

Advanced Algorithms Swiss Knife



Gabriel Rovesti

Table of Contents

[2 Graph – General Definitions 3](#_Toc163550045)

[3 Depth First Search - DFS 5](#_Toc163550046)

[3.1 Description 5](#_Toc163550047)

[3.2 Algorithm 5](#_Toc163550048)

[3.3 Complexity 5](#_Toc163550049)

[3.4 Applications 6](#_Toc163550050)

[4 Breadth First Search - BFS 7](#_Toc163550051)

[4.1 Description 7](#_Toc163550052)

[4.2 Algorithm 7](#_Toc163550053)

[4.3 Complexity 7](#_Toc163550054)

[4.4 Applications 8](#_Toc163550055)

[5 Minimum Spanning Tree – MST 9](#_Toc163550056)

[5.1 Generic Greedy Algorithm 9](#_Toc163550057)

[5.2 Definitions 9](#_Toc163550058)

[6 Prim’s Algorithm 10](#_Toc163550059)

[6.1 Description 10](#_Toc163550060)

[6.2 Algorithm 10](#_Toc163550061)

[6.3 Complexity 10](#_Toc163550062)

[7 Efficient Prim – Heap Implementation 11](#_Toc163550063)

[7.1 Description 11](#_Toc163550064)

[7.2 Algorithm 11](#_Toc163550065)

[7.3 Complexity 12](#_Toc163550066)

[8 Kruskal’s Algorithm 13](#_Toc163550067)

[8.1 Description 13](#_Toc163550068)

[8.2 Algorithm 13](#_Toc163550069)

[8.3 Complexity 13](#_Toc163550070)

[9 Efficient Kruskal – Union-Find 14](#_Toc163550071)

[9.1 Description 14](#_Toc163550072)

[9.2 Algorithm 14](#_Toc163550073)

[9.3 Complexity 14](#_Toc163550074)

[10 Shortest Path 15](#_Toc163550075)

[11 Single-Source Shortest Path (SSSP) 15](#_Toc163550076)

[12 Non-negative weights – Dijkstra 16](#_Toc163550077)

[12.1 Description 16](#_Toc163550078)

[12.2 Algorithm 16](#_Toc163550079)

[12.3 Complexity 16](#_Toc163550080)

[13 Efficient Dijkstra – Heap 17](#_Toc163550081)

[13.1 Description 17](#_Toc163550082)

[13.2 Algorithm 17](#_Toc163550083)

[13.3 Complexity 17](#_Toc163550084)

[14 General Case: SSSP Problem 18](#_Toc163550085)

[15 Bellman-Ford’s Algorithm 19](#_Toc163550086)

[15.1 Description 19](#_Toc163550087)

[15.2 Algorithm 19](#_Toc163550088)

[15.3 Complexity 19](#_Toc163550089)

[16 All-Pairs Shortest Paths (APSP) 20](#_Toc163550090)

[17 Floyd-Warshall’s Algorithm 21](#_Toc163550091)

[17.1 Description 21](#_Toc163550092)

[17.2 Algorithm 21](#_Toc163550093)

[17.3 Complexity 21](#_Toc163550094)

[18 Maximum Flows 22](#_Toc163550095)

[19 Ford-Fulkerson’s Algorithm 23](#_Toc163550096)

[19.1 Description 23](#_Toc163550097)

[19.2 Algorithm 24](#_Toc163550098)

[19.3 Complexity 24](#_Toc163550099)

[20 NP-Hardness 25](#_Toc163550100)

# Graph – General Definitions

* as the graph itself
  + = set of vertices (aka nodes)
  + (cartesian product = all) is a collection of edges
    - an edge is a pair of vertices
      * it indicates the connection between two nodes
      * a connection of vertices allows for repetition
* directed graphs, which happens if
* undirected graphs, which happens if
* arc = edge inside directed graphs (also called *directed edges*)
* given an edge
  + is incident on and (happens if vertex if one of endpoints in that edge)
  + and are adjacent (there is an edge between the two vertices)
* neighbors of a vertex: all vertices s.t.
  + all vertices directly connected to a given vertex by an edge
* degree of a vertex , denoted as or
  + the number of edges incident on
* path: and
  + finite/infinite sequence of nodes which joins a sequence of vertices via edges
* simple path: (all vertices) are all distinct
  + same definition as above and vertices/nodes are all distinct/so are the edges
  + e.g., has repeated twice so it’s not simple
* cycle: simple path s.t. (starts from a given vertex/ends at same node)
* subgraph:
  + the edges of are incident only on vertices of
  + in words: it is a subset of the larger original graph
* spanning subgraph: a subgraph with
  + a subgraph which “spans” the original graph (so there are all the vertices)
  + following other definitions
    - subgraph obtained by edge deletions only but retaining all vertices
    - so it’s a subgraph of with same vertex set as
* connected graph: if a path from to
* connected components: a partition of in subgraphs
  + is connected
  + there is no edge between and
* tree: connected graph without cycles
  + any two vertices are connected by *exactly* one path
* forest: set of trees (disjoint)
  + also = undirected graph in which any two vertices are connected by *at most* one path
* spanning tree: a spanning subgraph connected and without cycles
* spanning forest: a spanning subgraph without cycles

Generally, remember:

* (number of nodes)
* (number of edges)
* the size of a graph is

There are also multiple ways of representing:

* an adjacency list
  + an array of lists, one vertex (consider the example below)
  + each containing all the vertices adjacent to (represented by table below)

|  |  |
| --- | --- |
| 1 | 2,5 |
| 2 | 1,3,4,5 |
| 3 | 2,4 |
| 4 | 2,5,3 |
| 5 | 4,1,2 |

Immagine che contiene cerchio, linea, diagramma, schizzo

Descrizione generata automaticamente

What if directed? Only vertices pointed for that vertex.

* Pro: space usage i.e. linear
* Con: no quick way to determine if a given edge is in the graph
* an adjacency matrix
  + a matrix s.t. if , otherwise



Immagine che contiene schermata, numero, Carattere, diagramma

Descrizione generata automaticamente



* If graph is directed 🡪 the matrix is *asymmetric*
* If graph is undirected 🡪 the matrix is *symmetric*

In case of a *weighted graph*, each cell of the matrix has either the value of the edge weight (as number) or to represent null costs. This kind of graph represents costs, capacities, etc.

* Pro: Quick to determine if a given edge is present
* Con: Space required is 🡪 can be superlinear in the input size
  + if number of vertices increases, the space required by matrix grows quadratically

# Depth First Search - DFS

## Description

The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It may find:

* new edges (discovery edge)
* non-tree edges, linked to ancestors (back edges)

Visit a vertex, then a neighbor of the vertex, then a neighbor of the neighbor – these are all neighbours, classified with adjacency lists.

## Algorithm

## Complexity

Given:

* : number of vertices of (one invocation )
* : number of edges of (costs related to node, excluding recursive invocations inside)

The complexity overall is:

More in general: – vertices and edges

## Applications

There are several:

* path (between two generic vertices)
  + done adding a field
* finding cycles
  + use field on vertices and on edges
* find connected components
  + run the algorithm times
  + consider all untouched vertices
  + see which have back edges, meaning they “close” the cycle
  + otherwise, return
* find a spanning tree

# Breadth First Search - BFS

## Description

The algorithm is iterative, starts from a source vertex and visits all vertices connected to a specific component, partitioning them in levels according to their distance. It still has discovery edges:

* but adds cross edges – which connect vertices at different levels

## Algorithm

## Complexity

## Applications

* Same as for DFS in time

So, again:

* path (between two generic vertices)
  + done adding a field
* finding cycles
  + use field on vertices and on edges
* find connected components
  + run the algorithm times
  + consider all untouched vertices
  + see which have back edges, meaning they “close” the cycle
  + otherwise, return
* find a spanning tree

# Minimum Spanning Tree – MST

* Input: a graph undirected, connected and *weighted*
  + A weight
  + defines cost of edge
* Output: a spanning tree of s.t. is minimized
  + Goal is minimizing the sum for all weights of every edge of the tree

## Generic Greedy Algorithm

## Definitions

* A cutof graph is a partition of
  + in words, a partition of vertices into two disjoint subsets
  + it can be done on one or more edges
* An edge crosses a cut if and (or viceversa)
  + so, if its endpoints lie in different subsets of the partition defined by the cut
* A cut respects a set of edges if no edge of *crosses* the cut
* Given a cut, an edge that crosses the cut and is of minimum weight is called light edge(for that cut) 🡪 they are useful, because when included in MSTs, they have minimum weight

There is also the *minimum cut*, for which we have , where is a generic size of graph. Summing up all vertices, we obtain , concluding it’s .

# Prim’s Algorithm

## Description

The algorithm is iterative and selects light edges at every step, growing a spanning tree from there. Consider this [gif](https://en.wikipedia.org/wiki/Prim%27s_algorithm#/media/File:PrimAlgDemo.gif) to see the running. We have to preserve “safe edge” property – take only minimal edges not already inside of the tree.

## Algorithm

## Complexity

# Efficient Prim – Heap Implementation

## Description

The previous is not so efficient in large structures. The right kind of data structure to improve the algorithm is a *priority queue*, implemented with a heap.

* Recap about this data structure
  + 🡪 add an object to the heap (possibly fast)
  + 🡪 remove an object with the smallest key (highest priority)
  + 🡪 given a pointer to an object, remove it
* In a heap with objects, the complexity of these operations is

We can redefine the algorithm exploiting this efficient data structure basically with the same principle:

* consider a min heap starting from whatever vertex, which is the root
* from there, always extract the minimum value (means checking if it is min heap),
  + then update the path

## Algorithm

## Complexity

* 🡪
* 🡪 iterations
* 🡪

Total cost of only operations:

* *for loop*: executed times in total (every vertex is explored)
  + - this here is a simple check
    - two operations

Total cost of *for loop*: (iterating for all adjacent nodes, quantity equal to node degree)

This way, the total complexity of the algorithm is (since is connected, we recall) 🡪 near-linear time.

# Kruskal’s Algorithm

## Description

It picks the minimum weighted edge at first and the maximum weighted edge at last. It sorts edges by weight and then adds them continuously, preserving the “safe edge” property – take only the unexplored. It does so preventing the adding of cycles.

## Algorithm

## Complexity

* sorting:
* for loop: check whether closes a cycle is equivalent to check whether contains an path 🡪 DFS on 🡪 complexity:

Total: 🡪

# Efficient Kruskal – Union-Find

## Description

It can be implemented as fast as Prim’s, considering the most frequent operation here is cycle check (equivalently, path check), which happens when an edge is added to .

We create a new data structure supporting this operation fast and to do that, we use a structure called Union-Find (also called *disjoint set*). This is a structure to merge *disjoint sets* (also non-overlapping in their elements) of objects and supports at least three operations:

* *Init*: given an array of objects
  + it creates a Union-Find data structure with each object in its own set
* *Find*: given an object , return the name of the set that contains
  + depth: number of edges traversed by *Find*
* *Union*: given two objects merge the sets that contain and into a single set
  + done whenever the sets are distinct
  + if are already in the same set, this operation does nothing

## Algorithm

## Complexity

* Init:
* Sorting:
* Find:
* Union: 🡪 only when I go inside an “if” and when the edge is added
* updating:

In total:

# Shortest Path

* Given a weighted graph, the length of a path is defined as
* A shortest path from a vertex to a vertex is a path with minimum length among all paths
* The distance between vertices and , denoted as is the length of a shortest path from to ; if there is no path at all from to then

The problem itself is the following:

* Given a directed, weighted graph and a source vertex and a destination , compute the shortest path from to

# Single-Source Shortest Path (SSSP)

* input: a directed, weighted graph with edge weights and a source vertex
* output:
  + shortest path to all destinations

There are two major cases to solve: a special one and a more general one.

# Non-negative weights – Dijkstra

## Description

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices. It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.

* input: directed ,
* output:
  + with coming as shorthand form of the previous one

## Algorithm

## Complexity

# Efficient Dijkstra – Heap

## Description

Normal implementation uses adjacency list. This implementation improves efficiency by using a priority queue (usually implemented as a binary heap) to select the vertex with the smallest tentative distance efficiently. The implementation is almost identical to Prim with heaps.

## Algorithm

(almost identical to Prim’s implementation with heaps)

## Complexity

* considering graph as adjacency list, vertices and edges
* iterations because of heap usage

Total number of operations: (there are operations on heaps)

# General Case: SSSP Problem

We reformulate the previous problem a bit:

* Input: a directed weighted graph and a source vertex
* Output: one of the following
  + vertex
  + a declaration that contains a negative cycle

Need to forbid negative cycles in shortest paths, they lead to infinitely small paths, which is an NP-Hard problem.

The main Dijkstra problems are two:

* It never revisits/updates its decisions, but it should for all vertices!
  + Once a vertex is marked as “closed”, we will never develop this node again
  + If we have a vertex in open such that its cost is minimal - by adding any positive number to any vertex - the minimality will never change
  + Without the constraint on positive numbers - the above assumption is not true
  + It assumes them to be positive to make the algorithm run faster and does this to avoid considering paths which can’t be shorter
* should be an *estimated distance*, which needs to be updated for every vertex
  + how many times? edges times should be enough
  + maximum number of edges in a simple path between any two vertices

# Bellman-Ford’s Algorithm

## Description

* Input: A directed graph with edge weights and a source vertex
* Output: Either or a declaration that contains a negative cycle

The algorithms is used when the graph might possess negative weights and can even detect negative cycles. If the graph contains one, there is no cheapest path, instead one can make it cheaper by one more walk around said negative cycle (in iterations it reaches a fix-point, if it doesn’t it means a negative cycle exists). Still, it’s slower compared to Dijkstra.

## Algorithm

## Complexity

# All-Pairs Shortest Paths (APSP)

* Input: A directed, weighted graph
* Output: One of the following:
  + ordered vertex pair
  + a declaration that contains a negative cycle
    - this can be problematic in finding a shortest path
    - now we would have to output shortest paths

Consider:

* If we use Bellman-Ford - very high complexity 🡪
  + Using dynamic programming, the complexity can be reduced to
  + This holds rewriting B-F recurrence controlling the allowable size of the input

# Floyd-Warshall’s Algorithm

## Description

It’s used to find the shortest paths between all pairs of nodes in a weighted graph, with positive or negative edges.

* instead of restricting the number of edges allowed in a solution, restrict the identities of the vertices that are allowed in a solution
  + in other words, now paths can pass through only certain vertices
* Basically, it compares many possible paths through the graph between each pair of vertices

It iterates on vertices: i n 3 nested loops, testing whether using 𝑘 in the path is better.

## Algorithm

## Complexity

# Maximum Flows

* a flow network is a directed graph where each edge has a capacity , along with a designated source and sink
  + for convenience, write if , no edges enter and no edges leave
* a flow is a function satisfying the following constraints (how much stuff I send through the edges in general)
  + (*capacity*) – value of the flow at most capacity of that edge
  + (*conservation*) we have
  + the amount of flow going in nodes must be equal to the flow going out from those (conservation of flows)
    - initially, such flow is 0, which is “how much we can pass on the edge”
* the value of a flow is
  + basically, the sum of all flows going in and out vertices thanks to edges
  + as a matter of fact, the amount of stuff traveling from source to sink
  + such flow has to be less than or equal to the capacity

As for the problem itself:

* given a flow network, find a flow of maximum value. Such flow is measured on *the maximum value received in a sink node*

# Ford-Fulkerson’s Algorithm

## Description

Given a flow network a flow , the residual network of w.r.t (with respect to) flow , , is a network with vertex set and with edge set as follows:

* for every edge in
  + if , add to with capacity
  + if , add another edge to with capacity

The Ford-Fulkerson (F-F) algorithm repeatedly finds an path in (e.g., using BFS) and uses to increase the current flow.

* is called augmenting path
  + This is a path of edges in the residual graph with unused capacity greater than from the source to the sink
  + This can only flow on edges not fully saturated yet
* In an augmenting path, the *bottleneck* is the smallest edge on the path. We can use this one to augment the flow along the path

Immagine che contiene cerchio, schermata, linea, diagramma

Descrizione generata automaticamenteIn figure below, in orange the augmenting path, in light-blue as written the bottleneck:

* Augmenting the flow means updating the flow values along the augmenting path (left)
  + For forward edges, this means increasing the flow by the bottleneck value
* When augmenting the flow along the augmenting path
  + you also need to decrease the flow along each residual edge (backward edges) by the bottleneck value (right)
  + Immagine che contiene schermata, cerchio, linea, diagramma

    Descrizione generata automaticamenteresidual edges exist to “undo” bad augmenting paths which do not lead to a maximum flow

Immagine che contiene schermata, cerchio, linea, diagramma

Descrizione generata automaticamente

The residual graph, so, contains also residual edges. This algorithm continues to find augmenting paths and augments the flow until no more augmenting paths exist.

* The algorithm simply takes in every iteration the bottleneck
* Then considers the bottleneck and keeps incrementing selecting every possible path until max flow is reached
* Each iteration is then reported into the residual graph, accounting for the bottleneck
  + e.g. if we chose in an iteration, in the next iteration
    - forward (remaining)
    - backward (spent)

## Algorithm

## Complexity

* Assume capacities are integers; then
  + the flow value increases by is each iteration
  + the complexity of each iteration is
* Total complexity is , where is a max flow

# NP-Hardness

There are problems which can be solved in linear time:

* e.g. Eulerian circuit – a graph where edges are traversed all at once

There are problems which can be solved in polynomial time:

* e.g. Minimum Spanning Tree (MST), minimizing the weights inside all tree

There are also problems where no polynomial algorithms are present to solve the problem:

* e.g. Traveling Salesperson Problem (TSP), Hamiltonian Circuit

We define the following *complexity classes*:

1. is the set of decision problems that can be solved in polynomial time
2. is the set of decision problems with the following property:
   1. if the answer is YES, then there is a proof of this fact (called “certificate”) that can be checked in polynomial time
3. , which is essentially the opposite of :
   1. property: if the answer is NO, then there is a proof of this fact that can be checked in polynomial time

Other features of problems:

* a problem is said to be NP-Hard if a polynomial time algorithm for this one would imply the existence of a polynomial time algorithm for every problem in NP
* More formally, a problem is NP-Hard if every problem in NP reduces in polynomial time to it.
  1. unless , which is not yet solved
  2. if a problem is NP-Hard, it provides evidence the problem may not be in
* a problem is NP-Complete if it’s both in NP and NP-Hard
  1. e.g. the Cook-Levin Theorem for Boolean Satisfiability problem (SAT)
     1. made up of clauses with conjunction/disjunction, usually 3 (3-SAT)

We use a reduction as a very powerful tool:

* a reduction is an algorithm for transforming one problem into one another
* a problem reduces to if there is an algorithm able to solve can be translated into one which solves
* Remember reductions works from (problem I know to be hard) to (new problem)

There are other problems we can solve:

* Independent Set
  1. given a graph an independent set in is a subset with no edges between them
* (Maximum) Independent Set (this one will be referred to as simply “Independent Set” meaning the latter)
  1. compute an independent set of maximum size

Examples of reductions:

* Using Hamiltonian circuit to solve TSP 🡪
  1. If we had a fast algorithm for TSP, we would also solve Hamiltonian problem
* Using 3SAT to solve Independent Set 🡪 
  1. If we had a fast algorithm for Independent Set, we would also solve 3SAT