Combinatorial Optimization Project

Pierre-Louis Gstalter

July 6, 2022

Contents

1 Introduction

This report is part of the evaluation of the class "Combinatorial Optimization" lectured by professor Giovanni Righini at the University of Milan. The course tackles the interest of combinatorial optimization, which is to solve optimization problems that have a finite amount of issues. Though finite, the dataset of solutions can be large, and iterating over all the solutions is not always possible. The complexity can make some problems unsolvable. That is where heuristic algorithms can help. Hence, I chose to study that course as well during my Erasmus+ exchange in Milan. I believe the two work hand in hand. This project will present two classic algorithms.

2 The Project

2.1 Project Overview

During the Combinatorial Optimization class, many algorithms were presented to us in a thorough and complex way. I wanted to get down to coding to really understand how some of them worked. It was a challenge for me: I had coded in C++, but mainly knew my way around Python. I chose to implement two algorithms, using a mix of C and C++. The first one is Dijsktra's algorithm. Though not the most complicated combinatorial optimization algorithm, it is one of, if not the most famous. I figured it was a good start. I also decided to implement Kruskal's algorithm to have a more complete view. The goal was to test these two algorithms on different data representations.

Indeed, it is no secret that data can be stored in different ways: vectors, graphs, matrices... no one representation rules them all: it depends both on the algorithm and the instance, which makes it complicated to always use the best one.

My goal was to measure the actual computation time of different data sets on both algorithms. This way, I can compare the results to the theoretical complexity studied in class. I will also be able to understand which data structures work best with different algorithms.

I decided to use multiple data structures to represent graphs. The first one was the very classic "adjacency matrix", which is both simple to visualize and understand. But it is not always optimal. Indeed, if the graph is "scarce", meaning if there are not many edges compared to the amount of vertices, then the data structure stores many zeros in memory, which increases computation time. On the other end, it can be very useful for a dense graph.

The second implementation was one where each vertex is linked to an array storing its neighbours. This worked very well for scarce graphs, since it only stores the useful information. But on the contrary, it was overkill for dense graphs, where a simple adjacency matrix would do the trick better. It would be easier to compute, thus faster.

I also tried my way around Fibonacci heaps and implemented a structure. Yet it did not work well with the algorithm. Research tends to show that this heap has mainly theoretical advantages. Sure, the complexity might be better, but if the growth factor is enormous, a "normal" algorithm test won't do the Fibonacci heap justice.

2.2 Detail of the two algorithms

2.2.1 Dijkstra

Dijkstra's algorithm is amongst the most famous shortest path algorithms. Its downside is that it only works for graphs with positive weights. Given a undirected graph and two vertices (start and finish), it will output the shortest path from one to the other.

- The shortest path is the one minimizing the sum of weights across the path. Hence all vertices are assigned a value: 0 for the start, infinity for all the others. All vertices are set as unvisited.

- Then iterate over the unvisited neighbors of the last vertex (at the start, there is only the start vertex). Keep the one with the smallest edge weight and mark it as visited.
- Iterate until there are no unvisited vertices.

This algorithm is very famous and broadly used, yet Dijkstra said during an interview that he designed it during a coffee break in twenty minutes' time, without pen or pencil.

2.2.2 Kruskal

Kruskal's algorithm aims to solve a different problem: that of finding a minimum spanning forest. Kruskal's algorithm outputs a spanning tree if the graph is connected. Hence we will test Kruskal with connected graphs only.

Here is how it works for connected graphs:

- create a forest where each vertex in the graph is a separate tree
- create a set of the edges in the graph
- until the set of edges is not empty and the forest is not spanning, iterate by: removing an edge with minimum weight from the set of edges, adding it to the forest if it connects two trees into a single one.

The algorithm will thus output a minimum spanning forest. But if the graph is connected, then that forest is a single tree.

2.3 Code Overview

The project is two-fold. It provides an implementation of Dijsktra's and Kruskal's algorithms, written in C++ and compiled through a makefile, and also generates random graphs. To properly test an algorithm, it is crucial to have a representative dataset.

I used both Python and bash scripting to generate the graphs, which can then be implemented into either two of the basic data structures. They are referred to as "graph" and "std_graph". The first one is the adjacency matrix, the other one the vectors containing neighbours of a vertex.

Graphs are generated by calling a Python script. The user can play on the number of graphs and the size of them. That second parameter has two variables: the number of vertices, and the density of the graph, which is roughly whether a vertex has "many" edges or not.

The density is a float between 0 and 1. When given 1, the algorithm will compute connected graphs, which is great for testing Kruskal's algorithm implementation.

The script creates a C++ header file which can be fed to the Makefile. The computation time can then be measured, independently of the time needed to actually create the graphs.

2.4 Results

In order to check the complexity of Dijsktra's algorithm, I ran it on multiple graphs of various size and density, for both structures I had implemented.

First on the structure "Graph"'.

First, here (figure 1) is the evolution of computational time for Dijkstra's on the structure "Graph":

To obtain the figure (figure 1), the algorithm was run on graphs of 10 to 400 nodes. The figure on the left shows the evolution at a given density, while the one on the right shows the evolution with different densities.

It can be noted that for densities over .6, computation time increases with density, which can be expected. But for densities lower than that, there is no consensus. No matter the density, it increases gradually with the size, which is very logical. It can be thought that for densities too small, the data

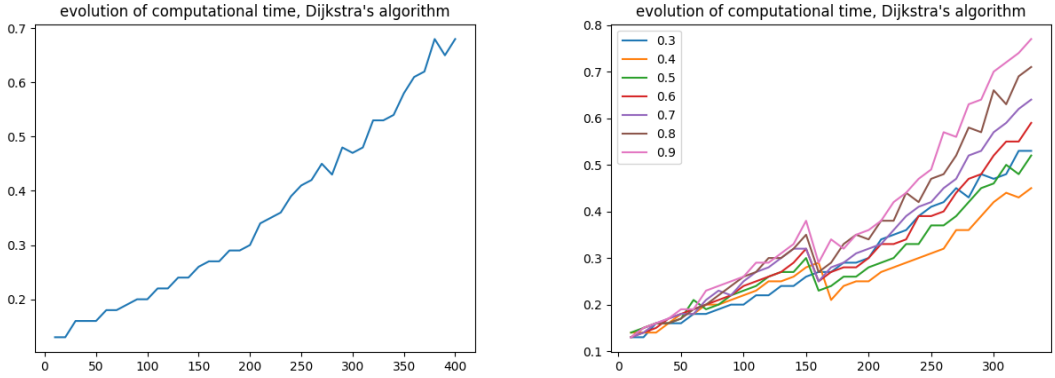

Figure 1: dijkstra_graph

structure is such that it does change much to have a lower density. There is some kind of threshold effect, at which the unadapted structure overcomes the density variation.

Dijkstra's algorithm has a $\mathcal{O}(e + v\log(v))$ complexity, where e is the number of edges and v the number of vertices. The curves obtained are not unlike that complexity of $v\log(v)$. Though the prefactor is unknown and the number of vertices in the graph stayed relatively small in my tests (max 400). It was already sufficient for my computer to run for quite some time

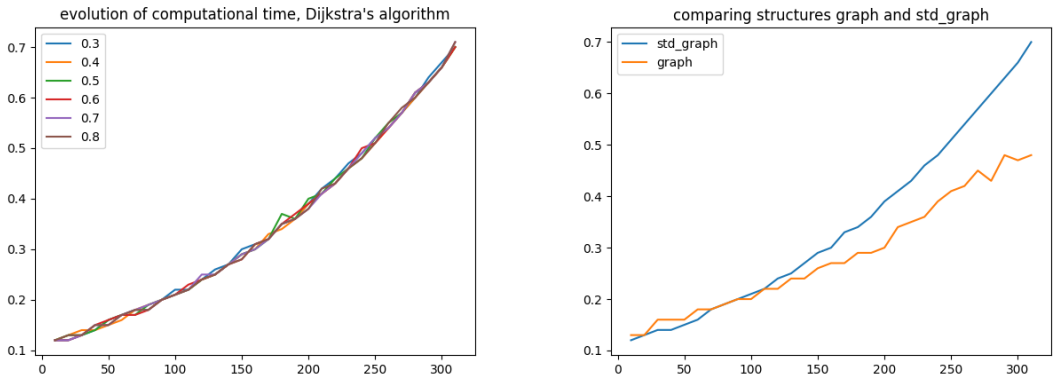

Figure 2: structure std_graph

Now, let's see the Structure "std_graph". First alone at different densities, then it can be compared to the results obtained with `graph` (figure 2).

It can be seen that `std_graph` tends to be slower as the number of nodes (and hence arcs) increases. That's probably due to the fact that the algorithm is very similar, but the data structure is larger when the density is big. The structure is less memory-efficient for large densities.

To check that hypothesis, let's compare the size of the files containing structures `graph` and `std_graph` at two different densities (figure 3).

As expected, the files are very similar when the density is high. `std_graph` always has the same size and is hence more relevant for **high densities**. On the contrary, when the density is low, `std_graph` contains lots of zeros, whereas `graph` only keeps the **relevant** information. That's why `std_graph` takes up twice as much space on low densities.

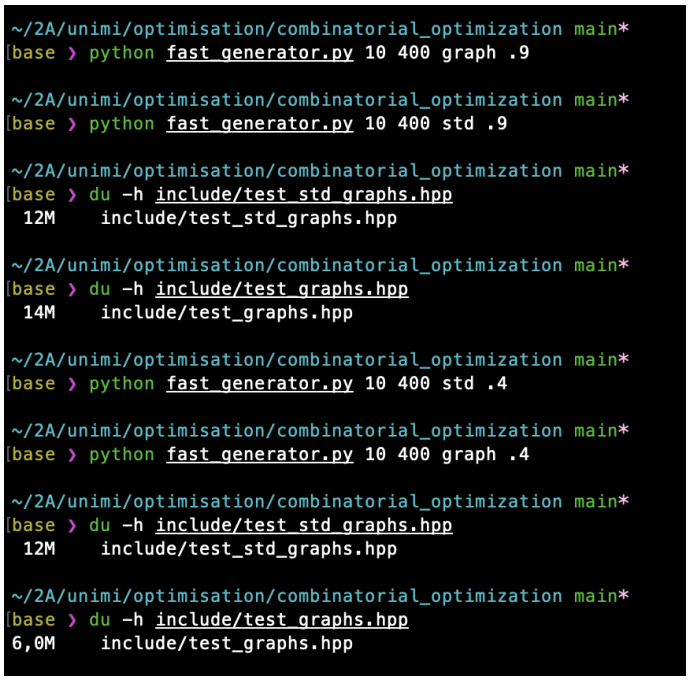

Figure 3: comparing_size

A point on **methodology**: to run the tests, I wrote some shell scripts to automatize the tasks of varying densities and number of vertices. To have good results, I ran each combination of density and number of vertices 10 times and computed the mean time of this, so as to reduce measuring uncertainty as much as possible. I also shut down all other activities on my computer so as to avoid noise on the measures. This way, I hope that the results are as precise as possible and reflect the reality of the algorithm's performance.

3 Thanks

I would like to thank Professor Giovanni Righini for his very interesting class. Coming from France, I had the opportunity to attend an Erasmus+ exchange in Milan and learned lots from it. The University of Milan was very well organised and the classes very challenging.

4 Appendix

4.1 Source Code

All the source code is available on github: github.com/plgstalter/combinatorial_optimization.

For the sake of simplicity, I will just present the organization of the files.

```
base ❯ tree
.
├── build
│   ├── BinaryHeap.o
│   ├── FibonacciHeap.o
│   ├── Graph.o
│   ├── dijkstra.o
│   └── std_graph.o
├── include
│   ├── BinaryHeap.h
│   ├── FibonacciHeap.h
│   ├── Graph.h
│   ├── dijkstra.h
│   ├── std_graph.h
│   ├── test_graphs.hpp
│   └── test_std_graphs.hpp
├── main.cpp
├── main.o
├── makefile
├── project
└── src
    ├── BinaryHeap.cpp
    ├── FibonacciHeap.cpp
    ├── Graph.cpp
    ├── dijkstra.cpp
    └── std_graph.cpp

3 directories, 21 files
```

```
base ❯ tree
.
├── fast_generator.py
├── generator.py
├── graphs.h
├── graphs.txt
├── kruskal.cpp
├── kruskal.h
├── kruskal.o
├── main.cpp
├── main.o
├── makefile
├── project
├── quicksort.c
├── test_graphs.h
├── test_graphs.txt
└── test_std_graphs.txt

0 directories, 15 files
```

Figure 4: Organization of source files for Dijkstra (left) and Kruskal (right)

The structure is pretty simple.

And the makefiles as well:

For Dijkstra:

```
CC=g++
CFLAGS= -std=c++17 -Wall -ggdb -Iinclude

src := $(wildcard src/*.cpp)
obj := $(subst src, build, $(src:.cpp=.o))

.PHONY: directories

all: directories project
@echo $(obj)

project: main.o build
$(CC) $(CFLAGS) -o project main.o build/*.o

main.o: main.cpp $(obj)
$(CC) $(CFLAGS) -c main.cpp

build/%.o: src/%.cpp include/%.h
$(CC) $(CFLAGS) -o $@ -c $<

directories: build
build:
mkdir -p $@

clean:
rm -r build
rm main.o
rm include/*.gch
```

And for Kruskal:

```
.PHONY: directories

all: project

project: main.o kruskal.o
g++ -o project main.o kruskal.o

main.o: main.cpp kruskal.h
g++ -c main.cpp

kruskal.o: kruskal.cpp
g++ -c kruskal.cpp

clean:
rm *.o
rm project
```