UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Verifying Dijkstra's Algorithm with KeY

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von
Volker Klasen

Erstgutachter: Prof. Dr. Bernhard Beckert
Institut für Theoretische Informatik, Karlsruher Institut für Technologie

Zweitgutachter: Prof. Dr. Ulrich Furbach
Institut für Informatik, AG Künstliche Intelligenz, Universität Koblenz

Koblenz, im März 2010

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Deutsche Zusammenfassung

Die Entwicklung von Algorithmen im Sinne des *Algorithm Engineering* geschieht zyklisch. Der entworfene Algorithmus wird theoretisch analysiert und anschließend implementiert. Nach der praktischen Evaluierung wird der Entwurf anhand der gewonnenen Kenntnisse weiter entwickelt. Formale Verifizierung der Implementation neben der praktischen Evaluierung kann den Entwicklungsprozess verbessern.

Mit der Java Modeling Language (JML) und dem KeY tool stehen eine einfache Spezifikationssprache und ein benutzerfreundliches, automatisiertes Verifikationstool zur Verfügung. Diese Arbeit untersucht, inwieweit das KeY tool für die Verifizierung von komplexeren Algorithmen geeignet ist und welche Rückmeldungen für Algorithmiker aus der Verifikation gewonnen werden können.

Die Untersuchung geschieht anhand von Dijkstras Algorithmus zur Berechnung von kürzesten Wegen in einem Graphen. Es sollen eine konkrete Implementation des Standard-Algorithmus und anschließend Implementationen weiterer Varianten verifiziert werden. Dies ahmt den Entwicklungsprozess des Algorithmus nach, um in jeder Iteration nach möglichen Rückmeldungen zu suchen.

Bei der Verifizierung der konkreten Implementation merken wir, dass es nötig ist, zuerst eine abstraktere Implementation mit einfacheren Datenstrukturen zu verifizieren. Mit den dort gewonnenen Kenntnissen können wir dann die Verifikation der konkreten Implementation fortführen. Auch die Varianten des Algorithmus können dank der vorangehenden Verifikationen verifiziert werden.

Die Komplexität von Dijkstras Algorithmus bereitet dem KeY tool einige Schwierigkeiten bezüglich der Performanz, weswegen wir während der Verifizierung die Automatisierung etwas reduzieren müssen. Auf der anderen Seite zeigt sich, dass sich aus der Verifikation einige Rückmeldungen ableiten lassen.

Contents

1 Introduction

The DFG Priority Programme Algorithm Engineering[1] focuses on a four stage cycle in the development of algorithms, consisting of design, analysis, implementation and experiment. Following the design stage, the (abstract) algorithm is formally analyzed and then implemented. The resulting concrete implementation is examined experimentally and with the obtained knowledge the cycle is started over again. The inclusion of a new stage, the formal analysis of the concrete implementation, which runs in parallel to the experimental stage, can improve the development process.

In particular, formal verification (as part of the formal analysis) of the concrete implementation can help the development process in several ways. On the one hand, the verification of a highly optimized implementation is less difficult with the knowledge gained by the iterative verification of previous cycles. On the other hand, the verification can point out spots where the algorithm can be further improved.

Because the formal verification is part of the development process, algorithm engineers, who are in general no experts in formal verification, should be able to perform the verification as well. Therefore, we are in need of tools that are highly automated and provide proper feedback if a proof attempt does not succeed. Also, those tools have to support a formal specification language which is easy to use.

The Java Modeling Language (JML)[2] [LBR06] is a specification language for Java. It uses a Java-like syntax and can be written in Java source files, which makes it easy to use. We will take a more detailed look at JML in section 1.1.

A tool for the verification of Java programs that can be used in conjunction with JML is the KeY tool [BHS07] (from now on called KeY). It is a highly automated interactive theorem prover for first-order Dynamic Logic for Java. JML specifications can be used to produce proof obligations to show the correctness of Java programs. More details can be found in section 1.2.

We want to determine, whether KeY is suitable for verifying algorithms as they occur in algorithm engineering, and what feedback can be given to algorithm engineers by evaluating the verification. For this, we will mimic the development process of a chosen algorithm by verifying implementations of several variants of that algorithm. We will use JML for the specification and KeY to verify the implementations.

The algorithm should already be well-known, so that we can concentrate more on finding a proof with KeY than on finding the right specification for the algorithm. We chose Dijkstra's Shortest Path Algorithm, which is used

[1] `http://www.algorithm-engineering.de/`
[2] `http://www.jmlspecs.org/`

in research and teaching in Prof. Dr. Dorothea Wagner's group Algorithmics I at the Karlsruhe Institute of Technology[3]. It is available in several variants, so that we can mimic the development process.

We start with the verification of a concrete implementation of Dijkstra's Algorithm and continue with some variants later on. Besides the evaluation of the verification with KeY, we also explore possible feedback for algorithm engineers.

In the following sections we will give a short introduction of the specification language JML, the theorem prover KeY and Dijkstra's Algorithm. Readers who are familiar with these topics may skip the remainder of this section.

1.1 Java Modeling Language

The Java Modeling Language (JML) is a formal behavioral interface specification language tailored to Java. It allows to specify the behavior of Java programs using pre- and postconditions and invariants.

The behavior of a method is given by a method contract consisting of preconditions, postconditions and a modifies clause. A method is correct with respect to its specification if after its execution the postconditions hold, provided it was called while the preconditions were true. Additionally, the method may only modify memory locations referenced in the modifies clause, except for local variables. It is permitted that a method has more than one contract. While multiple contracts are independent of each other, the method must fulfill each of them.

Properties of fields in a Java class can be specified by class invariants. All constructors must put these invariants in place and every non-helper method, i.e. every non-private method, has to preserve them. Only during the execution of a method the invariants can be temporarily violated. At the end of the execution and before calls to non-helper methods the invariants have to be restored again.

Though loop invariants are not needed for the interface specification, they can be given to support the verification and for a better understanding of the program code.

To check whether a program satisfies its specification, there are several tools available; one of them is KeY (see section 1.2).

JML specifications can be either written in an additional file or included as special annotation comments in the Java files themselves. To ease the handling, JML syntax is based upon Java syntax with the addition of some new expressions and keywords.

[3] `http://i11www.iti.uni-karlsruhe.de/en/`

1.1.1 A Specification Example

In listing 1 a small example of a JML specification is shown. The code consists of an integer field n, a setter *set(m)* and a getter *get()*. Comments whose first non-whitespace character is an @, are interpreted as JML specifications.

Listing 1: A specification example

```
1  public class Example {
2
3      //@ invariant n >= 0;
4      int n;
5
6      /*@ public normal_behavior
7        @   requires m >= 0;
8        @   ensures n == m;
9        @   modifies n;
10       @*/
11     void set(int m) {
12         n = m;
13     }
14
15     /*@ public normal_behavior
16       @   ensures \result == n;
17       @   ensures \result >= 0;
18       @*/
19     /*@ pure @*/ int get() {
20         return n;
21     }
22
23 }
```

In line 3 the keyword *invariant* is given to define the class invariant that the integer n is never smaller than 0.

In lines 6–10 a method contract for the method *set(m)* is specified. It is started in line 6 with the keywords *public*, which specifies the visibility of the contract, and *normal_behavior*, which states that the method will throw no exception (if the preconditions were holding). The keyword *requires* in line 7 begins a precondition stating that the parameter m may not be smaller than 0. In line 8 a postcondition is started with *ensures* and states that after the exectuion of the method the value of field n equals the value of the parameter m with which the method was called. In line 9 the keyword *modifies* is given to define the modifies clause consisting of n. Thus, the method is only allowed to modify the value of n, as well as local variables. Termination of the method is required implicitly.

In lines 15–19 a method contract for the method *get()* is given. Again, the method is declared to throw no exception. The postcondition in line 16 states that the return value of the method, which can be accessed by the keyword *\result*, equals the value of n. Another postcondition is specified in line 17 requiring the return value to not be smaller than 0. Multiple postconditions (as well as preconditions) are conjunctively linked, i.e. all postconditions have to be true after the execution of the method. The keyword *pure* in line 19 declares the method to be pure, meaning it modifies nothing besides local variables. Declaring a method as *pure* is equivalent to specifying an empty modifies clause (*modifies \nothing*).

There are still several other keywords in JML, which we will introduce when needed. The complete documentation of JML can be found in

[LPC+03].

1.2 KeY

The KeY tool (also called KeY system or just KeY) results from the KeY Project[4]. This project was founded in November 1998 at the University of Karlsruhe. In the meantime it became a joint project of the University of Karlsruhe, Chalmers University of Technology, Gothenburg, and the University of Koblenz. Besides much theoretical research in the field of software verification, an interactive theorem prover for first-order Dynamic Logic for Java was developed: the KeY tool.

KeY allows the verification of Java programs and is written in Java itself. The Dynamic Logic that KeY uses was developed for Java Card, a subset of Java. However, it supports some more features and thus can be used for the verification of almost any sequential Java program without dynamic class loading and floating point types.

To produce derivations, a sequent calculus is used. By application of rules (called taclets) a proof tree is built. The leaves are called goals and can be either open or closed. If all goals are closed, the proof is complete and the proof obligation is shown.

The application of taclets can be done automatically, for which KeY has an auto mode, or by hand. For this, KeY comes with a graphical user interface which is quite intuitive. Besides options to direct the automated proof search, which can be changed on-the-fly, the whole proof tree is displayed and every node with its sequent can be examined closely, including open goals for taclet application.

KeY provides an interface to JML (see section 1.1), so that proof obligations can be automatically generated using JML specifications.

1.2.1 Verification with KeY

To prove the correctness of a Java program annotated with JML specifications there are several proof obligations.

For every constructor and every non-helper method we must verify that after its execution (in case of a constructor this is after a new object was created) all class invariants hold. This proof obligation is called *PreserveInvs*.

Then we have to show that every method fulfills each of its contracts. On the one hand, the postconditions of the method must hold after its execution. On the other hand, the method may only modify memory locations referenced in the modifies clause. The former proof obligation is called *EnsuresPost*, the latter *RespectsModifies*.

For proof obligations for non-helper methods we can assume that the class invariants hold before the method call, making them additional preconditions

[4] `http://www.key-project.org/`

for the method.

In the verification of a method its body is symbolically executed, i.e. the effect of a statement is evaluated in the Dynamic Logic according to its semantics. While for basic statements like assignments this is straightforward, there are some special cases.

The handling of method calls can be done in two ways. A simple solution is to inline the method and symbolically execute its body. For modularization purposes however, it is better to use a contract of the called method. We can then use the postconditions and hide the method body, which reduces the proof size. In the proof tree the use of a method contract results in the three branches *Pre*, *Post* and *Exceptional Post*. In *Pre* we have to show that the method's preconditions hold. In *Post* and *Exceptional Post* we can use the postconditions to continue the verification. *Post* covers the method's execution without throwing an exception, *Exceptional Post* the execution with throwing an exception. Using a *normal_behavior* contract lets the branch *Exceptional Post* be closed right away, since no exception is thrown.

Loops in programs can be handled either by induction or loop invariants. Since we will only use loop invariants here we omit an introduction to induction. The use of a loop invariant leads to the four branches *Invariant Initially Valid*, *Body Preserves Invariant*, *Use Case* and *Termination* in the proof tree. In *Invariant Initially Valid* we must show that the loop invariant is valid before the loop is executed. Then, when the loop condition is true, an execution of the loop's body must preserve the invariant (*Body Preserves Invariant*). In *Use Case* we can use the loop invariant and the negation of the loop condition to show the postconditions. In the branch *Termination* we have to show that the loop condition will eventually be false, so that the loop terminates.

In another variation of the loop invariant we can replace the branch *Termination* by a variant. This variant must be non-negative and every execution of the loop body has to decrease it. Furthermore, if the variant is decreased to 0, the loop condition must evaluate to false. With these properties termination is shown.

The proof obligation *RespectsModifies* can be shown using weak loop invariants and method contracts. We also use only simple invariants, such that both *RespectsModifies* and *PreserveInvs* can be easier shown than *EnsuresPost*. Therefore we will only detail the verification of the proof obligation *EnsuresPost* in this work.

1.2.2 Settings

For the verification tasks in this work a version of KeY that was not released officially was used. The current release, KeY 1.4, lacks the ability to deal with loop invariants with variants, which we employed when dealing with loops. Therefore KeY was used in the version of July 17th 2009. Newer

versions should work as well, but differences in the proofs are possible.

For such complex proof obligations as we have here we were not able to complete the proofs with the default settings due to some performance issues KeY has (see section 7.2).

Normally, KeY would check for null pointers in cases where a *NullPointerException* could be raised, e.g. when accessing an array. This generates some overhead in the proof, so we deactivated these checks by setting *nullPointerPolicy* to *noNullCheck*. In principle the proofs could be done with null pointer checks, too.

The options for the proof search strategy control which rules can be automatically applied. Some of them had to be changed as well. For KeY to use method contracts instead of inlining method bodies, the option *Method treatment* was set to *Contracts*. We also set the option *Quantifier treatment* to *No splits*, which reduces the amount of quantifier instantiations KeY does automatically.

Though these are our default settings, in several cases we had to limit the automated rule application further for a successful verification.

1.3 Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm is an algorithm to find shortest paths in a graph with weighted edges. It was developed by Dutch computer scientist Edsger Wybe Dijkstra in 1959 [Dij59] and is still widely used today, e.g. in the network routing protocol OSPF [Moy98].

In its basic form it computes the shortest paths from a source node to all reachable nodes. It only requires all edge weights to be non-negative, and works for undirected as well as directed graphs. If the graph is not connected, i.e. there are unreachable nodes, their distances are set to ∞ and they are thus marked as not reachable. In listing 2 the algorithm for a graph *G = (nodes, edges)* is shown in pseudocode.

Listing 2: Dijkstra's Algorithm in pseudocode

```
1  for each node n in nodes
2      distance[n] := infinite
3  distance[s] := 0
4  visited := {}
5
6  while exist nodes not in visited
7      n := node with smallest distance[n] which is not in visited
8      if distance[n] = infinite
9          break
10     visited := visited union {n}
11
12     for each node m with (n, m) in edges which is not in visited
13         d := distance[n] + weight(n, m)
14
15         if d < distance[m]
16             distance[m] := d
```

The set *nodes* contains all nodes of the graph, the edges are stored as tupels of nodes in the set *edges*. The function *weight(n, m)* returns the weight of the edge between nodes n and m and is defined if the tupel *(n,*

m) is contained in *edges*. The array *distance* stores for all nodes the current distance from the source node. A distance of ∞ for a node n means that n is not (yet) reachable. A node becomes reachable, when a predecessor m is visited and the distances of m's successors are updated. The set *visited* contains nodes for which the shortest distance is found; those nodes are called visited.

The algorithm is initialized by setting the distance for every node to ∞ (lines 1–2), except for the source node s, whose distance is set to 0 (line 3). The set *visited* is initialized with the empty set (line 4). In this state only the source node is reachable and no node is visited.

While there exist unvisited nodes (line 6, the outer loop), the unvisited node n with the smallest distance (line 7) is set visited (line 10). For every unvisited successor m of node n (line 12, the inner loop) the distance d to node m via n is calculated (line 13) and compared to the current distance for m (line 15). If the new distance is shorter, the current distance is updated (line 16).

After one execution of the outer loop's body one more node is visited and that node's successors are reachable with updated distances (where applicable). By choosing to visit the node with the smallest distance it is ensured that there is no shorter path to this node.

When all nodes are visited, all shortest paths are found and the algorithm terminates. Since in every iteration of the outer loop one unvisited node is set visited and the number of nodes is finite, the algorithm terminates eventually.

If the graph is not connected, there are some nodes which can not be reached from the source node. At some iteration of the outer loop there are only those unreachable nodes left unvisited. Thus, if the node n with the smallest distance has a distance of ∞, the algorithm can be terminated (lines 8–9) prematurely.

The main invariant responsible for the correct computation of the shortest paths is that every visited node u, i.e. u is contained in the set *visited*, has a distance which is smaller than or equal to the distance of every unvisited node v. In the beginning, *visited* is empty and the invariant holds. By always adding the node with the smallest distance to *visited*, the invariant is preserved. Furthermore, when updating the successors m of node n, due to non-negative edge weights, m's updated distance is never smaller than the distance of n. Because n has the largest distance amongst all visited nodes, the invariant can not be violated.

Subsequently, if a node is visited, there can not exist a shorter path to that node.

We have to distinguish between different meanings of the term *unreachable node*. When speaking about the graph a node is unreachable if there exists no path from the source node to that node. When speaking about Dijkstra's Algorithm, which we do from now on if not stated otherwise, we

talk about the current iteration of the algorithm's outer loop. A node is reachable if one of its predecessors is visited or if it is the source node, which is reachable from the beginning. For the current iteration all other nodes are unreachable. However, those can become reachable in a later iteration. Only when the algorithm has terminated, both meanings are identical.

2 Concrete Implementation

The implementation of Dijkstra's Algorithm provided by Prof. Wagner's group was written in C++. Since we want to use JML and KeY for the specification and verification, we ported the implementation to Java. As the Java implementation was done as close as possible to the provided C++ code, the original code is not shown in this work. Both implementations include a framework for running and testing the algorithm, which we will not show as well. However, the complete Java implementation and the original C++ code can be found on the enclosed CD.

The implementation uses a small optimization we call t-optimization. The standard variant of Dijkstra's Algorithm computes the shortest paths from a source node to all reachable nodes. In cases where a shortest path from a source node to only one node (the target node) is sought, we can modify the algorithm slightly. We know that when a node is set visited, its shortest path is found (see section 1.3). With this knowledge we can stop the algorithm when the target node is set visited. This optimization was already described by E. W. Dijkstra in his original paper [Dij59].

2.1 Code

The code consists of three classes, *Graph*, *PriorityQueue* and *Dijkstra*. Class *Dijkstra* contains the algorithm itself, *Graph* is the representation of the graph and *PriorityQueue* helps in finding the node with the smallest distance.

While there are two inner classes *Node* and *Edge*, we identify nodes and edges by their index in the corresponding array of class *Graph* (see section 2.1.1). The classes *Node* and *Edge* are only used to store additional information for nodes and edges. When we talk about a node or an edge, we always mean its index, which is an integer between 0 and the array's length.

The algorithm is started by calling the main method *run(s, t)* (see section 2.1.3) with the source node s and the target node t as parameters. It then computes the shortest path from the source node to the target node. If t is an invalid node, i.e. an invalid index of the node array, the algorithm computes the shortest distances from the source node to all reachable nodes. The implementation is optimized for subsequent runs of the algorithm with the same graph, only different source and target nodes.

2.1.1 Graph

The graph is stored in a Forward and Reverse Star Representation [AMO93] using an adjacency array as it is presented for static graphs in conjunction with bidirectional techniques by D. Delling [Del09]. It consists of two arrays, one for the nodes, the other for the edges. Every node n has a pointer to its first edge in the edge array. The edges are ordered by their source node, so all following edges up to the first edge of the next node belong to node n, too. A dummy node at the end of the node array works as an upper bound for the last node's edges by pointing to the edge array's first invalid index. Accordingly, the number of nodes is the length of the node array minus 1. Every edge has a pointer to its target node and a weight. Additionally, since this data structure is also used for bidirectional techniques, edges have flags indicating an outgoing and/or an incoming edge. In this representation an edge is stored twice, as an outgoing edge for its source node and as an incoming edge for its target node. Figure 1 demonstrates the use of this data structure with a small example.

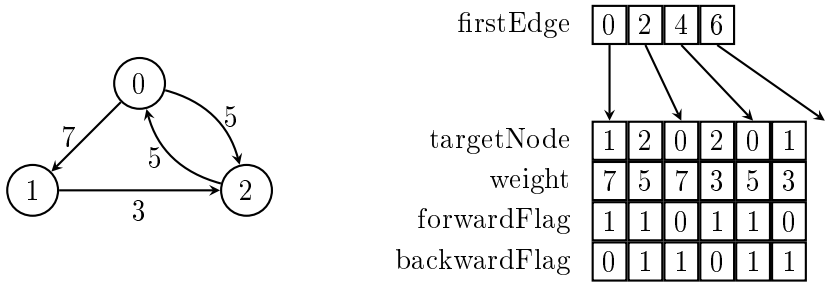

Figure 1: An example graph with its representation in the adjacency array

Listing 3: The data structure of the graph

```
1  public class Graph {
2      Node[] nodes;
3      Edge[] edges;
4
5      int numberOfNodes() {
6          return nodes.length - 1;
7      }
8
9      static class Node {
10         int firstEdge;
11     }
12
13     static class Edge {
14         int targetNode;
15         boolean forwardFlag;
16         boolean backwardFlag;
17         int weight;
18     }
19
20 }
```

Listing 3 shows the source code of the class *Graph*. In line 2 the node array *nodes* is declared. It is of type *Node* (lines 9–11), which points with the integer *firstEdge* to that node's first edge. Line 3 introduces the edge array

edges of type *Edge* (lines 13–18). *Edge* stores the additional information for an edge: the integer *targetNode* points to the target node of this edge, the boolean flags *forwardFlag* and *backwardFlag* indicate, whether the edge is an outgoing and/or an incoming edge, and the integer *weight* stores the edge weight. The method *numberOfNodes()* (lines 5–7) returns the number of nodes of the graph, which is, due to the dummy node at the end of the node array, *nodes.length − 1*.

2.1.2 PriorityQueue

In the abstract algorithm (see section 1.3) a set *visited* was used for the nodes whose shortest paths are determined. From the other (unvisited) nodes the one with the shortest distance was selected in the next iteration of the outer loop. Thus, only the complementary set of *visited* is interesting for the progression of the algorithm. We don't need to consider unreachable nodes, i.e. nodes with a distance of ∞, either.

Hence, this implementation only stores the set of nodes which are reachable and unvisited. This is done using a priority queue to easily access the node with the smallest distance. In the original code the queue was backed by a binary heap, while the Java implementation is simpler, but not as efficient.

Because the simple implementation does not correspond to the original implementation and the priority queue is not part of the algorithm itself, we refrain from verifying the priority queue. In this work, we only consider its interface, which is shown in listing 4.

Listing 4: The interface of the priority queue

```
1  public class PriorityQueue {
2      int[] nodes;
3      int[] distances;
4      int size;
5
6      void clear();
7      boolean empty();
8      int deleteMin();
9      void insert(int n, int d);
10     void decreaseKey(int n, int d);
11 }
```

The array *nodes* (line 2) holds the nodes that are contained in the queue. Their distances are stored in the array *distances* (line 3) using the same indices. The number of nodes currently in the queue is indicated by the integer *size* (line 4).

The method *clear()* (line 6) removes all nodes from the queue and the method *empty()* (line 7) tests for the empty queue. *deleteMin()* (line 8) returns the node with the smallest distance currently in the queue and removes that node from it. A node is added to the queue by the method *insert(n, d)* (line 9) where n is the node and d its distance.

If the distance of a node which is in the queue changes, the distance known to the queue needs to be updated, as well. Because the distance of a node is only getting smaller and is used in the queue as the key to order by,

the method to update the distance is called *decreaseKey(n, d)* (line 10). n is the node with the changed distance and d its new distance.

2.1.3 Dijkstra

The implementation of the algorithm, which is located in class *Dijkstra*, differs from the abstract algorithm described in section 1.3 in several points. Some are the results of implementing in an actual programming language and of using the graph's data structure and the priority queue described in the sections 2.1.1 and 2.1.2.

One difference is the implementation of the value ∞. Since there is no such value in Java (and also in C++), another approach had to be used. In this implementation the value ∞ is modeled by an additional reachability function. If a node's distance is smaller than ∞ it is flagged reachable and its distance is stored in the array *distance*. Otherwise it is flagged unreachable and the value of *distance* at that nodes' index can be arbitrary.

The reachability function is implemented using a counter which is increased in each run of the algorithm. If a node becomes reachable, the value of the array *runs* at the node's index is set to the value of the counter. Hence, a node is reachable if the value of *runs* for that node equals the value of the counter. For the next run the reachability of all nodes can be reset by increasing the counter.

Listing 5: Implementation of the algorithm without the method *run(s,t)*

```
1  public class Dijkstra {
2      Graph graph;
3      PriorityQueue queue;
4      int counter;
5      int[] runs;
6      int[] distance;
7      int settledNodes;
8      int relaxedEdges;
9      int target;
10
11     void init(int s, int t) {
12         counter++;
13         queue.clear();
14         distance[s] = 0;
15         queue.insert(s, 0);
16         runs[s] = counter;
17         target = t;
18         relaxedEdges = 0;
19         settledNodes = 0;
20     }
21
22     int getDistance() {
23         return distance[target];
24     }
25
26 }
```

In listing 5 the code of class *Dijkstra* is shown – without the main method *run(s, t)* which we present for itself. The graph is stored in the field *graph* (line 2), the priority queue in the field *queue* (line 3). Line 4 declares the counter to use for the values of the array *runs* (line 5) to indicate reachability. In the array *distance* (line 6) the distances of the nodes are stored. In lines 7–8 the integers *settledNodes* and *relaxedEdges* are declared, which are used

for statistical purposes. *settledNode* holds the number of nodes which are visited, *relaxedEdges* holds the number of edges which were considered for a shortest path, i.e. the distances via those edges were computed and compared to the distances known before. The target node with which the algorithm was called is stored in *target* (line 9).

For the initialization the method *init(s, t)* (lines 11–20) is called in the beginning of the main method *run(s, t)*. It increases the counter, thus setting all nodes unreachable and removes all nodes from the queue (lines 12–13). It then sets the distance of the source node s to 0, adds s to the queue und sets it reachable (lines 14–16). Finally, the target node is stored in *target* and *settledNodes* and *relaxedEdges* are reset to 0 (lines 17–19).

The method *getDistance()* (lines 22–24) returns the distance to the target node computed in the last run of the algorithm.

Listing 6: The algorithm's main method *run(s, t)*

```
1  void run(int s, int t) {
2      init(s, t);
3
4      while (!queue.empty()) {
5          int u = queue.deleteMin();
6          settledNodes++;
7
8          if (u == t) {
9              return;
10         }
11
12         for (int i = graph.nodes[u].firstEdge;
13                 i < graph.nodes[u + 1].firstEdge; i++) {
14
15             if (!graph.edges[i].forwardFlag) {
16                 continue;
17             }
18
19             relaxedEdges++;
20             int v = graph.edges[i].targetNode;
21             int d = distance[u] + graph.edges[i].weight;
22
23             if (runs[v] != counter) {
24                 distance[v] = d;
25                 queue.insert(v, d);
26                 runs[v] = counter;
27             } else if (d < distance[v]) {
28                 distance[v] = d;
29                 queue.decreaseKey(v, d);
30             }
31
32         }
33
34     }
35
36 }
```

Listing 6 shows the method *run(s, t)*, which is called to start the algorithm. Parameter s is the source node, t is the target node. The algorithm starts with the initialization by calling *init(s, t)* (line 2).

Then the outer loop (lines 4–34) is executed as long as there are nodes in the priority queue (line4). This corresponds to the abstract algorithm as the queue contains the nodes that are reachable and unvisited (see section 2.1.2). With a call to the queue's method *deleteMin()* the node with the smallest distance is removed from the queue and it is stored in u (line 5). The number of visited nodes is then incremented (line 6).

If node u is the target node, the algorithm is stopped (lines 8–10). Otherwise, the inner loop (lines 12–32) iterates over every edge of node u. It starts with the first edge of u in the graph's edge array *edges* and stops at the first edge of the next node ($u + 1$). In lines 15–17 we check, whether the current edge is an outgoing edge. In this case the rest of the loop body is executed. Otherwise, i.e. when *forwardFlag* is not set for this edge, the body is skipped.

If the current edge i is outgoing, the number of considered edges is incremented (line 19). In the variable v the target node of edge i is stored and the distance d to node v via edge i is computed (lines 20–21).

In line 23 it is checked if v is unreachable until now. In this case, it is added to the queue, it is set reachable and its distance is set to d (lines 24–26). If v is already reachable, the new distance d is compared to the current and updated, if necessary – both in the array *distance* and in the priority queue (lines 27–29).

After the inner loop has iterated over all edges, the outer loop continues with its next iteration. When there are no more nodes in the queue, the algorithm terminates.

2.2 Specification

In the beginning of the specification and verification process it became clear that working directly with the concrete implementation would not work. Although the implementation has only 20 lines of code (method *run(s, t)* without blank lines and closing brackets), initial verification attempts showed that the task had a higher complexity than expected. The (partial) proof became so large that we could not work with it to find the right specification. We decided to start with a simpler implementation.

We will present an abstract implementation without t-optimization and with simpler data structures in section 3 and add the t-optimization again in section 4. We will then continue with the specification and verification of the concrete implementation in section 5.

3 Abstract Implementation

We developed the abstract implementation to start the specification and verification with a simpler implementation. It uses no priority queue and a simpler data structure for the graph. We also omitted the t-optimization.

The algorithm's main method *run(s, t)* of the concrete implementation (see section 2) is now split into three methods. The method *innerLoop(s, u, i)* contains the body of the inner loop and is called by the method *outerLoop(s)* which contains the body of the outer loop. *outerLoop(s)* is called by the method *run(s)* which starts the algorithm and does not need the parameter t anymore.

3.1 Code

The abstract implementation consists of only the class *DijkstraAbstract*. Besides the three main methods *run(s)*, *outerLoop(s)* and *innerLoop(s, u, i)* there is the initialization method *init(s)* and two abstract methods *existsMin()* and *getMin()* which assume the role of the priority queue.

The graph's data structure is implemented using an adjacency matrix with the nodes being the indices of the matrix. Edges are identified by their source and target nodes. In contrast to the concrete implementation there cannot be two edges connecting the same nodes.

Listing 7: The abstract implementation without the methods *run(s)*, *outerLoop(s)* and *innerLoop(s, u, i)*

```
1  public abstract class DijkstraAbstract {
2      int nodeCount;
3      boolean[][] edge;
4      int[][] weight;
5      int[] distance;
6      boolean[] reachable;
7      boolean[] visited;
8      int visitedNodes;
9
10     abstract boolean existsMin();
11     abstract int getMin();
12
13     void init(int s) {
14
15         for(int i = 0; i < nodeCount; i++) {
16             reachable[i] = false;
17             visited[i] = false;
18         }
19
20         reachable[s] = true;
21         distance[s] = 0;
22         visitedNodes = 0;
23     }
24
25 }
```

Listing 7 shows the code of the abstract implementation – without the methods *run(s)*, *outerLoop(s)* and *innerLoop(s, u, i)* which are presented for themselves. The graph is declared in lines 2–4, *nodeCount* indicates the number of nodes and the arrays of array *edge* and *weight* implement the adjacency matrix. An edge between two nodes n and m exists if the value of *edge[n][m]* is *true*. In this case, the value of *weight[n][m]* denotes the weight of the edge.

As in the concrete implementation the array *distance* (line 5) stores the computed distances for all nodes. The reachability function is implemented here by the array *reachable* (line 6). A node n is reachable if the value of *reachable* at index n is *true*. Otherwise, n is unreachable. The same holds for the array *visited* which indicates whether a node is visited (line 7). The number of visited nodes is stored in *visitedNodes* (line 8) and corresponds to *settledNodes* of the concrete implementation.

The abstract method *existsMin()* (line 10) checks whether a node exists that is reachable and unvisited. It corresponds to the queue's method *empty()* of the concrete implementation. *getMin()* (line 11) corresponds to the queue's method *deleteMin()* and returns the node with the smallest dis-

tance which is reachable and unvisited. As with the queue's implementation (see section 2.1.2) we do not present the implementations of both methods because they are not part of the algorithm itself.

The initialization method *init(s)* in lines 13–23 sets all nodes unreachable and unvisited (lines 15–18). Then the source node is set reachable with a distance of 0 (lines 20–21) and the number of visited nodes is reset (line 22).

Listing 8: The main methods *run(s)*, *outerLoop(s)* and *innerLoop(s, u, i)* of the abstract implementation

```
1  void run(int s) {
2      init(s);
3
4      while (existsMin()) {
5          outerLoop(s);
6      }
7
8  }
9
10 void outerLoop(int s) {
11     int u = getMin();
12     visited[u] = true;
13     visitedNodes++;
14
15     for (int i = 0; i < nodeCount; i++) {
16
17         if (!edge[u][i]) {
18             continue;
19         }
20
21         innerLoop(s, u, i);
22     }
23
24 }
25
26 void innerLoop(int s, int u, int i) {
27     int d = distance[u] + weight[u][i];
28
29     if (!reachable[i]) {
30         reachable[i] = true;
31         distance[i] = d;
32     } else if (d < distance[i]) {
33         distance[i] = d;
34     }
35
36 }
```

The main methods of the abstract implementation are presented in listing 8.

With the method *run(s)* (lines 1–8) the algorithm is started. After the call to *init(s)* (line 2) the outer loop (lines 4–6) starts. As long as there are reachable and unvisited nodes, which is checked by *existsMin()*, *outerLoop(s)* is executed.

The method *outerLoop(s)* (lines 10–24) stores the node with the smallest distance which is reachable and unvisited in the variable u, sets it visited and increases the number of visited nodes (lines 11–13). The inner loop (lines 15–22) checks for every node i whether i is a successor of u and continues with the next iteration if it is not (lines 17–19). If it is, the method *innerLoop(s, u, i)* is executed.

For every successor i of node u, *innerLoop(s, u, i)* (lines 26–34) is executed. It computes the distance d to node i via node u (line 27). If i is not yet reachable, it is set reachable and its distance is set to d (lines 29–31). Otherwise, if the new distance d is smaller than the current distance to i,

the distance is updated (lines 32–33).

3.2 Specification

We start the specification process with the class invariants. In this abstract implementation they mostly deal with the lengths of arrays and aliasing. We need to specify the lengths of the used arrays in order to show that no *IndexOutOfBoundsException* will be thrown when accessing an array.

Aliasing in Java occurs not only when dealing with normal objects, but when dealing with arrays, too. If we have two variables a and b of type array of integer, where both point to the same array, changing the value of a at index i results in the same change of the value of b at index i. Thus, we have to specify that every array where a value gets changed is not the same array as any other array of the same type.

Listing 9: Class invariants for the abstract implementation

```
1  //@ invariant nodeCount >= 1;
2  int nodeCount;
3
4  /*@ invariant edge.length == nodeCount;
5    @ invariant (\forall int n; 0 <= n && n < nodeCount;
6    @      edge[n].length == nodeCount);
7    @*/
8  boolean[][] edge;
9
10 /*@ invariant weight.length == nodeCount;
11   @ invariant (\forall int n; 0 <= n && n < nodeCount;
12   @      weight[n].length == nodeCount);
13   @ invariant (\forall int n; 0 <= n && n < nodeCount;
14   @      (\forall int m; 0 <= m && m < nodeCount;
15   @          weight[n][m] >= 0));
16   @*/
17 int[][] weight;
18
19 //@ invariant distance.length == nodeCount;
20 int[] distance;
21
22 //@ invariant reachable.length == nodeCount;
23 boolean[] reachable;
24
25 //@ invariant visited.length == nodeCount;
26 boolean[] visited;
27
28 int visitedNodes;
29
30 /*@ invariant reachable != visited;
31   @ invariant (\forall int n; 0 <= n && n < nodeCount;
32   @      reachable != edge[n] && visited != edge[n]);
33   @ invariant (\forall int n; 0 <= n && n < nodeCount;
34   @      distance != weight[n]);
35   @ invariant (\forall DijkstraAbstract d; \created(d) && this != d;
36   @      reachable != d.reachable && reachable != d.visited &&
37   @      (\forall int n; 0 <= n && n < d.nodeCount; reachable != d.edge[n]));
38   @ invariant (\forall DijkstraAbstract d; \created(d) && this != d;
39   @      visited != d.visited && visited != d.reachable &&
40   @      (\forall int n; 0 <= n && n < d.nodeCount; visited != d.edge[n]));
41   @ invariant (\forall DijkstraAbstract d; \created(d) && this != d;
42   @      distance != d.distance && (\forall int n; 0 <= n && n < d.nodeCount;
43   @          distance != d.weight[n]));
44   @*/
```

The class invariants for the abstract implementation are shown in listing 9. Because we require a start node for the algorithm, the number of nodes in the graph must be at least 1, which is stated in line 1.

Lines 4–7 deal with the 2-dimensional boolean array *edge*. It is specified that the length of *edge* must be equal to the number of nodes. Otherwise, an

IndexOutOfBoundsException could be raised. Also, every inner array of *edge* must have this length, too. Iteration over all indices of an array is done with a universal quantifier. It begins with the keyword *\forall* followed by the type and name of the variable to iterate over. Optionally a range restriction can be given as we have done here. At last, the formula which should hold follows.

For the 2-dimensional integer array *weight* we have the same requirements (lines 10–12). Additionally, every value of *edge* must be at least 0 (lines 13–16), because the algorithm requires non-negative edge weights.

For the arrays *distance*, *reachable* and *visited* we only need to specify the lengths (lines 19–26). The rest is handled in the specification of the main methods. For *visitedNodes* (line 28) no specification is needed, because it is initialized in the algorithm itself.

In lines 30–44 we state that arrays where values are modified do not equal other arrays of the same type. E.g. the array *reachable* is not identical to the array *visited* (line 30), or the array *distance* does not equal the array *distance* or any inner array of *weight* in any other *DijkstraAbstract* instance (lines 41–44).

We will now present the method contracts for the six methods. We begin with *existsMin()*, *getMin()* and *init(s)*. The main methods will be dealt with in separate sections, *run(s)* in section 3.2.1, *outerLoop(s)* in section 3.2.2 and *innerLoop(s, u, i)* in section 3.2.3.

The method *existsMin()* checks whether there are reachable and unvisited nodes. It returns *true* if there is such a node, and *false* if there is none. The method contract is shown in listing 10.

Listing 10: Specification for the abstract method *existsMin()*

```
1  /*@ public normal_behavior
2    @   ensures \result <==> (\exists int n; 0 <= n && n < nodeCount;
3    @         reachable[n] && !visited[n]);
4    @   ensures visitedNodes == nodeCount ==> !\result;
5    @*/
6  /*@ pure @*/ abstract boolean existsMin();
```

The method contract is started in line 1 stating no exception will be thrown. *existsMin()* requires no precondition, but has two postconditions.

The first postcondition in lines 2–3 specifies that the return value *\result* is *true* if and only if there exists a node, i.e. a valid index between 0 and *nodeCount*, that is reachable and unvisited. The existential quantifier is started with the keyword *\exists* and uses the same syntax as the universal quantifier.

The second postcondition (line 4) is needed for KeY to find a proof for the termination of the while loop in method *run(s)*, see section 3.2.1. It states that if the number of visited nodes equals the number of all nodes, *existsMin()* will return *false*. Because in this case there are no unvisited nodes left, the method behaves according to the first postcondition.

Since *existsMin()* should not modify any values, it is declared *pure* in line 6.

If the method *existsMin()* returns *true*, then the method *getMin()* should return the node with the smallest distance which is reachable and unvisited. Otherwise, there is no node with this property and *getMin()* cannot return the minimum. Therefore, *getMin()* can only be called when *existsMin()* returns *true*.

When it is executed, it should return a reachable and unvisited node. This node must also be the one with the smallest distance among all nodes which are reachable and unvisited. In contrast to the method *deleteMin()* of the queue in the concrete implementation, it does not modify anything and the algorithm needs to set the returned node visited. The method contract is presented in listing 11.

Listing 11: Specification for the abstract method *getMin()*

```
1  /*@ public normal_behavior
2    @   requires existsMin();
3    @   ensures 0 <= \result && \result < nodeCount;
4    @   ensures reachable[\result] && !visited[\result];
5    @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
6    @        !visited[n]; distance[\result] <= distance[n]);
7    @*/
8  /*@ pure @*/ abstract int getMin();
```

Line 2 shows the precondition that the method *existsMin()* has to return *true*. The postconditions in lines 3–4 ensure that the return value is a valid index, i.e. a node, and that it is reachable and unvisited.

With the postcondition in lines 5–6 we ensure that of all reachable and unvisited nodes the returned node has the smallest distance. Finally, *getMin()* is declared *pure* in line 8.

The method *init(s)* is called in the beginning of the algorithm. Every node has to be set unreachable and unvisited. Accordingly, *visitedNodes* is set to 0. Only the source node s is set reachable with a distance of 0. Because the source node s must be valid node, we add this as a precondition. For this method the modifies clause is not empty, because it modifies some values. Listing 12 shows the contract for *init(s)*.

Listing 12: Specification for the method *init(s)*

```
1  /*@ public normal_behavior
2    @   requires 0 <= s && s < nodeCount;
3    @   ensures distance[s] == 0;
4    @   ensures reachable[s];
5    @   ensures (\forall int n; 0 <= n && n < nodeCount && n != s; !reachable[n]);
6    @   ensures (\forall int n; 0 <= n && n < nodeCount; !visited[n]);
7    @   ensures visitedNodes == 0;
8    @   modifies visitedNodes, visited[*], reachable[*], distance[s];
9    @*/
10 void init(int s) {
11
12     for(int i = 0; i < nodeCount; i++) {
13         reachable[i] = false;
14         visited[i] = false;
15     }
16
17     reachable[s] = true;
18     distance[s] = 0;
19     visitedNodes = 0;
20 }
```

Line 2 requires the parameter s to be a valid node. Then in lines 3–4 it is ensured that s is reachable and its distance is set to 0. Every other node will be unreachable (line 5). The postconditions in lines 6–7 states that all nodes are set unvisited and *visitedNodes* is reset. *init(s)* modifies *visitedNodes*, all values of the arrays *visited* and *reachable* and the value of *distance* at index s (line 8).

For the loop in *init(s)* we add a loop invariant with variant as briefly described in section 1.2.1. In an iteration of the loop, all nodes up to node i are already unreachable and unvisited. Then node i is set unreachable and unvisited, and i is incremented, making the invariant hold for the next iteration. The variant is $nodeCount - i$, which is equal to *nodeCount* in the beginning and 0 at the end and gets decremented in every iteration. When it is 0, the loop condition is false, because i equals *nodeCount*, and the loop terminates. Listing 13 presents the specification for the loop.

Listing 13: Loop invariant for the method *init(s)*

```
1  /*@ loop_invariant
2    @  (\forall int n; 0 <= n && n < i; !reachable[n] && !visited[n]) &&
3    @  0 <= i && i <= nodeCount;
4    @ decreases nodeCount - i;
5    @ modifies i, reachable[*], visited[*];
6    @*/
7  for(int i = 0; i < nodeCount; i++) {
8      reachable[i] = false;
9      visited[i] = false;
10 }
```

With the keyword *loop_invariant* (line 1) the loop invariant, which ends in line 3, is started. It states that every node up to node i is unreachable and unvisited (line 2), and that the value of i ranges between 0 and *nodeCount*. We need the latter to show termination.

The variant is given in line 5 beginning with the keyword *decreases*. The modifies clause in line 5 specifies which values the loop modifies. This is i and the values of the arrays *reachable* and *visited*.

3.2.1 run(s)

The method *run(s)* is the starting method for the algorithm and expects the source node as parameter s. The result is the shortest path from s stored in *distance* for each node, if that node is reachable. The reachability is stored in *reachable*.

The postconditions are based on the specification by Böhme, Leino and Wolff for a high-level implementation of Dijkstra's Algorithm [BLW08]. They can be split into two parts, the first one talks about the reachability of nodes, the second one about the distances of the reachable nodes.

A node is reachable, if it has a reachable predecessor. This chain starts with the source node, which is reachable by definition. Then, for all reachable nodes n all nodes m are reachable, if there is an edge between nodes n and m.

The distance of the source node is 0. The distance of every other node m is computed by the distance of one of its predecessors and the weight of the edge between the two nodes. So there exists a predecessor n such that the distance of n plus the weight of the edge between n and m is the distance of m.

Additionally, the distance must be the minimum distance, so for all two nodes n and m holds that, if n is reachable and there is an edge from n to m, then the distance of n plus the edge weight is not smaller than the distance of m. Otherwise a shorter path would exist.

Listing 14: Specification for the method *run(s)*

```
1  /*@ public normal_behavior
2    @   requires 0 <= s && s < nodeCount;
3    @   ensures reachable[s];
4    @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n];
5    @         (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
6    @               reachable[m]));
7    @   ensures distance[s] == 0;
8    @   ensures (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
9    @         (\exists int n; 0 <= n && n < nodeCount && reachable[n] && edge[n][m];
10   @               distance[m] == distance[n] + weight[n][m]));
11   @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n];
12   @         (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
13   @               distance[m] <= distance[n] + weight[n][m]));
14   @   modifies visitedNodes, visited[*], reachable[*], distance[*];
15   @*/
16 void run(int s) {
17     init(s);
18
19     while (existsMin()) {
20         outerLoop(s);
21     }
22
23 }
```

The complete method contract is shown in listing 14. The only precondition is that s is a valid node (line 2). In lines 3–6 the reachability is specified: s is reachable, as well as every successor of a reachable node.

The postconditions for the distances are given in lines 7–13. The distance of the source node is 0 (line 7) and all other distances are computer by the distance of a predecessor (lines 8–10). Lines 11–13 specify that the distances are minimal.

run(s) modifies the value of *visitedNodes* and the values of the arrays *visited*, *reachable* and *distance* (line 14). It does this not by itself, but through the methods *init(s)* and *outerLoop(s)*.

The loop invariant for the loop in *run(s)* consists of several parts. Since the loop is the last piece of code in the method *run(s)*, the postconditions have to be derivable from the invariant. Another part provides formulae, which are needed as preconditions for the call to *outerLoop(s)*. At last, the termination requires some formulae, as well.

To conclude the postconditions of *run(s)* we use similar formulae in the loop invariant. In an iteration of the loop not all successors of reachable nodes are reachable yet. Only the successors of visited nodes are reachable. The same holds for the distances. So for the loop invariant we require the predecessors to be visited, too. Because all visited nodes are reachable,

as well, we introduce an invariant, such that we can omit a visited node's reachability. In the loop body a reachable node becomes visited and its successors are updated. When the loop terminates, all reachable nodes are visited and thus we can derive the postconditions.

The method *outerLoop(s)* requires two additional invariants in order to be called. All distances must be at least 0 and the distances of visited nodes must not be greater than the distances of reachable and not visited nodes (see section 3.2.2).

To show the termination of the loop we again use a variant. In every iteration of the loop one unvisited node becomes visited; and a visited node never becomes unvisited again. So when all nodes are visited, no node is unvisited and the loop terminates. Hence, the variant is $nodeCount - visitedNodes$, with *nodeCount* being the number of all nodes and *visitedNodes* the number of visited nodes. *visitedNodes* starts being 0 and is incremented in every iteration of the loop. So its value ranges from 0 to *nodeCount*, as does the value of the variant, only descending. That the loop indeed terminates, when the variant becomes 0, i.e. *visitedNodes* equals *nodeCount*, cannot be proven with KeY using this knowledge only. A feature is needed that can count the visited nodes and deduce from there that if this number equals the number of all nodes, there cannot be an unvisited node. Though such a feature is implemented in KeY, it is not yet powerful enough for this purpose. We help ourselves by including a postcondition in the contract of *existsMin()*, as mentioned in section 3.2, stating that the method returns *false*, if *visitedNodes* equals *nodeCount*, thus terminating the loop.

Listing 15: Loop invariants for the method *run(s)*

```
1  /*@ loop_invariant
2    @  reachable[s] &&
3    @  (\forall int n; 0 <= n && n < nodeCount && visited[n];
4    @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
5    @            reachable[m])) &&
6    @  distance[s] == 0 &&
7    @  (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
8    @       (\exists int n; 0 <= n && n < nodeCount && visited[n] && edge[n][m];
9    @            distance[m] == distance[n] + weight[n][m])) &&
10   @  (\forall int n; 0 <= n && n < nodeCount && visited[n];
11   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
12   @            distance[m] <= distance[n] + weight[n][m])) &&
13   @  (\forall int n; 0 <= n && n < nodeCount && visited[n]; reachable[n]) &&
14   @  (\forall int n; 0 <= n && n < nodeCount && reachable[n];
15   @       distance[n] >= 0) &&
16   @  (\forall int m; 0 <= m && m < nodeCount && visited[m];
17   @       (\forall int n; 0 <= n && n < nodeCount && reachable[n] && !visited[n];
18   @            distance[m] <= distance[n])) &&
19   @  0 <= visitedNodes && visitedNodes <= nodeCount;
20   @ decreases nodeCount - visitedNodes;
21   @ modifies visitedNodes, visited[*], reachable[*], distance[*];
22   @*/
23 while (existsMin()) {
24     outerLoop(s);
25 }
```

The loop's specification is given in listing 15. The source node s is reachable and all successors m of visited nodes n, as well (lines 2–5). The distance of s is 0 and for every other reachable node m a visited predecessor n exists, such that the distance of m is the distance of n plus the weight of

the edge between n and m (lines 6–9). For any two nodes n and m, where n is visited and an edge exists between n and m, the distance of n plus the edge weight is not smaller than the distance of m (lines 10–12).

Line 13 states that every visited node is reachable, too. For every reachable node the distance is at least 0 and every visited node has a distance that is not greater than the distance of an unvisited node (lines 14–18). For termination it is specified that the value of *visitedNodes* ranges from 0 to *nodeCount* (line 19).

The variant, declared in line 20, is the number of unvisited nodes, i.e. $nodeCount - visitedNodes$; and the loop modifies *visitedNodes* and the values of the arrays *visited*, *reachable* and *distance* (line 21), which it does by calling *outerLoop*.

3.2.2 outerLoop(s)

The method *outerLoop(s)* represents the body of the outer loop and is thus called in the loop body of the method *run(s)*. The specification must be designed such that it allows the verification of the loop. It must be shown that *outerLoop(s)* preserves the loop invariants. Hence, we can assume the loop invariants as preconditions and we need to ensure them as postconditions, which leads to partly identical pre- and postconditions. Additionally, we must prove that the variant is decreased and stays non-negative.

The variant of the loop in *run(s)* is $nodeCount - visitedNodes$. In *outerLoop(s)* we increase *visitedNodes* by 1, so that the variant gets smaller. For the variant to not get smaller than 0, *visitedNodes* must be smaller than *nodeCount*. The loop invariant states that *visitedNodes* can be equal to *nodeCount*, however, in this case *outerLoop(s)* is not executed, because *existsMin()* returns *false* and the loop is terminated.

Another precondition is *existsMin()* to return *true* because *getMin()*, which is called in *outerLoop(s)*, requires it. *outerLoop(s)* requires two more preconditions which we already added as invariants to the loop in *run(s)* (see section 3.2.1). The distances of reachable nodes must be non-negative and the distances of visited nodes must not be greater than the distances of unvisited nodes. The latter is needed to conclude that the node u has the greatest distance amongst visited nodes, which we need for the method *innerLoop(s, u, i)* (see section 3.2.3). That the distances are at least 0 is also needed for *innerLoop(s, u, i)*, which we propagate through the loop invariant to the precondition of *innerLoop(s, u, i)*.

The rest of the pre- and postconditions are known from the invariants of the loop in *run(s)*. The complete contract can be found in listing 16.

In line 2–3 is is required that the method *existsMin()* returns *true* and *visitedNodes* is between 0 and *nodeCount*.

The loop invariants of *run(s)* are assumed in lines 4–21. The source node is reachable (line 4) and successors of visited nodes are reachable (lines 5–7).

Listing 16: Specification for the method *outerLoop(s)*

```
1  /*@ public normal_behavior
2    @  requires existsMin();
3    @  requires 0 <= visitedNodes && visitedNodes < nodeCount;
4    @  requires reachable[s];
5    @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n];
6    @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
7    @            reachable[m]));
8    @  requires distance[s] == 0;
9    @  requires (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
10   @       (\exists int n; 0 <= n && n < nodeCount && visited[n] && edge[n][m];
11   @            distance[m] == distance[n] + weight[n][m]));
12   @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n];
13   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
14   @            distance[m] <= distance[n] + weight[n][m]));
15   @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n];
16   @       reachable[n]);
17   @  requires (\forall int n; 0 <= n && n < nodeCount && reachable[n];
18   @       distance[n] >= 0);
19   @  requires (\forall int m; 0 <= m && m < nodeCount && visited[m];
20   @       (\forall int n; 0 <= n && n < nodeCount && reachable[n] && !visited[n];
21   @            distance[m] <= distance[n]));
22   @  ensures reachable[s];
23   @  ensures (\forall int n; 0 <= n && n < nodeCount && visited[n];
24   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
25   @            reachable[m]));
26   @  ensures distance[s] == 0;
27   @  ensures (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
28   @       (\exists int n; 0 <= n && n < nodeCount && visited[n] && edge[n][m];
29   @            distance[m] == distance[n] + weight[n][m]));
30   @  ensures (\forall int n; 0 <= n && n < nodeCount && visited[n];
31   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
32   @            distance[m] <= distance[n] + weight[n][m]));
33   @  ensures (\forall int n; 0 <= n && n < nodeCount && visited[n];
34   @       reachable[n]);
35   @  ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n];
36   @       distance[n] >= 0);
37   @  ensures (\forall int m; 0 <= m && m < nodeCount && visited[m];
38   @       (\forall int n; 0 <= n && n < nodeCount && reachable[n] && !visited[n];
39   @            distance[m] <= distance[n]));
40   @  ensures visitedNodes <= nodeCount;
41   @  ensures visitedNodes > \old(visitedNodes);
42   @  modifies visitedNodes, visited[*], reachable[*], distance[*];
43   @*/
44 void outerLoop(int s) {
45     int u = getMin();
46     visited[u] = true;
47     visitedNodes++;
48
49     for (int i = 0; i < nodeCount; i++) {
50
51         if (!edge[u][i]) {
52             continue;
53         }
54
55         innerLoop(s, u, i);
56     }
57
58 }
```

The distance of s is 0 and the distances of every other node are computed via a visited predecessor (lines 8–11). There are also no shorter distances via visited nodes (lines 12–14). Then, all visited nodes are reachable, too, all distances of reachable nodes are at least 0 and visited nodes have distances not greater than unvisited nodes (lines 15–21).

For the loop body to preserve the invariants, we use the preconditions of lines 4–21 as postconditions in lines 22–39. Additionally, it is stated that *visitedNodes* is not greater than *nodeCount* (line 40), so that the variant is non-negative, and that the new value of *visitedNodes* is greater than its old value (line 41), so that the variant is decreased.

outerLoop(s) modifies *visitedNodes* and the values of the arrays *visited*, *reachable* and *distance* (line 42).

The inner loop iterates over all successors of node u and updates their reachability and distances. In an iteration of the loop for all visited nodes except u, which was set visited just before the loop, holds that their successors have the same properties as specified in the preconditions. They are reachable, their distances are computed via a visited predecessor and there is no shorter path via a visited predecessor without u. For successors of u up to the current successor i, those properties hold as well. That a reachable node's distance is computed via a visited predecessor is independent of the current iteration and node u, so we can reuse this precondition unchanged.

We also take the preconditions that a visited node is also reachable, visited nodes have not greater distances than unvisited nodes and distances of reachable nodes are at least 0. We also need to state that the distance of node u is not smaller than the distance of any visited node. This is true because u was the unvisited node with the smallest distance.

At last we need a variant for the loop. In this case, this is $nodeCount - i$, with the value of i ranging from 0 to *nodeCount*.

In listing 17 the complete specification for the inner loop is shown. Lines 2–5 state that the source node is reachable and that every successor with a visited predecessor different from u is reachable, too. Also node u's successors up to the current node i are reachable (lines 6–7).

For the distances holds that the distance of the source node is 0 and the distances of all other nodes are computed via a visited predecessor (lines 8–11). For every node m there is no shorter path via a visited predecessor different from u (lines 12–14) or via node u, if m was updated already, i.e. m is smaller than i (lines 15–16).

As previously known, visited nodes are reachable, as well (line 17), and visited nodes have not greater distances than unvisited nodes (18–20). Also all distances are at least 0 and u's distance is not smaller than that of any visited node (lines 21–24).

The loop's variable i is bounded by 0 and *nodeCount*, with the variant being $nodeCount - i$, and the loop modifies i and the values of the arrays *reachable* and *distance* (lines 25–27).

Listing 17: Loop invariants for the method *outerLoop(s)*

```
1  /*@ loop_invariant
2    @  reachable[s] &&
3    @  (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != u;
4    @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
5    @            reachable[m])) &&
6    @  (\forall int m; 0 <= m && m < i && edge[u][m];
7    @       reachable[m]) &&
8    @  distance[s] == 0 &&
9    @  (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
10   @       (\exists int n; 0 <= n && n < nodeCount && visited[n] && edge[n][m];
11   @            distance[m] == distance[n] + weight[n][m])) &&
12   @  (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != u;
13   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
14   @            distance[m] <= distance[n] + weight[n][m])) &&
15   @  (\forall int m; 0 <= m && m < i && reachable[m] && edge[u][m];
16   @       distance[m] <= distance[u] + weight[u][m]) &&
17   @  (\forall int n; 0 <= n && n < nodeCount && visited[n]; reachable[n]) &&
18   @  (\forall int m; 0 <= m && m < nodeCount && visited[m];
19   @       (\forall int n; 0 <= n && n < nodeCount && reachable[n] && !visited[n];
20   @            distance[m] <= distance[n])) &&
21   @  (\forall int n; 0 <= n && n < nodeCount && reachable[n];
22   @       distance[n] >= 0) &&
23   @  (\forall int n; 0 <= n && n < nodeCount && visited[n];
24   @       distance[n] <= distance[u]) &&
25   @  0 <= i && i <= nodeCount;
26   @ decreases nodeCount - i;
27   @ modifies i, reachable[*], distance[*];
28   @*/
29 for (int i = 0; i < nodeCount; i++) {
30
31     if (!edge[u][i]) {
32         continue;
33     }
34
35     innerLoop(s, u, i);
36 }
```

3.2.3 innerLoop(s, u, i)

The method *innerLoop(s, u, i)* is called from the method *outerLoop(s)* for every node i which is a successor of the visited node u. It updates the reachability and distance of i, if necessary. The parameters u and i have to be valid nodes and an edge must exist between them; u also has to be visited.

innerLoop(s, u, i) has to preserve the invariants of the inner loop, which we specified in section 3.2.2. Again, we can assume the loop invariants and must ensure them. However, because *innerLoop(s, u, i)* only changes few values we do not need to include all invariants as postconditions. For the preconditions, we also do not need all of them.

We require that the source node is reachable and that all successors of visited nodes different from u are reachable, too. Since in this method we only deal with node i as successor of u, we do not need to know about its other successors. For the distances the same holds, the source node must have a distance of 0, the distance of every other reachable node is computed via a visited predecessor and for every reachable node there is no shorter path via a visited node different from u. Whether a shorter path via u exists, is, as well as the reachability, of no concern for the method. We also need the invariants that visited nodes are reachable and that the distance of every

visited node is not greater than the distance of any unvisited node.

To prove that the source node has a distance of 0 we need the information that all distances of reachable nodes are at least 0, what we are propagating starting in the loop invariant of the outer loop (see section 3.2.1). To show that the distance of a visited node is not changed we require that the distance of u is not smaller than the distance of any visited node. Because edge weights are non-negative, distances via u cannot be smaller than the distances of visited nodes and thus get changed.

For the postconditions, *innerLoop(s, u, i)* ensures that node i is reachable. Since it does not modify other values of *reachable*, we do not need to state the reachability for other nodes. If i is visited, its distance will not change. To preserve the invariants we ensure that the source node's distance is 0 and that for every other reachable node the distance is computed via a visited node. Again, for every node there is no shorter path via a visited node different from u. We ensure that the distance of i via node u is not smaller than the known distance of i.

At last, we state that again all distances of visited nodes are not greater than distances of unvisited nodes and that the method only modifies the values of *reachable* and *distance* at the index i.

Listing 18: Specification for the method *innerLoop(s, u, i)*

```
1  /*@ public normal_behavior
2    @  requires 0 <= u && u < nodeCount;
3    @  requires 0 <= i && i < nodeCount;
4    @  requires edge[u][i];
5    @  requires visited[u];
6    @  requires reachable[s];
7    @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != u;
8    @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
9    @            reachable[m]));
10   @  requires distance[s] == 0;
11   @  requires (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
12   @       (\exists int n; 0 <= n && n < nodeCount && visited[n] && edge[n][m];
13   @            distance[m] == distance[n] + weight[n][m]));
14   @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != u;
15   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
16   @            distance[m] <= distance[n] + weight[n][m]));
17   @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n];
18   @       reachable[n]);
19   @  requires (\forall int n; 0 <= n && n < nodeCount && reachable[n];
20   @       distance[n] >= 0);
21   @  requires (\forall int n; 0 <= n && n < nodeCount && visited[n];
22   @       distance[n] <= distance[u]);
23   @  requires (\forall int m; 0 <= m && m < nodeCount && visited[m];
24   @       (\forall int n; 0 <= n && n < nodeCount && reachable[n] && !visited[n];
25   @            distance[m] <= distance[n]));
26   @  ensures reachable[i];
27   @  ensures visited[i] ==> distance[i] == \old(distance[i]);
28   @  ensures distance[s] == 0;
29   @  ensures (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
30   @       (\exists int n; 0 <= n && n < nodeCount && visited[n] && edge[n][m];
31   @            distance[m] == distance[n] + weight[n][m]));
32   @  ensures (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != u;
33   @       (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
34   @            distance[m] <= distance[n] + weight[n][m]));
35   @  ensures distance[i] <= distance[u] + weight[u][i];
36   @  ensures (\forall int m; 0 <= m && m < nodeCount && visited[m];
37   @       (\forall int n; 0 <= n && n < nodeCount && reachable[n] && !visited[n];
38   @            distance[m] <= distance[n]));
39   @  modifies reachable[i], distance[i];
40   @*/
41 void innerLoop(int s, int u, int i) {
42     int d = distance[u] + weight[u][i];
43
44     if (!reachable[i]) {
```

```
45            reachable[i] = true;
46            distance[i] = d;
47        } else if (d < distance[i]) {
48            distance[i] = d;
49        }
50
51    }
```

The specification can be seen in listing 18. Lines 2–5 require u and i to be valid nodes, an edge to exist between the two and u to be visited. Lines 6–16 state the reachability for nodes and that the distances are computed via a visited node and that they are minimal for visited nodes different from u.

It follows that visited nodes are also reachable (lines 17–18), that reachable nodes have distances not smaller than 0 (lines 19–20), that u has the greatest distance among visited nodes (lines 21-22) and that visited nodes have not greater distances than unvisited nodes (lines 23-25).

innerLoop(s, u, i) ensures that i is reachable and if it visited, its distance is not changed (lines 26–27). Also, the distance of the source node is 0 and the distance of every other node is computed via a visited predecessor (lines 28–31). There exists no shorter path to any node via a visited node different from u and the distance of i is not greater than the distance via u (lines 32–35).

Lines 36–38 ensure that visited nodes have not greater distances than unvisited nodes and in line 39 the modifies clause is given.

3.3 Verification

Of the three proof obligations *EnsuresPost*, *RespectsModifies* and *PreserveInvs* we introduced in section 1.2.1 we will only demonstrate the proofs for *EnsuresPost*. The other proof obligations can be shown in a similar way. By using weaker loop invariants and method contracts those proofs will even be simpler.

For *EnsuresPost* we have to show that with the preconditions and invariants holding at the beginning of a method, the postconditions hold when that method finishes. The abstract implementation consists of six methods. *existsMin()* and *getMin()* are abstract methods, i.e. they have no implementation, so that we can not (and do not need to) prove their correctness.

The proof obligation for the method *init(s)* can be automatically proven by KeY with our default settings (see section 1.2.2). All other methods require user interaction to some extent. We will give some proof statistics for all proofs in section 7.1.

When we use our default settings in the proof for the method *run(s)* the proof becomes very large and much user interaction is required. To reduce the proof size and minimize user interaction we guide the proof search. In the beginning we limit the rule applications KeY does automatically to avoid unnecessary splits. Later, we lift some restrictions. For this, the option

Logical splitting is set to *Off* and *Quantifier treatment* to *None*. After the auto mode has run with these settings we need to manually apply some rules regarding the method call to *existsMin()*. Then we can start the auto mode again. Thereafter we let the auto mode run with the option *Quantifier treatment* reset to *No splits*, and then again with *Logical splitting* reset to *Delayed*. In the end we need to instantiate some quantifiers by hand before KeY finds the proof.

The verification of the method *outerLoop(s)* is done in a similar way to that of *run(s)*, we also restrict KeY in the rule application. However, we only need to apply rules by hand when the auto mode has run with the three different settings (1. *Logical splitting*: *Off*, *Quantifier treatment*: *None*; 2. *Quantifier treatment*: *No splits*; 3. *Logical splitting*: *Delayed*). The required user interactions are quantifer instantiations similar to the ones in the proof of *run(s)*.

innerLoop(s, u, i) requires the least user interaction and the auto mode can be started with our default settings from the beginning. Before the proof is found, we have to instantiate three quantifiers, again similar to the instantiations of the other proofs.

4 Abstract Implementation with T-Optimization

Before we transfer the specification of the abstract implementation on to the concrete implementation, we include the t-optimization in the abstract implementation, because the concrete implementation uses it, too.

According to the main invariant of Dijkstra's Algorithm (see section 1.3) once a node is set visited there cannot be a shorter path for that node. When searching for a shortest path from a source node to a target node, the algorithm can be stopped, after the target node is set visited. In the case that the target node is, in the graph, unreachable from the source node, or that the target node is not a valid node, it cannot be set visited and thus the algorithm behaves as without t-optimization.

4.1 Code

We modify the abstract implementation so that the t-optimization is implemented similar as in the concrete implementation. So we add an integer field *target* to store the target node of the algorithm's current run and also add a parameter t to the methods. Then we have to modify the code to stop the algorithm when the target node is set visited. The new class name for the abstract implementation with t-optimization is *DijkstraAbstractT*.

In listing 19 the modifications for the t-optimization are shown. We add an integer field *target* to store the target node (line 3). For the method *init(s, t)* (lines 6–9) we add a parameter t for the target node and store it in *target* (line 8).

Listing 19: Modifications for the t-optimized abstract implementation

```
1  public abstract class DijkstraAbstractT {
2      ...
3      int target
4      ...
5
6      void init(int s, int t) {
7          ...
8          target = t;
9      }
10
11     void run(int s, int t) {
12         init(s, t);
13
14         while (existsMin() && (t < 0 || t >= nodeCount || !visited[t])) {
15             outerLoop(s, t);
16         }
17
18     }
19
20     void outerLoop(int s, int t) {
21         ...
22
23         if (u == t) {
24             return;
25         }
26
27         for (int i = 0; i < nodeCount; i++) {
28             ...
29             innerLoop(s, t, u, i);
30         }
31
32     }
33
34     void innerLoop(int s, int t, int u, int i) {
35         ...
36     }
37
38 }
```

The method *run(s, t)* (lines 11–18), with which the algorithm is started, has an additional parameter *t*, too, which is propagated in the calls to *init(s, t)* (line 12) and *outerLoop(s, t)* (line 15). To stop the algorithm, when the target node is visited, we modify the loop condition of the outer loop (line 14). It now checks whether the target node is visited, but only if it is a valid node to avoid an *IndexOutOfBoundsException*. If the target node is visited, the loop terminates.

The method *outerLoop(s, t)* (lines 20–32), besides having the target node as additional parameter, checks if the node *u*, which it just set visited, is the target node, and exits in this case (lines 23–25). In line 29 the call to *innerLoop(s, t, u, i)* includes the new parameter *t*.

For the method *innerLoop(s, t, u, i)* (lines 34–36) no changes are necessary, we only included the target node as parameter, as well, for consistency reasons.

4.2 Specification

The inclusion of the t-optimization does not lead to any changes for the class invariants apart from the correction of the class name. However, the method contract for the method *init(s, t)* changes that it now sets the value of *target*

to its parameter *t*. The modifies clause is changed accordingly.

Listing 20: Modified specification for the method *init(s, t)*

```
1  /*@ public normal_behavior
2    @   ...
3    @   ensures target == t;
4    @   modifies target, visitedNodes, visited[*], reachable[*], distance[s];
5    @*/
6  void init(int s) {
7      ...
8      target = t;
9  }
```

Listing 20 shows the modified method contract for *init(s, t)*. In line 3 the new postcondition that *target* is assigned the target node is added. The modifies clause in line 4 additionally includes *target*.

Assuming the target node is a valid node and can be reached from the source node, stopping the algorithm when the target node is set visited, means that not all nodes are set reachable by the algorithm. Only those nodes are reachable that have at least one reachable predecessor with a distance smaller than that of the target node. The same holds for the minimal distances, so that for nodes there is no shorter path via a reachable node whose distance is smaller than that of the target node. Of course, for the target node the shortest path is found, as well.

However, if the target node is not a valid node or cannot be reached from the source node, its shortest path is not found, and for all other nodes the reachability and the distances are as before.

Listing 21: Modified specification for the method *run(s, t)*

```
1  /*@ public normal_behavior
2    @   ...
3    @   ensures target == t;
4    @   ...
5    @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
6    @        (t < 0 || t >= nodeCount || !reachable[t] || distance[n] < distance[t]);
7    @            (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
8    @                reachable[m]));
9    @   ...
10   @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
11   @        (t < 0 || t >= nodeCount || !reachable[t] || distance[n] < distance[t]);
12   @            (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
13   @                distance[m] <= distance[n] + weight[n][m]));
14   @   ensures (0 <= t && t < nodeCount && reachable[t]) ==>
15   @        (\forall int n; 0 <= n && n < nodeCount && reachable[n] && edge[n][t];
16   @            distance[t] <= distance[n] + weight[n][t]);
17   @   modifies target, visitedNodes, visited[*], reachable[*], distance[*];
18   @*/
19 void run(int s, int t) {
20     init(s, t);
21
22     while (existsMin() && (t < 0 || t >= nodeCount || !visited[t])) {
23         outerLoop(s, t);
24     }
25
26 }
```

The modification of *run(s, t)*'s specification can be seen in listing 21. Line 3 shows the new postcondition that *target* stores the target node, which is done in *init(s, t)*. For reachable nodes n all successors m are reachable, if n's distance is smaller than that of the target node t or t cannot be reached or is not a valid node (lines 5–8).

Lines 10–13 state that for nodes m there is no shorter path via a predecessor n, that has a smaller distance than the target node t or t cannot be reached or is not a valid node. If the target node t is a valid node and can be reached, there is no shorter path to t (lines 14–16).

The modifies clause in line 17 is changed so that it now includes *target* as well, which is modified in *init(s, t)*.

The loop invariants of *run(s, t)* are modified in a similar way. Because only the target node and nodes with a smaller distance are visited, we need to modify the invariants such that just all successors of visited nodes different from the target node are reachable. The same is done for the minimal distances.

Listing 22: Modified loop invariants for the method *run(s, t)*

```
1  /*@ loop_invariant
2    @   ...
3    @   (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != t;
4    @         (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
5    @              reachable[m])) &&
6    @   ...
7    @   (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != t;
8    @         (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
9    @              distance[m] <= distance[n] + weight[n][m])) &&
10   @   ...
11   @*/
12 while (existsMin() && (t < 0 || t >= nodeCount || !visited[t])) {
13     outerLoop(s, t);
14 }
```

Listing 22 presents the modified loop invariants for *run(s, t)*. Nodes m are reachable if they have a visited predecessor n which is not the target node t (lines 3–5). And for nodes m there is no shorter path via a visited predecessor n different from t (lines 7–9).

In the contract of the method *outerLoop(s, t)* we only need to modify the postconditions as we have done for the loop invariants of *run(s, t)*. Listing 23 shows the modifications.

Listing 23: Modified specification for the method *outerLoop(s, t)*

```
1  /*@ public normal_behavior
2    @   ...
3    @   ensures (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != t;
4    @         (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
5    @              reachable[m]));
6    @   ...
7    @   ensures (\forall int n; 0 <= n && n < nodeCount && visited[n] && n != t;
8    @         (\forall int m; 0 <= m && m < nodeCount && edge[n][m];
9    @              distance[m] <= distance[n] + weight[n][m]));
10   @   ...
11   @*/
12 void outerLoop(int s, int t) {
13     ...
14
15     if (u == t) {
16         return;
17     }
18
19     for (int i = 0; i < nodeCount; i++) {
20         ...
21         innerLoop(s, t, u, i);
22     }
23
24 }
```

Lines 3–5 state that nodes with a visited predecessor different from t are reachable. Also, for nodes there is no shorter distance via a visited predecessor which is not t (lines 7–9).

Changes in the loop invariants of *outerLoop(s, t)* or in the method contract of *innerLoop(s, t, u, i)* are not necessary.

4.3 Verification

The proofs for the four methods *init(s, t)*, *run(s, t)*, *outerLoop(s, t)* and *innerLoop(s, t, u, i)* are largly the same as without t-optimization (see section 3.3). The method *init(s, t)* can be automatically verified, again, and *innerLoop(s, t, u, i)* requires the same instantiations.

For the methods *run(s, t)* and *outerLoop(s, t)*, for which the auto mode is again run with the three different settings, most instantiations are similar as before. However, there are now more branches where quantifiers have to be instantiated by hand. Consequently, the proofs are bigger.

5 Concrete Implementation, cont'd

With the t-optimized abstract implementation specified and verified (see section 4), we can get back to the specification of the concrete implementation. However, in the abstract implementation we split the main method up into three. Therefore, we do the same with the concrete implementation.

5.1 Code

The difference to the code as we described it in section 2.1 is that we have split the method *run(s, t)* up into the three methods *run(s, t)*, *outerLoop(s, t)* and *innerLoop(s, t, u, i)* the same way as we did for the abstract implementation. Also, we added a method to the priority queue to check whether a node is contained in the queue. We need this method because of the split and for the specification, as well. So, the class *Graph* remains unchanged, *PriorityQueue* has got a new method *contains(n)* and in class *Dijkstra* we added the methods *outerLoop(s, t)* and *innerLoop(s, t, u, i)* and modified the method *run(s, t)*.

Listing 24: The modified interface of the priority queue

```
1 public class PriorityQueue {
2     ...
3     boolean contains(int n);
4     ...
5 }
```

The new method in class *PriorityQueue* is shown in listing 24. *contains(n)* (line 3) returns *true*, if the node n is contained in the queue. Otherwise, it returns *false*.

Listing 25: Modifications in *Dijkstra* for the concrete implementation

```
1  void run(int s, int t) {
2      init(s, t);
3
4      while (!queue.empty() && (t < 0 || t >= distance.length ||
5              runs[t] != counter || queue.contains(t))) {
6          outerLoop(s, t);
7      }
8
9  }
10
11 void outerLoop(int s, int t) {
12     int u = queue.deleteMin();
13     settledNodes++;
14
15     if (u == t) {
16         return;
17     }
18
19     for (int i = graph.nodes[u].firstEdge;
20             i < graph.nodes[u + 1].firstEdge; i++) {
21
22         if (!graph.edges[i].forwardFlag) {
23             continue;
24         }
25
26         innerLoop(s, t, u, i);
27     }
28
29 }
30
31 void innerLoop(int s, int t, int u, int i) {
32     relaxedEdges++;
33     int v = graph.edges[i].targetNode;
34     int d = distance[u] + graph.edges[i].weight;
35
36     if (runs[v] != counter) {
37         distance[v] = d;
38         queue.insert(v, d);
39         runs[v] = counter;
40     } else if (d < distance[v]) {
41         distance[v] = d;
42         queue.decreaseKey(v, d);
43     }
44
45 }
```

In listing 25 the three main methods are presented. In method *run(s, t)* (lines 1–9) the body of the outer loop is replaced by a call to the new method *outerLoop(s, t)* (line 6). Additionally, the loop condition is modified to stop the loop when the target node t is set visited (lines 4–5). Since the priority queue stores the nodes that are reachable and unvisited (see section 2.1.2), a node is visited if it is reachable and not in the queue. Thus, the loop condition checks whether t is in the queue, if it is reachable and a valid node.

The method *outerLoop(s, t)* (lines 11–29) represents the body of the outer loop. In line 16, when the newly set visited node u is the target node, the break-statement had to be replaced by a return-statement, because there is no loop to break. In order to stop the algorithm, we modified the loop condition of the outer loop, as previously described. As with the outer loop the body of the inner loop is replaced by a call to the new method *innerLoop(s, t, u, i)* (line 26).

The body of the inner loop is contained in the new method *innerLoop(s, t, u, i)* (lines 31–43). For this method no modification of the code was necessary.

5.2 Specification

Because the abstract and the concrete implementation implement the same algorithm, we can find correlations for key elements of the algorithm between both implementations.

In the abstract implementation *reachable[n]* denotes a reachable node n. In the concrete implementation a node n is reachable, if *runs[n]* equals *counter*. For visited nodes n in the concrete implementation, n must be reachable (*runs[n]* == *counter*) and contained in the priority queue (*contains(n)*); in the abstract implementation it suffices to check for *visited[n]*.

Checking for an edge between two nodes n and m is different in the implementations, too, due to the data structures of the graph. In the abstract implementation we only need to test *edge[n][m]*, in the concrete implementation we have to iterate over all of n's edges and look for m as target node of outgoing edges.

With the specification of the t-optimized abstract implementation at hand (see section 4.2), it would be best to use that specification and verify the concrete implementation using the correlations between both implementations. For this, there exists a feature called model fields in JML.

Model fields are used to specify the behavior of an interface without implementation. They behave as normal Java fields, however, they are only accessible in the specification. When the interface is implemented, the correlations between the implementation and the model fields is declared with so-called represents clauses. A represents clause indicates the counterpart of a model field in the implementation. With the specification (referencing the model fields), the implementation and the correlation, a prover can then verify the implementation.

Because KeY does not support model fields, especially represents clauses, yet, we have to change the specification of the abstract implementation to reference the concrete implementation. We do this using the correlations between the implementations, e.g. we translate the term *reachable[n]* (abstract) with *runs[n]* == *counter* (concrete).

Besides the translation we can use for the methods *init(s, t)*, *run(s, t)*, *outerLoop(s, t)* and *innerLoop(s, t, u, i)*, we have to specify the class invariants for class *Dijkstra* and give a full specification for the classes *Graph* and *PriorityQueue*.

5.2.1 Graph

For the graph we have to specify class invariants concerning the lengths of the used arrays, as we did for the abstract implementation, too. Aliasing cannot occur, because there are no objects or arrays of the same type. But additionally, we have to specify some constraints concerning valid values for nodes' *firstEdge* and edges' *targetNode*.

The node array *nodes* has a length of at least 1 because of the dummy node at the end of the array. The length of the edge array *edges* can be arbitrary. For a node to point to a valid edge, its *firstEdge* must have a value that lies between 0 and *edges.length*. Though *edges.length* is not a valid index of *edges*, it is a valid pointer to the first edge of a node, meaning this node (and every following node) has no edges. The first edge of the dummy node always points to *edges.length* to indicate the end of *edges*. The first edge of the first node always has the value 0.

Because the edges are ordered by their source node the first edge of a node is not greater than the first edge of the next node. The target node of edges must point to a valid node, i.e. an index of *nodes* except *nodes.length - 1*, which represents the dummy node.

Finally, we have to state that all edge weights are non-negative, and we have to provide a method contract for the method *numberOfNodes()*, which returns the number of nodes, which is the length of *nodes* minus 1 because of the dummy node.

Listing 26: Specification for the graph of the concrete implementation

```
1  public class Graph {
2      Node[] nodes;
3      Edge[] edges;
4
5      /*@ invariant nodes.length >= 1;
6        @ invariant (\forall int i; 1 <= i && i < nodes.length - 1;
7        @      0 <= nodes[i].firstEdge && nodes[i].firstEdge <= edges.length);
8        @ invariant nodes[nodes.length - 1].firstEdge == edges.length;
9        @ invariant nodes[0].firstEdge == 0;
10       @ invariant (\forall int i; 0 <= i && i < nodes.length - 1;
11       @      nodes[i].firstEdge <= nodes[i + 1].firstEdge);
12       @ invariant (\forall int i; 0 <= i && i < edges.length;
13       @      0 <= edges[i].targetNode && edges[i].targetNode < nodes.length - 1);
14       @*/
15
16     /*@ public normal_behavior
17       @  ensures \result == nodes.length - 1;
18       @*/
19     /*@ pure @*/ int numberOfNodes() {
20         return nodes.length - 1;
21     }
22
23     static class Node {
24         int firstEdge;
25     }
26
27     static class Edge {
28         int targetNode;
29         boolean forwardFlag;
30         boolean backwardFlag;
31
32         //@ invariant weight >= 0;
33         int weight;
34     }
35
36 }
```

Listing 26 shows the complete specification for the class *Graph* of the concrete implementation. The length of the array *nodes* must be at least 1 (line 5) and its values range between 0 and *edges.length* (lines 6–7). While the dummy node points for its first edge behind the array *edges* (line 8), the first node has its first edge at index 0 of *edges* (line 9).

Lines 10–11 state that the first edge of node i is not greater than that of the next node $i + 1$. The edges' target nodes must be valid nodes, i.e.

targetNode must be between 0 and *nodes.length - 1*.

The method contract for *numberOfNodes()* is shown in lines 16–19. With no preconditions it ensures that the return value equals *nodes.length - 1*. *numberOfNodes()* modifies nothing and is thus declared *pure*.

That all edge weights are non-negative, is required by line 32.

5.2.2 PriorityQueue

The number of nodes contained in the priority queue is indicated by *size* whose value is at least 0. The queue stores the contained nodes in the array *nodes* beginning at index 0 up to index *size - 1*. Thus, the length of *nodes* is at least equal to *size*. To be able to insert another node, the length must be greater than *size*. The same holds for the array *distances* in which the distances of the nodes are stored.

Because the arrays *nodes* and *distances* have the same type we have to state that they are not the same array. We also do this for other instances of class *PriorityQueue*.

Listing 27: Class invariants for the priority queue

```
1  public class PriorityQueue {
2      int[] nodes;
3      int[] distances;
4      int size;
5
6      /*@ invariant nodes.length > size;
7        @ invariant distances.length > size;
8        @ invariant nodes != distances;
9        @ invariant size >= 0;
10       @ invariant (\forall PriorityQueue q1; \created(q1);
11       @      (\forall PriorityQueue q2; \created(q2); (q1 != q2) ==>
12       @           (q1.nodes != q2.nodes && q1.distances != q2.distances &&
13       @           q1.nodes != q2.distances)));
14       @*/
15 }
```

The class invariants for the priority queue are presented in listing 27. The length of *nodes* and *distances* are greater than *size* (lines 6–7), which is not smaller than 0 (line 9).

The array *nodes* is not the same array as *distances* (line 8) and for two different instances *q1* and *q2* of type *PriorityQueue*, the arrays *nodes* and *distances* are pairwise different (lines 10–13).

The method *clear()* removes all nodes from the queue resulting in *size* being 0. To check whether the queue is empty, the method *empty()* returns *true* if *size* equals 0.

The method *contains(n)* checks whether the node *n* is contained in the queue. It returns *true*, if there is an index in the array *nodes* with the value being equal to *n*. The index must be smaller than *size*.

Listing 28 shows the method contracts for the methods *clear()*, *contains(n)* and *empty()*. The method *clear()* (lines 1–5) removes all nodes from the queue by at least setting *size* to 0 (line 2). Optionally, it can additionally modify the values of *nodes* and *distances* (line3).

Listing 28: Specification for the queue's methods *clear()*, *contains(n)* and *empty()*

```
1  /*@ public normal_behavior
2    @  ensures size == 0;
3    @  modifies size, nodes[*], distances[*];
4    @*/
5  void clear();
6
7  /*@ public normal_behavior
8    @  ensures \result == (\exists int i; 0 <= i && i < size; nodes[i] == n);
9    @*/
10 /*@ pure @*/ boolean contains(int n);
11
12 /*@ public normal_behavior
13   @  ensures \result == (size == 0);
14   @*/
15 /*@ pure @*/ boolean empty();
```

The method *contains(n)* (lines 7–10) modifies nothing (line 10) and returns *true* if an index i between 0 and *size* exists, where *nodes[i]* equals the parameter n.

The method *empty()* (lines 12–15), which is also *pure* (line 15), returns *true* if *size* is 0.

The specification of the method *deleteMin()* consists of several aspects. To delete and return the node with the smallest distance in the queue, there must be a node in the queue, i.e. *size* must be greater than 0. After the execution *size* is decremented by 1.

The returned node must be a node which was in the queue before the method call. Also, the distance of the returned node must not be greater than the distance of every other node in the queue. After the execution the returned node must not be in the queue, anymore.

Other than that, no other node can be deleted and no new node can be inserted. The distances of the remaining nodes have to be the same, also. Therefore, we declare that only a permutation is allowed. So for every node in the queue (except the returned node) before the execution, there exists an index for this node after the execution and the distance is the same as before. Furthermore, every node in the queue after the execution had to have an index with the same distance before the execution.

Listing 29: Specification for the queue's method *deleteMin()*

```
1  /*@ public normal_behavior
2    @  requires size > 0;
3    @  ensures size == \old(size) - 1;
4    @  ensures (\exists int i; 0 <= i && i < \old(size);
5    @       \result == \old(nodes[i]) && (\forall int j; 0 <= j &&
6    @       j < \old(size); \old(distances[i]) <= \old(distances[j])));
7    @  ensures (\forall int i; 0 <= i && i < \old(size) &&
8    @       \result != \old(nodes[i]); (\exists int j; 0 <= j && j < size;
9    @            \old(nodes[i]) == nodes[j] &&
10   @            \old(distances[i]) == distances[j]));
11   @  ensures (\forall int j; 0 <= j && j < size; (\exists int i; 0 <= i &&
12   @       i < \old(size); nodes[j] == \old(nodes[i]) &&
13   @       distances[j] == \old(distances[i])));
14   @  ensures !contains(\result);
15   @  modifies size, nodes[*], distances[*];
16   @*/
17 int deleteMin();
```

The method contract for *deleteMin()* can be found in listing 29. It requires *size* to be greater than 0 (line 2). Then it ensures that *size* is decremented by 1 (line 3) and the returned node is not in the queue, anymore (line 14). Lines 4–6 state that the returned node had to be at some index i and that its distance was the smallest in the queue.

The permutation is declared in lines 7–13. Lines 7–10 ensure that every node except the returned node is still in the queue at some index j with the same distance. Lines 11–13 ensure that every node was in the queue at some index i before, with the same distance.

deleteMin() modifies *size* and the values of the arrays *nodes* and *distances* (line 15).

Inserting a node n with a distance d by a call to *insert(n, d)* requires that n is not in the queue already. Also, the length of the array *nodes* and *distances* must be greater than *size*, which we already stated as class invariant. After the execution *size* is incremented by 1 and n is stored with its distance d at some index in the queue. Besides adding n only a permutation is allowed.

Listing 30: Specification for the queue's method *insert(n, d)*

```
1  /*@ public normal_behavior
2    @   requires !contains(n);
3    @   ensures size == \old(size) + 1;
4    @   ensures (\forall int i; 0 <= i && i < \old(size);
5    @        (\exists int j; 0 <= j && j < size; \old(nodes[i]) == nodes[j] &&
6    @        \old(distances[i]) == distances[j]));
7    @   ensures (\forall int j; 0 <= j && j < size && nodes[j] != n;
8    @        (\exists int i; 0 <= i && i < \old(size);
9    @             nodes[j] == \old(nodes[i]) &&
10   @             distances[j] == \old(distances[i])));
11   @   ensures (\exists int j; 0 <= j && j < size && nodes[j] == n;
12   @        distances[j] == d);
13   @   modifies size, nodes[*], distances[*];
14   @*/
15 void insert(int n, int d);
```

Listing 30 shows the method contract for *insert(n, d)*. It requires n not to be contained in the queue (line 2) and then ensures that *size* is increased by 1 (line 3). Node n can be found at some index j with the distance d (lines 11–12).

Lines 4–10 declare the permutation as for *deleteMin()*, only that every node i must still be in the queue and all nodes j except n had to be in the queue before. The method modifies *size* and the values of the arrays *nodes* and *distances* (line 13).

The last method *decreaseKey(n, d)* updates the distance in the queue for a given node n. n has to be in the queue and after the execution its distance is set to the parameter d. Further changes are only allowed for a permutation.

The contract for the method *decreaseKey(n, d)* is presented in listing 31. Line 2 requires n to be in the queue before the execution. After the execution it is still there at some index j with the new distance d (lines 10–11).

Lines 3–9 declare the permutation. All nodes i except n are still in the

Listing 31: Specification for the queue's method *decreaseKey(n, d)*

```
1  /*@ public normal_behavior
2    @  requires contains(n);
3    @  ensures (\forall int i; 0 <= i && i < \old(size) && \old(nodes[i]) != n;
4    @       (\exists int j; 0 <= j && j < size; \old(nodes[i]) == nodes[j] &&
5    @       \old(distances[i]) == distances[j]));
6    @  ensures (\forall int j; 0 <= j && j < size && nodes[j] != n;
7    @       (\exists int i; 0 <= i && i < \old(size);
8    @            nodes[j] == \old(nodes[i]) &&
9    @            distances[j] == \old(distances[i])));
10   @  ensures (\exists int j; 0 <= j && j < size && nodes[j] == n;
11   @       distances[j] == d);
12   @  modifies nodes[*], distances[*];
13   @*/
14 void decreaseKey(int n, int d);
```

queue with their old distances and all nodes j were in the queue before with their current distances. The modifies clause indicates, that the values of the arrays *nodes* and *distances* are changed (line 12).

5.2.3 Dijkstra

The class invariants for class *Dijkstra* are mostly dealing with the lengths of the arrays and aliasing.

The arrays *runs* and *distance* have a length of *graph.nodes.length − 1*, which is the graph's node count. Also the queue's arrays *nodes* and *distances* have that length to be able to insert all nodes of the graph.

The values of *runs* must not be greater than *counter*. Otherwise, by incrementing *counter* a node could become reachable in the initialization of the algorithm, while it should be unreachable.

The arrays *runs* and *distances* should not be the same array and for every instance of class *PriorityQueue* their arrays *nodes* and *distances* should be pairwise different from the arrays *runs* and *distance*.

Listing 32: Class invariants for class *Dijkstra* of the concrete implementation

```
1  public class Dijkstra {
2      Graph graph;
3      PriorityQueue queue;
4      int counter;
5      int[] runs;
6      int[] distance;
7      int settledNodes;
8      int relaxedEdges;
9      int target;
10
11     /*@ invariant runs.length == graph.nodes.length - 1;
12       @ invariant distance.length == graph.nodes.length - 1;
13       @ invariant distance != runs;
14       @ invariant (\forall int n; 0 <= n && n < distance.length;
15       @      runs[n] <= counter);
16       @*/
17
18     /*@ invariant (\forall PriorityQueue q1; \created(q1);
19       @      q1.nodes != runs && q1.distances != runs);
20       @ invariant (\forall PriorityQueue q1; \created(q1);
21       @      q1.nodes != distance && q1.distances != distance);
22       @ invariant queue.nodes.length == graph.nodes.length - 1;
23       @ invariant queue.distances.length == graph.nodes.length - 1;
24       @*/
25 }
```

The class invariants of class *Dijkstra* can be found in listing 32. The lengths of the arrays *runs*, *distance* and the queue's arrays *nodes* and *distances* are required to equal *graph.nodes.length − 1* (lines 11–12, 22–23).

All values of *runs* must not be greater than *counter* (lines 14–15). The arrays *distance* and *runs* must be different from each other (line 13) and different from the arrays *nodes* and *distances* of any instance of class *PriorityQueue* (lines 18–21).

5.2.4 init(s, t)

With the method *init(s, t)* we can begin to translate the existing specification of the t-optimized abstract implementation. The source node s must be a valid node, i.e. s must be between 0 and *distance.length*, which is the counterpart to *nodeCount* from the abstract implementation. The distance of s must be 0 and s must be reachable. In the concrete implementation s is reachable if the value of *runs* at index s equals the value of *counter*. All other nodes must be unreachable (their values of *runs* must be smaller than *counter*) and all nodes must be unvisited. The queue contains reachable nodes that are not visited, so a node is unvisited if it is unreachable or contained in the queue. *settledNodes*, which is the counterpart to *visitedNodes*, must be 0. *target* uses the same name as in the abstract implementation and must be equal to the parameter t denoting the target node. Because we don't need *relaxedEdges* for the specification, we do not include it in the postconditions.

Listing 33: Specification for the method *init(s, t)*

```
1  /*@ public normal_behavior
2    @   requires 0 <= s && s < distance.length;
3    @   ensures distance[s] == 0;
4    @   ensures runs[s] == counter;
5    @   ensures (\forall int n; 0 <= n && n < runs.length && n != s;
6    @       runs[n] < counter);
7    @   ensures (\forall int n; 0 <= n && n < runs.length;
8    @       runs[n] < counter || queue.contains(n));
9    @   ensures settledNodes == 0;
10   @   ensures target == t;
11   @   modifies counter, target, relaxedEdges, settledNodes, distance[*], runs[*],
12   @       queue.size, queue.nodes[*], queue.distances[*];
13   @*/
14 void init(int s, int t) {
15     counter++;
16     queue.clear();
17     distance[s] = 0;
18     queue.insert(s, 0);
19     runs[s] = counter;
20     target = t;
21     relaxedEdges = 0;
22     settledNodes = 0;
23 }
```

In listing 33 the contract for the method *init(s, t)* is shown. The only precondition is s being a valid node (line 2). The method ensures that only s is reachable (lines 4–6) and its distance is 0 (line 3). Lines 7–9 state that no node is visited and *settledNodes* is reset to 0 accordingly. The target node is stored in *target* (line 10). In lines 11–12 the modifies clause indicates the modification of *counter*, *target*, *relaxedEdges*, *settledNodes*, the queue's *size*

and the values of the arrays *distance* and *runs*, as well as the values of the queue's arrays *nodes* and *distances*.

5.2.5 run(s, t)

The specification for the method *run(s, t)* can be translated as we have done for the method *init(s, t)* (see section 5.2.4). Only the quantification over successors and predecessors looks slightly different.

Instead of quantifiying over each two nodes n and m with *edge[n][m]* being *true*, we quantify over each node n and all of its edges e for which *graph.edges[e].forwardFlag* is set.

For the existential quantification that each node m has a predecessor n with some properties, we have to use one more quantifier. For all nodes m exists a node n for which there exists an outgoing edge which has m as target node and n has some additional properties.

Listing 34: Specification for the method *run(s, t)*

```
1  /*@ public normal_behavior
2    @  requires 0 <= s && s < distance.length;
3    @  ensures target == t;
4    @  ensures runs[s] == counter;
5    @  ensures (\forall int n; 0 <= n && n < distance.length &&
6    @       runs[n] == counter && (t < 0 || t >= distance.length ||
7    @       runs[t] != counter || distance[n] < distance[t]);
8    @            (\forall int e; graph.nodes[n].firstEdge <= e &&
9    @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
10   @                 runs[graph.edges[e].targetNode] == counter));
11   @  ensures distance[s] == 0;
12   @  ensures (\forall int m; 0 <= m && m < distance.length &&
13   @       runs[m] == counter && m != s;
14   @            (\exists int n; 0 <= n && n < distance.length && runs[n] == counter;
15   @                 (\exists int e; graph.nodes[n].firstEdge <= e &&
16   @                 e < graph.nodes[n + 1].firstEdge &&
17   @                 graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
18   @                      distance[m] == distance[n] + graph.edges[e].weight)));
19   @  ensures (\forall int n; 0 <= n && n < distance.length &&
20   @       runs[n] == counter && (t < 0 || t >= distance.length ||
21   @       runs[t] != counter || distance[n] < distance[t]);
22   @            (\forall int e; graph.nodes[n].firstEdge <= e &&
23   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
24   @                 distance[graph.edges[e].targetNode] <= distance[n] +
25   @                 graph.edges[e].weight));
26   @  ensures (0 <= t && t < distance.length && runs[t] == counter) ==>
27   @       (\forall int n; 0 <= n && n < distance.length && runs[n] == counter;
28   @            (\forall int e; graph.nodes[n].firstEdge <= e &&
29   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag &&
30   @            graph.edges[e].targetNode == t;
31   @                 distance[t] <= distance[n] + graph.edges[e].weight));
32   @  modifies counter, target, relaxedEdges, settledNodes, distance[*], runs[*],
33   @       queue.size, queue.nodes[*], queue.distances[*];
34   @*/
35 void run(int s, int t) {
36     init(s, t);
37
38     while (!queue.empty() && (t < 0 || t >= distance.length ||
39             runs[t] != counter || queue.contains(t))) {
40         outerLoop(s, t);
41     }
42
43 }
```

The method contract for *run(s, t)* is shown in listing 34. It requires the parameter s to be a valid node (line 2) and ensures that the target node is stored in *target* (line 3).

The source node s is reachable (line 4) and the target nodes of outgoing

edges e originating from reachable nodes n are reachable, too, if n's distance is smaller than the distance of the target node t or t is not reachable or not a valid node (lines 5–10). Simplified, successors of reachable nodes with a distance smaller than that of the target node are reachable, as well.

Line 11 states that the distance of the source node is 0. The distances of other reachable nodes m is computed via reachable predecessors n. So for each reachable node m except s there exists a predecessor n with an outgoing edge e such that m's distance is equal to n's distance plus e's weight (lines 12–18).

Also to a node there is no shorter path via a node n which has a smaller distance than the target node. So for each node n with a distance smaller than that of t and for each of n's outgoing edges e the distance of e's target node is not greater than the distance via n and e (lines 19–25).

If the target node t is reachable and a valid node, there is no shorter path to it (lines 26–31). As the method *init(s, t)*, *run(s, t)* modifies *counter*, *target*, *relaxedEdges*, *settledNodes*, the queue's *size* and the values of the arrays *distance* and *runs*, as well as the values of the queue's arrays *nodes* and *distances*.

Listing 35: Loop invariants for the method *run(s, t)*

```
1  /*@ loop_invariant
2    @  runs[s] == counter &&
3    @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter &&
4    @       !queue.contains(n) && n != t;
5    @            (\forall int e; graph.nodes[n].firstEdge <= e &&
6    @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
7    @                 runs[graph.edges[e].targetNode] == counter)) &&
8    @  distance[s] == 0 &&
9    @  (\forall int m; 0 <= m && m < distance.length && runs[m] == counter &&
10   @       m != s;
11   @            (\exists int n; 0 <= n && n < distance.length &&
12   @            runs[n] == counter && !queue.contains(n);
13   @                 (\exists int e; graph.nodes[n].firstEdge <= e &&
14   @                 e < graph.nodes[n + 1].firstEdge &&
15   @                 graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
16   @                      distance[m] == distance[n] + graph.edges[e].weight))) &&
17   @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter &&
18   @       !queue.contains(n) && n != t;
19   @            (\forall int e; graph.nodes[n].firstEdge <= e &&
20   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
21   @                 distance[graph.edges[e].targetNode] <= distance[n] +
22   @                 graph.edges[e].weight)) &&
23   @  (\forall int n; 0 <= n && n < distance.length && queue.contains(n);
24   @       runs[n] == counter) &&
25   @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter;
26   @       distance[n] >= 0) &&
27   @  (\forall int m; 0 <= m && m < distance.length && runs[m] == counter &&
28   @       !queue.contains(m);
29   @            (\forall int n; 0 <= n && n < distance.length &&
30   @            runs[n] == counter && queue.contains(n);
31   @                 distance[m] <= distance[n])) &&
32   @  0 <= settledNodes && settledNodes <= distance.length;
33   @ decreases distance.length - settledNodes;
34   @ modifies relaxedEdges, settledNodes, distance[*], runs[*], queue.size,
35   @      queue.nodes[*], queue.distances[*];
36   @*/
37 while (!queue.empty() && (t < 0 || t >= distance.length ||
38          runs[t] != counter || queue.contains(t))) {
39     outerLoop(s, t);
40 }
```

The loop invariants can be translated the same way. *visited[n]* is here translated with *runs[n] == counter && !queue.contains(n)*. Because visited is defined by being reachable and not in the queue, we do not need

the invariant from the abstract implementation that visited nodes are also reachable. In exchange, we need a new invariant stating that if a node is in the queue, that node is reachable.

Listing 35 shows the loop invariants of the method *run(s, t)*. The source node is reachable (line 2) and successors of visited nodes different from the target node t are reachable, too (lines 3–7).

The distance of the source node is 0 (line 8) and the distances of other nodes are computed via a visited predecessor (lines 9–16).

To a node there is no shorter path via a visited node n different from t (lines 17–22), and all nodes contained in the queue are reachable (lines 23–24). Also all distances are at least 0 (lines 25–26) and every visited node has a distance that is not greater than the distance of any unvisited node (27–31).

For the termination the value of *settledNodes* lies between 0 and the number of nodes, which equals *distance.length* (line 32). The variant is *distance.length* − *settledNodes*, i.e. the number of unvisited nodes (line 33), and the loop modifies *relaxedEdges*, *settledNodes*, the queue's *size* and the values of the arrays *distance* and *runs*, as well as the values of the queue's arrays *nodes* and *distances*.

5.2.6 outerLoop(s, t)

For the method *outerLoop(s, t)* the specification can also be translated as before and is in parts identical to the loop invariants of *run(s, t)*.

Listing 36: Specification for the method *outerLoop(s, t)*

```
1  /*@ public normal_behavior
2    @  requires !queue.empty();
3    @  requires 0 <= settledNodes && settledNodes < distance.length;
4    @  requires runs[s] == counter;
5    @  requires (\forall int n; 0 <= n && n < distance.length &&
6    @       runs[n] == counter && !queue.contains(n);
7    @            (\forall int e; graph.nodes[n].firstEdge <= e &&
8    @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
9    @                runs[graph.edges[e].targetNode] == counter));
10   @  requires distance[s] == 0;
11   @  requires (\forall int m; 0 <= m && m < distance.length &&
12   @       runs[m] == counter && m != s;
13   @            (\exists int n; 0 <= n && n < distance.length &&
14   @            runs[n] == counter && !queue.contains(n);
15   @                (\exists int e; graph.nodes[n].firstEdge <= e &&
16   @                e < graph.nodes[n + 1].firstEdge &&
17   @                graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
18   @                    distance[m] == distance[n] + graph.edges[e].weight)));
19   @  requires (\forall int n; 0 <= n && n < distance.length &&
20   @       runs[n] == counter && !queue.contains(n);
21   @            (\forall int e; graph.nodes[n].firstEdge <= e &&
22   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
23   @                distance[graph.edges[e].targetNode] <= distance[n] +
24   @                graph.edges[e].weight));
25   @  requires (\forall int n; 0 <= n && n < distance.length &&
26   @       queue.contains(n); runs[n] == counter);
27   @  requires (\forall int n; 0 <= n && n < distance.length &&
28   @       runs[n] == counter; distance[n] >= 0);
29   @  requires (\forall int m; 0 <= m && m < distance.length &&
30   @       runs[m] == counter && !queue.contains(m);
31   @            (\forall int n; 0 <= n && n < distance.length &&
32   @            runs[n] == counter && queue.contains(n);
33   @                distance[m] <= distance[n]));
34   @  ensures runs[s] == counter;
35   @  ensures (\forall int n; 0 <= n && n < distance.length &&
```

```
36    @       runs[n] == counter && !queue.contains(n) && n != t;
37    @           (\forall int e; graph.nodes[n].firstEdge <= e &&
38    @           e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
39    @               runs[graph.edges[e].targetNode] == counter));
40    @   ensures distance[s] == 0;
41    @   ensures (\forall int m; 0 <= m && m < distance.length &&
42    @       runs[m] == counter && m != s;
43    @           (\exists int n; 0 <= n && n < distance.length &&
44    @           runs[n] == counter && !queue.contains(n);
45    @               (\exists int e; graph.nodes[n].firstEdge <= e &&
46    @               e < graph.nodes[n + 1].firstEdge &&
47    @               graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
48    @                   distance[m] == distance[n] + graph.edges[e].weight)));
49    @   ensures (\forall int n; 0 <= n && n < distance.length &&
50    @       runs[n] == counter && !queue.contains(n) && n != t;
51    @           (\forall int e; graph.nodes[n].firstEdge <= e &&
52    @           e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
53    @               distance[graph.edges[e].targetNode] <= distance[n] +
54    @               graph.edges[e].weight));
55    @   ensures (\forall int n; 0 <= n && n < distance.length &&
56    @       queue.contains(n); runs[n] == counter);
57    @   ensures (\forall int n; 0 <= n && n < distance.length &&
58    @       runs[n] == counter; distance[n] >= 0);
59    @   ensures (\forall int m; 0 <= m && m < distance.length &&
60    @       runs[m] == counter && !queue.contains(m);
61    @           (\forall int n; 0 <= n && n < distance.length &&
62    @           runs[n] == counter && queue.contains(n);
63    @               distance[m] <= distance[n]));
64    @   ensures settledNodes <= distance.length;
65    @   ensures settledNodes > \old(settledNodes);
66    @   modifies relaxedEdges, settledNodes, distance[*], runs[*], queue.size,
67    @       queue.nodes[*], queue.distances[*];
68    @*/
69  void outerLoop(int s, int t) {
70      int u = queue.deleteMin();
71      settledNodes++;
72
73      if (u == t) {
74          return;
75      }
76
77      for (int i = graph.nodes[u].firstEdge;
78              i < graph.nodes[u + 1].firstEdge; i++) {
79
80          if (!graph.edges[i].forwardFlag) {
81              continue;
82          }
83
84          innerLoop(s, t, u, i);
85      }
86
87  }
```

Listing 36 shows the complete method contract for the method *outerLoop(s, t)*. The priority queue is not allowed to be empty and the value of *settledNodes* must lie between 0 and *distance.length* (lines 2–3).

The source node s must be reachable (line 4) and successors of visited nodes different from the target node t are reachable, as well (lines 5–9). The distance of s is 0 (line 10) and the distances of all other nodes are computed via a visited node (lines 11–18). Again, to a node there is no shorter path via a visited node different from t (lines 19–24).

All nodes contained in the queue are reachable (lines 25–26), all distances are not smaller than 0 (lines 27–28) and the distances of visited nodes are not greater than that of an unvisited node (lines 29–33).

Those preconditions of lines 4–33 are also there as postconditions in lines 34–63. Additionally, *settledNodes* will be increased, but it will not be greater than *distance.length* (lines 64–65), because it is only incremented by 1. The modifies clause of lines 66–67 holds the same locations as the modifies clause of the outer loop in *run(s, t)*.

For the loop invariants, there are some differences between the implementations. While in the abstract implementation the loop iterated over all nodes i and checked for an edge between nodes u and i, here, due to the data structure of the graph, the loop only iterates over all edges i of node u and checks if edge i is an outgoing edge. Besides the different bounds of i, the target node of an edge, which was represented by i in the abstract implementation, is represented here by *graph.edges[i].targetNode*, which results in a slightly different specification. The rest of the specification, however, can be translated as usual.

Listing 37: Invariants for the method *outerLoop(s, t)*

```
1  /*@ loop_invariant
2    @  runs[s] == counter &&
3    @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter &&
4    @       !queue.contains(n) && n != u;
5    @            (\forall int e; graph.nodes[n].firstEdge <= e &&
6    @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
7    @                 runs[graph.edges[e].targetNode] == counter)) &&
8    @  (\forall int e; graph.nodes[u].firstEdge <= e && e < i &&
9    @       graph.edges[e].forwardFlag;
10   @            runs[graph.edges[e].targetNode] == counter) &&
11   @  distance[s] == 0 &&
12   @  (\forall int m; 0 <= m && m < distance.length && runs[m] == counter &&
13   @       m != s;
14   @            (\exists int n; 0 <= n && n < distance.length &&
15   @            runs[n] == counter && !queue.contains(n);
16   @                 (\exists int e; graph.nodes[n].firstEdge <= e &&
17   @                 e < graph.nodes[n + 1].firstEdge &&
18   @                 graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
19   @                      distance[m] == distance[n] + graph.edges[e].weight))) &&
20   @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter &&
21   @       !queue.contains(n) && n != u;
22   @            (\forall int e; graph.nodes[n].firstEdge <= e &&
23   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
24   @                 distance[graph.edges[e].targetNode] <= distance[n] +
25   @                 graph.edges[e].weight)) &&
26   @  (\forall int e; graph.nodes[u].firstEdge <= e && e < i &&
27   @       graph.edges[e].forwardFlag;
28   @            distance[graph.edges[e].targetNode] <= distance[u] +
29   @            graph.edges[e].weight) &&
30   @  (\forall int m; 0 <= m && m < distance.length && runs[m] == counter &&
31   @       !queue.contains(m);
32   @            (\forall int n; 0 <= n && n < distance.length &&
33   @            runs[n] == counter && queue.contains(n);
34   @                 distance[m] <= distance[n])) &&
35   @  (\forall int n; 0 <= n && n < distance.length && queue.contains(n);
36   @       runs[n] == counter) &&
37   @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter;
38   @       distance[n] >= 0) &&
39   @  (\forall int n; 0 <= n && n < distance.length && runs[n] == counter &&
40   @       !queue.contains(n); distance[n] <= distance[u]) &&
41   @  graph.nodes[u].firstEdge <= i && i <= graph.nodes[u + 1].firstEdge;
42   @ decreases graph.nodes[u + 1].firstEdge - i;
43   @ modifies relaxedEdges, distance[*], runs[*], queue.size, queue.nodes[*],
44   @      queue.distances[*];
45   @*/
```

The loop invariants for *outerLoop(s, t)* can be found in listing 37. The source node is reachable (line 2) and all nodes with a visited predecessor which is not u are visited (lines 3–7). Also, the target nodes of u's outgoing edges up to edge i are reachable (lines 8–10).

The distance of the source node is 0 (line 11) and the distance of every other node is computed via a visited predecessor (lines 12–19). Again, to a node there is no shorter path via a visited predecessor which is different from u (lines 20–25). And for all target nodes of u's outgoing edges up to edge i there is no shorter path via u (lines 26–29).

The distances of visited nodes are not greater than the distances of unvisited nodes (lines 30–34), nodes which are contained in the queue are also reachable (lines 35–36), and all distances are not smaller than 0 (lines 37–38). Also, the distances of visited nodes are not greater than the distance of u (lines 39–40).

The value of the loop variable i ranges from u's first edge to the first edge of the next node, which is $u + 1$ (line 41). The variant for the inner loop is $graph.nodes[u + 1].firstEdge - i$ (line 42) and the modifies clause is given in lines 43–44.

5.2.7 innerLoop(s, t, u, i)

As for the loop invariants of *outerLoop(s, t)* the meaning of the parameter i for *innerLoop(s, t, u, i)* has changed in this implementation, from the target node of an edge to the edge itself. Therefore, the translation of the specification results in a slightly different method contract.

Listing 38: Specification for the method *innerLoop(s, t, u, i)*

```
1  /*@ public normal_behavior
2    @  requires 0 <= u && u < distance.length;
3    @  requires graph.nodes[u].firstEdge <= i && i < graph.nodes[u + 1].firstEdge;
4    @  requires graph.edges[i].forwardFlag;
5    @  requires runs[u] == counter && !queue.contains(u);
6    @  requires runs[s] == counter;
7    @  requires (\forall int n; 0 <= n && n < distance.length &&
8    @       runs[n] == counter && !queue.contains(n) && n != u;
9    @            (\forall int e; graph.nodes[n].firstEdge <= e &&
10   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
11   @                 runs[graph.edges[e].targetNode] == counter));
12   @  requires distance[s] == 0;
13   @  requires (\forall int m; 0 <= m && m < distance.length &&
14   @       runs[m] == counter && m != s;
15   @            (\exists int n; 0 <= n && n < distance.length &&
16   @            runs[n] == counter && !queue.contains(n);
17   @                 (\exists int e; graph.nodes[n].firstEdge <= e &&
18   @                 e < graph.nodes[n + 1].firstEdge &&
19   @                 graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
20   @                      distance[m] == distance[n] + graph.edges[e].weight)));
21   @  requires (\forall int n; 0 <= n && n < distance.length &&
22   @       runs[n] == counter && !queue.contains(n) && n != u;
23   @            (\forall int e; graph.nodes[n].firstEdge <= e &&
24   @            e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
25   @                 distance[graph.edges[e].targetNode] <= distance[n] +
26   @                 graph.edges[e].weight));
27   @  requires (\forall int n; 0 <= n && n < distance.length &&
28   @       queue.contains(n); runs[n] == counter);
29   @  requires (\forall int n; 0 <= n && n < distance.length &&
30   @       runs[n] == counter; distance[n] >= 0);
31   @  requires (\forall int n; 0 <= n && n < distance.length &&
32   @       runs[n] == counter && !queue.contains(n); distance[n] <= distance[u]);
33   @  requires (\forall int m; 0 <= m && m < distance.length &&
34   @       runs[m] == counter && !queue.contains(m);
35   @            (\forall int n; 0 <= n && n < distance.length &&
36   @            runs[n] == counter && queue.contains(n);
37   @                 distance[m] <= distance[n]));
38   @  ensures runs[graph.edges[i].targetNode] == counter;
39   @  ensures !queue.contains(graph.edges[i].targetNode) ==>
40   @       distance[graph.edges[i].targetNode] ==
41   @       \old(distance[graph.edges[i].targetNode]);
42   @  ensures distance[s] == 0;
43   @  ensures (\forall int m; 0 <= m && m < distance.length &&
44   @       runs[m] == counter && m != s;
45   @            (\exists int n; 0 <= n && n < distance.length &&
46   @            runs[n] == counter && !queue.contains(n);
47   @                 (\exists int e; graph.nodes[n].firstEdge <= e &&
48   @                 e < graph.nodes[n + 1].firstEdge &&
49   @                 graph.edges[e].targetNode == m && graph.edges[e].forwardFlag;
50   @                      distance[m] == distance[n] + graph.edges[e].weight)));
51   @  ensures (\forall int n; 0 <= n && n < distance.length &&
```

```
52    @       runs[n] == counter && !queue.contains(n) && n != u;
53    @           (|forall int e; graph.nodes[n].firstEdge <= e &&
54    @           e < graph.nodes[n + 1].firstEdge && graph.edges[e].forwardFlag;
55    @               distance[graph.edges[e].targetNode] <= distance[n] +
56    @               graph.edges[e].weight));
57    @  ensures distance[graph.edges[i].targetNode] <= distance[u] +
58    @       graph.edges[i].weight;
59    @  ensures (|forall int n; 0 <= n && n < distance.length &&
60    @       queue.contains(n); runs[n] == counter);
61    @  ensures (|forall int m; 0 <= m && m < distance.length &&
62    @       runs[m] == counter && !queue.contains(m);
63    @           (|forall int n; 0 <= n && n < distance.length &&
64    @           runs[n] == counter && queue.contains(n);
65    @               distance[m] <= distance[n]));
66    @  modifies relaxedEdges, distance[graph.edges[i].targetNode],
67    @       runs[graph.edges[i].targetNode], queue.size, queue.nodes[*],
68    @       queue.distances[*];
69    @*/
70  void innerLoop(int s, int t, int u, int i) {
71      relaxedEdges++;
72      int v = graph.edges[i].targetNode;
73      int d = distance[u] + graph.edges[i].weight;
74
75      if (runs[v] != counter) {
76          distance[v] = d;
77          queue.insert(v, d);
78          runs[v] = counter;
79      } else if (d < distance[v]) {
80          distance[v] = d;
81          queue.decreaseKey(v, d);
82      }
83
84  }
```

The method contract of *innerLoop(s, t, u, i)* is presented in listing 38. u must be a valid node (line 2) and i must be an edge of u that is outgoing (lines 3–4). u must also be visited (line 5).

The source node must be reachable (line 6) and every other node which has a visited predecessor that is not u is reachable, as well (lines 7–11). The distance of the source node is 0 (line 12) and the distances of the other nodes are computed via a visited predecessor (lines 13–20). Also, to a node there is no shorter path via a visited predecessor different from u (lines 21–26).

All nodes in the queue must also be reachable (lines 27–28) and all distances are not smaller than 0 (lines 29–30). The distances of visited nodes are not greater than the distance of u (lines 31–32) and they are not greater than the distances of unvisited nodes (lines 33–37).

With these preconditions the method ensures that the target node of edge i is reachable (line 38) and if that target node is visited, its distance is not changed (lines 39–41). Then the distance of the source node is again 0 (line 42) and the distances of the other nodes are computed via a visited predecessor (lines 43–50).

To a node there is no shorter path via a visited predecessor different from u (lines 51–56) and the distance of the target node of edge i is not greater than the distance via u (lines 57–58).

All nodes contained in the queue are reachable, as well (lines 59–60) and the distances of visited nodes are not greater than the distances of unvisited nodes (lines 61–65). The modifies clause is given in lines 66–68 and includes the values of the arrays *runs* and *distance* at the index of the target node of edge i, as well as *relaxedEdges* and the queue's fields.

5.3 Verification

The verification of this implementation is considerably more difficult than the verification of the abstract implementation. Though the specification is more or less the same, the use of the different data structure leads to an increase of complexity, as the following example demonstrates.

The expression *distance[graph.edges[i].targetNode]* appears (with the help of the local variable v) in the method *innerLoop(s, t, u, i)*. Checking this expression for the occurrence of an *IndexOutOfBoundsException* requires to check the subexpression *graph.edges[i]* first. Regarding the bounds of i we have given as a precondition that i lies between *graph.nodes[u].firstNode* and *graph.nodes[u + 1].firstNode*. We have to deduce with the help of the invariants of the graph that i is in fact a valid index for the array *edges*. We can then show that the complete expression will not raise an *IndexOutOfBoundsException*, because an edge's *targetNode* is always a valid index for the array *distances*. In the abstract implementation, however, the corresponding expression is *distance[i]* for which a check is trivial.

The use of the priority queue is another major contributor to the complexity. Every method call leads to several new goals (*Pre*, *Post* and *Exceptional Post*), which increase the number of needed steps for the prover.

As we will illustrate in detail in section 7.2, KeY has performance issues with more complex proof obligations. With the proof tree becoming larger and larger, the performance decreases rapidly, i.e. the time KeY needs for a rule application increases. Eventually, KeY crashes with an *OutOfMemoryError*. To reduce the number of nodes in the proof we can further limit the rules that KeY is allowed to apply automatically. This avoids unnecessary rule applications, especially those that split the proof. However, this makes it much more difficult for the user, who has to do a lot more rule applications by hand.

For the method *init(s, t)* this is not yet an issue. With some user interaction the method can be verified using our default settings. For the other methods, however, we were not able to find a proof due to those reasons. We only succeeded in the verification of the method *innerLoop(s, u, i)* in a version of the concrete implementation without t-optimization with respect to an earlier specification. Because the methods *run(s, t)* and *outerLoop(s, t)* had had considerably more complex proofs than *innerLoop(s, t, u, i)*, we stopped trying to verify the concrete implementation.

6 Variations of the Algorithm

In this section we shall take a look at two further variants of Dijkstra's Algorithm. So far we verified the standard algorithm and the variant with t-optimization in abstract implementations. Because of the problems in verifying the concrete implementation (see section 5.3), we will present the new

variants in an abstract implementation only. The first variant is a bidirectional version of the algorithm allowing backward searches. The second variant is an optimization with so-called Arc Flags, which requires some precalculation but provides a faster search. For both variants, we build upon the abstract implementation with t-optimization from section 4.

6.1 Bidirectional

In the standard algorithm we only have forward search. This means, we start from the source node and follow outgoing edges to compute the distance from the source node to the target node (or every other node when the target node is omitted). If the graph is undirected, an edge between two nodes has the same weight in both directions, so the distance from the target node to the source node is equal to the distance from the source node to the target node. A backward search is here the same as a forward search regarding the distance, only with inverted meanings of source node and target node.

In directed graphs, however, the distance from the target node to the source node can be different. This distance can be computed by swapping the target node and the source node or by performing a backward search. The backward search begins at the source node – which now has the meaning of the target node – and follows incoming edges. So, the distance to the source node is computed.

Although swapping the nodes is easier than implementing a backward search, the latter is needed in some situations. If the distance from every node to a certain target node n is to be computed, the forward search cannot be used, since there is no source node which, however, is required. Therefore, a backward search must be used with n as source node and no target node, which computes the distance to n from each node. For example, the backward search is used for the precalculation in the variant with Arc Flags (see section 6.2).

In true bidirectional search algorithms the shortest distance from a source node n to a target node m is computed by starting a forward search from n and a backward search from m at the same time ([Poh71]). The simple variant we present here, however, can only start a forward or a backward search selectable through an additional parameter.

6.1.1 Code

The implementation of the backward search is simple. Instead of following outgoing edges from a node u to nodes i, represented in the implementation by *edge[u][i]*, we follow incoming edges from nodes i to a node u, represented by *edge[i][u]*. Thus we only need to flip the edges and use the corresponding edge weights. To be able to use both forward search and backward search, we add a boolean parameter *forward* to the methods' signatures. It

Listing 39: Modifications for the bidirectional variant

```
1  public abstract class DijkstraAbstractBi {
2      ...
3
4      void init(int s, int t, boolean forward) {
5          ...
6      }
7
8      void run(int s, int t, boolean forward) {
9          init(s, t, forward);
10
11         while (existsMin() && (t < 0 || t >= nodeCount || !visited[t])) {
12             outerLoop(s, t, forward);
13         }
14
15     }
16
17     void outerLoop(int s, int t, boolean forward) {
18         ...
19
20         for (int i = 0; i < nodeCount; i++) {
21
22             if (!(forward ? edge[u][i] : edge[i][u])) {
23                 continue;
24             }
25
26             innerLoop(s, t, forward, u, i);
27         }
28
29     }
30
31     void innerLoop(int s, int t, boolean forward, int u, int i) {
32         int d = distance[u] + (forward ? weight[u][i] : weight[i][u]);
33         ...
34     }
35
36 }
```

indicates which search should be performed. Depending on *forward* we use either outgoing or incoming edges, which we implemented with the help of if expressions.

Listing 39 presents the modifications to the t-optimized abstract implementation for a bidirectional search. The Java class is now called *DijkstraAbstractBi* (line 1). The four methods *init(s, t, forward)*, *run(s, t, forward)*, *outerLoop(s, t, forward)* and *innerLoop(s, t, forward, u, i)* have an additional parameter *forward* and the method calls are modified accordingly (lines 4–17, 26, 31). *outerLoop(s, t, forward)* now checks in the inner loop for an outgoing or incoming edge depending on *forward* (line 22). Also depending on *forward* the method *innerLoop(s, t, forward, u, i)* uses the corresponding edge weight (line 32).

6.1.2 Specification

The specification is modified in the same way. We replace occurrences of *edge[x][y]* and *weight[x][y]* with corresponding if expressions. We present the modified specification only for the method contract of *run(s, t, forward)*, the remaining specification is modified identically. The class invariants are not modified other than to use the new class name.

The modified specification for the method *run(s, t, forward)* is shown in

Listing 40: Modified specification for the method *run(s, t, forward)*

```
1  /*@ public normal_behavior
2    @   ...
3    @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
4    @        (t < 0 || t >= nodeCount || !reachable[t] || distance[n] < distance[t]);
5    @            (\forall int m; 0 <= m && m < nodeCount &&
6    @            (forward ? edge[n][m] : edge[m][n]);
7    @                reachable[m]));
8    @   ...
9    @   ensures (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
10   @        (\exists int n; 0 <= n && n < nodeCount && reachable[n] &&
11   @        (forward ? edge[n][m] : edge[m][n]);
12   @            distance[m] == distance[n] +
13   @            (forward ? weight[n][m] : weight[m][n])));
14   @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
15   @        (t < 0 || t >= nodeCount || !reachable[t] || distance[n] < distance[t]);
16   @            (\forall int m; 0 <= m && m < nodeCount &&
17   @            (forward ? edge[n][m] : edge[m][n]);
18   @                distance[m] <= distance[n] +
19   @                (forward ? weight[n][m] : weight[m][n])));
20   @   ensures (0 <= t && t < nodeCount && reachable[t]) ==>
21   @        (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
22   @        (forward ? edge[n][t] : edge[t][n]);
23   @            distance[t] <= distance[n] +
24   @            (forward ? weight[n][t] : weight[t][n]));
25   @   ...
26   @*/
27 void run(int s, int t, boolean forward) {
28     init(s, t, forward);
29
30     while (existsMin() && (t < 0 || t >= nodeCount || !visited[t])) {
31         outerLoop(s, t, forward);
32     }
33
34 }
```

listing 40. Successors or predecessors (depending on *forward*) of reachable nodes that have smaller distances than the target node are reachable, too (lines 3–7). The distances of all nodes except the source node are computed via a reachable predecessor (in a forward search) or successor (in a backward search) (lines 9–13).

For nodes there is no shorter path via a reachable predecessor or successor (depending on *forward*) that has a distance smaller than the target node (lines 14–19). Lines 20–24 ensure that if the target node is reachable, there is no shorter path to the target node.

6.1.3 Verification

In comparison to the verification of the t-optimized abstract implementation (see section 4.3), the proofs for the four methods are considerably bigger and require much more user interaction. This is due to the case distinction for the value of *forward*. Only the proof for *init(s, t, forward)* is almost the same, since its specification was not modified.

6.2 With Precalculation (Arc Flags)

To reduce the effort needed for a computation of a shortest path we can do some precalculation. Of course, this only pays off if enough queries are made

where the precalculated data can be used. In general, when the graph is static, this is the case.

In an extreme case we could compute the shortest distances from and to all nodes. When a search is started, we only need to look up the right distance. However, this costs much memory ($O(n^2)$) to store the computed information.

We use the approach presented in [Lau04], which partitions the graph into regions and computes for each edge e and each region r whether e is part of a shortest path into r. If e is not part of a shortest path into r we do not need to consider this edge e when searching for a shortest path to a target node belonging to region r.

In the precalculation we run a backward search for the nodes of every region. If an edge e is used in a shortest path to a node in region r, we set its arc flag for r. Otherwise, its arc flag for r remains unset. In the search for the shortest distance to some node in region r we only consider edges with an arc flag set for r. Opposed to the t-optimized implementation, the target node must here be a valid node, because we need the target region.

An inner, or non-boundary, node is a node which belongs to region r and has only predecessors that belong to r, as well. A path from a different region to an inner node will always go through a boundary node. Therefore, in the precalculation it suffices to compute the shortest paths to only boundary nodes of each region. Additionally, for each edge that connects nodes of the same region r we have to set its arc flag for r. Otherwise, an inner node would not be reachable.

How the graph is partitioned only has an impact on the amount of precalculation and on the performance of the algorithm. In the worst case the algorithm performs as the standard t-optimized algorithm – here all arc flags are set – and the precalculation was wasted. The only requirement is that every node belongs to exactly one region.

6.2.1 Code

For the implementation we build upon the t-optimized abstract implementation from section 4 and change the class name to *DijkstraAbstractArc*. As with the nodes, regions only exist implicitly through the declaration of *regionCount*, which represents the number of regions. The array *region* assigns each node a region, identified by a number between 0 and *regionCount*. The array *boundary* indicates whether a node is a boundary node or an inner node. Inner nodes have no predecessor in a different region.

Because we identify edges by their source and target node, we store the arc flags for each edge and each region in the 3-dimensional boolean array *arcFlag*. The edge between two nodes n and m has its arc flag for region r set if *arcFlag[n][m][r]* is *true*. For the algorithm to respect the arc flags, i.e. only consider edges where the arc flag is set, we do not call *innerLoop(s, t,*

u, i) in *outerLoop(s, t)* if the arc flag of the edge between u and i for the target region (*region[t]*) is not set.

The precalculation is done by the method *setArcFlags()*. It first computes for each node whether it is a boundary node and also sets the arc flag for edges between two nodes of the same region.

In the second part *setArcFlags()* performs a backward search to each boundary node v and sets the arc flag for v's region for every edge that is used in a shortest path to v.

The backward search is done by the bidirectional abstract implementation of section 6.1, whose four methods *init(s, t, forward)*, *run(s, t, forward)*, *outerLoop(s, t, forward)* and *innerLoop(s, t, forward, u, i)* are copied into this implementation. To distinguish them from the methods of the arc-flag-optimized algorithm and to mark them as being used in the precalculation, we modify their names by appending the suffix *Pre*.

Listing 41: Modifications for the variant with arc flags

```
1  public abstract class DijkstraAbstractArc {
2      ...
3      int regionCount;
4      int[] region;
5      boolean[][][] arcFlag;
6      boolean[] boundary;
7
8      void setArcFlags() {
9
10         for (int i = 0; i < nodeCount; i++) {
11             boundary[i] = false;
12         }
13
14         for (int n = 0; n < nodeCount; n++) {
15
16             for (int m = 0; m < nodeCount; m++) {
17
18                 if (!edge[n][m]) {
19                     continue;
20                 }
21
22                 for (int t = 0; t < regionCount; t++) {
23                     arcFlag[n][m][t] = false;
24                 }
25
26                 if (region[n] != region[m]) {
27                     boundary[m] = true;
28                 } else {
29                     arcFlag[n][m][region[n]] = true;
30                 }
31
32             }
33
34         }
35
36         for (int v = 0; v < nodeCount; v++) {
37
38             if (!boundary[v]) {
39                 continue;
40             }
41
42             runPre(v, -1, false);
43
44             for (int n = 0; n < nodeCount; n++) {
45
46                 for (int m = 0; m < nodeCount; m++) {
47
48                     if (!edge[n][m]) {
49                         continue;
50                     }
51
52                     if (reachable[n] && reachable[m] &&
53                             distance[n] - distance[m] == weight[n][m]) {
54                         arcFlag[n][m][region[v]] = true;
```

```
55                          }
56
57                      }
58
59                  }
60
61              }
62
63          }
64
65          ...
66
67          void run(int s, int t) {
68              init(s, t);
69
70              while (existsMin() && !visited[t]) {
71                  outerLoop(s, t);
72              }
73
74          }
75
76          void outerLoop(int s, int t) {
77              ...
78
79              for (int i = 0; i < nodeCount; i++) {
80
81                  if (!edge[u][i]) {
82                      continue;
83                  }
84
85                  if (!arcFlag[u][i][region[t]]) {
86                      continue;
87                  }
88
89                  innerLoop(s, t, u, i);
90              }
91
92          }
93
94          ...
95
96          void initPre(int s, int t, boolean forward);
97          void runPre(int s, int t, boolean forward);
98          void outerLoopPre(int s, int t, boolean forward);
99          void innerLoopPre(int s, int t, boolean forward, int u, int i);
100     }
```

Listing 41 shows the modified code for the arc-flag-optimized implementation. The number of regions is stored in *regionCount* (line 3) and the array *region* indicates to which region a node belongs (line 4). The 3-dimensional boolean array *arcFlag* holds whether an edge between a source and a target node must be considered for a target region (line 5). Boundary nodes, i.e. nodes with a predecessor from a different region, are marked by the array *boundary* (line 6).

The method *setArcFlags()* (lines 8–63) does the precalculation. It begins with setting all nodes to non-boundary (lines 10–12). For all nodes n and m with an edge between them (lines 14–20), it unsets the arc flags for all regions (lines 22–24). If the nodes n and m are from different regions, m is marked as boundary node, because it has a predecessor from a different region (lines 26–27). Otherwise the edge connects nodes from the same region and its arc flag for that region is set (line 29).

After all boundary nodes are found, *setArcFlags()* performs a backward search to each boundary node v (lines 36–42). For all nodes n and m with an edge between them (lines 44–50), it checks if that edge was used in a shortest path to v and sets its arc flag for the region of v (lines 52–55). An edge from n to m was used in a shortest path to v if the difference of n's

and m's distance is equal to the weight of that edge.

Because the target node t must be a valid node, we can omit to test this in the loop condition of *run(s, t)* (line 70).

For the algorithm to respect the arc flags only a modification of *outerLoop(s, t)* (lines 76–92) is necessary. Besides the check in the inner loop for an edge between nodes u and i, it now also tests whether the arc flag of that edge is set for the target region (lines 85–87). Only when it is set, that edge is considered and *innerLoop(s, t, u, i)* is called.

In lines 96–99 the signatures of the methods for the bidirectional algorithm are given, which are only used for the precalculation. Their implementation is identical to that we described in section 6.1.1.

6.2.2 Specification

The specification for the arc-flag-optimized algorithm is almost the same as for the t-optimized algorithm. We only have to add another condition for edges. There must not only be an edge between two nodes x and y (*edge[x][y]*), but also the arc flag of that edge has to be set for the target region (*arcFlag[x][y][region[t]]*).

Since the target node has to be a valid node we can omit the check for it. Additionally, we want the precalculation to be done before the algorithm is started. Therefore we use a boolean ghost field *arcFlagsSet* that indicates whether *setArcFlags()* was already called. Ghost fields are fields that can only be used in the specification. We will detail this below when we specify the class invariants and the precalculation.

Listing 42: Modified specification for the method *run(s, t)*

```
1  /*@ public normal_behavior
2    @   requires arcFlagsSet;
3    @   ...
4    @   requires 0 <= t && t < nodeCount;
5    @   ...
6    @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
7    @        (!reachable[t] || distance[n] < distance[t]);
8    @            (\forall int m; 0 <= m && m < nodeCount && edge[n][m] &&
9    @            arcFlag[n][m][region[t]];
10   @                 reachable[m]));
11   @   ...
12   @   ensures (\forall int m; 0 <= m && m < nodeCount && reachable[m] && m != s;
13   @            (\exists int n; 0 <= n && n < nodeCount && reachable[n] &&
14   @            edge[n][m] && arcFlag[n][m][region[t]];
15   @                 distance[m] == distance[n] + weight[n][m]));
16   @   ensures (\forall int n; 0 <= n && n < nodeCount && reachable[n] &&
17   @        (!reachable[t] || distance[n] < distance[t]);
18   @            (\forall int m; 0 <= m && m < nodeCount && edge[n][m] &&
19   @            arcFlag[n][m][region[t]];
20   @                 distance[m] <= distance[n] + weight[n][m]));
21   @   ensures reachable[t] ==> (\forall int n; 0 <= n && n < nodeCount &&
22   @        reachable[n] && edge[n][t] && arcFlag[n][t][region[t]];
23   @            distance[t] <= distance[n] + weight[n][t]);
24   @   ...
25   @*/
26 void run(int s, int t) {
27     init(s, t);
28
29     while (existsMin() && !visited[t]) {
30         outerLoop(s, t);
31     }
32
33 }
```

The specifications of the four methods are modified in the same way, so we only present the modifications in the method contract of *run(s, t)*.

Listing 42 shows the modifications in the method contract of *run(s, t)*. It is required that the precalculation is done which is indicated by *arcFlagsSet* (line 2) and that the target node t is a valid node (line 4). Nodes m are reachable if they have a reachable predecessor n that has a smaller distance than t and the arc flag of the edge is set (lines 6–10).

The distances of nodes except the source node s are computed via a predecessor where the arc flag of the edge is also set (lines 12–15). For nodes there is no shorter path via a predecessor with a smaller distance than that of t and where the arc flag of the edge is set (lines 16–20). If t is reachable, there is no shorter path to it via a predecessor where the arc flag of the edge is set (lines 21–23).

The four methods of the bidirectional algorithm for the precalculation already have their specifications (see section 6.1.2).

The existing class invariants need only to be modified to use the new class name. For the new fields we have to specify some new invariants. There must be at least one region such that a node can be assigned to one. And for all nodes to assign a region to the array *region* must have a length equal to the number of nodes. The values of this array must be valid regions, i.e. they must lie between 0 and *regionCount*.

The 3-dimensional boolean array *arcFlag* must allow the first index to be a node, i.e. the length must equal *nodeCount*. The same holds for the second index. The third is used for regions, so the innermost arrays must have a length equal to *regionCount*. At last, the array *boundary* has a length of *nodeCount* to mark every node as boundary or inner.

As mentioned, for the specification of *run(s, t)* we needed a boolean value to indicate whether the precalculation has already been done, for which we used a ghost field. Ghost fields can be used as normal Java fields, but are invisible to Java and only accessible in the specificaton. They are declared in the specification with the keyword *ghost* and can be assigned a value with the keyword *set*.

To specify how the values of *arcFlag* are set we need to store the results of each backward search. We use two ghost fields, *reachables* and *distances*. For each boundary node v, *reachables[v][n]* stores the reachability of node n in the backward search to node v. *distances[v][n]* holds the corresponding distance for n. Accordingly, the lengths of the arrays and the inner arrays must be equal to *nodeCount*. We will specify what values those arrays must have, later.

Because we added some more arrays we have to address aliasing, again. For the arrays *reachable*, *visited*, *boundary*, *edge*, *arcFlag* and *reachables*, as well as for *distance*, *region*, *weight* and *distances*, we have to state that they all are different from each other.

Listing 43: Additional class invariants for the implementation with arc flags

```
1  //@ invariant regionCount >= 1;
2  int regionCount;
3
4  /*@ invariant region.length == nodeCount;
5    @ invariant (\forall int n; 0 <= n && n < nodeCount;
6    @      0 <= region[n] && region[n] < regionCount);
7    @*/
8  int[] region;
9
10 /*@ invariant arcFlag.length == nodeCount;
11   @ invariant (\forall int n; 0 <= n && n < nodeCount;
12   @      arcFlag[n].length == nodeCount && (\forall int m; 0 <= m &&
13   @      m < nodeCount; arcFlag[n][m].length == regionCount));
14   @*/
15 boolean[][][][] arcFlag;
16
17 //@ invariant boundary.length == nodeCount;
18 boolean[] boundary;
19
20 //@ ghost boolean arcFlagsSet;
21
22 /*@ invariant reachables.length == nodeCount;
23   @ invariant (\forall int n; 0 <= n && n < nodeCount;
24   @      reachables[n].length == nodeCount);
25   @*/
26 //@ ghost boolean[][][] reachables;
27
28 /*@ invariant distances.length == nodeCount;
29   @ invariant (\forall int n; 0 <= n && n < nodeCount;
30   @      distances[n].length == nodeCount);
31   @*/
32 //@ ghost int[][][] distances;
33
34 /*@ invariant reachable != visited && reachable != boundary &&
35   @      visited != boundary;
36   @ invariant reachables != edge;
37   @ invariant distance != region;
38   @ invariant distances != weight;
39   @ invariant (\forall int n; 0 <= n && n < nodeCount; reachable != edge[n] &&
40   @      visited != edge[n] && boundary != edge[n] &&
41   @      reachable != reachables[n] && visited != reachables[n] &&
42   @      boundary != reachables[n] && (\forall int m; 0 <= m && m < nodeCount;
43   @           reachables[n] != edge[m] && n != m ==>
44   @                (reachables[n] != reachables[m] && edge[n] != edge[m])));
45   @ invariant (\forall int n; 0 <= n && n < nodeCount; arcFlag[n] != edge &&
46   @      arcFlag[n] != reachables && (\forall int m; 0 <= m && m < nodeCount;
47   @           arcFlag[n][m] != reachable && arcFlag[n][m] != visited &&
48   @           arcFlag[n][m] != boundary && (\forall int o; 0 <= o && o < nodeCount;
49   @                arcFlag[n][m] != edge[o] && arcFlag[n][m] != reachables[o])));
50   @ invariant (\forall int n; 0 <= n && n < nodeCount; (\forall int m; 0 <= m &&
51   @      m < nodeCount; (\forall int o; 0 <= o && o < nodeCount; (\forall int p;
52   @      0 <= p && p < nodeCount; (n != o || m != p) ==>
53   @           (arcFlag[n][m] != arcFlag[o][p])))));
54   @ invariant (\forall int n; 0 <= n && n < nodeCount; distance != weight[n] &&
55   @      distance != distances[n] && region != distances[n] && (\forall int m;
56   @      0 <= m && m < nodeCount; distances[n] != weight[m]));
57   @ invariant (\forall DijkstraAbstractArc d; \created(d) && this != d;
58   @      reachable != d.reachable && reachable != d.visited &&
59   @      reachable != d.boundary && (\forall int n; 0 <= n && n < d.nodeCount;
60   @           reachable != d.edge[n]));
61   @ invariant (\forall DijkstraAbstractArc d; \created(d) && this != d;
62   @      visited != d.visited && visited != d.reachable &&
63   @      visited != d.boundary && (\forall int n; 0 <= n && n < d.nodeCount;
64   @           visited != d.edge[n]));
65   @ invariant (\forall DijkstraAbstractArc d; \created(d) && this != d;
66   @      boundary != d.boundary && boundary != d.reachable &&
67   @      boundary != d.visited && (\forall int n; 0 <= n && n < d.nodeCount;
68   @           boundary != d.edge[n]));
69   @ invariant (\forall DijkstraAbstractArc d; \created(d) && this != d;
70   @      distance != d.distance && distance != d.region && (\forall int n;
71   @      0 <= n && n < d.nodeCount; distance != d.weight[n]));
72   @*/
```

The additional invariants for the fields in the arc-flag-optimized implementation are shown in listing 43. *regionCount* must be at least 1 (line 1). The length of *region* equals *nodeCount* and all values must lie between 0 and *regionCount* (lines 4–6). The array *arcFlag* has a length of *nodeCount*, the inner arrays, too. Only the innermost arrays have a length of *regionCount*

(lines 10–13). *boundary* has a length of *nodeCount* (line 17).

The ghost field *arcFlagsSet* is declared in line 20, but for it no invariants are necessary. The ghost fields *rechables* and *distances* are declared in lines 26 and 32. They and their inner arrays have lengths of *nodeCount* (lines 22–24, 28–30).

In lines 34–71 the invariants concerning the inequality of the arrays are given.

Until now we used the values of *arcFlag* without restricting them in any way. For the algorithm to correctly compute the shortest paths, the arc flags cannot be set arbitrarily. Setting an edge's arc flag is always allowed, because this gives the algorithm only the possibility of using the edge for a shortest path. However, not setting an edge's arc flag is only allowed under some conditions. Since an unset arc flag forbids the algorithm to use that edge, if the edge is part of a shortest path, the algorithm could not find this shortest path, anymore.

Thus, for each two nodes n and m with an edge e between them and each region r holds that if the arc flag *arcFlag[n][m][r]* is not set, there is no boundary node v from region r such that e is used in a shortest path to v. This is an invariant holding after the precalculation has been done, so it only has to hold, if *arcFlagsSet* is *true*.

That an edge from n to m is used in a shortest path to v can be deduced from the distances and the reachability of a backward search to v. Those are stored in *distances[v]* and *reachables[v]*. That edge is used if the difference of n's and m's distance is equal to the edge weight *weight[n][m]*.

In *reachables[v]* and *distances[v]* the shortest paths of the backward search to every boundary node v are stored. To specify this, we use the postconditions of *runPre(s, t, forward)* with v as source node, no target node and *forward* set to *false*. So for all boundary nodes v, v is reachable and all successors of reachable nodes are reachable, too. Also the distance of v is 0, the distances of all other nodes are computed via a reachable predecessor and there are no shorter paths. Again, this invariant has to hold, if *arcFlagsSet* is *true*, i.e. the precalculation has been done.

Now we need to specify which nodes are boundary nodes. *boundary[v]* is set only if there exists a predecessor u with u and v belonging to different regions. This invariant also only has to hold if the precalculation has been done.

In listing 44 the invariants concerning the values of *arcFlag*, *reachables*, *distances* and *boundary* are shown. If *arcFlagsSet* is set, an arc flag of an edge for a region may only be unset, if there exists no boundary node from that region such that the edge is used in a shortest path to that node (lines 1–7).

Lines 9–24 specify that if *arcFlagsSet* is *true*, the values of *reachables[v]* and *distances[v]* hold the correct values of a backward search to node v. Adapted postconditions of *run(s, t, forward)* from the bidirectional imple-

Listing 44: Invariants for the values of *arcFlag*, *reachables*, *distances* and *boundary*

```
1  /*@ invariant arcFlagsSet ==> (\forall int n; 0 <= n && n < nodeCount;
2    @      (\forall int m; 0 <= m && m < nodeCount; edge[n][m] ==>
3    @          (\forall int r; 0 <= r && r < regionCount; !arcFlag[n][m][r] ==>
4    @              !(\exists int v; 0 <= v && v < nodeCount && region[v] == r &&
5    @              boundary[v] && reachables[v][n] && reachables[v][m];
6    @                  distances[v][n] - distances[v][m] == weight[n][m]))));
7    @*/
8
9  /*@ invariant arcFlagsSet ==>
10   @      (\forall int v; 0 <= v && v < nodeCount && boundary[v];
11   @          reachables[v][v] &&
12   @          (\forall int n; 0 <= n && n < nodeCount && reachables[v][n];
13   @              (\forall int m; 0 <= m && m < nodeCount && edge[m][n];
14   @                  reachables[v][m])) &&
15   @          distances[v][v] == 0 &&
16   @          (\forall int m; 0 <= m && m < nodeCount &&
17   @              reachables[v][m] && m != v;
18   @                  (\exists int n; 0 <= n && n < nodeCount &&
19   @                  reachables[v][n] && edge[m][n];
20   @                      distances[v][m] == distances[v][n] + weight[m][n])) &&
21   @          (\forall int n; 0 <= n && n < nodeCount && reachables[v][n];
22   @              (\forall int m; 0 <= m && m < nodeCount && edge[m][n];
23   @                  distances[v][m] <= distances[v][n] + weight[m][n])));
24   @*/
25
26 /*@ invariant arcFlagsSet ==>
27   @      (\forall int v; 0 <= v && v < nodeCount; boundary[v] <==>
28   @          (\exists int u; 0 <= u && u < nodeCount && edge[u][v];
29   @              region[u] != region[v]));
30   @*/
```

mentation are used for this.

If *arcFlagsSet* is set, a node v is marked as boundary node only if it has a predecessor u from a different region (lines 26–30).

At last, we are left with the specification for the method *setArcFlags()*, which does the precalculation. The only postcondition is that it must set *arcFlagsSet* to *true*, so that those last three invariants can be used. More postconditions are not needed.

In order to put these invariants in place, *setArcFlags()* sets the values of the (ghost) arrays *reachables* and *distances* accordingly. It also sets *arcFlagsSet*.

Listing 45: Specification of the method *setArcFlags()*

```
1  /*@ public normal_behavior
2    @   ensures arcFlagsSet;
3    @   modifies arcFlagsSet, reachables[*][*], distances[*][*], boundary[*],
4    @       arcFlag[*][*][*], target, visitedNodes, visited[*], reachable[*],
5    @       distance[*];
6    @*/
7  void setArcFlags() {
8      ...
9      for (int v = 0; v < nodeCount; v++) {
10         ...
11         runPre(v, -1, false);
12
13         for (int n = 0; n < nodeCount; n++) {
14             //@ set reachables[v][n] = reachable[n];
15             //@ set distances[v][n] = distance[n];
16             ...
17         }
18
19     }
20
21     //@ set arcFlagsSet = true;
22 }
```

Listing 45 presents the specification of *setArcFlags()*. Line 2 states that *arcFlagsSet* is set to *true*, which is indeed done in line 21. According to the modifies clause of lines 3–5 the method modifies the values of *arcFlagsSet*, *target*, *visitedNodes*, and the values of the arrays *reachables*, *distances*, *boundary*, *arcFlag*, *visited*, *reachable* and *distance*; either by itself or through a call to *runPre(s, t, forward)*, which performs the backward search.

For each boundary node v and each node n, *setArcFlags()* stores the reachability and the distance of n from the backward search to v in *reachables[v][n]* and *distance[v][n]* (lines 14–15) to put in place the three invariants.

6.2.3 Verification

When we try to verify the methods, we find that the invariants that the method *setArcFlags()* puts in place are of such complexity that a verification is not possible for KeY due to its performance issues (see section 7.2). Only the correctness of the method *init(s, t)* can be shown.

However, those invariants are not needed to show the proof obligations *EnsuresPost* and *RespectsModifies* of the algorithm's methods (*init(s, t)*, *run(s, t)*, *outerLoop(s, t)* and *innerLoop(s, t, u, i)*). For *PreserveInvs* we would have to show that the invariants are preserved. Because the methods do not modify any locations referenced by those invariants, it is clear that they would preserve them.

Only for the verification of *setArcFlags()* those invariants are needed, since they implicitly represent the postconditions of that method. But, as *setArcFlags()* itself is very complex – it contains several nested loops and calls to backward searches – it would be questionable (because of KeY's performance issues) whether the verification could be done, even if there were no problems with the complexity of the invariants.

So we remove the invariants and verify only the four methods of the arc-flag-optimized algorithm. Since the methods for the backward search were already verified in section 6.1.3, we don't need to verify them here again.

Without the invariants, we can verify *init(s, t)* and *innerLoop(s, t, u, i)* the same way as before. For *run(s, t)* and *outerLoop(s, t)* we have to limit KeY further in the application of rules. After the auto mode is started with the settings *Logical splitting*: *Off* and *Quantifier treatment*: *None*, and then with *Quantifier treatment*: *No splits*, we keep the latter settings and apply rules for splitting the proof by hand. This keeps the proofs from bloating up too much due to unnecessary splits KeY would make automatically.

7 Results

We will sum up the verification of the different implementations and give some proof statistics in section 7.1. Then, in section 7.2, we will discuss the

issues we experienced with KeY during the verification. Finally, we will give some feedback for algorithm engineers in section 7.3.

7.1 Verification Summary

In this work we verified five different implementations of Dijkstra's Algorithm. Four of them were abstract implementations based on one another. We began with a standard implementation (section 3) and later included the t-optimization (section 4). Then we modified that version to also allow backward searches (section 6.1). In the end we presented an arc-flag-optimized implementation (section 6.2) which is based upon the t-optimized implementation and uses the bidirectional implementation in its precalculation. The fifth implementation is a concrete implementation using optimized data structures (sections 2 and 5). It corresponds to the t-optimized abstract implementation.

For the verification of each abstract implementation, except the first one, we used the specification of a previous implementation and modified it to incorporate the changes. Although the modifications to the specifications were only small, there were some bigger blow-ups in the proofs. Table 1 shows proof sizes and required user interactions for the verification of the four abstract implementations.

Table 1: Proof statistics for the abstract implementations

	standard	t-optimized	bidirectional	arc flags	
init	1640	1650	1647	3241	nodes
	11	11	11	11	branches
	0	0	0	0	interactions
run	20974	53390	85235	51813	nodes
	86	230	368	55	branches
	11	54	94	47	interactions.
outerLoop	45064	48846	85745	73039	nodes
	195	217	382	93	branches
	8	12	19	78	interactions
innerLoop	6691	6852	12747	16455	nodes
	35	37	70	43	branches
	3	3	6	3	interactions

While for most methods of the t-optimized implementation the proofs stayed almost the same in comparison to the standard implementation, there was a definite blow-up for the method *run*, in proof size as well as in required user interactions. We modified the loop invariant and the postconditions of *run* in different ways. This made it more difficult to conclude the postconditions using the loop invariant, which is the reason for the blow-up. The pre- and postconditions of the method *outerLoop* were modified in the same way, so that there was no blow-up.

The biggest blow-ups occurred in the verification of the arc-flag-optimized implementation. Due to the additional arc flag condition the proof gets split much more and thus proof size and required user interactions increase. For the methods *run* and *outerLoop* this is not apparent from the proof statistics, because we used more-restrictive auto mode settings in the proofs to reduce proof size. With the same settings we used in the other implementations, however, the proofs would be much larger.

Because of the increased proof sizes and required user interactions, the verification of an optimized implementation without a previous, simpler implementation to learn from would be much harder. This was the reason why we had to first implement and verify the abstract implementation before verifying the concrete implementation. By doing this we mimicked the development cycle of the algorithm. Thus, including verification in the development process of an algorithm makes the verification of an resulting, optimized implementation less difficult.

The proof statistics for the verification of the concrete implementation, which are shown in comparison to the t-optimized abstract implementation in table 2, emphasize this further. The proofs for the two methods we could verify are much larger than in the abstract implementation, especially for the method *innerLoop*. Without the previous verification of the abstract implementation we wouldn't have been able to find the right specification and complete the proof for the method *innerLoop*.

Table 2: Proof statistics for the t-optimized implementations

	abstract	concrete	
init	1650	9528	nodes
	11	86	branches
	0	4	interactions
run	53390	–	nodes
	230	–	branches
	54	–	interactions.
outerLoop	48846	–	nodes
	217	–	branches
	12	–	interactions
innerLoop	6852	166879	nodes
	37	1783	branches
	3	162	interactions

Even with the previous verification we were not able to complete the proofs for the methods *run* and *outerLoop*. In each implementation these methods have had considerably larger proofs than the method *innerLoop*. Because the proof for *innerLoop* in the concrete implementation almost brought KeY to its limit, we expect the methods *run* and *outerLoop* to not be provable by KeY due to its performance issues which we will detail in section 7.2.

7.2 Issues with KeY

We have seen that some proof obligations couldn't be proven with KeY. Mostly, this can be tracked back to performance issues KeY has with more complex proof obligations. It results in rule applications getting slower and slower, until eventually KeY crashes with an *OutOfMemoryError*. Depending on the available memory KeY (or rather the Java Virtual Machine) has at its disposal this happens sooner or later. In the standard installation KeY is started with a Java maximum heap of 1024 MB; we modified this to use a maximum of 2048 MB of memory. In tests we found that KeY would crash after about 150000 rule applications with the default setting. Using the increased heap size we reached 230000 rule applications, before KeY crashed. However, depending on the complexity of the proof obligation these numbers change extensively.

KeY is designed to be a user-friendly interactive theorem prover. The user shall be able to inspect every node of the proof tree and to apply rules to open goals by hand. To have all proof nodes available can be helpful in several ways. In the case that a user interaction is required for an open goal, the user can take a look at the path to this goal to find out what formula has to be proven and what steps have already been done for it. From this the user can get a hint which rule he has to apply or which instantiation to do. Also when having a completed proof, the user can check why the proof was closed. So besides the help for the user to get a proof closed, it provides some additional assurance that the proof can be trusted.

However, that rather high memory consumption follows from this design, because KeY needs to store and display the information of all intermediate proof steps. Based on 230000 rule applications with a heap size of 2048 MB until KeY crashes, this leaves only under 10 KB of memory for each node. With a complex proof obligation, like one for Dijkstra's Algorithm with Arc Flags, which contains many class invariants, such a size can be reached.

Before KeY crashes, its performance decreases rapidly the larger the proof becomes, resulting in less rule applications in a given time. In one case KeY needed over 10 hrs for about 20000 rule applications, which it otherwise does in only some minutes. KeY needs some free memory to create temporary (Java) objects in order to select a rule to apply. The larger the proof becomes, the less free memory is available, and the garbage collector needs more time to free it. Therefore, KeY needs more time, too.

Using the standard settings for the proof search strategy can result in very large proof trees that impact KeY's performance. One option responsible for this behavior is *Quantifier treatment*, which is set to *No splits with Progs* by default. When it is set to *No splits*, as we have done it for all of the verification tasks, KeY is restricted in the automated application of rules. While this setting sees to it that the size of the proof tree is not getting too big too quickly, of course, some proof obligations cannot be proven automatically,

anymore.

As an example we can take the abstract implementation of the standard algorithm and take a look at its method *innerLoop(s, u, i)* (see sections 3 and 7.1). In the verification with the settings we used we had to do three instantiations by hand, and the proof was closed with 6691 nodes and 35 branches. For this method we can set *Quantifier treatment* to the default value, *No splits with Progs*, and get a successful verification without interaction. However, it now takes 66010 nodes and 268 branches for the proof, which is ten times mor than in the old proof. Thus, using this setting for other proofs is out of the question, although that means we lose automation and require more user interaction.

The required user interaction is acceptable, as long as its amount stays within certain limits. For the method *run(s)* in the abstract implementation of the standard algorithm we had to do instantiations in six open goals. By adding the t-optimization this number was increased to 28. Most of the additional 22 goals resulted from KeY doing unnecessary case distinctions, and thus needed the same instantiations. In first verification attempts with the arc-flag-optimized implementation KeY would do even more case distinctions and leave us with 92 open goals. Again, most of them needed the same instantiation, but due to the case distinctions they had to be done now several times.

Besides the performance issues, there are some features KeY misses. For the termination of the loop in the method *run(s)* (in the abstract implementation of the standard algorithm, see section 3.2.1) we used the variant $nodeCount - visitedNodes$. One proof obligation for this variant is to show that if the variant is 0 the loop condition will be *false*. *visitedNodes* holds the number of nodes which are set visited, so the variant counts the number of unvisited nodes. The loop condition checks for the existence of an unvisited node and thus, if the variant is 0, there are no unvisited nodes and the loop condition will be *false*, as it is required. To verify this in KeY we need to show that *visitedNodes* holds indeed the number of visited nodes and that there can be no unvisited node, if it equals *nodeCount*. A feature that can count visited nodes is implemented, but is not powerful enough for our purpose.

Another feature KeY does not support are JML model fields, which we introduced in section 5.2. Especially when the verification process is integrated into the development process of an algorithm, model fields would be of use. While the implementation changes when the code is optimized, the specification of the interface could stay the same, because model fields allow to specify the interface without referencing the implementation. Without model fields the specification needs to be changed as well to reflect the new implementation. The specification with model fields is typically on a higher level of abstraction as an optimized implementation, because in general it uses only basic data structures and no optimizations. So the semantics of the

specification is clearer than if it was on the same level as the implementation.

A minor issue regarding the usability, but which has no impact on the power for proving, concerns the user interface which displays a node's sequent. In this sequent view the formulae of the antecedent and the succedent are not ordered. So when an interaction is required the user has to search for a formula where he can apply a rule. In bigger sequents, as we came across in our verification tasks, the formulae span several pages of a screen and a small formula can be overlooked easily while scrolling. An ordering which orders the formulae according to their structure would be helpful.

7.3 Feedback for Algorithm Engineers

To provide feedback for the algorithm engineers we need to analyze the specification and verification. We can divide this into two parts, the requirements and the conclusions. While the former only consists of the class invariants and the preconditions of the algorithm's starting method, the latter covers all logical consequences, including the postconditions of the starting method, loop invariants and the specification of the other methods. We do not include the preconditions (the requirements) of those other methods in the requirements part, because they are either logical consequences of or also requirements for the calling method.

In the requirement part we have to check if all of the requirements are needed for the verification. In our case there are only few requirements which are interesting to us, because the rest is only dealing with the lengths and the pairwise inequality of the used arrays. One invariant states that all edge weights have to be non-negative. This, as already explained in section 1.3, is necessary. Also, the precondition of the method *run*, stating that the source node s must be a valid index, is needed in each implementation. In the t-optimized implementation (see section 4) there is another parameter t denoting the target node. In contrast to s, no array is accessed at index t and thus, t needs not to be a valid index and so we did not include it as precondition.

Looking at the class invariants of the graph in the concrete implementation (see section 5.2.1) there seems to be missing an invariant concerning the relationship of outgoing and incoming edges; if there is an outgoing edge, then an incoming edge must exist for the target node, as well, and vice versa. However, we do not need this, because we only use outgoing edges in the algorithm. In fact, we can modify the data structure to only store outgoing edges, which would save time and memory.

For the second part of the analysis we take a look at the conclusions of the standard abstract implementation (see section 3.2). In the postconditions of the method *run(s)* we find the expression *... && edge[n][m]* in the formulae for the reachability and the distances of nodes. We can add here some other condition, indicating whether we want the algorithm to consider the edge

between nodes n and m. Most generally, we can use some oracle function as this condition. In the arc-flag-optimized implementation (see section 6.2) we used such a condition, where the precalculated arc flags indicated whether an edge is to be considered.

Another option to modify the implementation gives us the loop invariant in the method *run(s)* stating that visited nodes have not greater distances than unvisited ones. Together with the postcondition of the method *innerLoop(s, u, i)* that distances of visited nodes will not get modified, we can change the implementation to not check all successors i of node u, but only unvisited ones. Also, when we are interested in shortest paths to only some nodes, we can stop the algorithm once every such node is visited, because their distances will get any smaller. A simplified version, with only one target node we are interested in, is the t-optimization of section 4.

We already used most of this feedback in the implementations. This is due to Dijkstra's Algorithm being such a popular and old algorithm. It has been and is still being studied and optimized, so providing new feedback for this algorithm is harder than for a new one. But we have seen that the feedback we got from the analysis could be used.

8 Conclusion

We presented a concrete implementation of Dijkstra's Shortest Path Algorithm and its verification process with KeY, which included the prior verification of a more abstract implementation. We also verified two more variations of the algorithm based on the abstract implementation.

We have seen that, though there are some features KeY misses, it can be used for algorithm verification. However, the biggest issue is its performance, because of which we could not complete the verification of the concrete implementation.

The performance issues of KeY are due to the complexity of the proof obligations for the used algorithm. In the abstract implementation there are 46 lines of code opposed to 183 lines of specification. The optimized data structure of the concrete implementation increases the complexity even more.

Since we had to verify a more abstract implementation first, we mimicked the development process of the algorithm. Thus, it makes sense to integrate the verification process into the development process from the start, when an optimized implementation shall be verified.

In the analysis of the specification and verification we found some points providing feedback for algorithm engineers, which supports the integration of formal verification into the development process further. Those points were already implemented in subsequent variations of the algorithm, but this is due to the algorithm being so well-known.

8.1 Future Work

Before starting to integrate verification into the development process of an algorithm, which is the goal in the end, we should do some more evaluation with other algorithms.

For one we need to see where the complexity of Dijkstra's Algorithm is settled in comparison to other algorithms. Also, with other algorithms there might be more or less issues with KeY.

Moreover, with an algorithm which is not so well-known we can evaluate whether the verification pays off by providing new feedback.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows : theory, algorithms, and applications*. Prentice Hall, Upper Saddle River, NJ [u.a.], 1993.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BLW08] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. Holboogie - an interactive prover for the boogie program-verifier. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2008.

[Del09] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LPC+03] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. Jml reference manual, 2003.

[Moy98] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. Updated by RFC 5709.

[Poh71] Ira Pohl. Bi-directional Search. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Sixth Annual Machine Intelligence Workshop*, volume 6, pages 124–140. Edinburgh University Press, 1971.