Université Libre de Bruxelles

Master thesis

Speed-up techniques for Shortest Path problems

Author
Alexandre HENEFFE

Supervisors
Dr. Hugues Bersini
Matvei Tsishyn

April 23, 2020

Contents

1 Introduction

The shortest path problem is one of the most important in combinatorial optimization. We can be interested in finding the shortest path between 2 points. This problem is called the **single-pair** shortest path problem. There are several variants to this problem and they can be dealt with significantly more efficient algorithms than simply running a single-pair shortest path algorithm on all relevant pairs of vertices. The **single-source** shortest path problem consists in finding the shortest path from one single source vertex to all other vertices in the graph. The inverse is the **single-destination** shortest path problem in which we desire to find the shortest path from all vertices to a single destination vertex in the graph. If we invert the edges in the directed graph, we obtain the reduced problem, which is the single-source shortest path problem. Another variant is the **all-pairs** shortest path problem, which are the shortest path between all pairs of vertices in the graph. This can be solved using the so-called Floyd-Warshall algorithm.

The well-known Dijkstra algorithm solves the shortest path problem in any weighted graph with non-negative weights in polynomial time. If the weights are not non-negative, it can be solved using the Bellman-Ford algorithm which requires the additional condition that the graph does not contain any negative cycle reachable from the source node. Indeed, finding a shortest path in a graph containing negative cycles is known to be NP-hard. A path with no repeated vertices is called simple-path. The longest-path problem is the problem of finding a simple path of maximum length in a given graph. This problem is NP-hard (it can be proved using a reduction from the Hamiltonian path problem [10]). Its decision problem can be formulated as asking whether a path exists of a least some given length. This decision problem is NP-Complete (all problems which are in NP can be reduced to this problem in polynomial time), and therefore not solvable in polynomial time. The longest-path problem can be reduced to the shortest-path problem containing negative cycles to prove that this is NP-hard. Indeed, given a graph with only positive-weights. Suppose we have a polynomial algorithm for solving our problem. If we negate all the edge-weights and run this algorithm, it would give the longest path in the original graph. Since the longest-path problem is NP-hard, this is not possible to obtain the longest path, and therefore the original problem is NP-hard.

Despite the fact that there exists several algorithms that solve the shortest-path problems, such as Dijkstra's algorithm or Bellman-Ford's algorithm, there is a recent increase of the amount of data. We feel the need to drastically decrease the execution time of the algorithm. In the latest years, we observe the emergence of a lot of new speed-up techniques; among them there are: heuristic search, bidirectional search, landmarks, reaches, short-cuts and contractions. The efficiency of these techniques heavily depends on the topology of the graph. Moreover, graphs with more complex structures (as user-adapted graphs) do not support some of these techniques, at least as they are currently presented. From these observations, several questions arise. Which optimization methods are the most efficient depending on the graph topology? How

can we adapt these techniques to more complex graph structures? We will first explore the state of the art of the shortest path problem by describing several methods and algorithms that are most commonly used to solve this problem. Then we will look at different existing speed-up techniques and compare their performances on different graph topologies.

2 Real world applications

Finding shortest paths in a graph is a problem that applies in many real-world applications. If we represent a nondeterministic abstract machine as a graph where edges describe all transitions and vertices all states, we can use shortest path algorithms to find the sequence of actions needed to reach a given goal which is optimal in terms of time or any kind of weights. The shortest path is also widely used in networking or telecommunications. And of course, it can be seen as a routing problem.

A road network can be represented as a graph. The problem of finding the point-to-point shortest path in road networks has received great attention in recent years and even though Dijkstra's algorithm can solve this problem in almost linear time, there is a problem regarding the network size. Content-size road networks require more efficient algorithms. In the road network itself, the nodes represent road junctions and the edges of that graph have weights can correspond to the length of he roads, the time need to traverse the segment, or the cost of traversing it. If we want to constraint some roads to be one-way streets, we can use directed edges.
Some edges could be more important than others regarding their weights (Highways for instance are useful for long distance travel and are important to consider in priority when searching for the shortest path). This property has been formalised with the notion of highway dimension [12]. "Intuitively, a graph G = (V, E) has a small highway dimension if for every $r > 0$, there is a sparse set of vertices S_r such that every shortest path of length greater than r includes a vertex from S_r. A set is sparse if every ball of radius $O(r)$ contains a small number of elements of S_r." It has been shown that low highway dimension gives guarantees of efficiency for several algorithms, such as Reach, contraction hierarchies, transit node, etc. The algorithms that exploit this property can thus find the shortest path a lot quicker.

The fastest known algorithm is called hub labeling [13] and can find the shortest path of road networks of Europe or the USA in a fraction of a microsecond. This algorithm and other algorithms of this kind usually work in two phases. The first phase consists of a preprocessing on the graph without taking into account the source and the target node. The second phase is a query, where the source and the target node are known. Since the road network is static, the preprocessing can be done once and can be used for several queries on the same network. The preprocessing part is important since it reduces the computation time and allows algorithms to run in sub-linear time.

3 Formalities

3.1 Definition

Definition In graph theory, the shortest path problem consists in finding the path between two nodes (vertices) such that the sum of weights of each edges that constitute it is minimal.

Formal definition The shortest path problem can be defined for undirected, directed or mixed graphs. It is defined here for undirected graphs. For directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

Let us consider a undirected graph $G = (V, E)$ with the set of nodes V and the set of edges E. Let n be the cardinality of V.

Two vertices are adjacent when they are both incident to a common edge. A path in an undirected graph $G(V, E)$ is a sequence of vertices

$$P = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$$

such that v_i is adjacent to v_{i+1} for $1 \leq i < n$. Such a path P is called a path of length n-1 from v_1 to v_n. Here v_i is a variable and its number relates to its position in the sequence of nodes forming the path and not to a specific label with real signification.

If we used a directed graph, we must ensure that there is a directed edge from v_i to v_{i+1} and not from v_{i+1} to v_i for every pair of adjacent edges for $1 \leq i < n$. For mixed graphs, a path is constitued of both undirected edges and directed edges. Therefore, each pair of successive adjacent vertices v_i and v_{i+1} have to satify the same property as in directed graphs.

Let $e_{i,j}$ be the edge incident to both v_i and v_j. Given a real-valued weight function

$$f : E \to \mathbb{R}$$

and an undirected (simple) graph G, the shortest path from s to t, where s represents the starting node and t represents the destination node, is the path $P = (v_1, v_2, \ldots, v_n)$ (where $v_1 = s$ and $v_n = t$) that over all possible n minimizes the sum

$$\sum_{i=1}^{n-1} f(e_{i,i+1})$$

.

When all edge in the graph have a unit weight (the same constant value) or $f : E \to \{1\}$, this is equivalent to finding the path with fewest edges since each edge has the same importance when passing through it.

3.2 Integer program formulation

Since the shortest path problem consists in finding the shortest path, it is a combinatorial optimisation problem, where the optimal value we are searching is the shortest path regarding the configuration of the graph we are considering. Since all the quantifiable relationships in the problem are linear and the nodes are constrained in some way, we can use linear integer programming. Let us then define a linear integer programming formulation [11] of the classical shortest path problem using a directed graph.

We consider a directed graph $G = (V, A)$ with a set of vertices V and a set of arcs A. Let n and m be the cardinalities of V and E respectively. A path in the graph is the sequence of vertices

$$P = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$$

which is elementary. A path is elementary if no vertex appears more than once in the graph. We denote $\delta^+(i)$ the set of outgoing arcs of vertex i and $\delta^-(i)$ the set of incoming arcs of vertex i. $\delta^+(S)$ and $\delta^-(S)$ are the arcs leaving/entering the set $S \subseteq V$. We assume, without loss of generality, that $|\delta^-(s)| = |\delta^+(t)| = 0$. s is the starting vertex of the shortest path we want to find and t is the destination vertex. There is no arcs going into the starting node s and no arcs going out of the destination node t.

The classical integral programming formulation to determine a shortest path from vertex s to vertex t can be written like this (primal form):

3.2.1 Primal

$$\text{Minimize} \quad \sum_{(i,j)\in A}^{m} c_{ij}x_{ij}$$

$$\text{Subject to} \quad \sum_{(i,j)\in\delta^+(i)} x_{ij} - \sum_{(j,i)\in\delta^-(i)} x_{ji} = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{si } i = t \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i \in V$$

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} \leq 1 \qquad \forall i \in V$$

$$x_{i,j} \in \{0, 1\} \qquad \forall (i,j) \in A$$

---

where $c_{ij} \in \mathbb{R}$ are the edges costs, and

$$x_{ij} = \begin{cases} 1 & \text{if the edge (i,j) is part of the path} \\ 0 & \text{otherwise.} \end{cases}$$

It is obvious that we want to minimize the total costs, which is the sum of all arcs's costs that constitute the path. The first constraint is the flow conservation constraint. It ensures that for each vertex only has one arc going into it and one arc going out of it, except for the source node s (No arc is entering it) and for the destination node t (No arc is going out of it). The second constraint ensures that the outgoing degree of each node is at most one (it is 0 for the destination node).

3.2.2 Dual

We can formulate the dual version of this problem.

$$\begin{aligned} \text{Maximize} \quad & \pi_t - \pi_s \\ \text{Subject to} \quad & \pi_j - \pi_i \leq c_{ij} \quad \forall e = (i,j) \in A \\ & \pi \in \mathbb{R}^{|V|} \end{aligned}$$

---

where π_i in this problem can be interpreted as being some sort of "distance" (a real value) of each node i towards the starting node s. In this view, we can say that we want to put the most distance between s and t with respect to the cost matrix c. To do that, we maximize the difference between the distance towards t and the distance towards s. In this case, s is at distance 0 and when following an arc (i, j), we have $\pi_j \leq \pi_i + c_{ij}$. An interpretation of this linear program is that the graph is being embedded on a line, with π_i representing the position of the vertex i on a line, subject to the constraints that for every edge (i,j), there is a "string" preventing vertex j from being put more than $c_{i,j}$ units further along the line than vertex i. We can observe in particular that by linear programming duality, if $x_{ij} = 1$ (we choose to take arc (i,j) in the path of the primal), then by complementarity we must have

$$\pi_j = \pi_i + c_{ij}$$

Furthermore, note that we also need to satisfy all the other inequality constraints

$$\pi_j \leq \pi_k + c_{kj}$$

for the nodes that are not chosen in the corresponding primal path, which can only be satisfied when the distances correspond to at most the distance travelled in a shortest path.

3.2.3 Property

In combinatorial optimisation, if we have a primal problem and its dual, we can have a weak duality or a strong duality.

Given a primal problem P

Maximize $c^T x$

Subject to $Ax \leq b, \quad x \geq 0$

and its dual problem D

Minimize $b^T y$

Subject to $A^T y \geq c, \quad y \geq 0$

The weak duality states [6] that the duality gap (the difference between the primal and the dual solutions) is always greater than or equal to 0. It means that the solution of the primal problem is always greater than or equal to the solution of the associated dual problem : $c^T x \leq b^T y$. So any feasible solution to the dual problem corresponds to an upper bound on any solution to the primal problem. If $(x_1, x_2, \ldots, x_n)$ is a feasible solution for the primal linear program and $(y_1, y_2, \ldots, y_n)$ is a feasible solution for the dual linear program, then the weak duality theorem states that

$$\sum_{j=1}^{n} c_j xj \leq \sum_{i=1}^{m} b_i y_i$$

where c_j and b_i are the coefficients of the objective functions, respectively. Generally, if x is a feasible solution of the primal and y is a feasible solution for the dual, then the weak duality implies $f(x) \leq g(y)$, where f and g are the objective functions of the primal and the dual, respectively.

The strong duality also satisfies the weak duality condition and the additionnal condition that states that the primal optimal objective and the dual optimal objective are equal.

Formally, if there exists an optimal solution x' for P, then there exists an optimal solution y' for D and the value of x' in P equals the value of y' in D.

This property is very useful and the shortest path problem actually satisfies the strong duality property. Denote by π_1* an optimal solution to the dual problem and by $x*$ an optimal solution to the primal problem. Also denote by $v(\pi) = \pi_t$ and by $u(x) = \sum_{(i,j)\in A} c_{ij} x_{ij}$. We could prove strong duality by proving that

$$v(\pi_1*) = u(x*)$$

.

Since the shortest path problem satisfies the strong duality property, it is sufficient to find an optimal solution to the dual in order to obtain the optimal solution to the primal problem.

4 Algorithms

4.1 Classical algorithms

The shortest path problem can be solved using many algorithms and among them, the most important ones are described hereafter. Each one of them deals with different graph topologies and problem formulations. Dijkstra's algorithm solves the single-source shortest path problem with non-negative edge weights. Bellman–Ford algorithm solves the single-source problem if edge weights may be negative. A* search algorithm solves for single pair shortest path using heuristics to try to speed up the search. Floyd–Warshall algorithm solves all pairs shortest paths. Johnson's algorithm solves all pairs shortest paths, and may be faster than Floyd–Warshall on sparse graphs.

Dijkstra The well-known Dijkstra's algorithm [4] finds the shortest path between single nodes in a graph, as long as the edge weights are non-negative. There exists many variants, and a common variant computes the shortest path from a fixed source node to all other nodes in the graph, producing a tree.

The algorithm itself works with a priority queue and is quite simple.

- At the initialisation phase, each node in the graph is marked as "unvisited". A first set is created, containing all the unvisited nodes, and a second set is created, containing all the visited node (initially empty). Then, every node receives a *tentative* distance values, which is ∞, except for the starting node (0). The tentative distance values is the distance from the start node to a given node.

- Set the current node to be the initial node.

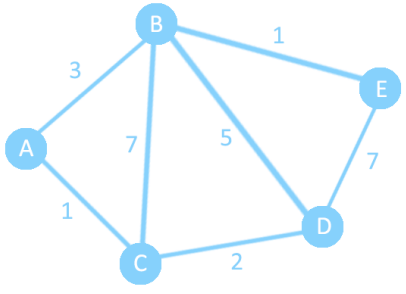
- Consider the current node. Get all the unvisited neighbors of this node and compute their *tentative* distance value. Formally, let the current node be c, along with a tentative value t and a neighbor n, with a tentative value x. The new *tentative* distance value assigned to n is $s = t + d(c, n)$ if $x > s$, where $d(c, n)$ is the weight of the edge (c, n). For example, if the current node A is marked with distance 6, its neighbor B with distance 9 and the edge connecting the two is 2, then B's tentative distance value has the new value $6 + 2 = 8 < 9$.

- Mark the current node as visited (add it to the visited set and remove it from the unvisited set so that it is never considered again by the algorithm)

- Stop whenever the destination node is set as current node and then marked as visited, or if the smallest tentative distance among the nodes in the unvisited set is ∞ (This can happen if the initial node and the remaining unvisited nodes have no connection between them). If those 2 conditions are not satisfied, then select the unvisited node that has the smallest tentative distance, set it to be the new current node and go back to the third step.

```
1:  function DIJKSTRA(Graph, start_node)
2:      unvisited ← priority queue
3:      visited ← empty set
4:      dist[start_node] ← 0
5:      for each vertex v in Graph do                    ▷ And v ≠ start_node
6:          dist[v] ← ∞                                  ▷ distance from start to v
7:          prev[v] ← None                               ▷ Predecessor of v
8:          unvisited.add_with_priority(v, dist[v]
9:          add v to unvisited
10:     end for
11:     while unvisited not empty do
12:         u ← unvisited.extract_minimum()              ▷ vertex with min dist[u]
13:         remove v from unvisited
14:         add v to visited
15:         for each neighbor v of u do                  ▷ Only unvisited neighbors
16:             tentative_dist ← dist[u] + d(u,v)        ▷ d(u,v) = edge length
17:             if tentative_dist < dist[v] then
18:                 dist[v] ← tentative_dist
19:                 prev[v] ← u
20:                 unvisited.decrease(v, tentative_dist)
21:             end if
22:         end for
23:     end while
24:     return dist[], prev[]
25: end function
```

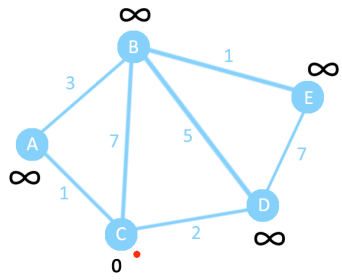
When the algorithm has found a solution, we must look at the "prev" array and start from the goal node, going back successively to the starting node to get the full shortest path.



We can develop an example by running the algorithm on this graph. The start node is C and the goal state is the node E. We expect the algorithm to find the shortest path which is

$$C - A - B - E$$

with a total distance of 5 in our case. The algorithm will terminate when the node E is visited.

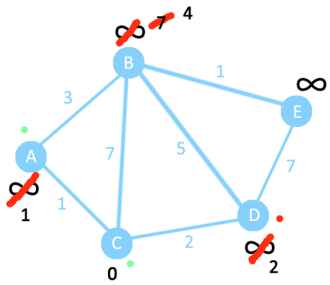


UNVISITED

Node	C	A	B	D	E
Distance	0	∞	∞	∞	∞
Previous	-	-	-	-	-

VISITED

Node	-
Distance	-
Previous	-

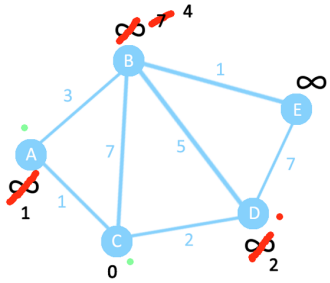


UNVISITED

Node	A	D	B	E
Distance	1	2	7	∞
Previous	C	C	C	-

VISITED

Node	C
Distance	0
Previous	-

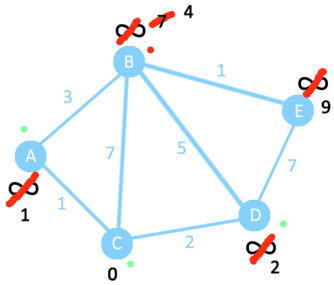


UNVISITED

Node	D	B	E
Distance	2	4	∞
Previous	C	A	-

VISITED

Node	C	A
Distance	0	1
Previous	-	C

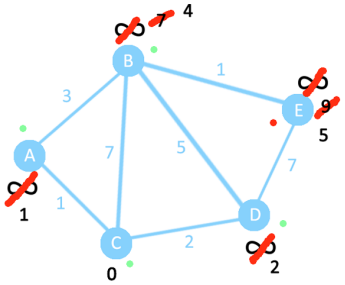


UNVISITED

Node	B	E
Distance	4	9
Previous	A	D

VISITED

Node	C	A	D
Distance	0	1	2
Previous	-	C	C

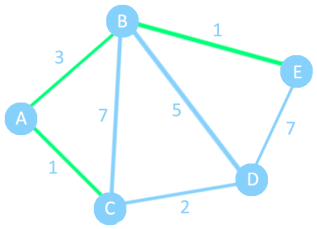


UNVISITED

Node	E
Distance	5
Previous	B

VISITED

Node	C	A	D	B
Distance	0	1	2	4
Previous	-	C	C	A



UNVISITED

Node	-
Distance	-
Previous	-

VISITED

Node	C	A	D	B	E
Distance	0	1	2	4	5
Previous	-	C	C	A	B

Since the node E has been visited by the algorithm, we can reconstitute the shortest path by looking at the succession of previous nodes in the visited set. E is preceded by B. B is preceded by A, and A is preceded by C. It gives us the shortest path

$$C - A - B - E$$

with a distance of 5, as expected. We could also have computed all shortest paths from node C. In terms of performances, Dijkstra's algorithm's time complexity depends on the number of edges $|E|$ and vertices $|V|$ and the data structure used to implement the "unvisited" set Q. If the set Q is represented as a simple array and the extraction time of the minimum value in it is linear, then the algorithms runs in $\prime(|E|+|V|^2) = \mathcal{O}(|V|^2)$. Generically, for any data structure of the set Q, the time complexity is $\mathcal{O}(|E|.T_d + |V|.T_{em})$, where T_d is the time needed to compute the *decrease* function (decrease the priority of keys in the set Q), and T_{em} is the time needed to compute the $extract\_minimum$ operation (extract the minimum value from the set Q).

Bellman-Ford Bellman-Ford algorithm [4] [1] finds the shortest path from a source node to all other nodes in a directed graph. Unlike Dijkstra's algorithm, this algorithm can deal with graphs containing negative weights. If there are negative cycles in the graph that are reachable from the source node, then Bellman-Ford cannot find any shortest path since any path with a node that is part of a negative cycle can be optimised by moving in that negative cycle. Bellman-Ford algorithm is able to detect negative cycles during its execution.
The algorithm itself is quite similar to Dijkstra's algorithm in the sense that it relaxes edge weights. The difference is that Dijkstra's uses a priority queue in order to extract the minimum distance value at each step whereas Bellman-Ford proceeds to relax all the nodes for $|V|$ iterations. We denote "relaxation" by the update of the distance of a vertex by a smaller tentative distance value. When the algorithm has finished relaxing the edges, it look at all edges again in order to detect negative weight cycles. If some edges can still be relaxed, it means that there is a negative cycle. The algorithm makes $|E|$ relaxations for every iteration and there are $|V|-1$ iterations. Thus, the worse-case execution time is $\mathcal{O}(|V|.|E|)$.

```
1: function BellmanFord(Graph, vertices, edges)
2:     distance ← map for every vertex
3:     previous ← map for every vertex
4:     dist[start_node] ← 0
5:     for each vertex v in Graph do                    ▷ And v ≠ start_node
6:         dist[v] ← ∞                                  ▷ distance from start to v
7:         prev[v] ← None                               ▷ Predecessor of v
8:     end for
9:     for i from 1 to size(vertices) - 1 do            ▷ |V| - 1 repetitions
10:        for each edge (u,v) with weight w in edges do
11:            if distance[u] + w < distance[v] then
12:                distance[v] ← distance[v] + w        ▷ Relax edge
13:                previous[v] ← u
14:            end if
15:        end for
16:    end for
17:    for each edge (u,v) with weight w in edges do    ▷ Checks for negative cycles
18:        if distance[u] + w < distance[v] then
19:            return error                             ▷ Contains negative weight cycle
20:        end if
21:    end for
22:    return distance[], previous[]
23: end function
```

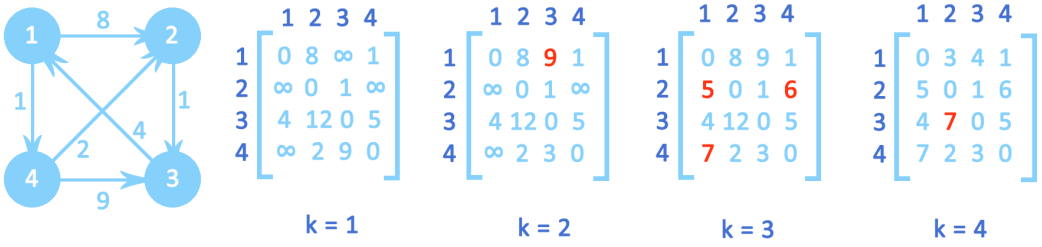
Floyd-Warshall The Floyd-Warshall algorithm [4] [7] [2] computes the shortest path from all pairs of nodes in a weighted graph with positive or negative edge weights, but not containing any negative cycles. The principle behind this algorithm is based on a recursive property. Let G be a graph with vertices V numbered from 1 to N. We define $ShortestPath(i, j, k)$, a function that returns the shortest path from node i to node j so that the intermediate points on that path belong to the set $\{1, 2, \ldots, N\}$. In the graph, for each pairs of vertices i, j, $ShortestPath(i, j, k)$ is either

- a path that contains only vertices from the set $\{1, \ldots, k-1\}$, not containing k.
- a path that contains k, from i to k and then from k to j, both using vertices from the set $\{1, \ldots, k-1\}$. If the shortest path happens to be from i to k and then from k to j, then it means that the length of this path from i to j, with k along, is the concatenation of the shortest path from i to k (only using intermediate nodes from $\{1, \ldots, k-1\}$), and the shortest path from k to j (only using intermediate vertices in $\{1, \ldots, k-1\}$) : $shortestPath(i, k, k-1) + shortestPath(k, j, k-1)$).

The base case of the property is : $SP(i, j, 0) = w(i, j)$, where $w(i, j)$ is the weight of the edge (i, j), and SP is an abbreviation of "$shortestPath$". The recursive case is

defined by : $SP(i,j,k) = min(SP(i,j,k-1), SP(i,k,k-1) + SP(k,j,k-1))$. The algorithm hereafter uses this recursive property and try different values of k from 1 to V, the number of vertices.

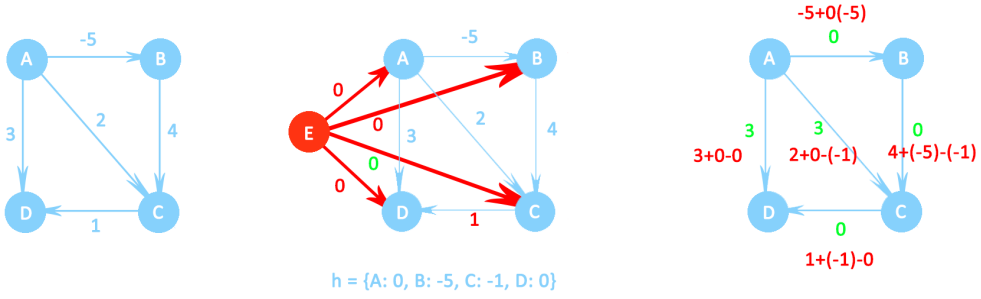
```
1:  function FloydWarshall(Graph, vertices, edges)
2:      dist ← array                                  ▷ Of size |V| x |V|
3:      for each edge (u,v) in edges do
4:          dist[u][v] ← w(u,v)
5:      end for
6:      for each vertex v in vertices do
7:          dist[v][v] ← 0
8:      end for
9:      for k from 1 to |V| do
10:         for i from 1 to |V| do
11:             for j from 1 to |V| do
12:                 if dist[i][j] > dist[i][k] + dist[k][j] then
13:                     dist[i][j] ← dist[i][k] + dist[k][j]
14:                 end if
15:             end for
16:         end for
17:     end for
18: end function
```



This algorithm runs in $\mathcal{O}(|V|^3)$ and is used to compute shortest path between all pairs of vertices in dense graphs. Indeed, to compute all $shortestPath(i,j,k)$ and all $shortestPath(i,j,k-1)$, it requires $2|V|^2$ operations. Also, the algorithms computes this function for each k from 1 to $|V|$. Therefore, the number of operations in total is $|V|.2|V|^2 = 2|V|^3 = \mathcal{O}(|V|^3)$ In sparse graphs with non-negative edge weights, Dijkstra is prefered as its running time is better ($\mathcal{O}(|E||V| + |V|^2 \log |V|)$ using fibonacci heaps). If we have sparse graphs with negative edges but no negative cycles, we can use Johnson's algorithm, that we describe herafter.

Jonhson's algorithm The Johnson's algorithm [4] [3], like Floyd-Warshall, finds the shortest path between all pairs of vertices in a weighted directed graph with possibly negative weights, but no negative-weight cycles. We could try to run Dijkstra for every vertex to obtain a set of single-source shortest paths. The problem is that it doesn't deal with negative weights. Johnson's algorithm actually uses two existing algorithms that we saw, to compute the shortest paths : Bellman-ford and Dijkstra's algorithm. The idea is to use Bellman-ford algorithm on the initial graph to find the shortest path from a additional node to all other nodes, and assign a weight to every vertex in the graph. Then, the weights of the graph are modified according to these vertex weights in order to get rid of negative weights. Finally, Dijkstra's algorithm is applied on the transformed graph. Formally,

- Let G be the considered graph. Add a new artificial node q to G, connected to every other nodes in the graph. Let G' be the new graph.
- Apply Bellman-Ford algorithm on G' to get all shortest paths from the source node q to all other nodes. It will find for each vertex v the minimum weight $h(v)$ of a path from q to v. Note that the algorithm can detect negative cycles if there exists some. In this case, the entire algorithm terminates.
- Update every weights of the graph. Given an edge from u and v, with length $w(u, v)$, its new value is $w(u, v) = w(u, v) + h(u) - h(v)$.
- Remove q and apply Dijkstra's algorithm to find the shortest paths from each starting node s to every other vertex in the new graph.



The algorithm takes $\mathcal{O}(|V||E|)$ time to run Bellman-Ford algorithm and $\mathcal{O}(|V| \log |V| + |E|)$ for each of the $|V|$ instanciations of Dijkstra's algorithm (using a Fibonacci heap). Thus, the time complexity of the Johnson's algorithm is $\mathcal{O}(|V|^2 \log |V| + |V||E|)$. Compared to Floyd-Warshall's algorithm, this algorithm is faster for sparse graphs as its time complexity depends onthe number of edges $|E|$, while Floyd-Warshall doesn't. If the number of edges in the graph is small (sparse graph), then $\mathcal{O}(|V|^2 \log |V| + |V||E|)$ is faster than $\mathcal{O}(|V|^3)$ (Floyd-Warshall).

4.2 Search algorithms

The shortest path problem can be seen as a search problem using agents. Indeed, an agent can establish goals that result in solving a problem by considering different sequences of actions that might achieve those goals. A goal is a desired state of the world. This goal can be thought as a set of world states in which it is satisfied or there could be multiple goals in the same world. The agent has the possibility to think about which action he can take in order to get him to the goal state. In our case, the world can be represented as a graph and the goal state is simply a path in this graph which is the shortest regarding the weights that constitute it. As said before, the goal could also be the shortest path from one node to all other nodes (single-source shortest path) or the shortest path from all nodes to one specific node (single-destination shortest path) or even the shortest path between all pairs of nodes in the graph (all-pairs shortest path). An action could be to pick up a node when searching for the shortest path in a graph. The problem is to decide what the agent should do by searching, and therefore assessing several possible sequences of actions leading to the goal state(s) and choosing the best one. To do so, we can use search algorithms that take a problem as input, formulated as a set of goals to be achieved and a set of possible actions to be applied and returns a sequence of action which is a solution. We need an initial state, which is the starting node(s), operators, which describe actions in terms of states reached by applying them, and a goal state, which is the destination node(s). The initial state and operators together define the `state space` of the problem which is the set of all states reachable from the initial state and any sequence of actions by applying the operators. We can describe a generic search algorithm framework [9] as follows:

```
1:  function AgendaSearch(start_state, actions, goal_test)   ▷ Returns an action sequence or failure
2:      seq                                                  ▷ an action sequence, initially empty
3:      agenda                                               ▷ a state sequence, initially contains start_state
4:      while True do
5:          if Empty(agenda) then
6:              return failure
7:          end if
8:          if goal_test(First(agenda)) then
9:              return seq
10:         end if
11:         agenda ← Queuing_Fn(agenda, Expand(First(agenda), actions))
12:         seq ← Append(seq, Action(First(agenda)))
13:     end while
14: end function
```

where `Queuing_Fn` is a function that determines what kind of search we are doing, `Expand` is a function that generates a set of states achieved by applying all possible

actions to the given state and `First` is a function that fives the first element of a set of elements (action of state). The `goal test` function is a boolean function over the states that gives true if the input is the goal state and false otherwise. We will describe several basic search algorithms that implements different `Queuing_Fn` functions and compare their performances in terms of complexity and capability to find the optimal solution of a given problem. The first three methods are uninformed strategies, i.e the way they expand their nodes is by only using information in the problem formulation. Then, there are informed or heuristic search strategies that use a quality measure not strictly in the problem formulation to generate solutions more effectively.

4.2.1 Uninformed search

The order of node expansion defines a strategy. Uninformed strategies only use information in the problem formulation (initial state, operator, goal state, path cost). it can be seen as a blind search or a brute force search.

Depth-first search (DFS) The state space of the problem can be structured as a tree where actions correspond to arcs and states correspond to nodes. The Depth-first search construct the state space tree by going down and going across when a leaf is encountered. The problem is that the search can be impossible if the space contains cycles or if the space is generated from continuous values because the algorithm will indefinitly going down. The depth-first search also depends on the order of the operators applications on nodes, since it could find wrong action sequences before good ones.

```
1: function DFS_SEARCH(G, start_node, end_node)          ▷ Returns a path
2:     s = Stack()                                                ▷ a stack
3:     s.push(start_node)
4:     explored = Set()
5:     while s not empty do
6:         current_node = s.pop()
7:         if current_node == end_node then
8:             return path from explored nodes
9:         end if
10:        if current_node not labeled as discovered then
11:            explored.add(current_node)
12:            for all edges from current_node to w in G.neighbors(current_node) do
    s.push(w)
13:            end for
14:        end if
15:    end while
16:    if q empty then return failure
17:    end if
18: end function
```

In the agenda search algorithm, the `Queuing_Fn` function would be implemented as a stack (Last in first out). It will remove the current node from the agenda and append its children to the front of the agenda.

Breadth-first search (BFS) An alternative method of DFS is the Breadth-First Search, which consrtuct the search tree by exploring its layers repeatedly and replacing each node by its children. This algorithm guarantees to find the solution if there exists one since it will explore all the nodes in the graph. It also guarantees to find the shortest action sequence. The problem is the tremendous amount of nodes that have to be explored if the search space becomes large, as it is usually the case in most real AI problems. In this case, the time needed to find the optimal sequence of actions is huge. In the agenda search algorithm, the `Queuing_Fn` function is implemented as a queue (First in first out). It will remove the current node from the agenda and append its children to the back of the agenda. At each step, each node of a given layer of the graph is being expanded.

```
1: function BFS_SEARCH(G, start_node, end_node)        ▷ Returns a solution or failure
2:     p = priorityQueue()                                          ▷ a priority queue
3:     p.push(start_node)
4:     explored = Set()
5:     while p not empty do
6:         current_node = p.first()                                 ▷ Lowest cost
7:         if current_node == end_node then
8:             return path from explored nodes
9:         end if
10:        if current_node not labeled as discovered then
11:            explored.add(current_node)
12:            for all edges from current_node to w in G.neighbors(current_node) do
13:                p.add(w)
14:            end for
15:        end if
16:    end while
17:    if q empty then return failure
18:    end if
19: end function
```

The compromise between Depth first search and breadth first search is the Iterative-deepening search. It consists in applying DFS but with a depth bound, forcing to behave like BFS. In the agenda search algorithm, we store the depth of each node at first (starting with a depth cutoff of 1), then only expand nodes with depths that are smaller to the depth cutoff. If it happened to be unsuccessful, we increment the depth and repeat the process again.

Uniform-cost search (UCS) The Uniform-cost search, also called Best-First Search, constructs the search tree by choosing the best node in the agenda after each expansion, according to the utility function. This function is computing the utility of a route, it can be the distance between two nodes for instance. As for the Breadth-first search, UCS guarantees to find the solution if there exists one and also guarantees to find the optimal solution (the best sequence of actions) in terms of utility. Nevertheless, as the BFS, it also suffers from the lack of efficiency when the search spaces become very large. It cannot guarantee to use less memory than BFS. In terms of agendas, `Queuing_Fn` function is a priority queue. It will remove the current node from the agenda and append its children to the agenda and then sort the whole agenda according to the utility function. To avoid recomputation of the cost so far at each node, this value is also stored in the agenda.

```
1: function UFS_SEARCH(G, start_node, end_node)          ▷ Returns a path
2:     s = Stack()                                                ▷ a stack
3:     s.push(start_node)
4:     explored = Set()
5:     while s not empty do
6:         current_node = s.pop()
7:         if current_node == end_node then
8:             return path from explored nodes
9:         end if
10:        if current_node not labeled as discovered then
11:            explored.add(current_node)
12:            for all edges from current_node to w in G.neighbors(current_node) do
   s.push(w)
13:            end for
14:        end if
15:    end while
16:    if q empty then return failure
17:    end if
18: end function
```

4.2.2 Informed search

Informed search strategies rely on heuristics, a quality measure that is not in the problem formulation and that allows to guide the order in which nodes are expanded. The informed search methods particular information about the state space in order to generate the solutions in a more efficient way. The heuristic describes the desirability of expanding nodes. It is an approximation since we cannot know in advance which node to expand in order to find the best solution. The most desirable nodes are processed first so that the solutions found are optimal.

Hill-climbing search Hill-climbing search, also called Greedy search, is the simplest kind of informed search. It uses a cost estimate from the current node to the solution. In the shortest path problem, the heuristic would be the straight-line distance to a given goal state and is based on the idea of geometrical distance estimates of distance in a graph. The agenda is sorted according to this heuristic and can be of any length. It is different from the Uniform Cost search that uses the actual cost so far. The problem is that we don't take the cost so far into consideration, and that can lead to a sub-optimal solution. If we want the search to be the most greedy, we can use an agenda of size 1, where we only store the best node at each step. Every other node are thrown away. it can lead to a local minimum.

A The A algorithm is a combination of Uniform Cost search and Hill-Climbing search. It take the cost so far (denoted $f(n)$) and the estimated cost remaining heuristic (denoted $h(n)$) into account. Together, it forms the overall utility function $f(n) = g(n) + h(n)$.

A* Like the A algorithm, A* [8] is an informed search algorithm and aim to find the cheapest path from a specific starting node in a graph to the given goal state. At each iteration of its main loop, it determine which of its paths to extend by evaluating the cost of the path and an estimate of the cost required to extend the path all the way to the goal. A*, as A, selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the next node on the path. $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal state. A* ends when the path it chooses to extend is a path that contains the start and the goal state or if there are no eligible path to be extended anymore. The difference with A algorithm is that now, A* uses heuristic which is admissible, meaning that it never overestimates the costs to get to the goal. For example, given a node B, if the estimated cost is X while the real cost to reach the goal state from B is $Y < X$, then the heuristic is inadmissible. A* guarantees to return the optimal solution if the heuristic function is admissible.

To implement that algorithm, we use a priority queue, which is called `open set`, in order to store the repeated selection of minimum cost nodes to expand. This open set can be implemented with a min-heap, a priority queue or a hash set. At each iteration, we remove the node with the lowest $f(n)$ value from the queue and the f and g values of its neighbors are updated. these neighbors are added to the queue. The algorithms terminates whenever a goal node has a lower f value than any node in the queue, or the queue is empty.

The solution found is the shortest path and its cost is the f value of the goal state we end up with since the admissible heuristic h value equals to 0 at the goal (if we take the straight-line distance towards the goal state for instance). In order to get the sequence of actions leading to the shortest path, the algorithm should also keep track

of all precedessors of each node. At the end of the execution, we could go back from the last node to the first node by following the predecessors.

```
function RECONSTRUCT_PATH(predecessor, current)
    full_path ← {current}
    while current in predecessor.keys do
        current ← predecessor[current]
        full_path.add(current)
    end while
    return full_path
end function
```

```
function ASTAR(start, goal, h)                          ▷ h is the heuristic function
    openSet ← {start}
    predecessor                          ▷ predecessor[n] = node preceding the node n
    g                                    ▷ g[n] = cost so far, default values set to Infinity
    g[start] ← 0                         ▷ f[n] = g[n] + h(n) = utility function
    f                                    ▷ default values set to Infinity
    f[start] ← h(start)
    while openSet not empty do
        current ← node in openSet with lowest f value
        if current = goal then
            return reconstruct_path(predecessor, current)
        end if
        openSet.remove(current)
        for each neighbor of current do
            neighbor_g ← g[current] + d(current, neighbor)    ▷ distance from start to neighbor through current
            if neighbor_g < g[neighbor] then
                predecessor[neighbor] ← current
                g[neighbor] ← neighbor_g
                f[neighbor] ← g[neighbor] + h(neighbor)
                if neighbor not in openSet then
                    openSet.add(neighbor)
                end if
            end if
        end for
    end while
    return failure                                  ▷ open set empty, goal not reached
end function
```

An interesting approach is to use a monotone, consistent heuristic. If the heuristic h

satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every arcs (x, y) of the graph, where d is the distance between nodes x and y, then h is considered as consistent. The advantage of this heuristic is that A* guarantees to find an optimal solution without preprocessing any node more than once. This version of A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) - h(x)$. In another point of view, Dijkstra's algorithm, as another example of uniform-cost search algorithm can be viewed as a special case of A* algorithm with a heuristic function $h(x) = 0$. Therefore, Dijkstra's algorithm can be implemented more efficiently by removing $h(x)$ value of each node.

4.2.3 Analysis

In order to evaluate those algorithms, we can assess these following criteria:

- Completness : If a solution exists for a given problem, the algorithm is always able to find a solution
- Time complexity : Time needed to find the best solution in terms of number of nodes generated
- Space complexity : How much memory did the algorithm use (maximum number of nodes in memory at once)
- Optimality : The algorithm is always able to find the least-cost solution.

The time and space complexity are measured according to some information about the search tree : b is the maximum branching factor of the search tree (if the tree is dense or sparse depending on the average number of successors per node), d is the depth of the optimal solution and m is the maximum depth of the state space.

Depth-first search Depth-first search is not complete due to loops. Indeed, it fails to find the solution if there are loops in the graph because it is stuck in a infinite depth search space. Its time complexity is $\mathcal{O}(b^m)$, which is huge if the maximum depth of the state space m is larger than d. In practice, if there are a lot of solutions in a problem, DFS is faster than BFS since it has better chance of finding a solution with a small amount of nodes in the search space. It take $\mathcal{O}(bm)$ in terms of space complexity. It only needs to keep track of one unique path from the root to the leaf node, along with unexpanded neighbors nodes for each node in the path. This algorithm is not optimal. It doesn't guarantee to always find the optimal solution.

breadth-first search Breadth-first search is complete. All nodes are examined if the branching factor b is finite. Its time complexity is exponential in b : $1 + b + b^2 + b^3 + \ldots + b^d = \mathcal{O}(b^d)$. Its space complexity is $\mathcal{O}(b^d)$ since it keeps track of every node in memory and that's a problem if the number of nodes is big. It's optimal only if the cost equals 1.

Uniform-cost search As for the breadth-first search, UCS is complete and guarantees to find a solution if there is one. It is also optimal since it always find the optimal solution as long as the step cost is positive. It runs in $\mathcal{O}(n)$ in the number of nodes with path cost less than the optimal solution. Same for the space complexity. The advantage of this algorithm is that it finds the cheapest one as soon as it reports the first solution it finds. It prevent from exploring the whole tree.

Iterative-deepening search Iterative-deepening search has the same performances as breadth-first search. It is complete, optimal if step cost equals 1, and it runs in $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = \mathcal{O}(b^d)$ time and space. This algorithm benefits from the same advantage as DFS which is the modest memory requirements due to the depth of the path in the graph.

A* A* is complete and guarantees to terminate on finite graphs with non negative edge weights, it will always find a solution if one exists. If the graph is inifinite with a finite branching factor b and edges costs bounded by 0 ($d(x;y) > \epsilon$ 0, for a fixed ϵ), then A* guarantees to terminate if there is a solution. A* is also optimal as long as the heuristic is admissible (it never opverestimates the costs to get to the goal state). The time complexity of A* depends on the chosen heuristic. If the heuristic function is well chosen, it can have a great impact on the performances of the algorithm since it prevent A* from expanding many of the b^d nodes that an uninformed search would have expanded. To measure the efficiency brought by a given heuristic, we can measure the *effective* branching factor b^*, which can be calculated empirically by measuring the number of nodes expanded, N, and the depth of the solution.

$$N + 1 = 1 + b* + (b*)^2 + \ldots + (b*)^d$$

. A good heuristic is characterised by a low *effective* branching factor. The optimal branching factor is 1 ($b^* = 1$). In the worst case, A* runs in exponential time $\mathcal{O}(b^d)$ in the number of nodes in the depth of the solution when the search space is unbounded. If the goal state is not reachable from the starting node and the state space is infinite, then A* will never terminate. If we don't look at the worst case scenario of A* algorithm, its time complexity would be polynomial if the search space is a tree and there is one goal state. The complexity can be expressed using the heuristic function $h : |h(x) - h(x^*)| = \mathcal{O}(\log h^*(x))$, where $h*$ is the optimal heuristic, meaning the exact cost to reach the goal state from a node x. That formula means that the error of the heuristic h will not grow faster than the logarithm of the optimal heuristic h^*. It gives us a bound on the time complexity of the algorithm. Concerning the space complexity, it is similar to the other search algorithms since it stores all generated nodes in memory.

5 Meta-heuristics

5.1 Ant Colony Optimisation

6 Speed-up techniques

6.1 ALT

blabla [5]

6.2 Heuristic search

6.3 Bidirectional search

6.4 Short-cuts and contractions

6.5 Landmarks reaches

7 Conclusion

lol

Bibliography

[1] Michael J. Bannister and David Eppstein. "Randomized Speedup of the Bellman–Ford Algorithm". In: *Computer Science Department, University of California, Irvine* (2011).

[2] Paul E. Black. *Floyd-Warshall algorithm, in Dictionary of Algorithms and Data Structures [online].* URL: `https://www.nist.gov/dads/HTML/floydWarshall.html`. ed. 17 December 2004.

[3] Paul E. Black. *Johnson's algorithm, in Dictionary of Algorithms and Data Structures [online].* URL: `https://www.nist.gov/dads/HTML/johnsonsAlgorithm.html`. ed. 17 December 2004.

[4] Cormen Thomas H. Leiserson Charles E. Rivest Ronald L. Stein Clifford. *Introduction to Algorithms (Second edition).* MIT Press and McGraw Hill, 2001. ISBN: 0-262-03293-7.

[5] Fabian Fuchs. "On Preprocessing the ALT-Algorithm". MA thesis. 2010.

[6] Teofilo F. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics.* CRC Press, 2007. ISBN: 9781420010749.

[7] Stefan Hougardy. "The Floyd-Warshall algorithm on graphs with negative cycles". In: (2010).

[8] Zeng W. Church R. L. "Finding shortest paths on real road networks : the case of A*". In: *International Journal of Geographical Information Science* (2009).

[9] Stuart J. Russell. *Artificial Intelligence: A Modern Approach (2nd ed.)* Upper Saddle River, New Jersey, Prentice Hall, 2003. ISBN: 0-13-790395-2.

[10] Yushi Uno Ryuhei Uehara. "Efficient Algorithms for the Longest Path Problem". In: . (2004).

[11] Leonardo Taccari. "Integer programming formulations for the elementary shortest path problem". In: . (2015). URL: `http://www.optimization-online.org/DB_FILE/2014/09/4560.pdf`.

[12] Ittai Abraham Amos Fiat Andrew V. Goldberg Renato F. Werneck. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms". In: . (2010). URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/2010/01/soda10.pdf`.

[13] Ittai Abraham Daniel Delling Andrew V. Goldberg Renato F. Werneck. "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks". In: . (2010).