Shortest paths algorithms in weighted graphs

Lluís Alsedà

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

Versió 1.6 (19 de maig de 2021)

UAB
Universitat Autònoma
de Barcelona

DEPARTAMENT DE **MATEMÀTIQUES**

Contingut

Shortest paths in weighted graphs

Índex

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 (II)

- 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).
  - The *in-degree* of a vertex is the number of edges that arrive to the vertex.
  - 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 *extreme* vertices.
- A vertex with valence larger than 2 is called a *branching vertex*.

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 only once in the path.

Basic graph definitions: Concatenation

Given two paths

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

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

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

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

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:

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

Basic definitions on 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.

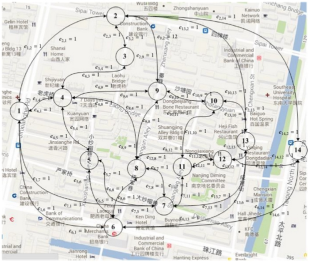
Also, we can encompass all the weights of a graph in a single *edge-weight function*:

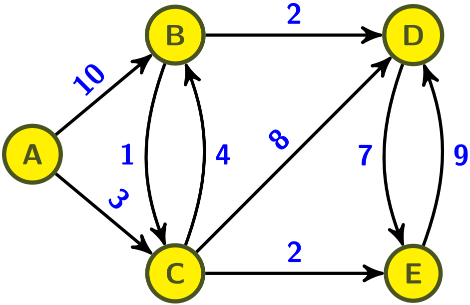
$$\begin{aligned} \omega : \quad E &\longrightarrow \mathbb{R} \\ a &\longmapsto \omega(a) \\ (x,y) &\longmapsto \omega((x,y)) \end{aligned}$$

---

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

Basic definitions on weighted graphs





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

Basic definitions on weighted graphs

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

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

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

Observation

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

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

Basic definitions on weighted graphs: shortest paths

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.

Basic definitions on weighted graphs: an example

The minimum weight may not be well defined when there is a negative weight cycle

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 $\mathbb{R}^+$: $\omega\colon E \longrightarrow \mathbb{R}^+$.

Basic definitions 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 $a \in E$.

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: Optimal substructure

Theorem (Optimality principle)

Any sub-path of a minimal path is minimal.

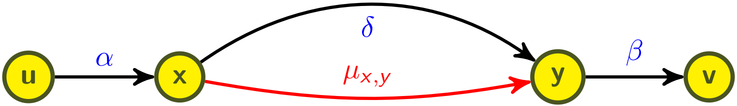
Proof

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 $\omega\big(\mu_{x,y}\big) < \omega(\delta)$ (in particular, $\mu_{x,y} \neq \delta$). So, $\alpha\mu_{x,y}\beta$ is another path from u to v such that

$$\omega\big(\alpha\mu_{x,y}\beta\big) = \omega(\alpha) + \omega\big(\mu_{x,y}\big) + \omega(\beta) < \omega(\alpha) + \omega(\delta) + \omega(\beta) = \omega(\alpha\delta\beta);$$

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

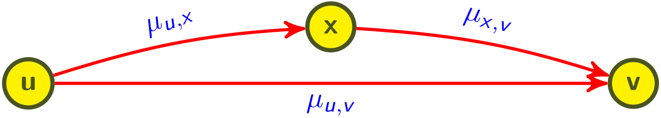
Observe that if either does not exist path from u to x or from x to v, then $\sigma(u, x) + \sigma(x, v) = \infty$, and the lemma holds. Otherwise, let $\mu_{u,x}$ be a minimal path from u to x (i.e. $\omega(\mu_{u,x}) = \sigma(u, x)$), and let $\mu_{x,v}$ be a minimal path from x to v (i.e. $\omega(\mu_{x,v})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

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)

$$\sigma(u, v) \leq \omega(\mu_{u,x}\mu_{x,v}) = \sigma(u, x) + \sigma(x, v).$$



Statement of the routing problem: 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.

Grafs: Definicions i Algorismes Bàsics, Pages 45 to 70,
`http://mat.uab.cat/~alseda/MatDoc/GrafsDefimovs.pdf`

Índex

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

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

```
procedure DIJKSTRA(graph G, source)
    Pq ← EmptyPriorityQueue
    expanded[G.order] ← initialized to false
    dist[G.order] ← initialized to $\infty$
    parent[G.order] ← uninitialized
        ▷ Declaration and initial assignment:
          expanded[v] = true $\Longleftrightarrow$ v is extract_min-taken-out from the list and expanded
          dist: distances vector from source to every node
          parent: previous vertices in an optimal path
    dist[source] ← 0
    parent[source] ← $\infty$
    Pq.add_with_priority(source, dist[source])
        ▷ Initialization: source has distance 0 to itself, has no parent and is enqueued
    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 $\in$ node.neighbours and not expanded[adj] do
            dist_aux ← dist[node] + $\omega$(node, adj)      ▷ New cost from source to adj through node
            if (dist[adj] > dist_aux) then
                if (dist[adj] = $\infty$) 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
                ▷ Relaxation step
        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.

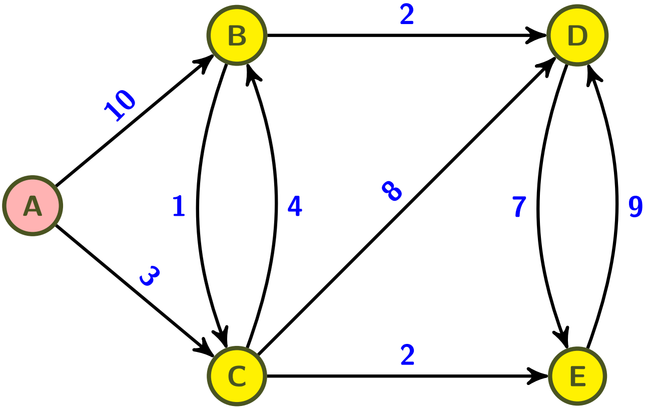
How the minimal spanning tree is specified?

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.

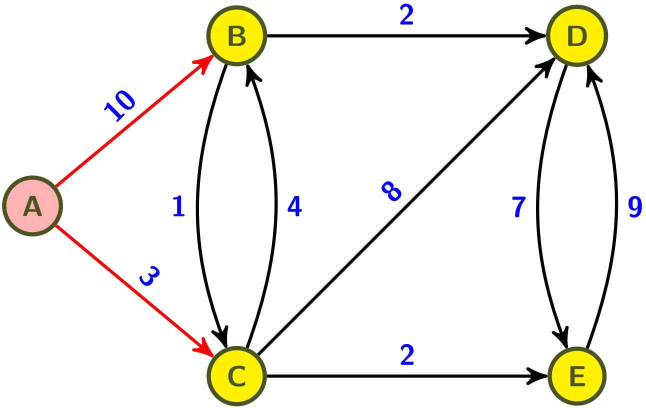
Thanks to the vector `parent` we can backwards construct the computed optimal paths to all vertices, thus building a minimal spanning tree.

An example of the Dijkstra's Algorithm



PriQueue	**A**
dist	0
parent	nil

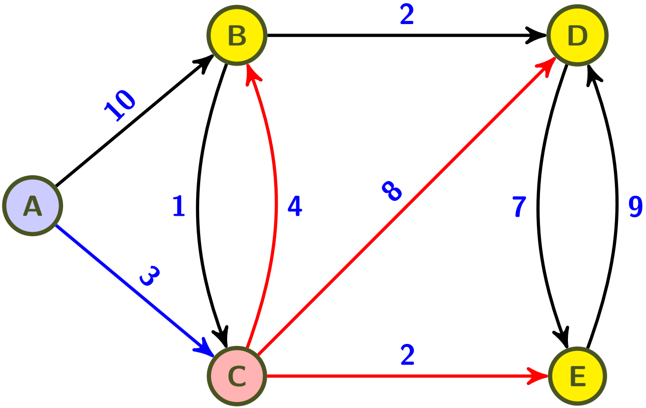
An example of the Dijkstra's Algorithm



expanded	A
dist	0
parent	nil

PriQueue	**C**	**B**
dist	3	10
parent	A	A

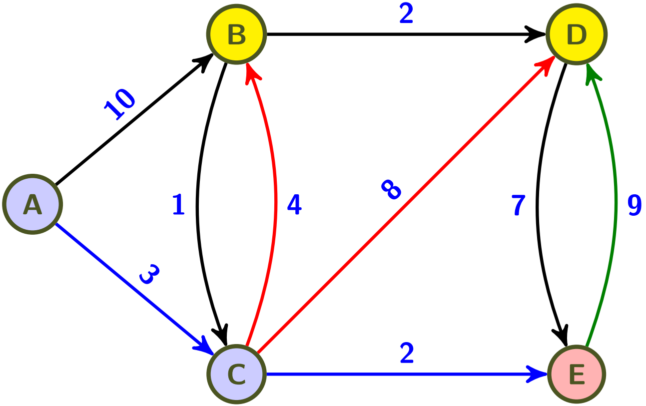
An example of the Dijkstra's Algorithm



expanded	A	C
dist	0	3
parent	nil	A

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

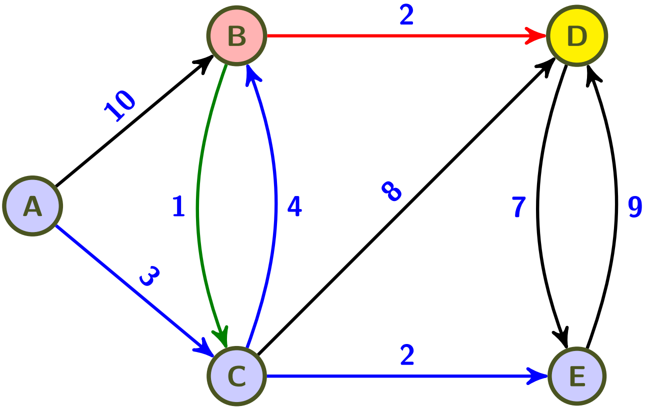
An example of the Dijkstra's Algorithm



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

PriQueue	**B**	**D**
dist	7	11
parent	C	C

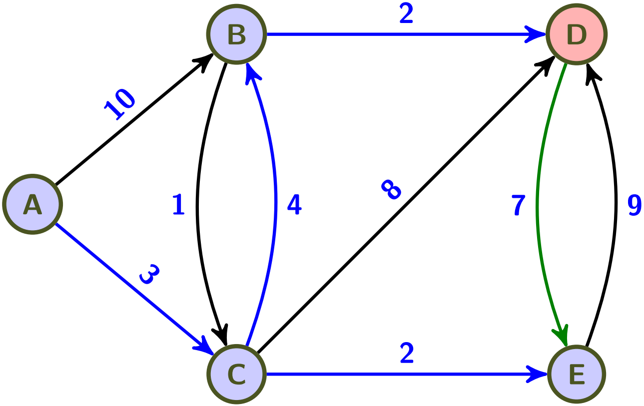
An example of the Dijkstra's Algorithm



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

PriQueue	D
dist	9
parent	B

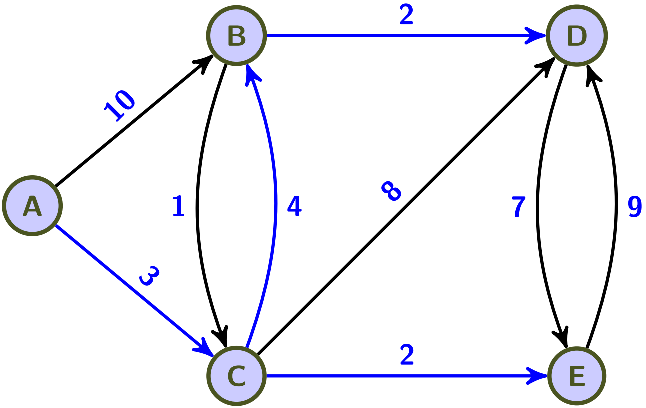
An example of the Dijkstra's Algorithm



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

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

Implementation of the *Dijkstra’s Algorithm* in C

Initializations and main

```
#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)
```

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);
}   }   }   }
```

Implementation of the *Dijkstra's Algorithm* 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

$$a < b \Longleftrightarrow \texttt{Graph[a->v].dist} \leq \texttt{Graph[b->v].dist}$$

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;
}
```

Implementation of the *Dijkstra's Algorithm* 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)`.

Implementation of the *Dijkstra's Algorithm* 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);
}
```

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

`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`.

Implementation of the *Dijkstra's Algorithm* 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` ⟺ `*Pq < pv` ⟺ `*Pq <= prepv < prepv->seg = pv`.

Implementation of the *Dijkstra's Algorithm* 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 25.

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;
    }
```

Implementation of the *Dijkstra's Algorithm* 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 24).

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` $\Longleftrightarrow$ `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` $\Longleftrightarrow$ `q->seg != pv` which implies `q->seg < pv`.

Implementation of the *Dijkstra's Algorithm* 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 25) 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)`.

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]` $\geq \sigma(\texttt{source}, v)$ holds at every iteration of the algorithm, for every vertex $v \in V$.

Proof of DA–Lemma 1

The initial assignment

```
dist[] ← **initialized to** $\infty$
dist[source] ← 0
```

guarantees that `dist[v]` $\geq \sigma(\texttt{source}, v)$ 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]` $< \sigma(\texttt{source}, v)$. 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,

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

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,

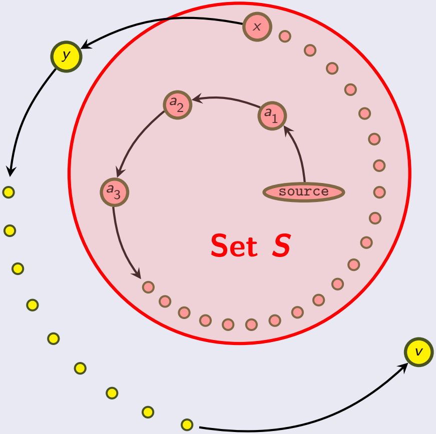
- `source` $\in S$,
- $v \notin S$ because the algorithm is just going to dequeue v, and
- since v is the first vertex that will be dequeued with `dist[v]` $> \sigma($`source`$, v)$, the equality `dist[u]` $= \sigma($`source`$, u)$ holds for every vertex $u \in S$ whenever it is dequeued, and it is maintained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

$$\texttt{dist[x]} = \sigma(\texttt{source}, x),$$

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

$$\texttt{dist[y]} = \sigma(\texttt{source}, y)$$

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 $\texttt{dist[y]} = \infty > \texttt{dist[v]}$ (recall that every node in the queue has finite `dist` value), or y is in the queue and $\texttt{dist[v]} \leq \texttt{dist[y]}$ 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 $\sigma(\texttt{source}, y) \leq \sigma(\texttt{source}, v)$.

Then, summarizing,

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

a contradiction.

Analysis of Dijkstra’s Algorithm efficiency

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] + $\omega$(node, adj)
        if (dist[adj] > dist_aux) then
            if (dist[adj] = $\infty$) 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 `decrease_priority`: T_{DP}
- `decrease_priority` is run $|E| - |V|$ times

- 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

Estimated average execution time

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

Analysis of Dijkstra's Algorithm efficiency (II)

Table of estimated average run times for several Dijkstra's Algorithm functions

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

Where $\overline{Q}$ denotes the average number of elements in the queue during the whole algorithm.

Remarks

- For the computation of the the estimates for the worst case scenarios: $\overline{Q} \le \lvert V\rvert$ and $\lvert E\rvert \in \mathcal{O}\left(\lvert V\rvert^2\right)$.
- Boolean State Vector is a vector of type `IsNodeInQueue[order]` (of size `order`): A node v is in the queue if and only if `IsNodeInQueue[v] = true`. This strategy, when the graph is big, wastes a lot of memory and really gives a "worst case scenario".

In the next pages one can find detailed justifications of the above estimated average run times.

Analysis of Dijkstra's Algorithm efficiency

Justification of the estimated average run times

In the next computations we set $n = |V|$ and we denote by Q_i the number of elements in the queue for the repetition i of the `while` loop, with $i = 1, 2, \dots, n$. Also, we denote by d_i (respectively a_i) the total number of times that the function `decrease_priority` (respectively `add_with_priority`) has been run at the repetition i of the `while` loop.

Observe that: $\sum_{i=1}^{n} d_i = |E| - n$ and $\sum_{i=1}^{n} a_i = n$.

Average run time of `extract_min` for a plain linked list not sorted:

The expected run time T_{EM} *at the repetition i of the `while` loop* is $\mathcal{O}\left(\frac{Q_i}{2}\right)$. Thus, the total run time average is:

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

Analysis of Dijkstra's Algorithm efficiency

Justification of the estimated average run times

Average run time of `add_with_priority` for a linked list sorted by priority:

The expected run time T_{AwP} *at the repetition i of the* `while` *loop* is $\mathcal{O}\left(\frac{Q_i}{2}\right)$. The total run time average is:

$$\frac{1}{n}\sum_{i=1}^{n}\left(K_{j_1}+K_{j_2}+\cdots+K_{j_{a_i}}\right)\frac{Q_i}{2} \le K\frac{1}{2n}\sum_{i=1}^{n}Q_i = \mathcal{O}\left(\frac{\overline{Q}}{2}\right).$$

Average run time of `decrease_priority` for a linked list sorted by priority:

The expected run time T_{DP} *at the repetition i of the* `while` *loop* is $\mathcal{O}\left(\frac{Q_i}{2}\right)$. The total run time average is:

$$\frac{1}{|E|-n}\sum_{i=1}^{n}\left(K_{j_1}+K_{j_2}+\cdots+K_{j_{d_i}}\right)\frac{Q_i}{2} \le \frac{Kn}{|E|-n}\frac{1}{2n}\sum_{i=1}^{n}Q_i = \mathcal{O}\left(\frac{\overline{Q}}{2}\right).$$

Analysis of Dijkstra's Algorithm efficiency

Justification of the estimated average run times

Average run time of `add_with_priority` for a binary heap sorted by priority:

The expected run time T_{AwP} *at the repetition i of the `while` loop* is $\mathcal{O}\Big(\log_2(Q_i)\Big)$. Since the $\log_2$ function is concave, by *Jensen's Inequality* we have that the total run time average is:

$$\frac{1}{n}\sum_{i=1}^{n}\Big(K_{j_1}+K_{j_2}+\cdots+K_{j_{a_i}}\Big)\log_2(Q_i) \le K\frac{1}{n}\sum_{i=1}^{n}\log_2(Q_i) \overset{\text{Jensen Ineq.}}{\le} K\log_2\Big(\overline{Q}\Big) \in \mathcal{O}\Big(\log_2\Big(\overline{Q}\Big)\Big).$$

Average run time of `decrease_priority` for a binary heap sorted by priority:

The expected run time T_{DP} *at the repetition i of the `while` loop* is $\mathcal{O}\Big(\log_2(Q_i)\Big)$. Since the $\log_2$ function is concave, by *Jensen's inequality* we have that the total run time average is:

$$\frac{1}{|E|-n}\sum_{i=1}^{n}\Big(K_{j_1}+K_{j_2}+\cdots+K_{j_{d_i}}\Big)\log_2(Q_i) \le \frac{Kn}{|E|-n}\frac{1}{n}\sum_{i=1}^{n}\log_2(Q_i) \overset{\text{Jensen Ineq.}}{\le} K\log_2\Big(\overline{Q}\Big) \in \mathcal{O}\Big(\log_2\Big(\overline{Q}\Big)\Big).$$

A* Algorithm

Índex

Introduction to A* Algorithm[5]

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.

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

---

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

Introduction to A* Algorithm

The heuristic function[6] 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.

---

[6] 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                     } ▷ General initialization
    parent[G.order] ← uninitialized               }
    g[G.order] ← initialized to ∞                 } → Important to detect the non-visited (and non-enqueued) nodes

    g[start] ← 0                                  } ▷ Open set initialization: start has distance 0
    parent[start] ← ∞                             }   to itself, has no parent and is enqueued
    Open.add_with_priority(start, g, h)           }

    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] + $\omega$(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)   { Relaxation step ◁
                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
```

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

The A* Philosophy

A*–Remark

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* Philosophy: Another view of A*

The basic operative of the A* Algorithm is based on the construction (exploration) of paths in the following sense:

Definition

Let $\alpha := \bigl(v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_{n-1} \longrightarrow v_n\bigr)$ be a path in the graph G. We say that α has been *constructed by the A* Algorithm* if, at some of the A* iterates, v_n is relaxed after the expansion of v_{n-1}, $g(v_i) < \infty$ for $i = 0, 1, \dots, n$, and $v_j = \texttt{parent}[v_{j+1}]$ for $j = 0, 1, \dots, n-1$.

Then, a basic result about the A* Algorithm that complements the A*–Remark is the following:

A* Basic Lemma

All paths constructed by the A* Algorithm are acyclic.

A consequence of the A* Basic Lemma is that the basic operative of the A* Algorithm constructs a subset of the acyclic paths strating at ξ, and traverses the subgraph of G formed by the union of these acyclic paths.

The A* Philosophy: Another view of A* — Proofs

Remark (the A* implemented path information does not allow cyclic paths)

The A* (and Dijkstra) strategy of constructing backwards the shortest paths, which is based on keeping track of the predecessor (`parent[]`) of each node on the computed shortest path, can never give as a result a cyclic path because every node can have a unique parent.

Thus, the implemented way of constructing the shortest paths (fortunately) agrees with the previous lemma.

Proof of A* Basic Lemma

Assume that A* has just constructed a cyclic path $(x_0 \longrightarrow x_1 \longrightarrow \cdots \longrightarrow x_k)\alpha$, where $\alpha := (v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_{n-1} \longrightarrow v_n)$ is a loop (i.e., $v_n = v_0$). Without loss of generality we may assume that α is acyclic (i.e., $v_0, v_1, \ldots, v_{n-1}$ are pairwise different). Then, prior to the relaxation of v_n after the expansion of v_{n-1} the nodes $v_0 = \texttt{parent}[v_1], v_1 = \texttt{parent}[v_2], \ldots, v_{n-2} = \texttt{parent}[v_{n-1}]$ and v_{n-1} have been previously relaxed. Thus, by A*–Remark,

$$\begin{aligned} g(v_0) = g(v_n) &> g(v_{n-1}) + \omega(v_{n-1}, v_n) \\ &= g(v_{n-2}) + \omega(v_{n-2}, v_{n-1}) + \omega(v_{n-1}, v_n) = \cdots \\ &= g(v_0) + \sum_{j=0}^{n-1} \omega(v_j, v_{j+1}) > g(v_0); \end{aligned}$$

a contradiction.

On the heuristic function

It is clearly seen that the whole algorithm and, in particular, its efficiency depend on the heuristic function.

As we will see, the best heuristic function is the one that estimates (but never overestimates) the actual cost to get to the goal.

Example

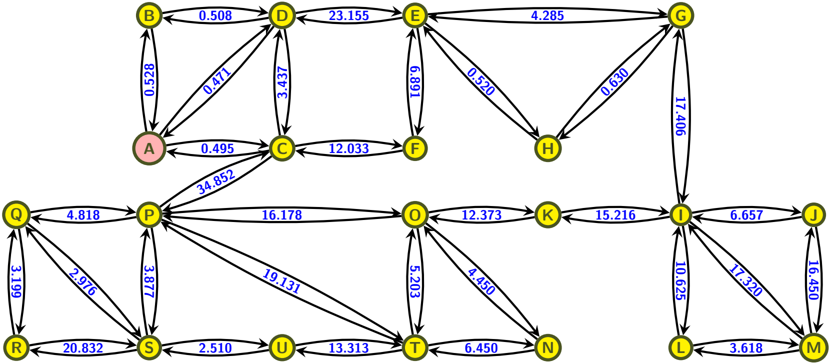
Let $G = (V, E, \omega)$ be a weighted graph and let γ denote the goal node. For every vertex $v \in V$ we set

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

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

An example of the $A^\star$ Algorithm

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

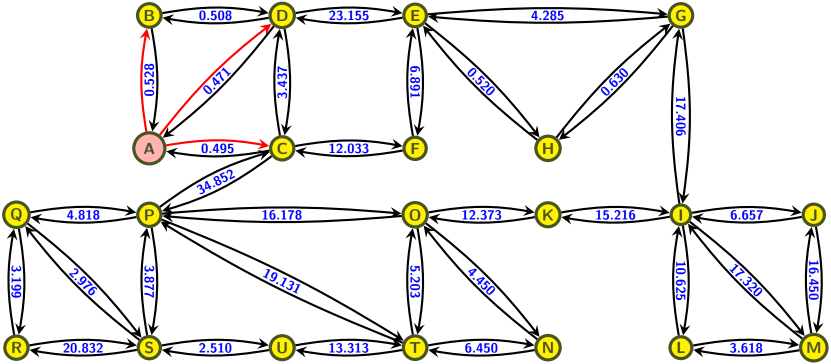


Open Queue	**A**
g	0
f	0.471
parent	nil

Observe that the f-values of the expanded nodes are non-decreasing and there is no re-opened node (to be expanded again), as prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



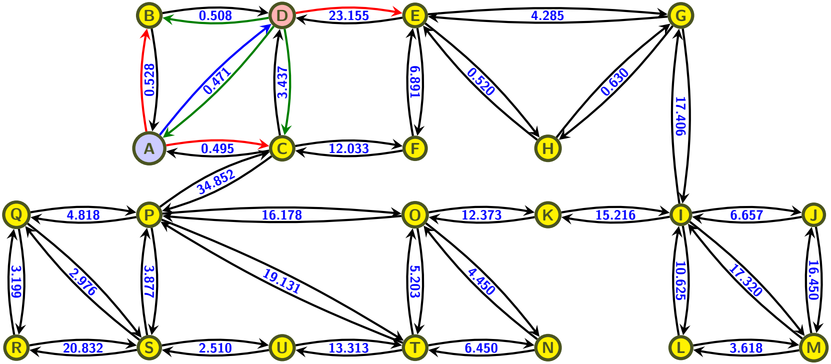
Open Queue	**D**	**C**	**B**
g	0.471	0.495	0.528
f	0.942	0.99	1.036
parent	A	A	A

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



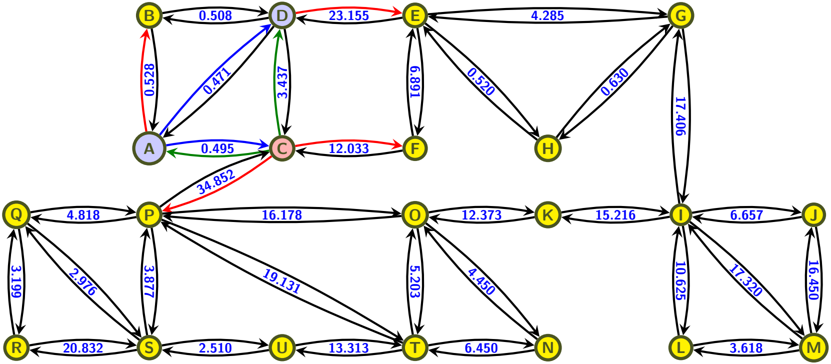
Open Queue	**C**	**B**	**E**
g	0.495	0.528	23.626
f	0.99	1.036	24.146
parent	A	A	D

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



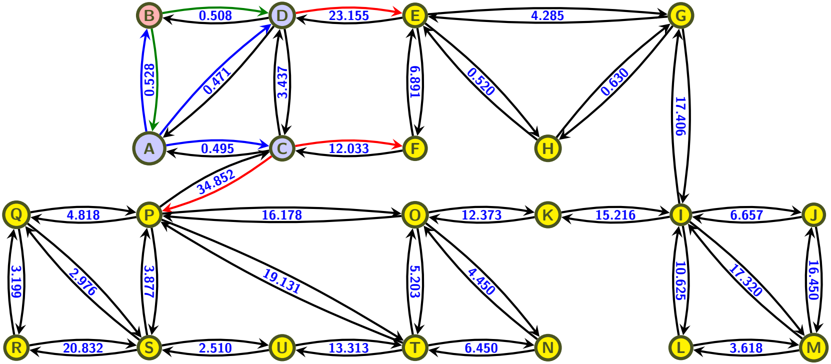
Open Queue	**B**	**F**	**E**	**P**
g	0.528	12.528	23.626	35.347
f	1.036	19.419	24.146	39.224
parent	A	C	D	C

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 prescribed by the fact that the heuristic is monotone.

An example of the A⋆ Algorithm

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



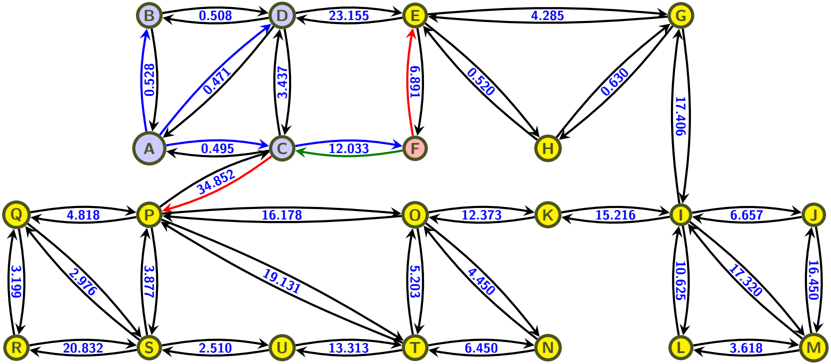
Open Queue	**F**	**E**	**P**
g	12.528	23.626	35.347
f	19.419	24.146	39.224
parent	C	D	C

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



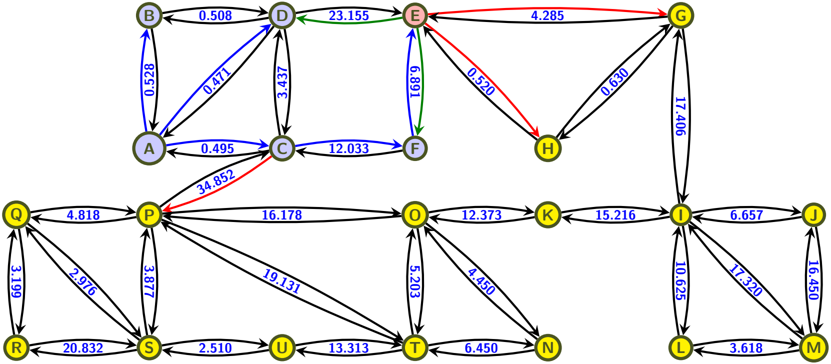
Open Queue	E	P
g	19.419	35.347
f	19.939	39.224
parent	F	C

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



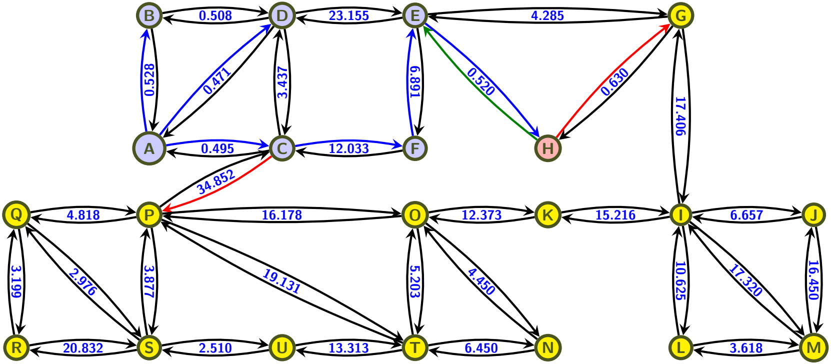
Open Queue	**H**	**G**	**P**
g	19.939	23.704	35.347
f	20.459	24.334	39.224
parent	E	E	C

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



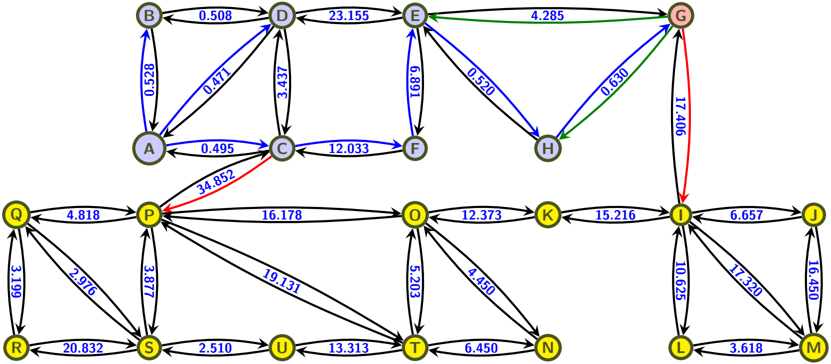
Open Queue	G	P
g	20.569	35.347
f	21.199	39.224
parent	H	C

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



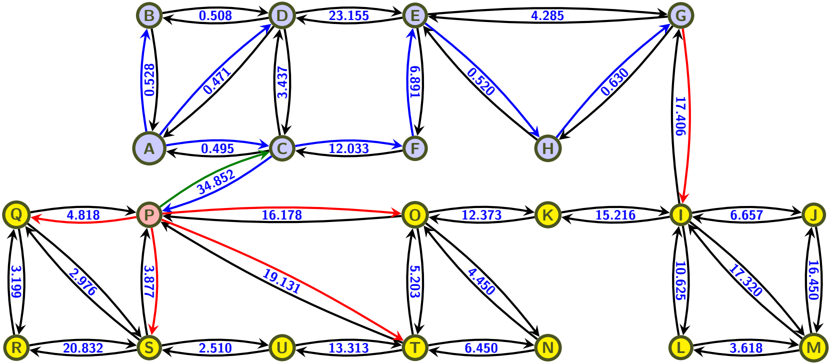
Open Queue	**P**	**I**
g	35.347	37.975
f	39.224	44.632
parent	C	G

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 prescribed by the fact that the heuristic is monotone.

An example of the A⋆ Algorithm

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



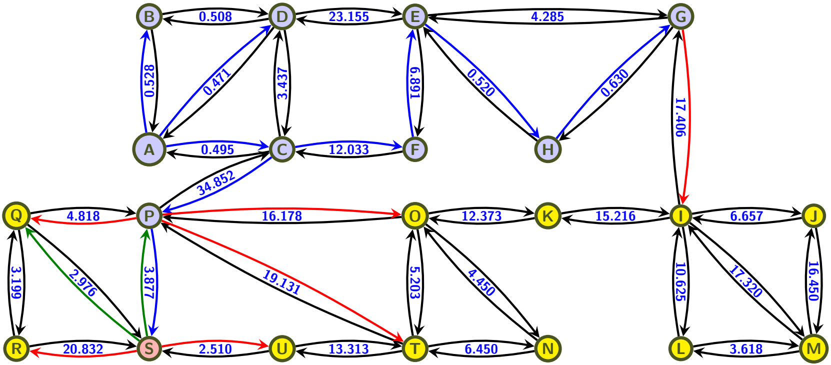
Open Queue	**S**	**Q**	**I**	**O**	**T**
g	39.224	40.165	37.975	51.525	54.478
f	41.734	43.141	44.632	55.975	59.681
parent	P	P	G	P	P

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



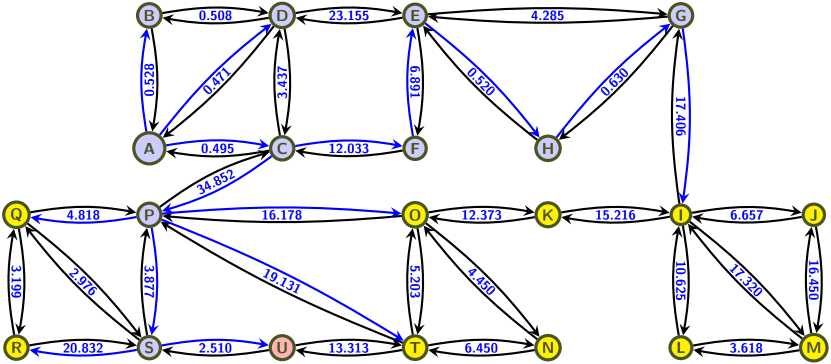
Open Queue	**U**	**Q**	**I**	**O**	**T**	**R**
g	41.734	40.165	37.975	51.525	54.478	60.056
f	41.734	43.141	44.632	55.975	59.681	63.255
parent	S	P	G	P	P	S

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 prescribed by the fact that the heuristic is monotone.

An example of the A* Algorithm

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



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 prescribed by the fact that the heuristic is monotone.

Implementation of the *$A^\star$ Algorithm* in **C**

Declarations and auxiliary functions

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; }
```

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`.

Question

Is the heuristic function a good one? 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 *$A^\star$ 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} }} };

  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
```

Implementation of the $A^\star$ *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 Dijkstra 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 Dijkstra 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 check easily whether a given node v belongs to the queue: It does so if and only if `Q[v].IsOpen` is `true`.

t we have
`g` $= 0.0$.

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;
}
```

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

The Dijkstra 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 $A^\star$ *Algorithm* in **C**

Priority queue functions code — Alike Dijkstra's algorithm

```
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;
}
```

Algorithmic properties of A*:
Termination and Completeness

Theorem

A* always terminates on finite graphs.

Remark (finiteness of acyclic paths starting at ξ)

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

Let m_ξ denote the out-degree of ξ in C, let m denote the maximum out-degree of a vertex in C, and let ℓ denote the number of vertices in C (including ξ).

Since a path is acyclic if and only if every vertex appears at most once in the path, the length of an acyclic path starting at ξ is smaller than or equal to $\ell - 1$. So, the number of acyclic paths starting at ξ can be brutally upper bounded by $m_\xi \cdot m^{\ell-2}$.

Proof

If A* does not stop after finding a solution (by extracting γ from the Open Queue with the function `extract_min`) then, by A* Basic Lemma and the above Remark, it will traverse the subgraph of G formed by the union of finitely many acyclic paths starting at ξ in finite time. Upon completion of this traversal, the Open Queue will become empty and A* will stop with failure.

Algorithmic properties of A*:

Termination and Completeness

Completeness

An algorithm is said to be *complete* if it terminates with a solution when one exists.

Completeness Theorem

A* is complete (even on infinite graphs).

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

A* is admissible.

Example (the heuristic function from Page 48 is admissible)

If $v = \gamma$ we have: $h(v) = h(\gamma) = 0 \leq \sigma(v, \gamma)$.
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 (Consistent) 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 $u, v \in V$.

A 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 (Consistent) 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

A 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 48 is monotone)

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*:

Properties of Monotone Heuristics

$\xi \in V$ denotes the source node

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,

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

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.

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 by the strict inequality

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