University of Konstanz
Department of Computer and Information Science

Bachelor Thesis for the degree

Bachelor of Science (B. Sc.) in Information Engineering

A workbench for the K* algorithm

by

Sebastian Haufe

(Matr.-Nr. 01/852827)

1st Referee: Prof. Dr. Stefan Leue

2nd Referee: Prof. Dr. Marcel Waldvogel

Thursday 12$^{\text{th}}$ January, 2017

Abstract

Finding shortest paths is an often discussed task in the domain of graphs. Many search algorithms for shortest paths have taken on this challenge. In recent decades, several directed search algorithms for finding shortest paths have been invented. One of these algorithms is the K* algorithm which has been implemented by Husain Aljazzar.
The K* algorithm is part of the model checker tool DiPro which has been created by Husain Aljazzar, Florian Leitner-Fischer and Dimitar Simeonov. Hence, the K* algorithm only handles a few types of graphs which are convenient for this tool. One of the advantages is that the K* algorithm focuses on the performance to handle big graphs.
In this thesis the isolation of the K* algorithm is explained. The purpose of this kind of realisation is to get the K* algorithm as a stand-alone version usable for arbitrary graphs.

Contents

List of Figures

1. Introduction

Finding solution paths in a given graph is a purpose of search algorithms. Further, search algorithms for shortest paths involve finding the optimal solution path, between a source vertex and a target vertex, with minimal cost. The most meaningful search algorithm for shortest paths is probably the Dijkstra algorithm [10].
In more detail, the approach of k-shortest-paths problem algorithms will be considered in this thesis. The k-shortest-paths problem deals with finding the k best solution paths in an ascending order on a directed weighted graph.
Search algorithms are held in high regard in several application areas. They are used to find optimal solutions in applications which can be represented as graphs. In this thesis, an example for such an application will be the representation of road maps.
Such an algorithm is the K* algorithm by Husain Aljazzar [1, 2]. One of the most meaningful advantages of the K* algorithm is the fact that it operates on-the-fly, which means that only the part of the graph which is needed to find the k wanted solution paths is loaded into memory space. The whole graph is not to be loaded completely. This makes a contribution to the K* algorithm being a very efficient algorithm to deal with big graphs.
The K* algorithm is implemented as a part of the model checker tool DiPro [5, 9] which has been created by Husain Aljazzar, Florian Leitner-Fischer and Dimitar Simeonov. This fact makes the algorithm executable only for a few types of graphs associated to this tool.
In this thesis, the isolation of the K* algorithm as a stand-alone version will be described. This will guarantee a usage of the K* algorithm while the tool DiPro is not needed. The result will be a version of the K* algorithm which will be able to handle an arbitrarily given graph.
Subsequently, the given case studies will prove that this new version of the K* algorithm is the same version as Husain Aljazzar implemented. Further, it will be proven that it is not worse in terms of performance than the original implementation.

1.1. Contributions

The contributions listed below can be found in this thesis:

1. The implementation will be described to get an overview about the K* algorithm structure. Further a little process description can be found.

2. The isolated K* version with explanations of all its modifications listed separately.

3. New implemented interfaces for a general use of the K* algorithm. In addition to this feature, it is shown how the user can handle own heuristics and graphs with the K* algorithm.

4. The K* algorithm implementation is designed for a system-independent use. Hence it will be used by a command-line interface. To get it completely executable, a construction to compile the whole K* algorithm version is explained.

5. An evaluation by case studies shows that the efficiency did not decrease by isolating the algorithm, compared to Husain Aljazzar's version. This comparison will also be a topic of the evaluation. Further, there will be a comparison between this implementation of the K* algorithm and a K* implementation of another study.

The main tasks of this thesis were focused on the isolation of the K* algorithm, as a stand-alone version. For the abstraction an implementation of interfaces was necessary to connect to the K* algorithm.

1.2. Orientation

This thesis is structured as follows: The preliminaries can be found in section 2. They include a basic description of the K* algorithm and an explanation of graph types and heuristics. The main part of this thesis, which includes the isolation process and modification of the K* algorithm for a stand-alone version, will be explained in section 3. Further the interface implementation and a database connection can be found there. An evaluation of the K* algorithm by case studies can be found in section 4 and a conclusion will be given in section 5.

2. Preliminaries

In order to get a better understanding of this thesis a short description about the preliminaries will be given in this section.

2.1. Heuristics

Judea Pearl described a heuristic as a criteria which decides between several options to find the most promising way [8]. If there is the task to find a way between two different points A and B, and both points are not necessarily directly connected, it is obvious to aim for the shortest possible way. Hence, if more than one possible way between these two points exist the best way will be found by a comparison of all possible ways.
This kind of decision is the same approach as a heuristic does. In case of the K* algorithm the heuristic calculates the probability of every explored vertex. The calculated value gives the information how good the way from this current vertex to the target is.
How Judea Pearl described it in his paper [8], in case of the A* algorithm [11] the heuristic function consists of two parts. The first part is the value g, which is the cost of the way between the source vertex and the current vertex. The second part is the value h, which is the estimation of the needed cost from the current vertex to the target vertex. Both values yield the estimation function: $f = g + h$.
The heuristic evaluation function provides an essential part of the search result. The search can be guided by using this estimation to find the target faster.

2.2. The K* Algorithm

The K* algorithm is a search algorithm of the k-shortest-paths problem algorithms (ksp) [3]. The reason for developing the K* algorithm was that Husain Aljazzar focused on the problems conventional ksp algorithms have [1, 2]. With the K* algorithm, he implemented an algorithm which maintains performance by searches of large graphs. The K* algorithm offers two advantages compared to the other algorithms: First, K* is an algorithm operating on-the-fly. Only the vertices which need to be explored by search are stored in memory. Second, the K* algorithm uses a heuristic evaluation function for a target-guided search.

The K* algorithm depends on two search algorithms: The A* algorithm [11], which explores the graph to find the shortest path and Dijkstra algorithm [10], which searches the k solution paths on a created path graph structure, like Eppstein described in his paper [3].
The path graph structure of the K* algorithm combines two binary min heaps for every explored vertex. One for all incoming edges and another heap as tree heap. Further information about the structures can be found in the work of Husain Aljazzar [1, 2].
Basically, the K* algorithm is able to handle every type of graph. It operates on the identifier of each node that guarantees an independent use across different graph type. More information and a detailed description about the K* algorithm can be found in the work of Husain Aljazzar [1, 2].

2.3. Types of Graphs

A graph is represented by a number of vertices and edges. Focussing, that the K* algorithm operates on directed weighted graphs, edges are marked with a numerical number. Every graph, depending on its kind, has different properties on a vertex. So, the algorithm is constructed to operate only with the given ID of a vertex. In that way it does not matter with which properties a node comes with.
Hence, for all different types of graphs the K* algorithm is able to handle them.

3. Isolation of K* Algorithm

The idea to get the K* algorithm isolated was to get the algorithm itself as an independent executable component. If the algorithm has no heuristic evaluation function as an input it should still be executable. In this case the K* algorithm executes like Dijkstra's algorithm search [1, 10]. Additionally the algorithm should not crash if there is no given graph.
Hence, the K* algorithm is boxed into a shell of interfaces for the heuristic function and the graph files. A roughly structured overview of this implementation is depicted in Figure 1.

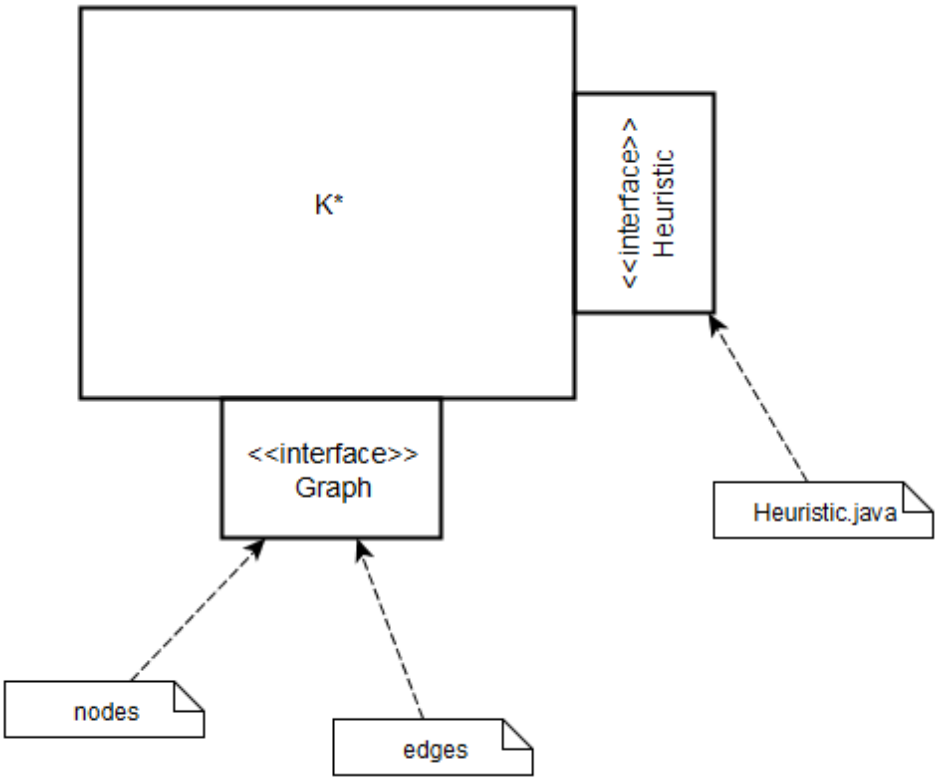

Figure 1: Rough structure of the K* algorithm.

Figure 1 shows that the K* algorithm is capsuled as an independent component. This guarantees that the algorithm is in a stable form as itself. For an execution of the K* algorithm it communicates via interfaces. Like it is shown in Figure 1, the K* algorithm needs a graph and a heuristic evaluation function for operation.
The K* algorithm handles the graph by two separate given attributes: the vertices and the edges. How the vertices and edges are given is irrelevant.

The only requirement for the allocation is as follows:

- As mentioned in the preliminaries the K* algorithm operates on the given identifier of the vertices. Hence, it is the only requirement for them. All additional properties of the vertices are only information for e.g. solution output.

- In case of directed weighted graphs, for the edges it is necessary to know the source vertex, the target vertex and the weight of the edge.

Theses properties are given by the user.

To get the description mentioned before as a result, several steps were required:

1. The tool DiPro [5, 9] has been looked through to get only these class files which are necessary for the K* algorithm. This guarantees a minimum of needed class files.

2. For the implementation of the heuristic and graph interface the initialization process of the K* algorithm had to be adapted. Further, the class files of DiPro only handle a few types of graph. Hence, these files had to be modified for a general usage.

3. Several existing class files of the K* algorithm itself had to be changed to handle the given input parameters by the interfaces.

4. To make the K* algorithm usable on several operating systems, this version has been designed for a execution on the command-line interface.

5. The complete solution output has been changed.

For the isolation process and modification of the files which are described in this thesis, the implementation is supported by [4, 13]. In case of the isolation and with the aim to get the same performance to the K* algorithm as Husain Aljazzar [1], the Java source files are extracted out of DiPro [5, 9]. These files are published under the public licences of GNU [15].

3.1. Isolated K* algorithm

The model checker tool DiPro [5, 9] was implemented for evaluating algorithmic approaches. Therefore, it collects a selection of a few search algorithms. One of these algorithms is the K* algorithm [1, 2]. The new version of the K* algorithm consists only of a few classes remaining from DiPro. In this section this implemented structure of the K* algorithm will be described in more detail. Because the whole K* algorithm is a

big construction, the class structure representation would be a cluttered overview. Hence, the description of it will be fragmented into packages and classes.

The implemented class structure of DiPro [5, 9] is maintained in the new K* algorithm. Subsequently, the existing packages of the K* algorithm will be explained. In order to avoid clutter, only the packages itself and the internal inheritances of the classes in the packages are represented.

It consists of the following packages:

alg: This package includes the algorithm structure of the K* algorithm, which will be explained in section 3.2. Additionally it contains the structures needed for the algorithms, like the path graph structure and the heaps.

context: By default this package is empty and should be filled with the context files. It contains all files for the context interface, explained in section 3.5.

db: This package is a completely new package in this version of the K* algorithm. In order to handle the K* algorithm with a database connection, it contains a class for the use of a graph or a heuristic. The database connection will be explained in section 3.6.

graph: This package contains the classes which represent the graph for the algorithm. It contains the default classes for the vertices and edges.

h: The heuristic file should be inserted into this package. It contains only the heuristic class which represents the heuristic interface. It will be explained in section 3.4.

run: This package includes the complete initialization process for the K* algorithm.

util: In this package all needed utilities of the K* algorithm are implemented and thus stored in here. E.g. the evaluation function, the solution collector and the algorithm reporter.

In the course of the isolation of the K* algorithm [1, 2] several classes were modified and a few new classes were implemented. These classes will be described in section 3.3.
After the isolation of the K* algorithm out of the tool DiPro [5, 9], it needs to get executable. In order to handle this problem a compilation class has been created, which will be explained in section 3.7.

Figure 2 shows the complete K* algorithm class diagram. The black framed classes are unchanged class files of the K* algorithm [1, 2]. The red framed classes are changed class files of the K* algorithm and the blue framed classes are newly created class files for the isolated K* algorithm.

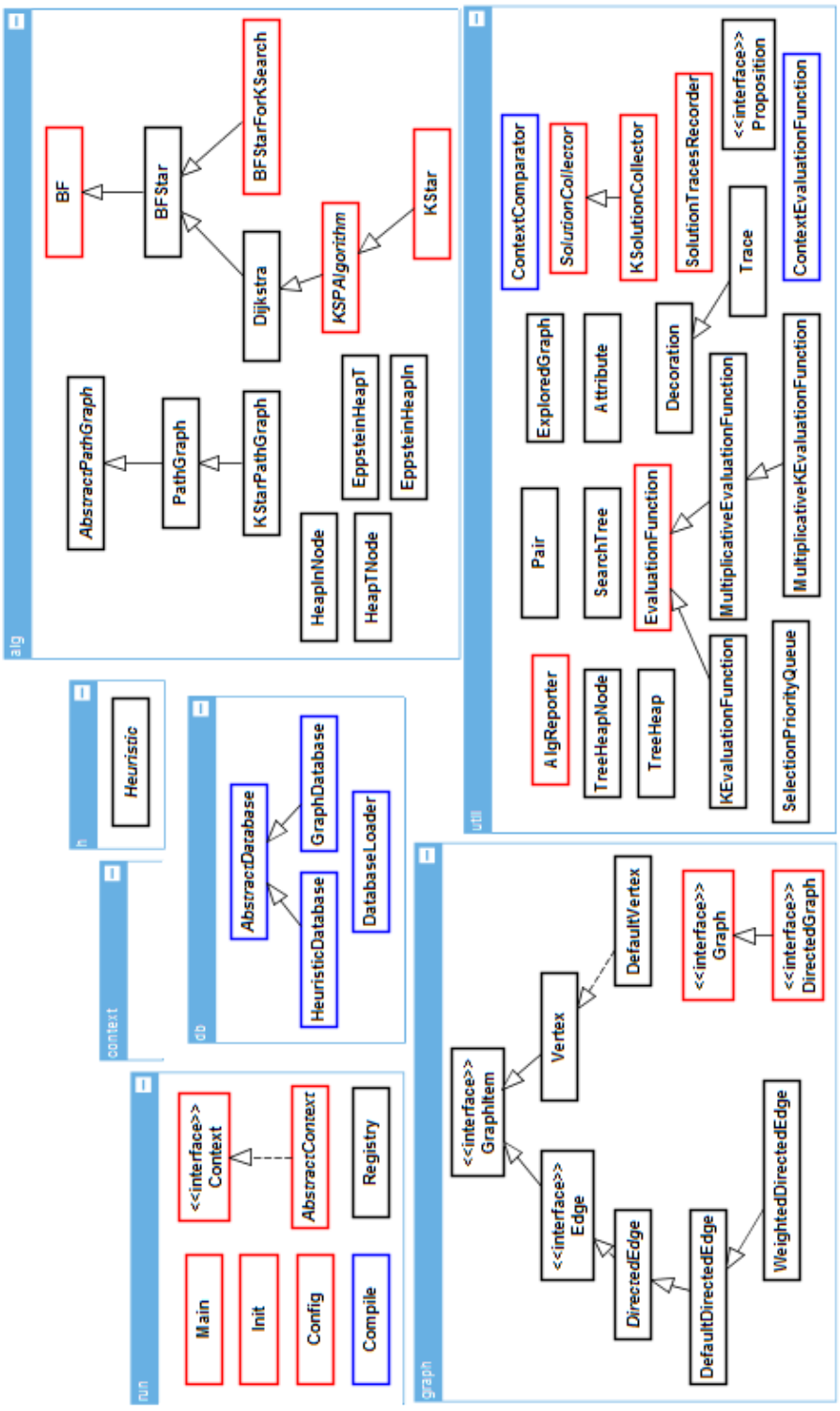

Figure 2: Packages of the isolated K* algorithm.

3.2. Initialization

The first important part of the whole K* algorithm is the initialization process. It handles the input parameters of the users' property file, offers a few default values for the properties at the beginning and contains the interfaces for loading of the users' heuristics and graphs. The sequence diagram in Figure 3 shows the initialization process for the K* algorithm. To keep the overview of the initialization process readable, this sequence diagram does not show the submethods of each instance.

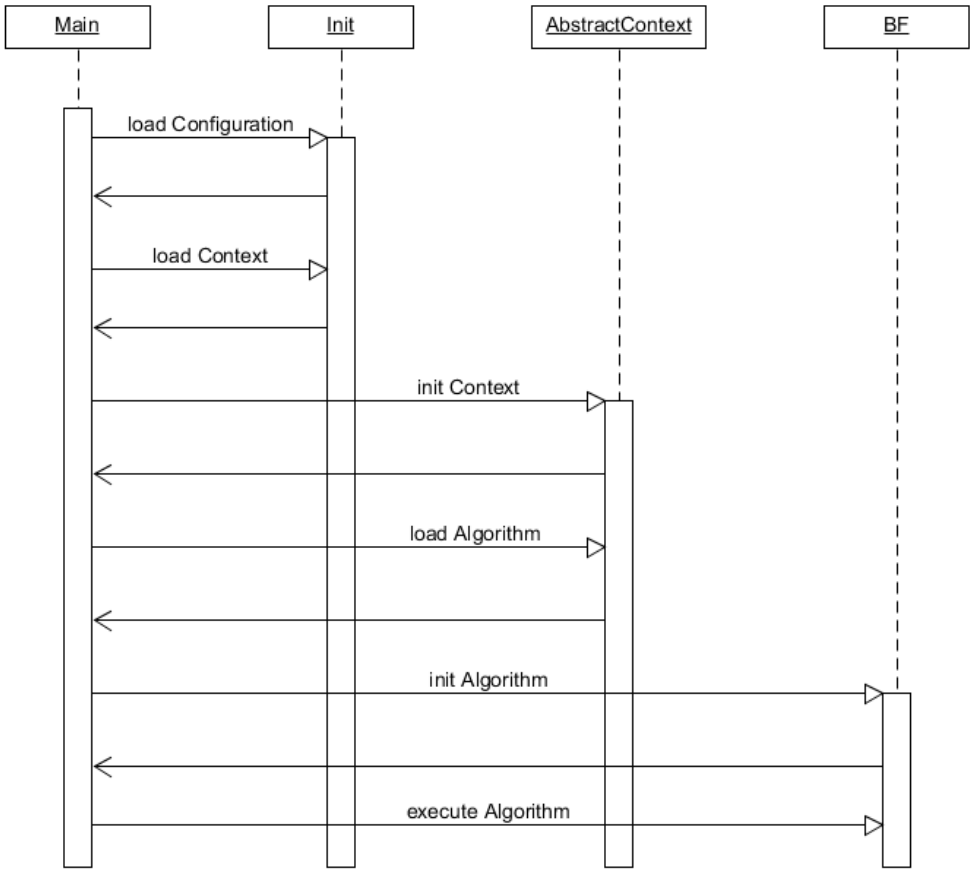

Figure 3: Initialization process of the K* algorithm.

The initialization process is basically maintained from DiPro [5, 9] with modifications for the new K* version, which will be described in the following. The initialization begins in the class `Main`, after the K* algorithm has been started via command-line interface. After setting the working

directory and initializing the registry, the first significant step is done by loading the configuration for the K* algorithm. Therefore, the default settings are loaded from the `Config` class. Then, the properties of the property file are parsed into the configuration. The next step is the first of two processes, where a user file is handled by loading the context file in the `Init` class. This is realized by reading the class name, which was parsed from the users' property file, and creating a new instance of this class. Next, the context file will be initialized. This is controlled by the context interface class `AbstractContext`. To initialize the context class the method *init()* of the `AbstractContext` class is called. This method calls the methods *readParameters()*, which parses the graph specific parameters and *loadModel()*, which loads the graph. Both methods have to be implemented by the user and will be explained in the section for the context interface, in section 3.5 of this thesis. After these steps the basic initialization is done and the algorithm can be loaded.
The K* algorithm will be initialized in the `AbstractContext` class by the method *loadAlgorithm()*. While this method loads several algorithms in DiPro [5, 9], it became unnecessary for the isolated K* algorithm. As mentioned in the description of the K* algorithm in the preliminaries, the K* algorithm is a combination of the A* algorithm [11] and Dijkstra algorithm [10]. Judea Pearl has described, that the A* algorithm is a modification of the BF* search algorithm, which in turn is a modification of the BF search [8]. The difference between the BF* and BF search algorithms is summarized by Husain Aljazzar in [1]. The BF* algorithm terminates if the target vertex should be expanded. That guarantees that the shortest path is found.
With this knowledge the structure of the K* algorithm has been implemented by Husain Aljazzar [1].

Figure 4 shows the inheritance structure of the implemented algorithms of the K* algorithm. A simplified version is as follows: The K* algorithm is a k-shortest-paths problem algorithm. Hence, the `KStar` class extends from `KSPAlgorithm`. In turn `KSPAlgorithm` extends from `Dijkstra` class, which is used for the path graph structure of the K* algorithm. As mentioned before, the A* algorithm [11] is a modified version of the BF search. Hence, the `BFStarForKSearch` class extends from `BFStar` which in turn extends from `BF` just as the `Dijkstra` class does. That is because the special case of no given heuristic will be equivalent to Dijkstra's algorithm [1, 10].

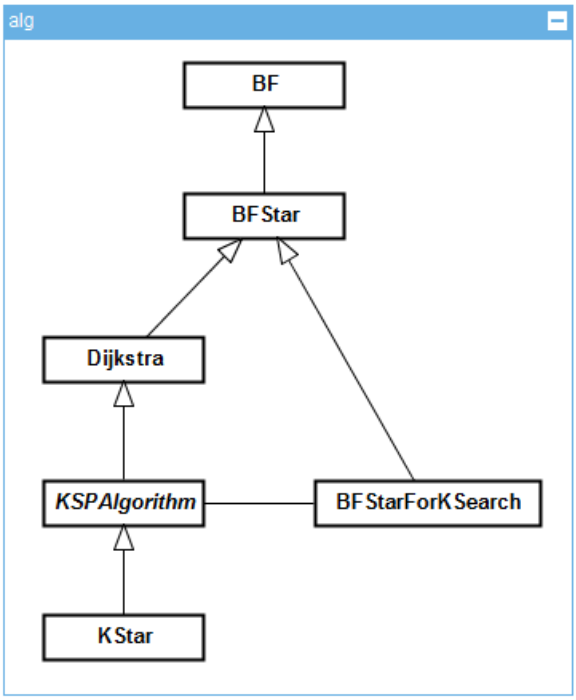

Figure 4: Inheritance of the K* algorithms.

By initializing the K* algorithm, two instances are created. An instance of the `KStar` class for Dijkstra search [10] on the path graph structure and an instance of the `BFStarForKSearch` class for the A* algorithm [11] to explore the graph.
In the last step, before the K* algorithm is running the heuristic function will be loaded from a given class file. This is the second process where a user file is loaded. Subsequently, the solution collector will be initialized and the graph properties will be set to the algorithm [1, 10].

3.3. Modification and Changes

As mentioned in the beginning of this thesis, a few modifications of the original K* algorithm class files had to be made, which are listed in the following sections.

3.3.1. Dynamic loading of Context and Heuristic

The most significant connecting component composes the interfaces between the K* algorithm [1, 2] and the class files for the heuristic and graph module. They form the prerequisite for the execution of arbitrary graphs by the K* algorithm. Hence, in this section it will be explained how the initialization process of the K* algorithm had to be changed, to integrate these interfaces to the complete algorithm structure.

The context is the first of the two Java classes which is loaded by the steps of the initialization, as previously explained in section 3.2. Before modifying the implementation, the context was originally loaded in the method *public Context loadContext(Config config)*. The explicit context class was selected by the given graph type parameter of the configuration object.
Now, this method is changed to load the context by the given file name of the property file. The context file name will be loaded from the parameter. A new instance will be created and returned as a type of the abstract superclass `AbstractContext`, which represents the context interface.
An illustration of the connection between the initialization and the context file is given in Figure 5.

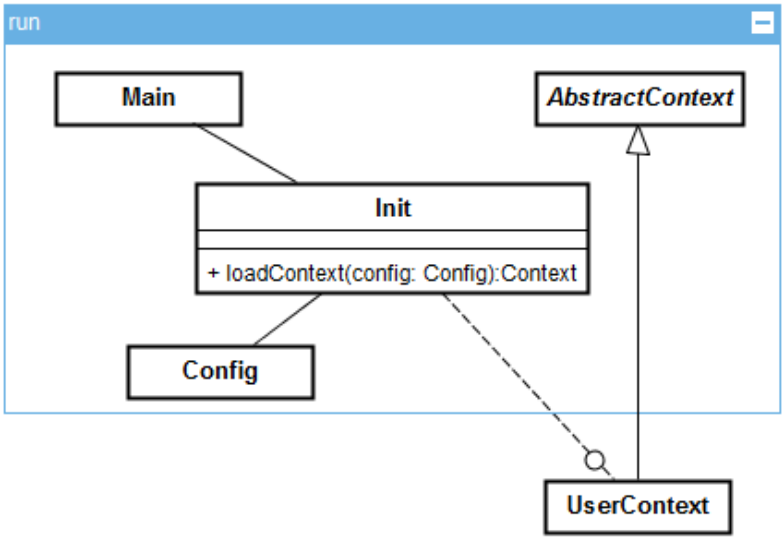

Figure 5: Class connection of the context.

The second loaded Java class is the heuristic by the initialization of the algorithm. The method *public void init()* of the class `BF` is loading the heuristic by calling the method *public Heuristic loadHeuristic(BF alg)*. The heuristic is loaded from a local given heuristic class file or from a given url of an existing database. The idea to load the local heuristic class by this way was implemented in DiPro [5, 9] and has been extended by the database selection. As the context file is loaded, the heuristic file will be loaded by a given file name from the property file. If no heuristic name or database url is given, the K* algorithm will execute without any heuristic. Figure 6 shows how the heuristic is called.

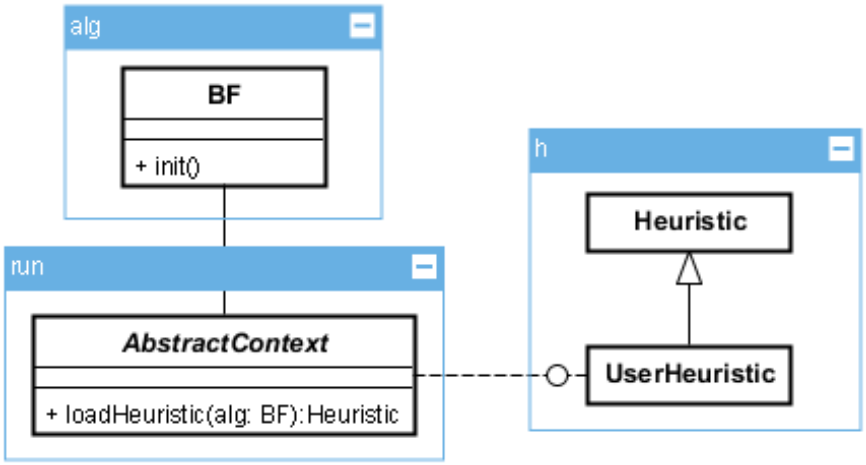

Figure 6: Class connection of the heuristic.

3.3.2. Properties of Execution

For execution of the K* algorithm several options exist to manipulate the search. This is useful to ensure the graph's compliance with the desired properties. The following properties have been implemented in DiPro [5, 9] and can be set by the user in a given property file:

greedy: This option brings the evaluation function to calculate the value f only by the heuristic function. The value g will be suppressed.

prune bound: All vertices, for which hold that their f value is higher than the prune bound will be pruned by the algorithm. In

that case the search will be optimized by saving memory space while pruning parts of the graph, which not promise optimal paths for solutions.

maxiter: The algorithm stops after the number of iterations is reached whether or not the wanted solution paths have been found.

maxtime: The algorithm stops after the limit of the runtime is reached whether or not the wanted solution paths have been found.

complete: If this option is set the whole graph will be explored.

These options are part of all options which can be set in the configuration of the K* algorithm in the initialization process. All options, including the new options, can be found in the appendix A.2 of this thesis and in the attached user manual.

3.3.3. Modification of the Context Class

The `AbstractContext` class forces the user to implement several methods for the context. Therefore, a few new classes are created and several abstract methods are newly declared.

New classes
The `ContextComparator` class is used to create a new instance of a comparator. Which type of result the comparator returns is implemented by the user in the context class. It is one of the methods the user has to implement and will be explained in section 3.5 for the context interface. The comparator is used for all comparisons in the K* algorithm. It is used in three ways. Firstly, it is used for the prune bound value as mentioned before. Secondly, it is used for the search queues and their f values to sort the selection in order of the best next vertex. Lastly, it is used for the delta values of the min heaps in the path graph structure. The delta value describes the detour by a sidetrack edge. A sidetrack edge is an edge which has been explored by the A* algorithm [11]. It is stored in the path graph structure and is not a part of the optimal solution path. More information about the detour and the sidetrack edge can be found in the paper of Husain Aljazzar [1, 2].
How the comparator will be implemented depends on the kind of the search.

A short example will illustrate the usage: For a distance graph the shortest way should be found. Therefore, the comparator supplies the lower value. Hence, a Double comparator can be selected. In case of the comparator needs to supply the higher value, the comparator can be negated. While the Comparator class in Java only returns lower, equal or higher as a result variable, a normal or negated comparison will make sense. For that, both versions have been implemented for the K* algorithm in DiPro [5, 9]. In addition to the new `ContextComparator` class, these comparators are implemented as subclasses in the `ContextComparator` and can be selected by the user in the context file. This will be explained in 3.5

The `ContextEvaluationFunction` class is used to create a new instance of the evaluation function f. There are two kinds of the evaluation function implemented in DiPro [5, 9]. One is an additive evaluation function, which calculates the value f by adding the values g and h, and the other one is a multiplicative evaluation function, which calculates the value f by multiplying the values g and h.
Usually, the additive evaluation function is used, because it is the common way for calculating the f value. With regard to the counterexample state space graphs, a multiplicative version for the evaluation function has been implemented to the model checker tool DiPro [5, 9].
For the isolated K* algorithm, both evaluation functions will be usable. The reason for this is as follows: By tracking a path on a graph the distance of the way increases. That is a positive progression. Therefore, only a summation or multiplication would be usual.

Figure 7 shows the implemented connectivity of these new classes to the context. The new classes `ContextComparator` and
`ContextEvaluationFunction` are connected with the context interface class `AbstractContext`, which calls the methods *loadComparator()* and *loadEvaluationFunction()* to get these properties from the users context file.

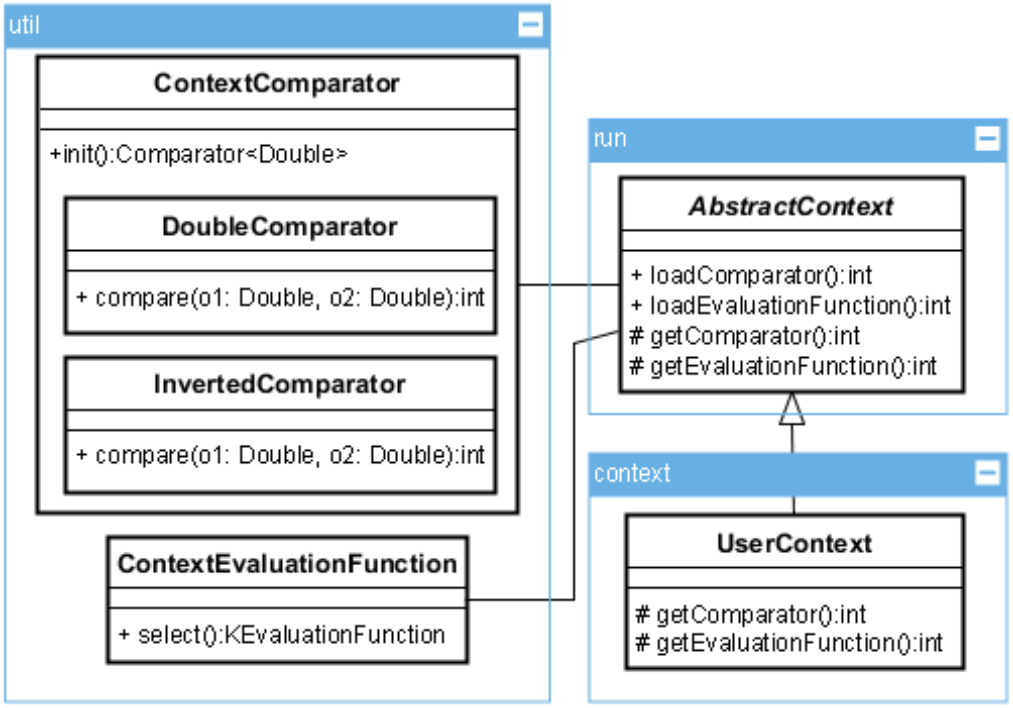

Figure 7: Integration of the new context classes.

New methods
The several solutionCollector files have changed to one common `KSolutionCollector` class, as default solution collector. Therefore, it is no longer necessary to load the solution collector in the context class. The solution collector will be created directly with a new instance by initializing the algorithm in the `BF` class.

The label of the solution output file is set to a static description. Therefore, the name of the given context file is used. In this case the method *getSolutionFileName()* is set to the `AbstractContext` class.

3.3.4. Solution Output

To get a clearly arranged summary of the K* algorithm the solution output has been adjusted. There are four several solution files possible now. These output files will be discussed in the following:

Search report
The search report is a logging file which was created for the tool DiPro [5, 9]. It provides five different logging levels, ranging from basic logging for some information to a detailed logging for debugging purposes. The logging can also be disabled.

Summary
The summary file was also created for the tool DiPro [5, 9]. For that, an own class file, the `AlgReporter`, was implemented. This class file gets a status report after each search iteration. In this version of the K* algorithm, the summary output file is adapted to be suitable for general use.

Each line shows:

- I = number of the current needed search iterations
- S = number of vertices which are currently explored
- E = number of edges which are currently explored
- V = value g of the current best trace
- N = number of current found solutions
- R = number of reopened vertices
- RT = complete runtime
- Sea_M = used search memory
- Sol_M = used solution memory
- Mem = complete used memory

Solution paths and path graph traces
With this option it, is possible to enable the output of all solution paths and the corresponding path graph traces, that are found by the K* algorithm. Therefore the solution collector class file has been changed to handle two output files at once. If the option to log the solution paths and path graph traces is set, two separate files will be created, the "_solution_paths" and the "_pathgraphtraces" file. In front of the first

underscore, a K* signature and the name of the used context file will be placed. The whole file name looks like this: ”kStar_search_<context name>_solution_paths.txt”. It can be selected between a normal text file ”.txt” or an XML file ”.xml”, as output file type.
A visual output of the solution is not implemented. Only the creation of output files is possible. Thus, a text file editor is needed to inspect these files. For the paths in the ”solution paths” file, the existing method *constructTrace()* is used now. This method was created for the tool DiPro [5, 9] and calculates all solution paths by considering the sidetracks.

3.4. Heuristic Interface

The K* implementation contains an abstract class `Heuristic` with the abstract method *public abstract double evaluate(Vertex v)*, which has been implemented by Husain Aljazzar [1] in DiPro. It has to be overwritten by the User in a given Java file, which extends from the `Heuristic` Java file. The `Heuristic` class is called in the search algorithms by relaxing the vertices. Therefore, the values h and f are calculated.
In case of the road map example in the case studies of this thesis, the used heuristic calculates the airline distance between two given vertices, which is described by Husain Aljazzar in his paper [1]. In the constructor of the `UserHeuristic` class the target vertex will be set. Every time calling the *evaluate()* method, the distance between the current given vertex and the target vertex will be calculated and returned.

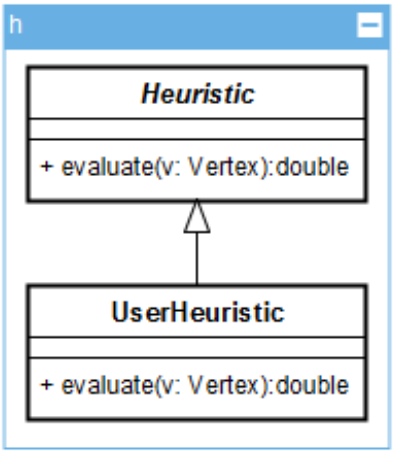

Figure 8: Structure of the heuristic interface.

3.5. Context Interface

To make the K* algorithm generally usable, it should be able to handle different types of graphs. Considering, keeping the usage of the K* algorithm for the user as simple as possible was one of the biggest challenges of this thesis. An acceptable agreement between a simple usage and an implementation, that is not too complex, is represented by the created context interface.
The context interface can be compared with a wizard who collects all needed information about the graph. It supplies all properties which differ for every graph. These properties are the graph itself, the property which indicates that the target is found and the specific vertex type.

The idea how this context interface is realized is based on the properties, which are needed by the K* algorithm to execute successfully. As mentioned in previous sections, the required properties for a search operation on the graph are known. Hence, for every graph a special type of vertex has to be implemented.
Moreover, every target solution of a given graph depends on his own set of properties, which have to be checked if they come true. In order to address this, a special property has to be implemented.
At last, the graph should be known. Thus, a class file has to represent it. The several Java class files are explained in detail as follows:

Context file
The context class file extends the abstract class `AbstractContext` and is constrained to implement the given abstract methods:

- **protected void readParameters():** This method reads the given parameters of the properties file which are given specifically for the graph.
- **protected void loadModel():** This method loads graph, property and start vertex.
- **protected DirectedGraph loadGraph():** This method loads the graph implemented by a class extending `DirectedGraph` class.
- **protected Proposition loadProperty():** This method loads the properties implemented by a class extending `Proposition` class.

- **protected <T> T getInitialState():** This method loads the start vertex. The type of the vertex should be given by a class which extends of `DefaultVertex` class, which in turn matches the graph node type.

- **protected int getComparator():** This method returns a flag for the comparator function. These flags are implemented in the `ContextComparator` class. The flag DOUBLE_COMP will be returned instead of a Double comparator while the flag INV_COMP will be returned instead of a inverted Double comparator.

- **protected int getEvaluationFunction():** This method returns a flag for the evaluation function class. These flags are implemented in the `ContextEvaluationFunction` class and can be enhanced if new types are needed. The flag ADDITIVE will be returned instead of the additive evaluation function f = g + h while the flag MULTIPLICATIVE will be returned instead of the multiplicative evaluation function f = g * h.

Graph file
The graph class file implements the `DirectedGraph` interface and has to implement its methods. In this class, the user can implement the graph, which is connected to a database or loaded from a local file, or something similar. Further, preprocessing steps can be made. The important methods to implement are:

- **public Iterator<? extends DirectedEdge> outgoingEdges (Vertex v):** This method returns all outgoing edges from the given vertex for the A* algorithm as an iterator. At this point the user can implemented preprocessing steps. For example, only edges are returned which keep a condition desired by the user.

- **public <T extends DefaultVertex> T getNode(int id):** This method returns the desired vertex by the given id.

- **public float weight(Edge e):** This method returns the weight of the current edge.

- **public void clear():** This method clears all used memory space after the algorithm has terminated.

Vertex file
As already mentioned in the preliminaries for the graph types, own properties exists for every graph. Hence, the vertex class file has to be implemented. It contains all necessary properties for the vertices of the graph. It extends the `DefaultVertex` class. Every vertex needs an unique id for the K* algorithm.

Property file
The property class file implements the `Proposition` interface. It holds and checks the target condition by the method *public int check(Vertex vertex)*, which has to be implemented by the user. The interface Proposition delivers three flags for checking. Firstly, the flag **never** if the current vertex never leads to a target. Secondly, the flag **false** if the current vertex is not a target vertex. Lastly, the flag **true** if the current vertex is a target vertex.

These four class files are the necessary context files, implemented by the user. If the user has to implement more files representing the graph, he/she is free to extend the context with more files. In Figure 9 a complete overview of the context structure is given and shows how the user files need to extend from the classes.

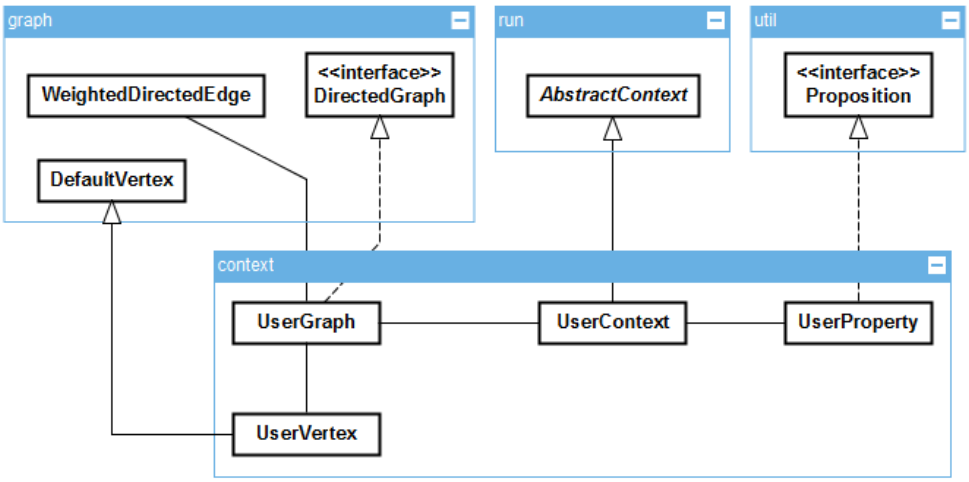

Figure 9: Class structure of the context package.

In the following section the use of the context folder will be explained by going through a simple example. The scenario is the road map graph used in the case studies of this thesis. It consists of four classes: RoadContext, RoadGraph, RoadNode and TargetPredicate.
The context class file `RoadContext` uses the *readParameters()* method to read the source vertex and target vertex. The *loadGraph()* and *loadProperty()* methods return a new instance of the graph class and property class, respectively.
The graph class file `RoadGraph` loads the graph from a given database. Hence, the connection is built up in the constructor of the class. For the execution of the K* algorithm the methods *outgoingEdges()*, *getNode()*, *weight()* and *clear()* are needed. The method *outgoingEdges()* selects all needed edges by the identification number of the source node and creates the edges as an instance of the class `WeightedDirectedEdge`. The initial vertex and all needed vertices are selected in the *getNode()* method. The method *weight()* returns the weight of the given edge and *clear()* closes the database connection, after all solution paths are found.
The class file `RoadNode` represents the type of the used vertex for this graph. In case of the road map the earth coordinates are needed. Hence, the created vertices have an identifier, the longitude and the latitude, as properties.
The property class `TargetPredicate` handles the condition of the target vertex. It checks, if the found identifier by the A* algorithm [11] is the desired target vertex. The structure of this road map example looks similar to Figure 9.

3.6. Database

Considering, using graphs or heuristics stored in databases the package "db" has been newly implemented. It consists of a database class for graphs, a database class for heuristics and a class with a database loader. The whole package and its implemented classes are shown in Figure 10. The implementation of this connection is based on the implementation description of [6]. To get a connection between the Java implementation and a database the Java Database Connectivity API (JDBC) [12] is needed. In this thesis, the given Java files in the database package and the given table structure in Figure 11 are optimized for the use of the case studies. This files can be changed by a user to run other types of graphs.

A usage of the class `DatabaseLoader` from the command prompt is not implemented.

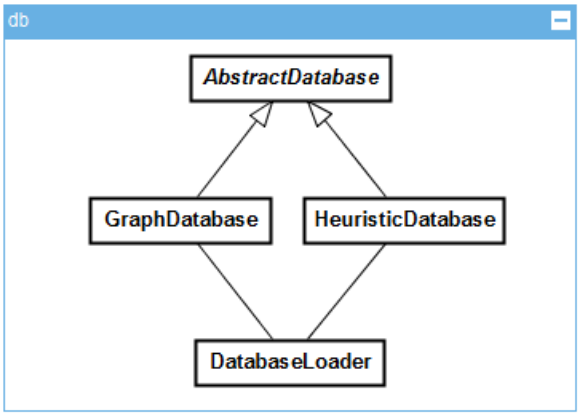

Figure 10: Database structure of the K* algorithm in package db.

The database package can be used for uploading graphs and heuristics to a database, defined by the user. Further, it enables to read graphs and heuristics from a database for the run of the K* algorithm. That makes it possible to use graphs and heuristics from remote locations. Figure 11 shows the used structure for the distances graphs of the case studies in this thesis.

A little example will demonstrate next, how the use of the database package works.
First of all, it is necessary to have a correct Java database connectivity driver for the database, to connect from the Java source code. For uploading a graph or a heuristic, the `DatabaseLoader` can be used. The connection parameters of the database just need to be inserted in the Java file and the main method can be executed.
To use a heuristic from a database the specific parameters have to be written to the properties file. In that case, not only the heuristic file name is needed. The whole command should include the token for the heuristic **-h**, the flag for loading from a database **-db** and then the connection parameters **heuristic name**, **heuristic url**, **heuristic driver**, **database user role** and **password** (if necessary) in exactly this order. Afterwards, heuristic file will be loaded from the database by the load method of the context interface.

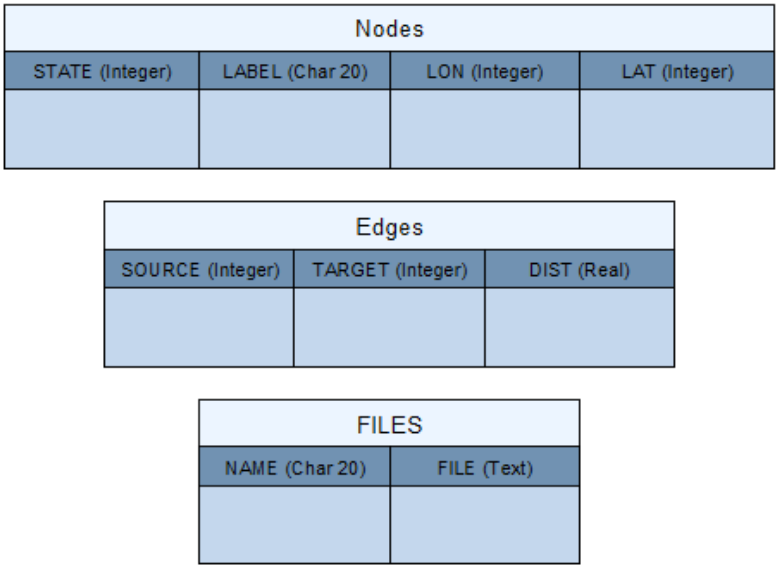

Figure 11: Database tables of the K* algorithm.

3.7. Compilation and Execution

The stand-alone version of the K* algorithm comes without any graphical user interface. It is build as a portable version. Hence, it is used by the command-line interface. The K* algorithm includes a Java class file, which has been newly implemented, for compiling the whole algorithm for execution. It is implemented and tested for the use on a Windows operating system or a Linux distribution. To use the K* algorithm the Java programming language compiler [14] is needed. If a usage with a database connection is wanted a Java database connectivity driver [12] is needed.
First, the whole K* algorithm has to be compiled. For that, the compile class file can be used. It will compile the whole source folder for an execution, afterwards. Subsequently, the K* algorithm can be started by calling the Main class in the run package with a property file as input parameter.
A complete step-by-step description can be found in the appendix A.1 of this thesis and in the attached user manual.

4. Case Studies

To show the efficiency of the newly isolated and modified version of the K* algorithm a few experiments will be conducted in this section.
The case studies are structured as follows: As a first experiment it will be proven that this version of the K* algorithm is not worse in terms of performance than Husain Aljazzar's implementation of the K* algorithm [1, 2]. As a second experiment the K* algorithm will be compared with the results of another implementation of the K* algorithm by the authors of the paper [7].
For the experiments the used graphs were stored in a relational database, like Husain Aljazzar did and the graphs represent road maps of USA, which are available on [16]. The used computer for the case studies is equipped with an Intel dual core 2.5 GHZ and 16 GB RAM. Notice, that this is not exactly the same equipment Husain Aljazzar used for his experiments. Thus, it should be considered, that the results have to be compared using approximation.

4.1. K* with a Distance Graph

For this experiment, the cases used by Husain Aljazzar were taken in order to be comparable [1, 2]. The K* algorithm runs four times with four different target vertices. Theses vertices lie in different directions. The number of solution paths is set to 1000 traces. For a heuristic the airline distance is used and the A* algorithm only grows 20% in each round of exploration. As mentioned before, the solution output has changed and will not be considered in the measurements. The results of the used memory space will only consider, like Husain Aljazzar did, the used search memory.

New York City
In the first round of the experiment the road map of New York City is selected. This map can be found on the homepage of the 9th DIMACS Implementation Challenge [16] and contains 264346 vertices and 733846 edges. The measurement considers the runtime in milliseconds and the maximum of needed memory space in kilobyte. The results are shown in Table 1 as well as Figure 12 and 13.

average runtime New York						
k	1	200	400	600	800	1000
ms	162782,5	176929,75	177052,25	177105,5	177157,25	177200
kB	4600326	5010109	5027253	5043962	5060352	5076430

Table 1: Average runtime in ms of the road map New York.

Table 1 shows the result for the New York City road map where k is the number of solution paths, ms is the runtime in milliseconds and kB is the memory space in kilobyte. The continuous line shows the result of this experiment and the dotted line shows the result of Husain Aljazzars' experiments [1] for comparison. In case of equal memory usage only the continuous line is visible in Figure 13.

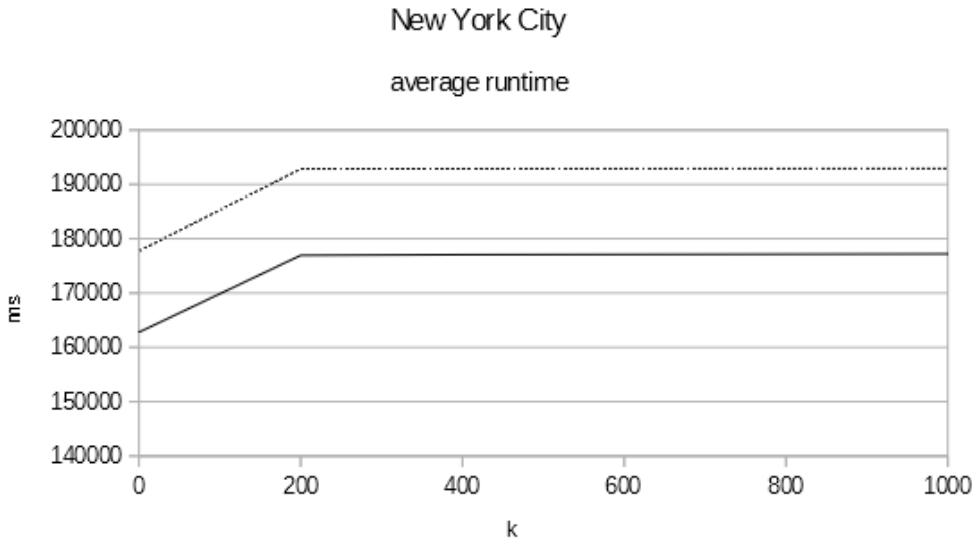

Figure 12: Average runtime in ms of the road map New York.

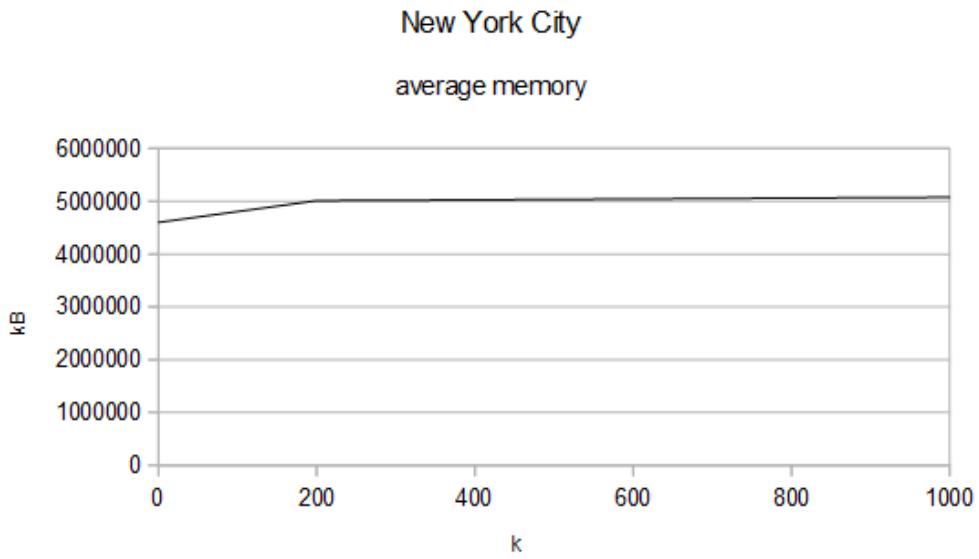

Figure 13: Average memory in kB of the road map New York.

Eastern USA
In the second round of the experiment the road map of the eastern USA, also from [16], is selected. This map contains 3598623 vertices and 8778114 edges. The results are shown in Table 2 as well as Figure 14 and 15. As well as in Figure 12, Figure 14 shows the result of this experiment by the continuous line and the result of Husain Aljazzars' experiments [1] are shown by the dotted line for comparison. In case of equal memory usage only the continuous line is visible in Figure 15.

average runtime East of USA						
k	1	200	400	600	800	1000
ms	722024,5	873806,75	873903,75	873968	874010,5	874053,25
kB	21038836	25316080	25332606	25348602	25364797	25381042

Table 2: Average runtime in ms of the road map East USA.

Table 2 shows the result for the Eastern USA road map where k is the number of solution paths, ms is the runtime in milliseconds and kB is the memory space in kilobyte. The increasing line in the first section of the diagrams is due to the fact that the A* algorithm is resuming in some runs of the K* algorithm.

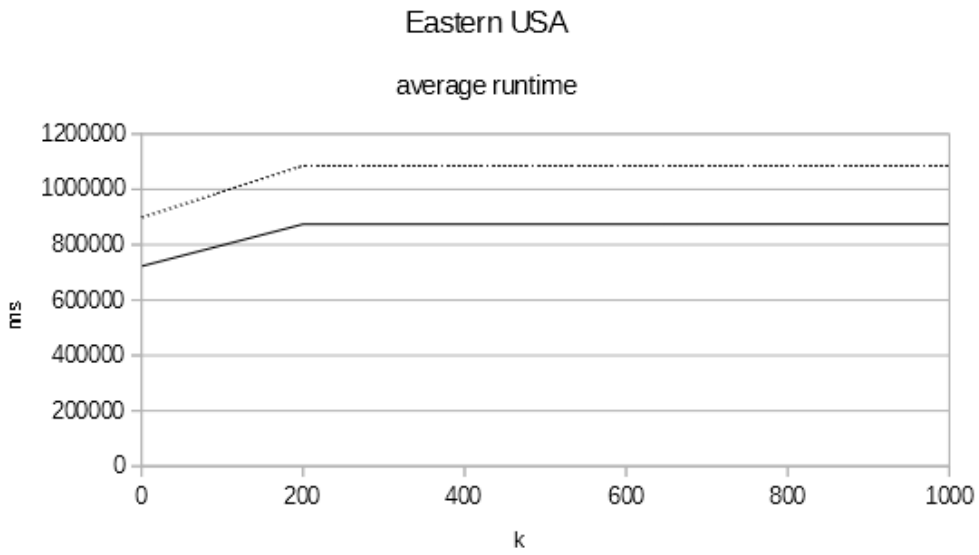

Figure 14: Average runtime in ms of the road map New York.

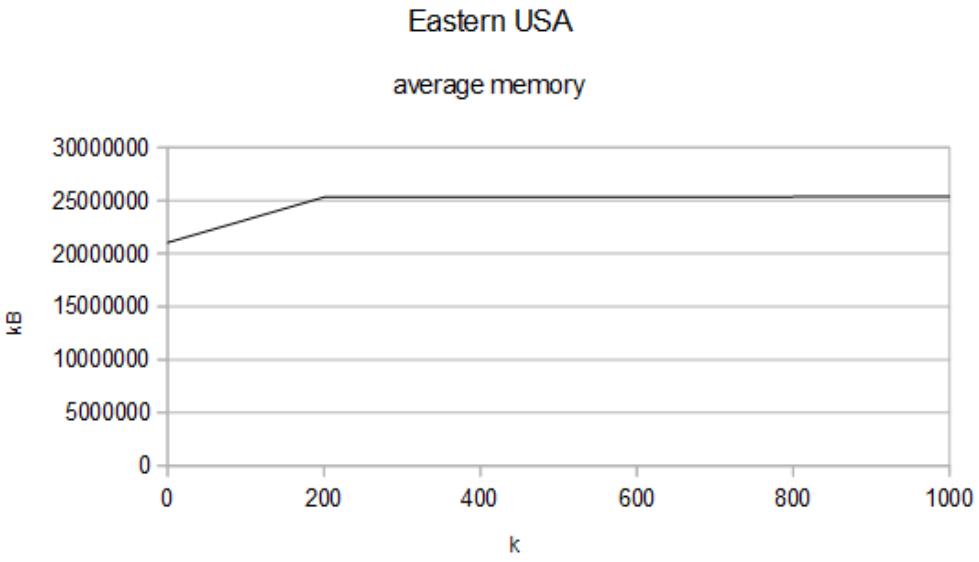

Figure 15: Average memory in kB of the road map New York.

4.2. Comparison with another K* Implementation

The second experiment delivers a comparison of this version of the K* algorithm and the implementation of the K* algorithm provided by the authors of [7]. The authors of [7] implemented a new k-shortest-paths algorithm and compared it to the K* algorithm. This experiment should prove if this version of the K* algorithm may be better in terms of performance.
For the experiment, the cases, the authors in [7] used, were taken. They took four road maps of the 9th DIMACS Implementation Challenge [16]. New York City, San Francisco Bay Area, Colorado and Florida. They generated fifty randomly selected s-t paths for each graph, where **s** is the source vertex and **t** the target vertex of a solution path. For each run the sixty optimal solution paths should be found and the airline distance, as heuristic evaluation function, will be used just as Husain Aljazzar did [1].

Because their newly invented algorithm operates not on-the-fly, this option was disabled for the K* algorithm in their experiments. To obtain a correct comparison of both K* algorithm this option will be disabled, too. Hence the complete graph will be loaded into the K* algorithm. The following Figures 16-23 show the runtime and memory results of the four road maps used in the experiment of this thesis. The results of the authors in [7] can be found in their paper.

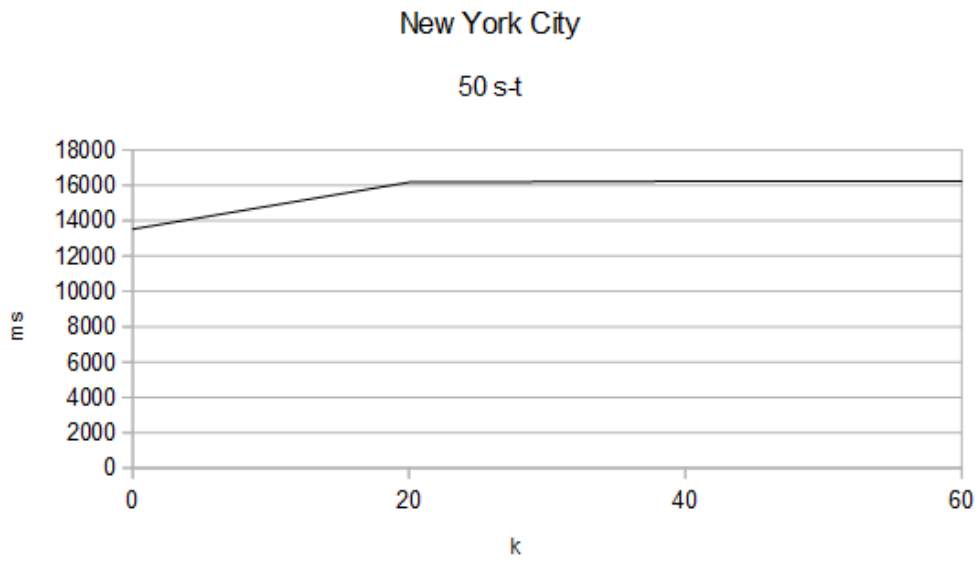

Figure 16: Average runtime in ms of 50 random runs New York City.

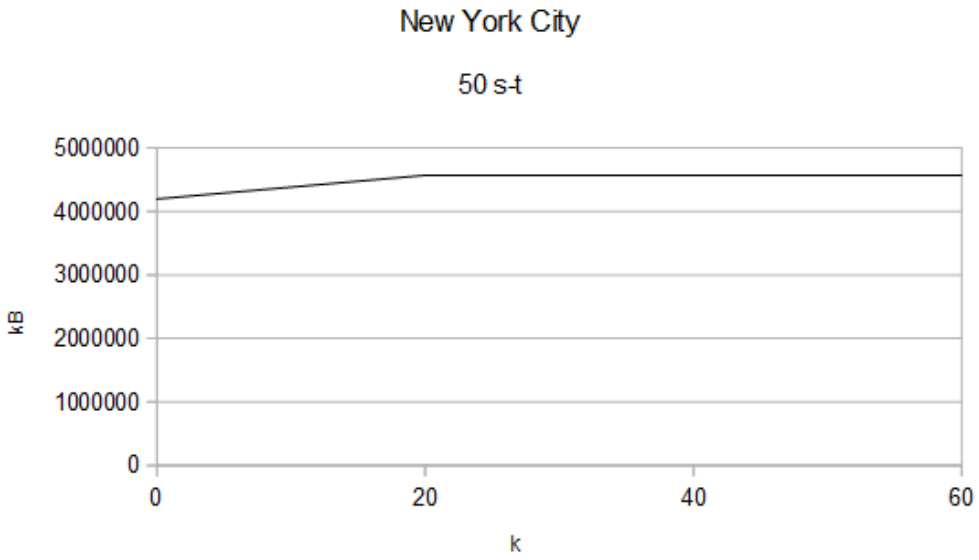

Figure 17: Average memory in kB of 50 random runs New York City.

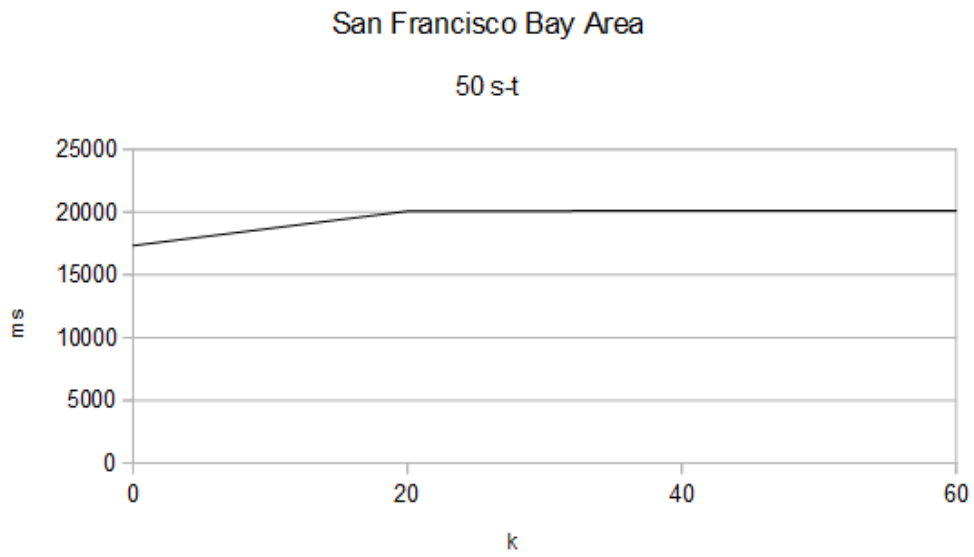

Figure 18: Average runtime in ms of 50 random runs San Francisco Bay Area.

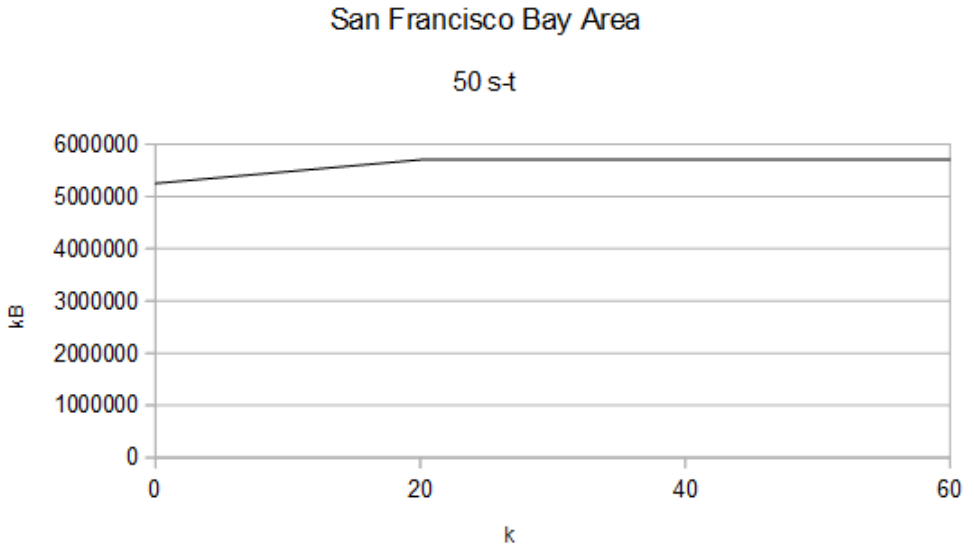

Figure 19: Average memory in kB of 50 random runs San Francisco Bay Area.

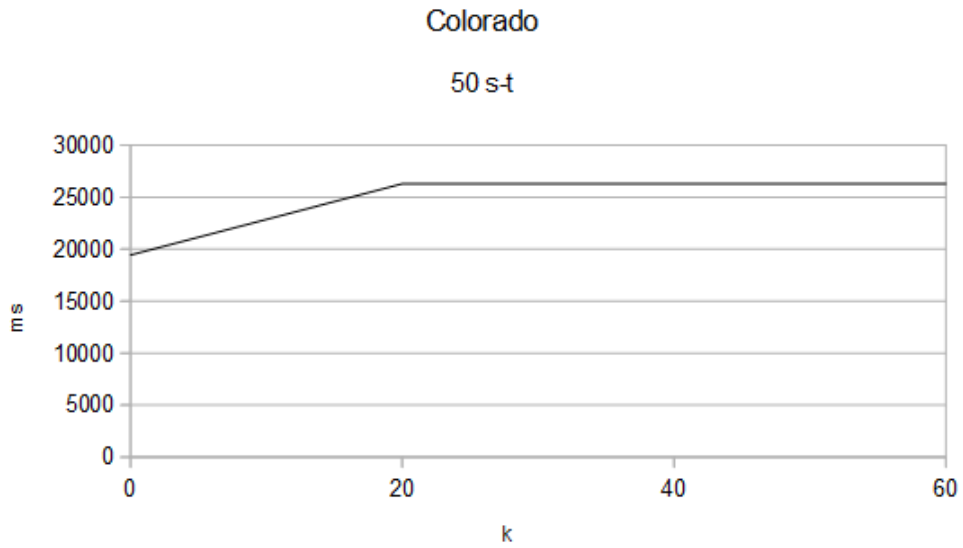

Figure 20: Average runtime in ms of 50 random runs Colorado.

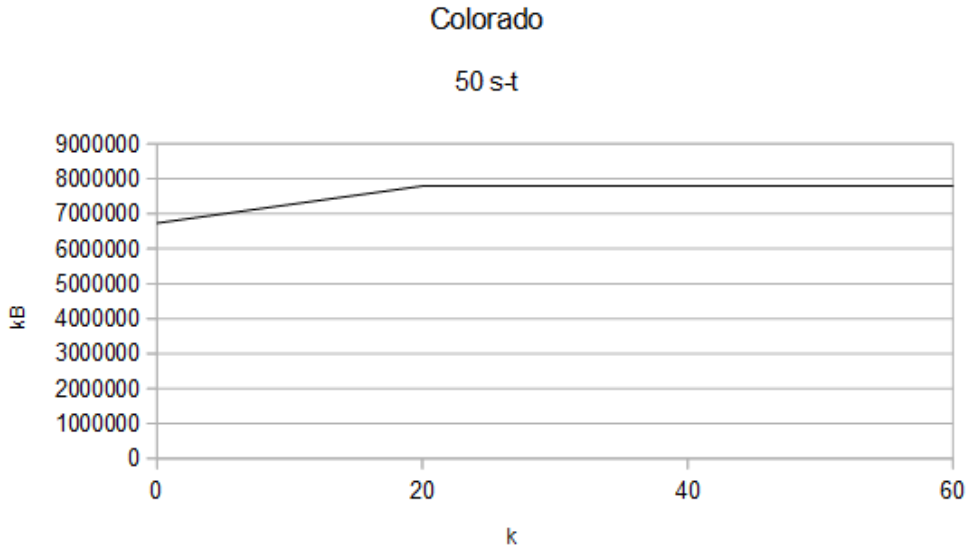

Figure 21: Average memory in kB of 50 random runs Colorado.

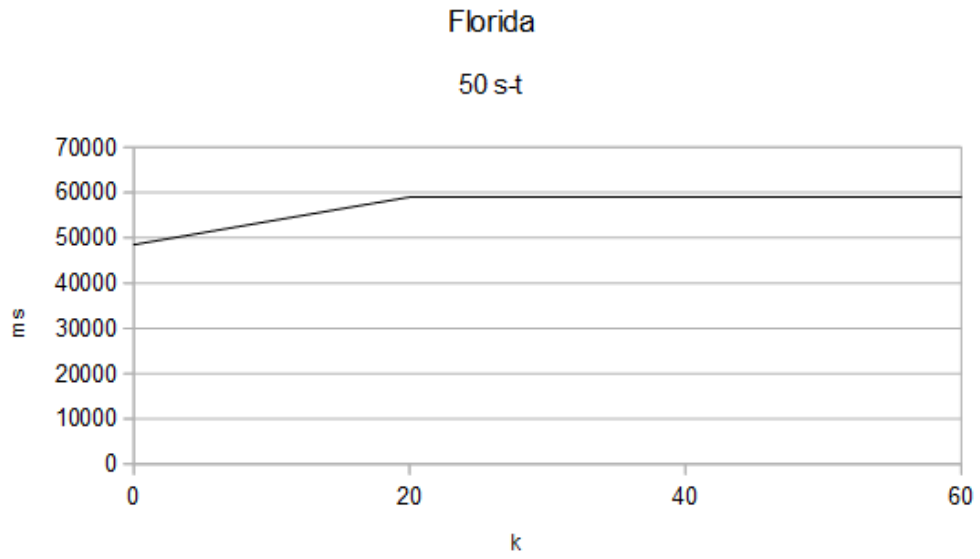

Figure 22: Average runtime in ms of 50 random runs Florida.

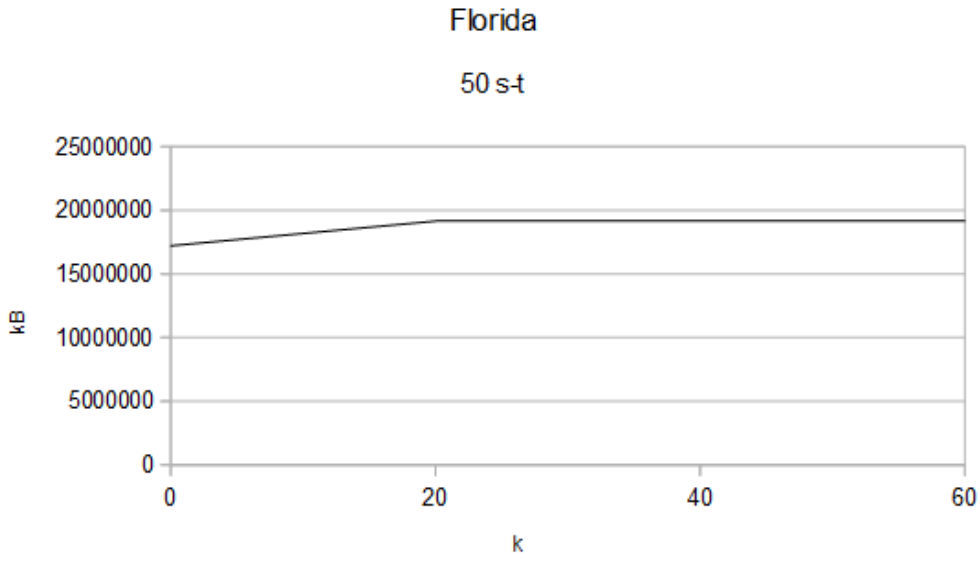

Figure 23: Average memory in kB of 50 random runs San Florida.

4.3. Results Discussion

Taking a look at the results in the first experiment, it is obvious that the used memory space compared to the results of Husain Aljazzar's experiments [1] is the same size. This result was expected because no changes were made at the search classes of the K* algorithm. In case of the solution output the maximum used memory space of the whole K* algorithm is not known from the results of Husain Aljazzar's experiments [1]. A look at the runtime results shows that the isolated K* algorithm is faster. The reason for this result depends on the different database connections. By comparing the summary reports, it appears that the new runtimes are slower in the beginning of the run. More time is needed to get the sql results from the database. But the more of the graph is explored, the faster the search will be.

For the second experiment the very same comparison is not possible. In case of the random built s-t paths of the case studies in [7] it is not possible to get the same s-t paths, because they are not known. Hence, only an approximation to the experiments is possible. Not the same runtimes and therefore not the same memory usage were reached for the experiment in this thesis. It is not known how long the s-t paths in the experiments in [7] were. In addition, the CPU of the computer used for the experiments in this thesis performs not as fast as the CPU of the authors in [7] does. Hence, not the runtimes and memory usages are compared, but the proportion of the road maps used for this experiments.

The Figures 16 - 23 show the results of the runtime and memory usage for the random generated 50 s-t paths, for all four road maps New York City, San Francisco Bay Area, Colorado and Florida. Setting the focus on the proportions between the four road maps by runtime, the experiment from [7] and the experiment in this thesis have nearly the same result. There will not be a conclusion about memory usage because it differs by the path on the graph. Some s-t paths are short in their distance from the source vertex to the target vertex, but the graph needs to be more explored by the A* algorithm [11] than some longer s-t paths with a higher distance. Thus, the memory usage is not significant for this comparison.
One crucial point known from these experiments is the creation of the path graph structure. The summary reports show, the more of the graph

is explored the longer the path graph structure needs to be built. Most of the needed runtime is spend on this step.

5. Conclusion

5.1. Conclusion

Until now, the K* algorithm was only usable as a part of the model checker tool DiPro [5, 9] in combination with a few graph types. It was the aim to get an isolated K* algorithm as stand-alone version to use it irrespective of the given graph type. In this thesis, it was shown how the K* algorithm has been isolated from the model checker tool DiPro [5, 9]. It was shown how the implementation works and which parts had to be adapted for a general use. In order to make this general use possible two abstract interfaces were implemented.
This new version of the K* algorithm is usable by the command-line interface. It is irrelevant which operating system is wanted to be used.

The test cases yield the conclusion, that there are no significant differences to the original K* algorithm in terms of runtime or memory usage. Furthermore, they show that this K* algorithm is the most efficient implementation at this time.

5.2. Future Work

The strength of the new K* algorithm lies in the independence as shown in this thesis. Hence, it would be a possible option to re-include this version of the K* algorithm back to the model checker tool DiPro [5, 9]. Furthermore, it could be included in other tools as well. Next, the context interface could be reduced in the number of needed Java classes. Therefore, the structure has to be changed. Lastly, a graphical user interface would be an addition, which replaces the use of the command-line interface. Moreover, the graphical user interface could provide a visualization of the solution paths as a completion to the solution output files.

References

[1] Aljazzar, H.: Directed Diagnostics of System Dependability Models. *Doctoral Dissertation*, 2009, chap. 2 + 5 + 7.

[2] Aljazzar, H., Leue, S.: K*: A Heuristic Search Algorithm for Finding the k Shortest Paths. *Artificial Intelligence, Volume 175, Number 18*, December 2011.

[3] Eppstein, D.: Finding the k Shortest Paths, *SIAM J. Computing*, 1998.

[4] Herold, H., Lurz, B., Wohlrab, J.: Grundlagen der Informatik, Pearson, 2012, 2nd version.

[5] Aljazzar, H., Leitner-Fischer, F., Leue, S., Simeonov, D.: DiPro - A Tool for Probabilistic Counterexample Generation. *In Proceedings of 18th International SPIN Workshop on Model Checking of Software* (SPIN 2011), 2011.

[6] Kemper, A., Eickler, A.: Datenbanksysteme, Eine Einführung, Oldenbourg Verlag, 2011, pages 144-151, 8th version.

[7] Liu, G., Qiu, Z., Qu, H., Ji, L., Takacs, A.: Computing k shortest paths from a source node to each other node, Springer Verlag, 2014, pages 2391-2402.

[8] Pearl, J.: Heuristics - Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1986, page 3, 46-49, 75, 99-110.

[9] DiPro, https://se.uni-konstanz.de/research1/tools/dipro/, 30.12.2016.

[10] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs. In *Numerische Mathematik*, 1959, pages 269-271.

[11] Hart, P. E., Nilsson, N. J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, 1968, pages 100-107.

[12] JDBC - Java Database Connectivity http://www.oracle.com/technetwork/java/javase/jdbc/index.html, 12.12.2016.

[13] Oracle Java Documentation, https://docs.oracle.com/javase/8/docs/, 19.12.2016.

[14] javac - Java programming language compiler http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, 12.12.2016.

[15] GNU Puplic Licence https://www.gnu.org/licenses/gpl.html, 15.12.2016.

[16] Demetrescu, C., Goldberg, A., Johnson, D.,: 9th DIMACS Implementation Challenge - Shortest Paths 2006 http://www.dis.uniroma1.it/challenge9/, 27.11.2016.

A. Appendix

A.1. Compilation manual

The K* algorithm comes without any graphical user interface. It is prepared to be used via command-line interface and is implemented for a use on a Windows system or a Linux system. Hence, an own compilation Java class is created. For compiling the K* algorithm, only the `Compile` class and the source folder are needed. The wanted heuristic Java file should be put into the source folder *h* and the whole graph files into the folder *context*. To compile and execute the K* algorithm the following steps should be done:

1. Open the command prompt (or Linux shell) and move to the directory where the source folder and Compile file is stored. It is important that the source folder and Compile Java file are in the same directory. The source folder contains all Java files for the K* algorithm. Put the heuristic file into the folder "src/h" and the complete graph context files into "src/context".

2. Compile all files by using the command

   "java Compile"

   The binary folders will be created.

3. To run the K* algorithm use the command

   "java run.Main [[prop file]]"

   where the prop file can be every readable text file e.g. prop.txt. The properties file should include all needed commands for the execution. For help take a look to the section properties below.

4. The K* algorithm is running and the result will be printed on the command prompt.

The path for the Javac compiler has to be set. Without the Javac compiler the class `Compile` can't be used. If the user desires to run the K* algorithm with a graph stored in a database, a Java database connector is needed. This connector has to be set to the classpath, too. If not, the

following commands should be tried to compile the K* algorithm:

"java Compile [[libs]]"

where libs should be the Java database connector. And run the algorithm with:

"java -cp .;[[path of lib]] run.Main"

where the complete path of the library file should be given. Notice, that the semi-colon is used on Windows systems. Use a colon on Linux systems.
It is not necessary to compile everything again, if the folder has been compiled once. To get the compilation working all needed folders in the source folder have to be filled with Java classes.

A.2. Properties manual

To run the K* algorithm with specific properties a property file with the following options can be used:

-greedy the algorithm runs in greedy mode which means that the value f is only calculated by the heuristic value h. The needed cost of the way won't be considered.

-complete the algorithm runs in complete mode which means that the whole graph will be explored.

-prune n the algorithm prunes nodes which f-value is not better than the set value n.

-maxiter n the algorithm stops after n iterations and ends without any solution if no solution path is found in time.

-maxtime n the algorithm stops after n minutes and ends without any solution if no solution path is found in time.

-k n n solution paths should be found.

-h n the given heuristic Java class n will be used for search.

-c n the given context Java class n will be used for graph.

-solTrace n the found solution paths should be logged in a solution output file. n can be set as txt for text file or xml for xml file.

-log n the logging level can be set from 0 to 5. 0 = disabled; 1 = basic; 2 = normal; 3 = detailed; 4 = verbose; 5 = debug.

In case of the given context file for the graph the user needs to put in specific properties like start vertex and target vertex.

A.3. DVD