

Advanced Algorithms Simple (for real)



Gabriel Rovesti

Table of Contents

[2 Course Presentation 3](#_Toc160528651)

[3 Graphs 5](#_Toc160528652)

[3.1 Terminology and Concepts 5](#_Toc160528653)

[3.2 Basic Problems, Notations and Properties 7](#_Toc160528654)

[3.2.1 Homeworks 8](#_Toc160528655)

[3.3 Graphs Representation 9](#_Toc160528656)

[3.4 Graphs Algorithms 10](#_Toc160528657)

[3.4.1 Depth-First Search (DFS) 11](#_Toc160528658)

[3.4.2 Breadth-First Search (BFS) 16](#_Toc160528659)

**Disclaimer**

# Course Presentation

(Usual general fluff, then the lectures will be present. This is the only slides-based part, found [here](https://stem.elearning.unipd.it/pluginfile.php/625081/course/section/70344/Advanced%20Algorithms%20-%20Spring%202024.pdf))

Algorithms have a general motivation: create a network of knowledge and allow, with pacing of times, different development and stories creation, while crafting new solutions. We might define them as a sequence of steps to solve the most mundane problems but also really complex ones.

There are different kinds of *applications*:

* network routing
* bioinformatics
* economics (e.g., game theory)
* fluid dynamics
* data mining
* cryptography
* machine learning

The point is this: even when making interviews, algorithms are both the logic and the solution to current problems, thinking *repeatedly and abstractly in a concrete (and fast) way*. Historically, there are still a lot of unsolved or still not found problems. That’s why the course is *mandatory*.

There are also different *goals*, wanting to introduce advanced principles of algorithm design and analysis. In particular, you’ll learn how to:

* Design algorithms for complex domains such as *graphs*
* Recognize “hard” problems and address them using *approximation algorithms*
* Use the power of *randomness* to design fast algorithms
  + and analyze them with appropriate mathematical tools

The *contents* of the course will be the following:

* (Basic) Graph algorithms
  + Graph search and its applications, minimum spanning trees, shortest paths, maximum flows 2 Approximation algorithms
* *Intractable* problems (not solvable in a reasonable amount of time)
  + NP-hardness and reductions between problems
  + Approximation algorithms for intractable problems
    - such as vertex cover, set cover, and the traveling salesperson problem
* Randomized algorithms
  + Main design techniques and analysis tools
    - with applications to problems such as sorting and minimum cuts

Although there are no formal prerequisites, an undergraduate course in algorithms and a good knowledge of (discrete) probability are assumed. Specifically, you should be familiar with:

* *Algorithm design techniques*: divide and conquer, greedy, dynamic programming
* *Data structures*: lists, stacks, queues, binary trees, search trees, heaps
* *Probability*: basic notions, discrete random variables

We want to discuss the *intuition* behind formulating algorithms and distill the core ideas making the algorithms work. Because we are computer scientists, we want to give *rigorousness*: algorithms without proofs are just conjecture and proofs give math logic to those.

We’ll follow, in part, an “active learning” approach:

* Will foster and encourage interaction during class
* Will frequently conclude class with 1-2 exercises
  + whose solution will be shown only at the beginning of next class
* Will frequently post on Moodle further readings
  + news/surveys/research articles/videos related to the topics covered in class
* There is no lab or coding assignments
  + but you are encouraged to code your favorite algorithms up and run them on real data

If you read until here, you sure wanna know: how is the exam?

* Written test, 2 hours. It consists of:
  + 3 questions
    - theory questions on the topics covered in class
    - aimed at verifying the student’s knowledge of the contents of the course
  + 2 problems
    - problems whose solution *requires some creativity*
    - aimed at verifying the student’s ability to use concepts
    - techniques learned during the course to solve new problems

# Graphs

(Suggested readings: The Algorithm, idiom of modern science [[here](https://www.cs.princeton.edu/~chazelle/pubs/algorithm.html)])

A graph is a repartition of the relationships between pairs of objects. In particular, we note:

* 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

In the following drawings, we find:

* directed graphs, which happens if
* undirected graphs, which happens if
* Immagine che contiene calligrafia, schizzo, Line art, disegno

  Descrizione generata automaticamenteImmagine che contiene calligrafia, Carattere, testo, schizzo

  Descrizione generata automaticamentearc = edge inside directed graphs (also called *directed edges*)

In this case, we’ll (mostly) use simple graphs, meaning:

* no parallel edges
* no self-loops

## Terminology and Concepts

We give some *terminology*:

* 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

In many ways, graphs are the main modality of data we receive from nature and here we give some *examples*:

* Road networks 🡪 (cities, roads)
* Computer networks 🡪 (computers, computers)
* World Wide Web (WWW) 🡪 (webpages, hyperlinks)
* Social networks (people, friendships relationships)
* Biological networks
  + e.g., molecules (atoms, chemical bonds)
  + e.g., brain (neurons, synapses)
* Finance 🡪 (accounts, transactions)

We give some concepts also:

* 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
* Immagine che contiene diagramma, linea

  Descrizione generata automaticamenteconnected components: a partition of in subgraphs
  + is connected
  + there is no edge between and

Immagine che contiene calligrafia, Carattere, bianco, testo

Descrizione generata automaticamente

Immagine che contiene cerchio, linea

Descrizione generata automaticamente

* tree: connected graph without cycles
  + any two vertices are connected by *exactly* one path

Immagine che contiene cerchio, linea, bianco, appendiabiti

Descrizione generata automaticamenteThere is also the concept of *rooted tree*:

* there is a root
* there is a father for each non-root node and each node is directly linked to the father
* going father to father, we reach

Continuing with definitions:

* forest: set of trees (disjoint)
  + also = undirected graph in which any two vertices are connected by *at most* one path
* Immagine che contiene schizzo, calligrafia, disegno, Carattere

  Descrizione generata automaticamentespanning tree: a spanning subgraph connected and without cycles
* Immagine che contiene schizzo, cerchio, design

  Descrizione generata automaticamentespanning forest: a spanning subgraph without cycles

## Basic Problems, Notations and Properties

There are different *basic problems*:

* Traversal (systematic exploring of graph e.g., crawling)
* Connectivity (tell if the graph is connected or not)
* Computing connected components (e.g., wireless networks)
* Spanning trees (e.g., efficient broadcasting in wireless networks)
* Minimum-weight spanning trees (e.g., navigator)
* Shortest paths (e.g., social media friend analysis)

Also consider some notations and properties:

* (number of nodes)
* (number of edges)
* the size of this graph is
  + is not enough (normally online you would find the size it’s = count of edges)
  + consider a scenario of a graph with vertices and no edges
  + the size of graph would be but we don’t consider vertices
    - we are accounting for both the “space” and the “connections” occupied

### Homeworks

Exercise (Properties of graphs)

*Let be a simple, connected graph with vertices and edges. Then:*

1. *is a tree*
2. *is connected*
3. *is acyclic (i.e., is a forest)*

*Prove the previous properties*.

My solution

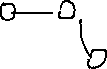
Consider an example for each one:

1. . The degree is the number of vertices incident in . The example clarifies it.



This happens because inside the summation, every edge is counted twice.

1. Consider vertices. This happens because we are choosing vertices out of to form an edge. In a simple graph, order does not matter, so we can select any two vertices, then arranging with all the possible arrangements and avoid counting each pair twice. Indeed, in a simple graph there are possible pairs of vertices.
2. If is a tree (=connected graph without cycles), consider and, you would have:



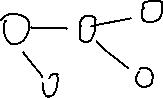
Fix a root. Then, represents father-child relationships, which are .

1. If is connected we would have



This happens because is a tree that may have cycles, thus it can only have more edges.

1. Consider an acyclic graph, then



This happens because is a tree that may not be connected, thus it can only have less edges.

Official solution

1. In the summation, every edge is counted exactly twice
2. In a simple graph, there are possible pairs of vertices
3. Fix a root on a vertex (so, consider as rooted tree, thanks to the equivalence between rooted tree and “free” tree). Then represent father-child relationships, which are (which means each non-root node has a unique father)
4. is a tree that may have cycles it can only have more edges than a tree
   1. Consider connectivity removes edges and keeps the graph connected without cycles, thanks to edges
5. is a tree that may not be connected it can only have less edges than a tree
   1. If it is a tree without cycles, it is a forest, and its maximum edges are

## Graphs Representation

How to encode a graph for use in an algorithm?

Consider a list of vertices and a list of edges . (they contain all information about and and the links between each other). Let’s consider vertices are called . This is useful but does not allow for fast algorithms overall.

To allow for *direct access to edges*, one of the following data structures are used in addition to pointers to , .

* 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*
  + edges are bidirectional 🡪 only half of matrix needs to be stored
  + operations here are more efficient in general

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

It may also depend on the number of edges:

* *dense* graph = number of edges close to maximal number
  + many cells inside adjacency matrix will be populated by non-zero values
  + adjacency matrix is mostly used here
    - allows to quickly test the presence of an edge and check its info
* *sparse* graph = number of edges with only a few edges
  + conversely, majority of values will be zero

## Graphs Algorithms

We are focusing over *graph search and its applications*, in particular traversal/exploration. They provide a systematic way to explore a graph starting from a vertex ( source vertex) visiting all the vertices (starting from a graph and a source vertex). The most famous algorithms are:

* *Depth-First Search (DFS)* 🡪 aggressive, goes in depth, then comes back and so forth
* *Breadth-First Search (BFS)* 🡪 non-aggressive, proceeding by levels inside graph

Immagine che contiene schizzo, disegno, cerchio, clipart

Descrizione generata automaticamenteImmagine che contiene schizzo, diagramma, cerchio, disegno

Descrizione generata automaticamenteIn particular, consider the following graphs; in each, the types of visits are defined already in color.



Observe that:

* DFS and BFS serve as design patterns, acting as building blocks
  + where the visit operation can be instantiated to solve specific problems
  + such as connectivity and spanning tree identification
* Traversing and lists also achieves complete graph exploration
  + However, the lack of systematic exploration makes it less useful for problem-solving
* The fundamental idea behind both algorithms is to prioritize visiting neighbors with lower IDs from the starting vertex

### Depth-First Search (DFS)

(Further readings for this one: [paper](https://stem.elearning.unipd.it/pluginfile.php/749331/mod_folder/intro/Depth-First%20Search%20and%20Linear%20Graph%20Algorithms%20-%20Tarjan%20%281972%29.pdf) and [survey](https://arxiv.org/abs/2201.07197))

This is a recursive algorithm which:

* starting from a source “visits” all vertices of the connected component containing
* uses adjacency list as graph representation
* every vertex has a field which can be either
  + if visited
  + otherwise
* every edge has a label which can be either
  + initially
  + or
    - first label indicates an edge which allows discovery of vertices
    - second label indicates non-tree edges
      * that go from a node in the DFS tree
      * to some ancestor of in the DFS tree
      * this is useful in order to find cycles

Consider the following procedure:

(*first invoke: )*



Because I like people understanding stuff, let’s comment human-like this code, considering we:

* take each vertex and we see if it was visited or not
  + this is done *on the connected component* touching all vertices and edges
  + we use adjacency lists to induce an order of visit in neighbors
* check if the current vertex (with ID field) has been visited or not
* loop on all edges incident to current vertex
* check if label of current edge was not labeled = it was not explored
* consider the opposite vertex = other endpoint of the edge
* if that opposite vertex has not been visited yet
  + edge leads to an unexplored vertex, indicating a discovery edge
    - a vertex is discoverable if there exists a path between and not visited
  + this will be labeled, indicating it’s the first time the edge is being traversed
  + then we recursively call the algorithm to explore the connected component
* else (aka it was already visited)
  + the edge is leading to an already explored vertex
  + the edge is labeled indicating a connection back to the ancestor

Immagine che contiene diagramma, cerchio, linea, disegno

Descrizione generata automaticamenteThe following is an example of the algorithm being applied:

#### Correctness

At the end of the algorithm:

1. all the vertices of have been visited and all the edges in are labelled either
2. the set of is a spanning tree of called “DFS tree”

*Proof*:

1. (short: by construction)

By contradiction, not visited. Since is connected, there is a path from to

. Let be the first unvisited vertex in the path (

We run into the contradiction: must have been executed and therefore is called (meaning was found not visited). This happens in contradiction to the hypothesis (it’s not possible to find a vertex unvisited and marked as such)

A vertex is visited only when is invoked DFS is called all incident edges on are labelled, by construction.

1. DFS is called , once, and , a vertex s.t. and is labelled and is invoked from . We say that gets “discovered” by and let’s call “father” of
   1. father (there exists a father and it is unique)
   2. going back father to father eventually is reached

Then, the set of is a *rooted tree* that *touches* all the vertices of and it’s a spanning tree of (unique path from every vertex to the source one and each is discovered by exactly one parent vertex).

#### Complexity

Given:

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

The complexity overall is:

(sum of degrees of each node proportional to number of edges)

Note that is connected, so:

* (connected, so for vertices we would have at least edges)
* (n. of edges at least proportional to n. of vertices)

#### Extension

The possible extension is to visit all the graph (aka: all components even if not connected):

Overall, the complexity if because it scans over all the vertices and nodes.

#### Homeworks

1. *Given a graph and two vertices determine, if it exists, a path from to .*
2. *Given a graph return a cycle (if any)*

My solution

1)

2)

Official solution

* 1st exercise ( path)
  + add a field .
  + Modify s.t. when a is labeled
    - then
  + Run . Check if has been visited
    - NO: then return “No path”
    - YES: starting from , follow the “parent” label, so as to build a path from to
  + Complexity: where is the number of edges of connected component
* 2nd exercise (cycle) 🡪 we go back thanks to back edges because they “close” the cycles
  + add a field and add a field
  + is a then
  + is a then
    - then is an ancestor of in the DFS tree
  + Run DFS on each connected component
  + Check all the edges
    - as soon as an edge is found as
    - and
    - then return a cycle adding to all the edges found in the path from to
    - if no is found, then return “No Cycles” (it would be a tree)

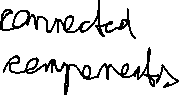
Complexity for both algorithms: 🡪 invoked DFS once for each connected component

#### More applications

More or less what we did until now with DFS was returning a spanning tree. Other problems which can be solved are the following:

* *graph connectivity*: return whether the graph is connected or not
* *connected components*: return a labeling of all the vertices of s.t. 2 vertices have the same label if and only if they are in the same connected component

We will modify the algorithm in such a way that  ~~(~~let’s generalize it, so) (integer, label of the k-th component)



Complexity of the whole thing:

#### Summary

Given a graph , the following problems can be solved in using DFS:

* test if is connected
* find the connected components of
* find a spanning tree of (if is connected – otherwise it’s called spanning forest)
* find a path between two vertices (if any)
* find a cycle (if any)

Possible questions for exam: Show how to find a spanning tree in linear time/Find if graph is connected in linear time, something like that.

### Breadth-First Search (BFS)

This is an iterative algorithm that starting from a source vertex “visits” all the vertices in the same connected component of , and partitioning the vertices in levels depending on their distance from (with distance we mean the shortest path). We’ll use adjacency list to represent :

* if not visited, if visited
* if has no label,
  + connect vertices at different levels (different labels)

An explanation step-by-step of the algorithm:

* visit the source vertex
* iterate over all levels
* explore neighbors starting from level
* create a set of vertices for the next level
* for all incident edges
  + if vertex has not been visited
  + we get the opposite vertex
  + if vertex has not been visited yet
    - mark edge as
    - visit vertex
    - mark it as visited
    - add the vertex to the set of vertices for the next level
  + else (if vertex has already been visited)
    - node represents crossing between different levels
    - it will be marked as
    - it connects two nodes that don’t share any ancestor-descendant relation
  + increment the level counter
  + algorithm terminates where there are no more vertices to visit

Here we do the following example (at first invoke: :

Immagine che contiene disegno, schizzo, Arte bambini, diagramma

Descrizione generata automaticamenteImmagine che contiene cerchio, diagramma, schizzo, linea

Descrizione generata automaticamente

The algorithm, when executed, it will behave like the following:

* + takes source vertex and marks it as visited
  + goes to next level and for each vertex takes its connected component (incident edge)
  + so, from goes to because it’s connected
    - given it was not explored yet, also jumps to
  + same thing for and
  + they were the connected components of and , jumping to the opposite edge
  + from we see a connected component, found within next level of adjacency list
  + given they were already explored, between two are discovered

#### Correctness

At the end of we have:

1. all vertices in are visited and all edges are labelled
2. the set of are a spanning tree (tree touching all the vertices) of
   1. analogously to DFS, we call it BFS tree (this is rooted in )
3. the path in from to has edges and every other path from to has exactly edges (e.g., edges)

Proof of (1) and (2): same as for the DFS

Proof of (3): let where is “discovered” from is a is a path of . By contradiction, assume a path : with (shorter). This implies that … or or … (might be on some levels before). This means that, since but this is a contradiction.

(in words: what you have is that the first node is in level zero, so on and so forth until the t-th node is in some list between and . If this were true, then we have that , because we have that . But this is absurd since we assumed that )

#### Complexity and applications

there is one iteration of the first for loop and iterations of the second for loop.

Complexity: (which becomes if is connected; in general, each for execution and access to lists

Some *applications*:

1. same as for DFS in time
2. given a graph and return the *shortest* path (in term of least amount of edges between two nodes) from to (if any)
   1. modify s.t. when is labeled then
   2. run and return the set of child-parent edges
   3. Complexity:

*Solutions*:

1)

Basically, if we explore all vertices and incident edges we would get a complexity of . This is infact what we are doing here.

2)

Explanation of this algorithm:

* starting from source vertex
* visit it and mark it as such
* create a list of vertices containing the source one
* for all vertices of list and for all incident edges to that vertex
* if vertex was not labelled yet
  + it is marked as
  + parent of becomes and it is now possible to go up
  + it is marked as visited
  + it’s inserted into the list
* otherwise, we mark it as cross edge

Complexity: where is the number of edges of connected component .