Verified Approximation Algorithms

Robin Eßmann[1], Tobias Nipkow[1], and Simon Robillard[2](✉)

[1] Technische Universität München, Munich, Germany
[2] IMT Atlantique, Nantes, France

Abstract. We present the first formal verification of approximation algorithms for NP-complete optimization problems: vertex cover, independent set, load balancing, and bin packing. We uncover incompletenesses in existing proofs and improve the approximation ratio in one case.

1 Introduction

Approximation algorithms for NP-complete problems [12] are a rich area of research untouched by automated verification. We present the first formal verifications of three classical and one lesser known approximation algorithm. Three of these algorithms had been verified on paper by program verification experts [3,4]. We found that their claimed invariants need additional conjuncts before they are strong enough to be real invariants. That is, their proofs are incomplete. The fourth algorithm only comes with a sketchy informal proof.

To put an end to this situation we formalized the correctness proofs of four approximation algorithms for fundamental NP-complete problems in the theorem prover Isabelle/HOL [9,10]. We verified (all proofs are online [6]) that

- the classic approximation algorithm for a minimal vertex cover is a k-approximation algorithm for rank k hypergraphs;
- Wei's algorithm for a maximal independent set [13] is a Δ-approximation algorithm for graphs with maximum degree Δ;
- the greedy algorithm for the load balancing problem is a $\frac{3}{2}$-approximation algorithm if job loads are sorted and a 2-approximation algorithm if job loads are unsorted [8];
- the bin packing algorithm by Berghammer and Reuter [4] is a $\frac{3}{2}$-approximation algorithm.

Isabelle not only helped finding mistakes in pen-and-paper proofs but also encouraged proof refactoring that led to simpler proofs, and in one case, to a stronger result: The invariant given by Berghammer and Müller for Wei's algorithm [3] is sufficient to show that the algorithm has an approximation ratio of $\Delta + 1$. We managed to simplify their argument significantly which lead to an improved approximation ratio of Δ.

All algorithms are expressed in a simple imperative *WHILE*-language. In each case we show that the approximation algorithm computes a valid solution

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12167, pp. 291–306, 2020.
https://doi.org/10.1007/978-3-030-51054-1_17

that is at most a constant factor worse than an optimum solution. The polynomial running time of the approximation algorithm is easy to see in each case.

2 Isabelle/HOL and Imperative Programs

Isabelle/HOL is largely based on standard mathematical notation but with some differences and extensions.

Type variables are denoted by $'a, 'b$, etc. The notation $t :: \tau$ means that term t has type τ. Except for function types $'a \Rightarrow 'b$, type constructors follow postfix syntax, e.g. $'a$ set is the type of sets of elements of type $'a$. Function *some* :: $'a\, set \Rightarrow 'a$ picks an arbitrary element from a set; the result is unspecified if the set is empty.

The types *nat* and *real* represent the sets $\mathbb{N}$ and $\mathbb{R}$. In this paper we drop the coercion function *real* :: *nat* $\Rightarrow$ *real*. The set $\{m..n\}$ is the closed interval $[m, n]$.

The Isabelle/HOL distribution comes with a simple implementation of Hoare logic where programs are annotated with pre- and post-conditions and invariants (all in HOL) as in this example, where all variables are of type *nat*:

$$\{m = 0 \wedge p = 0\}$$
$$WHILE\; m \neq a \;\; INV\; \{p = m * b\} \;\; DO\; p := p + b;\; m := m + 1\; OD$$
$$\{p = a * b\}$$

The box around the program means that it has been verified. All our proofs employ a VCG and essentially reduce to showing the preservation of the invariants.

3 Vertex Cover

We verify the proof in [3] that the classic greedy algorithm for vertex cover is a 2-approximation algorithm. In fact, we generalize the setup from graphs to hypergraphs. A hypergraph is simply a set of edges E, where an edge is a set of vertices of type $'a$. A vertex cover for E is a set of vertices C that intersects with every edge of E:

$$vc :: 'a\; set\; set \Rightarrow 'a\; set \;\Rightarrow\; bool$$
$$vc\; E\; C \;=\; (\forall\, e{\in}E.\; e \cap C \neq \emptyset)$$

A matching ($matching :: 'a\; set\; set \;\Rightarrow bool$) is a set of pairwise disjoint sets. The following is a key property that relates vc and *matching*:

$$finite\; C \;\wedge\; matching\; M \;\wedge\; M \;\subseteq\; E \;\wedge\; vc\; E\; C \;\longrightarrow\; |M| \leq |C|$$

We fix a rank-k hypergraph $E :: 'a\; set\; set$ assuming $\emptyset \notin E$, $finite\; E$ and $e \in E \longrightarrow finite\; e \;\wedge\; |e| \leq k$.

We have verified the well known greedy algorithm that computes a vertex cover C for E. It keeps picking an arbitrary edge that is not covered by C yet until all vertices are covered. The final C has at most k times as many vertices as any vertex cover of E (which is essentially optimal [1]).

$$
\begin{array}{l}
\{True\} \\
C := \emptyset;\; F := E; \\
WHILE\ F \neq \emptyset\ \ INV\ \{invar\ C\ F\} \\
DO\ C := C \cup some\ F;\; F := F - \{e' \in F \mid some\ F\ \cap\ e' \neq \emptyset\}\ OD \\
\{vc\ E\ C\ \wedge (\forall C'.\ finite\ C'\ \wedge vc\ E\ C' \longrightarrow |C| \leq k * |C'|)\}
\end{array}
$$

where *invar* is the following invariant:

$$
\begin{array}{l}
invar :: {'a}\ set\ \Rightarrow\ {'a}\ set\ set\ \Rightarrow\ bool \\
invar\ C\ F\ = \\
(F\ \subseteq\ E\ \wedge\ vc\ (E\ -\ F)\ C\ \wedge\ finite\ C\ \wedge\ (\exists\ M.\ inv\_matching\ C\ F\ M))
\end{array}
$$

$$
\begin{array}{l}
inv\_matching\ C\ F\ M\ = \\
(matching\ M\ \wedge\ M\ \subseteq\ E\ \wedge\ |C|\ \leq\ k\ *\ |M|\ \wedge\ (\forall\ e{\in}M.\ \forall f{\in}F.\ e \cap f\ =\ \emptyset))
\end{array}
$$

The key step in the program proof is that the invariant is invariant:

Lemma 1. $F \neq \emptyset\ \wedge\ invar\ C\ F\ \longrightarrow$
$invar\ (C\ \cup\ some\ F)\ (F - \{e' \in F \mid some\ F\ \cap\ e'\ \neq\ \emptyset\})$

Our invariant is stronger than the one in [3] which lacks $F \subseteq E$. But without $F \subseteq E$ the claimed invariant is not invariant (as acknowledged by Müller-Olm).

4 Independent Set

As in the previous section, a graph is a set of edges. An independent set of a graph E is a subset of its vertices such that no two vertices are adjacent.

$$
\begin{array}{l}
iv :: {'a}\ set\ set\ \Rightarrow\ {'a}\ set\ \Rightarrow\ bool \\
iv\ E\ S\ =\ (S\ \subseteq\ \bigcup\ E\ \wedge\ (\forall\ v_1\ v_2.\ v_1\ \in\ S\ \wedge\ v_2\ \in\ S\ \longrightarrow\ \{v_1,\ v_2\}\ \notin\ E))
\end{array}
$$

We fix a finite graph $E :: {'a}\ set\ set$ such that all edges of E are sets of cardinality 2. The set of vertices $\bigcup E$ is denoted V, and the maximum number of neighbors for any vertex in V is denoted Δ. We show that the greedy algorithm proposed by Wei is a Δ-approximation algorithm. The proof is inspired by one given in [3]. In particular, the proof relies on an auxiliary variable P, which is not needed for the execution of the algorithm, but is used for bookkeeping in the proof. In [3], P is initially a program variable and is later removed from the program and turned into an existentially quantified variable in the invariant. We directly use the latter representation.

$\{ \ True \ \}$
$S := \emptyset; \ X := \emptyset;$
$WHILE \ X \neq V \ INV \ \{ \ \exists P. \ inv\_partition \ S \ X \ P \ \}$
$DO \ x \ := \ some \, (V - X); \ S \ := \ S \cup \{x\}; \ X \ := \ X \cup neighbors \ x \cup \{x\} \ OD$
$\{ \ iv \ E \ S \ \wedge \ (\forall S'. \ iv \ E \ S' \ \longrightarrow \ |S'| \leq |S| * \Delta) \ \}$

To keep the size of definitions manageable, we split the invariant in two. The first part is not concerned with P, but suffices to prove the functional correctness of the algorithm, i.e. that it outputs an independent set of the graph:

$$inv\_iv :: {}'a \ set \ \Rightarrow \ {}'a \ set \ \Rightarrow \ bool$$
$$inv\_iv \ S \ X \ =$$
$$(iv \ E \ S \wedge \ X \subseteq V \wedge \ (\forall v_1 \in V \ - \ X. \ \forall v_2 \in S. \ \{v_1, \ v_2\} \ \notin \ E) \ \wedge \ S \ \subseteq \ X)$$

This invariant is taken almost verbatim from [3], except that in [3] it says that S is an independent set of the subgraph generated by X. This is later used to show that the x picked at each iteration from $V - X$ is not already in S. Defining subgraphs adds unnecessary complexity to the invariant. We simply state $S \subseteq X$, together with the fact that S is an independent set of the whole graph.

We now extend the invariant with properties of the auxiliary variable P.

$$inv\_partition :: {}'a \ set \ \Rightarrow \ {}'a \ set \ \Rightarrow \ {}'a \ set \ set \ \Rightarrow \ bool$$
$$inv\_partition \ S \ X \ P \ =$$
$$(inv\_iv \ S \ X \ \wedge$$
$$\bigcup \ P = X \wedge \ (\forall \, p \in P. \, \exists \, s \in V. \, p = \{s\} \ \cup \ neighbors \ s) \ \wedge \ |P| \ = \ |S| \ \wedge \ finite \ P)$$

We can view the set P as an auxiliary program variable. In order to satisfy the invariant, P would be initially empty and the loop body would include the assignement $P := P \cup \{neighbors \ x \ \cup \ \{x\}\}$. Intuitively, P contains the sets of vertices that are added to X at each iteration (or more precisely, an over-approximation, since some vertices in $neighbors \ x$ may have been added to X in a previous iteration). Instead of adding an unnecessary variable to the program, we only use the existentially quantified invariant. The assignments described above correspond directly to instantiations of the quantifier that are needed to solve proof obligations. This is illustrated with the following lemma, which corresponds to the preservation of the invariant:

Lemma 2. $(\exists \, P. \ inv\_partition \ S \ X \ P) \wedge x \in V - X \longrightarrow$
$(\exists \, P'. \ inv\_partition \ (S \ \cup \ \{x\}) \ (X \ \cup \ neighbors \ x \ \cup \ \{x\}) \ P')$

The existential quantifier in the antecedent yields a witness P. After instantiating the quantifier in the succedent with $P \cup \{neighbors \ x \ \cup \ \{x\}\}$, the goal can be solved straightforwardly. Finally, the following lemma combines the invariant and the negated post-condition to prove the approximation ratio:

Lemma 3. $inv\_partition \ S \ V \ P \ \longrightarrow \ (\forall S'. \ iv \ E \ S' \ \longrightarrow \ |S'| \ \leq \ |S| \ * \ \Delta)$

To prove it, we observe that any set $p \in P$ consists of a vertex x and its neighbors, therefore an independent set S' can contain at most Δ of the vertices in p, thus $|S'| \leq |P| * \Delta$. Furthermore, as indicated by the invariant, $|P| = |S|$.

Compared to the proof in [3], our invariant describes the contents of the set P more precisely, and thus yields a better approximation ratio. In [3], the invariant merely indicates that $X = \bigcup P$, together with two cardinality properties: $\forall p \in P.\ |p| \leq \Delta + 1$ and $|P| \leq |S|$. Taken with the negated post-condition, this invariant can be used to show that for any independent set S', we have $|S'| \leq |S| * (\Delta + 1)$. The proof of this lemma makes use of the following (in)equalities: $|S'| \leq |V|, |V| = |\bigcup P|$, $|\bigcup P| \leq |P| * (\Delta + 1)$ and finally $|P| * (\Delta + 1) \leq |S| * (\Delta + 1)$. Note that this only relies on the trivial fact that an independent set cannot contain more vertices than the graph. By contrast, our own argument takes into account information regarding the edges of the graph.

Although this proof results in a weaker approximation ratio than our own, it yields a useful insight: an approximation ratio is given by the cardinality of the largest set $p \in P$ (i.e., the largest number of vertices added to X during any given iteration). In the worst case, this is equal to $\Delta + 1$, but in practice the number may be smaller. This suggests a variant of the algorithm that stores that value in a variable r, as described in [3]. At every iteration, the variable r is assigned the value $max\ r\ |\{x\} \cup neighbors\ x - X|$. Ultimately, the algorithm returns both the independent set S and the value r, with the guarantee that $|S'| \leq |S| * r$ for any independent set S'.

We also formalized this variant and proved the aforementioned property. The proof follows the idea outlined above, but does away with the variable P entirely: instead, the invariant simply maintains that $inv\_iv\ S\ X \wedge |X| \leq |S| * r$, and the proof of preservation is adapted accordingly. Indeed, this demonstrates that the argument used in [3] does not require an auxiliary variable nor an existentially quantified invariant. For the proof of the approximation ratio Δ, a similar simplification is not as easy to obtain, because the argument relies on a global property of the graph (a constraint that edges impose on independent sets) that is not easy to summarize in an inductive invariant.

So far, we have only considered an algorithm where the vertex x is picked non-deterministically. An obvious heuristic is to pick, at every iteration, the vertex with the smallest number of neighbors among $V - X$. Halldórsson and Radhakrishnan [7] prove that this heuristic achieves an approximation ratio of $(\Delta + 2) / 3$. However their proof is far more complex than the arguments presented here. It is also not given as an inductive invariant, instead relying on case analysis for different types of graphs. This is beyond the scope of our paper.

5 Load Balancing

Our starting point for the load balancing problem is [8, Chapter 11.1]. We need to distribute $n :: nat$ jobs on $m :: nat$ machines with $0 < m$. A job $j \in \{1..n\}$ has a load $t(j) :: nat$. Variables m, n, and t are fixed throughout this section. A solution is described by a function A that maps machines to sets of jobs: $k \in \{1..m\}$ has job j assigned to it iff $j \in A(k)$. The sum of job loads on a machine is given by a function T that is derived from t and A: $(\sum j \in A\ k.\ t\ j) = T\ k$. Predicate lb defines when T and A are a partial solution for $j \leq n$ jobs:

$$
\begin{aligned}
&lb :: (nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat\ set) \Rightarrow nat \Rightarrow bool\\
&lb\ T\ A\ j\ =\\
&((\forall x \in \{1..m\}.\ \forall y \in \{1..m\}.\ x \neq y \longrightarrow A\ x \cap A\ y = \emptyset) \wedge\\
&\ (\textstyle\bigcup_{x \in \{1..m\}} A\ x) = \{1..j\} \wedge (\forall x \in \{1..m\}.\ (\sum y \in A\ x.\ t\ y) = T\ x))
\end{aligned}
$$

It consists of three conjuncts. The first ensures that the sets returned by A are pairwise disjoint, thus, no job appears in more than one machine. The second conjunct ensures that every job $x \in \{1..j\}$ is contained in at least one machine. It also ensures that only jobs $\{1..j\}$ have been added. The final conjunct ensures that T is correctly defined to be the total load on a machine. To ensure that jobs are distributed evenly, we need to consider the machine with maximum load. This load is referred to as the *makespan* of a solution (where f ‘ I is the image of f over I):

$$
\begin{aligned}
&makespan :: (nat \Rightarrow nat) \Rightarrow nat\\
&makespan\ T = Max\ (T\ `\{1..m\})
\end{aligned}
$$

The greedy approximation algorithm outlined in [8] relies on the ability to determine the machine $k \in \{1..m\}$ that has a minimum combined load. As the goal is to approximate the optimum in polynomial time, a linear scan through T suffices to find the machine with minimum load. However, other methods may be considered to further improve time complexity. To determine the machine with minimum load, we will use the following function:

$$
\begin{aligned}
&min_k :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat\\
&min_k\ T\ 0 = 1\\
&min_k\ T\ (x+1) =\\
&(\textsf{let}\ k = min_k\ T\ x\ \textsf{in}\ \textsf{if}\ T\ (x+1) < T\ k\ \textsf{then}\ x+1\ \textsf{else}\ k)
\end{aligned}
$$

We will focus on the approximation factor of $\frac{3}{2}$, which can be proved if the job loads are assumed to be sorted in descending order. The proof for the approximation factor of 2 if jobs are unsorted is very similar and we describe the differences at the end. We say that j jobs are sorted in descending order if *sorted* holds:

$$
\begin{aligned}
&sorted :: nat \Rightarrow bool\\
&sorted\ j = (\forall x \in \{1..j\}.\ \forall y \in \{1..x\}.\ t\ x \leq t\ y)
\end{aligned}
$$

Below we prove the following conditional Hoare triple that expresses the approximation factor and functional correctness of the algorithm given in [8]:

$$\begin{array}{l}
\mathit{sorted}\ n \longrightarrow \\
\{\mathit{True}\} \\
T := (\lambda_.\ 0);\ A := (\lambda_.\ \emptyset);\ j := 0; \\
\mathit{WHILE}\ j < n\ \mathit{INV}\ \{\mathit{inv}_2\ T\ A\ j\} \\
\mathit{DO}\ i := \mathit{min}_k\ T\ m;\ j := j + 1; \\
\quad A := A(i := A(i) \cup \{j\});\ T := T(i := T(i) + t(j)) \\
\mathit{OD} \\
\{\mathit{lb}\ T\ A\ n \wedge \\
\ (\forall T'\ A'.\ \mathit{lb}\ T'\ A'\ n \longrightarrow \mathit{makespan}\ T \leq 3\ /\ 2 * \mathit{makespan}\ T')\}
\end{array}$$

Property *sorted* n is not part of the precondition because it is not influenced by the algorithm and thus there is no need to prove that it remains unchanged. Therefore we made *sorted* n an assumption of the whole Hoare triple. The notation $f(a := b)$ denotes an updated version of function f that maps a to b and behaves like f otherwise. Thus an assignment $f := f(i := b)$ is nothing but the conventional imperative array update notation $f[i] := b$.

Functional correctness follows because each iteration extends a partial solution for j jobs to one for $j + 1$ jobs:

Lemma 4. $\mathit{lb}\ T\ A\ j \wedge x \in \{1..m\} \longrightarrow$
$\mathit{lb}\ (T(x := T\ x + t\ (j\ + 1)))\ (A(x\ :=\ A\ x \cup \{j\ + 1\}))\ (j\ + 1)$

Moreover, it is easy to see that the initialization establishes $\mathit{lb}\ T\ A\ j$.

To prove the approximation factor in both the sorted and unsorted case, the following lower bound is important:

Lemma 5. $\mathit{lb}\ T\ A\ j \longrightarrow (\sum_{x=1}^{j}\ t\ x)\ /\ m \leq \mathit{makespan}\ T$

This is a result of $\sum_{x=1}^{m} T(x) = \sum_{x=1}^{j} t(x)$ together with this general property of sums: $\mathit{finite}\ A \wedge A \neq \emptyset \longrightarrow (\sum a{\in}A.\ f\ a) \leq |A| * \mathit{Max}\ (f\ `\ A)$.

A similar observation applies to individual jobs. Any job must be a lower bound on some machine, as it is assigned to one and, by extension, it must also be a lower bound of the makespan:

Lemma 6. $\mathit{lb}\ T\ A\ j \longrightarrow \mathit{Max}_0\ (t`\{1..j\}) \leq \mathit{makespan}\ T$

As any job load is a lower bound on the makespan over the machines, the job with maximum load must also be such a lower bound. Note that Max_0 returns 0 for the empty set.

When jobs are sorted in descending order, a stricter lower bound for an individual job can be established. We observe that an added job is at most as large as the jobs preceding it. Therefore, if a machine contains at least two jobs, this added job is only at most *half* as large as the makespan. We can use this observation by assuming the machines to be filled with more than m jobs, as this will ensure that some machine must contain at least two jobs.

Lemma 7. $\mathit{lb}\ T\ A\ j \wedge m\ <\ j \wedge \mathit{sorted}\ j \longrightarrow 2 * t\ j \leq \mathit{makespan}\ T$

Note that this lower bound only holds if there are strictly more jobs than machines. One must, however, also consider how the algorithm behaves in the other case. One may intuitively see that the algorithm will be able to distribute the jobs such that every machine will only have at most one job assigned to it, making the algorithm trivially optimal. To prove this, we need to show the following behavior of min_k:

Lemma 8.
1. $x \in \{1..m\} \wedge T\ x = 0 \longrightarrow T\ (min_k\ T\ m) = 0$
2. $x \in \{1..m\} \wedge T\ x = 0 \longrightarrow min_k\ T\ m \leq x$

They can be shown by induction on the number of machines m.

As the proof in [8] is only informal, Kleinberg and Tardos do not provide any loop invariant. We propose the following invariant for sorted jobs:

$$
\begin{aligned}
&inv_2 :: (nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat\ set) \Rightarrow nat \Rightarrow bool\\
&inv_2\ T\ A\ j =\\
&(lb\ T\ A\ j \wedge j \leq n \wedge\\
&\ (\forall T'\ A'.\ lb\ T'\ A'\ j \longrightarrow makespan\ T \leq 3\ /\ 2 * makespan\ T') \wedge\\
&\ (\forall x > j.\ T\ x = 0) \wedge (j \leq m \longrightarrow makespan\ T = Max_0\ (t\ `\ \{1..j\})))
\end{aligned}
$$

The final two conjuncts relate to the trivially optimal behavior of the algorithm if $j \leq m$. The penultimate conjunct shows that only as many machines can be occupied as there are available jobs. The final conjunct ensures that every job is distributed on its own machine, making the makespan equivalent to the job with maximum load.

It should be noted that if the makespan is sufficiently large, an added job may not increase the makespan at all, as the machine with minimum load combined with the job may not exceed the previous makespan. As such, we will also consider the possibility that an added job can simply be ignored without affecting the overall makespan.

Lemma 9. $makespan\ (T(x := T\ x + y)) \neq T\ x + y \longrightarrow$
$makespan\ (T(x := T\ x + y)) = makespan\ T$

To make use of this observation, we need to be able to relate the makespan of a solution with the added job to the makespan of a solution without it. One can easily show the following by removing $j + 1$ from the solution:

Lemma 10. $lb\ T\ A\ (j + 1) \longrightarrow$
$(\exists T'\ A'.\ lb\ T'\ A'\ j \wedge makespan\ T' \leq makespan\ T)$

We can now prove the preservation of inv_2. Let $i = min_k\ T\ m$ be the machine with minimum load. We define:

$$T_g := T\ (i := T(i) + t(j + 1)) \qquad A_g := A\ (i := A(i) \cup \{j + 1\})$$

We begin with a case distinction. If $j+1 \leq m$, we can make use of the additional conjuncts to prove the trivially optimal behavior. We first note *in-range*: $j + 1 \in \{1..m\}$. Moreover, from the penultimate conjunct, $T(j + 1) = 0$. Combining this with Lemma 8.1, we can see that $T(i) = 0$. Therefore $T_g(i) = t(j{+}1)$ and with the final conjunct of the assumed invariant, the makespan of T_g remains equivalent to the job with maximum load. To prove that the penultimate conjunct is preserved, we again use *in-range*, $T(j + 1) = 0$, and Lemma 8.2 to prove that $i \le j + 1$. Moreover, T_g only differs from T by the modification of machine i. Thus, the penultimate conjunct for $j + 1$ jobs is preserved as well. From Lemma 6 we can then see that, as the makespan of T_g is equivalent to the job with maximum load, it must be trivially optimal. Functional correctness can be shown using Lemma 4, and proving the preservation of the remaining conjunct is trivial. We now come to the case $j+1 > m$. We first show that the penultimate conjunct is preserved (the final conjunct can be ignored, as $\neg\, j+1 \le m$). This follows from the correctness of min_k, as the index returned by it has to be in $\{1..m\}$ as long as $m > 0$. Therefore, we can simply show this from the penultimate conjunct of the assumed invariant. We now come to the proof of the approximation factor:

$$\forall T'\ A'.\ lb\ T'\ A'\ (j+1) \longrightarrow makespan\ T_g \le 3\ /\ 2 * makespan\ T'$$

To prove it, we fix T_1 and A_1 such that $lb\ T_1\ A_1\ (j+1)$. Using Lemma 10, one can now obtain T_0 and A_0 such that $lb\ T_0\ A_0\ j$ and MK: $makespan\ T_0 \le makespan\ T_1$. From the assumed loop invariant, we can now show:

$$\begin{aligned} makespan\ T &\le \frac{3}{2} makespan\ T_0 && \text{by } inv_2\text{-def} \\ &\le \frac{3}{2} makespan\ T_1 && \text{by } MK \end{aligned}$$

To prove the makespan for $j + 1$ jobs, there are now two cases to consider: The added job $j + 1$ contributes to the makespan or it does not. The case in which it does not can be shown by combining the previous calculation with Lemma 9. For the first case, we may then assume that $makespan\ T_g = T(i) + t(j + 1)$. Like in Lemma 5, we note that *sum-eq*: $(\sum_{x=1}^{m}\ T\ x) = (\sum_{x=1}^{j}\ t\ x)$. Moreover, *min-avg*: $m * T\ (min_k\ T\ m) \le (\sum_{i=1}^{m}\ T\ i)$. This allows us to calculate the following lower bound for $T(i)$:

$$\begin{aligned} & m * T(i) \le \sum_{i=1}^{m} T(i) = \sum_{i=1}^{j} t(i) && \text{by } min\text{-}avg \text{ and } sum\text{-}eq \\ \Longleftrightarrow\ & T(i) \le \frac{\sum_{i=1}^{j} t(i)}{m} && \text{because } m > 0 \\ & \le makespan\ T_0 \le makespan\ T_1 && \text{by Lemma 5 and } MK \end{aligned}$$

From Lemma 7 we can also show that $t(j + 1)$ is a lower bound for $\frac{1}{2}$ of the makespan of T_1. Therefore:

$$\begin{aligned} makespan\ T_g = T(i) + t(j+1) &\le makespan\ T_1 + \frac{makespan\ T_1}{2} \\ &= \frac{3}{2} makespan\ T_1 \end{aligned}$$

The proof of functional correctness and remaining conjuncts is again trivial.

Let us now consider the unsorted case where one can still show an approximation factor of 2. The algorithm is identical but the invariant is simpler:

$$\mathit{inv}_1\ T\ A\ j =$$
$$(\mathit{lb}\ T\ A\ j \land j \leq n \land (\forall\, T'\ A'.\ \mathit{lb}\ T'\ A'\ j \longrightarrow \mathit{makespan}\ T \leq 2 * \mathit{makespan}\ T'))$$

The proof for this invariant is a simpler version of the proof above: We do not need the initial case distinction (case $j+1 \leq m$ need not be considered separately) and to prove the approximation factor we use Lemma 6 instead of Lemma 7 to obtain a bound for $t(j+1)$.

6 Bin Packing

We finally consider the linear time $\frac{3}{2}$-approximation algorithm for the bin packing problem proposed by Berghammer and Reuter [4]. The bin packing problem is similar to the load balancing problem described in the previous section. The main distinction is that in the load balancing problem, the number of machines is fixed, while the load a single machine can hold is unbounded. With the bin packing problem, this is essentially reversed. The maximum capacity a single bin can hold is limited by some fixed c. However, we are free to use as many bins as necessary to achieve a solution. The goal is now to minimize this number of bins used instead of the maximum capacity of a bin.

For the bin packing problem we are given a finite, non-empty set of objects $U :: {}'a\ \mathit{set}$, whose *weights* are given by a function $w :: {}'a \Rightarrow \mathit{real}$. Note that in this paper *nat*s are implicitly converted to *real*s if needed. The weight of an object in U is strictly greater than zero, but bounded by a maximum capacity $c :: \mathit{nat}$. The abbreviation $W(B) \equiv \sum_{u \in B} w(u)$ denotes the weight of a bin $B \subseteq U$. The set U can also be separated into *small* and *large* objects. An object u is considered small if $w(u) \leq \frac{c}{2}$. An object is large if the opposite is the case. We will begin by assuming that all small objects in U can be found in a set S, and large objects in U can be found in a set L, such that $S \cup L = U$ and $S \cap L = \emptyset$. Of course L and S can also be computed from U in linear time. Variables U, w, c, L, and S are fixed throughout this section.

A solution P to the bin packing problem is then defined as follows:

$$\mathit{bp} :: {}'a\ \mathit{set}\ \mathit{set} \Rightarrow \mathit{bool}$$
$$\mathit{bp}\ P = (\mathit{partition\_on}\ U\ P \land (\forall\, B \in P.\ W\ B \leq c))$$

P contains all the bins necessary such that it is a correct partition of U. To check for this, we use a function $\mathit{partition\_on} :: {}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set}\ \mathit{set} \Rightarrow \mathit{bool}$ which can be found in the Isabelle HOL-Library. We add the final conjunct to ensure that no bin $B \in P$ exceeds the maximum capacity c.

The idea behind the algorithm proposed by Berghammer and Reuter is to split the solution P into two partial solutions P_1 and P_2. At every step of the algorithm we consider two bins B_1 and B_2 which we try to fill with remaining objects from $V \subseteq U$ that have not been assigned yet. If adding the object to

B_1 or B_2 would cause it to exceed its maximum capacity, the bin is moved into the partial solution P_1 or P_2 respectively and cleared. Once there are no small objects left, the solution is the union of the partial solutions P_1 and P_2, the bins B_1 and B_2 (if they still contain objects), and the remaining large objects, which each receive their own bin, as no two large objects can fit into a single bin. To ensure that no empty bins are added to the solution, we define:

$$[\![\cdot]\!] :: {'a\ set} \Rightarrow {'a\ set\ set}$$
$$[\![B]\!] = (\textit{if } B = \emptyset \textit{ then } \emptyset \textit{ else } \{B\})$$

The final union can now be written as $P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!] \cup \{\{v\} \mid v \in V\}$ where V contains the remaining large elements. The algorithm can be specified by the following Hoare triple:

$$\{True\}$$
$$P_1 := \emptyset;\ P_2 := \emptyset;\ B_1 := \emptyset;\ B_2 := \emptyset;\ V := U;$$
$$WHILE\ V \cap S \neq \emptyset\ INV\ \{inv_3\ P_1\ P_2\ B_1\ B_2\ V\}\ DO$$
$$IF\ B_1 \neq \emptyset\ THEN\ u := some\ (V \cap S)$$
$$ELSE\ IF\ V \cap L \neq \emptyset\ THEN\ u := some\ (V \cap L)$$
$$\quad ELSE\ u := some\ (V \cap S)\ FI\ FI;$$
$$V := V - \{u\};$$
$$IF\ W(B_1) + w(u) \leq c\ THEN\ B_1 := B_1 \cup \{u\}$$
$$ELSE\ IF\ W(B_2) + w(u) \leq c\ THEN\ B_2 := B_2 \cup \{u\}$$
$$\quad ELSE\ P_2 := P_2 \cup [\![B_2]\!];\ B_2 := \{u\}\ FI;$$
$$\quad P_1 := P_1 \cup [\![B_1]\!];\ B_1 := \emptyset\ FI$$
$$OD;$$
$$P := P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!] \cup \{\{v\} \mid v \in V\}$$
$$\{bp\ P \wedge (\forall Q.\ bp\ Q \longrightarrow |P| \leq 3\ /\ 2\ *\ |Q|)\}$$

Berghammer and Reuter prove functional correctness using a simplified version of this algorithm where an arbitrary element of V is assigned to u. This allows for fewer case distinctions, as the first $IF{-}THEN{-}ELSE$ block can be ignored. One needs to find a loop invariant that implies functional correctness and prove that it is preserved in the following cases:

Case 1. The object fits into B_1:

$$inv_1\ P_1\ P_2\ B_1\ B_2\ V \wedge u \in V \wedge W\ B_1 + w\ u \leq c \longrightarrow$$
$$inv_1\ P_1\ P_2\ (B_1 \cup \{u\})\ B_2\ (V - \{u\})$$

Case 2. The object fits into B_2:

$$inv_1\ P_1\ P_2\ B_1\ B_2\ V \wedge u \in V \wedge W\ B_2 + w\ u \leq c \longrightarrow$$
$$inv_1\ (P_1 \cup [\![B_1]\!])\ P_2\ \emptyset\ (B_2 \cup \{u\})\ (V - \{u\})$$

Case 3. The object fits into neither bin:

$$inv_1\ P_1\ P_2\ B_1\ B_2\ V \wedge u \in V \longrightarrow$$
$$inv_1\ (P_1 \cup [\![B_1]\!])\ (P_2 \cup [\![B_2]\!])\ \emptyset\ \{u\}\ (V - \{u\})$$

Berghammer and Reuter [4] define the following predicate as their loop invariant:

$$inv_1\ P_1\ P_2\ B_1\ B_2\ V\ =\ bp\ (P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!] \cup \{\{v\} \mid v \in V\})$$

As it turns out, this invariant is too weak. Assume $inv_1\ P_1\ P_2\ B_1\ B_2\ V$. Suppose P_1 (alternatively P_2) already contains the non-empty bin B_1. Note that this does not violate the invariant because $P_1 \cup [\![B_1]\!] = P_1$. Now, if the algorithm modifies B_1 by adding an element from V such that B_1 becomes some $B_1{}'$ then $B_1 \cap B_1{}' \neq \emptyset$ and $B_1 \in P_1$, i.e., $B_1{}'$ is no longer disjoint from the elements of P. The same issue arises with the added object $u \in V$, if $\{u\}$ is already in P_1 or P_2. To account for such cases, we will require additional conjuncts:

$$\begin{aligned}
&inv_1 :: {}'a\ set\ set\ \Rightarrow\ {}'a\ set\ set\ \Rightarrow\ {}'a\ set\ \Rightarrow\ {}'a\ set\ \Rightarrow\ {}'a\ set\ \Rightarrow\ bool\\
&inv_1\ P_1\ P_2\ B_1\ B_2\ V\ =\\
&(bp\ (P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!] \cup \{\{v\} \mid v \in V\}) \wedge\\
&\ \bigcup\ (P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!])\ =\ U - V\ \wedge\\
&\ B_1 \notin P_1 \cup P_2 \cup [\![B_2]\!]\ \wedge\\
&\ B_2 \notin P_1 \cup [\![B_1]\!] \cup\ P_2\ \wedge\\
&\ (P_1 \cup [\![B_1]\!]) \cap (P_2 \cup [\![B_2]\!])\ =\ \emptyset)
\end{aligned}$$

There are different ways to strengthen the original inv_1. We use the above additional conjuncts as they can be inserted in existing proofs with little modification, and their preservation in the invariant can be proved quite trivially. The first additional conjunct ensures that no element still in V is already in a bin or partial solution. The second and third additional conjuncts ensure distinctness of the bins B_1 and B_2 with the remaining solution. The final conjunct ensures that the partial solutions with their added bins are disjoint from each other. It should be noted that the last conjunct is *not* necessary to prove functional correctness. It will, however, be needed in later proofs, and as its preservation in this invariant for the simplified algorithm can be used in the proof of the full algorithm, one can save redundant case distinctions by proving it now. Another advantage of proving it now is that later invariants can remain identical to the invariants proposed in the paper.

We now prove the preservation of inv_1 in all three cases. As we assume the invariant to hold before the execution of the loop body, we can see from the first additional conjunct $\bigcup\ (P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!]) = U - V$ and the assumption $u \in V$ that *not-in:* $\forall B \in P_1 \cup [\![B_1]\!] \cup\ P_2 \cup [\![B_2]\!].\ u \notin B$ holds. This will be needed for all three cases. Now, we can begin with Case 1. We first show

$$bp\ (P_1 \cup [\![B_1 \cup \{u\}]\!] \cup P_2 \cup [\![B_2]\!] \cup \{\{v\} \mid v \in V\ -\ \{u\}\})$$

One can see that this union does not contain the empty set. The object u is now moved from a singleton set into B_1. Therefore, the union of all bins will again return U. To show that this union remains pairwise disjoint, we can use *not-in* and the second additional conjunct of inv_1 to show that u is not yet contained in the partial solution and B_1 is *distinct* from any other bin. Therefore, combined with the assumption that the union was pairwise disjoint before the modification, the union remains pairwise disjoint. To prove the preservation of the second conjunct of *bp*, we need to show that the bin weights do not exceed their maximum capacity c. The only bin that was changed in this step is B_1, which has increased its weight by $w(u)$. As we are in Case 1, we can assume that u fits into B_1, $W(B_1) + w(u) \leq c$. Therefore, this conjunct holds as well. Now, one only needs to show that the additional conjuncts are preserved. For the first additional conjunct, we can again use *not-in* to show:

$$\begin{aligned} & \bigcup (P_1 \cup [\![B_1 \cup \{u\}]\!] \cup P_2 \cup [\![B_2]\!]) = U - (V - \{u\}) & \\ \Longleftrightarrow \quad & \bigcup (P_1 \cup [\![B_1]\!] \cup P_2 \ \cup \ [\![B_2]\!]) \cup \{u\} = U - (V - \{u\}) & \text{by } \textit{not-in} \\ \Longleftrightarrow \quad & \bigcup (P_1 \cup [\![B_1]\!] \cup \ P_2 \cup [\![B_2]\!]) \cup \{u\} = U - V \ \cup \ \{u\} & \text{by } u \in U \end{aligned}$$

Using the first additional conjunct of the assumed invariant, one can see that this must hold. The remaining conjuncts

$$\begin{aligned} & B_1 \cup \{u\} \notin P_1 \cup P_2 \cup \ [\![B_2]\!] \\ & B_2 \notin P_1 \cup [\![B_1 \cup \{u\}]\!] \cup P_2 \\ & (P_1 \cup [\![B_1 \cup \{u\}]\!]) \cap (P_2 \cup [\![B_2]\!]) \ = \ \emptyset \end{aligned}$$

can be automatically proved in Isabelle using *not-in* and the assumption that the conjuncts of $inv_1\ P_1\ P_2\ B_1\ B_2\ V$ held before the modification. The proof for Case 2 is almost identical to that of Case 1. The main difference is that the focus now lies on B_2 and the fact that B_1 is now emptied and the previous contents added to the partial solution P_1. One therefore has to show that

$$bp\ (P_1 \cup [\![B_1]\!] \cup \ [\![\emptyset]\!] \cup P_2 \cup [\![B_2 \cup \{u\}]\!] \cup \{\{v\} \mid v \in V - \{u\}\})$$

holds. As $[\![\emptyset]\!]$ can be ignored, one can see that the act of emptying B_1 and adding it to the partial solution will not otherwise affect the proof. The proof of *bp* in Case 3 is trivial, as the modifications made in this step can simply be undone by applying the following steps:

$$\begin{aligned} & P_1 \cup \ [\![B_1]\!] \ \cup \ [\![\emptyset]\!] \cup (P_2 \cup [\![B_2]\!]) \cup [\![\{u\}]\!] \cup \{\{v\} \mid v \in V - \{u\}\} & \\ & = P_1 \cup [\![B_1]\!] \cup \ P_2 \cup \ [\![B_2]\!] \cup \{\{u\}\} \cup \{\{v\} \mid v \in V - \{u\}\} & \text{by } [\![\cdot]\!]{-}def \\ & = P_1 \cup [\![B_1]\!] \cup P_2 \cup [\![B_2]\!] \cup \{\{v\} \mid v \in V\} & \text{by } u \in V \end{aligned}$$

Now, one only needs to show that the remaining additional conjuncts hold. This can again be shown automatically using *not-in* and the fact that $inv_1\ P_1\ P_2\ B_1\ B_2\ V$ held before the modifications. Therefore, inv_1 is preserved in all three cases.

To prove the approximation factor, we proceed as in [4] and establish suitable lower bounds. The first lower bound

Lemma 11. $bp\ P \ \longrightarrow \ |L| \ \leq \ |P|$

holds because a bin can only contain at most one large object, and every large object needs to be in the solution. To prove this in Isabelle, we first make the observation that for every large object there exists a bin in P in which it is contained. Therefore, we may obtain a function f that returns this bin for every $u \in L$. Using the fact that any bin can hold at most one large object, we can show that this function has to be injective, as every large object must map to a unique bin. Hence, the number of large objects is equal to the number of bins f maps to. Moreover, the image of f has to be a subset of P. Thus, the number of large objects has to be a lower bound on the number of bins in P.

As it turns out, the algorithm will ensure that there is always at least one large object in a bin for the first partial solution as long as large objects are available. This means we can assume that:

$$V \cap L \neq \emptyset \longrightarrow (\forall\, B \in P_1 \cup [\![B_1]\!].\ B \cap L \neq \emptyset)$$

Therefore, we can use the previous lower bound to show the following:

Lemma 12. $bp\ P \wedge inv_1\ P_1\ P_2\ B_1\ B_2\ V \wedge (\forall B \in P_1 \cup [\![B_1]\!].\ B \cap L \neq \emptyset) \longrightarrow |P_1 \cup [\![B_1]\!] \cup \{\{v\} \mid v \in V \cap L\}| \leq |P|$

Another easy lower bound is this one:

Lemma 13. $bp\ P \longrightarrow (\sum_{u \in U} w\, u) \leq c\ *\ |P|$

The next lower bound arises from the fact that an object is only ever put into B_2, and therefore P_2, if it would have caused B_1 to overflow. As a result of this, we can define a bijective function f that maps every bin of P_1 to the object in $P_2 \cup [\![B_2]\!]$ that would have caused the bin to overflow. We define:

$$bij\_exists :: {'a}\ set\ set \Rightarrow {'a}\ set \Rightarrow bool$$
$$bij\_exists\ P\ V = (\exists f.\ bij\_betw\ f\ P\ V \wedge (\forall B \in P.\ c < W\ B + w\ (f\ B)))$$

From this, we can make the observation that the number of bins in P_1 is a *strict* lower bound on the number of bins of any correct bin packing P:

Lemma 14. $bp\ P \wedge inv_1\ P_1\ P_2\ B_1\ B_2\ V \wedge bij\_exists\ P_1\ (\bigcup (P_2 \cup [\![B_2]\!])) \longrightarrow |P_1| + 1 \leq |P|$

Unlike the proof outlined in [4], we begin with a case distinction on P_1. The reasoning behind this is that if P_1 is empty, the strict nature of the lower bound cannot be shown from the calculation that Berghammer and Reuter make. Therefore, we consider the case where P_1 is empty separately. If P_1 is empty, our goal is to prove that 1 is a lower bound on the number of bins in P. This follows from the fact that U is non-empty, and therefore any correct bin packing must contain at least one bin. For the other case, we may now assume that P_1 is non-empty. In the following proof, we will need the final conjunct of inv_1, $(P_1 \cup [\![B_1]\!]) \cap (P_2 \cup [\![B_2]\!]) = \emptyset$, which we can transform into *disjoint*: $P_1 \cap (P_2 \cup [\![B_2]\!]) = \emptyset$. We also obtain the bijective function f and observe that, as the object obtained from f for a bin $B \in P_1$ caused B to exceed its capacity, *exceed*: $c < W(B) + w(f(B))$ must hold. We can now perform the following calculation:

$$\begin{aligned}
c|P_1| &= \sum_{B \in P_1} c \\
&< \sum_{B \in P_1} W(B) + \sum_{B \in P_1} w(f(B)) && \text{by } P_1 \neq \emptyset \text{ and } \textit{exceed} \\
&= \sum_{B \in P_1} W(B) + \sum_{B \in P_2 \cup [\![B_2]\!]} W(B) && \text{by } f \text{ bijective} \\
&= \sum_{B \in P_1 \cup\ P_2 \cup\ [\![B_2]\!]} W(B) && \text{by } \textit{disjoint} \\
&\leq \sum_{u \in U} w(u) \leq c|P| && \text{by } \textit{inv}_1 \text{ and Lemma 13}
\end{aligned}$$

Therefore $|P_1| < |P|$ and, by extension, $|P_1| + 1 \leq |P|$.

We only sketch the rest of the proof because it is almost identical to that in [4]. First we need two extensions of inv_1 to show the approximation ratio:

$inv_2\ P_1\ P_2\ B_1\ B_2\ V\ =$
$(inv_1\ P_1\ P_2\ B_1\ B_2\ V\ \wedge$
$(V \cap L \neq \emptyset\ \longrightarrow\ (\forall\ B \in P_1 \cup [\![B_1]\!].\ B \cap L \neq \emptyset)) \wedge$
$bij\_exists\ P_1\ (\bigcup\ (P_2 \cup [\![B_2]\!])) \wedge 2 * |P_2| \leq |\bigcup\ P_2|)$
$inv_3\ P_1\ P_2\ B_1\ B_2\ V\ =\ (inv_2\ P_1\ P_2\ B_1\ B_2\ V\ \wedge\ B_2 \subseteq\ S)$

The motivation for the last conjunct in inv_2 is the following lower bound:

$inv_1\ P_1\ P_2\ B_1\ B_2\ V\ \wedge\ 2\ *|P_2| \leq |\bigcup\ P_2|\ \wedge\ bij\_exists\ P_1\ (\bigcup\ (P_2\ \cup\ [\![B_2]\!]))\ \longrightarrow$
$2 * |P_2\ \cup\ [\![B_2]\!]| \leq |P_1|\ +\ 1.$

The main lower bound lemma (Theorem 4.1 in [4]) is the following:

Lemma 15. $V \cap S = \emptyset \wedge inv_2\ P_1\ P_2\ B_1\ B_2\ V \wedge bp\ P\ \longrightarrow$
$|P_1 \cup [\![B_1]\!] \cup\ P_2 \cup\ [\![B_2]\!] \cup \{\{v\}\ |\ v \in V\}|\ \leq\ 3\ /\ 2\ *\ |P|$

From this lower bound the postcondition of the algorithm follows easily under the assumption that inv_2 holds at the end of the loop. This in turn follows because inv_3 can be shown to be a loop invariant.

7 Conclusion

In the first application of theorem proving to approximation algorithms we have verified three classical and one less well-known approximation algorithm for fundamental NP-complete problems, have corrected purported invariants from the literature and could even strengthen the approximation ratio in one case. Although we have demonstrated the benefits of formal verification of approximation algorithms, we have only scratched the surface of this rich theory. The next step is to explore the subject more systematically. As a large fraction of the theory of approximation algorithms is based on linear programming, this is a promising and challenging direction to explore. Some linear programming theory has been formalized in Isabelle already [5,11]. Approximation algorithms can also be formulated as relational programs, and verified accordingly. This approach was explored in [2], with some support from theorem provers, but has yet to be fully formalized.

Acknowledgement. Tobias Nipkow is supported by DFG grant NI 491/16-1.

References

1. Bansal, N., Khot, S.: Inapproximability of hypergraph vertex cover and applications to scheduling problems. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 250–261. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14165-2_22
2. Berghammer, R., Höfner, P., Stucke, I.: Cardinality of relations and relational approximation algorithms. J. Log. Algebraic Methods Program. **85**(2), 269–286 (2016)
3. Berghammer, R., Müller-Olm, M.: Formal development and verification of approximation algorithms using auxiliary variables. In: Bruynooghe, M. (ed.) LOPSTR 2003. LNCS, vol. 3018, pp. 59–74. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25938-1_6
4. Berghammer, R., Reuter, F.: A linear approximation algorithm for bin packing with absolute approximation factor 3/2. Sci. Comput. Program. **48**(1), 67–80 (2003). https://doi.org/10.1016/S0167-6423(03)00011-X
5. Bottesch, R., Haslbeck, M.W., Thiemann, R.: Verifying an incremental theory solver for linear arithmetic in Isabelle/HOL. In: Herzig, A., Popescu, A. (eds.) FroCoS 2019. LNCS (LNAI), vol. 11715, pp. 223–239. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29007-8_13
6. Eßmann, R., Nipkow, T., Robillard, S.: Verified approximation algorithms. Archive of Formal Proofs, Formal proof development, January 2020. http://isa-afp.org/entries/Approximation_Algorithms.html
7. Halldórsson, M.M., Radhakrishnan, J.: Greed is good: approximating independent sets in sparse and bounded-degree graphs. Algorithmica **18**(1), 145–163 (1997)
8. Kleinberg, J.M., Tardos, É.: Algorithm Design. Addison-Wesley (2006)
9. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer, Heidelberg (2014). http://concrete-semantics.org
10. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
11. Parsert, J., Kaliszyk, C.: Linear programming. Archive of Formal Proofs, Formal proof development, August 2019. http://isa-afp.org/entries/Linear_Programming.html
12. Vazirani, V.V.: Open problems. Approximation Algorithms, pp. 334–343. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-662-04565-7_30
13. Wei, V.: A lower bound for the stability number of a simple graph. Technical Memorandum 81–11217-9, Bell Laboratories (1981)