Working title: Verifying the Edmonds-Karp Algorithm
A Proof Pearl

S. Reza Sefidgar, Peter Lammich

Technische Universität München, {sefidgar,lammich}@in.tum.de

Abstract. the Ford-Fulkerson algorithm is a generic method that computes the maximum flow in a flow network. This paper outlines a formal proof of the Ford-Fulkerson theorem in Isabelle/HOL which follows the informal proof found in standard algorithm text books. This formalization is then used to develop an implementation of the algorithm in Standard ML. Refinement techniques are used to transform an abstract definition of the algorithm into an executable code.

1 Introduction

The Ford-Fulkerson algorithm [cite] is one of the important results of the graph theory and is used to find a solution to the maximum flow problem in flow networks. Many important mathematical problems like the maximum-bipartite-matching problem, the edge-disjoint-paths problem, the circulation-demand problem, and many other scheduling and resource allocating problems can be reduced to the maximum flow problem. Hence, the Ford-Fulkerson algorithm has major application in the field of mathematical optimization.

Despite its importance, no formalization of the Ford-Fulkerson algorithm has been developed in modern proof assistants. The only similar work that the authors are aware of is formalization of the Ford-Fulkerson algorithm in Mizar [cite]. This formalization defines and proves correctness of the algorithm at the level of graph manipulations without providing concrete implementation of the algorithm. Providing such an implementation is specially important, as it could provide the basis for many verfied programs with practical importance.

This paper present a formalization of the correctness proof of the Ford-Fulkerson algorithm in Isabelle/HOL. Our proof is based on the informal proof of the algorithm which is presented in the book "Introduction to algorithms" [cite]. Due to practical importance of the algorithm, we also present a verified implementation of the algorithm. Isabelle/HOL provides some automation for generating the code corresponding to a verified algorithm, however, in order to use the code generating features, one needs to do the formalization with executability in mind. Being limited only to executable concepts in the formalization has the disadvantage of cluttering the proofs with implementation details. Such approach makes the proofs more complicated, and may even render proofs of medium complex algorithms unmanageable.

One solution for the aforementioned problem is refinement [cite]. In order to generate executable code of an algorithm using refinement, we first formulate the algorithm on an abstract level. The abstract version of the algorithm has a clean correctness proof as it only captures the idea behind the algorithm. Next, we refine the abstract definition of the algorithm towards an executable implementation in possibly multiple refinement steps. During each step, we only need to prove the correctness of the implementation of a particular abstract concept. Hence, we can be sure about the correctness of the resulting executable program as each refinement step preserves the correctness.

There are several approaches to data refinement in Isabelle/HOL. We will be using the Autoref tool [cite], which has been used for proving more complex results such as formalized implementation of Hopcroft's DFA-minimization algorithm [cite]. Given an algorithm phrased over abstract concepts like sets and maps, it automatically synthesizes a concrete, executable algorithm and the corresponding refinement theorem. To make it applicable for the development of actual algorithms, Autoref is integrated with the Isabelle Refinement Framework [cite] and the Isabelle Collection Framework [cite].

2 Short Background on MinCutMaxFlow and FoFu

3 Formalizing MinCutMaxFlow and Fofu

highlighting nice aspects of our formalization. Do we use some new techniques. Elegantly exploit existing techniques? E.g. Refinement to elegantly describe the FoFu-Scheme, and the instantiate Edmonds-Karp.

2 and 3 perhaps intermixed

4 Refinement to executable code

5 Benchmarking

6 Conclusion

... and related work

Contributions

Formal proof of mincut maxflow fofo-scheme inst to edmonds karp refinement down to executable code. Roughly 5 times slower than Java. (What about OCaml) + NetCheck

What shines (its a Pearl) Min-Cut Max-Flow: Textboook like formal reasoning: Comprehensible proof, BUT machine checked (Present one (carefully worked) example in paperI)

Refinement based approach: Fofu-Scheme, instantiation to EdsKa. +1: Abstract Algo looks almost like pseudo-code you would expect in textbook. +2: Fofu-Scheme proved correct for all aug-path finders. EdsKa is instantiation of it.

+3: Modularity: Fofu-scheme and pathfinder developed+proved independently of each other.

Down to executable code, plugging in efficient data structures.

Some minor contributions: Reusable BFS algorithm Imperative matrix data structure (really minor).

References