IsaGraphLib: Formally Verified Efficient Graph Algorithms in Isabelle/HOL

Mohammad Abdulaziz[1], Thomas Ammer[1], Peter Lammich[2], and Adem Rimpapa[1]

[1] King's College London, London, United Kingdom
{mohammad.abdulaziz,thomas.ammer,adem.rimpapa}@kcl.ac.uk
[2] University of Twente, Enschede, The Netherlands
p.lammich@utwente.nl

Abstract. We present a library of formally verified graph algorithms in Isabelle/HOL. The library covers the main problems of combinatorial optimisation on graphs: reachability, shortest paths, minimum spanning trees, matching, maximum flow, and minimum-cost flow. The algorithms in the library use most of the standard algorithm design techniques: graph search, greedy algorithms on matroids, dynamic programming, augmenting paths, blossom contraction, blocking flows, push-relabel, scaling, the primal-dual method, randomisation, and certificate checking. The library has adaptors between different types of graphs (directed, undirected, multigraphs), which allow results to be reused. Each algorithm is verified independently of its data structures, and the main algorithms are refined to efficient imperative implementations, from which we generate code in Standard ML, Scala, and Go. We developed parts of the library with large language models (LLMs), improved the quality of their proofs with a method based on statistics of our own proofs, and labelled which parts were developed by humans, by LLMs, or collaboratively. We perform a detailed experimental analysis of representative algorithms in our library, comparing it with state-of-the-art mainstream libraries, showing that our implementations are practical.

Keywords: Formal Verification · Graph Algorithms · Interactive Theorem Proving

1 Introduction

Algorithms for combinatorial optimisation problems on graphs have influenced both the theory and the practice of computing. Theoretically, polynomial running time as the notion of efficient computation was proposed by Edmonds in his paper on matching in general graphs [27]. Practically, matching algorithms are used in kidney exchange programmes [17] and in allocating online advertisements [71], minimum-cost flow algorithms compute the cheapest way to transport goods [46], and shortest path algorithms are used in route planning [15], AI planning [19], and model checking [12]. Ensuring the correctness of implementations of these algorithms is difficult without formal verification, for two reasons.

Table 1. The verified algorithms, grouped by the problems they solve. A citation marks an algorithm whose formalisation was published before; the formalisations of the other algorithms are new. *Proved*: termination (T), correctness (C), optimality (O), and approximation or competitive ratio (A). *L.*: the listing in Appendix D with the main theorem. *Lines*: the number of lines of the theories specific to the algorithm. Table 3 in Appendix A gives the details of the verification of every algorithm, and Table 38 in Appendix C shows which of them have abstract (i.e. non-executable), functional, and imperative versions. The library has about 300,000 lines of Isabelle/HOL.

Algorithm	Proved	L.	Lines
Reachability and cycle detection			
Depth-first search [2]	T, C	L53	877
Breadth-first search [2]	T, C	L56	1,805
Cycle detection	T, C	L54	3,305
Directed cycle detection	T, C	L55	697
Shortest paths			
BFS with distances	T, C	L57	2,901
Bellman–Ford	C, O	L58	2,837
Dijkstra	T, C, O	L59	2,865
Maximum flow			
Dinic	T, C, O	L66	1,455
Blocking flow	T, C	L67	713
DFS with backtracking	T, C	L68	746
Push-relabel	T, C, O	L71	4,897
Minimum-cost flow			
Successive shortest paths [3]	T, C, O	L64	1,158
Capacity scaling [3]	T, C, O	L65	1,785
Orlin [3]	T, C, O	L1	12,080
Network simplex	T, C, O	L69	7,582
Acyclic flows	T, C	L70	10,691

Algorithm	Proved	L.	Lines
Spanning trees and matroids			
Best-in-greedy [5]	T, C, A, O	L60	5,750
Kruskal [5]	T, C, O	L61	1,559
Greedoid greedy [5]	T, C, O	L62	2,217
Jarník–Prim [5]	T, C, O	L63	1,277
Matroid intersection [5]	T, C, O	L80	2,646
Weighted matroid intersection	T, C, O	L81	5,274
Matroid partitioning	C, O	L82	1,215
Matching			
Edmonds' blossom [7]	T, C, O	L2	10,460
Weighted blossom	C, O	L72	5,801
Edmonds–Gallai decomposition	T, C	L73	5,007
Hopcroft–Karp	T, C, O	L74	2,381
Hungarian method [4]	T, C, O	L75	4,123
Primal-dual (Egerváry) [4]	T, C, O	L76	1,863
Online matching			
RANKING [6]	A	L77	6,973
RANKING, primal-dual [4]	A	L78	4,861
AdWords [4]	C, A	L79	1,347

First, their correctness arguments are mathematically involved, with many concepts and complicated constructions. Second, practical implementations employ low-level optimisations, e.g. storing graphs in the compressed sparse row (CSR) format and renumbering vertices to improve cache locality, whose correctness is hard to ensure within a large and complicated algorithm. We assume that the reader is familiar with common graph algorithms, otherwise we refer to standard literature [48,78,9,85].

In this paper, we present practical formally verified (implementations of) algorithms in Isabelle/HOL [74] for most of the main combinatorial optimisation problems on graphs (Table 1). In particular: 1. we present a library of these implementations, from which we generate code in Standard ML, Scala, and Go [37,82], 2. we describe the methodology underlying the library (Section 2), which extends that of Abdulaziz [2, Chapter 24] to substantially more involved algorithms, 3. we refine our algorithms to imperative implementations without a custom mechanism for non-determinism, which, as far as we are aware, is a first (Section 2.5), using a separation logic framework for Imperative HOL which we extended, e.g. with monadic folds and while-loops, 4. we report on developing

parts of the library with large language models (LLMs) (Section 4): we label parts of the library as developed by humans, by an LLM, or collaboratively, and we restyle LLM-generated proofs using statistics of our own proofs, and 5. we evaluate the implementations of representative algorithms experimentally against LEMON [24], SciPy [84], NetworkX [83], JGraphT [72], and Sage [23] (Section 3).

2 Methodology

Our library uses Abdulaziz's [2, Chapter 24] methodology for verifying graph algorithms: the abstract data type of graphs (specified as an Isabelle/HOL locale) and the mathematical representation of graphs as sets of edges (related to the abstract data type by an abstraction function) are both taken from that chapter. However, that chapter applies the methodology to depth-first search (DFS) and breadth-first search (BFS), while we apply it to a far larger set of algorithms [3,5,7,4] whose verification is substantially more involved: our formalisation of minimum-cost flow algorithms (L. 1) [3], including Orlin's algorithm, is 27,000 lines of Isabelle/HOL, that of Edmonds' blossom algorithm (L. 2) [7] is about 10,000 lines, while the formalisations of DFS (L. 3) and BFS (L. 4) accompanying the chapter are about 800 and 1,500 lines, respectively.

2.1 Mathematical Models of Graphs and Reasoning Principles

An initial motivation for our library was that the Isabelle community has produced a large number of graph libraries, covering both theory and algorithms. Each of those libraries came with its own definitions of graphs and lemmas about them: in the Archive of Formal Proofs (AFP) alone, a graph is represented as a record of a vertex set, a set of arcs, and two functions mapping every arc to its tail and head [75], as a set of vertices together with a set of edges, each of which is a set of vertices [25], as a multiset of such edges [26], as a set of unordered vertex pairs [38], as a value of an abstract type of undirected graphs whose edges are given as pairs of vertices [63], and as a function mapping every pair of vertices to a capacity [64]. Every result in a given representation has to be proved again for every other representation. As we designed our library, we were aiming for results concerning graphs to be proved once and then reused by all the algorithms that need them.

Our library has two main representations, shown in Fig. 1 along with the connections between them: directed graphs, i.e. sets of pairs of vertices [2, Chapter 24], and undirected graphs, i.e. sets of two-element sets of vertices [7]. Each of them has its own notions of walks, `vwalk` (L. 5) and `awalk` (L. 6) for directed graphs and `walk_betw` (L. 7) for undirected graphs. The two representations are connected by *adaptors*, which are theories relating the notions of one representation to those of the other, e.g. they relate paths, given as lists of vertices, of two distinct representations. For example, `walk_betw_iff_vwalk_bet` (L. 8) states that a walk in an undirected graph G is a walk in the directed graph that has both

orientations of every edge of G, and vice versa. Thus, results proved for one representation can be used for the other. Multigraphs, whose edges have an abstract type, are an example of this reuse: a path in a multigraph, `multigraph_path` (L. 9), is defined as a walk, `awalk`, over its edges mapped to vertex pairs by `make_pair`, so all lemmas about walks in directed graphs apply to multigraphs without being proved anew. Finally, adaptors to Noschinski's library [75], which was developed independently of ours, prove that our definitions of arcs, reachability, walks, trails, paths, cycles and decompositions of walks into cycles, (induced) subgraphs, connectivity, and strongly connected components of directed graphs agree with those of that library.

Some theorems about graphs are proved by induction on the graph's edge set. For example, `component_has_path_distinct` (L. 10) states that if every vertex of a finite graph G has degree at most two, then every connected component of G is the vertex set of a path without repeated vertices, and it is proved by induction on G. The induction step has to know how the connected components change when an edge $\{u_1, u_2\}$ is added to G, which is given by a case analysis `in_insert_connected_componentE` (L. 11) of Abdulaziz and Mehlhorn [7]: every connected component of the extended graph is a connected component of G, the new component $\{u_1, u_2\}$, the component of one endpoint extended by the other endpoint, or the union of the components of both endpoints.

The library has further reasoning principles of this kind for other concepts, none of which has been described before: (1) the same case analysis for walks (L. 12), for reachability (L. 13), for walks in directed graphs (L. 14), and for the union of two graphs (L. 15); (2) a lemma that splits reachability in the extended graph into reachability in G to and from the endpoints of the new edge (L. 16); (3) induction principles that build a walk one edge at a time, for walks in undirected graphs (L. 17), for walks in directed graphs given as lists of vertices, extended at the start (L. 18) or at the end (L. 19), or as lists of edges (L. 20), and for paths in multigraphs (L. 21); (4) induction on reachability in directed graphs, forwards from the source (L. 22) or backwards from the target (L. 23); and (5) an induction principle that reduces a property of all walks between two vertices to the walks without repeated vertices by removing closed subwalks (L. 24).

For proving algorithms correct, we need intricate graph-theoretic results, such as, for minimum-cost flows, a criterion for optimality and the preservation thereof (L. 112 and 113), or Berge's lemma (L. 25) for matchings. Although in this paper we clearly focus on the algorithmic side, note that our infrastructure allows formal proofs of deep graph-theoretic results that we do not use for proving algorithms correct, for example, the Tutte–Berge formula (L. 26), or the Edmonds–Gallai decomposition (L. 27) [30,31,27].

2.2 Implementing Graphs

The algorithms do not operate on the mathematical representations of graphs directly, but on abstract data types (ADTs) of graphs [2, Chapter 24]. An ADT is a locale (Section 2.3) that fixes the operations of a data structure and an invariant on its values, and assumes the specification of these operations in terms

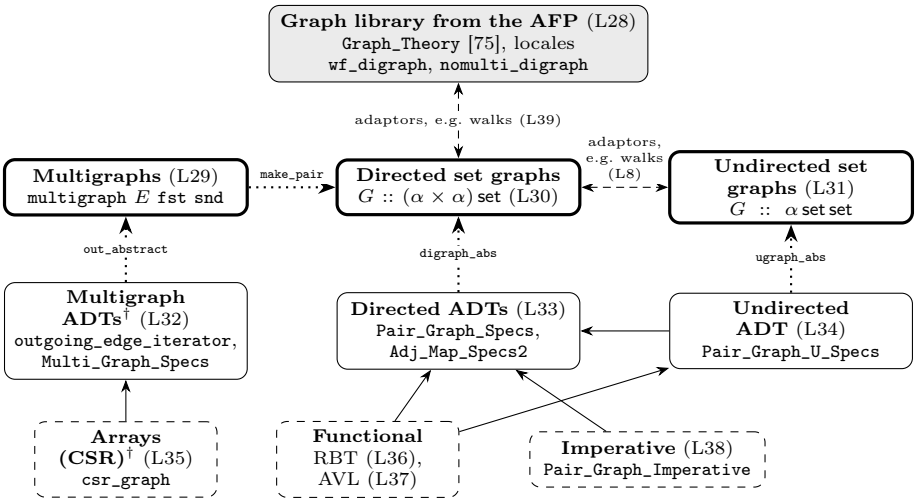


Fig. 1. Representations of graphs in the library and the connections between them. Thick boxes are graph representations, the grey box is an external library from the AFP, plain boxes are ADTs, and dashed boxes are their implementations. Dashed double arrows are adaptors; dotted arrows are abstraction functions; solid arrows are locale imports or interpretations. Daggers mark ADTs and implementations that are only in development branches of the library. Table 4 in Appendix A describes the abstract data types. A reference such as L1 is to Listing 1 in Appendix D.

of an abstraction function, which maps every value satisfying the invariant to one of the mathematical representations. For example, `Pair_Graph_Specs` (L. 33) combines a map ADT with an ADT of sets from which an element can be chosen, maps every vertex to its out-neighbourhood, and is related to directed graphs by `digraph_abs` (L. 40). Accordingly, an algorithm is defined in terms of the ADT's operations, and its proof uses the lemmas about the mathematical representation, which apply to the abstraction of the graph. Table 4 in Appendix A describes the graph ADTs of the library. The four ADTs in the main branch represent a graph as a map from every vertex to the set or list of its neighbours, which suits algorithms that look up and modify neighbourhoods, e.g. DFS and BFS. The ADTs in development branches represent multigraphs whose edges at a vertex are visited with a cursor, which suits algorithms that resume the scan of the edges of a vertex where an earlier scan stopped, e.g. Dijkstra's algorithm.

An implementation of an ADT is an interpretation of its locale with concrete operations, for which the locale assumptions have to be proved. The interpretation yields the definitions and theorems of every algorithm that is defined in terms of the ADT for these operations. The functional implementations use the red-black trees (RBTs) and AVL trees of the Isabelle/HOL distribution [2], both for the map from vertices to their neighbourhoods and for neighbourhoods themselves, e.g. the implementation of `Pair_Graph_Specs` with RBTs (L. 36). The imperative implementations use the array maps or hash maps of the separation

logic framework for Imperative HOL [61] for the map, and duplicate-free lists for the neighbourhoods (L. 41 and 42). In development branches, the CSR representation (L. 35) stores the edges in one array sorted by their sources, together with an array of the positions at which the edges of every vertex begin and an array of cursors. The edges leaving a vertex are thus contiguous in memory, which makes scanning them cache-efficient, and advancing or resetting the cursor of a vertex takes constant time. Note that since the first two arrays are never modified, the CSR representation suits algorithms whose graph does not change.

2.3 Modelling Algorithms

The main mechanism we use to structure the library is Isabelle/HOL's *locales* [14]. A locale is a named context that fixes a number of constants, the parameters of the locale, and states assumptions about them; definitions in the locale can refer to these constants, and theorems proved in the locale can use these assumptions. A locale can extend other locales with further constants and assumptions, and can later be instantiated by giving concrete values for its constants and proving its assumptions for these values. This yields instances of all the definitions and theorems of the locale for these values. We usually split locales modelling algorithms into a *specification locale*, which only fixes constants and their types, and a *proof locale*, which extends the former by the assumptions. For example, the specification locale `multigraph_spec` (L. 48) fixes an edge set E, functions `fst` and `snd` returning an edge's endpoints, and `create_edge` to construct an edge from two vertices. The proof locale `multigraph` (L. 29) assumes that E is finite, that `fst` and `snd` return the vertices given to `create_edge`, etc. We split locales in this way because a specification locale has no assumptions and therefore admits a *global interpretation* for arbitrary values of its constants. Such a global interpretation is what Isabelle/HOL's code generator needs to produce executable code from the locale definitions.

We model the algorithms following the methodology of Abdulaziz [2, Chapter 24], which was also used for the minimum-cost flow algorithms [3], the greedy and matroid intersection algorithms [5], and the Hungarian method [4]. The variables that change during a loop of an algorithm form its *program state*, which is modelled as a record with one field for each variable. The loop is modelled as a recursive function that maps a state to the state in which the loop stops. For instance, depth-first search, `DFS` (L. 49), is a function on records of type `DFS_state` (L. 50), whose fields are a stack, a set of visited vertices, and a return value. Termination is not proved when such a function is defined: instead, Isabelle/HOL's function package [51] provides a domain predicate, e.g. `DFS_dom`, which holds for the states on which the function terminates, and we first prove the function partially correct for all states in its domain and only then prove that the states that satisfy the invariants are in the domain.

Locales are used in two ways when modelling algorithms. First, the data structures used by the algorithm are parameters of its locale, specified by the ADTs of Section 2.2. Thus, the algorithm is defined and verified once and can be executed with every implementation of these ADTs. Second, a sub-procedure

that the algorithm calls is a parameter of its locale, specified by assumptions that state what the sub-procedure computes, so that the algorithm can be verified before the sub-procedure is defined. For instance, the locale of the main loop of Edmonds' blossom algorithm (L. 51) fixes a function that searches for an augmenting path and assumes that this function finds such a path whenever one exists and that every path it returns is augmenting. This is stepwise refinement [7]: the assumed sub-procedures are defined and verified in later steps, until no assumption is left that is not satisfied by a definition.

Table 1 states, for every algorithm in the library, which properties of it are proved and the size of the theories that verify it, and Table 3 in Appendix A also states whether its loops are modelled with the methodology of Section 2.4, the number of invariants of its loops, and its main theorem.

2.4 Proofs about Algorithms

The induction principle that the function package derives for a function modelling a loop contains the function's entire control flow, making it hard to use in interactive proofs [2, Chapter 24]. Accordingly, for every execution path of the loop, we define a predicate stating that the path is taken and a function computing the state at the end of the path, e.g. `DFS_call_1_conds` (L. 83) and `DFS_upd1` (L. 84) for the path that pushes an unvisited vertex onto the stack. From these definitions we prove three lemmas that characterise the function only in terms of its execution paths: (1) a case analysis, `DFS_cases` (L. 85), stating that one of the paths is taken; (2) simplification rules, `DFS_simps` (L. 86), stating the result of the function for each path; and (3) a computation induction principle, `DFS_induct` (L. 87), whose induction step has one hypothesis for every recursive path, only mentioning the predicate and the update function of that path. For every path predicate, we also prove introduction and elimination rules, e.g. the elimination rule `DFS_call_1_conds` (L. 88), which turns the predicate into the facts about the state that it implies.

Invariants are predicates on the state, and, instead of one comprehensive loop invariant, we state many separate invariants, e.g. sixteen for the search for alternating paths in Edmonds' blossom algorithm [7], each of which is proved to be preserved assuming whatever other invariants are needed. Every invariant is first proved to be preserved by every recursive path, e.g. `invar_1_holds_1` (L. 89), which is where the mathematical reasoning about the algorithm takes place and where the reasoning principles of Section 2.1 are used. It is then lifted to the whole function by the computation induction principle, e.g. `invar_1_holds` (L. 90). These two kinds of lemmas are combined by Isabelle/HOL's classical reasoner and simplifier: we collect the introduction and elimination rules of the path predicates and of the invariants, and the preservation lemmas, in named collections of theorems (L. 91). They are given to the classical reasoner as introduction and elimination rules, so that e.g. the proof of `invar_1_holds` consists of the computation induction, the case analysis, and a single call of `auto`. Finally, termination is proved by a measure that decreases on every recursive path of a state that satisfies some of the invariants, and the correctness theorems,

e.g. `DFS_correct_1` (L. 53), follow from the invariants and the predicates of the paths on which the loop returns.

The loops of breadth-first search (L. 92), the best-in-greedy algorithm, which yields Kruskal's algorithm (L. 93), successive shortest paths (L. 94), capacity scaling (L. 95), the two inner loops of Orlin's algorithm (L. 96 and 97), the Hungarian method (L. 98), Dinic's algorithm (L. 99), and matroid intersection (L. 100) are verified in the same way, where each link points to the computation induction principle of the loop. However, Edmonds' blossom algorithm, which was the first algorithm in the library, uses path predicates only in its search for alternating paths and reasons about its other loops with the induction principles that the function package derives.

2.5 Imperative Implementations

So far, we have described functional algorithms as locales parameterised over their data structures and operations. However, imperative implementations are much faster than functional ones: on the BFS benchmarks of Section 3, the imperative export to SML is about two orders of magnitude faster than the functional one on the smallest grids, and about four orders of magnitude faster on the largest grids and on the road networks (Tables 9 and 12). One approach to verifying imperative code in Isabelle/HOL is Lammich's Isabelle Refinement Framework [66] with its tool Sepref [57]. The Refinement Framework, however, requires the abstract algorithm to be stated in its non-determinism error monad. Accordingly, using it would require restating every algorithm of the library in that monad, and would give up a property of our methodology: an algorithm is a recursive function in a locale whose correctness is proved by induction. We favour this approach because reading or extending the library needs no vocabulary beyond what an average Isabelle/HOL user already knows. The problem we had to solve was thus to refine our algorithms to imperative implementations without changing the locale-based methodology. Note that the monad models non-determinism, e.g. choosing an arbitrary element from a set, and failure. We solve this problem with an intermediate functional implementation: we first instantiate the functional locale with a functional data structure, e.g. a list, which fixes the choices that the abstract algorithm leaves open, and we then refine this deterministic functional implementation to an imperative one.

We use the separation logic framework for partial correctness in Imperative HOL (SL-Framework for short), which we originally developed for total correctness and extended to partial correctness to implement a WebAssembly interpreter [87]. For this project, we improved its automation, added concepts such as monadic folds and while-loops, and proved that Hoare triples are admissible for fixed-point induction over recursive partial functions in the heap monad. We also reuse its library of imperative data structures.

Imperative HOL shallowly embeds imperative programs into a heap-error monad, from which Isabelle's code generator produces actual imperative programs [20]. The SL-Framework adds a separation logic and a verification condition generator. Its central concept is the Hoare triple $\langle P\rangle\, c\, \langle \lambda x.\ Q\ x\rangle$, with

separation logic assertions P and Q. It expresses partial correctness: if P holds for the initial heap and c returns without errors, then $Q\ x$ holds for the result x and the final heap. An error means non-termination or a runtime failure, e.g. a failed bounds check.

Since we mainly verify an imperative implementation w.r.t. the functional one, all our assertions are *refinement assertions*. Such an assertion $A\ a\ c$ relates an abstract value a to a concrete one c that may refer to the heap. For example, `is_iam m a` (L. 101) states that `a` points to an array on the heap that implements the map `m` from natural numbers to values, the indices being the keys.

To refine a graph algorithm, we set up a locale that inherits from the algorithm's functional locale and fixes imperative versions of its data structures and subroutines, assuming that they are correct. Thus, the functional locale refines abstract concepts such as sets and graphs to functional implementations, and the imperative locale refines these further to imperative ones.

In DFS, for instance, the open choice is the next successor to process, which the set ADT provides by a choice operation (L. 102), and we implement vertex sets by distinct lists, which we refine differently depending on their use. Successor sets remain purely functional lists, as DFS only picks elements from them and iterates over them. The visited set, however, needs an efficient membership test, so we refine its list back to a set and then use any suitable imperative set implementation, e.g. bit-sets or hash-sets. We note that fixing an implementation, e.g. of sets using lists, is only a workaround.

Depending on the complexity of the algorithm and the intended flexibility, the imperative locale either partially instantiates the functional locale with functional data structures, or leaves all instantiations to the last step. In both cases, it is parameterised over the imperative implementations, which we choose when instantiating it. For DFS, for example, we generate one implementation with natural numbers as vertices and array-based maps and sets (L. 103), and one based on hash maps and hash sets (L. 104).

We define a refinement assertion for the state, typically as the product of the refinements of the state components. We then implement refinements of the functions of the functional version, i.e. some helpers and the main loop, and prove that they indeed refine their functional counterparts: on related arguments, they return related results. The loop is implemented as a partial function [50,51] in Imperative HOL's heap monad, and we prove by fixed-point induction that it refines the tail-recursive partial function of the functional version. For DFS, we prove (L. 105) that, on implementations of a graph and a state, the imperative DFS leaves the graph unchanged and returns an implementation of the functional DFS's result state. The proof is largely canonical: inside the induction, we unfold one step of the functional implementation and split it into its conditional branches; for each branch, we unfold the imperative implementation, eliminate its other branches, and prove the refinement with the verification condition generator. This step can be largely automated.

Finally, we combine both refinements into a Hoare triple stating that the imperative algorithm satisfies the abstract specification, again in a canonical

and partially automated way. For example, from the result of the imperative DFS, we obtain a walk to the target or the absence of one (L. 106).

2.6 Other Methodological Points

Algorithms on abstract structures. Some algorithms are verified on abstract combinatorial structures and then instantiated to graphs, so that one correctness proof serves several algorithms. For instance, Kruskal's algorithm is an interpretation of the locale of the best-in-greedy algorithm on matroids, with the forests of the input graph as the independent sets (L. 107), and maximum cardinality bipartite matching is computed by matroid intersection, where the independent sets of the two matroids are the sets of edges that share no vertex on the left and on the right side, respectively (L. 108).

Verified algorithms as building blocks. Verified algorithms are also used as concrete sub-procedures of other algorithms, in contrast to the sub-procedures that are locale parameters (Section 2.3). For instance, the independence oracle of Kruskal's algorithm runs the verified DFS to decide whether an edge closes a cycle in the current forest (L. 109), and the proof of the oracle uses the correctness theorems of DFS.

Verified reductions. We verify reductions between problems, so that an algorithm for one problem solves another. For instance, we prove that every minimum-cost flow problem with capacities can be transformed into an uncapacitated one, such that flows of the two problems correspond, costs are preserved, and optimal flows are mapped to optimal flows (L. 110). Accordingly, Orlin's algorithm, which is verified for uncapacitated problems [3], also solves problems with capacities (L. 111).

Shared optimality theorems. We prove the characterisations of optimal solutions separately from the algorithms and share them between algorithms, so that the proof of an algorithm only needs to show the condition of the characterisation. For instance, Berge's lemma [16], i.e. that a matching has maximum cardinality if and only if there is no augmenting path (L. 25), is used in the proofs of the blossom algorithm, the Hungarian method, and RANKING, and the characterisation of minimum-cost flows by the absence of augmenting cycles of negative cost (L. 112) is used in the proofs of successive shortest paths, capacity scaling, and Orlin's algorithm.

Running-time analysis. We analyse running times in a locale that extends the locale of the algorithm with the costs of the ADT operations as parameters. It defines a function with the same recursion as the algorithm, which returns the time along with the result. For Orlin's algorithm, we prove in such a locale (L. 114) that the time is bounded by a sum of products of the operation costs and terms in the number of vertices, and that the result is that of the algorithm (L. 115). Accordingly, the algorithm and its correctness proof are not changed by the analysis.

Certification. In addition to verified algorithms, we use verified code to check, and if necessary to clean up, the output of an unverified solver, which is the

approach of LEDA's certifying algorithms [70], except that the checker itself is verified [11].

3 Empirical Evaluation

The library contains many verified implementations of graph algorithms and, given the space limitations, we focus on evaluating those for the main problems: 1. depth-first (DFS) and breadth-first search (BFS) for reachability, 2. Dijkstra's algorithm for single-source shortest paths, 3. Kruskal's algorithm [52] for minimum spanning forests, 4. push-relabel (PR) for maximum flow [34], 5. the Hungarian method (HM) [40,53,54] for minimum-weight perfect bipartite matching (minPBM), 6. (weighted) bipartite (b-)matching, with matroid intersection (MI) [67,29], and 7. hybrid solvers for minPBM and minimum-cost flows. Our focus is on running time: we compare the code exported from our verified implementations against unverified implementations in widely used graph libraries, such as SciPy (Cython), Sage and NetworkX [84,23,83] (Python), JGraphT [72] (Java), LEMON [24] (C++), which is widely recognised as one of the fastest graph libraries, and against reference solvers recommended for the DIMACS challenges, from which most of the problem families are also taken.

We export code to SML, Scala, and Go using Isabelle's code generator [37] and its Go extension [82], which we extended to export Imperative HOL to Go. The exceptions are the hybrid solvers, whose verified part we export only to SML. We export to 64-bit integers and compile SML with `mlton` with high inlining. We conduct the measurements on an AMD EPYC 7343 (Milan, Zen 3) with a timeout of 1200 s. The runs are scheduled to share the RAM but not the cache hierarchy. Because of the graphs' sizes, we only take between 3 and 10 instances per size, run each instance 3 times, take the median for each instance, and average over all instances.

If a library can combine building data structures or graph representations with input file parsing, we count the respective time as parsing, which is excluded from the measurement. If the data representation has to be built separately from parsing, as in the case of our verified code, it is included in the measurement. Except for Kruskal's algorithm, all our imperative implementations scan graphs by neighbourhoods, and we use the cache-efficient CSR representation. Despite the cost of CSR buildup being an additional $\mathcal{O}(n + m)$, we generally use it since our implementations of simpler algorithms, which themselves are $\mathcal{O}(n+m)$ or close, are intended as repeatedly executed sub-procedures of more complex algorithms, which amortises the cost. Note the convention that n denotes the number of vertices and m denotes the number of edges in a graph.

We present plots (running time in seconds against the number of vertices n on a log-log scale) and tables, with the latter in the appendix. Our code exports' plots usually have the same slope, with the vertical shift accounting for the constant factor from language differences. SML beats Scala, which in turn beats Go, and imperative implementations beat functional ones by orders of magnitude (OoM). Imperative SML, which is always our best-performing variant and which

we focus on, is plotted as ⊸. Generally, the gap to the top-performing reference implementations ('the libraries') is much smaller for simple algorithms, probably because there is less room for heuristics and low-level optimisations.

DFS and BFS. We let the solvers explore all vertices reachable from a random source on three families of graphs. Figure 2 shows a performance difference of OoMs between imperative (with CSR) and purely functional (with RBTs) code exports. For DFS, we also include a semi-imperative export with an adjacency list representation, whose performance is in between but closer to CSR-based (RBT-based) on sparse (dense) graphs. For the libraries, we measure both with a native graph representation that can be built during parsing and with the more efficient CSRs. SciPy's native representation is CSR, and here we have measurements with and without its buildup: the cost of buildup is clearly visible on dense graphs, where it is also asymptotically highest. Since the CSRs are only used once, CSR-based implementations of DFS and BFS generally perform worse in comparison to those where the representation is built during parsing. For sparse graphs, our imperative SML code with CSRs is on par with the libraries, and is within an OoM for the dense family. If the libraries are also charged with the cost of building CSRs, it seems that we can beat them by a small margin.

Dijkstra's (DA) and Kruskal's Algorithms (KA). Figure 2 shows that our SML is still within an OoM of the best library or narrows to it at scale. For DA, the difference is bigger than it was for DFS/BFS, which we attribute to the higher implementation complexity of the algorithm, in particular to the priority queue. For KA, where we reuse an implementation of a union-find data structure by Stevens and Ghidini [81], our SML is also well within an OoM of the best library's performance.

Push-Relabel (PR). Apart from the libraries, we evaluate our PR exports with highest-label (HL) and FIFO selection heuristics, an export of the older formalisation [65] (AFP) with FIFO, and Goldberg and Cherkassky's implementation [21] (GC), which has been tuned for decades. SciPy does not offer PR, so we use its implementation of Dinic's algorithm instead.

The algorithm has ample room for heuristics, e.g. all implementations use the gap heuristic [9,21], and Fig. 2 seemingly shows the effect: on RMF and WASH-RLG [43], JGraphT and NetworkX, both much slower for the simpler algorithms from above, beat (or start to beat at scale) both of our SML exports, which hit the timeout. In contrast to our implementation (and to LEMON), they offer an effective heuristic: BFS-based global relabelling (GL). Our two code exports offer HL and FIFO, of which HL is known to usually perform a bit better [9,21]. GL seems to be good enough to compensate for JGraphT's FIFO heuristic and the language difference. On RMF and WASH-RLG, we only beat the code export from the older formalisation [65]. The latter does not even work at larger sizes because of an overflow during the buildup of its memory-consuming data structures (adjacency matrices). Our code performs asymptotically and in terms of OoM far worse than GC. By design, the AK family affects the gap heuristic [21], which interacts with GL [21], and GL seems to become much less effective, which might reduce LEMON's disadvantage of lacking it. LEMON's

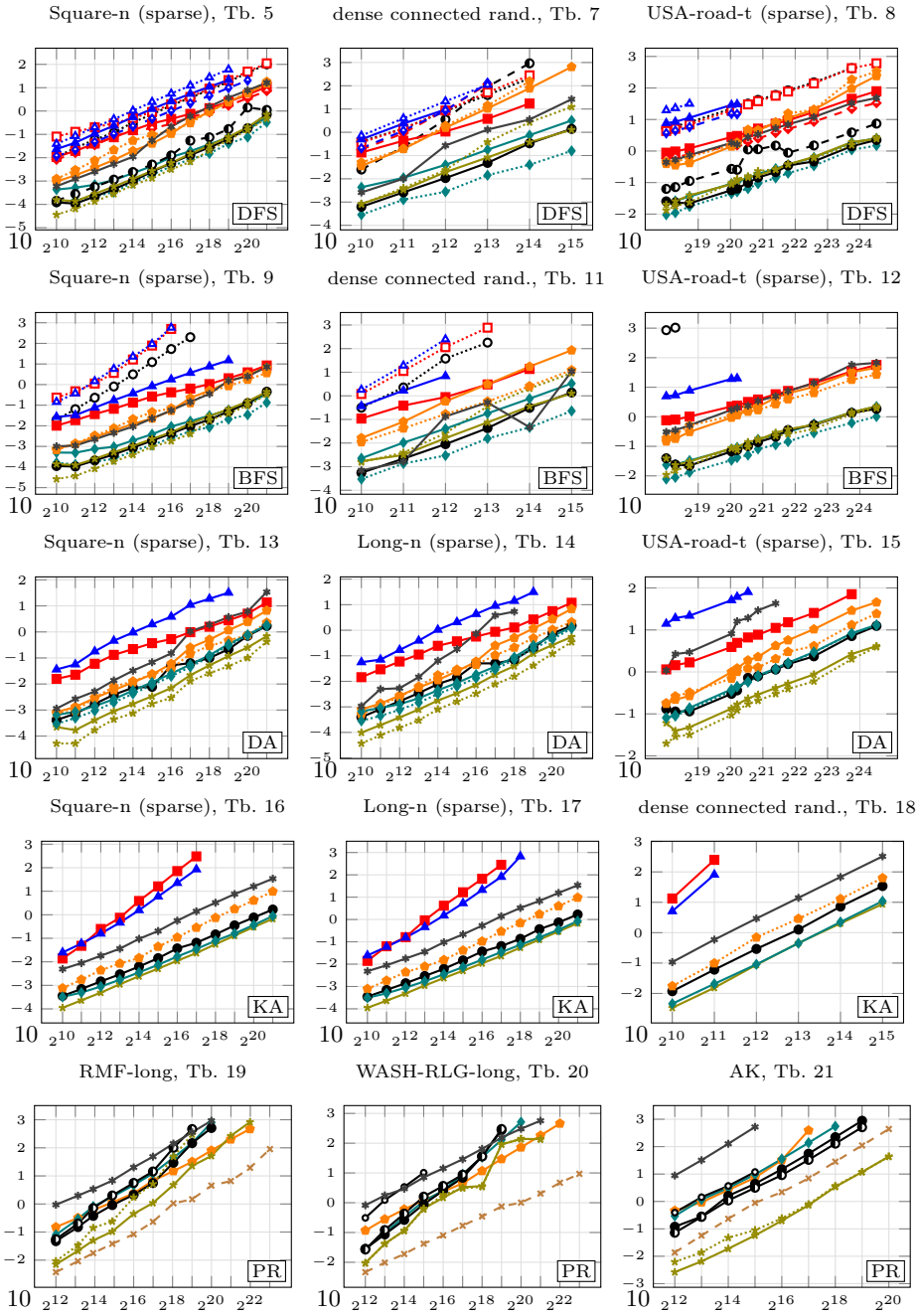


Fig. 2. SML CSR, SML adjacency lists, SML RBT, Scala CSR, Scala adjacency lists, Scala RBT, Go CSR, Go adjacency lists, Go RBT, LEMON CSR, LEMON linked, JGraphT CSR, JGraphT hash, SciPy CSR, SciPy prebuilt, NetworkX. PR: SML HL, SML FIFO, SML AFP, JGraphT, SciPy, GC.

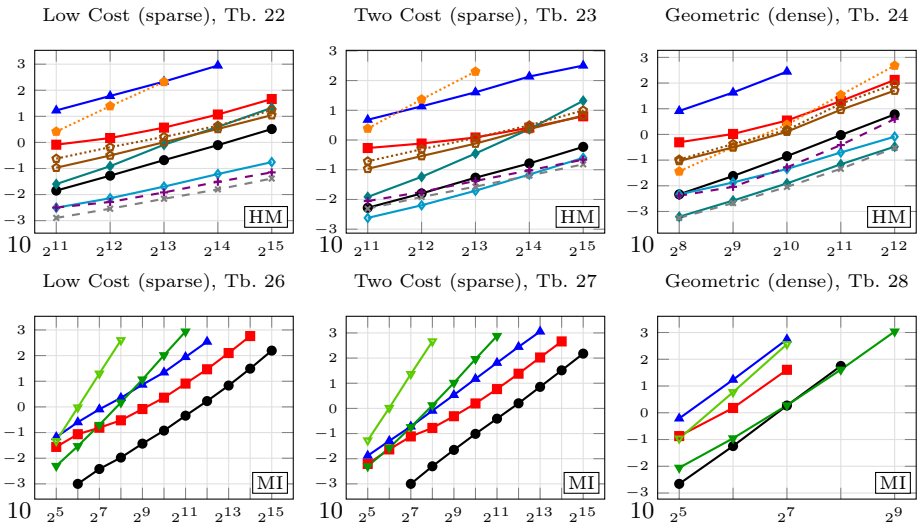


Fig. 3. SML, Scala, Go, SciPy LSA, SciPy MWFBM CSR, JGraphT KM, JGraphT MWBM CSR, JGraphT MWBM hash-backed, CSA CSR, OR-Tools, NetworkX wrapps Scipy, skipped. Sage (our oracle), Sage (plain).

plot is parallel to those of our SML exports and two OoMs below them, which indicates a large constant factor.

We plan to add GL to our PR, leading to the combination of gap heuristic, GL and HL, which is known for its strong performance [21]. We expect a significant speedup since GL is seemingly able to make NetworkX, which is pure Python, beat compiled SML.

Hungarian method (HM). Our HM for minPBM is optimised for sparse graphs, and we borrow a few, but not all, practical optimisations from Jonker and Volgenant's (JV) high-performing version [44]. We compare against SciPy's JV for dense (LSA) and sparse adjacency matrices (MWFBM), JGraphT's dense HM (KM) and sparse HM (MWBM), Goldberg and Kennedy's CSA solver [32], and Google's OR-Tools [76] (OR). The latter two implement cost-scaling algorithms. We use the families *low-cost (LC)*, *two-cost (TC)*, and *geometric (GEO)* [32].

In Fig. 3, our HM in SML is within two OoMs of the best three solvers, and for TC, even within one, but the effect does not seem to scale for LC. We think that the smaller gap for TC is due to 1. more productive primal-dual adjustments because there are only two cost values, and 2. more scaling phases because one of the two costs is large. Our measurements for PR and HM confirm the higher difficulty of producing verified implementations of complex and heuristic-heavy algorithms, especially ones that are somewhat competitive with respected libraries that have often been tuned for decades.

Matroid Intersection (MI). Matroids are used to model b-matching, in which every vertex v can be matched up to $b(v)$ times. The abstract algorithms obtain information about the concrete problem by *oracles*. One iteration of our MI takes $\mathcal{O}(m^2)$ oracle calls. We offer constant-time oracles to model bipartite matching as MI. Sage also offers unweighted and weighted MI in pure Python, though the weighted one uses a different algorithm. The pure algorithms also take $\mathcal{O}(m^2)$ oracle calls per iteration, and internal glue code adds a factor of $\mathcal{O}(m)$. An oracle built from Sage's primitives takes $\mathcal{O}(m)$ time, and our own customised one takes constant time, resulting in $\mathcal{O}(m^4)$ and $\mathcal{O}(m^3)$ per iteration, respectively. Bipartite matching as MI takes $\mathcal{O}(B \cdot \iota)$ time, where ι is the time of one iteration and $B = \sum \mathsf{min}\{\deg(v), b(v)\}$.

We benchmark the resulting solvers for (un-)weighted (b-)matching on LC, TC, and GEO. Figure 3 shows the results (see Fig. 6 in the appendix for more). The figures mostly seem to confirm the asymptotics. TC tends to be solved a bit faster than LC, which we attribute to the simpler weight structure. Unexpectedly, Sage instantiated with the faster Python oracle improves relatively for the dense GEO family, perhaps due to the weight variance. Usually, SML beats the fast Python implementation by OoMs, but for weighted matching on GEO, they are on par and SML times out for the biggest instance. However, for weighted b-matching (see Fig. 6 in the appendix), SML changes little and Sage falls behind, which must be the effect of allowing multiple matching edges per vertex.

Hybrid Solvers. We offer certificate checking for minimum-cost flows (MCF) and weighted bipartite matching (WBM). In combination with the verified network simplex and HM, respectively, we build hybrid solvers that first let an unverified program ('oracle') solve the problem and generate a certificate. If the verified checker rejects the certificate, we use our verified network simplex or HM, respectively, to solve the problem. In contrast to HM, the network simplex can be warm-started, i.e. it is initialised with the unverified solver's rejected solution. Both checkers examine a simple property of every edge, in $\mathcal{O}(n + m)$ time.

For example, by extending implementations of different algorithms from LEMON with certificate generation, we obtain four oracles for MCFs from which the user can select. The *cost ratio of verification (CV)* is, for the fastest oracle, the ratio between the total CPU time (which includes certificate generation and checking) and the time of the oracle's initialisation and main loop. Table 2 shows the results. On `netgen_deg`, where for fixed n the density grows up to n^2, the CV increases up to 3.69. Since both oracle and checker have to deal with bigger graphs, a roughly constant CV would be expected, or even a decreasing one since no oracle's running time is linear. These are worst-case running times, however, and we believe the CV increases because, as the graph grows, for a fixed increase in the checker's time, the oracle has the same or better chances to route the flow. For growth in n and moderate or no growth in density (`netgen_lo_sr` and `road_paths`), the CV decreases as expected. We benchmark on 18 families, and 3.69 is the highest CV that we measured for any family and size.

Table 2. CV derived from Tables 30, 31, 32, 35, 36, 37 in the appendix.

Family	Size	$\setminus j$	1	2	3	4	5	6	7	8	9 ...	12
`netgen_deg`	$n = 4096$, $m = 2^j n$		1.18	1.24	1.28	1.31	1.50	1.68	1.88	2.23	2.61 ...	3.69
`netgen_lo_sr`	$n = 2^{j+7}$, $m = n^{1.5}$		3.17	3.34	3.40	2.94	2.96	2.38	2.22	1.89	1.80	
`road_paths`	$n \approx 1.5 \cdot 10^4 \cdot 2^j$, $m \approx 2.4n$			1.91	1.63	1.40	1.22	1.29	1.16			
LC	$n = 2^{j+8}$, $m = n \log_2 n$				1.72	1.68	1.61	1.62	1.65	1.64	1.51 ...	1.26
TC	$n = 2^{j+8}$, $m = n \log_2 n$				1.18	1.16	1.16	1.16	1.18	1.18	1.10 ...	1.05
GEO	$n = 2^{j+7}$, $m = n^2/4$		2.21	2.35	2.43	2.36	2.13	1.97	1.69			

Table 2 also gives the CV of the hybrid solver for minPBM that combines the CSA solver [32] with our verified certificate checker. We use the same families as for HM, in which the graphs grow in both n and m, and observe the same behaviour as for `netgen_lo_sr` and `road_paths`: the CV decreases at scale. For LC and TC, the certificate checker does the same amount of work, but since CSA does worse on TC, the CV is smaller. On the dense GEO family, the CV is much higher since the greater density increases the checker's relative work.

Although hybrid solvers seem to perform almost by design within the lower half of an order of magnitude of unverified ones, we cannot generally reduce the CV to 1. Still, the hybrid approach is very practical, especially for dense instances, where a fully verified but slower main loop would hurt most.

4 AI-Assisted Development

Our verification methodology developed on a small number of algorithms: the graph representation and modelling loops as tail-recursive functions for the blossom algorithm [7], and the graph ADTs, the invariant reasoning, and the imperative refinement for depth- and breadth-first search [2]; it was then refined for minimum-cost flows [3], algorithms for matroids and greedoids [5], and primal-dual algorithms [4]. Our initial intention behind this methodology was pedagogical, e.g. using basic Isabelle/HOL formalisms like locales and recursive functions, instead of complex formalisms like non-determinism monads. It turns out that, according to our experience, LLM-generated algorithms and specifications, when done according to this methodology, are easier to read and to review than what an LLM would output by default. This mitigates the well-documented difficulty of ensuring that LLM-generated definitions, code, and theorem statements are aligned with the user's request [90,8,69]. We used two methods to improve this alignment, one for definitions and specifications, and one for proofs.

The first method is standard: the LLM works in phases, and we review its design and implementation decisions, in the form of an LLM-generated plan, before it implements them [42,13,41,80]. At that stage, we remove the most obvious flaws, e.g. wrong definitions or specifications, reinvented definitions, lemmas that are not reused, and deviations from our formalisation idioms.

For proofs, alignment is less a matter of correctness than of readability. This matters e.g. to understand the correctness argument when one wants to further

improve an algorithm. Since in our library proofs are more than eight times as long as the definitions and specifications, reviewing all LLM-generated proofs is infeasible. Instead, we compute a statistical signature of our own proofs, e.g. the share of calls to automated Isabelle proof methods, such as `auto`, among all proof steps, and prompt the LLM to restyle its already working proofs towards that signature, since LLMs write verbose proofs with many low-level steps [36,10]. For instance, in the LLM-generated proofs of the Hopcroft-Karp algorithm, `auto` accounts for 10.7% of the proof method calls, against 61.2% and 50.7% in our proofs of depth- and breadth-first search, and 38.0% of the lemmas have a single-step automated proof, against 76.5% and 63.1%; after restyling, 69.2% of Hopcroft-Karp's proofs have a single-step proof (Table 39 in Appendix C.1). To our knowledge, this is the first use of a statistical signature of one's own verified proofs to guide the restyling of LLM-generated proofs.

In addition to alignment, another concern we had related to using LLMs to generate proofs was attribution. We label parts of theory files at the lowest granularity we believe is practical: we label sections of Isabelle/HOL files as developed by humans, 'collaboratively' with an LLM, or by an LLM. We label a part as collaborative, e.g. if a human prompted the LLM in detail on how to formalise a concept, revised it, or manually formalised parts of it. Parts labelled as produced by an LLM alone are mostly auxiliary lemmas and low-level proofs. Counting the lines labelled that way, more than 55% of the lines of the library were developed purely by humans (Table 38 in Appendix C). This granularity gives data for further studies of the formalisation styles of LLMs and humans, something which we plan to do in the future.

5 Discussion

Graph libraries. In unverified graph libraries, such as LEDA [70], LEMON [24], Boost [79], and JGraphT [72], algorithms are written against an interface of graph operations and work with any implementation of that interface. Our ADT locales have the same role, and every interface has a specification, against which the correctness of every algorithm that uses it is proved. Algorithms in LEDA return the solution alongside a certificate that a simple checker program tests, which moves the trust to the checker, which can itself be wrong [11]. The graph library for Isabelle/HOL by Noschinski [75] and the graph theory in Lean's mathlib [1] formalise the mathematical theory of graphs without executable algorithms. Verification, on the other hand, has mostly been applied to one or a small family of algorithms at a time, each development with its own graph theory and representation, e.g. the Ford–Fulkerson method in Mizar [68], various graph algorithms in Isabelle/HOL [65,38,62], and C implementations in Coq [73]. In contrast, all algorithms in our library are stated in terms of the same few representations, which are connected by adaptors (Section 2.1).

Refinement. Obtaining efficient verified implementations by stepwise refinement [88] and data refinement [39] is an established approach in Isabelle/HOL. Its most developed instance is Lammich's Isabelle Refinement Framework [66,55],

which was used for graph algorithms, too [65,38]. Our approach differs: 1. instead of a nondeterministic monad, we write algorithms purely functionally, 2. instead of automated data refinement to the Isabelle Collections Framework [59,60], we apply data refinement by locale interpretations [14], and 3. the resulting functional implementation is the intermediate step from which the imperative implementation is refined (Section 2.5). Our functional implementations have to be deterministic, but we can state them as ordinary recursive functions and conduct proofs in the style of textbooks on verified functional algorithms [2]. Alternatively, others verify the low-level code that is actually run, either directly, e.g. C implementations of graph algorithms [73], or by abstracting C programs to higher-level specifications [77,35].

LLM-based development of formal libraries. Theorem proving with LLMs has mostly been studied as proving individual theorems against a fixed library, using a variety of techniques [28,41,89,86,80,91,69]. Developing a library aided by AI is a different task, since much of the work is in choosing definitions, representations, and the statements of lemmas so that later developments can reuse them, and none of these choices is checked by the proof checker. In our development (Section 4), this is where the decisions of the LLM had to be reviewed, and the methodology of Section 2 gave that review a fixed reference: a definition, a specification, or a proof that does not follow its idioms is easy to identify.

The restyling of LLM-generated proofs in Section 4 is related to recent work on automated proof optimisation. ImProver [10] and ProofOptimizer [36] optimise Lean proofs that already check w.r.t. a single criterion, e.g. length. Our guideline is derived from a statistical signature of our own verified proofs, and the aim of the restyling is that the LLM-generated proofs match the style of those proofs, rather than that they minimise one measure. Measuring style is not new [18], but to our knowledge, it has not previously been used to guide or to evaluate LLM-generated proofs.

Conclusion. We presented a library of formally verified graph algorithms in Isabelle/HOL. For simple algorithms, we are within an order of magnitude of standard libraries such as SciPy [84] or LEMON [24], or even on par, whereas for more complex algorithms, which come with much more room for heuristics and tuning, it is harder as our code is sometimes two orders of magnitude slower. We plan to add more optimisations, which come with more low-level details, but the effort should be greatly reduced by AI. Keeping the library's style, especially maintaining the abstraction barriers, is a challenge. Also, some of the complex implementations in the libraries we benchmarked against have been tuned for decades, whereas we started including efficient code less than a year ago.

The algorithms share one methodology, so that a lemma or a data structure developed for one algorithm is available to the others. The methodology also allowed us to develop about a third of the library collaboratively with LLMs while still being able to review what they produced. Directions for future work include: 1. proving running-time bounds for all algorithms of the library in the model of Section 2.6, as we did for Orlin's algorithm, and relating them to a

concrete model of computation using our framework [45], and 2. refining the imperative implementations further to LLVM [56,58].

Data Availability

The supporting material contains our formal development, the style guideline that we gave to the LLM when it restyled its proofs, and the two scripts that compute the style statistics of Table 39 from the theory sources (Appendix C.1). It also contains an index theory file, which identifies, for every listing in the appendix, where its formal counterpart is in the library. If the paper is accepted, we will archive the formal development on Software Heritage, and instead of citing listings in the appendix from the body of the paper, we will link to permalinks into that archive.

References

1. Leanprover-community/mathlib4. leanprover-community (2024), https://github.com/leanprover-community/mathlib4
2. Functional Data Structures and Algorithms: A Proof Assistant Approach. 1 edn. (2025)
3. Abdulaziz, M., Ammer, T.: A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows. In: 15th Int. Conf. Interact. Theorem Proving ITP (2024). https://doi.org/10.4230/LIPICS.ITP.2024.3
4. Abdulaziz, M., Ammer, T., Madlener, C.: Formal Primal-Dual Algorithm Analysis. In: 17th Int. Conf. Interact. Theorem Proving ITP. LIPIcs (2026). https://doi.org/10.4230/LIPIcs.ITP.2026.33
5. Abdulaziz, M., Ammer, T., Meenakshisundaram, S., Rimpapa, A.: A Formal Analysis of Algorithms for Matroids and Greedoids. In: 16th Int. Conf. Interact. Theorem Proving ITP (2025). https://doi.org/10.48550/arXiv.2505.19816
6. Abdulaziz, M., Madlener, C.: A Formal Analysis of RANKING. In: 14th Conf. Interact. Theorem Proving ITP (2023). https://doi.org/10.48550/arXiv.2302.13747
7. Abdulaziz, M., Mehlhorn, K.: A Formal Correctness Proof of Edmonds' Blossom Shrinking Algorithm. Accept. Publ. J. Autom. Reason. (2025)
8. Aggarwal, P., Parno, B., Welleck, S.: AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and treefinement. In: 42nd Int. Conf. Mach. Learn. ICML. Proc. Mach. Learn. Res., vol. 267, pp. 587–615 (2025)
9. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall (1993)
10. Ahuja, R., Avigad, J., Tetali, P., Welleck, S.: ImProver: Agent-based automated proof optimization. In: 13th Int. Conf. Learn. Represent. ICLR (2025)
11. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A Framework for the Verification of Certifying Computations. J. Autom. Reason. (2014). https://doi.org/10.1007/S10817-013-9289-2
12. Baier, C., Katoen, J.P.: Principles of Model Checking (2008)
13. Bairi, R., Sonwane, A., Kanade, A., C, V.D., Iyer, A., Parthasarathy, S., Rajamani, S., Ashok, B., Shet, S.: CodePlan: Repository-level coding using LLMs and planning. Proc. ACM Softw. Eng. **1**(FSE), 675–698 (2024). https://doi.org/10.1145/3643757
14. Ballarin, C.: Locales: A Module System for Mathematical Theories. J. Autom. Reason. (2014). https://doi.org/10.1007/s10817-013-9284-7
15. Bellman, R.: On a routing problem. Q. Appl. Math. (1958). https://doi.org/10.1090/qam/102435

16. Berge, C.: Two theorems in graph theory. Proc. Natl. Acad. Sci. (1957). https://doi.org/10.1073/pnas.43.9.842
17. Biró, P., van de Klundert, J., Manlove, D., Pettersson, W., Andersson, T., Burnapp, L., Chromy, P., Delgado, P., Dworczak, P., Haase, B., Hemke, A., Johnson, R., Klimentova, X., Kuypers, D., Nanni Costa, A., Smeulders, B., Spieksma, F., Valentín, M.O., Viana, A.: Modelling and optimisation in European Kidney Exchange Programmes. Eur. J. Oper. Res. (2021). https://doi.org/10.1016/j.ejor.2019.09.006
18. Blanchette, J.C., Haslbeck, M., Matichuk, D., Nipkow, T.: Mining the Archive of Formal Proofs. In: Intell. Comput. Math. CICM. LNCS, vol. 9150, pp. 3–17 (2015). https://doi.org/10.1007/978-3-319-20615-8_1
19. Bonet, B., Geffner, H.: Planning as heuristic search. Artif. Intell. (2001). https://doi.org/10.1016/S0004-3702(01)00108-4
20. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with isabelle/hol. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics. pp. 134–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
21. Cherkassky, B.V., Goldberg, A.V.: On implementing the push-relabel method for the maximum flow problem. Algorithmica **19**(4), 390–410 (1997). https://doi.org/10.1007/PL00009180, hi_pr 3.5
22. Demetrescu, C., Goldberg, A.V., Johnson, D.S. (eds.): The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 74. American Mathematical Society (2009). https://doi.org/10.1090/dimacs/074
23. Developers, T.S.: The Abstract Matroid Class - Matroid Theory, https://doc.sagemath.org/html/en/reference/matroids/sage/matroids/matroid.html, last accessed on 25/03/2025
24. Dezső, B., Jüttner, A., Kovács, P.: LEMON – an open source C++ graph template library. Electron. Notes Theor. Comput. Sci. **264**(5), 23–45 (2011). https://doi.org/10.1016/j.entcs.2011.06.003
25. Edmonds, C.: Undirected Graph Theory. Arch. Formal Proofs (Sep 2022), https://isa-afp.org/entries/Undirected_Graph_Theory.html
26. Edmonds, C.: Hypergraphs. Arch. Formal Proofs (Sep 2023), https://isa-afp.org/entries/Hypergraph_Basics.html
27. Edmonds, J.: Paths, Trees, and Flowers. Can. J. Math. (1965/ed). https://doi.org/10.4153/CJM-1965-045-4
28. First, E., Rabe, M.N., Ringer, T., Brun, Y.: Baldur: Whole-Proof Generation and Repair with Large Language Models. In: Proc. 31st ACM Jt. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (2023). https://doi.org/10.1145/3611643.3616243
29. Frank, A.: A Weighted Matroid Intersection Algorithm. J. Algorithms (1981). https://doi.org/10.1016/0196-6774(81)90032-8
30. Gallai, T.: Kritische Graphen II. Magyar Tud. Akad. Mat. Kutató Int. Közl. **8**, 373–395 (1963)
31. Gallai, T.: Maximale Systeme unabhängiger Kanten. Magyar Tud. Akad. Mat. Kutató Int. Közl. **9**, 401–413 (1964)
32. Goldberg, A.V., Kennedy, R.: An efficient cost scaling algorithm for the assignment problem. Math. Program. **71**(2), 153–177 (1995). https://doi.org/10.1007/BF01585996, CSA 1.2
33. Goldberg, A.V., Kharitonov, M.: On implementing scaling push-relabel algorithms for the minimum-cost flow problem. In: Johnson, D.S., McGeoch, C.C. (eds.) Network Flows and Matching: First DIMACS Implementation Challenge, DIMACS

Series in Discrete Mathematics and Theoretical Computer Science, vol. 12, pp. 157–198. American Mathematical Society (1993). https://doi.org/10.1090/dimacs/012/07
34. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM **35**(4), 921–940 (1988). https://doi.org/10.1145/48014.61051
35. Greenaway, D., Andronick, J., Klein, G.: Bridging the Gap: Automatic Verified Abstraction of C. In: Interact. Theorem Proving (2012). https://doi.org/10.1007/978-3-642-32347-8_8
36. Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., Markosyan, A.H.: ProofOptimizer: Training language models to simplify proofs without human demonstrations. In: 14th Int. Conf. Learn. Represent. ICLR (2026)
37. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In: Funct. Log. Program. FLOPS. pp. 103–117 (2010). https://doi.org/10.1007/978-3-642-12251-4_9
38. Haslbeck, M.P.L., Lammich, P., Biendarra, J.: Kruskal's Algorithm for Minimum Spanning Forest. Arch. Formal Proofs (Feb 2019), https://isa-afp.org/entries/Kruskal.html
39. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica (1972). https://doi.org/10.1007/BF00289507
40. Jacobi, C.G.: De investigando Ordine Systematis Aequationum Differentialum vulgarium cuiuscumque. Journal für die reine und angewandte Mathematik **64**, 297–320 (1890), https://www.lix.polytechnique.fr/~ollivier/JACOBI/Jacobi_De_Investigando.pdf
41. Jiang, A.Q., Welleck, S., Zhou, J.P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., Lample, G.: Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In: 11th Int. Conf. Learn. Represent. ICLR (2023)
42. Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., Jiao, W.: Self-planning code generation with large language models. ACM Trans. Softw. Eng. Methodol. **33**(7), 1–30 (2024). https://doi.org/10.1145/3672456
43. Johnson, D.S., McGeoch, C.C. (eds.): Network Flows and Matching: First DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 12. American Mathematical Society (1993). https://doi.org/10.1090/dimacs/012
44. Jonker, R., Volgenant, A.: A shortest augmenting path algorithm for dense and sparse linear assignment problems. Computing **38**(4), 325–340 (1987). https://doi.org/10.1007/BF02278710
45. Kappelmann, K., Huch, F., Stevens, L., Abdulaziz, M.: Proof-Producing Translation of Functional Programs into a Time & Space Reasonable Model (2025). https://doi.org/10.48550/arXiv.2503.02975
46. Klein, M.: A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems. Manag. Sci. (1967). https://doi.org/10.1287/mnsc.14.3.205
47. Klingman, D., Napier, A., Stutz, J.: NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. Manag. Sci. **20**(5), 814–821 (1974). https://doi.org/10.1287/mnsc.20.5.814
48. Korte, B., Vygen, J.: Combinatorial Optimization (2012). https://doi.org/10.1007/978-3-642-24488-9
49. Kovács, P.: Minimum-cost flow algorithms: An experimental evaluation. Optim. Methods Softw. **30**(1), 94–127 (2015). https://doi.org/10.1080/10556788.2014.895828

50. Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) Automated Reasoning. pp. 589–603. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
51. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis, Technical University Munich (2009), http://mediatum2.ub.tum.de/doc/681651/document.pdf
52. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. (1956). https://doi.org/10.1090/S0002-9939-1956-0078686-7
53. Kuhn, H.W.: The Hungarian method for the assignment problem. Nav. Res. Logist. Q. (1955). https://doi.org/10.1002/nav.3800020109
54. Kuhn, H.W.: Variants of the Hungarian method for assignment problems. Nav. Res. Logist. Q. **3**(4), 253–258 (1956). https://doi.org/10.1002/nav.3800030404
55. Lammich, P.: Refinement for Monadic Programs. Arch Form. Proofs (2012), https://www.isa-afp.org/entries/Refine_Monadic.shtml
56. Lammich, P.: Generating Verified LLVM from Isabelle/HOL. In: 10th Int. Conf. Interact. Theorem Proving ITP 2019 (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22
57. Lammich, P.: Refinement to Imperative HOL. J. Autom. Reason. (2019). https://doi.org/10.1007/s10817-017-9437-1
58. Lammich, P.: Refinement of Parallel Algorithms down to LLVM. In: 13th Int. Conf. Interact. Theorem Proving ITP (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.24
59. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Interact. Theorem Proving (2010). https://doi.org/10.1007/978-3-642-14052-5_24
60. Lammich, P., Lochbihler, A.: Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches. J Autom Reasoning (2019). https://doi.org/10.1007/s10817-018-9461-9
61. Lammich, P., Meis, R.: A Separation Logic Framework for Imperative HOL. Arch Form. Proofs (2012), https://www.isa-afp.org/entries/Separation_Logic_Imperative_HOL.shtml
62. Lammich, P., Nipkow, T.: Proof Pearl: Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra. In: 10th Int. Conf. Interact. Theorem Proving ITP (2019). https://doi.org/10.4230/LIPICS.ITP.2019.23
63. Lammich, P., Nipkow, T.: Purely Functional, Simple, and Efficient Implementation of Prim and Dijkstra. Arch. Formal Proofs (Jun 2019), https://isa-afp.org/entries/Prim_Dijkstra_Simple.html
64. Lammich, P., Sefidgar, S.R.: Flow Networks and the Min-Cut-Max-Flow Theorem. Arch. Formal Proofs (Jun 2017), https://isa-afp.org/entries/Flow_Networks.html
65. Lammich, P., Sefidgar, S.R.: Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. J. Autom. Reason. (2019). https://doi.org/10.1007/s10817-017-9442-4
66. Lammich, P., Tuerk, T.: Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In: Interact. Theorem Proving (2012). https://doi.org/10.1007/978-3-642-32347-8_12
67. Lawler, E.L.: Matroid Intersection Algorithms. Math. Program. (1975). https://doi.org/10.1007/BF01681329
68. Lee, G.: Correctnesss of ford-fulkerson's maximum flow algorithm1. Formaliz. Math. (2005)

69. Lu, J., Wan, Y., Huang, Y., Xiong, J., Liu, Z., Guo, Z.: FormalAlign: Automated alignment evaluation for autoformalization. In: 13th Int. Conf. Learn. Represent. ICLR (2025)
70. Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing (1999), http://www.mpi-sb.mpg.de/%7Emehlhorn/LEDAbook.html
71. Mehta, A., Saberi, A., Vazirani, U.V., Vazirani, V.V.: AdWords and generalized online matching. J. ACM (2007). https://doi.org/10.1145/1284320.1284321
72. Michail, D., Kinable, J., Naveh, B., Sichi, J.V.: JGraphT – a Java library for graph data structures and algorithms. ACM Trans. Math. Softw. **46**(2), 1–29 (2020). https://doi.org/10.1145/3381449
73. Mohan, A., Leow, W.X., Hobor, A.: Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms. In: Comput. Aided Verification (2021). https://doi.org/10.1007/978-3-030-81688-9_37
74. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic (2002). https://doi.org/10.1007/3-540-45949-9
75. Noschinski, L.: A Graph Library for Isabelle. Math. Comput. Sci. (2015). https://doi.org/10.1007/S11786-014-0183-Z
76. Perron, L., Furnon, V.: OR-Tools, https://developers.google.com/optimization/, OR-Tools 9.15.6755. Last accessed on 13/10/2026
77. Schirmer, N.: A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In: Log. Program. Artif. Intell. Reason. (2005). https://doi.org/10.1007/978-3-540-32275-7_26
78. Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency (2003)
79. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002), https://www.boost.org/doc/libs/release/libs/graph/
80. Song, P., Yang, K., Anandkumar, A.: Lean Copilot: Large language models as copilots for theorem proving in Lean. In: 2nd Int. Conf. Neuro-Symb. Syst. NeuS. Proc. Mach. Learn. Res., vol. 288, pp. 144–169 (2025)
81. Stevens, L., Ghidini, R.: Simplified and verified: A second look at a proof-producing union-find algorithm. In: Automated Deduction – CADE 30: 30th International Conference on Automated Deduction, Stuttgart, Germany, July 28-31, 2025, Proceedings. p. 261–279. Springer-Verlag, Berlin, Heidelberg (2025). https://doi.org/10.1007/978-3-031-99984-0_15, https://doi.org/10.1007/978-3-031-99984-0_15
82. Stübinger, T., Hupel, L.: Extending Isabelle/HOL's Code Generator with Support for the Go Programming Language. In: Form. Methods (2025). https://doi.org/10.1007/978-3-031-71177-0_1
83. The NetworkX Developers: NetworkX reference, https://networkx.org/documentation/stable/reference/index.html, NetworkX 3.4.2. Last accessed on 10/10/2026
84. The SciPy Community: SciPy API reference, https://docs.scipy.org/doc/scipy/reference/, SciPy 1.15.3. Last accessed on 10/10/2026
85. Vazirani, V.V.: Approximation Algorithms
86. Wang, H., Xin, H., Liu, Z., Li, W., Huang, Y., Lu, J., Yang, Z., Tang, J., Yin, J., Li, Z., Liang, X.: Proving Theorems Recursively. Adv. Neural Inf. Process. Syst. (2024), https://papers.nips.cc/paper_files/paper/2024/hash/9de7a49945898da86e062e7029baa284-Abstract-Conference.html
87. Watt, C., Trela, M., Lammich, P., Märkl, F.: Wasmref-isabelle: A verified monadic interpreter and industrial fuzzing oracle for webassembly. Proc. ACM Program. Lang. **7**(PLDI), 100–123 (2023). https://doi.org/10.1145/3591224, https://doi.org/10.1145/3591224

88. Wirth, N.: Program Development by Stepwise Refinement. Commun. ACM (1971). https://doi.org/10.1145/362575.362577
89. Xin, H., Ren, Z.Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., Gao, W., Zhu, Q., Yang, D., Gou, Z., Wu, Z.F., Luo, F., Ruan, C.: DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search (2024). https://doi.org/10.48550/arXiv.2408.08152
90. Zhong, Z., Raghunathan, A., Carlini, N.: ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. In: 14th Int. Conf. Learn. Represent. ICLR (2026)
91. Zhou, J.P., Staats, C.E., Li, W., Szegedy, C., Weinberger, K.Q., Wu, Y.: Don't Trust: Verify – Grounding LLM Quantitative Reasoning with Autoformalization. In: The Twelfth International Conference on Learning Representations (2023), https://openreview.net/forum?id=V5tdi14ple

A Verified Algorithms and Graph Data Types

Table 3 gives the details of the verification of the algorithms of Table 1, and Table 4 describes the graph ADTs of Section 2.2.

Table 3. The verified algorithms. *Pat.*: loops are modelled as in Section 2.4 (✓), in part ((✓)), or not (−), or the algorithm is an instance of another one (inst.). *Inv.*: named invariants of the loops, without invariants of data structures; numbers in parentheses are inherited. *Proved*: termination (T), correctness (C), optimality (O), approximation or competitive ratio (A); running-time analyses are not included. *Lines*: theories specific to the algorithm, without running-time analyses, executable instances, refinements, as well as the abstract graph-theoretic reasoning. Daggers mark algorithms only in development branches. A reference such as L1 is to the main theorem in Listing 1 in Appendix D.

Algorithm	Pat.	Inv.	Proved	Main theorem	Lines
Graph search and shortest paths					
Depth-first search (L53)	✓	5	T, C	Finds a path from the source to the target iff one exists	877
Cycle detection (L54)	✓	16	T, C	Reports a cycle iff the graph has one	3,305
Directed cycle detection† (L55)	✓	5 (1)	T, C	Reports a cycle iff the searched part has one	697
Breadth-first search (L56)	✓	13	T, C	Visits exactly the reachable vertices, along shortest paths	1,805
BFS with distances† (L57)	✓	15	T, C	Also computes distances and shortest-path parents	2,901
Bellman–Ford (L58)	–	8	C, O	Without negative cycles, computes shortest paths	2,837
Dijkstra† (L59)	✓	14	T, C, O	Computes shortest paths to the settled vertices	2,865
Greedy algorithms and spanning trees					
Best-in-greedy (L60)	✓	4	T, C, A, O	Rank-quotient approximation; optimal for all weights iff matroid	5,750
Kruskal (L61)	inst.	(4)	T, C, O	Computes a maximum-weight spanning forest	1,559
Greedoid greedy (L62)	–	2	T, C, O	Optimal for all modular weights iff strong exchange	2,217
Jarník–Prim (L63)	inst.	(2)	T, C, O	Computes a maximum-weight spanning tree	1,277
Flows					
Successive shortest paths (L64)	✓	3	T, C, O	Computes a minimum-cost flow or reports none	1,158
Capacity scaling (L65)	✓	3	T, C, O	As for successive shortest paths	1,785
Orlin (L1)	✓	30	T, C, O	As for successive shortest paths	12,080
Dinic (L66)	✓	1	T, C, O	Computes a maximum flow	1,455
Blocking flow (L67)	(✓)	3	T, C	Computes a blocking flow	713
DFS with backtracking (L68)	✓	5	T, C	Removed edges lie on no source–target path	746
Push-relabel† (L71)	✓	7	T, C, O	Computes a maximum flow	4,897
Network simplex† (L69)	✓	7	T, C, O	Minimum-cost flow or infinite negative cycle	7,582
Acyclic flows† (L70)	✓	9	T, C	Makes a flow acyclic, not raising its cost	10,691
Matchings					
Edmonds' blossom (L2)	(✓)	15	T, C, O	Computes a maximum-cardinality matching	10,460
Weighted blossom† (L72)	(✓)	11	C, O	Min-weight perfect matching or reports none, if it terminates	5,801
Edmonds–Gallai decomposition† (L73)	–	0	T, C	Computes the decomposition from a maximum matching	5,007
Hopcroft–Karp† (L74)	✓	7	T, C, O	Computes a maximum-cardinality bipartite matching	2,381
Hungarian method (L75)	✓	8	T, C, O	Min-weight perfect matching or reports none	4,123
Primal-dual (Egerváry) (L76)	(✓)	0	T, C, O	Max-weight matching and an optimal dual	1,863
RANKING (L77)	–	0	A	Competitive ratio tending to $1 - 1/e$	6,973
RANKING, primal-dual (L78)	–	0	A	Ratio $1 - 1/e$, also vertex-weighted	4,861
AdWords (L79)	–	0	C, A	Ratio in terms of bids and budgets	1,347
Matroid intersection					
Matroid intersection (L80)	✓	1	T, C, O	Max-cardinality common independent set	2,646
Weighted matroid intersection† (L81)	✓	1	T, C, O	Max-weight common independent set	5,274
Matroid partitioning† (L82)	–	0	C, O	Optimal partition from a max-cardinality common independent set	1,215

Table 4. The graph ADTs of the library and their implementations. The costs are the standard bounds of the data structures, where n is the number of vertices, m the number of edges, and d the degree of the vertex concerned; the library does not verify them. Daggers mark ADTs that are only in development branches of the library. A reference such as L1 is to Listing 1 in Appendix D.

ADT	Graph; abstraction	Implementation	Costs	Reasons and users
`Pair_Graph_Specs` (L33)	Directed; vertex to set of out-neighbours; `digraph_abs`	Red-black trees (L36)	Looking up a neighbourhood, adding or deleting an edge: $O(\log n + \log d)$; choosing a neighbour: $O(1)$	Purely functional; vertices of any linearly ordered type; e.g. depth-first and breadth-first search, Dinic's algorithm, Prim's algorithm
		AVL trees (L37)	As for red-black trees	As for red-black trees; no algorithm in the main branch uses it
`Pair_Graph_U_Specs` (L34)	Undirected; both orientations of every edge; `ugraph_abs`	Red-black trees	As for `Pair_Graph_Specs`, with every edge added or deleted twice	Reuses the directed ADT and its lemmas; e.g. Kruskal's algorithm
`Adj_Map_Specs2` (L43)	Directed, over a map without deletion; `digraph_abs`	Red-black trees	Looking up a neighbourhood, adding an edge: $O(\log n + \log d)$	Requires no deletion from the map; the forest that Orlin's algorithm maintains
`Pair_Graph_Imperative` (L38)	Directed, on the heap; vertex to list of out-neighbours; `graph_assn`	Array maps and lists (L41)	Looking up a neighbourhood: $O(1)$; adding an edge: $O(d)$, since a neighbourhood is a list without duplicates	In-place updates; vertices must be numbers below a fixed bound; e.g. imperative depth-first search
		Hash maps and lists (L42)	Looking up a neighbourhood: expected $O(1)$; adding an edge: $O(d)$	In-place updates; vertices of any hashable type; e.g. imperative depth-first search
`Multi_Graph_Specs`† (L44)	Multigraph; vertex to set of its edges; `multigraph_abs`	Arrays (CSR)	Accessing the edges of a vertex: $O(1)$	Parallel edges; e.g. breadth-first search on multigraphs
`indexed_iterable_set`† (L45)	One set per index, e.g. per vertex, each with a cursor; `idx_abstract`	Arrays (CSR), `csr_graph` and, with weights, `weighted_csr` (L35, L46)	Element and weight at a cursor, advancing or resetting a cursor: $O(1)$; building the arrays: $O(n + m)$	A scan resumes where it stopped; the edges of a vertex are contiguous in memory, which is cache-efficient, but the graph cannot change; e.g. the Hungarian method
`outgoing_edge_iterator`, `ingoing_edge_iterator`† (L32, L47)	Multigraph whose out- or in-edges at every vertex are an `indexed_iterable_set`; `out_abstract`, `in_abstract`	Arrays (CSR)	As for `indexed_iterable_set`	As for `indexed_iterable_set`; e.g. Dijkstra's algorithm, the network simplex method

B Running Times

Tables 9 to 12 give the running times for breadth-first search, Tables 13 to 15 those for Dijkstra's algorithm, Tables 19 to 21 those for push-relabel, Tables 22 to 24 those for the Hungarian method and Tables 25 to 28 those for matroid intersection. Each row is an instance size n and each column a solver. An entry is the mean CPU time in seconds of the solver call over the instances of that size, where <0.001 is a mean below a millisecond. Next to it (below it in the BFS, Dijkstra, push-relabel and Hungarian tables), in parentheses, is that mean normalised by SML (by the imperative SML export in the BFS tables, by SML HL in the push-relabel tables), i.e. divided by the SML mean for the same size, so SML's own ratio is (1.00) and a value below 1 means the solver was faster than SML. A superscript k/m marks a mean over only the k of m instances that finished. TO means that no instance finished within 1200 s (larger sizes were then not run), OOM that the solver ran out of memory, OVF that the AFP export's integer index overflowed, n/a that the instances violate its precondition of no antiparallel arcs, and (–) that the SML mean is 0 or missing, so no ratio can be given.

Measurements in Java and Scala are done after a dummy call to the solver enforcing a JVM warm-up.

Table 5. Depth-first search, Square-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML CSR	SML lists	SML RBT	Scala CSR	Scala lists	Scala RBT	Go CSR	Go lists	Go RBT
1024	0.0001	0.0001	0.011	0.011	0.008	0.080	0.023	0.012	0.039
	(1.00)	(1.16)	(87.5)	(88.1)	(67.4)	(653)	(190)	(94.9)	(320)
2025	0.0001	0.0003	0.026	0.020	0.016	0.128	0.044	0.022	0.085
	(1.00)	(2.58)	(244)	(181)	(150)	(1187)	(408)	(200)	(791)
4096	0.0002	0.0006	0.059	0.040	0.035	0.201	0.107	0.047	0.202
	(1.00)	(2.59)	(273)	(186)	(160)	(925)	(494)	(218)	(929)
8190	0.0004	0.001	0.118	0.075	0.079	0.297	0.235	0.102	0.446
	(1.00)	(2.55)	(262)	(168)	(176)	(660)	(523)	(226)	(992)
16384	0.001	0.002	0.326	0.151	0.152	0.501	0.548	0.218	1.04
	(1.00)	(2.43)	(326)	(150)	(151)	(500)	(547)	(217)	(1037)
32761	0.003	0.005	0.721	0.294	0.179	0.912	1.23	0.466	2.45
	(1.00)	(2.02)	(286)	(117)	(71.0)	(362)	(487)	(185)	(973)
65536	0.006	0.013	1.75	0.458	0.281	1.95	2.63	1.01	5.58
	(1.00)	(2.16)	(301)	(78.8)	(48.4)	(336)	(453)	(174)	(961)
131044	0.013	0.054	3.70	0.758	0.536	4.00	5.49	2.15	12.4
	(1.00)	(4.12)	(284)	(58.2)	(41.2)	(307)	(421)	(165)	(951)
262144	0.028	0.075	8.11	1.32	0.900	9.03	10.9	4.52	27.7
	(1.00)	(2.70)	(293)	(47.9)	(32.6)	(327)	(396)	(163)	(1003)
524176	0.061	0.169	17.9	2.50	1.72	21.1	21.7	9.78	60.5
	(1.00)	(2.75)	(292)	(40.8)	(28.0)	(344)	(354)	(160)	(987)
1048576	0.184	1.42	43.8	4.98	3.56	49.1	OOM	20.4	OOM
	(1.00)	(7.72)	(238)	(27.0)	(19.3)	(266)		(110)	
2096704	0.729	1.11	101	10.9	7.27	111	OOM	OOM	OOM
	(1.00)	(1.53)	(138)	(15.0)	(9.98)	(152)			

n	LEMON CSR [code 38]	LEMON linked [code 38]	JGraphT CSR [code 29]	JGraphT hash [code 29]	SciPy CSR [code 55]	SciPy prebuilt [code 55]	NetworkX [code 44]
1024	0.0002	0.0000	0.0009	0.001	0.0004	0.0001	0.0006
	(1.27)	(0.29)	(7.72)	(10.1)	(3.52)	(1.05)	(4.85)
2025	0.0001	0.0001	0.002	0.003	0.0005	0.0002	0.001
	(1.27)	(0.60)	(16.3)	(27.3)	(4.81)	(1.42)	(11.3)
4096	0.0003	0.0001	0.004	0.008	0.0007	0.0002	0.003
	(1.30)	(0.62)	(17.1)	(34.6)	(3.22)	(1.00)	(11.6)
8190	0.0006	0.0003	0.008	0.021	0.001	0.0003	0.005
	(1.29)	(0.63)	(17.8)	(47.9)	(2.43)	(0.76)	(11.8)
16384	0.001	0.0007	0.019	0.054	0.002	0.0008	0.011
	(1.24)	(0.67)	(19.4)	(54.1)	(2.18)	(0.84)	(10.9)
32761	0.003	0.001	0.041	0.096	0.004	0.002	0.055
	(1.09)	(0.53)	(16.1)	(38.2)	(1.73)	(0.74)	(21.9)
65536	0.006	0.003	0.107	0.141	0.011	0.004	0.213
	(1.03)	(0.55)	(18.4)	(24.3)	(1.82)	(0.74)	(36.7)
131044	0.015	0.007	0.370	0.305	0.019	0.009	0.605
	(1.13)	(0.52)	(28.4)	(23.4)	(1.47)	(0.66)	(46.4)
262144	0.034	0.021	0.937	0.946	0.037	0.015	1.54
	(1.21)	(0.74)	(33.9)	(34.2)	(1.34)	(0.55)	(55.8)
524176	0.078	0.058	2.49	2.37	0.076	0.040	3.74
	(1.27)	(0.95)	(40.7)	(38.7)	(1.24)	(0.66)	(61.0)
1048576	0.203	0.152	5.71	6.40	0.156	0.076	7.78
	(1.10)	(0.83)	(30.9)	(34.7)	(0.84)	(0.41)	(42.2)
2096704	0.705	0.472	15.2	18.0	0.551	0.331	16.3
	(0.97)	(0.65)	(20.8)	(24.7)	(0.76)	(0.45)	(22.3)

Table 6. Depth-first search, Long-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML CSR	SML lists	SML RBT	Scala CSR	Scala lists	Scala RBT	Go CSR	Go lists	Go RBT
1024	0.0001	0.0001	0.023	0.010	0.007	0.077	0.022	0.012	0.039
	(1.00)	(2.49)	(397)	(182)	(126)	(1346)	(387)	(206)	(680)
2^{11}	0.0001	0.0003	0.025	0.020	0.016	0.130	0.044	0.022	0.087
	(1.00)	(2.54)	(232)	(180)	(148)	(1197)	(407)	(201)	(802)
2^{12}	0.0002	0.0005	0.056	0.038	0.035	0.190	0.102	0.047	0.196
	(1.00)	(2.52)	(257)	(176)	(160)	(877)	(472)	(217)	(905)
2^{13}	0.0004	0.001	0.111	0.070	0.076	0.293	0.236	0.101	0.439
	(1.00)	(2.57)	(251)	(158)	(172)	(663)	(535)	(227)	(992)
2^{14}	0.0010	0.002	0.312	0.150	0.146	0.493	0.543	0.216	1.01
	(1.00)	(2.37)	(313)	(151)	(147)	(494)	(544)	(216)	(1011)
2^{15}	0.002	0.005	0.684	0.296	0.179	0.888	1.21	0.458	2.31
	(1.00)	(2.11)	(284)	(123)	(74.2)	(369)	(501)	(190)	(959)
2^{16}	0.006	0.013	1.66	0.450	0.265	1.93	2.56	0.976	5.44
	(1.00)	(2.22)	(293)	(79.3)	(46.7)	(340)	(452)	(172)	(959)
2^{17}	0.013	0.051	3.64	0.741	0.513	3.87	5.34	2.10	12.2
	(1.00)	(4.05)	(288)	(58.7)	(40.6)	(307)	(423)	(167)	(964)
2^{18}	0.027	0.072	8.08	1.29	0.827	8.80	10.6	4.39	26.6
	(1.00)	(2.71)	(305)	(48.7)	(31.2)	(332)	(401)	(166)	(1005)
2^{19}	0.059	0.173	19.5	2.48	1.64	20.0	21.5	9.52	59.3
	(1.00)	(2.94)	(331)	(42.2)	(27.9)	(341)	(367)	(162)	(1010)
2^{20}	0.169	1.39	38.3	4.81	3.33	47.5	OOM	19.7	OOM
	(1.00)	(8.19)	(226)	(28.4)	(19.7)	(281)		(116)	
2^{21}	0.708	1.04	98.3	10.9	6.97	108	OOM	OOM	OOM
	(1.00)	(1.47)	(139)	(15.4)	(9.85)	(152)			

n	LEMON CSR [code 38]	LEMON linked [code 38]	JGraphT CSR [code 29]	JGraphT hash [code 29]	SciPy CSR [code 55]	SciPy prebuilt [code 55]	NetworkX [code 44]
1024	0.0001	0.0000	0.001	0.001	0.0005	0.0001	0.0006
	(1.25)	(0.82)	(17.9)	(26.0)	(9.04)	(2.40)	(10.5)
2^{11}	0.0001	0.0001	0.002	0.005	0.0006	0.0002	0.001
	(1.27)	(0.59)	(20.5)	(44.5)	(5.66)	(1.43)	(11.5)
2^{12}	0.0003	0.0001	0.006	0.014	0.0007	0.0002	0.003
	(1.29)	(0.59)	(26.2)	(63.8)	(3.25)	(0.98)	(11.6)
2^{13}	0.0006	0.0003	0.017	0.055	0.001	0.0003	0.005
	(1.29)	(0.67)	(38.0)	(123)	(2.54)	(0.78)	(11.9)
2^{14}	0.001	0.0006	0.058	0.100	0.002	0.0008	0.011
	(1.23)	(0.65)	(57.7)	(101)	(2.12)	(0.79)	(11.1)
2^{15}	0.003	0.001	0.199	0.311	0.005	0.002	0.049
	(1.12)	(0.59)	(82.7)	(129)	(1.88)	(0.75)	(20.4)
2^{16}	0.006	0.003	1.06	1.36	0.010	0.004	0.203
	(1.05)	(0.59)	(187)	(240)	(1.71)	(0.74)	(35.8)
2^{17}	0.014	0.007	4.56	5.88	0.019	0.009	0.488
	(1.12)	(0.52)	(362)	(466)	(1.53)	(0.69)	(38.7)
2^{18}	0.034	0.019	19.5	22.4	0.036	0.016	1.21
	(1.30)	(0.71)	(735)	(846)	(1.36)	(0.59)	(45.8)
2^{19}	0.101	0.064	74.5	83.7	0.074	0.037	2.91
	(1.72)	(1.09)	(1268)	(1424)	(1.26)	(0.62)	(49.5)
2^{20}	0.246	0.170	296	373	0.158	0.079	6.53
	(1.45)	(1.00)	(1746)	(2202)	(0.93)	(0.46)	(38.6)
2^{21}	0.616	0.513	TO	TO	0.550	0.356	15.0
	(0.87)	(0.72)			(0.78)	(0.50)	(21.2)

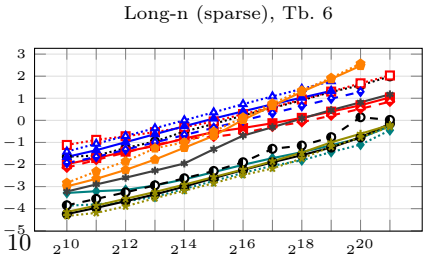


Fig. 4. DFS: filled marks are exports over CSR arrays, half-filled marks those over adjacency lists, hollow marks those over red-black trees. SML CSR, SML lists, SML RBT, Scala CSR, Scala lists, Scala RBT, Go CSR, Go lists, Go RBT, LEMON CSR, LEMON linked, JGraphT CSR, JGraphT hash, SciPy CSR, SciPy prebuilt, NetworkX.

Table 7. Depth-first search, dense connected rand. family [code 6] (10 instances per size, median of 3 repeats each). The instances are those of Table 11.

n	SML CSR	SML lists	SML RBT	Scala CSR	Scala lists	Scala RBT	Go CSR	Go lists	Go RBT
1024	0.0006	0.025	0.399	0.135	0.180	0.365	0.496	0.205	0.716
	(1.00)	(39.9)	(635)	(215)	(286)	(581)	(790)	(327)	(1140)
2^{11}	0.003	0.234	1.47	0.432	0.913	1.42	2.37	1.17	3.96
	(1.00)	(89.2)	(560)	(165)	(349)	(540)	(904)	(447)	(1512)
2^{12}	0.011	3.56	8.15	1.08	8.86	8.62	9.55	8.51	21.4
	(1.00)	(330)	(757)	(100)	(823)	(800)	(887)	(790)	(1983)
2^{13}	0.048	90.6	39.9	3.79	119	51.6	OOM	133	110
	(1.00)	(1876)	(826)	(78.5)	(2462)	(1068)		(2757)	(2282)
2^{14}	0.335	923	210	17.3	TO	281	OOM	OOM	OOM
	(1.00)	(2751)	(626)	(51.5)		(837)			
2^{15}	1.40	TO	$1134^{3/10}$	OOM	TO	TO	OOM	OOM	OOM
	(1.00)		(812)						

n	LEMON CSR [code 38]	LEMON linked [code 38]	JGraphT CSR [code 29]	JGraphT hash [code 29]	SciPy CSR [code 55]	SciPy prebuilt [code 55]	NetworkX [code 44]
1024	0.0008	0.0008	0.036	0.050	0.004	0.0003	0.003
	(1.33)	(1.29)	(56.9)	(80.0)	(6.88)	(0.45)	(4.12)
2^{11}	0.003	0.004	0.191	0.196	0.011	0.001	0.010
	(1.28)	(1.54)	(73.0)	(74.7)	(4.21)	(0.48)	(3.85)
2^{12}	0.019	0.024	1.57	1.90	0.040	0.003	0.269
	(1.79)	(2.21)	(146)	(177)	(3.70)	(0.26)	(24.9)
2^{13}	0.082	0.379	10.6	14.1	0.178	0.014	1.30
	(1.71)	(7.84)	(220)	(292)	(3.68)	(0.30)	(27.0)
2^{14}	0.375	2.69	77.7	155	0.749	0.040	3.54
	(1.12)	(8.03)	(232)	(464)	(2.23)	(0.12)	(10.6)
2^{15}	1.28	12.3	647	TO	3.21	0.159	26.3
	(0.92)	(8.79)	(464)		(2.30)	(0.11)	(18.8)

Table 8. Depth-first search, USA-road-t family [code 7] (one network per size, 10 sources each, median of 3 repeats each).

Network	n	SML CSR	SML lists	SML RBT	Scala CSR	Scala lists	Scala RBT	Go CSR	Go lists	Go RBT
NY	264346	0.025	0.063	5.27	0.887	0.586	4.32	7.66	3.65	19.7
		(1.00)	(2.48)	(208)	(35.0)	(23.1)	(171)	(302)	(144)	(780)
BAY	321270	0.023	0.072	6.07	0.967	0.657	5.09	8.44	4.42	23.0
		(1.00)	(3.08)	(260)	(41.4)	(28.2)	(218)	(362)	(189)	(985)
COL	435666	0.022	0.112	8.18	1.21	0.758	6.95	11.4	5.77	31.7
		(1.00)	(5.14)	(374)	(55.4)	(34.7)	(318)	(521)	(264)	(1449)
FLA	1070376	0.057	0.271	22.1	2.78	1.55	19.3	28.9	15.2	OOM
		(1.00)	(4.72)	(385)	(48.4)	(27.0)	(337)	(504)	(265)	
NW	1207945	0.064	0.254	24.8	3.00	1.52	22.2	30.5	14.8	OOM
		(1.00)	(3.95)	(386)	(46.7)	(23.7)	(345)	(474)	(230)	
NE	1524453	0.099	1.12	31.9	3.98	1.99	30.1	OOM	OOM	OOM
		(1.00)	(11.4)	(324)	(40.4)	(20.2)	(306)			
CAL	1890815	0.145	1.14	41.8	5.16	2.50	38.1	OOM	OOM	OOM
		(1.00)	(7.87)	(290)	(35.7)	(17.3)	(263)			
LKS	2758119	0.214	1.48	60.2	7.36	3.78	56.7	OOM	OOM	OOM
		(1.00)	(6.93)	(281)	(34.4)	(17.6)	(265)			
E	3598623	0.346	0.882	83.9	9.40	5.00	77.0	OOM	OOM	OOM
		(1.00)	(2.55)	(242)	(27.1)	(14.4)	(222)			
W	6262104	0.469	1.50	151	17.3	8.53	139	OOM	OOM	OOM
		(1.00)	(3.19)	(321)	(36.9)	(18.2)	(296)			
CTR	14081816	1.42	3.87	425	45.3	21.7	425	OOM	OOM	OOM
		(1.00)	(2.73)	(299)	(31.9)	(15.3)	(299)			
USA	23947347	2.24	7.39	$643^{3/10}$	78.8	33.2	605	OOM	OOM	OOM
		(1.00)	(3.30)	(287)	(35.2)	(14.8)	(270)			

Network	n	LEMON CSR [code 38]	LEMON linked [code 38]	JGraphT CSR [code 29]	JGraphT hash [code 29]	SciPy CSR [code 55]	SciPy prebuilt [code 55]	NetworkX [code 44]
NY	264346	0.020	0.013	0.392	0.455	0.024	0.010	0.440
		(0.79)	(0.52)	(15.5)	(18.0)	(0.94)	(0.38)	(17.4)
BAY	321270	0.026	0.018	0.353	0.437	0.026	0.011	0.526
		(1.10)	(0.77)	(15.1)	(18.7)	(1.13)	(0.47)	(22.6)
COL	435666	0.039	0.024	0.419	0.656	0.036	0.017	0.739
		(1.80)	(1.08)	(19.2)	(30.0)	(1.65)	(0.80)	(33.8)
FLA	1070376	0.089	0.099	1.40	1.54	0.093	0.045	1.81
		(1.54)	(1.73)	(24.4)	(26.8)	(1.61)	(0.78)	(31.6)
NW	1207945	0.110	0.130	1.70	1.85	0.103	0.051	1.62
		(1.71)	(2.01)	(26.4)	(28.7)	(1.60)	(0.79)	(25.2)
NE	1524453	0.145	0.158	4.34	4.74	0.136	0.064	2.76
		(1.47)	(1.60)	(44.1)	(48.1)	(1.38)	(0.65)	(28.1)
CAL	1890815	0.168	0.232	4.45	4.58	0.188	0.092	3.43
		(1.16)	(1.60)	(30.8)	(31.7)	(1.30)	(0.64)	(23.7)
LKS	2758119	0.257	0.266	8.17	7.76	0.276	0.140	5.32
		(1.20)	(1.24)	(38.1)	(36.2)	(1.29)	(0.65)	(24.8)
E	3598623	0.377	0.382	11.1	15.5	0.369	0.198	7.01
		(1.09)	(1.10)	(31.9)	(44.9)	(1.06)	(0.57)	(20.2)
W	6262104	0.652	0.601	20.3	20.4	0.627	0.341	11.9
		(1.39)	(1.28)	(43.3)	(43.4)	(1.34)	(0.73)	(25.4)
CTR	14081816	1.92	1.55	95.6	185	1.86	1.09	33.1
		(1.35)	(1.09)	(67.3)	(131)	(1.31)	(0.77)	(23.3)
USA	23947347	2.44	2.72	239	331	2.51	1.49	48.2
		(1.09)	(1.21)	(107)	(148)	(1.12)	(0.66)	(21.5)

Table 9. Breadth-first search, Square-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML imp.	SML func.	Scala imp.	Scala func.	Go imp.	Go func.
1024	0.0001	0.019	0.010	0.234	0.027	0.145
	(1.00)	(174)	(90.8)	(2085)	(241)	(1293)
2025	0.0001	0.064	0.019	0.473	0.033	0.376
	(1.00)	(605)	(178)	(4462)	(310)	(3547)
4096	0.0002	0.230	0.037	1.12	0.076	1.47
	(1.00)	(1105)	(178)	(5384)	(366)	(7045)
8190	0.0004	0.808	0.067	3.68	0.174	5.76
	(1.00)	(1980)	(164)	(9023)	(426)	(14126)
16384	0.0009	3.15	0.134	16.5	0.375	22.7
	(1.00)	(3592)	(153)	(18794)	(428)	(25969)
32761	0.002	12.3	0.264	78.3	0.836	92.5
	(1.00)	(6453)	(138)	(41005)	(438)	(48419)
65536	0.004	53.2	0.421	501	1.78	577
	(1.00)	(12532)	(99.1)	(118038)	(420)	(135887)
131044	0.010	200	0.639	TO	3.71	TO
	(1.00)	(20687)	(66.3)		(384)	
262144	0.020	TO	1.06	TO	7.58	TO
	(1.00)		(52.6)		(377)	
524176	0.049	TO	2.03	TO	15.1	TO
	(1.00)		(41.8)		(310)	
1048576	0.125	TO	3.91	TO	OOM	TO
	(1.00)		(31.3)			
2096704	0.452	TO	8.46	TO	OOM	TO
	(1.00)		(18.7)			

n	LEMON CSR [code 35]	LEMON linked [code 35]	JGraphT CSR [code 28]	JGraphT hash [code 28]	SciPy CSR [code 54]	SciPy prebuilt [code 54]	NetworkX [code 43]
1024	0.0001	0.0000	0.0008	0.0007	0.0005	0.0002	0.001
	(1.24)	(0.23)	(6.82)	(6.01)	(4.50)	(1.45)	(9.05)
2025	0.0001	0.0000	0.001	0.001	0.0005	0.0001	0.001
	(1.16)	(0.35)	(12.5)	(12.9)	(4.66)	(1.26)	(10.3)
4096	0.0003	0.0001	0.003	0.003	0.0008	0.0002	0.002
	(1.21)	(0.40)	(13.6)	(16.1)	(3.62)	(0.88)	(10.6)
8190	0.0005	0.0002	0.006	0.008	0.0010	0.0003	0.005
	(1.28)	(0.46)	(14.4)	(18.6)	(2.35)	(0.61)	(11.1)
16384	0.001	0.0004	0.012	0.021	0.002	0.0006	0.009
	(1.21)	(0.45)	(14.2)	(24.0)	(2.21)	(0.69)	(10.5)
32761	0.002	0.0010	0.023	0.046	0.004	0.001	0.023
	(1.20)	(0.50)	(11.9)	(24.0)	(2.05)	(0.61)	(11.9)
65536	0.005	0.002	0.054	0.071	0.009	0.003	0.056
	(1.15)	(0.43)	(12.6)	(16.6)	(2.17)	(0.62)	(13.1)
131044	0.013	0.004	0.226	0.127	0.017	0.005	0.144
	(1.30)	(0.42)	(23.4)	(13.1)	(1.74)	(0.56)	(15.0)
262144	0.029	0.018	0.511	0.460	0.031	0.009	0.354
	(1.47)	(0.92)	(25.5)	(22.9)	(1.55)	(0.42)	(17.6)
524176	0.064	0.040	1.32	0.854	0.058	0.021	1.55
	(1.32)	(0.82)	(27.3)	(17.6)	(1.20)	(0.43)	(32.0)
1048576	0.152	0.095	3.11	1.74	0.118	0.034	2.49
	(1.22)	(0.76)	(24.8)	(13.9)	(0.94)	(0.28)	(19.9)
2096704	0.449	0.331	6.62	3.64	0.346	0.134	7.28
	(0.99)	(0.73)	(14.6)	(8.06)	(0.77)	(0.30)	(16.1)

Table 10. Breadth-first search, Long-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML imp.	SML func.	Scala imp.	Scala func.	Go imp.	Go func.
1024	0.0001	0.019	0.010	0.239	0.039	0.104
	(1.00)	(371)	(195)	(4588)	(743)	(2000)
2^{11}	0.0001	0.063	0.020	0.438	0.033	0.382
	(1.00)	(599)	(185)	(4136)	(316)	(3603)
2^{12}	0.0002	0.226	0.037	1.14	0.076	1.44
	(1.00)	(1126)	(186)	(5691)	(380)	(7140)
2^{13}	0.0004	0.793	0.072	3.47	0.169	5.66
	(1.00)	(1992)	(181)	(8726)	(426)	(14217)
2^{14}	0.0009	3.07	0.138	19.8	0.375	22.1
	(1.00)	(3564)	(161)	(22951)	(435)	(25582)
2^{15}	0.002	11.9	0.264	66.5	0.817	91.1
	(1.00)	(6241)	(139)	(34831)	(428)	(47750)
2^{16}	0.004	51.4	0.437	561	1.75	549
	(1.00)	(12221)	(104)	(133494)	(416)	(130735)
2^{17}	0.010	191	0.648	TO	3.66	TO
	(1.00)	(19408)	(65.7)		(371)	
2^{18}	0.020	TO	1.11	TO	7.42	TO
	(1.00)		(55.6)		(371)	
2^{19}	0.045	TO	2.05	TO	14.9	TO
	(1.00)		(45.9)		(334)	
2^{20}	0.123	TO	3.93	TO	OOM	TO
	(1.00)		(31.9)			
2^{21}	0.459	TO	8.45	TO	OOM	TO
	(1.00)		(18.4)			

n	LEMON CSR [code 35]	LEMON linked [code 35]	JGraphT CSR [code 28]	JGraphT hash [code 28]	SciPy CSR [code 54]	SciPy prebuilt [code 54]	NetworkX [code 43]
1024	0.0001	0.0000	0.0008	0.0007	0.0004	0.0001	0.0006
	(2.65)	(0.40)	(14.5)	(13.6)	(7.96)	(2.23)	(10.6)
2^{11}	0.0001	0.0000	0.001	0.002	0.0005	0.0001	0.001
	(1.15)	(0.38)	(13.2)	(14.2)	(4.68)	(1.29)	(10.4)
2^{12}	0.0002	0.0001	0.003	0.003	0.0007	0.0002	0.002
	(1.23)	(0.42)	(13.0)	(14.1)	(3.33)	(0.88)	(11.0)
2^{13}	0.0005	0.0002	0.006	0.007	0.0010	0.0003	0.004
	(1.26)	(0.43)	(14.0)	(18.2)	(2.44)	(0.72)	(11.2)
2^{14}	0.001	0.0004	0.012	0.019	0.002	0.0006	0.009
	(1.21)	(0.43)	(13.3)	(21.7)	(2.04)	(0.65)	(10.6)
2^{15}	0.002	0.0009	0.023	0.043	0.004	0.001	0.022
	(1.17)	(0.45)	(12.0)	(22.4)	(2.03)	(0.63)	(11.3)
2^{16}	0.005	0.002	0.056	0.065	0.008	0.003	0.056
	(1.14)	(0.41)	(13.2)	(15.5)	(1.99)	(0.61)	(13.4)
2^{17}	0.012	0.004	0.222	0.124	0.016	0.006	0.139
	(1.24)	(0.42)	(22.5)	(12.5)	(1.65)	(0.56)	(14.1)
2^{18}	0.029	0.019	0.562	0.416	0.031	0.009	0.270
	(1.43)	(0.93)	(28.1)	(20.8)	(1.53)	(0.46)	(13.5)
2^{19}	0.087	0.042	1.37	0.836	0.058	0.021	0.623
	(1.95)	(0.95)	(30.8)	(18.7)	(1.31)	(0.46)	(13.9)
2^{20}	0.168	0.081	3.15	1.67	0.114	0.036	1.39
	(1.36)	(0.66)	(25.5)	(13.5)	(0.93)	(0.29)	(11.3)
2^{21}	0.393	0.282	6.90	3.39	0.358	0.129	3.34
	(0.85)	(0.61)	(15.0)	(7.39)	(0.78)	(0.28)	(7.28)

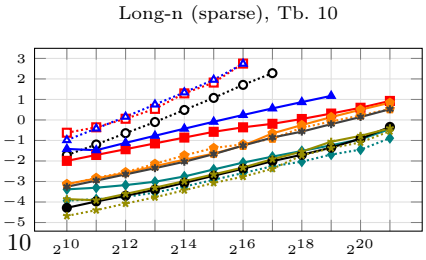


Fig. 5. BFS: filled marks are exports over CSR arrays, hollow marks those over red-black trees. SML imp., SML func., Scala imp., Scala func., Go imp., Go func., LEMON CSR, LEMON linked, JGraphT CSR, JGraphT hash, SciPy CSR, SciPy prebuilt, NetworkX.

Table 11. Breadth-first search, dense connected rand. family [code 6] (10 instances per size, median of 3 repeats each). Random graphs from our own generator: connect all vertices in the order of a random permutation to ensure strong connectivity. Then, for $m - n$ times, we randomly draw an edge, retry if a self-loop is drawn, and insert the resulting random edge to the graph. For a number of vertices n, we set $m = n^2/16$ arcs, so the average degree grows from 64 at $n = 1024$ to 2048 at $n = 32768$.

n	SML imp.	SML func.	Scala imp.	Scala func.	Go imp.	Go func.
1024	0.0006	0.315	0.109	1.20	0.366	1.81
	(1.00)	(573)	(198)	(2179)	(666)	(3297)
2^{11}	0.002	2.27	0.386	11.3	1.68	19.1
	(1.00)	(1040)	(177)	(5182)	(769)	(8760)
2^{12}	0.009	37.9	0.884	115	6.85	248
	(1.00)	(4232)	(98.7)	(12876)	(765)	(27722)
2^{13}	0.043	180	3.04	772	OOM	TO
	(1.00)	(4205)	(70.9)	(18027)		
2^{14}	0.310	TO	13.7	TO	OOM	TO
	(1.00)		(44.3)			
2^{15}	1.38	TO	OOM	TO	OOM	TO
	(1.00)					

n	LEMON CSR [code 35]	LEMON linked [code 35]	JGraphT CSR [code 28]	JGraphT hash [code 28]	SciPy CSR [code 54]	SciPy prebuilt [code 54]	NetworkX [code 43]
1024	0.002	0.002	0.016	0.011	0.002	0.0003	0.0007
	(3.10)	(2.80)	(29.7)	(19.9)	(4.12)	(0.54)	(1.26)
2^{11}	0.003	0.004	0.080	0.042	0.010	0.001	0.002
	(1.43)	(1.79)	(36.6)	(19.2)	(4.76)	(0.62)	(0.74)
2^{12}	0.016	0.026	0.587	0.169	0.041	0.003	0.134
	(1.80)	(2.93)	(65.6)	(18.8)	(4.52)	(0.33)	(14.9)
2^{13}	0.075	0.359	3.09	0.577	0.180	0.016	0.522
	(1.75)	(8.37)	(72.2)	(13.5)	(4.21)	(0.37)	(12.2)
2^{14}	0.359	2.06	17.4	2.57	0.760	0.046	0.047
	(1.16)	(6.63)	(56.0)	(8.28)	(2.45)	(0.15)	(0.15)
2^{15}	1.26	9.22	85.2	12.3	3.34	0.228	10.8
	(0.91)	(6.68)	(61.8)	(8.90)	(2.42)	(0.17)	(7.86)

Table 12. Breadth-first search, USA-road-t family [code 7] (one network per size, 10 source-target pairs each, median of 3 repeats each).

Network	n	SML imp.	SML func.	Scala imp.	Scala func.	Go imp.	Go func.
NY	264346	0.039	854	0.761	TO	4.95	TO
		(1.00)	(21644)	(19.3)		(125)	
BAY	321270	0.024	1037	0.803	TO	5.33	TO
		(1.00)	(42977)	(33.3)		(221)	
COL	435666	0.022	TO	1.02	TO	7.68	TO
		(1.00)		(45.8)		(346)	
FLA	1070376	0.067	TO	2.23	TO	19.1	TO
		(1.00)		(33.3)		(285)	
NW	1207945	0.075	TO	2.39	TO	19.9	TO
		(1.00)		(31.9)		(266)	
NE	1524453	0.102	TO	3.18	TO	OOM	TO
		(1.00)		(31.3)			
CAL	1890815	0.138	TO	3.85	TO	OOM	TO
		(1.00)		(27.9)			
LKS	2758119	0.210	TO	5.79	TO	OOM	TO
		(1.00)		(27.5)			
E	3598623	0.358	TO	7.51	TO	OOM	TO
		(1.00)		(21.0)			
W	6262104	0.528	TO	13.7	TO	OOM	TO
		(1.00)		(25.9)			
CTR	14081816	1.36	TO	34.2	TO	OOM	TO
		(1.00)		(25.1)			
USA	23947347	1.85	TO	56.8	TO	OOM	TO
		(1.00)		(30.7)			

Network	n	LEMON CSR [code 35]	LEMON linked [code 35]	JGraphT CSR [code 28]	JGraphT hash [code 28]	SciPy CSR [code 54]	SciPy prebuilt [code 54]	NetworkX [code 43]
NY	264346	0.039	0.011	0.147	0.188	0.023	0.008	0.308
		(0.99)	(0.28)	(3.73)	(4.77)	(0.60)	(0.20)	(7.80)
BAY	321270	0.023	0.015	0.181	0.261	0.024	0.009	0.353
		(0.97)	(0.64)	(7.51)	(10.8)	(1.01)	(0.37)	(14.6)
COL	435666	0.030	0.022	0.308	0.507	0.033	0.013	0.514
		(1.37)	(0.97)	(13.9)	(22.9)	(1.49)	(0.60)	(23.2)
FLA	1070376	0.086	0.086	0.940	1.06	0.081	0.034	1.71
		(1.29)	(1.28)	(14.0)	(15.8)	(1.21)	(0.50)	(25.5)
NW	1207945	0.096	0.075	1.14	1.20	0.092	0.042	1.99
		(1.28)	(1.00)	(15.2)	(16.0)	(1.23)	(0.56)	(26.6)
NE	1524453	0.139	0.095	2.04	1.50	0.120	0.050	2.36
		(1.37)	(0.93)	(20.0)	(14.7)	(1.19)	(0.50)	(23.2)
CAL	1890815	0.164	0.181	2.18	1.69	0.173	0.081	3.44
		(1.19)	(1.31)	(15.8)	(12.2)	(1.26)	(0.59)	(24.9)
LKS	2758119	0.288	0.226	4.43	2.69	0.247	0.113	4.97
		(1.37)	(1.07)	(21.0)	(12.8)	(1.18)	(0.54)	(23.6)
E	3598623	0.376	0.261	6.22	3.93	0.307	0.144	7.41
		(1.05)	(0.73)	(17.4)	(11.0)	(0.86)	(0.40)	(20.7)
W	6262104	0.540	0.389	10.3	6.41	0.575	0.284	13.9
		(1.02)	(0.74)	(19.6)	(12.1)	(1.09)	(0.54)	(26.3)
CTR	14081816	1.43	1.24	32.3	18.0	1.40	0.621	57.4
		(1.05)	(0.91)	(23.7)	(13.2)	(1.03)	(0.46)	(42.1)
USA	23947347	2.26	1.95	45.8	26.8	2.22	1.02	67.1
		(1.22)	(1.05)	(24.8)	(14.5)	(1.20)	(0.55)	(36.3)

Table 13. Dijkstra's algorithm, Square-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	LEMON CSR [code 39]	LEMON linked [code 39]
1024	0.0004	0.016	0.036	0.0002	0.0001
	(1.00)	(38.1)	(87.5)	(0.54)	(0.13)
2025	0.0007	0.023	0.057	0.0002	0.0001
	(1.00)	(31.3)	(78.1)	(0.23)	(0.07)
4096	0.002	0.060	0.178	0.0004	0.0002
	(1.00)	(34.9)	(104)	(0.24)	(0.10)
8190	0.004	0.132	0.459	0.0009	0.0004
	(1.00)	(33.8)	(118)	(0.23)	(0.11)
16384	0.007	0.220	0.953	0.002	0.0008
	(1.00)	(30.6)	(133)	(0.24)	(0.10)
32761	0.008	0.370	1.97	0.003	0.002
	(1.00)	(47.6)	(253)	(0.43)	(0.22)
65536	0.052	0.530	3.85	0.007	0.003
	(1.00)	(10.2)	(73.8)	(0.13)	(0.06)
131044	0.063	0.978	11.0	0.022	0.013
	(1.00)	(15.4)	(174)	(0.34)	(0.20)
262144	0.100	1.58	18.8	0.052	0.026
	(1.00)	(15.9)	(188)	(0.52)	(0.26)
524176	0.216	2.80	32.2	0.118	0.048
	(1.00)	(12.9)	(149)	(0.55)	(0.22)
1048576	0.707	5.02	OOM	0.252	0.101
	(1.00)	(7.10)		(0.36)	(0.14)
2096704	1.70	13.6	OOM	0.691	0.425
	(1.00)	(8.00)		(0.41)	(0.25)

n	JGraphT CSR [code 30]	JGraphT hash [code 30]	SciPy CSR [code 56]	SciPy prebuilt [code 56]	NetworkX [code 47]
1024	0.0008	0.0007	0.0006	0.0003	0.001
	(2.01)	(1.79)	(1.56)	(0.71)	(2.75)
2025	0.001	0.0009	0.0009	0.0005	0.003
	(1.81)	(1.27)	(1.19)	(0.68)	(3.66)
4096	0.003	0.003	0.001	0.0009	0.005
	(1.69)	(1.75)	(0.83)	(0.54)	(3.03)
8190	0.006	0.008	0.003	0.002	0.014
	(1.49)	(2.01)	(0.67)	(0.50)	(3.52)
16384	0.012	0.013	0.006	0.004	0.032
	(1.61)	(1.76)	(0.78)	(0.59)	(4.50)
32761	0.024	0.024	0.012	0.009	0.068
	(3.02)	(3.03)	(1.49)	(1.18)	(8.79)
65536	0.059	0.038	0.025	0.020	0.150
	(1.14)	(0.73)	(0.48)	(0.38)	(2.88)
131044	0.249	0.145	0.056	0.046	1.06
	(3.94)	(2.29)	(0.88)	(0.72)	(16.7)
262144	0.505	0.290	0.127	0.109	1.89
	(5.07)	(2.91)	(1.27)	(1.09)	(18.9)
524176	1.18	0.468	0.343	0.304	3.71
	(5.44)	(2.17)	(1.59)	(1.40)	(17.2)
1048576	2.45	0.753	0.778	0.736	6.08
	(3.47)	(1.06)	(1.10)	(1.04)	(8.60)
2096704	6.65	2.28	1.85	1.70	34.1
	(3.92)	(1.35)	(1.09)	(1.00)	(20.1)

Table 14. Dijkstra's algorithm, Long-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	LEMON CSR [code 39]	LEMON linked [code 39]
1024	0.0004	0.015	0.056	0.0001	0.0000
	(1.00)	(36.6)	(141)	(0.25)	(0.09)
2^{11}	0.0008	0.030	0.071	0.0002	0.0001
	(1.00)	(36.2)	(86.6)	(0.24)	(0.10)
2^{12}	0.002	0.060	0.170	0.0004	0.0002
	(1.00)	(35.8)	(102)	(0.23)	(0.09)
2^{13}	0.003	0.112	0.391	0.0008	0.0003
	(1.00)	(32.2)	(113)	(0.23)	(0.09)
2^{14}	0.008	0.246	1.04	0.002	0.0008
	(1.00)	(31.5)	(134)	(0.23)	(0.10)
2^{15}	0.014	0.368	2.08	0.003	0.002
	(1.00)	(26.4)	(149)	(0.25)	(0.12)
2^{16}	0.051	0.565	4.31	0.007	0.003
	(1.00)	(11.2)	(85.2)	(0.14)	(0.07)
2^{17}	0.050	0.870	9.01	0.018	0.008
	(1.00)	(17.3)	(179)	(0.36)	(0.15)
2^{18}	0.070	1.28	14.1	0.039	0.016
	(1.00)	(18.2)	(200)	(0.56)	(0.22)
2^{19}	0.202	2.64	31.3	0.123	0.042
	(1.00)	(13.1)	(155)	(0.61)	(0.21)
2^{20}	0.750	5.58	OOM	0.264	0.121
	(1.00)	(7.45)		(0.35)	(0.16)
2^{21}	1.51	11.9	OOM	0.548	0.344
	(1.00)	(7.92)		(0.36)	(0.23)

n	JGraphT CSR [code 30]	JGraphT hash [code 30]	SciPy CSR [code 56]	SciPy prebuilt [code 56]	NetworkX [code 47]
1024	0.0008	0.0005	0.0007	0.0003	0.001
	(2.02)	(1.29)	(1.68)	(0.72)	(2.69)
2^{11}	0.001	0.001	0.0009	0.0004	0.005
	(1.65)	(1.49)	(1.12)	(0.55)	(6.05)
2^{12}	0.003	0.002	0.001	0.0008	0.005
	(1.66)	(1.45)	(0.77)	(0.46)	(3.23)
2^{13}	0.006	0.006	0.002	0.001	0.015
	(1.65)	(1.80)	(0.62)	(0.42)	(4.22)
2^{14}	0.011	0.017	0.004	0.003	0.064
	(1.44)	(2.22)	(0.58)	(0.41)	(8.16)
2^{15}	0.025	0.029	0.009	0.006	0.182
	(1.77)	(2.07)	(0.62)	(0.45)	(13.0)
2^{16}	0.059	0.045	0.018	0.014	0.727
	(1.17)	(0.88)	(0.37)	(0.27)	(14.4)
2^{17}	0.251	0.106	0.038	0.029	3.81
	(4.98)	(2.10)	(0.75)	(0.57)	(75.8)
2^{18}	0.507	0.211	0.088	0.071	5.36
	(7.21)	(3.00)	(1.25)	(1.01)	(76.2)
2^{19}	1.17	0.488	0.256	0.192	$31.2^{9/10}$
	(5.78)	(2.42)	(1.27)	(0.95)	(155)
2^{20}	2.72	1.05	0.594	0.449	$19.9^{4/10}$
	(3.63)	(1.40)	(0.79)	(0.60)	(26.5)
2^{21}	6.84	2.07	1.31	1.17	OOM
	(4.53)	(1.37)	(0.87)	(0.77)	

Table 15. Dijkstra's algorithm, USA-road-t family [code 7] (one network per size, 10 source-target pairs each, median of 3 repeats each).

Network	n	SML	Scala	Go	LEMON CSR [code 39]	LEMON linked [code 39]
NY	264346	0.131	1.15	14.1	0.061	0.020
		(1.00)	(8.76)	(107)	(0.46)	(0.15)
BAY	321270	0.114	1.45	19.2	0.039	0.029
		(1.00)	(12.7)	(168)	(0.34)	(0.25)
COL	435666	0.115	1.69	21.8	0.048	0.032
		(1.00)	(14.7)	(190)	(0.42)	(0.28)
FLA	1070376	0.300	3.93	51.1	0.134	0.094
		(1.00)	(13.1)	(170)	(0.45)	(0.31)
NW	1207945	0.363	4.93	61.4	0.168	0.123
		(1.00)	(13.6)	(169)	(0.46)	(0.34)
NE	1524453	0.724	6.63	80.1	0.233	0.172
		(1.00)	(9.16)	(111)	(0.32)	(0.24)
CAL	1890815	0.788	7.71	OOM	0.289	0.208
		(1.00)	(9.78)		(0.37)	(0.26)
LKS	2758119	1.14	11.2	OOM	0.415	0.300
		(1.00)	(9.83)		(0.36)	(0.26)
E	3598623	1.54	15.2	OOM	0.528	0.394
		(1.00)	(9.91)		(0.34)	(0.26)
W	6262104	2.35	25.4	OOM	0.870	0.598
		(1.00)	(10.8)		(0.37)	(0.25)
CTR	14081816	7.13	71.1	OOM	2.67	2.02
		(1.00)	(9.97)		(0.37)	(0.28)
USA	23947347	12.5	$110^{6/10}$	OOM	4.13	3.90
		(1.00)	(8.83)		(0.33)	(0.31)

Network	n	JGraphT CSR [code 30]	JGraphT hash [code 30]	SciPy CSR [code 56]	SciPy prebuilt [code 56]	NetworkX [code 47]
NY	264346	0.175	0.180	0.082	0.080	1.08
		(1.33)	(1.38)	(0.62)	(0.61)	(8.24)
BAY	321270	0.219	0.263	0.094	0.088	2.66
		(1.92)	(2.30)	(0.82)	(0.77)	(23.3)
COL	435666	0.267	0.324	0.140	0.128	2.96
		(2.33)	(2.83)	(1.22)	(1.11)	(25.8)
FLA	1070376	0.995	0.695	0.384	0.347	8.14
		(3.31)	(2.31)	(1.28)	(1.16)	(27.1)
NW	1207945	1.22	1.01	0.452	0.440	16.2
		(3.36)	(2.78)	(1.25)	(1.21)	(44.6)
NE	1524453	1.89	1.22	0.586	0.573	19.3
		(2.61)	(1.68)	(0.81)	(0.79)	(26.6)
CAL	1890815	2.28	1.28	0.819	0.784	29.3
		(2.90)	(1.63)	(1.04)	(0.99)	(37.2)
LKS	2758119	4.22	2.05	1.21	1.14	42.9
		(3.70)	(1.80)	(1.06)	(1.00)	(37.7)
E	3598623	5.70	3.02	1.65	1.60	$47.5^{9/10}$
		(3.71)	(1.96)	(1.07)	(1.04)	(30.9)
W	6262104	10.4	4.15	2.97	2.92	OOM
		(4.43)	(1.77)	(1.26)	(1.24)	
CTR	14081816	29.0	13.2	8.30	8.17	OOM
		(4.07)	(1.85)	(1.16)	(1.15)	
USA	23947347	45.6	24.4	13.2	12.9	OOM
		(3.66)	(1.95)	(1.06)	(1.03)	

Table 16. Kruskal's algorithm, Square-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	LEMON [code 40]	JGraphT [code 31]	SciPy [code 58]	NetworkX [code 45]
1024	0.0004	0.014	0.025	0.0001	0.0008	0.0003	0.005
	(1.00)	(39.0)	(70.3)	(0.31)	(2.13)	(0.91)	(13.8)
2025	0.0007	0.048	0.060	0.0002	0.002	0.0005	0.009
	(1.00)	(68.3)	(85.6)	(0.33)	(2.50)	(0.70)	(12.6)
4096	0.001	0.249	0.158	0.0005	0.004	0.0009	0.018
	(1.00)	(170)	(108)	(0.33)	(2.98)	(0.61)	(12.3)
8190	0.003	0.746	0.464	0.001	0.009	0.002	0.036
	(1.00)	(254)	(158)	(0.38)	(2.90)	(0.59)	(12.3)
16384	0.006	3.86	1.57	0.002	0.015	0.004	0.095
	(1.00)	(615)	(250)	(0.39)	(2.31)	(0.57)	(15.1)
32761	0.014	16.1	5.97	0.005	0.044	0.007	0.204
	(1.00)	(1125)	(418)	(0.36)	(3.08)	(0.52)	(14.2)
65536	0.037	72.0	22.5	0.011	0.109	0.017	0.562
	(1.00)	(1935)	(605)	(0.30)	(2.94)	(0.45)	(15.1)
131044	0.064	303	85.1	0.024	0.285	0.035	1.40
	(1.00)	(4708)	(1321)	(0.37)	(4.42)	(0.55)	(21.7)
262144	0.146	TO	$728^{8/10}$	0.056	0.743	0.078	3.26
	(1.00)		(4981)	(0.38)	(5.08)	(0.54)	(22.3)
524176	0.354	TO	TO	0.129	1.76	0.160	7.56
	(1.00)			(0.37)	(4.98)	(0.45)	(21.4)
1048576	0.765	TO	TO	0.301	4.06	0.365	15.9
	(1.00)			(0.39)	(5.30)	(0.48)	(20.8)
2096704	1.67	TO	TO	0.669	9.80	0.882	34.2
	(1.00)			(0.40)	(5.87)	(0.53)	(20.5)

Table 17. Kruskal's algorithm, Long-n family [code 6] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	LEMON [code 40]	JGraphT [code 31]	SciPy [code 58]	NetworkX [code 45]
1024	0.0004	0.014	0.025	0.0001	0.0008	0.0003	0.005
	(1.00)	(39.8)	(70.1)	(0.32)	(2.23)	(0.89)	(13.7)
2^{11}	0.0007	0.063	0.060	0.0002	0.002	0.0005	0.009
	(1.00)	(90.6)	(85.3)	(0.33)	(2.63)	(0.71)	(12.7)
2^{12}	0.001	0.162	0.154	0.0005	0.004	0.0009	0.018
	(1.00)	(115)	(110)	(0.35)	(3.12)	(0.63)	(12.7)
2^{13}	0.003	0.919	0.457	0.001	0.008	0.002	0.036
	(1.00)	(312)	(155)	(0.38)	(2.67)	(0.58)	(12.1)
2^{14}	0.006	4.15	1.51	0.002	0.015	0.004	0.096
	(1.00)	(674)	(246)	(0.39)	(2.51)	(0.57)	(15.5)
2^{15}	0.015	16.6	5.36	0.005	0.041	0.007	0.216
	(1.00)	(1121)	(362)	(0.34)	(2.78)	(0.50)	(14.6)
2^{16}	0.037	66.7	21.2	0.011	0.105	0.016	0.529
	(1.00)	(1810)	(576)	(0.30)	(2.86)	(0.43)	(14.4)
2^{17}	0.064	282	81.4	0.023	0.291	0.035	1.40
	(1.00)	(4373)	(1264)	(0.36)	(4.52)	(0.54)	(21.7)
2^{18}	0.136	$1147^{3/10}$	672	0.055	0.703	0.073	3.29
	(1.00)	(8427)	(4935)	(0.40)	(5.17)	(0.54)	(24.2)
2^{19}	0.368	TO	TO	0.126	1.73	0.158	6.79
	(1.00)			(0.34)	(4.69)	(0.43)	(18.4)
2^{20}	0.739	TO	TO	0.292	3.99	0.348	15.5
	(1.00)			(0.39)	(5.40)	(0.47)	(21.0)
2^{21}	1.67	TO	TO	0.681	9.60	0.852	34.2
	(1.00)			(0.41)	(5.73)	(0.51)	(20.4)

Table 18. Kruskal's algorithm, dense connected rand. family [code 6] (10 instances per size, median of 3 repeats each). The instances are those of Table 11.

n	SML	Scala	Go	LEMON [code 40]	JGraphT [code 31]	SciPy [code 58]	NetworkX [code 45]
1024	0.012	13.3	5.08	0.003	0.017	0.005	0.108
	(1.00)	(1127)	(429)	(0.28)	(1.48)	(0.39)	(9.09)
2^{11}	0.061	249	81.1	0.015	0.099	0.021	0.589
	(1.00)	(4106)	(1339)	(0.25)	(1.63)	(0.34)	(9.73)
2^{12}	0.291	TO	TO	0.087	0.698	0.091	2.92
	(1.00)			(0.30)	(2.39)	(0.31)	(10.0)
2^{13}	1.26	TO	TO	0.456	2.88	0.448	14.0
	(1.00)			(0.36)	(2.28)	(0.35)	(11.1)
2^{14}	7.23	TO	TO	2.02	12.9	2.24	67.9
	(1.00)			(0.28)	(1.78)	(0.31)	(9.39)
2^{15}	34.1	TO	TO	8.71	62.9	10.9	324
	(1.00)			(0.26)	(1.84)	(0.32)	(9.50)

Table 19. Push-relabel, RMF-long family [code 1] (3 instances per size). Plots were cut off at $n = 2^{23}$. The family does not meet the structural prerequisites for SML AFP [code 27].

n	SML HL	SML FIFO	SML AFP [code 27]	LEMON CSR [code 42]	LEMON linked [code 42]	JGraphT [code 34]	SciPy Dinic CSR [code 57]	SciPy Dinic prebuilt [code 57]	NetworkX [code 46]	GC [code 10]
4096	0.047	0.056	n/a	0.007	0.009	0.150	0.081	0.080	0.941	0.004
	(1.00)	(1.17)		(0.15)	(0.19)	(3.17)	(1.72)	(1.69)	(19.9)	(0.08)
9000	0.151	0.201	n/a	0.021	0.034	0.320	0.288	0.290	1.98	0.009
	(1.00)	(1.33)		(0.14)	(0.23)	(2.12)	(1.91)	(1.92)	(13.1)	(0.06)
15488	0.377	0.714	n/a	0.050	0.141	0.610	0.816	0.817	3.36	0.018
	(1.00)	(1.89)		(0.13)	(0.38)	(1.62)	(2.17)	(2.17)	(8.93)	(0.05)
30589	0.914	2.01	n/a	0.105	0.240	1.13	1.99	1.80	6.77	0.038
	(1.00)	(2.20)		(0.11)	(0.26)	(1.24)	(2.18)	(1.97)	(7.41)	(0.04)
65536	2.12	5.63	n/a	0.445	1.72	2.40	4.61	4.69	20.1	0.084
	(1.00)	(2.65)		(0.21)	(0.81)	(1.13)	(2.17)	(2.21)	(9.48)	(0.04)
130321	5.57	14.4	n/a	1.09	5.15	6.81	12.4	12.2	48.6	0.232
	(1.00)	(2.58)		(0.20)	(0.93)	(1.22)	(2.23)	(2.20)	(8.72)	(0.04)
270319	29.4	100	n/a	4.78	48.6	14.9	44.5	44.2	142	1.06
	(1.00)	(3.41)		(0.16)	(1.66)	(0.51)	(1.52)	(1.51)	(4.83)	(0.04)
527067	147	467	n/a	22.7	300	31.9	147	150	358	1.47
	(1.00)	(3.17)		(0.15)	(2.04)	(0.22)	(1.00)	(1.02)	(2.43)	(0.01)
1046528	509	TO	n/a	51.5	TO	80.4	743	792	923	4.53
	(1.00)			(0.10)		(0.16)	(1.46)	(1.56)	(1.82)	(0.01)
2090912	TO	TO	n/a	267	TO	200	TO	TO	TO	6.56
				(–)		(–)				(–)
4147200	TO	TO	n/a	827	TO	469	TO	TO	TO	19.4
				(–)		(–)				(–)
8444736	TO	TO	n/a	TO	TO	TO	TO	TO	TO	90.2
										(–)
16777216	TO	TO	n/a	TO	TO	TO	TO	TO	TO	242
										(–)
33460368	TO	TO	n/a	TO	TO	TO	TO	TO	TO	598
										(–)
67109224	TO	TO	n/a	TO	TO	TO	TO	TO	TO	TO

Table 20. Push-relabel, WASH-RLG-long family [code 2] (3 instances per size). Plots were cut off at $n = 2^{23}$.

n	SML HL	SML FIFO	SML AFP [code 27]	LEMON CSR [code 42]	LEMON linked [code 42]	JGraphT [code 34]	SciPy Dinic CSR [code 57]	SciPy Dinic prebuilt [code 57]	NetworkX [code 46]	GC [code 10]
4098	0.028	0.027	0.310	0.009	0.009	0.117	0.028	0.028	0.824	0.005
	(1.00)	(0.94)	(10.9)	(0.33)	(0.32)	(4.11)	(0.99)	(0.97)	(29.0)	(0.16)
8194	0.086	0.121	1.23	0.042	0.042	0.280	0.097	0.096	1.76	0.010
	(1.00)	(1.40)	(14.2)	(0.48)	(0.48)	(3.24)	(1.12)	(1.11)	(20.3)	(0.11)
16386	0.264	0.430	3.39	0.116	0.113	0.600	0.338	0.339	2.96	0.019
	(1.00)	(1.63)	(12.8)	(0.44)	(0.43)	(2.27)	(1.28)	(1.28)	(11.2)	(0.07)
32770	0.889	1.60	10.2	0.640	0.599	1.23	1.13	1.12	7.09	0.042
	(1.00)	(1.80)	(11.5)	(0.72)	(0.67)	(1.38)	(1.27)	(1.26)	(7.97)	(0.05)
65538	2.54	3.67	OVF	1.47	1.85	2.38	2.69	2.60	14.2	0.080
	(1.00)	(1.44)		(0.58)	(0.73)	(0.94)	(1.06)	(1.02)	(5.58)	(0.03)
131074	6.89	8.76	OVF	3.30	3.05	4.47	8.93	8.98	28.8	0.170
	(1.00)	(1.27)		(0.48)	(0.44)	(0.65)	(1.30)	(1.30)	(4.18)	(0.02)
262146	36.6	35.8	OVF	3.63	3.39	11.7	35.5	34.0	64.4	0.347
	(1.00)	(0.98)		(0.10)	(0.09)	(0.32)	(0.97)	(0.93)	(1.76)	(0.01)
524290	273	299	OVF	90.2	92.9	29.4	130	129	154	0.748
	(1.00)	(1.09)		(0.33)	(0.34)	(0.11)	(0.48)	(0.47)	(0.56)	(0.00)
1048578	TO	TO	OVF	139	138	72.4	498	512	304	1.01
				(–)	(–)	(–)	(–)	(–)	(–)	(–)
2097154	TO	TO	OVF	137	134	180	TO	TO	568	2.04
				(–)	(–)	(–)			(–)	(–)
4194306	TO	TO	OVF	$350^{1/3}$	$353^{1/3}$	459	TO	TO	TO	4.69
				(–)	(–)	(–)				(–)
8388610	TO	TO	OVF	TO	TO	TO	TO	TO	TO	9.33
										(–)
16777218	TO	TO	OVF	TO	TO	TO	TO	TO	TO	17.3
										(–)
33554434	TO	TO	OVF	TO	TO	TO	TO	TO	TO	38.8
										(–)
67108866	TO	TO	OVF	TO	TO	TO	TO	TO	TO	81.4
										(–)

Table 21. Push-relabel, AK family [code 8] (1 instance per size, median of 3 repeats).

n	SML HL	SML FIFO	SML AFP [code 27]	LEMON CSR [code 42]	LEMON linked [code 42]	JGraphT [code 34]	SciPy Dinic CSR [code 57]	SciPy Dinic prebuilt [code 57]	NetworkX [code 46]	GC [code 10]
4102	0.120	0.072	0.392	0.003	0.006	0.460	0.135	0.294	8.81	0.014
	(1.00)	(0.60)	(3.27)	(0.02)	(0.05)	(3.84)	(1.13)	(2.46)	(73.6)	(0.11)
8198	0.280	0.273	1.43	0.007	0.014	0.930	0.539	1.17	32.0	0.057
	(1.00)	(0.97)	(5.11)	(0.02)	(0.05)	(3.32)	(1.92)	(4.18)	(114)	(0.20)
16390	1.61	1.09	3.69	0.019	0.047	2.63	2.16	3.13	127	0.233
	(1.00)	(0.68)	(2.30)	(0.01)	(0.03)	(1.64)	(1.34)	(1.95)	(79.3)	(0.14)
32774	4.41	3.16	11.9	0.058	0.090	7.39	8.76	9.63	523	0.889
	(1.00)	(0.72)	(2.70)	(0.01)	(0.02)	(1.68)	(1.99)	(2.18)	(119)	(0.20)
65542	15.0	9.14	OVF	0.198	0.242	32.5	34.9	35.5	TO	2.19
	(1.00)	(0.61)		(0.01)	(0.02)	(2.17)	(2.33)	(2.37)		(0.15)
131078	55.3	33.0	OVF	0.730	0.793	392	138	138	TO	6.96
	(1.00)	(0.60)		(0.01)	(0.01)	(7.09)	(2.51)	(2.51)		(0.13)
262150	222	129	OVF	3.41	3.68	TO	553	553	TO	28.3
	(1.00)	(0.58)		(0.02)	(0.02)		(2.49)	(2.49)		(0.13)
524294	883	510	OVF	11.9	11.9	TO	TO	TO	TO	111
	(1.00)	(0.58)		(0.01)	(0.01)					(0.13)
1048582	TO	TO	OVF	43.8	43.0	TO	TO	TO	TO	441
				(–)	(–)					(–)

Table 22. Hungarian method, low-cost family [code 3] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	SciPy LSA [code 52]	SciPy MWFBM CSR [code 53]
2^{11}	0.014	0.817	16.8	0.025	0.003
	(1.00)	(57.6)	(1187)	(1.78)	(0.22)
2^{12}	0.053	1.47	60.4	0.122	0.007
	(1.00)	(27.8)	(1139)	(2.31)	(0.14)
2^{13}	0.209	3.69	215	0.813	0.020
	(1.00)	(17.6)	(1026)	(3.88)	(0.10)
2^{14}	0.786	11.7	885	3.93	0.061
	(1.00)	(14.9)	(1126)	(5.01)	(0.08)
2^{15}	3.25	46.0	TO	21.2	0.175
	(1.00)	(14.2)		(6.53)	(0.05)

n	JGraphT KM with GC [code 32]	JGraphT MWBM CSR [code 33]	JGraphT MWBM hash [code 33]	CSA CSR [code 9]	OR-Tools [code 11]
2^{11}	2.60	0.107	0.238	0.001	0.003
	(184)	(7.55)	(16.8)	(0.09)	(0.22)
2^{12}	24.7	0.314	0.655	0.003	0.005
	(467)	(5.93)	(12.4)	(0.05)	(0.10)
2^{13}	210	1.02	1.63	0.007	0.012
	(1001)	(4.86)	(7.78)	(0.03)	(0.06)
2^{14}	TO	3.28	4.36	0.016	0.031
		(4.18)	(5.55)	(0.02)	(0.04)
2^{15}	TO	11.1	17.3	0.041	0.071
		(3.41)	(5.32)	(0.01)	(0.02)

Table 23. Hungarian method, two-cost family [code 4] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	SciPy LSA [code 52]	SciPy MWFBM CSR [code 53]
2^{11}	0.005	0.537	4.81	0.012	0.002
	(1.00)	(102)	(916)	(2.32)	(0.46)
2^{12}	0.016	0.763	13.6	0.058	0.006
	(1.00)	(47.9)	(853)	(3.62)	(0.40)
2^{13}	0.054	1.22	40.4	0.351	0.020
	(1.00)	(22.5)	(748)	(6.50)	(0.36)
2^{14}	0.163	2.61	137	2.32	0.069
	(1.00)	(16.0)	(838)	(14.2)	(0.42)
2^{15}	0.588	6.28	320	20.9	0.251
	(1.00)	(10.7)	(544)	(35.5)	(0.43)

n	JGraphT KM with GC [code 32]	JGraphT MWBM CSR [code 33]	JGraphT MWBM hash [code 33]	CSA CSR [code 9]	OR-Tools [code 11]
2^{11}	2.43	0.108	0.192	0.005	0.009
	(463)	(20.5)	(36.5)	(0.93)	(1.69)
2^{12}	23.3	0.288	0.489	0.012	0.018
	(1460)	(18.1)	(30.7)	(0.76)	(1.13)
2^{13}	201	0.758	1.18	0.027	0.042
	(3715)	(14.0)	(21.9)	(0.50)	(0.78)
2^{14}	TO	2.33	3.11	0.064	0.095
		(14.3)	(19.1)	(0.39)	(0.58)
2^{15}	TO	6.68	9.89	0.152	0.218
		(11.4)	(16.8)	(0.26)	(0.37)

Table 24. Hungarian method, geometric family [code 5] (10 instances per size, median of 3 repeats each).

n	SML	Scala	Go	SciPy LSA [code 52]	SciPy MWFBM CSR [code 53]
256	0.005	0.494	8.20	0.0006	0.005
	(1.00)	(106)	(1766)	(0.13)	(0.99)
512	0.024	1.04	42.8	0.003	0.013
	(1.00)	(43.5)	(1784)	(0.11)	(0.56)
1024	0.143	3.48	279	0.012	0.045
	(1.00)	(24.4)	(1953)	(0.09)	(0.32)
2^{11}	0.945	19.9	$1186^{2/10}$	0.069	0.198
	(1.00)	(21.0)	(1256)	(0.07)	(0.21)
2^{12}	6.03	133	TO	0.333	0.817
	(1.00)	(22.0)		(0.06)	(0.14)

n	JGraphT KM with GC [code 32]	JGraphT MWBM CSR [code 33]	JGraphT MWBM hash [code 33]	CSA CSR [code 9]	OR-Tools [code 11]
256	0.036	0.093	0.105	0.0006	0.004
	(7.83)	(19.9)	(22.7)	(0.12)	(0.88)
512	0.321	0.313	0.425	0.002	0.009
	(13.4)	(13.0)	(17.7)	(0.09)	(0.38)
1024	2.38	1.28	1.49	0.009	0.052
	(16.7)	(8.97)	(10.4)	(0.06)	(0.37)
2^{11}	34.6	9.13	15.7	0.046	0.386
	(36.6)	(9.66)	(16.7)	(0.05)	(0.41)
2^{12}	480	50.8	90.7	0.300	3.97
	(79.7)	(8.43)	(15.0)	(0.05)	(0.66)

b-Matching. For b-matching, we set the capacities $b(x) = \mathsf{min}\{\beta, \deg(x)\}$ where β is selected uniformly at random between 1 and 20 and $\deg(x)$ is the vertex degree.

Table 25. Matroid intersection, unweighted problems on the low-cost and two-cost families [code 4], which share their graphs (5 instances per size, run once in each family, mean over both runs).

n	SML	Scala	Go	Sage (our oracle) [code 49]	Sage (plain) [code 49,50,51]
Matching					
32	<0.001 (–)	0.038 (–)	0.008 (–)	0.003 (–)	0.021 (–)
64	<0.001 (–)	0.068 (–)	0.025 (–)	0.009 (–)	0.137 (–)
128	0.0006 (1.00)	0.123 (206)	0.094 (157)	0.053 (88.8)	1.49 (2480)
256	0.002 (1.00)	0.229 (99.3)	0.337 (146)	0.337 (147)	14.8 (6421)
512	0.009 (1.00)	0.488 (53.6)	1.52 (167)	4.42 (485)	176 (19357)
1024	0.032 (1.00)	0.830 (26.0)	5.70 (179)	46.3 (1452)	TO
2^{11}	0.146 (1.00)	2.27 (15.5)	28.1 (193)	TO	TO
2^{12}	0.394 (1.00)	7.29 (18.5)	116 (294)	TO	TO
2^{13}	1.67 (1.00)	29.1 (17.4)	476 (284)	TO	TO
2^{14}	7.93 (1.00)	123 (15.5)	TO	TO	TO
2^{15}	37.0 (1.00)	530 (14.3)	TO	TO	TO
b-matching					
32	<0.001 (–)	0.013 (–)	0.019 (–)	0.011 (–)	0.233 (–)
64	<0.001 (–)	0.039 (–)	0.067 (–)	0.076 (–)	2.56 (–)
128	0.002 (1.00)	0.234 (117)	0.623 (312)	1.63 (815)	40.2 (20079)
256	0.018 (1.00)	0.614 (34.9)	4.79 (272)	31.2 (1775)	591 (33606)
512	0.121 (1.00)	2.24 (18.5)	30.1 (249)	539 (4460)	TO
1024	0.885 (1.00)	11.8 (13.3)	201 (227)	TO	TO
2^{11}	4.83 (1.00)	59.6 (12.4)	1069 (221)	TO	TO
2^{12}	22.0 (1.00)	267 (12.2)	TO	TO	TO
2^{13}	123 (1.00)	TO	TO	TO	TO
2^{14}	537 (1.00)	TO	TO	TO	TO
2^{15}	TO	TO	TO	TO	TO

Table 26. Matroid intersection, weighted problems on the low-cost family [code 3] (5 instances per size, one run each). The unweighted problems are in Table 25.

n	SML	Scala	Go	Sage (our oracle) [code 48]	Sage (plain) [code 48,50,51]
Weighted matching					
32	<0.001 (–)	0.028 (–)	0.067 (–)	0.005 (–)	0.047 (–)
64	0.001 (1.00)	0.086 (86.4)	0.253 (253)	0.030 (29.6)	0.954 (954)
128	0.004 (1.00)	0.154 (40.6)	0.807 (212)	0.198 (52.2)	19.9 (5231)
256	0.011 (1.00)	0.300 (28.3)	2.24 (212)	1.49 (140)	398 (37580)
512	0.037 (1.00)	0.826 (22.3)	7.38 (199)	11.7 (317)	TO
1024	0.119 (1.00)	2.29 (19.2)	22.1 (186)	103 (868)	TO
2^{11}	0.456 (1.00)	8.12 (17.8)	88.1 (193)	868 (1904)	TO
2^{12}	1.69 (1.00)	29.3 (17.4)	349 (206)	TO	TO
2^{13}	6.81 (1.00)	126 (18.5)	TO	TO	TO
2^{14}	31.0 (1.00)	577 (18.6)	TO	TO	TO
2^{15}	157 (1.00)	TO	TO	TO	TO
Weighted b-matching					
32	0.001 (1.00)	0.021 (21.2)	0.209 (209)	0.024 (23.8)	1.90 (1905)
64	0.003 (1.00)	0.060 (23.2)	0.662 (255)	0.166 (63.7)	41.2 (15831)
128	0.011 (1.00)	0.215 (19.9)	2.74 (254)	2.12 (197)	$1015^{1/5}$ (93994)
256	0.044 (1.00)	0.840 (19.2)	10.8 (248)	23.9 (546)	TO
512	0.193 (1.00)	3.13 (16.2)	44.1 (229)	259 (1346)	TO
1024	0.997 (1.00)	13.5 (13.6)	192 (193)	TO	TO
2^{11}	4.69 (1.00)	61.8 (13.2)	866 (185)	TO	TO
2^{12}	22.7 (1.00)	290 (12.8)	TO	TO	TO
2^{13}	114 (1.00)	TO	TO	TO	TO
2^{14}	591 (1.00)	TO	TO	TO	TO
2^{15}	TO	TO	TO	TO	TO

Table 27. Matroid intersection, weighted problems on the two-cost family [code 4] (5 instances per size, one run each). The unweighted problems are in Table 25.

n	SML	Scala	Go	Sage (our oracle) [code 48]	Sage (plain) [code 48,50,51]
Weighted matching					
32	<0.001 (–)	0.006 (–)	0.013 (–)	0.005 (–)	0.054 (–)
64	<0.001 (–)	0.023 (–)	0.051 (–)	0.027 (–)	1.04 (–)
128	0.001 (1.00)	0.077 (77.0)	0.191 (191)	0.179 (179)	23.1 (23107)
256	0.005 (1.00)	0.167 (33.4)	0.810 (162)	1.33 (266)	462 (92370)
512	0.022 (1.00)	0.488 (21.8)	3.42 (153)	10.4 (463)	TO
1024	0.097 (1.00)	1.57 (16.2)	15.0 (154)	91.5 (942)	TO
2^{11}	0.391 (1.00)	5.86 (15.0)	65.0 (166)	755 (1929)	TO
2^{12}	1.59 (1.00)	24.0 (15.0)	280 (175)	TO	TO
2^{13}	7.17 (1.00)	106 (14.7)	1137 (158)	TO	TO
2^{14}	32.6 (1.00)	459 (14.1)	TO	TO	TO
2^{15}	151 (1.00)	TO	TO	TO	TO
Weighted b-matching					
32	0.0002 (1.00)	0.008 (38.0)	0.051 (257)	0.022 (112)	1.74 (8711)
64	0.002 (1.00)	0.025 (15.7)	0.235 (147)	0.150 (93.8)	37.0 (23125)
128	0.006 (1.00)	0.116 (19.3)	1.32 (220)	2.11 (351)	$792^{1/5}$ (131998)
256	0.030 (1.00)	0.573 (19.2)	7.48 (251)	23.3 (781)	TO
512	0.165 (1.00)	2.88 (17.5)	39.2 (238)	257 (1557)	TO
1024	1.07 (1.00)	15.0 (14.0)	209 (196)	TO	TO
2^{11}	5.84 (1.00)	76.0 (13.0)	1050 (180)	TO	TO
2^{12}	30.3 (1.00)	373 (12.3)	TO	TO	TO
2^{13}	164 (1.00)	TO	TO	TO	TO
2^{14}	856 (1.00)	TO	TO	TO	TO
2^{15}	TO	TO	TO	TO	TO

Table 28. Matroid intersection, geometric family [code 5] (5 instances per size, one run each).

n	SML	Scala	Go	Sage (our oracle) [code 48,49]	Sage (plain) [code 48,49,50,51]
Matching					
32	<0.001 (–)	0.007 (–)	0.012 (–)	0.004 (–)	0.038 (–)
64	0.001 (1.00)	0.043 (43.4)	0.084 (84.0)	0.023 (23.0)	0.650 (650)
128	0.003 (1.00)	0.281 (82.7)	0.666 (196)	0.296 (87.1)	20.0 (5892)
256	0.029 (1.00)	0.751 (26.3)	5.34 (187)	4.32 (151)	680 (23790)
512	0.233 (1.00)	4.56 (19.6)	39.7 (171)	66.3 (285)	TO
1024	1.72 (1.00)	37.4 (21.7)	306 (178)	TO	TO
2^{11}	13.1 (1.00)	OOM	TO	TO	TO
2^{12}	105 (1.00)	OOM	TO	TO	TO
b-matching					
32	0.0002 (1.00)	0.044 (222)	0.076 (381)	0.050 (252)	0.882 (4409)
64	0.003 (1.00)	0.258 (99.3)	0.877 (337)	0.759 (292)	25.2 (9702)
128	0.012 (1.00)	0.566 (48.8)	4.27 (368)	12.9 (1113)	$908^{2/5}$ (78304)
256	0.104 (1.00)	2.44 (23.6)	30.9 (298)	132 (1277)	TO
512	0.374 (1.00)	9.17 (24.5)	94.9 (254)	125 (333)	TO
1024	8.53 (1.00)	176 (20.6)	TO	TO	TO
2^{11}	42.9 (1.00)	OOM	TO	TO	TO
2^{12}	402 (1.00)	OOM	TO	TO	TO
Weighted matching					
32	0.002 (1.00)	0.135 (61.2)	0.613 (279)	0.009 (4.00)	0.109 (49.5)
64	0.057 (1.00)	1.51 (26.6)	17.2 (302)	0.111 (1.94)	5.89 (103)
128	1.86 (1.00)	40.4 (21.7)	556 (298)	1.90 (1.02)	367 (197)
256	56.1 (1.00)	$1120^{4/5}$ (20.0)	TO	40.8 (0.73)	TO
512	TO	TO	TO	1089 (–)	TO
1024	TO	TO	TO	TO	TO
2^{11}	TO	TO	TO	TO	TO
2^{12}	TO	TO	TO	TO	TO
Weighted b-matching					
32	0.004 (1.00)	0.111 (25.3)	1.41 (320)	0.084 (19.0)	9.99 (2270)
64	0.088 (1.00)	2.04 (23.3)	33.2 (378)	2.14 (24.4)	$786^{2/5}$ (8954)
128	3.93 (1.00)	81.4 (20.7)	$372^{1/5}$ (94.5)	54.0 (13.7)	TO
256	154 (1.00)	TO	TO	TO	TO
512	TO	TO	TO	TO	TO
1024	TO	TO	TO	TO	TO
2^{11}	TO	TO	TO	TO	TO
2^{12}	TO	TO	TO	TO	TO

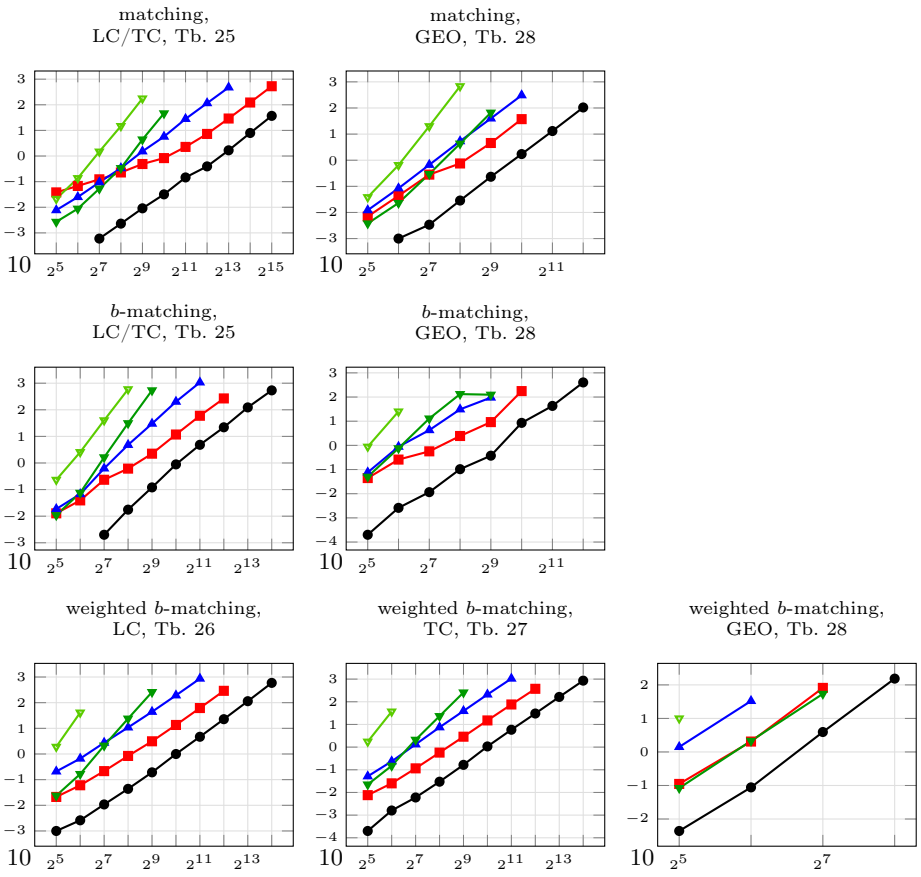


Fig. 6. Matroid Intersection, matching, b-matching and weighted b-matching; on matching and b-matching, LC and TC share their instances (LC/TC): SML, Scala, Go, Sage (our oracle), Sage (plain).

Hybrid Solver for Minimum-Cost Flows. We evaluated on 18 families [49] in total, generated by NETGEN [47], GRIDGEN, GOTO [33] and GRIDGRAPH or converted from road networks [22] and vision instances. These are `goto_8` [code 12], `goto_sr` [code 13], `grid_long` [code 17], `grid_square` [code 18], `grid_wide` [code 19], `gridgen_8` [code 14], `gridgen_deg` [code 15], `gridgen_sr` [code 16], `netgen_8` [code 20], `netgen_deg` [code 21], `netgen_lo_8` [code 22], `netgen_lo_sr` [code 23], `netgen_sr` [code 24], `road_flow` [code 25], `road_paths` [code 25], `vision_inv` [code 26], `vision_prop` [code 26] and `vision_rnd` [code 26]. For the sake of brevity, we only display five of them, `gridgen_deg`, `netgen_deg`, `netgen_lo_sr` (which performed worst), `road_paths` (within the range of $\leq$ 10 % to $\leq$ 30 % for all others) and `goto_8` (with exceptionally good performance), in Tables 29 to 33. The first two proceed by density at fixed node count, the third by node count with increasing density, and the last two by the number of nodes at fixed density. As oracles we use network simplex [code 41], cost scaling [code 37], capacity scaling and successive shortest paths [code 36] whose code was taken from LEMON. For every family, size and algorithm, five different instances were run once each, and every time is the mean of these five runs. All times are compute times in milliseconds, measured after parsing. *Oracle* is the C++ solver's initialisation and main loop, *Cert.* the emission of the certificate and *Check* the verified check. *Glue* is what is left of the solver's entire compute time *Total*. The last column is the relative cost of verification, i.e. Total divided by Oracle. For each size, the row of the algorithm whose oracle is fastest is shaded grey. Note that we here use a higher timeout of 2 hours.

Table 29. Hybrid min-cost flow solver, `gridgen_deg` [code 15] family generated by GRIDGEN. A density sweep induced by degree increase at the fixed node count $n = 4097$. The first column is the average in-degree m/n which always equals the density.

deg	m	Alg.	Oracle	Cert.	Check	Glue	Total	Total/Oracle
4	16388	NS	5.70	0.042	1.59	0.964	8.30	1.46
		COS	13.0	0.227	1.72	1.39	16.3	1.26
		CAS	65.1	0.152	1.62	1.44	68.3	1.05
		SSP	9.86	0.101	1.59	1.38	12.9	1.31
8	32776	NS	14.9	0.196	2.08	1.60	18.8	1.26
		COS	27.6	0.439	2.33	2.51	32.9	1.19
		CAS	228	0.385	2.08	2.56	233	1.02
		SSP	544	0.363	2.10	2.60	549	1.01
16	65552	NS	23.0	0.321	3.97	3.05	30.0	1.32
		COS	38.0	0.810	4.53	4.74	48.0	1.26
		CAS	420	0.661	3.98	4.95	430	1.02
		SSP	1017	0.694	3.96	4.34	1026	1.01
32	131104	NS	30.0	0.621	11.5	6.06	48.0	1.60
		COS	60.0	1.46	12.6	9.38	83.0	1.39
		CAS	812	1.12	11.3	9.93	835	1.03
		SSP	1208	1.09	11.4	9.56	1230	1.02
64	262208	NS	48.0	1.20	23.2	11.4	84.0	1.74
		COS	108	2.70	25.3	19.3	155	1.44
		CAS	2182	1.93	23.2	19.9	2227	1.02
		SSP	1964	1.94	23.1	21.0	2010	1.02
128	524416	NS	74.0	2.21	45.4	24.1	146	1.97
		COS	204	4.47	49.8	40.4	299	1.46
		CAS	6048	3.47	45.6	42.9	6140	1.01
		SSP	2947	3.15	45.5	42.3	3038	1.03
256	1048832	NS	107	3.93	60.6	80.9	252	2.37
		COS	459	8.47	70.3	127	665	1.45
		CAS	16650	6.45	60.6	136	16853	1.01
		SSP	4574	6.48	60.6	139	4780	1.04
512	2097664	NS	174	8.59	181	109	472	2.71
		COS	1149	16.5	199	221	1585	1.38
		CAS	115550	13.5	180	237	115980	1.00
		SSP	6478	13.8	183	245	6920	1.07
1024	4195328	NS	316	16.6	365	233	930	2.95
		COS	2507	32.6	401	481	3421	1.36
		CAS	501030	27.6	361	511	501930	1.00
		SSP	12384	25.3	361	511	13281	1.07
2^{11}	8390656	NS	546	29.1	728	438	1741	3.19
		COS	5715	56.1	797	977	7545	1.32
		CAS	1424300	60.5	734	1005	1426100	1.00
		SSP	24552	52.0	724	1041	26369	1.07
2^{12}	16781312	NS	957	59.2	1471	889	3377	3.53
		COS	11638	127	1614	1988	15367	1.32
		CAS	TO	TO	TO	TO	TO	TO
		SSP	43839	107	1455	2138	47539	1.08

Table 30. Hybrid min-cost flow solver, `netgen_deg` [code 21] family generated by NETGEN [47]. A density sweep by increasing degrees at the fixed node count $n = 4096$. The first column is the average in-degree m/n which is equal to the out-degree and the density.

deg	m	Alg.	Oracle	Cert.	Check	Glue	Total	Total/Oracle
2	8192	NS	7.79	0.021	0.649	0.697	9.15	1.18
		COS	20.7	0.119	0.696	0.946	22.5	1.08
		CAS	103	0.081	0.674	0.965	105	1.02
		SSP	123	0.084	0.675	0.961	125	1.01
4	16384	NS	10.9	0.048	1.61	1.00	13.5	1.24
		COS	19.3	0.244	1.72	1.42	22.7	1.18
		CAS	286	0.198	1.62	1.47	290	1.01
		SSP	212	0.195	1.63	1.46	215	1.01
8	32768	NS	16.8	0.198	2.10	2.43	21.6	1.28
		COS	26.8	0.434	2.36	3.45	33.0	1.23
		CAS	972	0.377	2.09	3.46	978	1.01
		SSP	537	0.370	2.12	3.46	543	1.01
16	65536	NS	24.0	0.330	4.07	3.06	31.0	1.31
		COS	38.0	0.803	4.62	4.80	48.0	1.27
		CAS	3843	0.802	4.06	4.14	3852	1.00
		SSP	911	0.720	4.06	4.76	921	1.01
32	131072	NS	36.0	0.628	11.5	6.03	54.0	1.50
		COS	56.0	1.45	12.7	9.38	79.0	1.42
		CAS	8854	1.26	11.5	10.2	8877	1.00
		SSP	1265	1.12	11.5	9.39	1287	1.02
64	262144	NS	52.0	1.21	22.7	11.7	88.0	1.68
		COS	94.0	2.67	25.1	19.3	141	1.50
		CAS	23755	2.40	22.8	18.8	23799	1.00
		SSP	1910	1.94	22.8	19.3	1954	1.02
128	524288	NS	84.0	2.29	45.3	26.6	158	1.88
		COS	191	4.46	49.6	38.4	283	1.49
		CAS	75297	4.21	45.5	41.3	75388	1.00
		SSP	3335	3.43	45.2	40.4	3424	1.03
256	1048576	NS	123	4.20	89.8	56.9	274	2.23
		COS	409	8.52	99.4	83.8	601	1.47
		CAS	281740	8.03	91.1	90.9	281930	1.00
		SSP	4564	6.09	89.8	84.1	4744	1.04
512	2097152	NS	190	8.44	181	117	495	2.61
		COS	1058	16.0	197	182	1453	1.37
		CAS	924820	15.7	182	192	925210	1.00
		SSP	7854	12.7	179	186	8231	1.05
1024	4194304	NS	339	15.8	358	243	956	2.82
		COS	2309	32.1	396	408	3146	1.36
		CAS	TO	TO	TO	TO	TO	TO
		SSP	13945	26.2	357	390	14718	1.05
2^{11}	8388608	NS	575	29.6	716	512	1833	3.19
		COS	5100	56.4	797	828	6782	1.33
		CAS	TO	TO	TO	TO	TO	TO
		SSP	26008	48.8	709	807	27572	1.06
2^{12}	16252992	NS	916	53.5	1489	919	3378	3.69
		COS	10320	110	1638	1501	13569	1.31
		CAS	TO	TO	TO	TO	TO	TO
		SSP	51289	115	1522	1585	54511	1.06

Table 31. Hybrid min-cost flow solver, `netgen_lo_sr` [code 23] family generated by NETGEN [47]. A node sweep at the density $m = n^{1.5}$.

n	m	Alg.	Oracle	Cert.	Check	Glue	Total	Total/Oracle
256	4096	NS	0.266	0.010	0.279	0.287	0.842	3.17
		COS	0.936	0.022	0.324	0.393	1.68	1.79
		CAS	0.635	0.017	0.279	0.405	1.34	2.10
		SSP	0.792	0.017	0.284	0.398	1.49	1.88
512	11585	NS	0.696	0.015	1.04	0.576	2.33	3.34
		COS	2.60	0.069	1.16	0.893	4.72	1.82
		CAS	2.09	0.025	1.04	0.903	4.06	1.94
		SSP	3.11	0.025	1.03	0.895	5.06	1.63
1024	32768	NS	1.87	0.163	2.84	1.50	6.37	3.40
		COS	7.13	0.337	3.14	2.35	13.0	1.82
		CAS	7.01	0.244	2.86	2.42	12.5	1.79
		SSP	10.3	0.248	2.82	2.37	15.7	1.53
2^{11}	92682	NS	6.21	0.352	7.85	3.84	18.2	2.94
		COS	20.4	0.900	8.70	6.43	36.4	1.79
		CAS	27.7	0.702	7.89	6.71	43.0	1.55
		SSP	40.0	0.715	7.95	6.68	55.3	1.38
2^{12}	262144	NS	17.8	1.01	22.5	11.6	52.9	2.96
		COS	62.5	2.65	25.0	18.8	109	1.74
		CAS	86.5	1.94	22.6	19.1	130	1.50
		SSP	145	1.92	22.5	19.0	188	1.30
2^{13}	741455	NS	58.7	3.06	43.1	34.7	140	2.38
		COS	203	6.08	50.2	57.3	317	1.56
		CAS	278	4.54	43.1	59.5	386	1.39
		SSP	562	4.82	43.2	60.6	671	1.19
2^{14}	2097152	NS	203	8.52	123	117	450	2.22
		COS	822	16.7	142	187	1168	1.42
		CAS	1081	11.4	124	194	1411	1.30
		SSP	2104	12.9	122	198	2437	1.16
2^{15}	5931642	NS	754	20.8	354	298	1427	1.89
		COS	2832	44.3	409	566	3851	1.36
		CAS	3539	30.9	353	597	4520	1.28
		SSP	7417	33.6	353	594	8397	1.13
2^{16}	16777216	NS	2822	54.0	1102	1102	5080	1.80
		COS	9482	114	1267	1731	12594	1.33
		CAS	12679	104	1106	1886	15775	1.24
		SSP	27502	109	1106	1899	30616	1.11

Table 32. Hybrid min-cost flow solver, `road_paths` [code 25] family on the road networks of [22]. US state road network (real data). The first column lists the instances, ordered by increasing number of nodes n.

Instance	n	m	m/n	Alg.	Oracle	Cert.	Check	Glue	Total	Total/Oracle
01_DC	9559	29768	3.11	NS	15.2	0.165	2.09	1.85	19.3	1.27
				COS	31.0	0.415	2.24	2.54	36.2	1.17
				CAS	4.75	0.164	2.08	2.66	9.65	2.03
				SSP	4.73	0.174	2.08	2.69	9.67	2.05
02_DE	49109	120576	2.46	NS	143	1.23	9.11	8.05	162	1.13
				COS	239	2.25	9.27	10.7	261	1.09
				CAS	23.2	0.634	8.63	11.8	44.3	1.91
				SSP	23.2	0.617	8.83	11.6	44.3	1.91
03_NH	116920	265402	2.27	NS	678	2.28	19.2	17.9	718	1.06
				COS	845	4.13	20.6	24.1	894	1.06
				CAS	76.7	1.47	19.3	27.3	125	1.63
				SSP	76.9	1.45	19.2	26.9	124	1.62
04_NV	261155	620924	2.38	NS	2854	4.53	45.6	40.9	2945	1.03
				COS	3339	8.91	49.0	58.0	3455	1.03
				CAS	280	2.98	46.0	63.6	393	1.40
				SSP	279	2.92	45.3	63.5	391	1.40
05_WI	519157	1266876	2.44	NS	9150	10.7	96.2	87.1	9344	1.02
				COS	8859	19.8	103	122	9104	1.03
				CAS	1121	6.27	96.7	144	1368	1.22
				SSP	1116	5.74	96.6	142	1360	1.22
06_FL	1048506	2653624	2.53	NS	TO	TO	TO	TO	TO	TO
				COS	TO	TO	TO	TO	TO	TO
				CAS	1842	12.5	211	309	2375	1.29
				SSP	1859	11.4	212	302	2384	1.28
07_TX	2073870	5156088	2.49	NS	TO	TO	TO	TO	TO	TO
				COS	TO	TO	TO	TO	TO	TO
				CAS	6688	21.4	427	590	7727	1.16
				SSP	6665	24.2	425	609	7723	1.16

Table 33. Hybrid min-cost flow solver, `goto_8` [code 12] family generated by GOTO [33]. A node sweep at the fixed density $m = 8n$.

n	m	Alg.	Oracle	Cert.	Check	Glue	Total	Total/Oracle
256	2^{11}	NS	0.638	0.011	0.175	0.211	1.03	1.62
		COS	2.81	0.024	0.186	0.260	3.28	1.17
		CAS	5.39	0.025	0.177	0.264	5.86	1.09
		SSP	32.9	0.045	0.170	0.276	33.4	1.01
2^{11}	2^{14}	NS	23.0	0.060	1.57	0.842	26.0	1.11
		COS	40.0	0.260	1.62	1.27	43.0	1.08
		CAS	132	0.205	1.57	1.29	135	1.02
		SSP	3672	0.220	1.54	1.24	3675	1.00
2^{12}	2^{15}	NS	108	0.293	2.26	2.49	113	1.05
		COS	139	0.598	2.34	3.34	145	1.04
		CAS	2054	0.362	2.25	3.39	2060	1.00
		SSP	26607	0.380	2.25	3.37	26613	1.00
2^{13}	2^{16}	NS	318	0.505	4.29	3.13	326	1.02
		COS	247	1.01	4.52	4.87	258	1.04
		CAS	1253	0.619	4.26	5.12	1263	1.01
		SSP	69189	0.622	4.27	5.11	69199	1.00
2^{14}	2^{17}	NS	1271	0.823	8.44	5.74	1286	1.01
		COS	655	1.74	9.01	9.79	676	1.03
		CAS	3523	1.12	8.53	10.3	3543	1.01
		SSP	274510	1.32	8.48	10.2	274530	1.00
2^{15}	2^{18}	NS	31593	1.69	17.2	12.1	31624	1.00
		COS	2342	3.04	18.7	22.2	2386	1.02
		CAS	185270	2.29	17.2	20.5	185310	1.00
		SSP	1270300	2.30	17.2	80.5	1270400	1.00
2^{16}	2^{19}	NS	35666	3.08	34.1	25.8	35729	1.00
		COS	5140	5.88	36.3	42.9	5225	1.02
		CAS	34066	4.14	34.1	44.8	34149	1.00
		SSP	3148600	4.13	34.1	61.8	3148700	1.00

Table 34. Hybrid min-cost flow solver, `goto_8` family (continued from Table 33).

n	m	Alg.	Oracle	Cert.	Check	Glue	Total	Total/Oracle
2^{17}	2^{20}	NS	152080	6.73	68.7	54.5	152210	1.00
		COS	19734	12.9	73.1	92.0	19912	1.01
		CAS	105990	8.02	68.9	93.1	106160	1.00
		SSP	TO	TO	TO	TO	TO	TO
2^{18}	2^{21}	NS	3203400	12.9	140	147	3203700	1.00
		COS	41971	25.8	155	211	42363	1.01
		CAS	TO	TO	TO	TO	TO	TO
		SSP	TO	TO	TO	TO	TO	TO
2^{19}	2^{22}	NS	2413900	24.9	290	285	2414500	1.00
		COS	270890	43.7	308	388	271630	1.00
		CAS	TO	TO	TO	TO	TO	TO
		SSP	TO	TO	TO	TO	TO	TO
2^{20}	2^{23}	NS	TO	TO	TO	TO	TO	TO
		COS	1009100	85.7	640	874	1010700	1.00
		CAS	TO	TO	TO	TO	TO	TO
		SSP	TO	TO	TO	TO	TO	TO
2^{21}	2^{24}	NS	TO	TO	TO	TO	TO	TO
		COS	656740	150	1489	1931	660310	1.00
		CAS	TO	TO	TO	TO	TO	TO
		SSP	TO	TO	TO	TO	TO	TO
2^{22}	2^{25}	NS	TO	TO	TO	TO	TO	TO
		COS	TO	TO	TO	TO	TO	TO
		CAS	TO	TO	TO	TO	TO	TO
		SSP	TO	TO	TO	TO	TO	TO

Hybrid Solver for Weighted Bipartite Matching. We ran the same pipeline with CSA [code 9] as oracle on the families of Tables 22 to 24, with columns as above and each time the mean over ten instances, each the median of three runs (Tables 35 to 37).

Table 35. Hybrid bipartite matching solver with CSA [code 9] as oracle, low-cost family [code 3] (10 instances per size, median of 3 repeats each). A node sweep at the density $m = n \log_2 n$.

n	m	Oracle	Cert.	Check	Glue	Total	Total/Oracle
2^{11}	22528	1.29	0.116	0.042	0.774	2.22	1.72
2^{12}	49152	2.90	0.247	0.092	1.64	4.88	1.68
2^{13}	106496	7.12	0.555	0.197	3.63	11.5	1.61
2^{14}	229376	16.3	1.35	0.437	8.25	26.3	1.62
2^{15}	491520	41.8	4.96	1.50	20.8	69.1	1.65
2^{16}	1048576	99.4	14.0	4.08	45.4	163	1.64
2^{17}	2228224	284	37.3	9.75	96.9	427	1.51
2^{18}	4718592	877	122	22.9	212	1234	1.41
2^{19}	9961472	2193	227	45.0	424	2890	1.32
2^{20}	20971520	6237	545	98.9	967	7849	1.26

Table 36. Hybrid bipartite matching solver with CSA [code 9] as oracle, two-cost family [code 4] (10 instances per size, median of 3 repeats each). A node sweep at the density $m = n \log_2 n$.

n	m	Oracle	Cert.	Check	Glue	Total	Total/Oracle
2^{11}	22528	4.89	0.116	0.041	0.745	5.79	1.18
2^{12}	49152	12.2	0.248	0.092	1.63	14.2	1.16
2^{13}	106496	27.5	0.583	0.198	3.65	31.9	1.16
2^{14}	229376	65.6	1.73	0.471	8.15	76.0	1.16
2^{15}	491520	150	5.23	1.49	20.3	177	1.18
2^{16}	1048576	356	14.9	3.96	43.4	418	1.18
2^{17}	2228224	1438	37.0	9.47	94.3	1579	1.10
2^{18}	4718592	4362	114	22.9	200	4699	1.08
2^{19}	9961472	10805	264	49.7	431	11550	1.07
2^{20}	20971520	31019	552	102	956	32630	1.05

Table 37. Hybrid bipartite matching solver with CSA [code 9] as oracle, geometric family [code 5] (10 instances per size, median of 3 repeats each). A node sweep on complete bipartite graphs, $m = n^2/4$.

n	m	Oracle	Cert.	Check	Glue	Total	Total/Oracle
256	2^{14}	0.545	0.061	0.023	0.577	1.21	2.21
512	2^{16}	2.09	0.410	0.090	2.31	4.90	2.35
1024	2^{18}	8.63	0.749	0.352	11.2	21.0	2.43
2^{11}	2^{20}	44.2	4.53	1.83	53.9	104	2.36
2^{12}	2^{22}	258	45.0	6.97	241	551	2.13
2^{13}	2^{24}	1348	171	34.6	1101	2654	1.97
2^{14}	2^{26}	6990	641	193	4000	11824	1.69

Code References

code 1. 13th DIMACS Implementation Challenge: Network Flows 2.0: genrmf, http://coral.ise.lehigh.edu/flow-challenge-2-0/files/2026/09/genrmfDimacs.zip, last accessed on 13/10/2026

code 2. 13th DIMACS Implementation Challenge: Network Flows 2.0: washington, https://coral.ise.lehigh.edu//wp-content/uploads/washington.c, last accessed on 13/10/2026

code 3. 1st DIMACS Implementation Challenge: Network Flows and Matching: assign.c, https://archive.dimacs.rutgers.edu/pub/netflow/generators/network/assign.c, last accessed on 13/10/2026

code 4. 1st DIMACS Implementation Challenge: Network Flows and Matching: assign.c and twocost.a, https://archive.dimacs.rutgers.edu/pub/netflow/generators/network/assign.c, with https://archive.dimacs.rutgers.edu/pub/netflow/generators/network/twocost.a. Last accessed on 13/10/2026

code 5. 1st DIMACS Implementation Challenge: Network Flows and Matching: dcube.c and geomasn.a, https://archive.dimacs.rutgers.edu/pub/netflow/generators/matching/dcube.c, with https://archive.dimacs.rutgers.edu/pub/netflow/generators/network/geomasn.a. Last accessed on 13/10/2026

code 6. 9th DIMACS Implementation Challenge: Shortest Paths: Code suite ch9-1.1 (spgrid, sprand), http://www.diag.uniroma1.it/~challenge9/code/ch9-1.1.tar.gz, last accessed on 13/10/2026
code 7. 9th DIMACS Implementation Challenge: Shortest Paths: USA-road-t, download, http://www.diag.uniroma1.it/~challenge9/download.shtml, last accessed on 13/10/2026
code 8. Cherkassky, B.V., Goldberg, A.V.: ak. PRF 1.8, Andrew Goldberg's Network Optimization Library, http://web.archive.org/web/2018id_/http://www.avglab.com/andrew/soft/prf.tar, last accessed on 13/10/2026
code 9. Goldberg, A.V.: CSA. Andrew Goldberg's Network Optimization Library, http://web.archive.org/web/20191209133545/http://www.avglab.com:80/andrew/soft.html, CSA 1.2, built as csa_q. Last accessed on 13/10/2026
code 10. Goldberg, A.V.: HIPR. Andrew Goldberg's Network Optimization Library, http://web.archive.org/web/20191209133545/http://www.avglab.com:80/andrew/soft.html, hi_pr 3.5. Last accessed on 13/10/2026
code 11. Google OR-Tools: Solving an assignment problem: Linear sum assignment solver, https://developers.google.com/optimization/assignment/linear_assignment, OR-Tools 9.15.6755. Last accessed on 10/10/2026
code 12. Kovács, P.: goto_8.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/goto_8.sh, last accessed on 14/10/2026
code 13. Kovács, P.: goto_sr.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/goto_sr.sh, last accessed on 14/10/2026
code 14. Kovács, P.: gridgen_8.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/gridgen_8.sh, last accessed on 14/10/2026
code 15. Kovács, P.: gridgen_deg.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/gridgen_deg.sh, last accessed on 14/10/2026
code 16. Kovács, P.: gridgen_sr.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/gridgen_sr.sh, last accessed on 14/10/2026
code 17. Kovács, P.: grid_long.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/grid_long.sh, last accessed on 14/10/2026
code 18. Kovács, P.: grid_square.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/grid_square.sh, last accessed on 14/10/2026
code 19. Kovács, P.: grid_wide.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/grid_wide.sh, last accessed on 14/10/2026
code 20. Kovács, P.: netgen_8.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/netgen_8.sh, last accessed on 14/10/2026
code 21. Kovács, P.: netgen_deg.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/netgen_deg.sh, last accessed on 14/10/2026
code 22. Kovács, P.: netgen_lo_8.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/netgen_lo_8.sh, last accessed on 14/10/2026

code 23. Kovács, P.: netgen_lo_sr.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/netgen_lo_sr.sh, last accessed on 14/10/2026
code 24. Kovács, P.: netgen_sr.sh. Benchmark Data for the Minimum-Cost Flow Problem, https://lemon.cs.elte.hu/trac/lemon/raw-attachment/wiki/MinCostFlowData/netgen_sr.sh, last accessed on 14/10/2026
code 25. Kovács, P.: Road instances. Benchmark Data for the Minimum-Cost Flow Problem, http://lime.cs.elte.hu/~kpeter/data/mcf/road/, last accessed on 14/10/2026
code 26. Kovács, P.: Vision instances. Benchmark Data for the Minimum-Cost Flow Problem, http://lime.cs.elte.hu/~kpeter/data/mcf/vision/, last accessed on 14/10/2026
code 27. Lammich, P., Sefidgar, S.R.: Formalizing push-relabel algorithms. Arch. Formal Proofs (Jun 2017), https://isa-afp.org/entries/Prpu_Maxflow.html
code 28. The JGraphT Developers: BreadthFirstIterator, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/traverse/BreadthFirstIterator.html, JGraphT 1.5.3. Last accessed on 12/10/2026
code 29. The JGraphT Developers: DepthFirstIterator, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/traverse/DepthFirstIterator.html, JGraphT 1.5.3. Last accessed on 15/10/2026
code 30. The JGraphT Developers: IntVertexDijkstraShortestPath, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/shortestpath/IntVertexDijkstraShortestPath.html, JGraphT 1.5.3. Last accessed on 11/10/2026
code 31. The JGraphT Developers: KruskalMinimumSpanningTree, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/spanning/KruskalMinimumSpanningTree.html, JGraphT 1.5.3. Last accessed on 15/10/2026
code 32. The JGraphT Developers: KuhnMunkresMinimalWeightBipartitePerfectMatching, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/matching/KuhnMunkresMinimalWeightBipartitePerfectMatching.html, JGraphT 1.5.3. Last accessed on 10/10/2026
code 33. The JGraphT Developers: MaximumWeightBipartiteMatching, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/matching/MaximumWeightBipartiteMatching.html, JGraphT 1.5.3. Last accessed on 10/10/2026
code 34. The JGraphT Developers: PushRelabelMFImpl, https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/flow/PushRelabelMFImpl.html, JGraphT 1.5.3. Last accessed on 13/10/2026
code 35. The LEMON Developers: Bfs< GR, TR > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00051.html, LEMON 1.3.1. Last accessed on 12/10/2026
code 36. The LEMON Developers: Capacityscaling< GR, V, C, TR > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00074.html, LEMON 1.3.1. Last accessed on 14/10/2026
code 37. The LEMON Developers: Costscaling< GR, V, C, TR > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00109.html, LEMON 1.3.1. Last accessed on 14/10/2026
code 38. The LEMON Developers: Dfs< GR, TR > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00119.html, LEMON 1.3.1. Last accessed on 15/10/2026

code 39. The LEMON Developers: Dijkstra< GR, LEN, TR > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00133.html, LEMON 1.3.1. Last accessed on 11/10/2026
code 40. The LEMON Developers: Minimum spanning tree algorithms: kruskal(), https://lemon.cs.elte.hu/pub/doc/1.3.1/a00610.html, LEMON 1.3.1. Last accessed on 15/10/2026
code 41. The LEMON Developers: Networksimplex< GR, V, C > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00276.html, LEMON 1.3.1. Last accessed on 14/10/2026
code 42. The LEMON Developers: Preflow< GR, CAP, TR > class template reference, https://lemon.cs.elte.hu/pub/doc/1.3.1/a00311.html, LEMON 1.3.1. Last accessed on 13/10/2026
code 43. The NetworkX Developers: networkx.bfs_predecessors, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.traversal.breadth_first_search.bfs_predecessors.html, NetworkX 3.4.2. Last accessed on 12/10/2026
code 44. The NetworkX Developers: networkx.dfs_predecessors, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.traversal.depth_first_search.dfs_predecessors.html, NetworkX 3.4.2. Last accessed on 15/10/2026
code 45. The NetworkX Developers: networkx.minimum_spanning_edges, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.tree.mst.minimum_spanning_edges.html, NetworkX 3.4.2. Last accessed on 15/10/2026
code 46. The NetworkX Developers: networkx.preflow_push, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.preflow_push.html, NetworkX 3.4.2. Last accessed on 13/10/2026
code 47. The NetworkX Developers: networkx.single_source_dijkstra, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.single_source_dijkstra.html, NetworkX 3.4.2. Last accessed on 11/10/2026
code 48. The Sage Developers: Matroid.intersection, https://doc.sagemath.org/html/en/reference/matroids/sage/matroids/matroid.html#sage.matroids.matroid.Matroid.intersection, passagemath 10.6.48. Last accessed on 11/10/2026
code 49. The Sage Developers: Matroid.intersection_unweighted, https://doc.sagemath.org/html/en/reference/matroids/sage/matroids/matroid.html#sage.matroids.matroid.Matroid.intersection_unweighted, passagemath 10.6.48. Last accessed on 11/10/2026
code 50. The Sage Developers: Partitionmatroid, https://doc.sagemath.org/html/en/reference/matroids/sage/matroids/union_matroid.html#sage.matroids.union_matroid.PartitionMatroid, passagemath 10.6.48. Last accessed on 11/10/2026
code 51. The Sage Developers: Rankmatroid, https://doc.sagemath.org/html/en/reference/matroids/sage/matroids/rank_matroid.html#sage.matroids.rank_matroid.RankMatroid, passagemath 10.6.48. Last accessed on 11/10/2026
code 52. The SciPy Community: scipy.optimize.linear_sum_assignment, https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html, SciPy 1.15.3. Last accessed on 10/10/2026

code 53. The SciPy Community: scipy.sparse.csgraph.min_weight_full_bipartite_matching, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.min_weight_full_bipartite_matching.html, SciPy 1.15.3. Last accessed on 10/10/2026
code 54. The SciPy Community: scipy.sparse.csgraph.breadth_first_order, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.breadth_first_order.html, SciPy 1.15.3. Last accessed on 12/10/2026
code 55. The SciPy Community: scipy.sparse.csgraph.depth_first_order, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.depth_first_order.html, SciPy 1.15.3. Last accessed on 15/10/2026
code 56. The SciPy Community: scipy.sparse.csgraph.dijkstra, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.dijkstra.html, SciPy 1.15.3. Last accessed on 11/10/2026
code 57. The SciPy Community: scipy.sparse.csgraph.maximum_flow, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.maximum_flow.html, SciPy 1.15.3. Last accessed on 13/10/2026
code 58. The SciPy Community: scipy.sparse.csgraph.minimum_spanning_tree, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.minimum_spanning_tree.html, SciPy 1.15.3. Last accessed on 15/10/2026

C AI-Assisted Development

C.1 Style Statistics

To make the stylistic conventions of the library measurable, we use a set of statistics that are computed mechanically from the theory sources. The depth-first and breadth-first search theories (`DFS.thy` and `BFS_2.thy`) serve as the reference: their values define the target range. The other columns of Table 39 give the values for the Hopcroft–Karp development before and after the restyling. Both columns cover five theories. The restyling renamed and reorganised them: the characterisation of augmenting paths, the top loop, a new generic search for disjoint paths, the implementation of a phase, and the executable example. For the layering statistics, the reference columns cover the whole families, i.e. also `DFS_Example`, `BFS_Subprocedures` and `BFS_Example`.

All statistics are computed by two scripts. They do not use Isabelle; they work on the source text, as follows:

1. Comments are removed.
2. The contents of inner-syntax strings and cartouches are masked, so that keywords occurring inside terms are not counted.
3. Each theory is split into its top-level commands.
4. Each *lemma* (a `lemma`, `theorem`, `corollary` or `proposition` command; `lemmas` is not counted) is split into its statement and its proof. The proof begins at the first proof keyword (`by`, `apply`, `proof`, `using`, `unfolding`, `including`, `supply`, `subgoal`).

Unless stated otherwise, "per lemma" means a total over all proofs divided by the number of lemmas. A share of method calls is taken over all method calls in the group.

Algorithm	Abstract	Functional	Imperative
Graph search and shortest paths			
Depth-first search	–	✗	(✓CSR version only)
Cycle detection	–	✗	–
Directed cycle detection	✓	✓	✓
Breadth-first search	–	✗	✗
BFS with distances	–	✗	–
Bellman–Ford	–	✗	–
Dijkstra	–	✓	✓
Greedy algorithms and spanning trees			
Best-in-greedy	–	✗	✓
Kruskal	–	✗	✓
Greedoid greedy	–	✗	–
Jarník–Prim	–	✗	–
Flows			
DFS with backtracking	–	✗	–
Blocking flow	–	✗	–
Dinic	–	✗	–
Push-relabel	–	✓	✓
Successive shortest paths	✗	–	–
Capacity scaling	✗	–	–
Orlin	–	✗	–
Acyclic flows	–	✓	✓
Network simplex	–	✓	✓
Matchings			
Edmonds' blossom	✗	✓	✓
Hopcroft–Karp	–	✓	✓
Hungarian method	–	✗(shortcut ✓)	✓
Primal-dual (Egerváry)	✗	–	–
RANKING	✗	–	–
RANKING, primal-dual	✗	–	–
AdWords	✗	–	–
Matroids			
Matroid intersection	–	✗	✓
Weighted matroid intersection	–	✓	✓
Matroid partitioning	✓	–	–

Table 38. The parts of each algorithm that were developed or verified using AI. ✓means AI contribution, ✗only human contributors, and − signifies that this variant of the algorithm is not present. Readers can obtain a more fine-grained understanding of the extent of AI contribution by reading through the theory scripts, which have labels indicating human/human-AI/AI contributions.

Statistic	DFS	BFS	Before	After
Size				
Proof lines	323	1041	1342	1055
Definition and specification lines	355	552	472	917
Lemmas	81	103	108	182
of which loop-template lemmas	65	83	3	92
Proof form (share of lemmas)				
Single-step proofs	76.5%	63.1%	38.0%	69.2%
Structured (`proof` ... `qed`)	21.0%	27.2%	61.1%	27.5%
With `apply`	2.5%	9.7%	0.0%	3.3%
Proof length (non-blank lines)				
Median	1	2	8	2
Mean	3.7	9.9	12.0	5.6
90th percentile	8	31	30	15
Proof lines per statement line	1.50	3.15	4.43	2.02
Steps per structured proof	2.4	8.4	8.7	4.1
Handling of facts (per lemma)				
Labelled `have`/`show`/`obtain`	0.01	0.21	2.29	0.12
`[OF ...]`	0.26	0.61	2.10	0.74
`using`	0.44	1.66	3.10	0.79
Facts quoted as cartouches	0.07	1.64	0.02	0.19
Named assumptions	0.00	0.00	0.51	0.00
Proof methods (share of all method calls)				
`auto`	61.2%	50.7%	10.7%	40.2%
`force` + `fastforce`	9.3%	12.4%	0.4%	1.9%
`simp`	2.2%	4.3%	29.0%	11.6%
`rule`	10.1%	7.5%	23.8%	13.8%
`metis` + `smt`	0.0%	4.1%	0.0%	0.0%
Closing calls inside structured proofs				
`by (rule ...)`	2.6%	1.0%	25.9%	14.2%
Bare `by simp`/`auto`/`blast`/...	20.5%	21.6%	46.4%	34.7%
Method modifiers (per lemma)				
`elim!:`	0.38	0.50	0.00	0.14
`intro!:`	0.46	0.49	0.00	0.16
Statements				
Plain form $[\![A;\ B]\!] \Longrightarrow C$	79.0%	77.7%	57.4%	79.1%
$[\![$ per statement	0.69	0.79	0.00	0.34
Lemmas with an attribute	80.2%	67.0%	6.5%	45.6%
Layout				
`by` opening its own line	100%	95.0%	16.8%	100%
`using` opening its own line	100%	99.4%	34.9%	100%
`proof-`/`proof(` (no space)	4/15	15/43	0/0	30/28
`proof -`/`proof (` (space)	0/0	0/0	41/52	0/0
Line length, 95th percentile	119	117	99	104
Definitions and program state				
State `records`	1	1	0	2
Loops as `function (domintros)` over one record	1	1	0	2
Loops written as `fun`	0	0	1	0
Executable twins (`partial_function`)	1	1	1	2
`named_theorems` declared	9	9	0	14
Untyped `definitions`	22/22	23/23	24/26	50/50
Invariants				
`invar_*` definitions	5	13	0	8
Complete families (elim., intro., per branch, loop)	5	7	0	8
Uses of `[invar_holds_intros]`	26	33	0	33
Naming				
Mean lemma-name length (characters)	18.2	22.1	14.2	22.8
Mean lemma-name parts	4.0	4.5	3.0	4.0
Constants with the prefix `hk_`	–	–	11	0
Global abbreviations named with ≤ 3 characters	0	0	9	0
Layering				
Interpretations in algorithm theories (global/plain)	0/0	0/0	2/3	0/0
Interpretations in the sub-procedure theory (global/plain)	–	0/0	–	4/3[a]
Interpretations in the example theory (global/plain)	1/0	2/0	1/1	1/1
`sublocale` commands	0	0	0	0
Algorithm theories importing concrete data structures	0	0	1	0
Locales with qualified data-structure parents	1	1	0	2
Sub-procedure contracts in the definitions locale	–	yes	no	yes

[a] All in `Hopcroft_Karp_Subprocedures`, a recorded deviation: the phase is implemented there at the concrete operations, since the alternating BFS and the incremental DFS of the Micali–Vazirani development are used through their concrete instances.

Table 39. Style statistics of the depth- and breadth-first search theories (the ground truth) and of the Hopcroft–Karp theories before and after the restyling.

Size. Proofs and specifications are counted separately, in non-blank lines. Comments, top-level `text` blocks and section headings are not counted in either.

Proof lines The lines of all proofs: the proofs of lemmas, the proofs of `interpretation`, `global_interpretation` and `sublocale` commands, and the proofs of `function` definitions.

Definition and specification lines The lines of all lemma statements, all definitions (`definition`, `abbreviation`, `fun`, `function`, `partial_function`, `primrec`, `record`, `datatype`), the headers of locales and contexts (parents, `fixes`, `assumes`), and the instantiation part of interpretations.

Lemmas The number of lemma commands as defined above. It is the denominator of all per-lemma figures.

Loop-template lemmas The lemmas that the loop template of the reference theories prescribes, recognised by their names. They are:
- branch conditions;
- the four standard lemmas `_cases`, `_simps`, `_induct` and `_domintros`;
- the elimination, introduction and preservation lemmas of the invariants;
- termination;
- the initial state, the return branches, the final lemmas and the executable twin.

They are short and uniform. In the reference theories they make up about four fifths of all lemmas.

Proof form. These statistics classify each proof by the commands it uses. They indicate how much of the reasoning is delegated to a single call of an automatic method.

Single-step proofs The share of lemmas whose proof contains no `proof` and no `apply`. Such a proof consists of one `by` call, possibly preceded by `using` or `unfolding`, and possibly with two methods (`by` m_1 m_2).

Structured The share of lemmas whose proof contains at least one `proof`, i.e. an Isar proof with intermediate steps.

With `apply` The share of lemmas whose proof is a sequence of `apply` steps, with no `proof`. A structured proof that uses `apply` inside it, as the loop lemmas of DFS do, is counted as structured.

Proof length. The length of a proof is its number of non-blank lines, from its first proof keyword to the end of the command.

Median, mean and 90th percentile The distribution of proof lengths over all lemmas. A median of one or two lines means that most lemmas are proved by one call.

Proof lines per statement line The total number of proof lines divided by the total number of statement lines. It relates the effort spent on proofs to the size of what is proved.

Steps per structured proof The mean number of `have`, `show`, `hence`, `thus` and `obtain` commands in a structured proof. It measures how finely a structured proof is subdivided.

Handling of facts. These statistics describe how intermediate and assumed facts are named and passed on. Explicit names and explicit forward application point to proofs that spell out book-keeping which the automatic methods could find themselves.

Labelled `have`/`show`/`obtain` The number of intermediate facts given a name (`have h: ...`, also with `show`, `obtain`, `hence` and `thus`), per lemma.

`[OF ...]` The number of explicit instantiations of a theorem with given facts, per lemma.

`using` The number of `using` commands, per lemma. Each one inserts named facts into a goal.

Facts quoted as cartouches The number of cartouches in proofs that do not open a `text` block or a comment, per lemma. These are mostly references to earlier facts by their statement. The count also includes cartouches used as method arguments, so it is approximate.

Named assumptions The number of assumptions given a name in the statement (`assumes h: ...` or `and h: ...`), per lemma. The reference theories refer to assumptions by position (`assms(2)`) or by statement instead.

Proof methods. For each `by`, `apply` and `proof`, the name of the first method that follows is recorded. Each row gives the share of one method among all recorded calls.

`auto`, `force` and `fastforce` The automatic methods that combine simplification with classical reasoning. They are the usual closing methods in the reference theories.

`simp` Simplification alone.

`rule` The application of one named rule. A high share indicates step-by-step forward reasoning.

`metis` and `smt` Calls to proof reconstruction from external provers. They are excluded by the rules of this project, although the BFS theory still contains some.

Closing calls inside structured proofs. These statistics are computed over the `by` calls inside structured proofs only.

`by (rule ...)` The share of calls that apply a single rule.

Bare `by simp`/`auto`/`blast`/`...` The share of calls to an automatic method without any arguments (`simp`, `auto`, `blast`, `force`, `fastforce`, `linarith`, `presburger`, `simp_all`, `arith`, `fast`). A high share indicates many small steps, each of which is trivial.

Method modifiers. The number of occurrences of a modifier in proofs, per lemma.

`elim!:` and `intro!:` Rules declared as safe elimination and introduction rules for a single call. In the reference theories these modifiers mostly carry the named theorem collections (`call_cond_elims`, `invar_props_elims`, `invar_props_intros`). Their frequency therefore shows how far proofs are driven by these collections rather than by individual facts.

Statements.

Plain form The share of lemmas stated as a single proposition, without `assumes`/`shows`. Premises are then written as $[\![A; B]\!] \Longrightarrow C$.

[\[per statement The number of premise brackets in statements, per lemma. It shows how often several premises are stated in this compact form.

Lemmas with an attribute The share of lemmas declared with an attribute, such as `[simp]` or `[invar_holds_intros]`. Attributes make a lemma available to later automatic calls, so this share measures how far the theory prepares its own automation.

Layout.

`by` (`using`) opening its own line The share of occurrences of `by` (`using`) inside proofs that begin a new line. The keyword that opens a proof is counted as beginning a line, so the figure mainly reflects the steps inside structured proofs. The reference theories place `using` and `by` on their own indented lines; they do not append them to the stated fact.

`proof-`/`proof(` versus `proof -`/`proof (` The number of occurrences of each spelling. The reference theories write these keywords without a space.

Line length, 95th percentile The length, in characters, that 95% of the non-blank lines do not exceed.

Definitions and program state.

State `records` The number of `record` commands. In the reference theories, a record holds the complete state of an algorithm.

Loops as `function (domintros)` over one record The number of loops defined by the function package with domain introduction rules generated, whose single argument is a state record. The number of arguments is counted by the script; that the argument is a record was checked in the source.

Loops written as `fun` Recursive algorithm definitions by `fun`, which the reference theories do not use for loops.

Executable twins The number of `partial_function (tailrec)` definitions. Each is an executable copy of a loop, related to it by a lemma.

`named_theorems` declared The number of named theorem collections declared, such as `invar_holds_intros` and `call_cond_elims`. They are the means by which proofs refer to whole families of lemmas.

Untyped `definitions` The number of definitions given as an equation without a type signature, out of all definitions.

Invariants.

`invar_*` definitions The number of definitions whose name begins with `invar_`, i.e. the number of separately stated invariants of the algorithm states.

Complete families The number of these invariants that have the full family of lemmas of the reference theories:

- an elimination form `invar_`x`_props`;
- an introduction form `invar_`x`_intro`;
- at least one preservation lemma for a branch of the loop (`invar_`x`_holds_`i);
- the preservation lemma for the whole loop (`invar_`x`_holds`).

Uses of `[invar_holds_intros]` The number of lemmas registered in the collection of preservation lemmas.

Naming.

Mean lemma-name length and parts The mean number of characters, and the mean number of parts separated by underscores, over all named lemmas. Short names with few parts point to names that do not say what the lemma is about.

Constants with the prefix `hk_` The number of defined constants whose names begin with `hk_`. The library writes the name of an algorithm out in full.

Global abbreviations named with ≤ 3 characters The number of `abbreviation`s with names of at most three characters, such as `es` or `CG`.

Layering. These statistics concern the arrangement of algorithms, sub-procedures and implementations in locales and theories. Each theory is in one of three classes:

- the *example theory* (`*_Example`), the only theory in which the reference algorithms are instantiated with concrete data structures;
- the *sub-procedure theory* (`*_Subprocedures`), which implements the sub-procedures of an algorithm;
- an *algorithm theory*, which is any other theory.

Interpretations in algorithm theories, in the sub-procedure theory and in the example theory The number of `global_interpretation` and `interpretation` commands in theories of each class, given as global/plain. In the reference families, interpretations occur only in the example theories.

`sublocale` commands The number of `sublocale` commands. The reference theories do not use them to relate algorithms.

Algorithm theories importing concrete data structures The number of algorithm theories that import an implementation by red-black trees, or a theory that builds an executable instance on one.

Locales with qualified data-structure parents The number of locale definitions that import a data-structure specification as a qualified parent (`Graph: Pair_Graph_Specs`, `set_ops: Set2`). Locales that only inherit such a parent are not counted.

Sub-procedure contracts in the definitions locale Whether the specifications of the sub-procedures, which are parameters of a loop, are assumptions of the locale that defines the loop (as in BFS), or of a separate locale used only for proofs. This was determined by reading the source; "–" means that there are no sub-procedures.

Limitations. The statistics are computed from the source text, not from the logical content that Isabelle constructs, so some of them are approximations:

- the count of cartouches used as fact references;
- the classification of a `by` at the start of a proof as beginning a line;
- the recording of only the first method of a call with two methods.

They measure conformity of form; they do not measure the difficulty of the mathematics. A theory without loops, such as the characterisation of augmenting paths in `Alt_BFS_Bipartite`, naturally has longer structured proofs than a loop theory, even when it is written in the same style.

D Formal Listings

This appendix contains the statements of the definitions, locales, and theorems that the body of the paper cites, without their proofs. The index theory file in the supporting material identifies, for every listing, the repository, the theory, and the lines where it is.

Listing 1. `theorem initial_state_orlins_dom_and_results` (`Orlins.thy`)

```
theorem initial_state_orlins_dom_and_results:
  assumes "state' = orlins (send_flow initial)"
    shows "orlins_dom (send_flow initial)"
          "return state' = success ⟹ is_Opt b (a_current_flow state')"
          "return state' = infeasible ⟹ ∄ f. f is b flow"
          "return state' = notyetterm ⟹ False"
          "send_flow_dom initial"
          "orlins_impl (send_flow_impl initial) = state'"
```

Listing 2. `lemma find_max_matching_works` (`Blossom_Algo_Main_Loop.thy`)

```
lemma find_max_matching_works:
  shows "(find_max_matching {}) ⊆ G"
    "matching (find_max_matching {})"
    "finite (find_max_matching {})"
    "∀M. matching M ∧ M ⊆ G ∧ finite M ⟶ card M ≤ card (find_max_matching {})"
```

Listing 3. `lemma DFS_correct_1` (`DFS.thy`, FDS book version)

```
lemma DFS_correct_1:
  assumes "return (DFS initial_state) = NotReachable"
  shows   "∄p. distinct p ∧ vwalk_bet (Graph.digraph_abs G) s p t"
```

Listing 4. `lemma BFS_correct_3` (`BFS_2.thy`, FDS book version)

```
lemma BFS_correct_3:
  "⟦u ∈ t_set srcs; Vwalk.vwalk_bet (Graph.digraph_abs (parents (BFS initial_state))) u p v⟧
        ⟹ length p - 1 = distance_set (Graph.digraph_abs G) (t_set srcs) v"
```

Listing 5. `inductive vwalk` (`Vwalk.thy`)

```
inductive vwalk where
  vwalk0: "vwalk []" |
  vwalk1: "v ∈ dVs G ⟹ vwalk [v]" |
  vwalk2: "(v,v') ∈ G ⟹ vwalk (v'#vs) ⟹ vwalk (v#v'#vs)"
```

Listing 6. `definition awalk` (`Awalk_Defs.thy`)

```
definition awalk :: "'a dgraph ⇒ 'a ⇒ ('a × 'a) list ⇒ 'a ⇒ bool" where
  "awalk E u p v = ( u ∈ dVs E ∧ set p ⊆ E ∧ cas u p v)"
```

Listing 7. `definition walk_betw` (`Paths.thy`)

```
definition "walk_betw G u p v = (p ≠ [] ∧ path G p ∧ hd p = u ∧ last p = v)"
```

Listing 8. `lemma walk_betw_iff_vwalk_bet` (`Pair_Graph_Berge_Adaptor.thy`)

```
lemma walk_betw_iff_vwalk_bet:
  "walk_betw G u p v ⟷ vwalk_bet D u p v"
```

Listing 9. `definition multigraph_path` (`Multigraph.thy`)

```
definition "multigraph_path  (es::('edge list)) ⟷ (es = [] ∨ (es ≠ [] ∧
                      awalk UNIV (fst (hd es)) (map make_pair es) (snd (last es))))"
```

Listing 10. `theorem component_has_path_distinct` (`Connected_Components.thy`)

```
theorem component_has_path_distinct:
  assumes "finite G" and "dblton_graph G" and
    "C ∈ connected_components G" and
    "⋀v. v ∈ Vs G ⟹ degree G v ≤ 2"
  shows "∃p. path G p ∧ C = (set p) ∧ distinct p"
```

Listing 11. `lemma in_insert_connected_componentE` (`Connected_Components.thy`)

```
lemma in_insert_connected_componentE[case_names both_nin one_in two_in]:
  assumes "C ∈ connected_components (insert {u,v} G)"
    "⟦u ∉ Vs G; v ∉ Vs G;
     C ∈ insert {u,v} (connected_components G)⟧
     ⟹ P"
    "⋀u' v'.
     ⟦u' ∈ {u,v}; v' ∈ {u, v}; u' ∈ Vs G; v' ∉ Vs G;
     C ∈ insert (insert v' (connected_component G u')) (connected_components G - {
    connected_component G u'})⟧
     ⟹ P"
    "⟦u ∈ Vs G; v ∈ Vs G; connected_component G u ≠ connected_component G v;
     C ∈ insert (connected_component G u ∪ connected_component G v)
                    ((connected_components G) - {connected_component G u,
    connected_component G v})⟧
     ⟹ P"
    "C ∈ (connected_components G) ⟹ P"
  shows "P"
```

Listing 12. `lemma walk_betw_insertE` (`Paths.thy`)

```
lemma walk_betw_insertE[case_names nil sing1 sing2 in_e in_E]:
   "⟦walk_betw (insert e G) v1 p v2;
     (⟦v1∈Vs (insert e G); v1 = v2; p = []⟧ ⟹ P);
     (⋀v. p = [v] ⟹ v = v1 ⟹ v = v2 ⟹ v ∈ e ⟹ P);
     (⋀v. p = [v] ⟹ v = v1 ⟹ v = v2 ⟹ v ∈ Vs G ⟹ P);
     (⋀p' v3. ⟦walk_betw {e} v1 [v1,v3] v3; walk_betw (insert e G) v3 (v3 # p') v2; p = v1 #
     v3 # p'⟧ ⟹ P);
     (⋀p' v3. ⟦walk_betw G v1 [v1,v3] v3; walk_betw (insert e G) v3 (v3 # p') v2; p = v1 #
    v3 # p'⟧ ⟹ P)⟧
    ⟹ P"
```

Listing 13. `lemma reachable_insertE` (`Paths.thy`)

```
lemma reachable_insertE[case_names nil sing1 sing2 in_e in_E]:
   "⟦reachable (insert e G) v1 v2;
     (⟦v1 ∈ e; v1 = v2⟧ ⟹ P);
     (⟦v1 ∈ Vs G; v1 = v2⟧ ⟹ P);
     (⋀p' v3. ⟦reachable {e} v1 v3; reachable (insert e G) v3 v2⟧ ⟹ P);
     (⋀p' v3. ⟦reachable G v1 v3; reachable (insert e G) v3 v2⟧ ⟹ P)⟧
    ⟹ P"
```

Listing 14. `lemma vwalk_bet_insertE` (`Vwalk.thy`)

```
lemma vwalk_bet_insertE[case_names nil sing1 sing2 in_e in_E]:
   "⟦vwalk_bet (insert e E) v1 p v2;
     (⟦v1∈dVs (insert e E); v1 = v2; p = []⟧ ⟹ P);
     (⋀u v. p = [u] ⟹ e = (u,v) ⟹ P);
     (⋀u v. p = [v] ⟹ e = (u,v) ⟹ P);
     (⋀v. p = [v] ⟹ v = v1 ⟹ v = v2 ⟹ v ∈ dVs E ⟹ P);
     (⋀p' v3. ⟦vwalk_bet {e} v1 [v1,v3] v3; vwalk_bet (insert e E) v3 (v3 # p') v2; p = v1 #
     v3 # p'⟧ ⟹ P);
     (⋀p' v3. ⟦vwalk_bet E v1 [v1,v3] v3; vwalk_bet (insert e E) v3 (v3 # p') v2; p = v1 #
    v3 # p'⟧ ⟹ P)⟧
    ⟹ P"
```

Listing 15. `lemma walk_betw_unionE` (`Paths.thy`)

```
lemma walk_betw_unionE[case_names nil sing1 sing2 in_e in_E]:
   "⟦walk_betw (G1 ∪ G2) v1 p v2;
     (⟦v1∈Vs (G1 ∪ G2); v1 = v2; p = []⟧ ⟹ P);
     (⋀v. p = [v] ⟹ v = v1 ⟹ v = v2 ⟹ v ∈ Vs G1 ⟹ P);
     (⋀v. p = [v] ⟹ v = v1 ⟹ v = v2 ⟹ v ∈ Vs G2 ⟹ P);
     (⋀p' v3. ⟦walk_betw G1 v1 [v1,v3] v3; walk_betw (G1 ∪ G2) v3 (v3 # p') v2; p = v1 # v3
    # p'⟧ ⟹ P);
     (⋀p' v3. ⟦walk_betw G2 v1 [v1,v3] v3; walk_betw (G1 ∪ G2) v3 (v3 # p') v2; p = v1 # v3
    # p'⟧ ⟹ P)⟧
    ⟹ P"
```

Listing 16. `lemma reachability_split` (`Paths.thy`)

```
lemma reachability_split:
  "⟦reachable (insert {w, x} G) u v; w ∈ Vs G; x ∈ Vs G⟧ ⟹
        (reachable G u v) ∨
         (reachable G u w ∧ reachable G x v) ∨
         (reachable G u x ∧ reachable G w v)"
```

Listing 17. `lemma induct_walk_betw` (`Paths.thy`)

```
lemma induct_walk_betw[case_names path1 path2, consumes 1, induct set: walk_betw]:
  assumes "walk_betw G a p b"
  assumes Path1: "⋀v. v ∈ Vs G ⟹ P v [v] v"
  assumes Path2: "⋀v v' vs b. {v, v'} ∈ G ⟹ walk_betw G v' (v' # vs) b ⟹ P v' (v' # vs)
     b ⟹ P v (v # v' # vs) b"
  shows "P a p b"
```

Listing 18. `lemma induct_vwalk_bet` (`Vwalk.thy`)

```
lemma induct_vwalk_bet[case_names path1 path2, consumes 1, induct set: vwalk_bet]:
  assumes "vwalk_bet E a p b"
  assumes Path1: "⋀v. v ∈ dVs E ⟹ P E [v] v v"
  assumes Path2: "⋀v v' vs b. (v, v') ∈ E ⟹ vwalk_bet E v' (v' # vs) b ⟹ P E (v' # vs)
    v' b ⟹ P E (v # v' # vs) v b"
  shows "P E p a b"
```

Listing 19. `lemma vwalk_alt_induct` (`Vwalk.thy`)

```
lemma vwalk_alt_induct[consumes 1, case_names Single Snoc]:
  assumes
    "vwalk E p"  "P []" "(⋀x. P [x])"
    "⋀y x xs. (y, x) ∈ E ⟹ vwalk E (xs @ [y]) ⟹ P (xs @ [y]) ⟹ P (xs @ [y, x])"
  shows "P p"
```

Listing 20. `lemma awalk_induct` (`Awalk.thy`)

```
lemma awalk_induct:
  assumes "es ≠ []"
          "awalk E u es v "
          "⋀ E u v e. e ∈ E ⟹ u = fst e ⟹ v = snd e ⟹ P E u [e] v"
          "⋀ E u v w e d es. e ∈ E ⟹ u = fst e ⟹ v = snd e⟹ v = fst (hd (d#es))⟹
                            P E v (d#es) w  ⟹ P E u (e#d#es) w"
  shows   " P E u es v"
```

Listing 21. `lemma multigraph_path_induct` (`Multigraph.thy`)

```
lemma multigraph_path_induct:
assumes "multigraph_path x2"
        "(⋀g. P  [])"
        "(⋀g e.  P  [e])"
        "(⋀g e es. snd e = fst (hd es) ⟹ multigraph_path  es ⟹ P  es ⟹ P  (e # es))"
      shows   "P  x2"
```

Listing 22. `lemma reachable_induct` (`Pair_Graph.thy`)

```
lemma reachable_induct[consumes 1, case_names base step]:
  assumes major: "u →*E v"
    and cases: "⟦u ∈ dVs E⟧ ⟹ P u"
      "⋀x y. ⟦u →*E x; (x,y) ∈ E; P x⟧ ⟹ P y"
  shows "P v"
```

Listing 23. `lemma converse_reachable_induct` (`Pair_Graph.thy`)

```
lemma converse_reachable_induct[consumes 1, case_names base step, induct pred: reachable]:
  assumes major: "u →*E v"
    and cases: "v ∈ dVs E ⟹ P v"
      "⋀x y. ⟦(x,y) ∈ E; y →*E v; P y⟧ ⟹ P x"
    shows "P u"
```

Listing 24. `lemma vwalk_bet_to_distinct_induct` (`Vwalk.thy`)

```
lemma vwalk_bet_to_distinct_induct [consumes 1, case_names path decomp]:
  assumes "vwalk_bet E u p v"
  assumes distinct: "⋀p. ⟦ vwalk_bet E u p v; distinct p ⟧ ⟹ P p"
  assumes
    decomp: "⋀p q r s. ⟦ vwalk_bet E u p v; ¬ distinct p;
      closed_vwalk_bet_decomp E p = (q, r, s); P (q @ tl s) ⟧ ⟹ P p"
  shows "P p"
```

Listing 25. `theorem Berge` (`Berge.thy`)

```
theorem Berge:
  assumes matching: "finite M" "matching M" "M ⊆ G" and
    graph: "dblton_graph G" "finite (Vs G)"
  shows "(∃p. matching_augmenting_path M p ∧ path G p ∧ distinct p) =
            (∃M' ⊆ G. matching M' ∧ card M < card M')"
```

Listing 26. `lemma berge_formula` (`Berge_Formula.thy`)

```
lemma berge_formula:
  assumes "graph_matching G M"
  assumes "X ⊆ Vs G"
  assumes "∀Y ⊆ Vs G. int (card (odd_comps_in_diff G X)) - int (card X) ≥
          int (card (odd_comps_in_diff G Y)) - int (card Y)"
  assumes "∀M'. graph_matching G M' ⟶ card M ≥ card M'"
  shows "2 * (card M) + card (odd_comps_in_diff G X) - card X = card (Vs G)"
```

Listing 27. `theorem edmonds_gallai_decomposition_all` (`Edmonds_Gallai.thy`, branch `weighted_blossom` on a development fork)

```
theorem edmonds_gallai_decomposition_all:
  assumes "graph_invar G" "max_card_matching G M"
  shows
    "⟦X ∈ 𝒪𝒞 G (𝒜 G); x ∈ X⟧ ⟹ ∃M. graph_matching ( G ⟦X⟧) M ∧ Vs M = X - {x}"
    "X ∈ 𝒪𝒞 G (𝒜 G) ⟹ ∃x. x ∈ X ∧ Vs ( M ⟦X⟧) = X - {x}"
    "perfect_matching (G \ ⋃ (𝒪𝒞 G (𝒜 G)) ∪ 𝒜 G) (M \ ⋃ (𝒪𝒞 G (𝒜 G)) ∪ 𝒜 G)"
    "⟦X ⊆ 𝒜 G; X ≠ {}⟧ ⟹ card X < card {D |D. D ∈ 𝒪𝒞 G (𝒜 G) ∧ X ⟷⇘G⇙ D}"
    "∃m. inj_on m (𝒜 G) ∧ m ` 𝒜 G ⊆ ⋃ (𝒪𝒞 G (𝒜 G)) ∧
        (∀x∈𝒜 G. {x, m x} ∈ M) ∧
        (∀x y. x ∈ 𝒜 G ∧ y ∈ 𝒜 G ∧ x ≠ y ⟶
        (∃D1 D2. D1 ∈ 𝒪𝒞 G (𝒜 G) ∧ D2 ∈ 𝒪𝒞 G (𝒜 G) ∧ D1 ≠ D2 ∧ m x ∈ D1 ∧ m y ∈ D2))"
    "2 * ν G + (int (card (𝒪𝒞 G (𝒜 G))) - int (card (𝒜 G))) = card (Vs G)"
    "⟦e ∈ M ; ⟦e ⊆ Vs G - ⋃ (𝒪𝒞 G (𝒜 G)) - 𝒜 G⟧ ⟹ P;
      ⋀x y D. ⟦e = {x, y}; x ∈ 𝒜 G; y ∈ D; D ∈ 𝒪𝒞 G (𝒜 G); x ≠ y⟧ ⟹ P;
      ⋀D. ⟦D ∈ 𝒪𝒞 G (𝒜 G); e ⊆ D⟧ ⟹ P⟧ ⟹ P"
    "⋃ (𝒪𝒞 G (𝒜 G)) = ℐ G"
    "∃m. inj_on m (Vs G - Vs M) ∧ m ` (Vs G - Vs M) ⊆ 𝒪𝒞 G (𝒜 G)"
    "⟦e ∈ G; e ⊆ Vs G - ⋃ (𝒪𝒞 G (𝒜 G)) - 𝒜 G ⟹ P; e ⊆ 𝒜 G ⟹ P;
      ⋀x y D. ⟦e = {x, y}; x ∈ 𝒜 G; y ∈ D; D ∈ 𝒪𝒞 G (𝒜 G); x ≠ y⟧ ⟹ P;
      ⋀x y. ⟦e = {x, y}; x ∈ 𝒜 G; y ∈ Vs G - ⋃ (𝒪𝒞 G (𝒜 G)) - 𝒜 G; x ≠ y⟧ ⟹ P;
      ⋀D. ⟦D ∈ 𝒪𝒞 G (𝒜 G); e ⊆ D⟧ ⟹ P⟧ ⟹ P"
```

Listing 28. `locale wf_digraph` (`Digraph.thy`, AFP)

```
locale wf_digraph = pre_digraph +
  assumes tail_in_verts[simp]: "e ∈ arcs G ⟹ tail G e ∈ verts G"
  assumes head_in_verts[simp]: "e ∈ arcs G ⟹ head G e ∈ verts G"
```

Listing 29. `locale multigraph` (`Multigraph.thy`)

```
locale multigraph = multigraph_spec where ℰ = "ℰ::'edge set" for ℰ +
assumes fst_create_edge:"⋀ x y. fst (create_edge x y) = x"
assumes snd_create_edge:"⋀ x y. snd (create_edge x y) = y"
assumes finite_E: "finite ℰ"
assumes E_not_empty: "ℰ ≠ {}"
```

Listing 30. `definition dVs` (`Pair_Graph.thy`)

```
definition "dVs G = ⋃ {{v1,v2} | v1 v2. (v1, v2) ∈ G}"
```

Listing 31. `locale graph_abs` (`Undirected_Set_Graphs.thy`)

```
locale graph_abs =
  graph_defs +
  assumes graph: "graph_invar G"
```

Listing 32. `locale outgoing_edge_iterator` (`Acyclic_Flow.thy`, branch `more_algorithms`)

```
locale outgoing_edge_iterator =
  multigraph_spec where fst = fst +
  outg: indexed_iterable_set where
      idx_invar     = out_invar and
      idx_abstract  = out_abstract and
      idx_current   = out_current and
      idx_has       = out_has and
      idx_iterated  = out_iterated and
      idx_remaining = out_remaining and
      idx_move      = out_move and
      idx_reset     = out_reset and
      K = 𝒱
  for fst :: "'e ⇒ 'v"
  and out_invar     :: "'g ⇒ bool"
  and out_abstract  :: "'g ⇒ 'v ⇒ 'e set"
  and out_current   :: "'g ⇒ 'v ⇒ 'e"
  and out_has       :: "'g ⇒ 'v ⇒ bool"
  and out_iterated  :: "'g ⇒ 'v ⇒ 'e set"
  and out_remaining :: "'g ⇒ 'v ⇒ 'e set"
  and out_move      :: "'g ⇒ 'v ⇒ 'g"
  and out_reset     :: "'g ⇒ 'v ⇒ 'g"
```

Listing 33. `locale Pair_Graph_Specs` (`Pair_Graph_Specs.thy`)

```
locale Pair_Graph_Specs =
 adjmap: Map
 where update = update and invar = adjmap_inv +

 vset: Set_Choose
 where empty = vset_empty and delete = vset_delete and invar = vset_inv

 for update :: "'v ⇒ 'vset ⇒ 'adjmap ⇒ 'adjmap" and adjmap_inv :: "'adjmap ⇒ bool"  and

     vset_empty :: "'vset" and vset_delete :: "'v ⇒ 'vset ⇒ 'vset" and
     vset_inv
```

Listing 34. `locale Pair_Graph_U_Specs` (`Pair_Graph_U_Specs.thy`)

```
locale Pair_Graph_U_Specs =
pair_graph_specs: Pair_Graph_Specs
  where lookup = lookup for lookup :: "'adjmap ⇒ ('v::linorder) ⇒ 'vset option"
```

Listing 35. `locale csr_graph` (`CSR_Graph.thy`, branch `more_algorithms`)

```
locale csr_graph =
  fixes n :: nat
    and K :: "nat set"
    and B :: "nat list"
    and E :: "'e::heap list"
  assumes K_bound: "K ⊆ {..<n}"
    and B_length: "length B = Suc n"
    and B_mono: "⋀v. v < n ⟹ B ! v ≤ B ! Suc v"
    and B_bound: "⋀v. v < n ⟹ B ! Suc v ≤ length E"
    and E_distinct: "distinct E"
```

Listing 36. `interpretation G: Pair_Graph_Specs` (`Pair_Graph_RBT.thy`)

```
interpretation G: Pair_Graph_Specs RBT_Set.empty RBT_Map.delete lookup insert_rbt isin Tree2.
    set_tree sel
     update M.invar Leaf RBT.delete vset_inv
```

Listing 37. `interpretation G: Pair_Graph_Specs` (`Pair_Graph_AVL.thy`)

```
interpretation G: Pair_Graph_Specs
  where empty=AVL_Set_Code.empty and update=update and delete = AVL_Map.delete and
        lookup =lookup and adjmap_inv = M.invar  and vset_empty = AVL_Set_Code.empty and
        insert = AVL_Set_Code.insert and vset_delete = AVL_Set_Code.delete and isin = isin
    and
        t_set = "set o inorder" and vset_inv = S.invar and sel = avl_sel
```

Listing 38. `locale Pair_Graph_Imperative` (`Pair_Graph_Imperative.thy`)

```
locale Pair_Graph_Imperative =
  imp_map_empty where empty=mempty_impl +
  imp_map_lookup where lookup=mlookup_impl +
  imp_map_update where update=mupdate_impl
  for mempty_impl mlookup_impl mupdate_impl
  +
  constrains is_map :: "('v ⇀ 'v list) ⇒ 'gimpl ⇒ assn"
```

Listing 39. `lemma vwalk_iff` (`Pair_Graph_Library_Adaptor.thy`)

```
lemma vwalk_iff:
  assumes "u ∈ dVs D"
  shows "vwalk (u # xs) G ⟷ Vwalk.vwalk D (u # xs)"
```

Listing 40. `definition digraph_abs` (`Pair_Graph_Specs.thy`)

```
definition digraph_abs ("[_]_g") where "digraph_abs G = {(u,v). v ∈_G (𝒩_G G u)}"
```

Listing 41. `interpretation iam_graph: Pair_Graph_Imperative` (`DFS_Imperative.thy`)

```
interpretation iam_graph: Pair_Graph_Imperative
  is_iam iam_new iam_lookup iam_update
```

Listing 42. `interpretation hm_graph: Pair_Graph_Imperative` (`DFS_Imperative.thy`)

```
interpretation hm_graph: Pair_Graph_Imperative
  is_hashmap hm_new hm_lookup hm_update
```

Listing 43. `locale Adj_Map_Specs2` (`Orlins_Preparation.thy`)

```
locale Adj_Map_Specs2 =
 adjmap: Map'  where update = update and invar = adjmap_inv +
 vset: Set_Choose where empty = vset_empty and delete = vset_delete and insert = vset_insert
     and
                        invar = vset_inv and isin = isin
for update :: "'a ⇒ 'vset ⇒ 'adjmap ⇒ 'adjmap" and
    adjmap_inv :: "'adjmap ⇒ bool"  and
    vset_empty :: "'vset"  ("∅_N") and
    vset_delete :: "'a ⇒ 'vset ⇒ 'vset" and
    vset_insert and
    vset_inv and
    isin
```

Listing 44. `locale Multi_Graph_Specs` (`BFS_Multi.thy`, branch `bfs_refinement`)

```
locale Multi_Graph_Specs =
 adjmap: Map
 where update = update and invar = adjmap_inv +

 conset: Set_Choose
```

```
 where empty = conset_empty and delete = conset_delete and invar = conset_inv

 for update :: "'v ⇒ 'conset ⇒ 'adjmap ⇒ 'adjmap" and adjmap_inv :: "'adjmap ⇒ bool"
    and

     conset_empty :: "'conset" and conset_delete :: "'con ⇒ 'conset ⇒ 'conset" and
     conset_inv
  +
   fixes fst::"'con ⇒ 'v"
    and snd::"'con ⇒ 'v"
```

Listing 45. `locale indexed_iterable_set` (`Iterable_Set_Specs.thy`, branch `more_algorithms`)

```
locale indexed_iterable_set =
  fixes idx_invar     :: "'coll ⇒ bool"
    and idx_abstract  :: "'coll ⇒ 'i ⇒ 'a set"
    and idx_current   :: "'coll ⇒ 'i ⇒ 'a"
    and idx_has       :: "'coll ⇒ 'i ⇒ bool"
    and idx_iterated  :: "'coll ⇒ 'i ⇒ 'a set"
    and idx_remaining :: "'coll ⇒ 'i ⇒ 'a set"
    and idx_move      :: "'coll ⇒ 'i ⇒ 'coll"
    and idx_reset     :: "'coll ⇒ 'i ⇒ 'coll"
    and K             :: "'i set"
  assumes idx_partition_disjoint:
      "⋀C i. ⟦idx_invar C; i ∈ K⟧ ⟹ idx_iterated C i ∩ idx_remaining C i = {}"
    and idx_partition_union:
      "⋀C i.⟦idx_invar C; i ∈ K⟧  ⟹ idx_iterated C i ∪ idx_remaining C i = idx_abstract C
     i"
    and idx_has:
      "⋀C i. ⟦idx_invar C; i ∈ K⟧  ⟹ idx_has C i ⟷ idx_remaining C i ≠ {}"
    and idx_current:
      "⋀C i. ⟦idx_invar C; i ∈ K; idx_remaining C i ≠ {}⟧ ⟹ idx_current C i ∈
    idx_remaining C i"
    and idx_move_invar:
      "⋀C i. ⟦idx_invar C; i ∈ K⟧ ⟹ idx_invar (idx_move C i)"
    and idx_move_abstract:
      "⋀C i j. ⟦idx_invar C; i ∈ K; idx_remaining C i ≠ {}⟧ ⟹ idx_abstract (idx_move C i)
     j = idx_abstract C j"
    and idx_move_remaining:
      "⋀C i. ⟦idx_invar C; i ∈ K; idx_remaining C i ≠ {}⟧ ⟹ idx_remaining (idx_move C i)
    i = idx_remaining C i - {idx_current C i}"
    and idx_move_iterated:
      "⋀C i. ⟦idx_invar C; i ∈ K; idx_remaining C i ≠ {}⟧ ⟹ idx_iterated (idx_move C i) i
     = idx_iterated C i ∪ {idx_current C i}"
    and idx_move_remaining_other:
      "⋀C i j. ⟦idx_invar C;i ∈ K; idx_remaining C i ≠ {}; j ≠ i⟧ ⟹ idx_remaining (
    idx_move C i) j = idx_remaining C j"
    and idx_move_iterated_other:
      "⋀C i j. ⟦idx_invar C;i ∈ K; idx_remaining C i ≠ {}; j ≠ i⟧ ⟹ idx_iterated (
    idx_move C i) j = idx_iterated C j"
    and idx_reset_invar:
      "⋀C i. ⟦idx_invar C; i ∈ K⟧ ⟹ idx_invar (idx_reset C i)"
    and idx_reset_abstract:
      "⋀C i j. ⟦idx_invar C; i ∈ K⟧ ⟹ idx_abstract (idx_reset C i) j = idx_abstract C j"
    and idx_reset_iterated:
      "⋀C i. ⟦idx_invar C; i ∈ K⟧ ⟹ idx_iterated (idx_reset C i) i = {}"
    and idx_reset_remaining:
      "⋀C i. ⟦idx_invar C; i ∈ K⟧ ⟹ idx_remaining (idx_reset C i) i = idx_abstract C i"
    and idx_reset_remaining_other:
      "⋀C i j. ⟦idx_invar C; i ∈ K; j ≠ i⟧ ⟹ idx_remaining (idx_reset C i) j =
    idx_remaining C j"
    and idx_reset_iterated_other:
      "⋀C i j. ⟦idx_invar C; i ∈ K; j ≠ i⟧ ⟹ idx_iterated (idx_reset C i) j =
    idx_iterated C j"
```

Listing 46. `locale weighted_csr` (`CSR_Graph.thy`, branch `more_algorithms`)

```
locale weighted_csr = csr_graph n K B E
  for n K B and E :: "'e::heap list" +
  fixes tgt :: "'e ⇒ 'a"
    and W :: "'w::heap list"
    and cost :: "nat ⇒ 'a ⇒ real"
    and wval :: "'w ⇒ real"
  assumes tgt_inj: "v ∈ K ⟹ inj_on tgt ((!) E ` {B ! v..<B ! Suc v})"
    and W_length: "length W = length E"
    and W_cost: "⟦v ∈ K; B ! v ≤ j; j < B ! Suc v⟧ ⟹ wval (W ! j) = cost v (tgt (E ! j))"
```

Listing 47. `locale ingoing_edge_iterator` (`Acyclic_Flow.thy`, branch `more_algorithms`)

```
locale ingoing_edge_iterator =
  multigraph_spec where fst = fst +
  ing: indexed_iterable_set where
      idx_invar     = in_invar and
      idx_abstract  = in_abstract and
      idx_current   = in_current and
      idx_has       = in_has and
      idx_iterated  = in_iterated and
      idx_remaining = in_remaining and
      idx_move      = in_move and
      idx_reset     = in_reset and
      K = 𝒱
  for fst :: "'e ⇒ 'v"
  and in_invar     :: "'g ⇒ bool"
  and in_abstract  :: "'g ⇒ 'v ⇒ 'e set"
  and in_current   :: "'g ⇒ 'v ⇒ 'e"
  and in_has       :: "'g ⇒ 'v ⇒ bool"
  and in_iterated  :: "'g ⇒ 'v ⇒ 'e set"
  and in_remaining :: "'g ⇒ 'v ⇒ 'e set"
  and in_move      :: "'g ⇒ 'v ⇒ 'g"
  and in_reset     :: "'g ⇒ 'v ⇒ 'g"
```

Listing 48. `locale multigraph_spec` (`Multigraph.thy`)

```
locale multigraph_spec =
  fixes ℰ::"'edge set"
and fst::"'edge ⇒ 'a"
and snd::"'edge ⇒ 'a"
and create_edge:: "'a ⇒ 'a ⇒ 'edge"
```

Listing 49. `function DFS` (`DFS.thy`)

```
function (domintros) DFS::"('v, 'vset) DFS_state ⇒ ('v, 'vset) DFS_state" where
  "DFS dfs_state =
     (case (stack dfs_state) of (v # stack_tl) ⇒
       (if v = t then
          (dfs_state (|return := Reachable|))
        else ((if (𝒩_G v -_G (seen dfs_state)) ≠ ∅_N then
                  let u = (sel ((𝒩_G v) -_G (seen dfs_state)));
                      stack' = u# (stack dfs_state);
                      seen' = insert u (seen dfs_state)
                  in
                    (DFS (dfs_state (|stack := stack',
                                      seen := seen' |)))
                else
                  let stack' = stack_tl in
                     DFS (dfs_state (|stack := stack'|)))
              )
      )
     | _ ⇒ (dfs_state (|return := NotReachable|))
    )"
```

Listing 50. `record DFS_state` (`DFS.thy`)

```
record ('ver, 'vset) DFS_state = stack:: "'ver list" seen:: "'vset"  return:: return
```

Listing 51. `locale find_max_match` (`Blossom_Algo_Main_Loop.thy`)

```
locale find_max_match = graph_abs G for G +
  fixes aug_path_search::"'a set set ⇒ 'a set set ⇒ ('a list) option"
  assumes
    aug_path_search_complete:
    "⟦matching M; M ⊆ G; finite M; (∃p. graph_augmenting_path G M p)⟧
     ⟹ (∃p. aug_path_search G M = Some p)" and
    aug_path_search_sound:
    "⟦matching M; M ⊆ G; finite M; aug_path_search G M = Some p⟧
     ⟹ graph_augmenting_path G M p"
begin
```

Listing 52. `locale BFS` (`BFS_2.thy`)

```
locale BFS =
  Graph: Pair_Graph_Specs
    where lookup = lookup +
  set_ops: Set2 vset_empty vset_delete _ t_set vset_inv insert

for lookup :: "'adjmap ⇒ 'ver ⇒ 'vset option" +

fixes
srcs::"'vset" and
G::"'adjmap" and expand_tree::"'adjmap ⇒ 'vset ⇒ 'vset ⇒ 'adjmap" and
next_frontier::"'vset ⇒ 'vset ⇒ 'vset"

assumes
   expand_tree[simp]:
     "⟦Graph.graph_inv BFS_tree; vset_inv frontier; vset_inv vis; Graph.graph_inv G⟧ ⟹
       Graph.graph_inv (expand_tree BFS_tree frontier vis)"
     "⟦Graph.graph_inv BFS_tree; vset_inv frontier; vset_inv vis; Graph.graph_inv G⟧ ⟹
        Graph.digraph_abs (expand_tree BFS_tree frontier vis) =
         (Graph.digraph_abs BFS_tree) ∪
         {(u,v) | u v. u ∈ t_set (frontier) ∧
                       v ∈ (Pair_Graph.neighbourhood (Graph.digraph_abs G) u -
                       t_set vis)}"
    "⟦Graph.graph_inv BFS_tree; vset_inv frontier; vset_inv vis; Graph.graph_inv G;
        Graph.finite_graph BFS_tree⟧ ⟹
        Graph.finite_graph (expand_tree BFS_tree frontier vis)" and
   next_frontier[simp]:
    "⟦vset_inv frontier; vset_inv vis; Graph.graph_inv G⟧ ⟹  vset_inv (next_frontier
    frontier vis)"
    "⟦vset_inv frontier; vset_inv vis; Graph.graph_inv G⟧ ⟹
       t_set (next_frontier frontier vis) =
         (⋃ {Pair_Graph.neighbourhood (Graph.digraph_abs G) u | u . u ∈ t_set frontier}) -
    t_set vis"
```

Listing 53. `lemmas DFS_correct_1, DFS_correct_1_strong and DFS_correct_2` (`DFS.thy`)

```
lemma DFS_correct_1:
  assumes "return (DFS initial_state) = NotReachable"
  shows   "∄p. distinct p ∧ vwalk_bet (Graph.digraph_abs G) s p t"

lemma DFS_correct_1_strong:
  assumes "return (DFS initial_state) = NotReachable"
  shows   "∄p. vwalk_bet (Graph.digraph_abs G) s p t"

lemma DFS_correct_2:
  assumes  "return (DFS initial_state) = Reachable"
```

```
  shows "vwalk_bet (Graph.digraph_abs G) s (rev (stack (DFS initial_state))) t" (is ?thesis1)
        "distinct (rev (stack (DFS initial_state))) " (is ?thesis2)
```

Listing 54. lemmas DFS_Cycles_correct_1 and DFS_Cycles_correct_2 (DFS_Cycles.thy)

```
lemma DFS_Cycles_correct_1:
  assumes "cycle (DFS_Cycles initial_state)"
  shows "(∃c. cycle' (Graph.digraph_abs G) c)"

lemma DFS_Cycles_correct_2:
  assumes "¬cycle (DFS_Cycles initial_state)"
  shows "∄c. cycle' (Graph.digraph_abs G) c"
```

Listing 55. theorems DFS_dircycle_sound and DFS_dircycle_complete (DFS_DirCycle.thy, branch dfs_refactor)

```
theorem DFS_dircycle_sound:
  assumes "cycle (dc.DFS_skel dircycle_initial_state)"
  shows "∃c. Awalk_Defs.cycle (Graph.digraph_abs G) c"

theorem DFS_dircycle_complete:
  assumes "¬ cycle (dc.DFS_skel dircycle_initial_state)"
  shows "∄c. Awalk_Defs.cycle (Graph.digraph_abs G ↓ t_set (seen (dc.DFS_skel
    dircycle_initial_state))) c"
```

Listing 56. lemmas BFS_correct_1, BFS_correct_2, BFS_correct_3 and BFS_correct_4 (BFS_2.thy)

```
lemma BFS_correct_1:
  "⟦u ∈ t_set srcs; t ∉ t_set (visited (BFS initial_state))⟧
         ⟹ ∄p. vwalk_bet (Graph.digraph_abs G) u p t"

lemma BFS_correct_2:
  "⟦t ∈ t_set (visited (BFS initial_state)) - t_set srcs⟧
         ⟹ distance_set (Graph.digraph_abs G) (t_set srcs) t =
         distance_set (Graph.digraph_abs (parents (BFS initial_state))) (t_set srcs) t"

lemma BFS_correct_3:
  "⟦u ∈ t_set srcs; Vwalk.vwalk_bet (Graph.digraph_abs (parents (BFS initial_state))) u p v⟧
         ⟹ length p - 1 = distance_set (Graph.digraph_abs G) (t_set srcs) v"

lemma BFS_correct_4:
  "(Graph.digraph_abs (parents (BFS initial_state))) ⊆ (Graph.digraph_abs G)"
```

Listing 57. theorem BFS_par_correct (BFS_3.thy, branch more_algorithms)

```
theorem BFS_par_correct:
  assumes "final_state = BFS_par_impl initial_par_state"
  shows "x ∈ t_set (visited final_state) ⟷
           (∃u ∈ t_set srcs. ∃q. vwalk_bet (Graph.digraph_abs G) u q x)"
    and "x ∈ t_set (visited final_state) ⟹
           dist_lookup (dists final_state) x = distance_set (Graph.digraph_abs G) (t_set srcs
    ) x"
    and "x ∈ t_set (visited final_state) - t_set srcs ⟹
           parent_lookup (parent final_state) x ∈ t_set (visited final_state) ∧
           (parent_lookup (parent final_state) x, x) ∈ Graph.digraph_abs G ∧
           distance_set (Graph.digraph_abs G) (t_set srcs) x =
             distance_set (Graph.digraph_abs G) (t_set srcs) (parent_lookup (parent
    final_state) x) + 1"
    and "x ∈ t_set (visited final_state) ⟹
           ∃u ∈ t_set srcs.
             vwalk_bet (Graph.digraph_abs G) u (parent_path (dists final_state) (parent
    final_state) x []) x ∧
```

```
    length (parent_path (dists final_state) (parent final_state) x []) - 1 =
      distance_set (Graph.digraph_abs G) (t_set srcs) x"
```

Listing 58. theorem bellman_ford_computes_length_of_optpath (Bellman_Ford.thy)

```
theorem bellman_ford_computes_length_of_optpath:
  assumes "s ≠ t" "s ∈ set vs" "t∈ set vs"
  obtains P where "opt_vs_path s t P"
    "weight P = snd (the (connection_lookup (local.bellman_ford (length vs - 1)
                         (bellman_ford_init s)) t))"
```

Listing 59. theorem dijkstra_compute_optimal (Dijkstra.thy, branch more_algorithms)

```
theorem dijkstra_compute_optimal:
  assumes "v ∈ seen_abstract (dij_seen dijkstra_compute)"
  shows "ereal (dist_lookup (dij_dist dijkstra_compute) v) = ag.distance_set w (src_abstract
    srcs) v"
```

Listing 60. lemma BestInGreedy_matroid_iff_opt (Best_In_Greedy.thy)

```
lemma BestInGreedy_matroid_iff_opt:
  "(matroid.matroid_axioms carrier indep_fn) =
    (∀c. nonnegative c ⟶ (∀order. valid_order order ⟶ (∀X. valid_solution X ⟶
    c_set c (to_set (result (BestInGreedy (initial_state c order)))) ≥ c_set c (to_set X))))
    "
```

Listing 61. corollary kruskal_computes_max_spanning_forest (Kruskal_Greedy.thy)

```
corollary kruskal_computes_max_spanning_forest:
  "max_forest (t_set (result (kruskal  input_G (kruskal_init c order))))"
```

Listing 62. theorem greedoid_characterisation (Greedoids_Optimisation.thy)

```
theorem greedoid_characterisation:
  "(∀ c es. valid_modular_weight_func E c ∧  E = set es ∧ distinct es
          ⟶ opt_basis c (set (greedoid_greedy es c Nil)))
 ⟷ strong_exchange_property E F"
```

Listing 63. theorem jp_maximum_stree (Jarnik_Prim.thy)

```
theorem jp_maximum_stree:
  "stree_around_seed final_arborescence"
  "∄ T. stree_around_seed T ∧ cc T > cc final_arborescence"
```

Listing 64. theorem total_correctness (Successive_Shortest_Path.thy)

```
theorem total_correctness:
  assumes "final = SSP ⦇current_flow = (λ e. 0), balance = b,  return = notyetterm⦈"
  shows   "return final = success ⟹ is_Opt b (current_flow final)"
          "return final = failure ⟹ ∄ f. is_Opt b f"
          "return final = notyetterm ⟹ False"
```

Listing 65. theorem total_correctness (Scaling.thy)

```
theorem total_correctness:
  assumes "L = ⌈ log 2 (max 1 ((∑ v ∈ 𝒱. abs (b v))/2))⌉"
```

```
        "final = Scaling L (|current_flow = (λ e. 0), balance = b,  return = notyetterm|)"
  shows   "return final = success ⟹ is_Opt b (current_flow final)" (is "?succ ⟹ ?opt")
          "return final = failure ⟹ ∄ f. f is b flow" (is "?fail ⟹ ?noopt")
          "return final = notyetterm ⟹ False" (is "?no_term ⟹ _")
```

Listing 66. `lemma dinic_total_correctness` (`Dinic.thy`)

```
lemma dinic_total_correctness:
  "is_max_flow s t (abstract_real_map (flow_lookup (current_flow (dinic dinic_initial))))"
  "dinic_dom dinic_initial"
```

Listing 67. `lemma correctness` (`Compute_Blocking_Simple.thy`)

```
lemma correctness:
  "flow (compute_blocking_loop initial_state) = flow_empty
        ⟹ (∄ p. vwalk_bet (Graph.digraph_abs G) s p t)"
  "flow  (compute_blocking_loop initial_state) ≠flow_empty
        ⟹ ∃ p. vwalk_bet (Graph.digraph_abs G) s p t"
  "flow  (compute_blocking_loop initial_state) ≠flow_empty
        ⟹ flow_network_spec.is_blocking_flow fst snd (Graph.digraph_abs G) u s t
             (abstract_real_map (flow_lookup (flow (compute_blocking_loop initial_state))))
    "
  "compute_blocking_loop_dom initial_state"
  "flow  (compute_blocking_loop initial_state) ≠flow_empty  ⟹ dom (flow_lookup (flow (
    compute_blocking_loop initial_state))) ⊆ Graph.digraph_abs G"
  "flow  (compute_blocking_loop initial_state) ≠flow_empty  ⟹ flow_invar  (flow (
    compute_blocking_loop initial_state))"
```

Listing 68. `lemma dfs_backtrack_final` (`DFS_Collect_Backtrack.thy`)

```
lemma dfs_backtrack_final:
  "invar_dfs_backtrack_1 (DFS_collect_backtrack dfs_backtrack_initial_state)"
  "invar_dfs_backtrack_2 (DFS_collect_backtrack dfs_backtrack_initial_state)"
  "invar_dfs_backtrack_3 (DFS_collect_backtrack dfs_backtrack_initial_state)"
  "invar_dfs_backtrack_4 (DFS_collect_backtrack dfs_backtrack_initial_state)"
  "dir_acyc (Graph.digraph_abs G) ⟹
 invar_dfs_backtrack_5 (DFS_collect_backtrack dfs_backtrack_initial_state)"
```

Listing 69. `theorem network_simplex_correct` (`Network_Simplex_Final.thy`, branch `more_algorithms`)

```
theorem network_simplex_correct:
  "ns_loop_dom init_state"
  "return (ns_loop init_state) = unbounded ⟹ has_neg_infty_cycle make_pair ℰ c u"
  "return (ns_loop init_state) = success ⟹ is_Opt b (ns_flow_of (ns_loop init_state))"
```

Listing 70. `theorem make_acyclic_correct_unconditional_bflow` (`Acyclic_Flow.thy`, branch `more_algorithms`)

```
theorem make_acyclic_correct_unconditional_bflow:
  assumes mi0: "multigraph_inv out_arr in_arr"
    and ff: "flow_invar f0" and feas: "af_feasible b f0"
    and vv: "vit_invar all_vertices" and vabsV: "vit_abstract all_vertices = 𝒱"
    and vit0: "vit_iterated all_vertices = {}"
    and fresh: "∀u. out_iterated out_arr u = {} ∧ in_iterated in_arr u = {}"
    and res: "make_acyclic f0 = Some f'"
  shows "af_feasible b f' ∧ 𝒞 (h ∘ flow_lookup f') ≤ 𝒞 (h ∘ flow_lookup f0) ∧ acyclic_flow
    (h ∘ flow_lookup f')"
```

Listing 71. `theorem push_relabel_correct` (`Push_Relabel_Generic.thy`, branch `more_algorithms`)

```
theorem push_relabel_correct: "is_max_flow s t (emap_get push_relabel)"
```

Listing 72. theorem naive_min_weight_perfect_matching_partial_correctness (Naive_Weighted_Blossom.thy, branch weighted_blossom)

```
theorem naive_min_weight_perfect_matching_partial_correctness:
  assumes "top_loop_dom (π₀, 𝔒₀)"
  shows "naive_min_weight_perfect_matching = None
          ⟹ ∄ M. perfect_matching G M"
        "naive_min_weight_perfect_matching = Some M
          ⟹ min_weight_perfect_matching G w M"
```

Listing 73. lemma find_edg_correct (Edmonds_Gallai_Computation.thy, branch weighted_blossom)

```
lemma find_edg_correct:
  assumes  "graph_invar E" "max_card_matching E M"
  shows "edmonds_gallai E M (fst (find_edg E M)) (snd (find_edg E M))"
```

Listing 74. lemma Hopcroft_Karp_Loop_correct_1 (Hopcroft_Karp_Top_Loop.thy, development version)

```
lemma Hopcroft_Karp_Loop_correct_1:
  "max_card_matching ugraph_abs (matching_abstract (current_matching (Hopcroft_Karp_Loop
    initial_state)))"
```

Listing 75. theorem hungarian_correctness (Hungarian_Method_Top_Loop.thy)

```
theorem hungarian_correctness:
 "hungarian = None ⟹ ∄ M. perfect_matching G M"
 "hungarian = Some M ⟹ min_weight_perfect_matching G edge_costs (matching_abstract M)"
```

Listing 76. lemma naive_primal_dual_total_correctness (Naive_Primal_Dual.thy)

```
lemma naive_primal_dual_total_correctness:
  assumes "multiples_of α w G"
  shows   "max_weight_matching G w (g_to_set (naive_primal_dual init_potential))"
    "∃ π. min_feasible_max_dual LR G w (potential π)"
```

Listing 77. theorem ranking_comp_ratio (Ranking_Probabilistic.thy)

```
theorem ranking_comp_ratio: "measure_pmf.expectation ranking card / (card M) ≥ 1 - (1 - 1/(
    card M + 1)) ^ (card M)"
```

Listing 78. theorem ranking_competitive_ratio (Ranking_Order_Probabilistic.thy)

```
theorem
  assumes "bipartite G L R"
  assumes "finite L" "finite R" "Vs G = L ∪ R"

  assumes "G ≠ {}"
  assumes "π ∈ permutations_of_set R"
  assumes "max_card_matching G M"

  shows ranking_competitive_ratio:
    "prob_space.expectation (Π⇩M i ∈ L. uniform_measure lborel {0..1::real}) (λY. card (
    ranking (linorder_from_keys L Y) G π)) / card M
      ≥ 1 - exp(-1)" (is "?EY ≥ _")

  and ranking_competitive_ratio_random_linorder:
    "prob_space.expectation (uniform_measure (count_space (Pow (L × L))) (linorders_on L)) (
    λr. card (ranking r G π)) / card M
      ≥ 1 - exp(-1)" (is "?Er ≥ _")
```

Listing 79. `theorem adwords_competitive_ratio` (`AdWords.thy`)

```
theorem adwords_competitive_ratio:
  assumes "max_value_allocation M"
  shows "allocation_value (allocation (adwords π)) / allocation_value M ≥ (1 - 1/c) * (1 -
    R_max)"
```

Listing 80. `lemma matroid_intersection_total_correctness` (`Matroid_Intersection_Algorithm.thy`)

```
lemma matroid_intersection_total_correctness:
  "is_max (to_set (sol (matroid_intersection initial_state)))"
  "matroid_intersection_dom initial_state"
```

Listing 81. `theorem weighted_matroid_intersection_correct` (`Matroid_Weighted_Intersection_Path_Loop.thy`, branch `more_algorithms`)

```
theorem weighted_matroid_intersection_correct:
  assumes "c_invar c"
  defines "s ≡ weighted_matroid_intersection (weighted_initial_state c)"
  shows "weighted_double_matroid.is_opt indep1 indep2 (c_lookup c) (to_set (wbest s))"
```

Listing 82. `corollary partition_opt_from_is_max` (`Matroid_Partitioning.thy`, branch `more_algorithms`)

```
corollary partition_opt_from_is_max:
  assumes "double_matroid.is_max (pindep1 carrier) (pindep2 carrier) Q"
  shows "partition_opt (fst ` Q)"
```

Listing 83. `definition DFS_call_1_conds` (`DFS.thy`)

```
definition "DFS_call_1_conds dfs_state =
    (case stack dfs_state of (v # stack_tl) ⇒
       (if v = t then
          False
        else ((if ((𝒩_G v) -_G (seen dfs_state)) ≠ (∅_N) then
                  True
                else False)
              )
      )
     | _ ⇒ False
    )"
```

Listing 84. `definition DFS_upd1` (`DFS.thy`)

```
definition "DFS_upd1 dfs_state = (
    let
      N = (𝒩_G (hd (stack dfs_state)));
      u = (sel ((N -_G (seen dfs_state))));
      stack' = u # (stack dfs_state);
      seen' = insert u (seen dfs_state)
    in
      dfs_state (|stack := stack', seen := seen'|))"
```

Listing 85. `lemma DFS_cases` (`DFS.thy`)

```
lemma DFS_cases:
  assumes "DFS_call_1_conds dfs_state ⟹ P"
      "DFS_call_2_conds dfs_state ⟹ P"
      "DFS_ret_1_conds dfs_state ⟹ P"
      "DFS_ret_2_conds dfs_state ⟹ P"
```

Listing 86. `lemma DFS_simps` (`DFS.thy`)

```
lemma DFS_simps:
  assumes "DFS_dom dfs_state"
  shows"DFS_call_1_conds dfs_state ⟹ DFS dfs_state = DFS (DFS_upd1 dfs_state)"
      "DFS_call_2_conds dfs_state ⟹ DFS dfs_state = DFS (DFS_upd2 dfs_state)"
      "DFS_ret_1_conds dfs_state ⟹ DFS dfs_state = DFS_ret1 dfs_state"
```

Listing 87. `lemma DFS_induct` (`DFS.thy`)

```
lemma DFS_induct:
  assumes "DFS_dom dfs_state"
  assumes "⋀dfs_state. ⟦DFS_dom dfs_state;
                       DFS_call_1_conds dfs_state ⟹ P (DFS_upd1 dfs_state);
                       DFS_call_2_conds dfs_state ⟹ P (DFS_upd2 dfs_state)⟧ ⟹ P
    dfs_state"
  shows "P dfs_state"
```

Listing 88. `lemma DFS_call_1_conds` (`DFS.thy`)

```
lemma DFS_call_1_conds[call_cond_elims]:
  "DFS_call_1_conds dfs_state ⟹
   ⟦⟦∃v stack_tl. stack dfs_state = v # stack_tl;
    hd (stack dfs_state) ≠ t;
    (𝒩_G (hd (stack dfs_state))) -_G (seen dfs_state) ≠ (∅_N)⟧ ⟹ P⟧ ⟹
```

Listing 89. `lemma invar_1_holds_1` (`DFS.thy`)

```
lemma invar_1_holds_1[invar_holds_intros]:
  "⟦DFS_call_1_conds dfs_state; invar_1 dfs_state⟧ ⟹
      invar_1 (DFS_upd1 dfs_state)"
```

Listing 90. `lemma invar_1_holds` (`DFS.thy`)

```
lemma invar_1_holds[invar_holds_intros]:
   assumes "DFS_dom dfs_state" "invar_1 dfs_state"
   shows "invar_1 (DFS dfs_state)"
```

Listing 91. collections of theorems (`named_theorems`) (`DFS.thy`)

```
named_theorems call_cond_elims
named_theorems call_cond_intros
named_theorems ret_holds_intros
named_theorems invar_props_intros
named_theorems invar_props_elims
named_theorems invar_holds_intros
named_theorems state_rel_intros
named_theorems state_rel_holds_intros
```

Listing 92. `lemma BFS_induct` (`BFS_2.thy`)

```
lemma BFS_induct:
  assumes "BFS_dom bfs_state"
  assumes "⋀bfs_state. ⟦BFS_dom bfs_state;
                       (BFS_call_1_conds bfs_state ⟹ P (BFS_upd1 bfs_state))⟧
             ⟹ P bfs_state"
```

Listing 93. `lemma BestInGreedy_induct` (`Best_In_Greedy.thy`)

```
lemma BestInGreedy_induct:
  assumes "BestInGreedy_dom best_in_greedy_state"
  assumes "⋀best_in_greedy_state. ⟦BestInGreedy_dom best_in_greedy_state;
```

```
    BestInGreedy_call_1_conds best_in_greedy_state ⟹ P (BestInGreedy_upd1
    best_in_greedy_state);
    BestInGreedy_call_2_conds best_in_greedy_state ⟹ P (BestInGreedy_upd2
    best_in_greedy_state)⟧ ⟹
      P best_in_greedy_state"
```

Listing 94. `lemma SSP_induct` (`Successive_Shortest_Path.thy`)

```
lemma SSP_induct:
  assumes "SSP_dom state"
  assumes "⋀state. ⟦SSP_dom state;
                    SSP_call_4_cond state ⟹ P (SSP_upd4 state)⟧ ⟹ P state"
```

Listing 95. `lemma Scaling_induct` (`Scaling.thy`)

```
lemma Scaling_induct:
  assumes "Scaling_dom (k,state)"
  assumes "⋀k state. ⟦Scaling_dom (k, state);
                     Scaling_call_4_cond k state ⟹ P (k-1) (Scaling_upd4 k state)⟧
          ⟹ P k state"
```

Listing 96. `lemma send_flow_induct` (`Send_Flow.thy`)

```
lemma send_flow_induct:
  assumes "send_flow_dom state"
    "⋀ state. ⟦ send_flow_dom state ;
              send_flow_call1_cond state ⟹ P (send_flow_call1_upd state);
              send_flow_call2_cond state ⟹ P (send_flow_call2_upd state) ⟧ ⟹ P state"
```

Listing 97. `lemma maintain_forest_induct` (`Maintain_Forest.thy`)

```
lemma maintain_forest_induct:
  assumes "maintain_forest_dom state"
  assumes "⋀state. ⟦maintain_forest_dom state;
                    maintain_forest_call_cond state ⟹ P (maintain_forest_upd state)⟧ ⟹
    P state"
  shows "P state"
```

Listing 98. `lemma hungarian_loop_induct` (`Hungarian_Method_Top_Loop.thy`)

```
lemma hungarian_loop_induct:
  assumes "hungarian_loop_dom state"
          "⋀ state. ⟦hungarian_loop_dom state;
                     hungarian_loop_cont_cond state ⟹ P (hungarian_loop_upd state)⟧
                    ⟹ P state"
```

Listing 99. `lemma dinic_induct` (`Dinic.thy`)

```
lemma dinic_induct:
  assumes "dinic_dom state"
  assumes "⋀ state.
         ⟦ dinic_dom state; dinic_upd_cond state  ⟹ P (dinic_upd state)⟧ ⟹ P state"
```

Listing 100. `lemma matroid_intersection_induct` (`Matroid_Intersection_Algorithm.thy`)

```
lemma matroid_intersection_induct:
  assumes "matroid_intersection_dom state"
    "⋀ state. matroid_intersection_dom state  ⟹
                  (matroid_intersection_recurse_cond state
 ⟹ P (matroid_intersection_recurse_upd state)) ⟹ P state"
```

Listing 101. `definition is_iam` (`Array_Map_Impl.thy`)

```
definition is_iam :: "(nat⇀'a) ⇒ ('a::heap) array_map ⇒ assn" where
  "is_iam m a ≡ ∃A l. a↦ s l * ↑(m=iam_of_list l)"
```

Listing 102. `locale Set_Choose` (`Set_Choose.thy`, branch `bfs_refinement`)

```
locale Set_Choose = set: Set
  where set = t_set for t_set ("[_]s") +
fixes sel ::"'s ⇒ 'a"

assumes choose [simp]: "s ≠ empty ⟹ isin s (sel s)"
```

Listing 103. `interpretation iam_dfs: imp_DFS` (`DFS_Imperative.thy`, branch `bfs_refinement`)

```
interpretation iam_dfs: imp_DFS
  is_iam iam_new iam_lookup iam_update
  is_ias ias_new ias_memb ias_ins ias_delete
```

Listing 104. `interpretation hm_dfs: imp_DFS` (`DFS_Imperative.thy`, branch `bfs_refinement`)

```
interpretation hm_dfs: imp_DFS
  is_hashmap hm_new hm_lookup hm_update
  is_hashset hs_new hs_memb hs_ins hs_delete
```

Listing 105. `lemma DFS_imperative_rule` (`DFS_Imperative.thy`)

```
lemma DFS_imperative_rule[sep_heap_rules]: "<graph_assn G Gi * DFS_state_assn dfs_state
    dfs_statei> DFS_imperative Gi t dfs_statei <λri. graph_assn G Gi * DFS_state_assn (
    DFS_impl G t dfs_state) ri>"
```

Listing 106. `lemma DFS_imperative'_absrl` (`DFS_Imperative.thy`, branch `bfs_refinement`)

```
lemma DFS_imperative'_absrl:
      "<graph_assn G Gi> DFS_imperative' Gi s t <λri. graph_assn G Gi * ↑(
      case DFS_state.return ri of
        NotReachable ⇒ ∄p. distinct p ∧ vwalk_bet (digraph_abs G) s p t
      | Reachable ⇒ vwalk_bet (digraph_abs G) s (rev (stack (local.DFS G t (initial_state s)
    ))) t
    )>"
```

Listing 107. `global_interpretation Kruskal_Greedy: BestInGreedy_exec` (`Kruskal_Functional.thy`)

```
global_interpretation Kruskal_Greedy: BestInGreedy_exec
  where set_empty = "⟨⟩" and set_insert = insert_rbt and set_delete = delete_rbt
    and set_inv = "vset_inv::(('v::linorder × 'v) rbt ⇒ bool)" and
    set_isin = isin and to_set = "Tree2.set_tree" and union = "RBT.union" and inter = "RBT.
    inter" and diff = "RBT.diff" and
    cardinality = size and
    carrier = input_G and
    indep_fn = "indep_graph_matroid  input_G" and
    sort_desc = insort_key_desc and
    local_indep_oracle = "local_indep_oracle  input_G"
    and wrapper_insert = insert_rbt
    and wrapper_invar = "vset_inv::(('v × 'v) rbt ⇒ bool)"
    and remove_wrapper = id
    and wrapper_empty = Leaf
  for  input_G
  defines kruskal' = Kruskal_Greedy.BestInGreedy_exec
```

Listing 108. `theorem solution` (`Max_Bipartite_Matching_Matroid.thy`)

```
theorem solution:"graph_matching (to_dbltn ` Edges)
                  (to_dbltn ` (set_matching (sol (matroid_intersection_circuit_impl
                                  left_vertex right_vertex s t Edges_impl initial_state))))"
  (is ?thesis1)
  and   "∄ M. graph_matching (to_dbltn ` Edges) M ∧
               card M > card  (set_matching (sol (matroid_intersection_circuit_impl
                                 left_vertex right_vertex s t Edges_impl initial_state)))"
  (is ?thesis2)
```

Listing 109. `definition local_indep_oracle` (`Kruskal_Greedy.thy`)

```
definition "local_indep_oracle e X =
((Card_Set2_RBT.subseteq (vset_insert e X) input_G) ∧
 (lookup (Kruskal_E_to_G X) (fst e) ≠ None ∧ lookup (Kruskal_E_to_G X) (snd e) ≠ None⟶
           return (dfs_impl (Kruskal_E_to_G X) (snd e) (dfs_initial_state (fst e)))
                   = NotReachable))"
```

Listing 110. `theorem reduction_of_min_cost_flow_to_hitchcock_general` (`Hitchcock_Reduction.thy`)

```
theorem reduction_of_min_cost_flow_to_hitchcock_general:
  fixes c b D
  assumes "flow_network fstv sndv create_edge (u::'edge ⇒ ereal) 𝓔"
  defines "make_pair ≡ multigraph_spec.make_pair fstv sndv"
  defines "fstv' ≡ new_fstv_gen fstv"
  defines "sndv' ≡ new_sndv_gen fstv sndv"
  defines "make_pair' ≡ multigraph_spec.make_pair fstv' sndv'"
  defines "create_edge' ≡ new_create_edge_gen"
  defines "𝓔1 ≡ new_𝓔1_gen 𝓔 u"
  defines "𝓔2 ≡ new_𝓔2_gen 𝓔 u"
  defines "𝓔3 ≡ new_𝓔3_gen 𝓔 u"
  defines "𝓔' ≡ 𝓔1 ∪ 𝓔2 ∪ 𝓔3"
  defines "u' ≡ (λ e. PInfty)"
  defines "c' ≡ new_c_gen D fstv 𝓔 u c"
  defines "b' ≡ new_b_gen fstv 𝓔 u b"
  shows
    "flow_network fstv' sndv' create_edge' u' 𝓔'" (is ?case1) and
    "⋀ f f' . ⟦flow_network_spec.isbflow fstv sndv  𝓔 u f b; f' = new_f_gen fstv 𝓔 u  f⟧
       ⟹flow_network_spec.isbflow fstv' sndv'  𝓔' u' f' b' ∧ (cost_flow_spec.𝓒 𝓔 c f =
    cost_flow_spec.𝓒 𝓔' c' f')"
    (is "⋀ f f' . ?a f f' ⟹ ?b f f'  ⟹ ?c f f'") and
    "⋀ f f'. ⟦flow_network_spec.isbflow fstv' sndv' 𝓔' u' f' b';
          f = old_f_gen 𝓔 u  f' ⟧ ⟹
          flow_network_spec.isbflow fstv sndv 𝓔 u f b
        ∧ (cost_flow_spec.𝓒 𝓔 c f = cost_flow_spec.𝓒 𝓔' c' f')"
    (is "⋀ f f'. ?x f f' ⟹ ?y f f' ⟹ ?z f f'") and
    "has_neg_cycle make_pair' 𝓔' c' ⟷ has_neg_infty_cycle make_pair 𝓔 c u" and
    "⋀ f f' . ⟦ f = old_f_gen 𝓔 u f'; cost_flow_spec.is_Opt fstv' sndv' u' 𝓔' c' b' f'⟧
           ⟹ cost_flow_spec.is_Opt fstv sndv u 𝓔 c b f"
    (is "⋀ f f'.  ?b1 f f'⟹ ?c1 f' ⟹ ?d1 f ") and
    "⋀ f f' . ⟦ f' = new_f_gen fstv 𝓔 u f; cost_flow_spec.is_Opt fstv sndv  u 𝓔 c b f⟧
⟹  cost_flow_spec.is_Opt fstv' sndv'  u' 𝓔' c' b' f'"
    (is "⋀ f f'.  ?b2 f f'⟹ ?c2 f ⟹ ?d3 f' ")
```

Listing 111. `corollary correctness_of_implementation_success` (`Usage_Capacitated.thy`)

```
corollary correctness_of_implementation_success:
 "return (final_state_cap) = success ⟹
        is_Opt b (abstract_flow_map (final_flow_impl_original))"
```

Listing 112. `corollary is_opt_iff_no_augcycle` (`Cost_Optimality.thy`)

```
corollary is_opt_iff_no_augcycle:
  assumes "f is b flow"
  shows "is_Opt b f ⟷ (∄ cs. augcycle f cs)"
```

Listing 113. `theorem path_aug_opt_pres` (`Path_Aug_Opt.thy`)

```
theorem path_aug_opt_pres:
  assumes "s ≠ t" "is_Opt b f " "γ ≤ Rcap f (set P)"  " 0 ≤ γ"  "is_min_path f s t P"
    and    f'_def: "f' = augment_edges f γ P"
    and    b'_def: "b' = (λv. if v = fstv (hd P) then b v + γ
                          else if v = sndv (last P) then b v - γ else b v)"
  shows   "is_Opt b' f'"
```

Listing 114. `locale orlins_time` (`Orlins_Time.thy`)

```
locale orlins_time =
 maintain_forest_time where fst = fst +
 send_flow_time where fst = fst +
 orlins where fst = fst
for fst::"'edge ⇒ 'a"+
fixes t_OC::nat and  t_OB::nat
```

Listing 115. `theorem running_time_initial` (`Orlins_Time.thy`)

```
theorem running_time_initial:
  assumes "final = orlins_time t_OC (send_flow initial)"
  shows "prod.fst final + prod.fst (send_flow_time initial) ≤
             (N - 1) * (t_Fuf + t_FC + t_FB + t_SC + t_SB + t_Suf)
          +   (N * (l + k + 2) - 1)* (t_SF + t_SC + t_Suf +
                                      t_Fuf + t_FC + t_OC + t_OB )
          +   ((l + 1) * (2 * N - 1)) *(t_SC + t_SB + t_Suf)
          +   (t_SF + t_SC + t_Suf ) + t_OC

                         ∧ prod.snd final = orlins (send_flow initial)"
```

E Additional Material