Artificial Intelligence in the Maximum Clique Finding Problem

Applications

DENISS KUMLANDER
Department of Informatics
Tallinn University of Technology
Raja St.15, 12617 Tallinn
ESTONIA

Abstract: In this paper we propose collecting different maximum clique finding algorithms into a meta-algorithm, which enables to solve this NP-hard problem much more efficiently. We provide guidelines on how this intelligent meta-algorithm can be built, what information is needed from the maximum clique finding point of view and propose an elementary structure of it. Besides we review a test environment issue for the maximum clique finding area. This topic usually is undervalued, although enables to provide knowledge on algorithms behaviors and connections between algorithms and graph types, which later could be converted into the intelligent meta-algorithm’s rules and definitions. We describe in this paper the test environment model, define each part of it and propose integration principles.

Key-Words: Artificial intelligence, test environment, maximum clique finding, NP-complete task

1 Introduction

Let \( G=(V,E) \) be an undirected graph, where \( V \) is the set of vertices and \( E \) is the set of edges. A clique is a complete subgraph of \( G \), i.e. the vertices of which are pairwise adjacent. The maximum clique problem is a problem of finding the maximum complete subgraph of \( G \), i.e. a set of vertices from \( G \) that are pairwise adjacent. Those problems are NP-hard on general graphs [8], no polynomial time algorithms are expected to be found. There is a great interest in developing a fast exact algorithm for instances with a reasonable number of vertices since it can be used in several important practical applications. Examples are efficient register allocation [6], on-line bin-stretching [1], scheduling of parallel jobs [3] and a lot of others [4].

This article’s aim is to put together different ideas, possibilities and needs arising in the maximum clique finding and synthesize an intelligent algorithm that could address all those issues. Here we look on the maximum clique finding problem from the programming, i.e. applying point of view rather than from a poor mathematical point of view. Ideas about an intelligence of algorithms are widely discussed in data analyses, data mining and similar areas and less in the NP-problems; although some ideas are used in heuristic algorithms – see for example Jagota and Sanchis 2001 [9]. Therefore we try to start this discussion by this article. Another topic we discuss is a testing environment needed for the maximum clique algorithms. This is an infrastructure for the algorithms’ research procedure and is a way to derive a lot of knowledge about algorithms behaviors and connections between algorithms and graph types. We provide a model of building the test environment basing of a huge number of experiments we did earlier [12, 13]. This part of the work contains a discussion started by Johnson [10].

2 Artificial Intelligence for the maximum clique finding

Here we are going to present a philosophy of building an algorithm that concentrates inside itself all best algorithms and is intelligent enough to apply the right one. This idea means that we have to have a meta-algorithm that will collect data, the meta-algorithm that will have some intelligence. Different types of intelligence could be used. The easiest way is to have an “expert systems” type meta-algorithm, which will have fixed type rules. The more complex one could be clever enough to learn like, for example, neural networks do.

2.1 “Expert” type intelligence

Unfortunately there is no universal algorithm that solves all graphs cases faster than other algorithms. It is rather common to have a set of algorithms or modifications of those that have different strong sides and therefore are good in solving one or another particular graph case. Besides there are
graph types that can be solved in a polynomial time by dedicated algorithms.

Therefore, the first idea could be to collect all this knowledge inside one algorithm, i.e. build the meta-algorithm with fixed rules that will select the best algorithm basing on the preliminary information about a graph to be solved, or basing on an initial analyse of the graph. The easiest information that we usually have before running the main algorithm is the graph’s density. For example, it is well known that if the density is no bigger than 10% then there is no better algorithm for solving the maximum clique problem than the trivial and powerful Carraghan and Pardalos one [5] that solves the problem directly without spending valuable time to any unnecessary additional steps [12].

Another type of information that we can have is the type of a graph. Of course this information is not always available, but if you have it or know how the graph is built then it is possible to save a lot of time by applying the right algorithm to find the maximum clique. For example, there are permutation, perfect graphs, interval graphs and some others that can be easily solved by corresponding algorithms in the polynomial time and there are no points to apply for them algorithms targeted to solve all possible types/structures of graphs [2, 4, 11]. Certainly only some graphs can by solved in the polynomial time, but even for graphs that are hard to solve there could be algorithms that suit more. Besides usually researches do not tune their algorithms to perform better on one or another graph type, but this could be done and this provides a lot of possibilities to come up with tuning ideas for existing algorithms to make those better on certain graph types. It is logical also, that all those modifications should be available to the meta-algorithm to choose, which of them to run.

The meta-algorithm should follow the next general rules:

- If the type of a graph is known then it should run the best algorithm for that type;
  
  **Note:** It means that the meta-algorithm should have some knowledge base into addition to available algorithms that will allow choosing the right algorithm to run.

- Choose an algorithm basing on the graph’s density;

- Repeat previous step for each parameter the meta-algorithm can consider;

- Run the algorithm that was chosen.

### 2.2 Algorithm Learning and Results

#### Knowledge Base

In the previous chapter we have reviewed possibilities to use fixed type rules. We have built the meta-algorithm, which is an expert in the maximum clique finding. We used “Expert systems” ideas and provided our meta-algorithm with all knowledge we have at the moment. Unfortunately, we do again an assumption that we have to invent an algorithm that will deal with very different graphs and that should solve any graphs. There is one motto that is widely used nowadays – “Think globally, operate locally”. Any particular case could have its own aspects, properties etc. of graphs to be solved. We could not foresee all those aspects and moreover those can be opposite from what we were expecting, or those requirements could be opposite from an algorithm building point of view. Therefore the ideal case will be a self-learning algorithm. Of course, we do not talk about a meta-algorithm that will invent new algorithms to find the maximum clique. Probably it is too self-confidently to try invent such right know. What we mean is a meta-algorithm that will be able to collect some statistic about algorithms’ performance on graphs that were solved and later, basing on this statistic, will be able to choose, which algorithm to run. This meta-algorithm will adapt to a particular environment and graphs existing in this environment, to the environment where it has to operate. This adaptation will mean that we move from the general “expert system” to the more evolving algorithm, which is able to “survive” in any particular environment in the best way.

The information collecting, which algorithm/modification is better, generally means that the meta-algorithm will try to run all algorithms/modification with all graphs. Otherwise it will not be possible to answer a question: “Will any other algorithm perform better than the one we are going to use?”. Another important aspect we should think about is providing more information than the meta-algorithm can collect by itself like a graph’s density or number of vertices. Is there any additional information on the graph? Are all graphs the same or you know their types? Is it possible to distinguish a source of the graph? Any such information will be useful to keep statistic and better adapt for any particular graph cases.

Now, when we know all additional information and we can pass those details into the meta-algorithm. The main question is how to start collecting data? This can be done in two ways.
The first way is to run other algorithms while in the stand-by mode. Sometimes the meta-algorithm that finds the maximum clique is asked rarely to do it. So it doesn’t have to resolve immediately another problem after the previous one. In this case, after returning an answer, the meta-algorithm can use available free resources. It can try all other different algorithms, which weren’t used to give the answer, to find if there was a better/quicker way to perform the task. Basing on the collected information each algorithm could receive points (for example 1 point each time to the fastest one). Basing on those points an algorithm to be used the next time should be chosen. If there is a high probability that a new task will arrive soon then the meta-algorithm should try algorithms in the already obtained points’ sequence. This will allow trying first of all the most probable one to be the fastest then the next probable one and so forth.

Another idea is to train the meta-algorithm use algorithms available inside him as sub-algorithms for finding the maximum clique prior to the real using. The idea allows collecting statistic before you start to use the meta-algorithm and it will not be necessary to spend resources on later statistic collecting while in operation.

The training could be done by asking to solve as many different types of graphs as possible in all required modes if any exists – like a requirement to stop after, for example 10 seconds and provide the best found solution etc. For any type/mode as many graph examples as possible should be used. Instead of using artificial examples, it is always advisable to use such examples that will likely occur later during the real using of the meta-algorithm.

It is also important to monitor the performance and changes in the environment. If graphs to be solved are changing due some changes in the requirements or you suspect that the meta-algorithm is not providing its best, then it is time to re-train the meta-algorithm.

Both ways have another very important advantage into addition to described – those ways allows collecting information that makes possible also to learn (you) how well graphs are solved and which algorithms are used to solve any particular graph case. This gives a possibility to analyse collected statistic and may be invent even better modifications of existing algorithms.

3 Testing Environment

Here we are going to describe a testing environment that can be used to test maximum clique finding algorithms. This environment has mainly figured out during our works on maximum clique algorithms [12, 13]. This discussion can be seen as the next step of experimental analysis of algorithms discussion started by Johnson in the year 2002 [10]. The goal of the testing environment is to test different algorithms for finding the maximum clique and mainly measure a time needed to find a solution, although some other parameters can be measured in case corresponding parts are implemented for each module (algorithm) to be tested. The following requirements have figured out as essential needs for a testing environment of our type, i.e. for algorithms finding the maximum clique:

- It should be able to test different types of graph classes, like random and external graphs. Random graphs means that the system should have a module that is able to generate random graphs. Note that a „true” randomisation is required, since each time a lot of graphs of the same type should be provided. There is no point to generate graphs that are very similar and moreover it should not happen that randomisation is restarted each time a graph is generated and it leads to generating exactly the same graphs. Another type that could be loaded into the environment is external graphs. Note that currently different standards exist therefore the following types of graphs definitions should be supported: DIMACS format graphs – both compressed and decompressed versions [7]; Adjacency matrix graphs, i.e. graphs that are defined by an adjacency matrix.
- It should be able to solve both problems: finding the maximum clique and finding the maximum independent set using the same modules (algorithms), since those problems are equivalent and there are different graphs for both problems.

Getting into account those requirements we have figured out the testing environment that contains the following main parts:

- Algorithms or modules that implement one or another algorithm;
- Utilities’ modules that generates graphs;
- A meta-algorithm that makes tests by rerunning algorithms with the same graphs;
- A user interface
- Providing a feedback, i.e. info on events and the current processes’ status;
- Allowing defining algorithms to be tested and graphs to be used for testing.

Let’s review each module separately.
3.1 Modules
Modules are parts of the environment that are implementing algorithms. Each module should have two main properties:

- It should be standard from the input/output interface point of view;
- It should be written using the same programming language and techniques as other modules, as much as possible. This will ensure that neither algorithm is better due the better programming. So, all tunings made for any algorithm should be transferred to others if it is possible.

So, each algorithm is implemented as a standard module and can be easily added into / removed from the testing environment. The input parameter is a graph to be solved and the output is the size of the maximum clique. It is required to control if all algorithms are working right and size of the maximum clique of a graph obtained by different algorithm is the same. Note that for test we are mainly concentrating on spent times and sizes of the maximum clique rather than an actual maximum clique vertices as an output.

It is also possible to measure some other parameters by programming into modules a standard block for that. The block is programmed once and then adopted inside each algorithm. The ideally programmed block should not require any adaptation since otherwise similarity of algorithms will decrease because of such measuring. This way we measured a number of analysed branches / iterations made by algorithms. The ideal way to activate such blocks is a global variable. Although it is not advised to have global variable, here it looks to be the best way to go since allows controlling of algorithms work from one central place and makes algorithms easily moveable between the meta–algorithms slots that are activating algorithms’ modules.

3.2 Utilities
This is a part of the environment that provides a general level functionality. First of all those are input/output functions:

- Function allowing reading external graphs;
- Function allowing generating a random graph;
- Function allowing saving results in an output file.

As we already stated before 2 main formats have to be used: DIMACS and Adjacency matrix. The first one is the main format that is used in researches. Graphs of this format are often compressed and stored in so called binary format, although the decompression algorithm can be easily found in the Internet or in the same ftp folder of the DIMACS program, where graphs are stored. The second format is used in some university classes, since a graph definition using an adjacency matrix is more visual and therefore is easily understandable by students.

The only note that we can do on generating random graphs - sometimes it is necessary to generate random graphs of a predefined type. So, it is possible to use more than one graphs’ generation technique and choose one of them using an option somewhere on the main user interface. Please note that whatever way a graph is generated or whatever format of an external graph is used, internally the graph should be saved in one, “standard” for this test environment way. This ensures that all graphs are treated in the same way by modules. Besides, the graph’s reference i.e. the input parameter stream for modules will be the same for all cases.

3.3 Meta-algorithm
A meta-algorithm is a main part of the testing environment that mainly glue parts together and manage those parts work. The main goal of the meta-algorithm is to run algorithms one by one using the same parameters and capture a time spent of finding the maximum clique and check correctness of algorithms work by comparing the result – size of the maximum clique produced by different algorithms. The process of testing is done in iterations for all densities and numbers of vertices that are required to be tested as many times as it is required. An alternative process is providing algorithms with an externally defined graph and capturing the same output parameters, as it was defined above. Anyway, each time exactly the same graph should be provided for each algorithm to be tested. Note that each testing iteration should be able to involve activating different modules – options in the user interface should define which algorithms to test.

Another important feature of the meta-algorithm is storing results and calculating statistic – minimal, maximal and average results. We have found that it is useful to output as individual number as the statistical information since the statistical information is the main research result while individual numbers allow understanding trends and make other calculations in case those were not planned in advance.
An ideal structure of activating modules can be the next:

- Modules should be built using the same base class, which will have a starting function having common input/output parameters – those have been described earlier;
- The meta-algorithm should have a set of slots (array or collection), which can contain base classes, so any module can be placed into any slot. Those modules are put into slots if and only if those should be tested in the general testing iteration – it is defined by options at the user interface;
- Modules from each slot should be run one by one either for each generated graph or for an external graph and modules’ output properties captured. Note that ideally the same slots should be able to store those output parameters;
- The activation, which is described on the previous step, should be done as many times as it is defined in the user interface. We used to run each test 100 times to collect enough data to make a trustable statistic. The result of this process is an output using the utilities’ module to an external file;
- If randomly generated graphs are tested then the previous step should be done for each vertices number / density. Densities should be defined as a range allowing testing more than one density at once (during one test iteration). Note that best practices make us to advise defining a vertices’ number for each density rather than one vertices number for all densities since a time spent on finding the maximum clique on different densities for the same time differ dramatically. Therefore it is useful to orient on the spent time you want to have rather than on the number of vertices.

The meta-algorithm should also produce events allowing seeing a status of the testing process. So, the meta-algorithm is a core part of the testing environment that manipulates modules and storing results of the testing process.

3.4 User interface

This is the last element of the testing environment but it doesn’t mean that this element is not important. Of course, it looks like the testing, i.e. the meta-algorithm and modules are main important parts, but it isn’t quite true. The visual feedback is very important as well as a possibility to define options in the easy and comprehensive manner. It makes the environment user friendly and allows testing more and quicker.

The user interface should first of all allow defining (will graphs be generated or provided externally. If the graph is provided externally then the source of the file containing the graph description should be defined and the type of the graph description, i.e. DIMACS or Adjacency matrix. If graphs should be generated then a range of densities and the step of moving inside the range should be defined. For example densities from 10% to 90% with a step equal to 10%. In addition to that a range of numbers of vertices and the step of moving inside the range or numbers of vertices for each density you are planning to have in the testing and a number of times graphs should be produced and tested for each density and number of vertices. The last but not least parameters are a destination’s file for the output of the testing and which algorithms to test.

The user interface should also show the status of the testing and a message on end of the testing process. This helps to orient in the testing process workflow and immediately recognize situation when resolving graph cases do require much more time to find the maximum clique that was planned initially.

3.4 Integration

This last subchapter puts all modules together and shows how those are integrated / work together. The integration can be done if and only if all parts are using same standards / interfaces, raising standard events and returning expected outputs. It was shown earlier in the “Modules” and the “Meta-algorithm” subchapters what the standardization means for the modules’ structure and in the utilities’ part for graphs to be read or generated. The test environments parts can be divided into three layers. The high-level model has three layers. The first layer contains the user interface, which lets a user to work with the meta-algorithm. This layer is an intermediate layer that disables users to work / interact directly with the meta-algorithm or modules (for example in a DOS like mode). This layer verifies correctness of input parameters. The second layer contains the meta-algorithm – the core of the system that receives parameters from the user interface and run tests. It is a testing logic layer. The third layer contains both utilities and modules. Those objects are indirectly interacting using a graph object that is created by utilities and consumed by modules (implementing algorithms), which are finding the maximum clique in it.
4 Conclusion

In this paper we described some ideas of building an intelligent meta-algorithm that is able to adapt for graphs that have to be resolved in a particular environment and apply the best variation of maximum clique finding algorithms for each individual graph case. Those ideas came from expert systems and data analyses and can be successfully used in the maximum clique finding from our point of view. This discussion is more an applying part of the maximum clique finding although this cannot be done without understanding of what the maximum clique finding problem is and which properties should be tracked by the meta-algorithm. Using of such methods should increase the general performance of algorithms in solving this NP-hard problem. It will also enable to save a lot of person hours when a system is intelligent enough to make decisions. In the final part of the meta-algorithm approach description we came up with some advices for the meta-algorithm rules, which can get those properties into account.

Another part of this work contains a description of the algorithms testing environment for the maximum clique finding problem. This topic is usually undervalued although is an essential part of any research, even mathematical one, since allows proving results and modelling real dependencies between different parameters, like for example a graph type and a number of branches to be analyzed. During our tests some guidelines were worked out in enormous number of different experiments, mistakes and mis-modellings of the test environment. We hope that this model can be useful for other developers and researches at least as a starting point. Anyway it contains all our experience in testing maximum clique algorithms [12, 13] in a form of a set of advices and as our vision on the testing environment.

One of the most interesting topics for the future researches in this area can be building international standards on programming algorithms (by inheriting from some base class), graphs presentation and storing and outputting data. Of course there are some standards, for example the DIMACS standard for graphs, but no more. What we mean under international standards are standards like are used in the XML for data exchange, activating services remotely and so forth.

References: