Karlsruhe Institute of Technology

Linear Space All-Pairs Shortest-Paths Computation on Road Networks

Study Thesis of

Jan-Ole Sasse

At the faculty of Computer Science
Institute of Theoretical Informatics
Algorithmics I

Reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	Dipl.-Math. Reinhard Bauer
Second advisor:	Dipl.-Inform. Thomas Pajor

Processing Time: 21. September 2009 – 21. March 2010

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related work . . . 1

2 Basics **3**
- 2.1 Preliminaries . . . 3
- 2.2 The *2-core* . . . 3
- 2.3 The Linear Space All-Pairs Shortest-Paths Problem . . . 6
- 2.4 A simple solution . . . 6

3 Algorithm **7**
- 3.1 CoreAPSP . . . 7
- 3.2 Stage 1: *2-core* extraction . . . 7
- 3.3 Stage 2: Rebuilding the graph and distance precalculations . . . 7
- 3.4 The distance vector calculation algorithms . . . 9
  - 3.4.1 CoreDijkstra . . . 9
  - 3.4.2 treeNode . . . 11
  - 3.4.3 pathNode . . . 11
- 3.5 Stage 3: The scheduler . . . 13

4 Experimental Evaluation **17**
- 4.1 Hardware . . . 17
- 4.2 The implementation . . . 17
- 4.3 The Testset . . . 17
- 4.4 Results . . . 19
  - 4.4.1 Precalculation time . . . 20
  - 4.4.2 Graph structure . . . 20
  - 4.4.3 The Schedulers . . . 21
  - 4.4.4 Runtimes on variable sized graphs . . . 24
  - 4.4.5 Runtimes with different numbers of concurrent distance vectors . . . 25

5 Conclusion **29**

Bibliography **31**

1. Introduction

1.1 Motivation

Route planning is omnipresent nowadays. Big logistics companies optimize their itinerary with the help of computers and many car drivers rely predominantly on their navigation system. The demand of path advices is growing and navigation software has to be improved to keep up with it. While it is possible today to retrieve a shortest path very fast, all of the advanced techniques rely on precalculated data and some of them need to calculate all shortest paths to determine it. These calculations can last very long, making it time-consuming to process large graphs. Additionally, it is impossible to store all of the data, because the number of shortest paths is quadratic to the number of nodes the graph contains. So, what is needed, is a fast algorithm to compute the paths while using only constant-size memory in doing so.

This study thesis deals with such an algorithm. It exploits the structure of road networks to gain information about spots the routes have to pass and uses information from other shortest paths during the calculation of new ones. Depending on the graph, computation times can thereby be decreased by a factor up to five on average compared to a classic implementation of Dijkstra's algorithm.

1.2 Related work

Finding a shortest path is one of the oldest and most studied problems in graph theory. Till today, Edsger Wybe Dijkstra's algorithm from 1959 [Dij59] still is one of the most important approaches to solve the problem on weighted and directed graphs. The algorithm has been able to stay on top of the game that long because it is rather a calculation scheme than a precise specification, and the performance of the used data structures was improving over time. The state of the art priority queue in the general case is the Fibonacci Heap of Fredman and Tarjan from 1987 [TF87], yielding a time bound of $O(m + n \log n)$ for Dijkstra's algorithm, where n is the number of nodes and m the number of edges in the graph. If the edge weights are bounded to integer values between 0 and C, Dial proposed an $O(m + nC)$ time algorithm using bucket queues [Dia69]. An advantage of Dijkstra's algorithm is that it not only calculates one shortest path but solves the Single Source Shortest Paths problem, where all shortest paths from one source to all nodes of the graph are desired.

The All-Pairs Shortest-Paths (APSP) problem deals with the task of calculating the shortest path for all pairs of nodes. This study thesis deals with the case of the problem where the graph is directed and weighted with non-negative edge weights. Note that a graph with negative edge weights, but without negative cycles, can be transformed into a new graph without negative edge weights and preserved shortest paths in $O(mn)$ time with Johnson's algorithm [Joh77]. The easiest way of solving the APSP problem is solving the Single Source Shortest Paths problem for all nodes of the graph, needing $O(nm + n^2 \log n)$ time for real and $O(mn + n^2 C)$ for integral edge weights. Karger, Koller and Phillips [KKP93] as well as McGeoch [McG95] tuned this approach to $O(m^* n + n^2 \log n)$, where m^* is the number of edges taking part in any shortest path. Pettie improved on this bound by introducing an $O(mn + n^2 \log \log n)$ time algorithm [Pet04]. If the Word RAM model is taken as the basis for the calculations, Hagerup [Hag00] obtained an $O(mn + n \log \log n)$ time algorithm.

Furthermore algorithms based on matrix multiplication have been developed (see [Zwi01] for further references), but are not introduced here because our approach focuses on limited space consumption. Also approaches based on scaling [Gab85, GT89, Gol95] or fast integer sorting [Tho04] are skipped here, because they only work with integral edge weights.

2. Basics

2.1 Preliminaries

Throughout the work $G = (V, E, len)$ denotes a directed and weighted graph with n nodes, m edges and a positive length function $len : E \to \mathbb{R}^+$. Given a node v, $\mathcal{N}(v)$ denotes the set of neighbors of v, that is the set of nodes $u \in V$ such that $(u, v) \in E$ or $(v, u) \in E$. The cardinality of $\mathcal{N}(v)$ is called degree of v. Note that $u \in \mathcal{N}(v)$ implies $v \in \mathcal{N}(u)$. Given a set S of nodes, the *neighborhood* of S is the set $S \cup \bigcup_{u \in S} \mathcal{N}(u)$. A path P from x_1 to x_n in G is a finite sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of nodes such that $(x_i, x_{i+1}) \in E$, $i = 1, \ldots, n-1$. Given a sequence $p = \langle x_1, x_2, ..., x_n \rangle$ we write $v \in p$ if there is an i such that $x_i = v$ and if the sequence represents a path, we say v is on that path. The *length* of a path P in G is the sum of the length of all edges in P. A shortest path between nodes s and t is a path from s to t with minimum length. By $P(s, t)$ we denote the set of all shortest s-t-paths. Given two nodes s and t the distance $dist(s, t)$ from s to t is the length of a shortest path between s and t or infinity, if no path exists.

Given a graph $G = (V, E, len)$ and a subset $V' \subseteq V$, the node induced subgraph $G_{V'} = (V', E', len)$ consists of V', the subset E' of E for which both their source and their target nodes are in V', and len, the length function of G.

The corresponding simple, unweighted, undirected graph $G' = (V', E')$ of a directed and weighted graph $G = (V, E, len)$ consists of $V' = V$ and $E' = \{\{u, v\} | (u, v) \in E\}$, which is the set containing a corresponding, undirected edge for every edge $e \in E$.

Given a graph $G = (V, E, len)$ and two nodes v and w in V, we say that v and w are connected if there exists a path from v to w in G. If no path exists, v and w are not connected. G is connected, if all nodes of its corresponding simple, unweighted, undirected graph G' are pairwise connected.

2.2 The *2-core*

Given a weighted graph $G = (V, E, len)$ and a source node s in V, the distance vector is a function that maps every target node t in V to the shortest path distance from s to t in V. This section deals with the theoretical analysis of our speedup techniques for distance vector calculation.

Examining a graph that represents a road network, one can notice that many nodes are located in regions which only have one or two paths connecting them with the rest of the

graph. For example, highways are modeled as sequences of edges between the slip roads and nodes in dead ends only have one path leading back to the rest of the street network. This chapter introduces the *2-core* and the *2-core* degree, two definitions that help to identify nodes for which such cases apply.

The *2-core* was first introduced by Bollobas [Bol84] in the more general form of a K-core and can be used to help with the analysis of graph structures and properties. In our case, it is used to identify subgraphs forming trees or paths.

Definition 2.2.1 *The* 2-core *of an undirected graph is the maximal node induced subgraph of minimum node degree 2. The* 2-core *of a directed graph is the* 2-core *of the corresponding simple, unweighted, undirected graph.*

Nodes that are part of the *2-core* are called core nodes and those not being part are called 1-shell nodes. The set of all 1-shell nodes is called the 1-shell. Note that connected components within the 1-shell are trees. We distinct two types of trees.

Definition 2.2.2 *Given a graph G and its corresponding simple, undirected, unweighted graph G', an* unattached tree $\mathbb{T} = (V_{\mathbb{T}}, E_{\mathbb{T}})$ *is a maximal, connected, node induced subgraph of G' such that all nodes $v \in V_{\mathbb{T}}$ are part of the 1-shell and are not connected to a node from the* 2-core *of G'.*

Definition 2.2.3 *Given a graph G, its corresponding simple, undirected, unweighted graph G' and a core node $r \in V$, the* attached tree *rooted at r is the maximal, connected, node induced subgraph $\mathbb{T}_r = (V_{\mathbb{T}_r}, E_{\mathbb{T}_r})$ of G' such that $V_{\mathbb{T}_r}$ contains r and $V_{\mathbb{T}_r} \setminus r$ is a subset of the 1-shell of G'.*

Note that $\mathbb{T}_r$ can only contain r, that all nodes in unattached trees only have 1-shell nodes in their neighborhood, and that all nodes in attached trees, except the root node, also only have 1-shell nodes or the root node in their neighborhood. The distances from a source node that is a member of an unattached tree to all target nodes t in V can be calculated in $O(m)$ time using a depth first search, as all paths are unique. The distance to all nodes that are not members of the same unattached tree is infinity. If the source is located in an attached tree, all distances to nodes inside the tree can also be calculated via a depth first search that is limited to 1-shell nodes and r. To determine the leftover distances, the following Lemma 2.2.4 can be used.

Lemma 2.2.4 *Given a graph $G = (V, E, len)$, an arbitrary attached tree $\mathbb{T}_r$ with root node r, a source node $s \in \mathbb{T}_r$ and a target node $t \notin \mathbb{T}_r$ or a source node $s \notin \mathbb{T}_r$ and a target node $t \in \mathbb{T}_r$, it holds for every shortest path that $dist(s,t) = dist(s,r) + dist(r,t)$.*

Proof Let $p = \langle s, v_1, v_2, ..., v_n, t \rangle \in P(s,t)$ be an arbitrary shortest path from s to t. We distinct the following cases:

- $s \in \mathbb{T}_r$, $t \notin \mathbb{T}_r$: If $i \in \mathbb{N}$ is the highest index such that v_i is in $\mathbb{T}_r$, v_i has to be r because it is the only node in $\mathbb{T}_r$ that has nodes in its neighborhood that are not from within the same attached tree. By the shortest path property, that states that every sub-path of a shortest path also has to be a shortest path, and according to the fact that r has to be on this shortest path, it follows that $dist(s,t) = dist(s,r) + dist(r,t)$.
- $s \notin \mathbb{T}_r$, $t \in \mathbb{T}_r$: If $i \in \mathbb{N}$ is the lowest index such that v_i is in $\mathbb{T}_r$, v_i has to be r. The rest follows in analogy to the first case.

Lemma 2.2.4 implies an algorithm that can calculate distance vectors of source nodes that are located in trees but are not the root of an attached tree by initializing the distance vector with $dist(s,r)+dist(r,t)$ for all nodes $t \in V$ before updating the local distances via a depth first search. This procedure takes $O(m)$ time, if the distance vector of the root node and the distance from the source node to the root node are available. CoreAPSP makes use of such an algorithm. It is called TreeNode and will be introduced in section 3.4.2. The fact that all paths leading into an attached tree have to pass the tree root is used to speed up CoreDijkstra which is introduced in Section 3.4.1.

For the following Lemma 2.2.7, we need to extend the notations of neighborhood and degree in the *2-core* context. The set $\mathcal{N}_{\text{core}}(v)$ of a node v contains all nodes from its neighborhood $\mathcal{N}(v)$ that are part of the *2-core* and is called *2-core* neighborhood. The *2-core* degree $\deg_{\text{core}}(v)$ of a node v is the cardinality of $\mathcal{N}_{\text{core}}(v)$ and is only defined for nodes that are part of the *2-core* themselves. The *2-core* degree helps us to identify graph structures from within the *2-core*, that have properties we can make use of, like *2-core* paths from Definition 2.2.5.

Definition 2.2.5 *A* 2-core *path is a sequence of nodes* $\mathrm{P}_2 = \langle v_1, v_2, ..., v_l \rangle$ *that is a path in the corresponding simple, unweighted, undirected graph for which all nodes* $v \in \mathrm{P}_2$*, except* v_1 *and* v_l*, have* 2-core *degree two. Both these nodes are called separators and have a* 2-core *degree greater than two.*

Note that all nodes except the separators only have their predecessor and their successor from the sequence in their *2-core* neighborhood, that all nodes except the separators have to be pairwise distinct and that there neither has to be a directed path between v_1 and v_l, nor between v_l and v_1. All nodes that are member of a *2-core* path and are none of the separators, are called path nodes. Figure 2.1 shows a little example graph with marked 1-shell and path nodes.

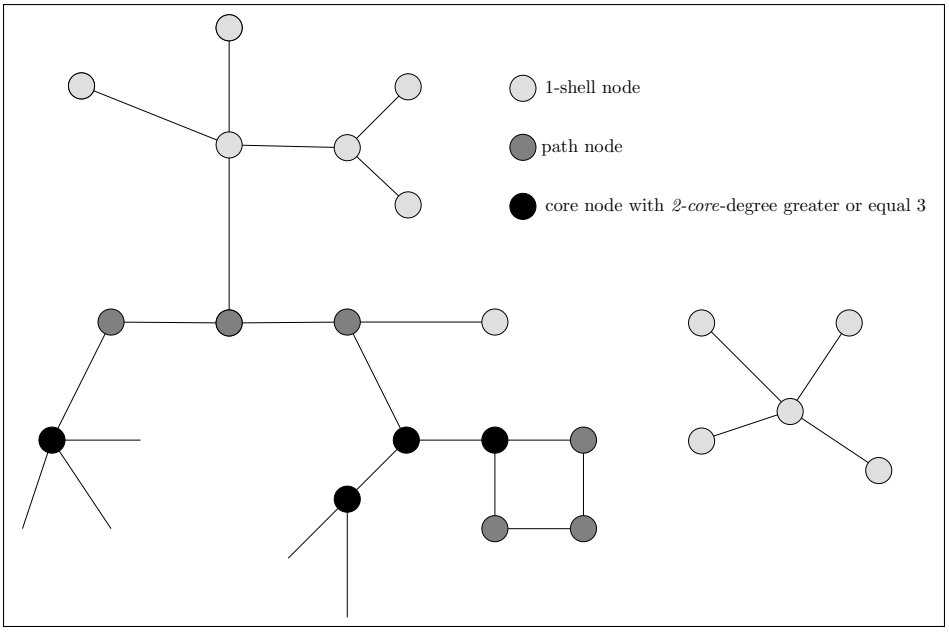

Figure 2.1: Example graph with tagged nodes

Every node v on a *2-core* path can be the root of an attached tree. Recall that $\mathbb{T}_v$ is the set that contains v and all nodes inside the tree that v is the root of and that a tree can only contain the root node. To simplify Lemma 2.2.7, we first introduce the following Definition 2.2.6.

Definition 2.2.6 *The* 2-core *path region* $\mathcal{R}_{path}(\mathrm{P}_2)$ *of a* 2-core *path* P_2 *is the union of* $\mathbb{T}_v$ *for all* v *on* P_2.

Lemma 2.2.7 *Given a weighted graph* $G = (V, E, len)$*, an arbitrary* 2-core *path* P_2 *in* G *with its separators* v_1 *and* v_l*, a source node* $s \in \mathcal{R}_{path}(\mathrm{P}_2)$ *and a target node* $t \notin \mathcal{R}_{path}(\mathrm{P}_2)$ *or a source node* $s \notin \mathcal{R}_{path}(\mathrm{P}_2)$ *and a target node* $t \in \mathcal{R}_{path}(\mathrm{P}_2)$*, it holds that the shortest path distance from* s *to* t *equals* $min(dist(s, v_1) + dist(v_1, t), dist(s, v_l) + dist(v_l, t))$.

Proof Let $p = \langle s, u_1, u_2, ..., u_n, t\rangle$ be a shortest path from s to t. We distinct the following cases:

- $s \in \mathcal{R}_{path}(\mathrm{P}_2)$, $t \notin \mathcal{R}_{path}(\mathrm{P}_2)$: Let $i \in \mathbb{N}$ be the highest index for which u_i is in $\mathcal{R}_{path}(\mathrm{P}_2)$. As all nodes from $\mathcal{R}_{path}(\mathrm{P}_2)$ that are located in attached trees only have nodes from the tree or the root node in their neighborhood, u_i has to be a node out of P_2. As in P_2 only v_1 and v_l have neighbors that are not in $\mathcal{R}_{path}(\mathrm{P}_2)$, u_i either has to be v_1 or v_l. The shortest path property implies that $dist(s, t)$ is either $dist(s, v_1) + dist(v_1, t)$ or $dist(s, v_l) + dist(v_l, t)$. As we are searching for the shortest path, it has to be the one with the smaller length.
- $s \notin \mathcal{R}_{path}(\mathrm{P}_2)$, $t \in \mathcal{R}_{path}(\mathrm{P}_2)$: If $i \in \mathbb{N}$ is the lowest index such that u_i is in $\mathcal{R}_{path}(\mathrm{P}_2)$, u_i either has to be v_1 or v_l. The rest follows in analogy to the first case.

Note that the distance between two nodes is infinity if no path exists. Lemma 2.2.7 implies an algorithm that can calculate distance vectors of source nodes located on a *2-core* path P_2 by initializing the distance vector with $\min(dist(s, v_1) + dist(v_1, t), dist(s, v_l) + dist(v_l, t))$, before using a limited depth first search to update the distances to nodes in $\mathcal{R}_{path}(\mathrm{P}_2)$. This procedure takes $O(m)$ time, if the distance vectors for v_1 and v_l and the distances from s to v_1 and v_l are already present. CoreAPSP makes use of such an algorithm. It is called PathNode and will be introduced in section 3.4.3. The fact that all paths entering a *2-core* path region have to pass one of the separators will be exploited in CoreDijkstra from Section 3.4.1 to decrease the number of priority queue operations.

2.3 The Linear Space All-Pairs Shortest-Paths Problem

The problem of finding all shortest paths, from every node to every node, is as well known as the problem of finding a single shortest path. As computation time is not the only factor limiting the size of processable problem instances, some more restrictions have been added to the definition of the problem.

Definition 2.3.1 *Given a graph* $G = (V, E, len)$*, the Linear Space All-Pairs Shortest-Paths problem is that of calculating the distance vectors* $\vec{d}_s$ *for all nodes* $s \in V$*, using only* $O(n+m)$ *space in total to store the graph representation and the results of the computations.*

Note that it is, according to the problem definition, not always possible to have all distances stored after the calculations are done, because this would require $O(n^2)$ space. Hence this is only possible, if m is in $\Theta(n^2)$.

2.4 A simple solution

The Linear Space All-Pairs Shortest-Paths problem can be solved in $O(n(n \log n + m))$ time by using Dijkstra's algorithm [Dij59] n times, whereas every node $s \in V$ is the source node once. As Dijkstra's algorithm only needs $O(n)$ space and the calculations for the different sources are independent, the used space can be freed after all reachable nodes for a source have been settled. The timebound is derived from the fact that one run of Dijkstra's algorithm requires $O(n \log n + m)$ time when using a Fibonacci Heap [TF87].

3. CoreAPSP

The CoreAPSP algorithm can be divided into three stages. The first one, is the extraction of the *2-core* and the determination of the *2-core* degree for all core nodes by the use of Extract2core. Afterwards, the representation of the graph is changed and some data is precalculated. The target is both to store node information closely together that is accessed in succession to lessen the number of cache misses, and to reduce overhead. Also information about incoming edges is discarded, because the algorithm does not need it after the precalculations are done. Section 3.3 describes the process in detail. The third stage is the main part of the algorithm where the distance vectors are calculated. Its central component is a scheduler that determines the order and the methods the distance vectors are calculated with. The scheduler and the implementations of the distance vector calculation algorithms, that are introduced in the Section 2.2, are described in Section 3.4 and 3.5.

3.1 Stage 1: *2-core* extraction

The algorithm used by CoreAPSP to extract the *2-core* and to determine the *2-core* degree during the precalculations is called Extract2core. Its pseudo code is listed under Algorithm 1. It is an iterative procedure that extracts the core and determines the *2-core* degree of all core nodes in $O(m)$ time.

The algorithm initializes the *2-core* degree $\deg_{\text{core}}(v)$ of every node $v \in V$ with its degree $\deg(v)$. To obtain the correct *2-core* degree values, it substitutes the number of neighbors that are not part of the *2-core*. Extract2core identifies 1-shell nodes iteratively and decreases the value of their neighbors *2-core* degree by one. A node u is identified as a 1-shell node if $\deg(u)$ is one or $\deg_{\text{core}}(u)$ is decreased to one by one of u's neighbors. After all nodes identified as not being part of the *2-core* are processed, the algorithm finishes.

3.2 Stage 2: Rebuilding the graph and distance precalculations

The identification of 1-shell nodes and the *2-core* degree of each core node is the only data that has to be precomputed for the distance vector calculations to work. Nonetheless, CoreAPSP uses further precalculations to improve the workflow and the performance.

To reduce cache misses, the identification numbers of the nodes are reorganized in order to store data closely together that is accessed in succession. For example nodes that are located within the same tree or on the same *2-core* path satisfy this criterion.

Algorithm 1: Extract2core(Graph)

```
1  degreeOneNodes ← ∅
2  deg_core: Array[0..n] of ℕ
3  forall v ∈ V do
4      deg_core[v] ← degree of v
5      if deg_core[v] ≤ 1 then
6          degreeOneNodes ← degreeOneNodes ∪ {v}
7  while degreeOneNodes ≠ ∅ do
8      v ← arbitrary node from degreeOneNodes
9      deg_core[v] ← 0 // set node deleted
10     u ← unique neighbor of v with deg_core(u) ≥ 1
11     deg_core[u] ← deg_core[u] − 1
12     if deg_core[u] = 1 then
13         degreeOneNodes ← degreeOneNodes ∪ {u}
14     degreeOneNodes ← degreeOneNodes \ {v}
```

To reduce the number of priority queue operations during distance vector calculations, some specific distances are precalculated. Both the distances from the root node of a tree to the nodes inside the tree and the length of the subpaths of *2-core* paths, leading from the separators to all nodes on the path, are determined. Since the trees contain no circles and path nodes only have two neighboring nodes from within the core, these distances can be determined in linear time using depth first searches limited either to 1-shell nodes and the root of the corresponding tree or to core nodes with a *2-core* degree of two. The new node identification numbers are set to the same order as the nodes are settled by the search and the results are saved as sets of tuples containing the nodes and the distance to them. The sets with the distances on the *2-core* paths are associated with newly inserted shortcuts or loops, depending on whether a directed path to the other separator exists or not.

As stated in Section 2.2, the distance from 1-shell nodes or path nodes to their corresponding separators can be used to help speeding up distance vector calcualtions. Note that for path nodes the length of the subpath of the *2-core* path from the path node to the separator is sufficient. All these distances can be calculated using limited backward depth first searches. Recall that nodes on *2-core* paths can have no directed path to any of their separators and that there might be a path from a tree node to the corresponding root but not from the root to the tree node or the other way around.

All of these calculations are triggered during an iteration over the graph's edges. If one of the following cases applies to an edge, a limited depth first search determines some of the required distances.

the source is a core node, the target is a 1-shell node: These edges are identified as edges leading into an attached tree. A depth first search limited to 1-shell nodes is initialized with the target node and the weight of the current edge to determine all distances from the tree root to nodes inside the tree. Note that this search can be triggered multiple times for an attached tree, if the root node is adjacent to more than one of the attached tree's nodes. At the same time, the new identification numbers for all reached nodes are defined in the same order as they are removed from the search stack.

the source is a 1-shell node, the target is a core node: The edge is connecting an attached tree, or at least a part of it, with the core, and all distances from nodes in this part

of the tree to the tree's root can be determined using a depth first search that is limited to 1-shell nodes and follows edges in reverse order.

$\deg_{\text{core}}(source) \geq 3$, $\deg_{\text{core}}(target) = 2$: This edge is leading onto a *2-core* path. A depth first search limited to nodes with a *2-core* degree of 2 is triggered to determine the distance from the separator to the nodes on the path. The identification numbers for the path nodes are concurrently set according to the order in which they are found by the search, if they have not been defined by another search yet.

$\deg_{\text{core}}(source) = 2$, $\deg_{\text{core}}(target) \geq 3$: The edge is connecting a *2-core* path with the core. All distances from nodes on the *2-core* path to the source of the edge, and thus to one of the path's separators, can be determined via a depth first search that is limited to core nodes with a *2-core* degree of 2 and follows edges in reverse order.

3.3 The distance vector calculation algorithms

CoreAPSP uses multiple methods for distance vector calculation. All of them are encapsulated into algorithms and are described in the following sections.

3.3.1 CoreDijkstra

The first distance vector calculation algorithm to introduce is called CoreDijkstra and is a modification of Dijkstra's algorithm. It tries to decrease the number of priority queue operations by using precalculated distances from Section 3.3 and by exploiting the shortest path properties introduced in Lemma 2.2.4 and Lemma 2.2.7. Recall that shortest paths leading into attached trees have to contain the root node as well as paths entering a *2-core* path have to contain one of the separators v_1 or v_l.

The algorithm acts like a usual Dijkstra search, but only inserts core nodes with a *2-core* degree of at least three into its priority queue. To determine the remaining values, the distances from tree roots to the nodes inside the tree and the length of the subpaths of the *2-core* path from the path separators to the nodes on the path, as precalculated in Section 3.3, are used. If the search settles the separator of a *2-core* path, the precalculated subpath distances from separators to all nodes on the path are used to set the distance of the corresponding path nodes to the minimum of the currently stored distance and the distance to the separator plus the distance from the separator to the path node. As this will be done for both separators, the distances are set to the correct values according to Lemma 2.2.7. If the search settles a tree root, all distances to nodes inside the tree are set to the minimum of their currently stored distance and the distance to the root node plus their precalculated distance from the root. It is important to do a minimum check here out of two reasons. First, the search might have been started in the attached tree and the currently stored distance is already correct. Second, the root node might be a path node and its currently determined distance might be wrong. The shortcuts that are inserted during the precalculations assure the correctness of paths containing *2-core* paths.

To speed up distance vector calculations for source nodes in an attached tree or in a *2-core* path region, CoreDijkstra does not start the Dijkstra search at the source node, but begins with a depth first search limited to 1-shell nodes and core nodes with a *2-core* degree of two to determine the local distances and the distances to the nodes connecting the attached tree or the *2-core* path region with the core. Whenever a core node with a *2-core* degree greater or equal to 3 is reached, it is inserted into the priority queue to be used by the main loop after the search. Note that for *2-core* paths both separators might be equal. In this case the key of the separator has to be set to the smaller path length. Algorithm 2 gives the pseudo code of the whole procedure.

Algorithm 2: CoreDijkstra(s,$\vec{d}_s$)

Input: source node s and distance vector $\vec{d}_s$

```
1  PQ ← ∅
2  d⃗_s ← initialize with infinity
3  if s is a core node with deg_core(s) ≥ 3 then
4      PQ.insert((s,0))
5  else
6      dfsStack.push({(s, 0)})
7      while dfsStack ≠ ∅ do
8          (currentNode,currentDistance) ← dfsStack.pop();
9          d⃗_s[currentNode] ← currentDistance
10         forall edges e going out of currentNode do
11             if e's target has already been processed then
12                 continue
13             if e's target is a 1-shell node or a core node with 2-core degree = 2 then
14                 dfsStack.push((target of e, currentDistance + weight of e))
15             if e's target is a core node with 2-core degree ≥ 3 then
16                 if e's target is not in the priority queue then
17                     PQ.insert((target of e, currentDistance + weight of e))
18                 else
19                     PQ.decreaseKey((target of e, currentDistance + weight of e))
20 while PQ ≠ ∅ do
21     (currentNode,currentDistance) ← PQ.deleteMin()
22     d⃗[currentNode] ← currentDistance
23     forall (treeNode,distanceFromRoot) in tree attached to currentNode do
24         d⃗_s[treeNode] ← min(d⃗_s[treeNode],currentDistance + distanceFromRoot)
25     forall edges e going out of currentNode do
26         if target of e is a 1-shell node or a path node then
27             continue
28         targetDistance ← currentDistance + weight of e
29         if targetDistance < d⃗_s[target of e] then
30             if d⃗_s[target of e] = infinity then
31                 PQ.insert((target of e,targetDistance))
32             else
33                 PQ.decreaseKey(target of e,targetDistance)
34         if e is a shortcut then
35             forall (pathNode,distanceFromSeparator) associated with the shortcut do
36                 d⃗_s[pathNode] ← min(d⃗_s[pathNode],currentDistance +
                   distanceFromSeparator)
37                 forall (treeNode,distanceFromRoot) in tree attached to curretNode do
38                     d⃗_s[treeNode] ← min(d⃗_s[treeNode],currentDistance +
                       distanceFromSeparator + distanceFromRoot)
```

3.3.2 TreeNode

The second distance vector calculation algorithm is called TreeNode and only works for source nodes located in trees.

Recall Lemma 2.2.4, which stated that the distance of every path from within an attached tree to a node that is not part of the same tree is the sum of the distance to the root and the distance from the root to the target. All distances inside a tree can be determined in linear time using a depth first search that is limited to 1-shell nodes and the root node. Algorithm 3 lists TreeNode which implements this procedure in $O(m)$ time. As we already precalculated the path length to the corresponding tree root for every 1-shell node, it is possible to initialize the distance vector right at the beginning of the procedure and replace the false distance values afterwards. If the source is located in an unattached tree, the root node argument has to be set to nullNode, which will tell the algorithm to initialize the distance vector with infinity.

Algorithm 3: TreeNode(s,$\vec{d_s}$,r,$\vec{d_r}$)

Input: source node s and distance vector $\vec{d_s}$, root node r [1] and its distance vector $\vec{d_r}$

```
1  deg_core ← precalculated array with 2-core degree value
2  distanceToRoot ← precalculated distance to root node
3  if r is not nullNode then
4      d_s = d_r + distanceToRoot
5  else
6      d_s is initialized with infinity
7  dfsStack ← ∅
8  dfsStack.push( (s,0) )
9  while dfsStack ≠ ∅ do
10     (currentNode,currentDistance) ← dfsStack.pop()
11     d_s[currentNode] ← currentDistance
12     forall outgoing edges e out of currentNode do
13         if e's target has already been discovered then
14             continue
15         if e's target = r then
16             distanceToRoot = currentDistance + weight of e
17             dfsStack.push( (target of e,currentDistance + weight of e) )
18         if e's target is a 1-shell node then
19             dfsStack.push( (target of e,currentDistance + weight of e) )
```

The implementation depends on precalculations but does not do any of them by itself, because their management is intended to be left to the scheduler. A 1-shell node s as source, a distance vector $\vec{d_s}$, which does not have to be initialized in any way, to store the results, the root node r or nullNode if s is located in an unattached tree, and the distance vector of r, $\vec{d_r}$, have to be passed as arguments.

3.3.3 PathNode

The third distance vector calculation algorithm is called PathNode and only works for source nodes located on *2-core* paths.

As stated in Lemma 2.2.7, the distance $dist(s,t)$ from a source node s that is located on a *2-core* path to a target node t outside the *2-core* path region of s equals the minimum

[1] must be set to nullNode for unattached trees

of $dist(s, v_1) + dist(v_1, t)$ and $dist(s, v_l) + dist(v_l, t)$, where v_1 and v_l are the respective separators. Furthermore the distances to nodes inside the *2-core* path region can be determined in linear time using a depth first search limited to nodes with a *2-core* degree of two combined with the information about the distances from tree roots to the nodes inside the tree. Algorithm 4 lists PathNode who takes advantage of this and calculates distance vectors for nodes on *2-core* paths in $O(m)$ time. As the distances to the separators of the path have already been calculated during the preprocessing, the distance vector can be initialized before the depth first search determines the local distances. If there is no directed path from the source to one or both the separators, the distance vector has to be initialized respectively.

Algorithm 4: PathNode(s,$\vec{d}_s$,$sep1$,$\vec{d}_{sep1}$,$sep2$,$\vec{d}_{sep2}$)

Input: source node s and distance vector $\vec{d}_s$, first separator $sep1$ and its distance vector $\vec{d}_{sep1}$, second separator $sep2$ and its distance vector $\vec{d}_{sep2}$

```
1  deg_core ← precalculated array with 2-core degree values
2  nodesInAttachedTree ← precalculated array with information about tree nodes
3  if sep1 ≠ nullNode and sep2 = nullNode then
4      distanceToSep1 ← precalculated distance to sep1
5      d_s = d_sep1 + distanceToSep1
6  if sep1 ≠ nullNode and sep2 ≠ nullNode then
7      distanceToSep1 ← precalculated distance to sep1
8      distanceToSep2 ← precalculated distance to sep2
9      d_s = min(d_sep1 + distanceToSep1, d_sep2 + distanceToSep2)
10 if sep1 = nullNode and sep2 = nullNode then
11     d_s is initialized with infinity
12 bfsStack ← ∅
13 bfsStack.push( (s,0) )
14 while bfsStack ≠ ∅ do
15     (currentNode,currentDistance) ← bfsStack.pop()
16     d_s[currentNode] ← min(d_s[currentNode],currentDistance)
17     forall (treeNode,treeDistance) of the tree attached to currentNode do
18         d_s[treeNode] = min(d_s[treeNode],currentDistance + treeDistance)
19     forall outgoing edges e out of currentNode do
20         if a shorter path to es target has already been found then
21             continue
22         if deg_core[e.target] = 2 then
23             bfsStack.push( (target of e,currentDistance + weight of e) )
```

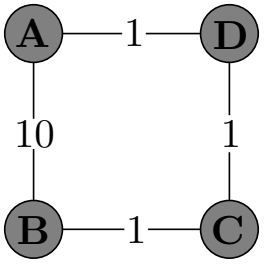

Figure 3.1: A difficult case

During the depth first search that determines the distances to the nodes in the *2-core* path region, there is one special case that has to be taken care of. An example is shown in Figure 3.1. In this case, the source node, which is located on a *2-core* path that forms a ring, has no separators and thus is not connected to any core node with a *2-core* degree greater then two. A normal depth first search with source node **A** could fail determining the right distance values, when it processes the nodes in the following order: First, node **A** is settled and **B** is pushed on the stack after **D**. Then **B** is popped off the stack, and **C** paired with the incorrect distance 11 is pushed on it. To avoid this error, the depth first search of the PathNodealgorithm checks if the distance of nodes popped from the stack is shorter than the distance already saved to the distance vector and pushes already discovered nodes on the stack, if a shorter path to them has been found. Note that errors in the distance vector can only appear if there is no core node with a *2-core* degree of at least three on the path. If such a node exists, all wrong distances will be set to the correct ones when the *2-core* paths, including the one the source node is located on, adjacent to this node are updated. Also note that the search still only needs $O(m)$ time.

Again, none of the required precalculations are done by the algorithm itself. The argument list contains a source node s, a distance vectors $\vec{d}_s$ to store the results, and the separators with their respective distance vectors. Note that the separator argument has to be set to nullNode, if no directed path from the source to the separator exists.

3.4 Stage 3: The scheduler

The scheduler is the central component of CoreAPSP and decides in which order and by the use of which algorithms the distance vectors are calculated. These decisions are fundamental for good performance, as every decision leading to a distance vector calculation with CoreDijkstra where TreeNode or PathNode could have been used as well as every recalculation will extend the total running time.

CoreAPSP has 5 different schedulers to choose from. The first one, called All Plain Dijkstra Scheduler, uses an unoptimized implementation of Dijkstra's algorithm for all distance vector calculations and only has one distance vector stored at a time. It is intended to produce reference values with which the running times can be compared.

The second one, called All CoreDijkstra Scheduler, uses CoreDijkstra for all distance vector calculations, only has one distance vector stored at a time, and is intended to measure the speedup of the optimized Dijkstra variant compared to the plain one.

The CoreDijkstra and TreeNode Scheduler uses the TreeNode algorithm to calculate the distance vectors of all 1-shell nodes and CoreDijkstra for the others. In doing so, it has at most two distance vectors stored concurrently.

The CDTP Simple Scheduler, whose pseudocode is listed under Algorithm 5, implements a simple scheduler that uses CoreDijkstra, TreeNode and PathNode. It iterates once over all core nodes with a *2-core* degree of at least three, determines their distance vector with CoreDijkstra and tries to use this information to speed up the distance vector calculations for adjacent 1-shell or path nodes. This is done by calculating the distance vectors of the target nodes of all outgoing shortcuts. After a distance vector is calculated, all path nodes associated with the shortcut and trees attached to those nodes are processed. After all shortcuts have been processed, the distance vectors of the members of the attached tree rooted at the current core node are determined. While this is working well with the 1-shell nodes as their root is unique, it can slow down the algorithm with a useless CoreDijkstra call if a *2-core* path is only one node long or if all nodes on it have already been processed. Note that the distance vectors of the shortcut target will be discarded and calculated again at another point during the scheduler's execution and that the maximal number of

concurrently stored distance vectors is four. The TreeNode and the PathNode algorithm are also used to determine distance vectors of 1-shell nodes in attached trees and path nodes that are associated with a shortcut. Member nodes of unattached trees and path nodes that are not connected to their separators are also processed with the respective algorithms. In the end, the distance vectors of all unprocessed vertices are calculated with CoreDijkstra. These unprocessed nodes contain path nodes who are connected to at least one of their separators but none of their separators is connected to them and attached trees rooted at such path nodes. The reason why distance vectors of such path nodes are not calculated with the PathNode algorithm is that they are not associated with any shortcut and thus the scheduler is not able to identify them as members of a certain *2-core* path. The use of this scheduler is implementing a very basic approach to handle the problem of finding a good schedule and it thereby delivers kind of a lower bound for other schedulers to compete with.

Algorithm 5: defaultScheduler()

```
1  processed: Array[0..n] of bool initialised with false
2  forall nodes v ∈ V do
3      if not processed[v] then
4          if v is a core node with deg_core(v) ≥ 3 then
5              CoreDijkstra(v,d_v)
6              processed[v] ← true
7              forall tree nodes t in attached tree T_v do
8                  TreeNode(t,d_t,v,d_v)
9                  processed[t] ← true
10             forall shortcuts s going out of v do
11                 if s' target equals s' source then
12                     forall unprocessed vertices p associated with s do
13                         PathNode(p,d_p,v,d_v,v,d_v)
14                         processed[p] ← true
15                         forall unprocessed tree nodes t in attached tree T_p do
16                             TreeNode(t,d_t,p,d_p)
17                             processed[t] ← true
18                 else
19                     w ← target node of s
20                     CoreDijkstra(w,d_w)
21                     forall unprocessed vertices p associated with s do
22                         PathNode(p,d_p,v,d_v,w,d_w)
23                         processed[p] ← true
24                         forall unprocessed tree nodes t in attached tree T_p do
25                             TreeNode(t,d_t,p,d_p)
26                             processed[t] ← true
27         if v is located within an unattached tree then
28             TreeNode(v,d_v,nullNode,d_nullNode)
29             processed[v] ← true
30         if v is located on a 2-core path but has no directed path to its separators then
31             PathNode(v,d_v,nullNode,d_nullNode,nullNode,d_nullNode)
32             processed[v] ← true
33 forall unprocessed nodes v ∈ V do
34     CoreDijkstra(v,d_v)
35     processed[v] ← true
```

The CDTP Simple Checking Scheduler is an improved version of the CDTP Simple Scheduler, that only calculates the distance vector of shortcut targets and processes the associated nodes with the PathNode algorithm if at least two of the nodes associated with the shortcut are unprocessed. If there is only one unprocessed node, the node and all members of the attached tree it is the root of are processed using CoreDijkstra.

The CDTP Waves Scheduler is the most advanced scheduler currently implemented in CoreAPSP. It is the only scheduler that works with a dynamic number of distance vectors greater or equal than four and hence the only one in need of memory management.

The scheduler calculates all distance vectors of 1-shell nodes with the TreeNode algorithm and uses the PathNode algorithm for path nodes. The excellence of this approach strongly depends on the number of recalculations made during the whole process. It is important to prefer the processing of nodes that have few neighboring *2-core* paths, because the number of newly calculated distance vectors for separators on the other side of the paths is low and so is the space needed to store these. On the other hand, using the information from distance vectors as long as they are still available is also of great significance.

To fulfill both these requirements, the CDTP Waves Scheduler begins traversing the subgraph that consists of all separators of *2-core* paths and edges between those who are separators of the same path in a breadth first search style. A node having fewest adjacent shortcuts among all unprocessed nodes is inserted into a double ended queue and processed. All nodes whose distance vector is calculated along the way are appended to the end of the dequeue. When the algorithm is done with the calculations concerning the current node, the nodes in attached trees rooted at it and the nodes on adjacent *2-core* paths, the next iteration starts by taking the node at the front of the queue and by repeating the whole progress for it. If the queue runs empty, a new unprocessed node with the lowest amount of adjacent shortcuts among all is appended to it until all nodes separating paths are processed.

After this crucial part of the algorithm, all nodes on *2-core* paths but with no directed path to a separator as well as all core nodes with a *2-core* degree greater or equal than three and all 1-shell nodes in trees rooted at one of them are processed, before, in the final stage, all nodes located in unattached trees are processed.

Algorithm 6: getVector(s,distanceVectors,maxStored) : $\vec{d_s}$

Input: source node s, set of vectors $distanceVectors$, maximum cardinality for distanceVectors $maxStored$

```
1 if distance vector of s is stored in distanceVectors then
2     return stored distance vector of s
3 else
4     if distanceVectors contains maxStored distance vectors then
5         remove distance vector from distanceVectors that was inserted earliest
6     CoreDijkstra(s,d_s)
7     distanceVectors ← distanceVectors ∪ {d_s}
8     return d_s
```

Algorithm 7: wavesScheduler(concurrentTrees)

Input: maximum number of trees stored at the same time *concurrentTrees*

distanceVectors ← ∅
PQ ← ∅
forall *nodes n who are a separator of at least one path* **do**
    s_n ← number of shortcuts going out of n
    PQ.insert((n,s_n))
while *PQ* ≠ ∅ **do**
    (n,key) ← PQ.deleteMin()
    if *n has already been processed* **then**
        continue
    FIFO ← ∅
    FIFO.pushBack(n)
    while *FIFO* ≠ ∅ **do**
        m ← FIFO.popFront()
        if *m has already been processed* **then**
            continue
        $\vec{d}_m$ ← getVector(m,distanceVectors,concurrentTrees -2)
        forall *unprocessed tree nodes t in attached tree* $\mathbb{T}_m$ **do**
            TreeNode(t,$\vec{d}_t$,m,$\vec{d}_m$)
        forall *unprocessed path nodes p that m is a separator of* **do**
            $\vec{d}_m$ ← getVector(m,distanceVectors,concurrentTrees -2)
            if *p has a second separator* **then**
                o ← second separator of p
                $\vec{d}_o$ ← getVector(o,distanceVectors,concurrentTrees -2)
                if *distance vector of s2 was calculated during getVector* **then**
                    FIFO.pushBack(o)
                PathNode(p,$\vec{d}_p$,m,$\vec{d}_m$,o,$\vec{d}_o$)
            else
                PathNode(p,$\vec{d}_p$,m,$\vec{d}_m$,m,$\vec{d}_m$)
            forall *unprocessed tree nodes t in attached tree* $\mathbb{T}_p$ **do**
                TreeNode(t,$\vec{d}_t$,m,$\vec{d}_m$)
forall *core nodes n with* $\deg_{\text{core}}(n) = 2$ *and no separators* **do**
    PathNode(n,$\vec{d}_n$,nullNode,$\vec{d}_{nullNode}$,nullNode,$\vec{d}_{nullNode}$)
    forall *unprocessed tree nodes t in attached tree* $\mathbb{T}_n$ **do**
        TreeNode(t,$\vec{d}_t$,n,$\vec{d}_n$)
forall *core nodes n with* $\deg_{\text{core}}(n) \geq 3$ **do**
    CoreDijkstra(n,$\vec{d}_n$)
    forall *unprocessed tree nodes t in attached tree* $\mathbb{T}_n$ **do**
        TreeNode(t,$\vec{d}_t$,n,$\vec{d}_n$)
forall *nodes t in unattached trees* **do**
    TreeNode(t,$\vec{d}_t$,nullNode,$\vec{d}_{nullNode}$)

4. Experimental Evaluation

4.1 Hardware

In this section, the results of an experimental evaluation of CoreAPSP are presented. All experiments were conducted on one core of an AMD Opteron 2,218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16GB of RAM, and 2 $\times$ 1 MB of L2 cache. The implementation was compiled using GCC 4.3.2 with optimization level 3. Further details about the implementation can be found in Section 4.2.

4.2 The implementation

CoreAPSP has been implemented in C++. It makes use of classes and contains, without libraries, nearly 6000 lines of code. For the graph representation, the Boost Graph Library [SLL02] has been used, with vectors from the Standard Template Library as containers for both the nodes and the edges. The priority queue used during all experiments is a fast implementation of a binary heap [Sch08]. To avoid the initialization of distance vectors with infinity, timestamps are used. Every distance vector has a search time and every distance has a corresponding timestamp. A distance is only valid if the corresponding timestamp equals the search time. If they differ, the distance is infinity. To initialize a distance vector with infinity, the search time has to be set to a timestamp that was never used before.

4.3 The Testset

The set of graphs used during the tests largely consisted of road networks, as CoreAPSP is primarily intended to be used during the precalculation process of speedup techniques for route planing. Nonetheless, the tests ought not only measure how good the implemented techniques work on their primary problem class, but should also highlight for which other graph types the approach performs good and for which it does not. Hence, some randomly generated and some real world instances have been included into the setup. The instances can be partitioned into three problem classes.

Road Networks: To conduct a precise experimental analysis, many different road networks have been used. All of them are subgraphs of either the European or the North American road network. The European road network was kindly given to us by the PTV [PTV08] and contains data from 2006. The American road network was

taken from the TIGER website of the U.S. Census Bureau [tig10]. The networks can be divided into three classes. The first class contains city graphs. It must be pointed out that these have a quite different structure. North American cities are strictly organized in grids and European cities are not. New York and the Ruhrpott (Figure 4.1), an area in Germany with multiple cities located closely together, have been chosen as test graphs. In the second class, the German North Sea region and a sparsely populated region within Texas (Figure 4.2) have been selected as examples of rural road networks. These graphs are composed of many long roads and fewer junctions. The third class is composed of subgraphs of the European road network, where we select a geographic center uniformly at random and extract a rectangular region around the respective center such that the number of nodes in the extracted subgraph meets our requirements.

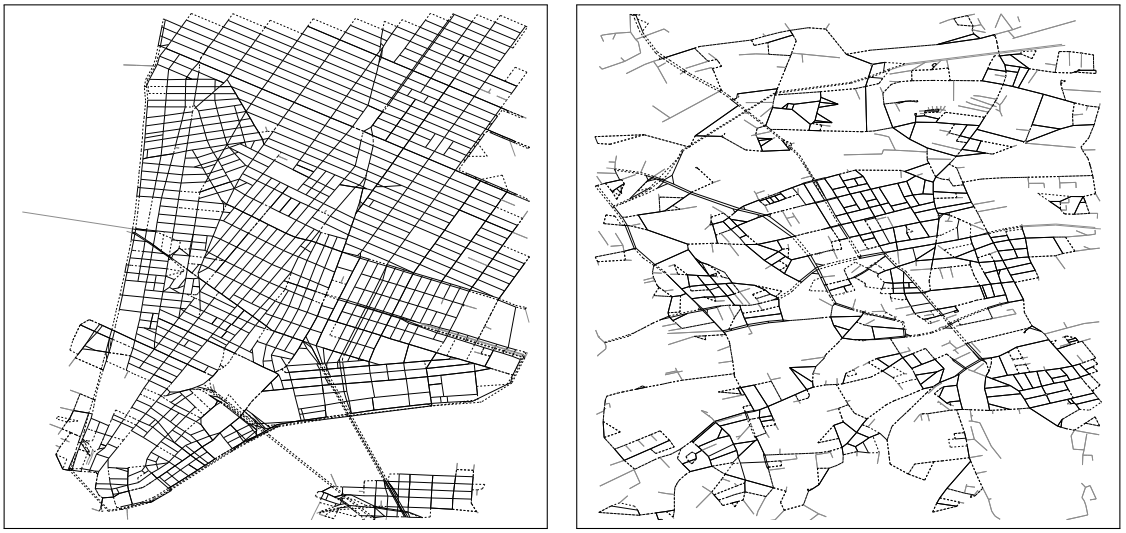

Figure 4.1: New York and the Ruhrpott, 2 000 nodes, *2-core* paths dotted, trees in light gray

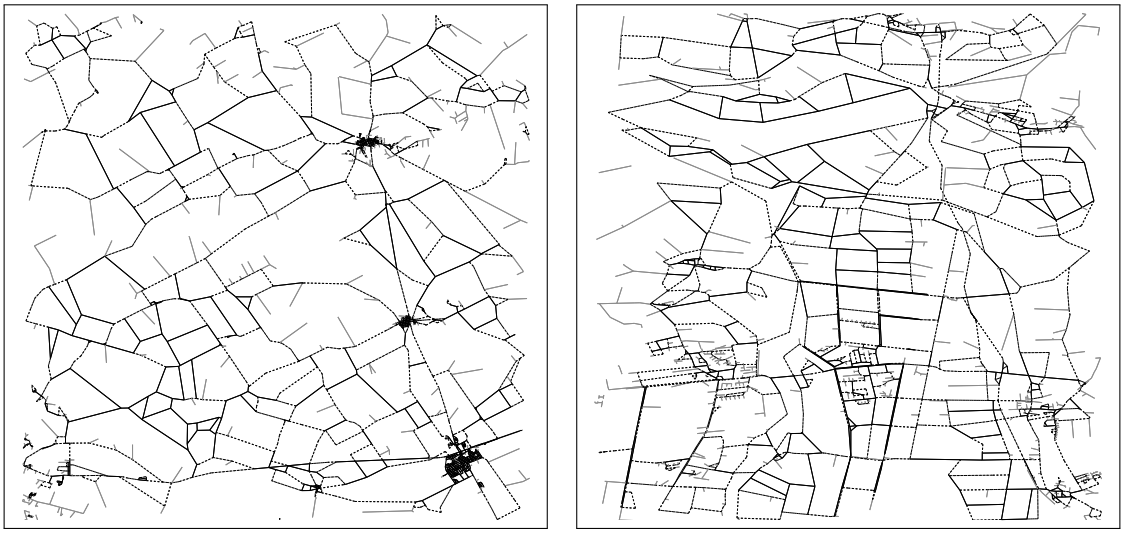

Figure 4.2: Texas, German North Sea, 2 000 nodes, *2-core* paths dotted, trees in light gray

Other real world: Besides the road graphs, we also considered further real-world instances: station graphs of public-transportation networks and a graph replicating

the structure of the Internet. The Rail Europe graph is a static modeling of the European railway system, where every node represents a train station and an edge between nodes u and v equals a connection between the respective train stations without a stopover [PSWZ07]. The graph is very sparse, but still connected, and contains many long *2-core* paths. It is based on the timetable data of the winter period 1996/1997 and was kindly given to us by HaCon [HaC08]. The station NY graph models the bus network of New York. Every node represents a bus station and an edge equals a bus connection between the stations with no stopover or a walking route. All edges are weighted with their respective travel time. The structure of this graph is quite similar to the Rail Europe graph. The data it is based on is the timetable of 2010 and is freely available through a Google transit data feed [Fee10]. The Internet graph, taken from [Uni08], represents the Internet as of 2008/3/26 on the level of autonomous systems. Every of the 28 000 nodes represents such an autonomous system, and every edge equals a connection between them. The weight of all edges is one. Although the degrees in the corresponding simple, unweighted, undirected graph G' vary between 1 and 2 342, they are quite low on average.

Generated: Three different types of randomly generated graphs have been used. Grid graphs, Delauny Triangulations (both in Figure 4.3) and Unit Disc graphs. The nodes of the Grid graph are arranged in a grid pattern and the weights of the edges are chosen uniformly at random between 20 and 300. The Unit Disc Graph generator first chooses a random set of nodes in the plane and then adds an edge between all pairs of nodes whose Euclidean distance falls below a certain bound. The edges are weighted by the distance of their incident nodes. The Delauny Triangulation of a set of points in the plane is a triangulation such that no point is inside the circumcircle of any triangle. It is equal to the dual graph of the Voronoi tessellation.

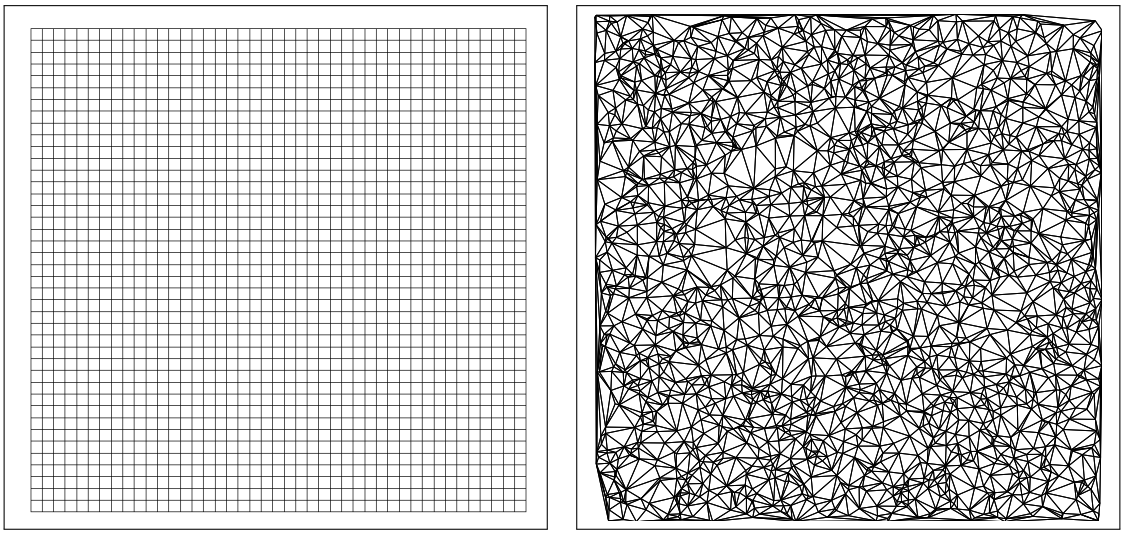

Figure 4.3: Grid and Delauny, 2 000 nodes

4.4 Results

The following sections present the results of the experiments. Four types of experiments have been conducted. At the beginning, all graphs have been analyzed regarding their segmentation into 1-shell nodes, path nodes, and core nodes with a *2-core* degree of at least three (Section 4.4.2). In Section 4.4.3, the performance of the schedulers on the different instances has been compared. In the last two sections, the CDTP Waves Scheduler's

running time is analyzed on different graph sizes (Section 4.4.5) and with different numbers of concurrent distance vectors (Section 4.4.4).

4.4.1 Precalculation time

All running time results include the time needed for the precalculations. They are not listed separately because they do not have a big influence on the total running time, as they only amount 1% of it in most cases. This was only different on the Internet graph, where up to 41% of the running time was needed for the precalculations when using the CDTP Waves Scheduler. The reason is the existence of nodes with very high degree in the graph and a big number of memory operations resulting from it.

4.4.2 Graph structure

CoreAPSP gains its speedup by exploiting *2-core* paths and trees. The higher the fraction of nodes belonging to such structures is, the more possibilities CoreAPSP has to play on its strengths. Table 4.1 reports the figures for the number of 1-shell nodes, path nodes, and core nodes with a *2-core* degree of at least three for all instances. Note that there are no numbers given for the random graphs, since they vary a lot and that the graph with 100 000 nodes is taken as a representative for instances with graphs of different size.

	nodes	edges	1-shell	%	$\deg_{\text{core}} = 2$	%	$\deg_{\text{core}} \geq 3$	%
New York	99915	244876	9283	9	18463	18	72169	72
Ruhrpott	99979	224554	26783	27	35719	36	37477	37
Texas	99954	259260	18531	19	26283	26	173782	55
North Sea	100050	236250	28629	29	35378	35	36043	36
Rail Europe	30517	88091	4013	13	15541	50	10963	36
Station NY	13060	14980	3454	26	7392	57	2214	17
Internet	27909	114474	10368	37	11354	40	6187	22
Delauny	100000	599926	0	0	0	0	100000	100
Grid	10000	39600	0	0	4	0	9996	100
Unit Disc 7	99586	696016	780	1	2283	2	96523	97
Unit Disc 10	99977	993044	60	0	260	0	99657	100

Table 4.1: Node segmentation in test set graphs

Viewing the New York, Ruhrpott, Texas, and North Sea graphs, it emerges that Ruhrpott and North Sea have higher fractions of 1-shell nodes and nodes located on *2-core* paths, which is due to the grid structure of American cities. In a grid, all nodes except the ones located at the boundaries have a *2-core* degree of four. It is to be expected, that CoreAPSP will perform better on the European road networks than on the North American ones.

The station graphs have a very high fraction of nodes on *2-core* paths (especially the Station NY graph) and a low fraction of core nodes with a *2-core* degree of at least three. Very good speedups of CoreAPSP compared to an implementation only using the classic Dijkstra algorithm can be expected for those.

The Internet graph also offers great potential for CoreAPSP because of its low fraction of core nodes with a *2-core* degree of a least three. On the other hand, the initially mentioned existence of nodes with very high degree makes it difficult to find a good schedule.

For the Delauny, Grid, and Unit Disc graphs, no good speedup is to be expected because they contain a very low fraction of 1-shell and path nodes. Thus CoreAPSP is expected to act similar to a plain Dijkstra implementation.

4.4.3 The Schedulers

This experiment was conducted to compare the running time of all schedulers on the different instances. The maximal number of concurrent trees used by the CDTP Waves scheduler was set to 1 100 for the Internet graph, and to 100 for all others. Note that these numbers are big enough to make the scheduler work without any recalculations.

The columns *CoreDijkstra* (CoreDij.) and *recalls* contain the number of calls to the CoreDijkstra algorithm. A distinction between calls with processed and calls with unprocessed source nodes is made. If the source is unprocessed, the call is counted in the CoreDijkstra column. If the source has already been processed before by any of the distance vector calculation algorithms, the call is counted in the recalls column. The *PathNode* (p.Node) and *TreeNode* (t.Node) columns list the number of calls to the respective algorithm. *Runtime* is the total execution time of the algorithm. *Speedup* (Spd.up) is the factor of the running time of the All Plain Dijkstra Scheduler divided by the runtime of the examined scheduler.

	CoreDij.	recalls	p.Node	t.Node	runtime	Spd.up
New York (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	86.42m	-
All CoreDijkstra	99915	0	0	0	71.33m	1.21
CoreDijkstra and TreeNode	90632	0	0	9283	64.74m	1.33
CDTP Simple	72260	20176	18399	9256	66.19m	1.31
CDTP Simple Checking	85599	3128	8621	5695	63.44m	1.36
CDTP Waves	72169	0	18463	9283	51.74m	1.67
Ruhrpott (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	89.20m	-
All CoreDijkstra	99979	0	0	0	51.72m	1.72
CoreDijkstra and TreeNode	73196	0	0	26783	38.30m	2.33
CDTP Simple	37570	33030	35678	26731	37.71m	2.37
CDTP Simple Checking	55169	7447	24923	19887	33.34m	2.68
CDTP Waves	37477	0	35719	26783	20.36m	4.38
Texas (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	88.79m	-
All CoreDijkstra	99954	0	0	0	63.38m	1.40
CoreDijkstra and TreeNode	81423	0	0	18531	51.78m	1.71
CDTP Simple	55161	30938	26269	18524	55.45m	1.60
CDTP Simple Checking	72744	4837	15000	12210	49.68m	1.79
CDTP Waves	55140	0	26283	18531	35.54m	2.50
North Sea (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	89.22m	-
All CoreDijkstra	100050	0	0	0	52.13m	1.71
CoreDijkstra and TreeNode	71421	0	0	28629	37,70m	2.37
CDTP Simple	36070	38302	35363	28617	39,79m	2.24
CDTP Simple Checking	55830	7684	23525	20695	33.84m	2.64
CDTP Waves	36043	0	35378	28629	19.72m	4.52

Table 4.2: Comparison of the schedulers on the different road networks with 100000 nodes

Table 4.2 contains the comparisons of the schedulers for the road networks. The least speedup is always scored by the *All CoreDijkstra Scheduler*. This is due to the fact that the CoreDijkstra distance vector algorithm is used by the other schedulers too, and the All CoreDijkstra Scheduler is, except the All Plain Dijkstra Scheduler, the only one that

does not use any further optimizations. The speedup amounts between 1.21 for the New York and 1.71 for the North Sea graph and can be attributed to the fact that 72% of the nodes in the New York graph but only 36% of the North Sea graphs nodes are part of the core and have a *2-core* degree greater or equal than three.

The *CoreDijkstra and TreeNode Scheduler* is able to improve this by speeding up the calculation of nodes in trees with the help of Lemma 2.2.4. The gained speedup varies between 1.33 on the New York graph and 2.37 on the North Sea graph. The great difference is due to the fact that only 9% of the nodes in the New York graph are part of the 1-shell, in contrast to a faction of 29% in the North Sea graph.

The *CDTP Simple Scheduler* gains results comparable to those of the CoreDijkstra and TreeNode Scheduler. This demonstrates that the calculation of *2-core* path nodes with the use of Lemma 2.2.7 has pros and cons. On the one side, PathNode is able to determine distance vectors much faster than CoreDijkstra, on the other side, the overhead needed to calculate the separators distance vectors can even increase the total calculation time.

Because the *CDTP Simple Checking Scheduler* only uses PathNode if this improves the overall calculation time, its results are superior compared to those of the Default Scheduler and the CoreDijkstra and TreeNode Scheduler. Its speedup varies between 1.36 on the New York graph and 2.68 on the Ruhrpott graph.

The *CDTP Waves Scheduler* outperformed all of the other schedulers by using PathNode for every node on a *2-core* path and TreeNode for every 1-shell node without having to make any recalculations like the CDTP Simple or the CDTP Simple Checking Scheduler. Recall that it only has to store 100 concurrent trees at a time to achieve this. The great variance of the speedup between 1.76 on the New York graph and 4.52 on the North Sea graph demonstrates how much the values depend on the graph structure.

	CoreDij.	recalls	p.Node	t.Node	runtime	Spd.up
Rail Europe (30,000 nodes)						
All Plain Dijkstra	-	-	-	-	6.95m	-
All CoreDijkstra	30517	0	0	0	4.10m	1.70
CoreDijkstra and TreeNode	26504	0	0	4013	3.64m	1.91
CDTP Simple	10963	14318	15541	4013	3.57m	1.94
CDTP Simple Checking	15785	2646	10920	3812	2.61m	2.66
CDTP Waves	10963	0	15541	4013	1.60m	4.36
Station NY (13,000 nodes)						
All Plain Dijkstra	-	-	-	-	27.90s	-
All CoreDijkstra	13060	0	0	0	11.51s	2.42
CoreDijkstra and TreeNode	9606	0	0	3454	9.78s	2.85
CDTP Simple	3226	1780	6807	3027	5.92s	4.71
CDTP Simple Checking	4100	978	6005	2955	5.91s	4.72
CDTP Waves	2214	0	7392	3454	2.94s	9.49

Table 4.3: Comparison of the schedulers on the Station networks

The results of the experiments on the station graphs are listed in Table 4.3. Again, the CDTP Waves Scheduler performed best and the All CoreDijkstra Scheduler worst. Although the speedup of the CDTP Simple and the CoreDijkstra and TreeNode Scheduler are quite similar on the Rail Europe graph, the CDTP Simple Scheduler outperforms the CoreDijkstra and TreeNode Scheduler on the Station NY graph. This primarily arises from the number of recalculations. On the Rail Europe graph, the number of recalculations equals 47% of the total number of nodes in the graph, compared to only 14% of the number of nodes on the Station NY graph. Also, the Station NY graph only has 17% *2-core* nodes

with a *2-core* degree of at least three, and the Rail Europe Graph has 36%. Since the number of concurrent distance vectors was set to 100 for both graphs, this also was an advantage for the smaller Station NY graph. The very good speedups between 2.42 and 9.40 on the Station NY graph can be ascribed to the graph's very low fraction of core nodes with a *2-core* degree of at least three.

	CoreDij.	recalls	p.Node	t.Node	runtime	Spd.up
Internet (28,000 nodes)						
All Plain Dijkstra	-	-	-	-	9.07m	-
All CoreDijkstra	27909	0	0	0	5.39m	1.68
CoreDijkstra and TreeNode	17541	0	0	10368	3.69m	2.46
CDTP Simple	6187	22112	11354	10368	4.79m	1.90
CDTP Simple Checking	17342	259	557	10010	2.93m	3.10
CDTP Waves	6187	0	11354	10368	1.87m	4.84

Table 4.4: Comparison of the schedulers on the Internet graph

The results of the experiments on the Internet graph are listed in Table 4.4. The facts that the CDTP Simple Scheduler is outperformed by the CoreDijkstra and TreeNode Scheduler and that the CDTP Waves Scheduler needs 1 100 distance vectors concurrently stored, show that a good distance vector calculation schedule is crucial on this graph. Nevertheless, 1.68 speedup reached by the All CoreDijkstra and the 4.85 reached by the CDTP Waves Scheduler show that our techniques work quite well. As mentioned in Section 4.4.1, the Internet graph is the only instance where the precalculation times are of influence. They amount 45s in total. If we substitute this from the time needed by the All Plain Dijkstra Scheduler, the speedup of the CDTP Waves Scheduler only amounts 4.45.

	CoreDij.	recalls	p.Node	t.Node	runtime	Spd.up
Delauny (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	1.68h	-
All CoreDijkstra	100000	0	0	0	1.68h	1.00
CoreDijkstra and TreeNode	100000	0	0	0	1.67h	1.01
CDTP Simple	100000	0	0	0	1.67h	1.01
CDTP Simple Checking	100000	0	0	0	1.68h	1.00
CDTP Waves	100000	0	0	0	1.68h	1.00

Table 4.5: Comparison of the schedulers on a Delauny graph

The results of the scheduler comparison on the Delauny graph are listed in Table 4.5 and the results of the experiments on the Grid graph are listed in Table 4.6. As the graphs do not contain any 1-shell or path nodes, CoreAPSP is not able to improve the running time compared to a plain Dijkstra implementation, because it basically performs the same computations. The small differences in running time are due to small measurement errors.

The results of the Unit Disc graph experiment are listed in Table 4.7 and are quite similar to those on the Delauny graph. In the case of the CDTP Simple Scheduler, the pre- and recalculations even lead to an increase of the total running time, since they outweigh the gain obtained by the use of PathNode and TreeNode. The best speedup is again reached by the CDTP Waves scheduler, and amounts 1.05 on the graph with an average degree of 7 and only 1.01 on the graph with an average degree of 10. Note that the Unit Disc 10 graph's fraction of 1-shell and path nodes is lower than the Unit Disc 7 graph's. The fact that the runtimes on the Unit Disc graphs are longer than on the Grid and Delauny graphs, results from the bigger number of edges.

	CoreDij.	recalls	p.Node	t.Node	runtime	Spd.up
Grid (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	1.38h	-
All CoreDijkstra	99856	0	0	0	1.38h	1.00
CoreDijkstra and TreeNode	99856	0	0	0	1.38h	1.00
CDTP Simple	99852	8	4	0	1.38h	1.00
CDTP Simple Checking	99856	0	0	0	1.38h	1.00
CDTP Waves	99852	0	4	0	1.38h	1.00

Table 4.6: Comparison of the schedulers on a Grid graph

	CoreDij.	recalls	p.Node	t.Node	runtime	Spd.up
UnitDisc 7 (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	2.26h	-
All CoreDijkstra	99586	0	0	0	2.25h	1.00
CoreDijkstra and TreeNode	98806	0	0	780	2.24h	1.01
CDTP Simple	96523	3732	2283	780	2.28h	0.99
CDTP Simple Checking	98434	125	542	610	2.23h	1.01
CDTP Waves	96523	0	2283	780	2.16h	1.05
UnitDisc 10 (100,000 nodes)						
All Plain Dijkstra	-	-	-	-	2.79h	-
All CoreDijkstra	99977	0	0	0	2.79h	1.00
CoreDijkstra and TreeNode	99917	0	0	60	2.80h	1.00
CDTP Simple	99657	484	260	60	2.81h	0.99
CDTP Simple Checking	99905	8	26	46	2.79h	1.00
CDTP Waves	99657	0	260	60	2.77h	1.01

Table 4.7: Comparison of the schedulers on two Unit Disc graphs

4.4.4 Runtimes on variable sized graphs

This section evaluates the data gained from runtime experiments conducted on random road networks of different size and on four networks with 300 000 nodes. The intention is to test both the robustness of CoreAPSP's speedup on road networks and as its potential to speed up large instances. As the preceding section showed that the CDTP Waves Scheduler is superior, the other schedulers have been ruled out.

The first experiment examined the robustness of the speedups. It was conducted on random subgraphs of the European road network containing 5 000, 10 000, 50 000 and 100 000 nodes. For each of theses sizes, 100 instances have been used in the test set. Refer to Section 4.3 for further information about their generation. Every graph was processed by the All Plain Dijkstra Scheduler and the CDTP Waves Scheduler. The speedups of the CDTP Waves Scheduler compared to the All Plain Dijkstra Scheduler have been determined for every graph and are displayed in Figure 4.4.

The distribution of the speedup values is large. Although half the values are between 3 and 9, outliers between 1 and 21 appear. This again points out the fact that the speedup depends on the structure of the graph. The larger the size of a graph is the less likely it is that it has a very high or low fraction of core nodes with a *2-core* degree of at least three and the smaller is the quartile region.

The second experiment compares the running times of the All Plain Dijkstra Scheduler and the CDTP Waves Scheduler on the New York, Ruhrpott, Texas and North Sea instances,

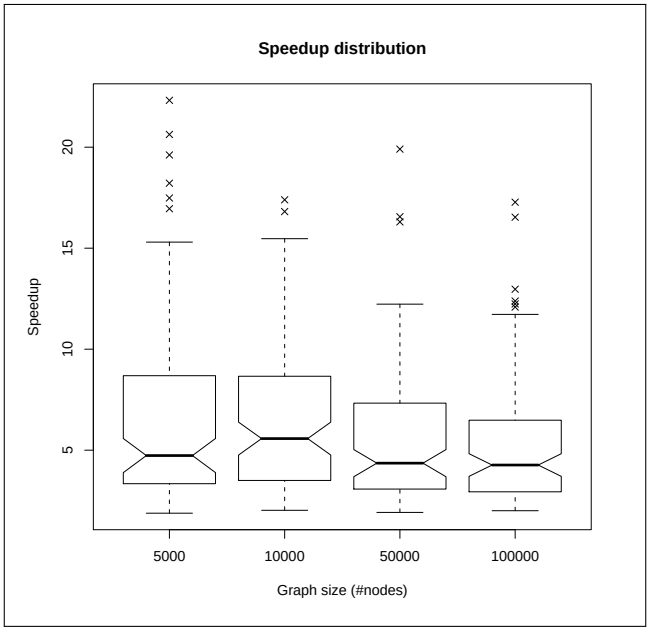

Figure 4.4: The CDTP Waves Scheduler's speedup on random road networks

each with 5 000, 10 000, 50 000, 100 000 and 300 000 nodes. The results can be found in Figure 4.5.

The results show that our techniques can cope with large graphs, as the speedups increase with the number of nodes in the graph. The reason why this does not hold for the Texas graph is, that the 300 000 nodes graph does not have as high fractions of 1-shell and path nodes as the one with 100 000 nodes.

4.4.5 Runtimes with different numbers of concurrent distance vectors

In this section, the running times of the CDTP Waves Scheduler with different numbers of concurrent distance vectors are examined on the Texas, North Sea, Ruhrpott, and Internet graph.

The results on the road networks of Texas and North Sea are presented in Figure 4.6, the results of Ruhrpott in Figure 4.7. The scheduler needed less then 20 concurrent distance vectors to determine all shortest paths with less then 200 recalculations. Regarding the simplicity of the CDTP Waves Scheduler, this shows that finding a good schedule is easy on road networks. The high correlation of the running time and the number of recalls display the fact that if TreeNode is used for every 1-shell node and PathNode for every path node, the number of recalculations is the last crucial factor for short computation times. The influence of cache effects was lower, but still existent. On the Texas graph, the running times are different for 8 up to 20 concurrent distance vectors, despite the fact that the number of recalls is always 0.

The high correlation of the running time and the number of recalls can also be noticed in the results of the Internet graph (Figure 4.7), but the number of concurrent distance vectors the CDTP Waves Scheduler needed to execute the calculations without recalls is higher than on the road network instances. The big difference in the number of recalculations between 1 000 and 1 100 concurrent distance vectors mirror the schedulers lack of ability to handle nodes with many adjacent *2-core* paths. If such a nodes is processed, all distance vectors that are needed to calculate the nodes on the adjacent *2-core* paths have to be stored. A future task could be developing a scheduler that works like a depth first search and thus does not have this weakness.

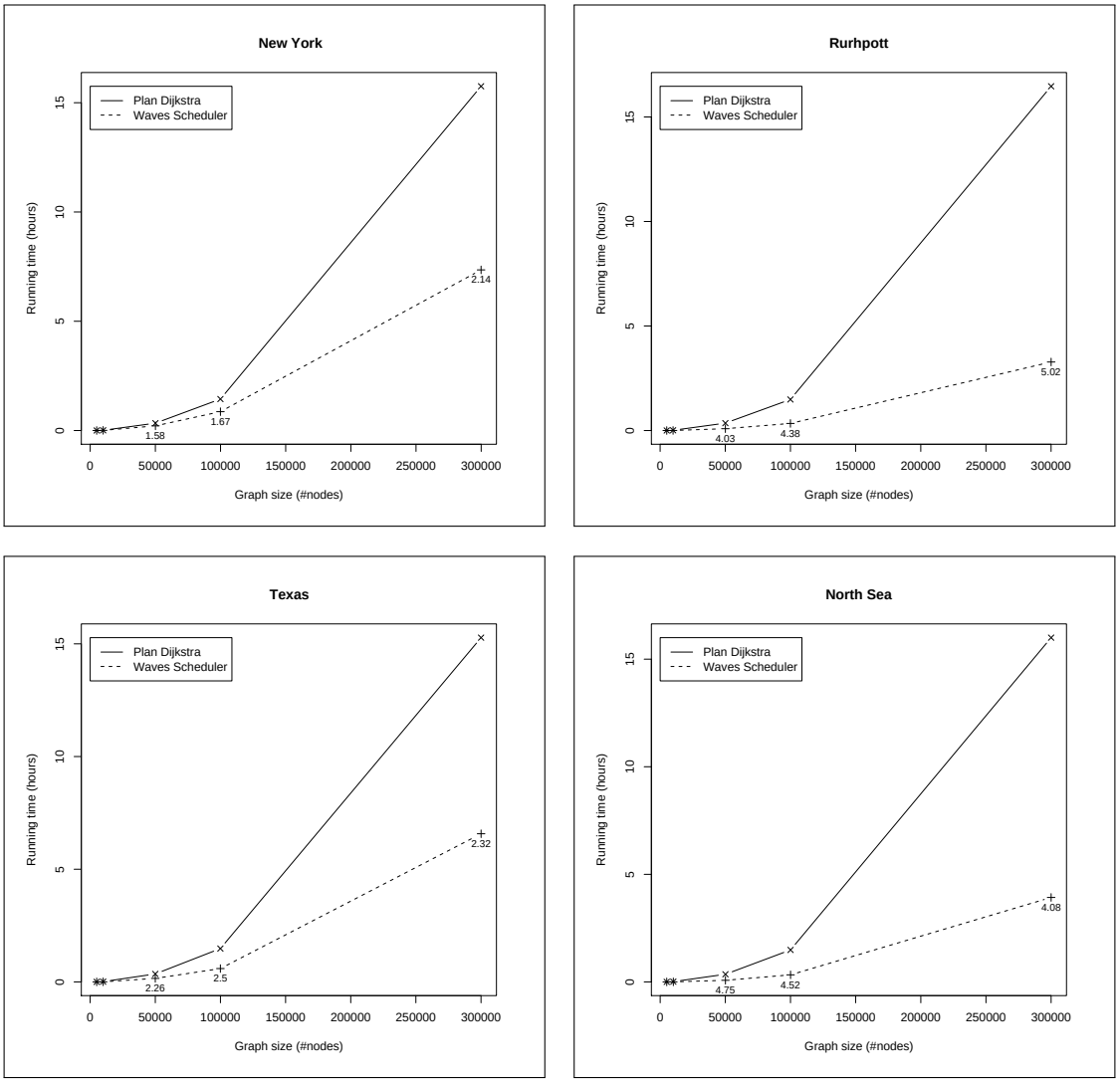

Figure 4.5: Runtimes of All Plain Dijkstra and CDTP Waves on graphs of different size and annotated speedups

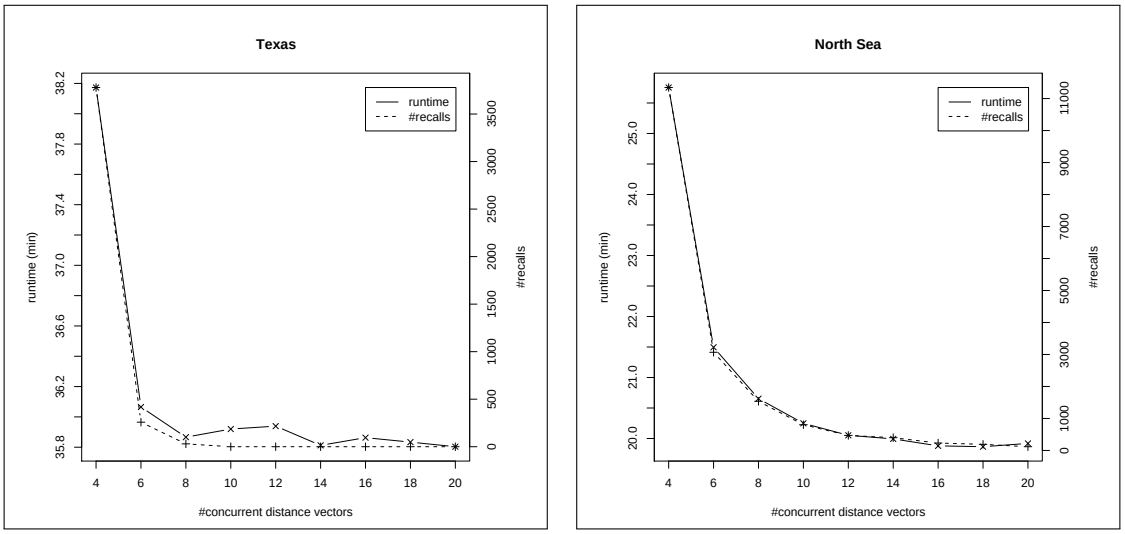

Figure 4.6: Runtimes using the CDTP Waves Scheduler on the Texas (100 000 nodes) and North Sea (100 000 nodes) graphs

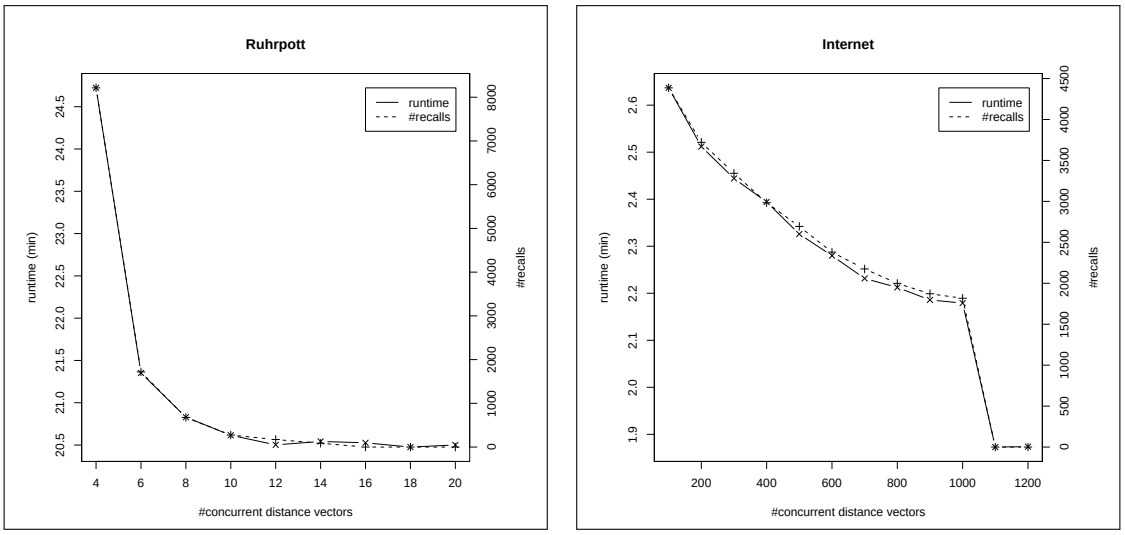

Figure 4.7: Runtimes using the CDTP Waves Scheduler on the Ruhrpott (100 000 nodes) and Internet (30 000 nodes) graphs

5. Conclusion

This study thesis investigated possibilities to exploit structures common to road networks during the shortest path calculations for all pairs. It includes two lemmas about shortest paths leading into or out of an attached tree or a *2-core* path region, stating that their distance is the sum of the distance from the source to a separator and from this separator to the target.

With CoreAPSP, an algorithm solving the The Linear Space All-Pairs Shortest-Paths Problem based on techniques making use of the lemmas was given and evaluated. It could be shown, that speedups up to 5 are possible on road networks, if the fraction of 1-shell and path nodes is high enough. Another result of the experiments is, that the number of concurrently stored distance vectors and the schedule of their calculation is crucial to the algorithms running time. It was also shown, that the number of concurrently stored distance vectors needed for a schedule without recalculations is low on road networks.

Regarding the use of CoreAPSP on other types of graphs, the results have been diverse. Since the speedup techniques are based on graph structures that can only be found in sparse graphs, it does not decrease running times on dense graphs or graphs where every node has a degree of at least three. It was also shown, that the CDTP Waves Scheduler used by CoreAPSP has difficulties finding a good schedule on problem instances like the Internet graph.

Further work could be put into the development of new schedulers or the extension of the pool of distance vector calculation algorithms used by them. In this context, a Dynamic Dijkstra or an algorithm for nodes with a *2-core* degree of three or four are imaginable. Also, further fine tuning could be applied to CoreAPSP and the idea of the *2-core* paths could be extended to regions with nodes having *2-core* degrees greater then two. The size of the region would not matter, the important thing is, that only few edges lead into it and only few out of it. In this case, the idea of shortcuts could also be extended to shortcuts skipping such regions.

A theoretical question could be, how difficult it is to find a perfect schedule and how much concurrent trees will at least be needed for it.

Bibliography

[Bol84] B. Bollobás, "The Evolution of Sparse Graphs," in *Cambridge Combinatorial Conference in honor of Paul Erdos*, ser. Graph Theory and Combinatorics. Academic Press, 1984, pp. 35–57.

[Dia69] R. B. Dial, "Algorithm 360: Shortest-Path Forest with Topological Ordering [H]," *Communications of the ACM*, vol. 12, no. 11, pp. 632–633, 1969.

[Dij59] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[Fee10] G. T. Feed, "http://code.google.com/p/googletransitdatafeed/," 2010.

[Gab85] H. N. Gabow, "Scaling Algorithms for Network Problems," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 148–168, September 1985.

[Gol95] A. V. Goldberg, "Scaling Algorithms for the Shortest Paths Problem," *SIAM Journal on Computing*, vol. 24, no. 3, pp. 494–504, 1995.

[GT89] H. N. Gabow and R. Tarjan, "Faster Scaling Algorithms for Network Problems," *SIAM Journal on Computing*, vol. 18, no. 5, pp. 1013–1036, 1989.

[HaC08] HaCon - Ingenieurgesellschaft mbH, "http://www.hacon.de," 2008.

[Hag00] T. Hagerup, "Improved shortest paths on the word RAM," in *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC'00)*, ser. Lecture Notes in Computer Science, 2000, pp. 61–72.

[Joh77] D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, vol. 24, no. 1, pp. 1–13, January 1977.

[KKP93] D. R. Karger, D. Koller, and S. Phillips, "Finding the Hidden Path: Time Bounds for All-Pairs Shortest Paths," *SIAM Journal on Computing*, vol. 22, no. 6, pp. 1117–1349, 1993.

[McG95] C. C. McGeoch, "All-pairs shortest paths and the essential subgraph," *Algorithmica*, vol. 13, no. 5, pp. 426–441, May 1995.

[Pet04] S. Pettie, "A new approach to all-pairs shortest paths on real-weighted graphs," *Theoretical Computer Science*, vol. 312, no. 1, pp. 47–74, January 2004.

[PSWZ07] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis, "Efficient Models for Timetable Information in Public Transportation Systems," *ACM Journal of Experimental Algorithmics*, vol. 12, p. Article 2.4, 2007.

[PTV08] PTV AG - Planung Transport Verkehr, "http://www.ptv.de," 2008.

[Sch08] D. Schultes, "Route Planning in Road Networks," Ph.D. dissertation, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008, http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf.

[SLL02] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[TF87] R. E. Tarjan and M. L. Fredman, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, July 1987.

[Tho04] M. Thorup, "Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem," *Journal of Computer and System Sciences*, vol. 69, no. 3, pp. 330–353, 2004.

[tig10] "http://www.census.gov/geo/www/tiger/," 2010.

[Uni08] University of Oregon Routeviews Project, "http://www.routeviews.org/," 2008.

[Zwi01] U. Zwick, "Exact and approximate distances in graphs - a survey," in *Proceedings of the 9th Annual European Symposium on Algorithms (ESA'01)*, ser. Lecture Notes in Computer Science, vol. 2161, 2001, pp. 33–48.