Monotonicity of the sequence of f–values)

Monotonicity implies that the sequence $\{f(v_i)\}_{i=1}^{\ell}$ of f–values of the sequence of vertices $\{v_i\}_{i=1}^{\ell}$ expanded by A* is non-decreasing[8].

Theorem (Easy expansion conditions)

If h is a monotone heuristic, then the necessary condition for expanding a vertex $v \in V$ is given by

$$\boxed{\sigma(\xi, v) + h(v) \leq \sigma(\xi, \gamma),}$$

and the sufficient condition is given by the strict inequality

$$\boxed{\sigma(\xi, v) + h(v) < \sigma(\xi, \gamma).}$$

[8] And this gives a way to check when an heuristic function is *not* monotone.

Implementation of the *A* Algorithm* in **C**

Declarations and auxiliary functions

To save memory we only store the `f`-values separately (and not the `h`-values). The value of `h = f-g` will then have to be computed from `f` and `g`.

Graph declarations and auxiliary functions

```
typedef char bool; enum {false, true};
typedef struct{ unsigned vertexto; float weight; } weighted_arrow;
typedef struct{ char name; unsigned arrows_num; weighted_arrow arrow[5]; } graph_vertex;
typedef struct { float g; unsigned parent; } AStarPath;

bool AStar(graph_vertex *, AStarPath *, unsigned, unsigned, unsigned);

void ExitError(const char *miss, int errcode) {
     fprintf (stderr, "\nERROR: %s.\nStopping...\n\n", miss); exit(errcode);
}
```

Priority Queue and `A`* declarations and auxiliary functions

```
typedef struct QueueElementstruct { unsigned v; struct QueueElementstruct *seg; } QueueElement;
typedef QueueElement * PriorityQueue;
typedef struct { float f; bool IsOpen; } AStarControlData;

float heuristic(graph_vertex *Graph, unsigned vertex, unsigned goal){ register unsigned short i;
if(vertex == goal) return 0.0;
float minw = Graph[vertex].arrow[0].weight;
for(i=1; i < Graph[vertex].arrows_num ; i++){
if( Graph[vertex].arrow[i].weight < minw ) minw = Graph[vertex].arrow[i].weight;
}
return minw; }
```

Question

Is the heuristic function good? If not, how to improve it?

To implement the function `Open.BelongsTo()` efficiently in time

Instead of sequentially explore the whole queue to determine whether a given node v belongs to the list, it is much simpler to check if `ASCD[v].IsOpen` is `true`. The drawback is that this `bool` variable costs one byte more per node, and its maintenance must be done manually (`add_with_priority` automatically sets this variable for easiness).

Implementation of the *A* Algorithm* in **C**

main program and results

```
#define GraphOrder 21

int main() {
graph_vertex Graph[GraphOrder] = {
{'A', 3, { {1, 0.528}, {2, 0.495}, {3, 0.471} }},
{'B', 2, { {0, 0.528}, {3, 0.508} }},
... ... ... ... ... ...
{'U', 2, { {18, 2.510}, {19, 13.313} }} };
AStarPath PathData[GraphOrder];
unsigned node_start = 0U, node_goal = 20U;

bool r = AStar(Graph, PathData, GraphOrder, node_start, node_goal);
if(r == -1) ExitError("in allocating memory for the OPEN list in AStar", 21);
else if(!r) ExitError("no solution found in AStar", 7);

register unsigned v=node_goal, pv=PathData[v].parent, ppv; PathData[node_goal].parent=UINT_MAX;
while(v != node_start) { ppv=PathData[pv].parent; PathData[pv].parent=v; v=pv; pv=ppv; }

printf("Optimal path found:\nNode name | Distance\n----------|---------\n");
printf("  %c (%3.3u) |  Source\n", Graph[node_start].name, node_start);
for(v=PathData[node_start].parent ; v !=UINT_MAX ; v=PathData[v].parent)
printf("  %c (%3.3u) | %7.3f\n", Graph[v].name, v, PathData[v].g);
return 0; }
```

Output: Shortest path

```
Node name | Distance
----------|---------
  A (000) |  Source
  C (002) |   0.495
  P (015) |  35.347
  S (018) |  39.224
  U (020) |  41.734
```

Starting at `node_goal`, reverse the parents path so that `successor` becomes `parent` and, conversely, `parent` becomes `successor`. Then, we can write the optimal path forward; starting at `node_start` until we arrive at `node_goal`.

Implementation of the *A* Algorithm* in **C**

main program and results

```
#define GraphOrder 21

int main() {
```

```
graph_vertex Graph[GraphOrder] = {
{'A', 3, { {1, 0.528}, {2, 0.495}, {3, 0.471} }},
{'B', 2, { {0, 0.528}, {3, 0.508} }},
{'C', 4, { {0, 0.495}, {3, 3.437}, {5, 12.033}, {15, 34.852} }},
{'D', 4, { {0, 0.471}, {1, 0.508}, {2, 3.437}, {4, 23.155} }},
{'E', 4, { {3, 23.155}, {5, 6.891}, {6, 4.285}, {7, 0.520} }},
{'F', 2, { {2, 12.033}, {4, 6.8910} }}, {'G', 3, { {4, 4.285}, {7, 0.630}, {8, 17.406} }},
{'H', 2, { {4, 0.520}, {6, 0.630} }},
{'I', 5, { {6, 17.406}, {9, 6.657}, {10, 15.216}, {11, 10.625}, {12, 17.320} }},
{'J', 2, { {8, 6.657}, {12, 16.450} }}, {'K', 2, { {8, 15.216}, {14, 12.373} }},
{'L', 2, { {8, 10.625}, {12, 3.618} }}, {'M', 3, { {8, 17.320}, {9, 16.450}, {11, 3.618} }},
{'N', 2, { {14, 4.450}, {19, 6.450} }},
{'O', 4, { {10, 12.373}, {13, 4.450}, {15, 16.178}, {19, 5.203} }},
{'P', 5, { {2, 34.852}, {14, 16.178}, {16, 4.818}, {18, 3.877}, {19, 19.131} }},
{'Q', 3, { {15, 4.818}, {17, 3.199}, {18, 2.976} }}, {'R', 2, { {16, 3.199}, {18, 20.832} }},
{'S', 4, { {15, 3.877}, {16, 2.976}, {17, 20.832}, {20, 2.510} }},
{'T', 4, { {13, 6.450}, {14, 5.203}, {15, 19.131}, {20, 13.313} }},
{'U', 2, { {18, 2.510}, {19, 13.313} }} };
```

```
if(r == -1) ExitError("in allocating memory for the OPEN list in AStar", 21);
else if(!r) ExitError("no solution found in AStar", 7);

register unsigned v=node_goal, pv=PathData[v].parent, ppv; PathData[node_goal].parent=UINT_MAX;
while(v != node_start) { ppv=PathData[pv].parent; PathData[pv].parent=v; v=pv; pv=ppv; }

printf("Optimal path found:\nNode name | Distance\n----------|---------\n");
printf("  %c (%3.3u) |  Source\n", Graph[node_start].name, node_start);
for(v=PathData[node_start].parent ; v !=UINT_MAX ; v=PathData[v].parent)
printf("  %c (%3.3u) | %7.3f\n", Graph[v].name, v, PathData[v].g);
return 0; }
```

Output: Shortest path

```
Node name | Distance
----------|---------
  (000) |  Source
  (002) |   0.495
  (015) |  35.347
  (018) |  39.224
  (020) |  41.734
```

Implementation of the *A* Algorithm* in C

main program and results

```
#define GraphOrder 21

int main() {
graph_vertex Graph[GraphOrder] = {
{'A', 3, { {1, 0.528}, {2, 0.495}, {3, 0.471} }},
{'B', 2, { {0, 0.528}, {3, 0.508} }},
... ... ... ... ... ... ...
{'U', 2, { {18, 2.510}, {19, 13.313} }} };
AStarPath PathData[GraphOrder];
unsigned node_start = 0U, node_goal = 20U;

bool r = AStar(Graph, PathData, GraphOrder, node_start, node_goal);
if(r == -1) ExitError("in allocating memory for the OPEN list in AStar", 21);
else if(!r) ExitError("no solution found in AStar", 7);

register unsigned v=node_goal, pv=PathData[v].parent, ppv; PathData[node_goal].parent=UINT_MAX;
while(v != node_start) { ppv=PathData[pv].parent; PathData[pv].parent=v; v=pv; pv=ppv; }

printf("Optimal path found:\nNode name | Distance\n----------|---------\n");
printf("  %c (%3.3u) |  Source\n", Graph[node_start].name, node_start);
for(v=PathData[node_start].parent ; v !=UINT_MAX ; v=PathData[v].parent)
printf("  %c (%3.3u) | %7.3f\n", Graph[v].name, v, PathData[v].g);
return 0; }
```

Output: Shortest path

```
Node name | Distance
----------|---------
  A (000) |  Source
  C (002) |   0.495
  P (015) |  35.347
  S (018) |  39.224
  U (020) |  41.734
```

Starting at `node_goal`, reverse the parents path so that `successor` becomes `parent` and, conversely, `parent` becomes `successor`. Then, we can write the optimal path forward; starting at `node_start` until we arrive at `node_goal`.

Implementation of the *A* Algorithm* in C

The *AStar* function code

```
bool AStar(graph_vertex *Graph, AStarPath *PathData, unsigned GrOrder,
unsigned node_start, unsigned node_goal){ register unsigned i;
PriorityQueue Open = NULL;
AStarControlData *Q;

if((Q = (AStarControlData *) malloc(GrOrder*sizeof(AStarControlData))) == NULL)
ExitError("when allocating memory for the AStar Control Data vector", 73);
for(i=0; i < GrOrder; i++) { PathData[i].g = MAXFLOAT; Q[i].IsOpen = false; }

PathData[node_start].g = 0.0; PathData[node_start].parent = ULONG_MAX;
Q[node_start].f = heuristic(Graph, node_start, node_goal);
if(!add_with_priority(node_start, &Open, Q)) return -1;

while(!IsEmpty(Open)){ unsigned node_curr;
if((node_curr = extract_min(&Open)) == node_goal) { free(Q); return true; }
for(i=0; i < Graph[node_curr].arrows_num ; i++){
unsigned node_succ = Graph[node_curr].arrow[i].vertexto;
float g_curr_node_succ = PathData[node_curr].g + Graph[node_curr].arrow[i].weight;
if( g_curr_node_succ < PathData[node_succ].g ){
PathData[node_succ].parent = node_curr;
Q[node_succ].f = g_curr_node_succ + ((PathData[node_succ].g == MAXFLOAT) ?
heuristic(Graph, node_succ, node_goal) : (Q[node_succ].f-PathData[node_succ].g) );
PathData[node_succ].g = g_curr_node_succ;
if(!Q[node_succ].IsOpen) { if(!add_with_priority(node_succ, &Open, Q)) return -1; }
else requeue_with_priority(node_succ, &Open, Q);
}
}
Q[node_curr].IsOpen = false;
} /* Main loop while */
return false;
}
```

To check easily whether a given node v belongs to the queue: It does so if and only if `Q[v].IsOpen` is `true`.

For `node_start` we have `f = h` because `g` = 0.0.

Implementation of the *A* Algorithm* in C

The *AStar* function code

```
bool AStar(graph_vertex *Graph, AStarPath *PathData, unsigned GrOrder,
unsigned node_start, unsigned node_goal){ register unsigned i;
PriorityQueue Open = NULL;
AStarControlData *Q;

if((Q = (AStarControlData *) malloc(GrOrder*sizeof(AStarControlData))) == NULL)
ExitError("when allocating memory for the AStar Control Data vector", 73);
for(i=0; i < GrOrder; i++) { PathData[i].g = MAXFLOAT; Q[i].IsOpen = false; }
```

To save computational effort we call the `heuristic` function to compute `h`:

`h(node_succ) = heuristic(Graph, node_succ, node_goal)`

only the first time that we visit a node (`PathData[node_succ].g == MAXFLOAT`). When a node `node_succ` has been already visited we recover the value of `h(node_succ) = f(node_succ) - g(node_succ)` (recall that we are not storing the `h`-values separately) from the formula

`f(node_succ) - g(node_succ) = Q[node_succ].f-PathData[node_succ].g.`

For efficiency, the computation of

`Q[node_succ].f = PathData[node_succ].g_new + h(node_succ)`

is implemented by means of an *arithmetic if*.

```
PathData[node_succ].parent = node_curr;
Q[node_succ].f = g_curr_node_succ + ((PathData[node_succ].g == MAXFLOAT) ?
heuristic(Graph, node_succ, node_goal) : (Q[node_succ].f-PathData[node_succ].g) );
PathData[node_succ].g = g_curr_node_succ;
if(!Q[node_succ].IsOpen) { if(!add_with_priority(node_succ, &Open, Q)) return -1; }
else requeue_with_priority(node_succ, &Open, Q);
}
}
Q[node_curr].IsOpen = false;
} /* Main loop while */
return false;
}
```

To check easily whether a given node v belongs to the queue: It does so if and only if `Q[v].IsOpen` is `true`.

Implementation of the *A* Algorithm* in C

The *AStar* function code

```
bool AStar(graph_vertex *Graph, AStarPath *PathData, unsigned GrOrder,
unsigned node_start, unsigned node_goal){ register unsigned i;
PriorityQueue Open = NULL;
AStarControlData *Q;

if((Q = (AStarControlData *) malloc(GrOrder*sizeof(AStarControlData))) == NULL)
ExitError("when allocating memory for the AStar Control Data vector", 73);
for(i=0; i < GrOrder; i++) { PathData[i].g = MAXFLOAT; Q[i].IsOpen = false; }

PathData[node_start].g = 0.0; PathData[node_start].parent = ULONG_MAX;
Q[node_start].f = heuristic(Graph, node_start, node_goal);
if(!add_with_priority(node_start, &Open, Q)) return -1;

while(!IsEmpty(Open)){ unsigned node_curr;
if((node_curr = extract_min(&Open)) == node_goal) { free(Q); return true; }
for(i=0; i < Graph[node_curr].arrows_num ; i++){
unsigned node_succ = Graph[node_curr].arrow[i].vertexto;
float g_curr_node_succ = PathData[node_curr].g + Graph[node_curr].arrow[i].weight;
if( g_curr_node_succ < PathData[node_succ].g ){
PathData[node_succ].parent = node_curr;
Q[node_succ].f = g_curr_node_succ + ((PathData[node_succ].g == MAXFLOAT) ?
heuristic(Graph, node_succ, node_goal) : (Q[node_succ].f-PathData[node_succ].g) );
PathData[node_succ].g = g_curr_node_succ;
if(!Q[node_succ].IsOpen) { if(!add_with_priority(node_succ, &Open, Q)) return -1; }
else requeue_with_priority(node_succ, &Open, Q);
}
}
Q[node_curr].IsOpen = false;
} /* Main loop while */
return false;
}
```

To check easily whether a given node v belongs to the queue: It does so if and only if `Q[v].IsOpen` is `true`.

For `node_start` we have `f = h` because `g` = 0.0.

Lluís Alsedà Shortest paths algorithms in weighted graphs ▸ General TOC 89/90

```
bool IsEmpty(PriorityQueue Pq){
return ((bool) (Pq == NULL));
}
unsigned extract_min(
PriorityQueue *Pq){
PriorityQueue first = *Pq;
unsigned v = first->v;

*Pq = (*Pq)->seg;
free(first);
return v; }
```

```
void requeue_with_priority(unsigned v, PriorityQueue *Pq,
AStarControlData * Q){
register QueueElement * prepv;
if((*Pq)->v == v) return;

for(prepv = *Pq; prepv->seg->v != v; prepv = prepv->seg);
QueueElement * pv = prepv->seg;
prepv->seg = pv->seg;
free(pv);

add_with_priority(v, Pq, Q); }
```

```
bool add_with_priority(unsigned v, PriorityQueue *Pq, AStarControlData * Q){
register QueueElement * q;
QueueElement *aux = (QueueElement *) malloc(sizeof(QueueElement));
if(aux == NULL) return false;

aux->v = v;
float costv = Q[v].f;
Q[v].IsOpen = true;

if( *Pq == NULL || !(costv > Q[(*Pq)->v].f) ) {
aux->seg = *Pq; *Pq = aux;
return true;
}

for(q = *Pq; q->seg && Q[q->seg->v].f < costv; q = q->seg ) ;
aux->seg = q->seg; q->seg = aux;
return true;
}
```