Finding Shortest Paths in Large Scale Networks

Leonid Antsfeld

School of Computer Science and Engineering

University of New South Wales

A thesis submitted for the degree of

Doctor of Philosophy

31-01-2014

I dedicate this thesis in memory of my late father Dmitriy Antsfeld who instilled in me passion of "creative thinking" and my grandparents Shlomo Goldisman and Larisa Serebrovkiy, who always believed in me, but unfortunately could not see me graduate.

Acknowledgements

First of all I would like to thank Prof. Toby Walsh for providing me with this great opportunity to work on this interesting topic. You were always very helpful, starting from my initial Ph.D. inquiry all the way till the submission of this thesis. I could not wish for a better mentor.

I also thank my co-supervisors Dr. Phil Kilby and Dr. Aleksandar Ignjatovic. You were always helpful when I needed a professional advice.

I would like to thank NICTA and all the people not mentioned by name (starting from admin to IT support), by providing me an excellent ecosystem for doing a quality research in highly professional environment. UNSW travel grants allowed me to travel and present my research overseas at various conferences and also meet very interesting people. I am very grateful for this.

Special thanks to Helen Bryson, *Elite Athletes and Performers Program* coordinator and UNSW Sport and Recreation Center. With your help and support I could continue to train and race at very high level. I have to admit that many of the ideas presented in this work came while being on a bike.

I spent three months as an intern and later continued to work part-time in Buzzhives. I would like to thank Dr. Tim Cooper and the founder Claus von Hessberg for providing me with this opportunity that became a "kickstarter" of this research.

At the beginning of my research and in the last 1.5 years I was lucky to work in Intelligent Fleet Logistics (IFL), NICTA project. I am very thankful to the team and the management for being so flexible and providing me the opportunity and conditions to combine my studies with practical work. It is very rewarding to see results of my research being used in the real-world product.

Thank you, Dr. Chen Cai, Dr. William Uther, Dr. Adi Botea, Dr. Daniel Harabor, Julian Dibbelt, Thomas Pajor, Christopher Gross, Dr. Nina Narodytska, Maxim Iorsh and Adrian Schoenig for helpful discussions and your willingness to listen and discuss my ideas.

Last, but not least, I would like to thank to my family, while being thousands kilometers away, you were always very supportive and motivating. I wouldn't be in this position without you, Mom. You couldn't attend my M.Sc. ceremony and I hope to keep my promise to you to have another chance. Special thanks to my good friend David Fuchs, who was unconditionally always there for me in times of *joy or sorrow*. I wish you to submit your Ph.D. thesis a.s.a.p.

Finally, I am thankful to my fate that brought three beautiful women into my life - Shawn, Sienna and Tamzyn.

Abstract

Finding the shortest path between two points in a network is a fundamental problem in computer science with many applications. By exploiting properties of the underlying networks we improve and extend one of the state-of-the-art algorithms for finding shortest paths in road networks, Transit Node Routing (TNR). We develop a new algorithm for finding shortest paths in public multi-modal transport networks, where we need to deal with other requirements such as transfers, multi-objectiveness, user preferences, etc. Finally we extend our technique to the new domain of grid networks, where one of the challenges is to deal with path symmetries.

Contents

List of Figures

List of Tables

List of Publications

Leonid Antsfeld and Toby Walsh
"Fast Shortest Path Queries in Road Networks"
EURO, European Conference on Operational Research, Rome 2013

Leonid Antsfeld
"Efficient Routing in Large Scale Networks"
ICAPS, International Conference on Automated Planning and Scheduling (DC)

Leonid Antsfeld and Toby Walsh,
"Incremental updates in Transit Algorithm"
Vehicle Routing and Logistics Optimization Conference, pp. 13, Bologna, June 2012

Leonid Antsfeld and Toby Walsh
"Finding Multi-criteria Optimal Paths in Multi-modal Public Transportation Networks using the Transit Algorithm"
19th World Congress on Intelligent Transport Systems, pp. 25, Vienna September 2012

Leonid Antsfeld, Daniel Harabor, Toby Walsh and Phil Kilby
"TRANSIT Routing on Video Game Maps"
8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), pp. 2, Stanford, October 2012

Leonid Antsfeld and Toby Walsh
"Finding Optimal Paths in Multi-modal Public Transportation Networks using Hub Nodes and TRANSIT Algorithm"
3rd International Workshop on Artificial Intelligence and Logisticsc (AILog), pp.7, Montpellier, August 2012

Chapter 1

Introduction

1.1 Motivation

Efficient routing is a very common problem that arises in many different areas such as in car navigation, public transportation, logistics optimization and even in video games. For road navigation, this is an important component of any GPS navigation system. In logistics, it is an integral part of any solver of Vehicle Routing Problem (VRP). Living in the world of very complex public transportation networks, being able to get from 'A' to 'B' by finding and following an efficient itinerary is a daily task for each of us. In Artificial Intelligence (AI) path finding also emerges in robotics and games as a core problem. In addition, the recent penetration of mobile devices and online services requires very efficient algorithms to solve this problem.

The basic idea of efficient routing in networks is to model a specific problem as a graph and then to find the shortest path in this graph. While there are well known algorithms that can find the shortest path in a graph for straightforward

cases, the real world introduce many challenges. For example, road networks are inherently dynamic, due to traffic jams, road closures, accidents, to name a few. Routing in public transportation networks introduces new challenges which are non-existent in road networks, such as transfers, waiting, tickets, etc. Grid networks have a lot of symmetries, which do not exist in road and public transport networks all together. More accurate and careful engineering is required in order to meet the complexities of each particular problem.

The goal of this research is to extend a state-of-the-art technique to find the shortest paths in road, public transportation and grid networks. At the same time, we want to keep reasonable hardware requirements and relatively simple implementation.

1.2 Our contribution

We introduce new efficient algorithms in three major areas of route planning, specifically, road, public transportation and grid networks. Our ideas were inspired by Transit Node Routing (TNR) [Bast et al., 2006], which was one of the most efficient algorithms known at the time of beginning this research. Even today it continues to be very attractive due to it intuitive approach and relatively simple implementation. We measure the efficiency of the algorithm, both analytically and experimentally, using three criteria: *precomputation time*, *additional space requirements* and *query time*. Our algorithms provide different trade-offs between those criteria and in some of the cases are more efficient in all three criteria than other recently introduced techniques.

1.2.1 Road Networks

Most of the recent and the fastest speed up techniques use (offline) precomputed auxiliary data in order to answer (online) very fast shortest path queries. One of the pitfalls of those approaches is that in reality road networks are dynamic and not static. For example, roads can be closed for repairs, special events, accidents, etc. We have developed a new algorithm, based on Transit Node Routing (TNR) [Bast et al., 2006] which allows under certain assumptions incremental updates of only the affected parts of the network and does not recompute the whole auxiliary data from scratch. We show a new, very efficient way to extract the path itself as well as the distance. In addition, we describe and formally prove a way to minimize the offline storage, allowing us to achieve the fastest possible queries of the Grid-TNR. Finally, we develop a new algorithm that improves TNR in all three criteria and is very competitive with the recent state-of-the-art Hub Labeling (HL) [Abraham et al., 2011] algorithm.

1.2.2 Public Transportation Networks

Due to many inherent complications in public transportation networks, most of the speed up techniques that worked for road networks are not directly applicable here. While they are still theoretically correct, in practice they yield unreasonably long precomputation time and/or prohibitively large additional storage requirements. We show how efficiently to apply the aforementioned Transit Node Routing (TNR) idea to public transportation networks without losing optimality of the final queries. We not only propose a new algorithm, but also create a new time expanded model of the network, which is more efficient than the previously

known time expanded model.

1.2.3 Grid Networks

Many speed-up techniques have been developed for accelerating shortest paths queries in transportation networks. Whilst grid networks somewhat resemble road network structure (relatively low degree, planar), there are other inherent properties of grids that make it less obvious whether speed-up techniques for roads will work well in practice on grids. For example, grids do not have a clear hierarchy between edges and there are many symmetrical paths. In this work, we are the first to apply TNR on the grid networks domain and provide an analysis on a set of popular grid-based video-game benchmarks taken from the AI pathfinding literature. We show that in the presence of path symmetries, which are inherent to most grids but usually not road networks, TNR is strongly and negatively impacted, both in terms of performance and memory requirements. We address this problem by developing a new general symmetry breaking technique. Using our enhancements, TNR achieves *up to four orders of magnitude* speed improvement vs. A* search and uses in many cases only a small or modest amount of memory. We also compare TNR with Compressed Paths Database (CPD), a recent and very fast database-driven pathfinding approach. We find the algorithms have complementary strengths but also identify a class of problems for which TNR is *up to two orders of magnitude* faster than CPDs using a comparable amount of memory.

1.3 Overview

We present an overview of the structure of this dissertation. In Chapter 2 we introduce basic definitions of graph theory in order to have consistent terminology and notations.The thesis is divided into three major parts - road networks (Chapter 3), public transportation networks (Chapter 4) and grid networks (Chapter 5). Each Section contains a comprehensive discussion of the topic, including a review of the related work, proposed algorithms, theoretical analysis and experimental results. Finally, we conclude the thesis in Chapter 6, and discuss possible future research directions.

Chapter 2

Preliminaries

In this chapter, we introduce basic data structures, algorithms, and some notation that are used throughout this thesis. The presented concepts are covered in more detail by most textbooks on algorithms, e.g. [14, 83].

2.1 Basic Concepts in Graph Theory

Definition 2.1. A ***directed graph*** G is pair (V, E), where V is a finite set and E is a subset of $V \times V$. The set V is called the ***vertex set*** of G, and its elements are called ***vertices*** (or ***nodes***). The set E is called the ***edge set*** of G, and its elements are called ***edges*** (or ***links***). A ***weighted graph***, is a graph for which each edge has an associated ***weight*** (or ***cost***), typically given by a ***weight (cost) function*** $w : E \to \mathbb{R}$.

Definition 2.2. If (u, v) is an edge in a directed graph G, we say that (u, v) is ***incident from*** or ***leaves*** vertex u and is ***incident to*** or ***enters*** vertex v. If (u, v) is an edge in undirected graph we say that (u, v) is ***incident on*** vertices

u and v. The ***degree*** of a vertex in undirected graph is the number of edges incident on it. In a directed graph, the ***out-degree*** of a vertex is the number of edges leaving it, and the ***in-degree*** of a vertex is the number of edges entering it. The ***degree*** of a vertex in a directed graph is its in-degree plus out-degree.

Definition 2.3. A ***path*** of ***length*** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $< v_0, v_1, v_2, ..., v_k >$ of vertices, such that $u = v_0$ and $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$. We say that the path ***contains*** the vertices $v_0, v_1, ..., v_k$ and the edges $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. If there is a path p from u to u', we say that u' is ***reachable*** from u via p, which we sometimes write as $u \rightsquigarrow u'$. A ***subpath*** of a path $p =< v_0, v_1, v_2, ..., v_k >$ is a contiguous subsequence of its vertices.

Definition 2.4. Let $G = (V, E)$ be an undirected or directed graph. Two paths p_1 and p_2 from $s \in V$ to $t \in V$ are called ***edge-disjoint*** if they do not share any edges. Please notice that edge-disjoint paths may pass through the same vertex (or vertices). Similarly, we say that paths are ***internally vertex disjoint*** if they do not share any internal vertices. Please notice that vertex disjoint path may start and finish at the same vertex.

Definition 2.5. An undirected graph is ***connected*** if every pair of vertices is connected by a path.

Definition 2.6. A path $p =< v_0, v_1, v_2, ..., v_k >$ called a ***cycle*** if $v_0 = v_k$ and the path contains at least one edge. A graph with no cycles is ***acyclic***.

Definition 2.7. A ***tree*** is a connected, acyclic graph. A ***rooted tree*** is a tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the ***root*** of the tree.

2.2 Shortest Paths

In a ***shortest-path problem***, we are given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The ***weight*** (or ***cost***) of a path $p =< v_0, v_1, v_2, ..., v_k >$ is the sum of the weights of its constituent edges: $\sum_{i=1}^{k} w(v_{i-1}, v_i)$.

The ***shortest-path*** from vertex u to a vertex v (if it exists) is defined as the path with the minimum weight over all the paths between u and v.

Variants

Given a graph $G = (V, E)$

Single source shortest path problem: Find the shortest path from a given ***source*** vertex $s \in V$ to each vertex $v \in V$

Single destination shortest path problem: Find a shortest path to a given ***destination*** vertex t from each vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Single pair shortest path problem: Find the shortest path from u to v for given vertices u and v.

All pairs shortest path problem: Find the shortest path from u to v for every pair of vertices u and v. Although this problem can be solved by running a single source algorithm once for each source vertex, it can usually be solved faster.

Optimal substructure of a shortest path

Shortest path algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. More formally:

Theorem 2.1. ***Subpaths of shortest paths are shortest paths*** *Given a weighted, directed graph* $G = (V, E)$ *with weight function* $w : E \to \mathbb{R}$*, let* $p =< v_0, v_1, v_2, ..., v_k >$ *be a shortest path from vertex* v_1 *to vertex* v_k*. For any* i *and* j *such that* $1 \leq i \leq \text{J} \leq k$*, let* $p_{ij} =< v_i, v_{i+1}, ..., v_j >$ *be the subpath of* p *from vertex* v_i *to vertex* v_j*. Then,* p_{ij} *is a shortest path from* v_i *to* v_j*.*

Definition 2.8. A ***shortest-path tree*** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

(1) V' is the set of vertices reachable from s in G

(2) G' forms a rooted tree with root s, and

(3) for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G

Dijkstra's Algorithm

Dijkstra's algorithm [Cormen et al., 2001] (p. 595) is a greedy algorithm that solves the single-source shortest path problem on a weighted, directed graph $G = (V, E)$ where all the edge weights are nonnegative. The algorithm starts at the source vertex, s, it grows a tree, T, by exploring all the vertices one by one in order of their distance from s. The obtained tree T is called ***Dijkstra's tree***

2.3 Flow in Networks

A ***flow network*** N is given by a directed graph $G = (V, E)$, a function $c : E \to \mathbb{R}$ assigning nonnegative capacities to the edges, and two distinct vertices $s, t \in V$ designated as the source and the sink, respectively. A flow f from s to t, or an $s-t$-flow for short, is a function $f : E \to \mathbb{R}$ satisfying the following constraints:

(1) **Capacity constraints:** $\forall e \in E : 0 \leq f(e) \leq c(e)$.

(2) **Skew Symmetry:** $\forall u, v \in V : f(u, v) = -f(v, u)$.

(3) **Flow conservation:** $\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = 0$.

The quantity $f(u, v)$, which can be positive, zero or negative, is called the ***flow*** from vertex u to vertex v. The value of a ***flow*** f is defined as:

$$|f| = \sum_{v \in V} f(s, v),$$

that is, the total flow out if the source. In the ***maximum flow problem***, we are given a flow network N with source s and sink t, and we wish to find a flow of maximum value.

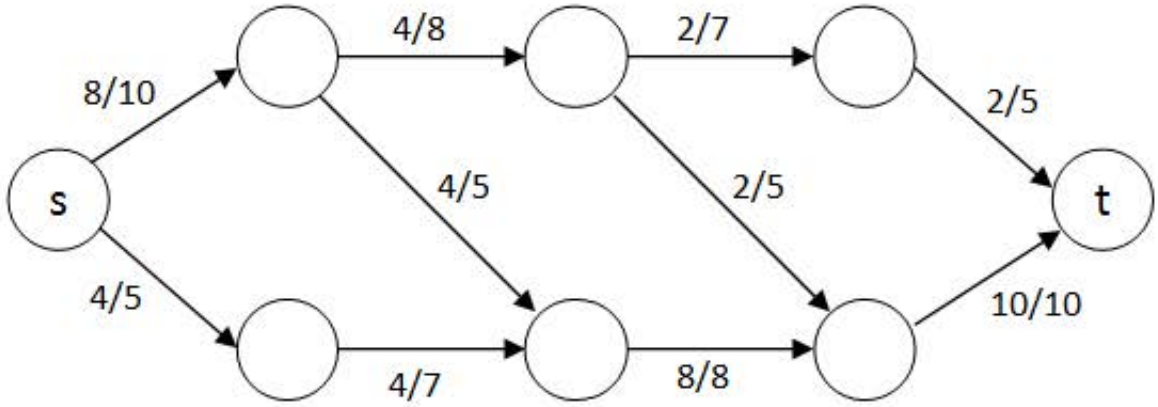

Figure 2.1: Example of a flow network.

Definition 2.9. Intuitively, given a flow network and a flow, the residual network consist of edges that can admit more flow. More formally, suppose that we have a flow netowrk N with source s and sink t. Let f be a flow in N, and consider a pair of vertices $u, v \in V$. The amount of additional flow we can push from u to v before exceeding the capacity c(u,v) is the ***residual capacity*** of (u, v), given by $c_f(u, v) = c(u, v) - f(u, v)$

The ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$

Each edge of the residual network, or ***residual network***, can admit a flow greater than 0.

Theorem 2.2. *In a network N with unit capacities ($c(e) = 1$ for all $e \in E$) the value of the maximum flow equals to the maximum number of edge disjoint paths.*

Proof. Let f^* be a maximum flow and assume by contradiction that there are strictly less than $|f^*|$ edge disjoint paths in N, i.e. $m < |f^*|$. It means that we can push a flow via all those paths and obtain a valid flow with value greater than $|f|$, contradicting maximality of f.
In the other direction, let m be the maximum number of edge disjoint paths. Assume by contradiction that the flow f (that is flowing via those paths) is not the maximum flow and the maximum flow value is $|f^*| > m$. Then, by flow decomposition theorem, we can decompose the maximal flow to f^* paths. Since in N all capacities are one, those paths have to be edge disjoint, and this contradicts maximality of m. □

Corollary 1. *In a network N with unit capacities ($c(e) = 1$ for all $e \in E$) a max flow algorithm give us the largest possible set of edge-disjoint paths.*

Definition 2.10. A ***cut*** (S, T) of a flow network N is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. if f is a flow, then the ***net flow*** across the cut (S,T) is defined to be $f(S, T) = \sum_{x \in S} \sum_{y \ inT} f(x, y)$. The ***capacity*** of the cut (S, T) is $c(S, T) = \sum_{x \in S} \sum_{y \ inT} c(x, y)$. A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

Theorem 2.3. ***Max-flow min-cut theorem*** *In a flow network N the value of the maximal flow equals to the value of the minimum cut [Cormen et al., 2001] (p. 657).*

2.4 Bipartite Graphs and Matching

Definition 2.11. A ***bipartite graph*** is an graph $G = (V, E)$, where V can be partitioned into two sets V_1 and V_2 such that any edge $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$

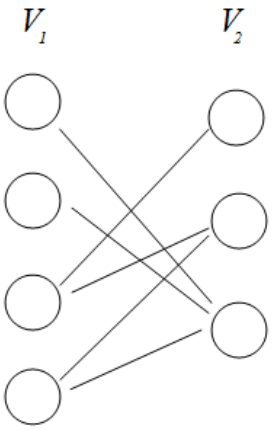

Figure 2.2: Example of a bipartite graph

Definition 2.12. Given a graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$, such that for all vertices $v \in V$, at most one edge of M is incident on v. A ***maximum matching*** is a matching of maximum cardinality.

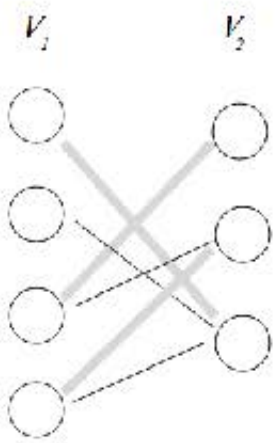

Figure 2.3: Example of a matching in a bipartite graph

2.5 Vertex Cover

Definition 2.13. A ***vertex cover*** of a graph $(G = V, E)$ is a subset $V' \subseteq V$ such that for every edge (u, v) in G either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it. The ***vertex cover problem*** is to find a vertex cover of minimum size.

The vertex cover problem is NP-complete [Cormen et al., 2001] (p. 1006). Nevertheless, for bipartite graphs, we can efficiently find a minimum vertex cover in polynomial time, by finding a maximum flow in a corresponding flow network [Cormen et al., 2001] (pp. 664-668).

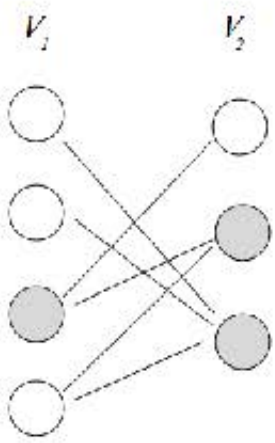

Figure 2.4: Example of a vertex cover in a bipartite graph

Theorem 2.4. ***König's Theorem*** *[Konig, 1931]*

In any bipartite graph, the number of edges in a maximum matching equals to the number of vertices in a minimum vertex cover.

2.6 Graph Connectivity

Definition 2.14. Let s and t be distinct vertices in a graph G. An $s-t$ separating vertex set in G is a set of vertices (others than s and t) whose removal destroys all $s \rightsquigarrow t$ paths in G.

Similarly, let S and T be subsets of vertices in a graph G. A $S-T$ separating vertex set in G, is a set of vertices whose removal destroys all $s \rightsquigarrow t$ paths between any node in S to any node in T.

Theorem 2.5. ***Menger's Theorem (Vertex version)****[Menger, 1927]*

Let s and t be two non adjacent vertices in a graph G, then the minimum number

of vertices in an $s-t$ separating vertex set is equal to the maximum number of internally vertex disjoint $s-t$ paths in G.

2.7 Hitting Set

Definition 2.15. Let U be a finite set and S a collection of its subsets $S = \{S_1, S_2, ..., S_n\}$. A ***hitting set*** of S is a subset $H \subseteq S$ such that H contains at least one element from each subset in S. A ***minimum hitting set*** is a hitting set whose cardinality is minimum over all possible hitting sets.

Definition 2.16. The ***frequency*** of an element of U is the number of sets in S that contain the element. The frequency of the most frequent element of U is denoted by f. The special case of $f = 2$ is also known as the vertex cover problem [Cormen et al., 2001] (p. 1006).

Since minimum vertex cover is NP-complete [Cormen et al., 2001] (p. 1006), the minimum hitting set problem is NP-complete as well.

Definition 2.17. We say that an algorithm for a problem has an ***approximation ratio*** of α if, for any input, the cost C of the solution produced by the algorithm is within a factor of α of the cost C^* of an optimal solution: $max(\frac{C}{C^*}, \frac{C^*}{C}) \leq \alpha$. We also call an algorithm that achieves an approximation ratio of α a ***α-approximation algorithm***.

The greedy algorithm below is H_k approximation algorithm for the minimum hitting set problem [Fernandez de la Vega et al., 1992], where $k = max\{|S_i| : S_i \in S\}$ and

$$H_k = \sum_{i=1}^{k} \frac{1}{i}$$

.

Greedy Hitting Set Algorithm

The greedy hitting set algorithm works in iterations. At each iteration, the algorithm picks the set in S that hits the largest number of elements in U and removes the covered elements from U. It stops when U is emptied. The final picked sub-collection is returned then as the solution. Below we present the pseudocode of the algorithm.

$H \leftarrow \phi$

while $U \neq \phi$

    do select $S_i \in S$ that maximizes $|S \cap U|$

        $U \leftarrow U - S$

        $H \leftarrow H \cup S_i$

return H

Chapter 3

Road Networks

Part of the work in this chapter was presented in [Antsfeld, 2013; Antsfeld and Walsh, 2012c; Antsfeld et al., 2013].

3.1 Related Work

Following the pioneering work of Dijkstra (1959) in finding shortest paths in a graph, there has been a plethora of work on finding shortest paths in graphs in general and in road networks in particular [Delling et al., 2009b; Wagner and Willhalm, 2007]. Very fast speed-up techniques have been developed that can answer shortest path distance queries in a matter of milliseconds [R. Geisberger, 2012], microseconds[Bast et al., 2006; Sanders and Schultes, 2006a] and recently even in nanoseconds [Abraham et al., 2011] on a typical modern server [Intel]. We will mention the classical, the most recent and the most relevant approaches.

Dijkstra algorithm

The Dijkstra algorithm [Dijkstra, 1959] will find the shortest path in a graph with non-negative weights in $O(m + n \log m)$, where n is the number of nodes and m is the number of edges in the graph [Cormen et al., 2001]. For road networks, where the degree of a node is bounded by a small constant, the time complexity of finding the shortest path between two nodes becomes $O(n \log n)$. Starting from a *source* node, the algorithms explores nodes one by one, in order of their distances to the *source*, until it reaches the *target* node. For a large network, it may take more than a second to find a solution, which is a prohibitively slow for many real world applications.

An immediate improvement, called *bidirectional Dijkstra* is to perform the search from both directions, i.e. from a *source* and *destination* simultaneously, until the two search spaces meet. In a road network, where search spaces have roughly circular shape, the *bidirectional* variant of Dijkstra grows two (roughly) circles around *source* and *destination*. The radius of those circles are approximately two times smaller than the circle grown in the original Dijkstra, therefore we can expect a speedup of about factor of two.

A* (A Star)

A* algorithm [Hart et al., 1972] can be seen as another improvement of the original Dijkstra. It directs the search toward the *target*. Similar to Dijkstra it explores nodes one by one until it reaches the *target* node, but the order in which nodes are explored is sum of a distance of the current node to the *source* plus the estimated (a.k.a. heuristic) distance of the node to the *target*. If the heuristic

is admissible, i.e. not overestimating the distance between a node and a target, the A* algorithm is guaranteed to find the shortest path. For road networks, a commonly used admissible heuristics is simply the flying distance between two nodes. We can easily see, that if we choose the heuristic to be a constant of zero, A* becomes a simple Dijkstra search. The perfect heuristic (i.e. the exact graph distance between two nodes) will direct A* straight to the *target* exploring only nodes along the shortest path. Similar to Dijkstra, with a little bit more caution, A* can also be speeded-up using a bidirectional approach [Geisberger, 2011].

Geometric Containers

Geometric Containers [Wagner and Willhalm, 2003] are another attempt to prune the search space and accelerate Dijkstra. The authors observe and exploit a property of the shortest paths: an edge that is not a first edge on the shortest path toward a target, won't appear on any shortest path toward this target. For every edge e a set of nodes $S(e)$ is precomputed, such that $S(e)$ contains all the nodes that can be reached by a shortest path that starts with e. Then during a Dijkstra search, edges e for which target is not in $S(e)$ can be ignored. Since storing all such sets will require a prohibitively large memory footprint, the authors use geometric objects (called *containers*), for each edge that contain nodes of $S(e)$ (and possible more). They considered many different *containers*, such as disk, ellipse, box, sector, and others. Surprisingly, the simplest container, bounding box, outperformed all other geometric objects. Later, this algorithm was improved by combining it with A* and a multilevel approach [Holzer et al., 2004].

The ALT Algorithm (Landmarks)

The ALT algorithm [Goldberg and Harrelson, 2005] speeds-up A* search by providing tighter lower bound for the A* heuristic. The basic idea of ALT is to choose a subset of nodes $L \subset V$, called *landmarks* and precompute distances between any node of the given graph and L. Then during the search we can use this information to have a tighter lower bound of the shortest distance to the target. For example, let $v \in V$ and $l \in L$. In road networks, where Euclidian distances are used as link weights, A* will use a simple geometrical distance between v and target t as an heuristic that estimates shortest distance between v and t. Using landmarks and the triangle inequality allow us to have a better estimate. We notice that $d(v,t) \geq d(v,l) - d(t,l)$ and $d(v,t) \geq d(l,t) - d(l,v)$, therefore $d(v,t) \geq max_{l \in L}\{d(v,l) - d(t,l), d(l,t) - d(l,v)\} = h(v)$. Bidirectional A* with the heuristic function above is called ALT.

The speed up factor highly depends on how well landmarks are distributed over the graph. Finding the smallest set L that will yield the fastest search is still an open problem. The most common landmark selection methods are *avoid* and *maxCover* [Goldberg and Werneck, 2005].

Arc Flags

Arc Flags algorithm [Möhring et al., 2007] precomputes and stores for each edge a flag which will be later used to guide Dijkstra search. Initially the graph is partitioned into k regions. Each edge of a graph has a k-bits binary flag. Bit $i \in \{1, 2, .., k\}$ is set to 1 indicates that this edge potentially can be used along some shortest paths toward some node in region i. Complementary, if the i bit is

set to 0 it means that there is no shortest path traversing via this edge toward any node in the region i. Then, during the query, knowing the region of the target, while executing Dijkstra we can safely disregard all edges with bit corresponding to this region set to 0.

While the search procedure is quite simple, the preprocessing is not that straightforward. The naive approach would be to execute all pairs shortest paths computation of the whole graph, which of course would be prohibitively long. A better approach would be to execute Dijkstra only from the border nodes of each region. More sophisticated techniques exist. Higler [Hilger, 2007] preprocessed the Western Europe road network with about 18×10^6 nodes using the Arc Flags algorithm in about 17 hours and archived query times thousand times faster than the original Dijkstra. In addition, to a relatively slow preprocessing time, the other disadvantages of this approach is relatively slow queries when two nodes are in the same region. In this case, a query becomes a regular Dijkstra and can be improved by performing a bidirectional search.

REACH

REACH algorithm [Goldberg et al., 2006] is based on a definition of the *reach* of a node [Gutman, 2004]. Intuitively a node has a large *reach* if this node is in a middle of a long shortest path. During the query, the idea is to reduce the bidirectional Dijkstra search space by not visiting small *reach* vertices. Later Goldberg improved their algorithm by combining it with ALT [Goldberg et al., 2007]. Another improvement was achieved by clever integration with *shortcuts* [Sanders and Schultes, 2005, 2006b].

Precomputed Cluster Distances (PCD)

PCD (similar to A*) is a goal directed technique. PCD [Maue et al., 2006] partitions the network to clusters and then precomputes and stores the shortest distances between all the clusters. Then, during the query, the algorithm executes a Dijkstra search and constantly maintains upper and lower bounds of the shortest path, by applying this preprocessed information. This allows to significantly reduce the search space and speed-up the Dijkstra. It is a very flexible algorithm, because its preprocessing time does not depend on the number of the border nodes of the clusters. In addition, its asymptotic preprocessing time and space is better than ALT and Arc Flags mentioned above [Maue et al., 2010].

Highway Hierarchies

The Highway Hierarchies (HH) [Sanders and Schultes, 2005, 2006b; Schultes, 2008] algorithm is the first algorithm that reported average query time measured in *milli*seconds for continental sized road networks. The algorithm is based on the basic idea that when traveling a long distance away, we usually at some stage, hop on the highway network and at some stage hop off the highway network to reach our destination. The algorithm consist of two stages - offline preprocessing and real-time query. During the preprocessing highway hierarchies are constructed by alternating between two basic routines *nodes reduction* and *edge reduction*. Node reduction routine removes nodes with low degree, by bypassing them with shortcut edges. Edge reduction routine removes *non-highway* edges, i.e., edges that only appear on shortest paths close to source or target. A query is a restricted bidirectional Dijkstra, where non-highway edges need not be expanded when the

search is sufficiently far from the source or the target.

SHARC [Bauer and Delling, 2010] is a clever combination of HH with Arc Flags. SHARC (**SH**ortcuts + **ARC** Flags) was designed to deal with queries where bidirectional search is impossible, such as time dependent scenarios. Generally, SHARC can be used with static networks, yielding even greater speedup.

Contraction Hierarchies

Contraction Hierarchies (CH) [Geisberger et al., 2008; R. Geisberger, 2012] is another (very successful) speedup technique of the Dijkstra algorithm. During preprocessing, nodes are assigned a strict ordering (a.k.a. importance) and then are sequentially contracted according to this order (starting from the less important). The central idea of node contraction is to remove a node from a graph, and add shortcut edges to preserve shortest-path distances between the remaining nodes. A contracted node v is removed and replaced by a shortcut edge between its neighbors u and w, if and only if the shortest path between u and w contains v. Good node ordering will yield less shortcuts, which consequently will make preprocessing and final query more efficient. Generally, finding the optimal order that will minimize number of shortcuts is a NP-Hard problem [Bauer et al., 2010a], but good heuristics exist [Geisberger, 2008]. A query is performed by restricted bidirectional Dijkstra, where only more important nodes are explored.

CHASE [Bauer et al., 2010b] (**CH** + **ARC** Flags) is another clever combination of CH with Arc Flags. Initially, a complete contraction hierarchy is created. Then, arc flags are computed, but only on a core of the most important nodes (including shortcuts). The query is similar to CH until the search reaches

the core, then arc flags are used to guide the search. This results in a very fast algorithm with query times below 20μs for the road network of Wester Europe.

Customizable Route Planning

Most of the previous algorithms require some preprocessing effort which is computed with a given cost function (metric), fastest time for example. If we are interested to find a shortest path using a different metric (e.g. shortest distance instead of fastest time) we need to perform the whole precomputation from scratch again using a new metric. The Customizable Route Planning (CRP) [Delling et al., 2011] algorithm was designed to address this issue. The main idea is to perform metric-independent partitioning of the network first. This defines a topology for the *overlay graph* [Holzer et al., 2009]. PUNCH [Delling et al., 2010] was proposed as a very successful partitioning heuristic tailored specifically to road networks. The second stage *customization* is much faster. It precomputes the actual costs of the *overlay arcs* and will be run each time the metric change. Finally, the query uses information of the previous two stages to find a shortest paths in real time (milliseconds). While this algorithm is relatively slow, it can be useful when metric is not known in advance or can change frequently. Recently another improvement was proposed [Delling and Werneck, 2013], which speeds up the *customization* stage to a fraction of a second.

Transit Node Routing

In 2006, Bast, Funke and Matijevic were first to introduce the notion of Transit Node Routing (TNR) [Bast et al., 2006]. Their primary idea was based on the

simple observation inspired from real-life navigation: when traveling between two locations that are "far away" one must inevitably use some small set of nodes that are common to many shortest paths. a.k.a. *transit* nodes. In order to identify those *transit* nodes, the authors subdivided the network into cells by overlaying it with a simple grid. Border nodes of the cells which are on "far away" shortest paths defined to be as *access* nodes of that cell. The union of all *access* nodes constitutes the set of aforesaid *transit* nodes. Next, distance tables between all transit nodes and every node and its associated transit nodes are precomputed and stored.

A query between any two "far away" *src* and *dst* locations is performed as follows: we fetch the transit nodes associated with cells containing *src* and *dst* and choose those two that will give us a minimal cost of the combined three subpaths: $src \rightsquigarrow T_{src}$, $T_{src} \rightsquigarrow T_{dst}$, $T_{dst} \rightsquigarrow dst$. For all other queries we apply any efficient search algorithm; A* for example. On average the number of access nodes was a relatively small constant number, which allowed very fast distance queries.

Since our work was highly inspired by TNR, we will discuss and analyze this algorithm in more details in Section 3.2.

TNR+HH [Bast et al., 2007] relies on very similar observation as TNR, but based on the HH to preselect the set of transit nodes rather than identify them using a grid[1]. In [Bast et al., 2007] the two approaches are compared. One of the advantages of TNR+HH is that it answers all types of queries, unlike Grid-TNR which efficiently answers only "far away" queries. In addition Grid-TNR

[1] This was also developed in parallel by by P. Sanders and D. Shultes [Sanders and Schultes, 2006a].

has more modest memory requirements, TNR+HH has faster preprocessing and average query time (i.e. all types of queries are considered). Since then TNR+HH was superseded by Hub Labeling [Abraham et al., 2011] and CH-TNR [Arz et al., 2013], we will therefore compare our results with the later two.

TNR+Arc Flags [Bauer et al., 2010b] is a speed-up of TNR "far away" queries. The authors noticed that table lookups consume most of the TNR query time. Naturally, the idea was to reduce the number of those lookups by discarding unnecessarily ones and considering only lookups that may lead us to the destination. With reasonable overhead of preprocessing time and additional storage, the query time was reduced by factor of 1.8.

CH-TNR [Arz et al., 2013] is the most recent result that combines TNR with CH. Similar to TNR+HH this method preselects transit nodes by using the top nodes from CH. The authors report very competitive results with other methods, where as expected, there is a tradeoff between additional storage space, preprocessing and final query time. Another noticeable contribution of this work is replacing a geometrical locality filter with a purely graph-theoretical one. Locality filter is a function, where given a source and destination, it indicates whether a query requires a local path search.

Hub Labeling Algorithm

One of the most prominent methods in static road network routing, called *Hub Labeling* (HL) was introduced by [Abraham et al., 2011]. Essentially it can be seen as a speedup of CH query. HL precomputes and stores search spaces of CH and simply intersects them during the query time, rather than performing a

bidirectional search as CH does. Using fine tuning and sophisticated engineering to optimize memory accesses and to prevent cache misses, the authors achieve the fastest known distance query times, up until today. However, its practical applicability may be limited due to a very large memory requirement and quite complicated implementation. In [Abraham et al., 2012], the authors introduce Hierarchical HL, which slightly improves the original HL in both memory requirements and query time. Nevertheless, the offline generated data is often prohibitively large for practical usage. Hub Label Compression [Delling et al., 2013], one of the most recent techniques, manages to reduce the memory footprint significantly, but at a price of almost an order of magnitude slower queries.

Highway Dimension

While all of the mentioned algorithms performs very well for road networks, it is relatively easy to build instances of networks where these algorithms will fail to perform as effectively. This raised a natural question:"*what are the theoretical properties of road networks that made those approached to be successful ?*" To answer this question I. Abraham et al. defined a notion of *Highway Dimension* (HD) [Abraham et al., 2010]. Informally, HD is defined as the size of the smallest set of nodes, such that any "long" shortest path is covered by a node from this set. The authors formally showed that low HD of the network will guarantee good query performance. Then they argue that road networks indeed have a low HD. This explains why many the aforementioned shortest-path algorithms (HH, CH, TNR, SHARC) are performing so well in practice.

3.2 Transit Node Routing

Our research was inspired by the original idea of H. Bast et al., the Transit Node Routing (TNR) algorithm [Bast et al., 2006]. At the beginning of this research it was one of the fastest shortest path distance queries algorithms. Since then TNR was generalized as a more generic framework [Schultes, 2008] and several TNR variants were proposed [Bauer et al., 2010b; Eisner and Funke, 2012; Sanders and Schultes, 2006a]. Until today TNR is considered to be one of the best algorithms and still attracts new research [Arz et al., 2013]. In the recent literature the original grid based TNR is referred to as Grid-TNR, therefore for the sake of consistency we also adopt this notation.

3.2.1 Contributions

In what follows we will describe the original Grid-TNR and will extend it to provide the fastest times as well as the shortest distance queries. The main difference is that in a road network, the distance between two adjacent nodes A and B is the same in both directions, but traveling time may not be. That means that Grid-TNR will now have to deal with directed links rather than undirected as it was originally designed. This generalization for directed graphs was left by the authors as an open and non trivial problem. We will show how Grid-TNR can be adjusted to deal with directed graphs. Next, we will describe a more efficient way to extract the shortest path itself (rather than its value). We will show even more improvements in Section 5.5.

We point out a mistake in the original Grid-TNR paper [Bast et al., 2006] which requires a re-evaluation of the original results. We also give a proof of cor-

rectness of Grid-TNR and perform theoretical time and approximate asymptotic space complexity analysis.

We tackle a more difficult problem that until now was left untreated, updating the precomputed database caused by small changes in an underlying road network, due to traffic jams for example. Of course, a naive and highly inefficient way would be to recompute everything from scratch. We are the first to show an efficient way to incrementally update Grid-TNR precomputed tables without the need of full recomputation.

Finally, we design and present a new TNR based algorithm, called CHAT (Cluster, Hierachify and Hit). It addresses weaknesses of Grid-TNR and improves it in both additional storage space and final query time. Moreover, as we will show CHAT is very competitive with the latest state-of-the-art algorithms HL [Abraham et al., 2011] and TNR-CH [Arz et al., 2013]

The original Grid-TNR algorithm is based on a very simple intuition inspired from real-life navigation: when traveling between two locations that are "far away" one must inevitably use some small set of nodes that are common to many shortest paths. This set was named as "transit nodes" for which the algorithm is called . The algorithm proceeds in two phases: (i) an offline precomputation phase and (ii) an online query phase.

3.2.2 Precomputation

There are two steps to Grid-TNR's precomputation phase. The first step identifies transit nodes and the second step builds a database of exact costs between the

transit nodes and between transit nodes and other nodes in the graph. We will describe each step in turn.

Identifying Transit Nodes

Grid-TNR begins by dividing an input map into a grid of equal-sized cells. To achieve this the algorithm computes a bounding box for the entire map and divides this box into $g \times g$ equal-size cells. Let C denote such a cell. Further, let I (Inner) and O (Outer) be squares having C in the center, as depicted in Fig. 3.6. The size of the cell C and squares I and O can be arbitrary without compromising correctness. Their exact values however will directly impact factors such as an algorithm preprocessing time, storage requirements and online query times.

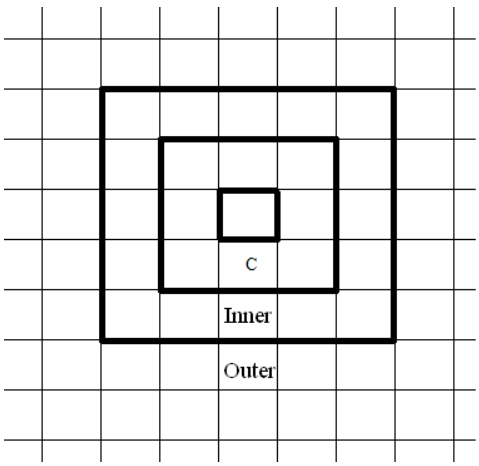

Figure 3.1: Example of the Grid-TNR grid; also cells, inner and outer squares.

In what follows we will compute shortest paths between nodes in C and border nodes of O and look for transit nodes among the endpoints of edges that cross the border of I. Let V_C be set of nodes as follows: for every link that has one of its endpoints inside C and the other outside C, V_C will contain the endpoint

inside C. Similarly, define V_I and V_O by considering links that cross I and O accordingly. Now, the set of transit nodes for the cell C (a.k.a. set of access nodes of C) is the set of nodes $v \in V_I$ with the property that there exists a shortest path from some node in V_C to some node in V_O which passes through v. For the directed graphs, we will differentiate between two types of access nodes (i) *outgoing* access nodes - is the set of nodes $v \in V_I$ such that there exists a shortest path from some node in V_C to some node in V_O which passes through v (ii) *incoming* access nodes - is the set of nodes $v \in V_I$ such that there exists a shortest path from some node in V_O to some node in V_C which passes through v. We associate every node inside C with the set of access nodes of C. Next, we iterate over all cells and similarly identify access nodes for every other cell. The union of all access nodes comprise a set of transit nodes.

Computation and Storage of Distances

Once we have identified all transit nodes we compute and store, for every node of the graph, the shortest distance (or fastest time) from this node to all its access nodes. Recall from the previous section that every such node $v \in V$ is associated with the set of access nodes that were found for its cell. In addition we also compute and store the shortest distance (or fastest time) from each transit node to every other transit node. We store those distances in three tables (*node-to-transit* (N2T), *transit-to-transit* (T2T), *transit-to-node* (T2N). In Section 3.4.7 we will describe how to store this information efficiently.

In an undirected graph it was enough to compute and store costs in only one direction. For a directed graph we will have to perform those two tasks in the opposite direction as well. This can be efficiently done by using reverse links.

3.2.3 Local Search Radius

Grid-TNR distinguishes between two types of queries: local and global. Two nodes for which horizontal or vertical distance (as measured in cells) is greater than some *local search radius* are considered to be "far away" and the query between them is called global. We define the local search radius to be equal to the size of the inner square I plus the distance from I to the outer square O. This definition guarantees that for each global query two important conditions are satisfied: (i) the start node *src* and destination node *dst* are not inside the outer squares of each other (ii) their corresponding inner squares do not overlap. Both conditions are necessary to ensure that the Grid-TNR is correct and optimal.

3.2.4 The Shortest Distance Query

For every global query from *src* to *dst* we fetch the access nodes T_{src}, T_{dst} of the cells containing *src* and *dst* and choose those two that will give us a minimal cost of the combined three subpaths: $src \rightsquigarrow t_{src}$, $t_{src} \rightsquigarrow t_{dst}$, $t_{dst} \rightsquigarrow dst$, where $t_{src} \in T_{src}$, $t_{dst} \in T_{dst}$. More formally:

$$dist(src, dst) = \min_{\substack{t_{src} \in T_{src} \\ t_{dst} \in T_{dst}}} dist(src, t_{src}) + dist(t_{src}, t_{dst}) + dist(t_{dst}, dst)$$

The idea is depicted in Figure 3.2 below.

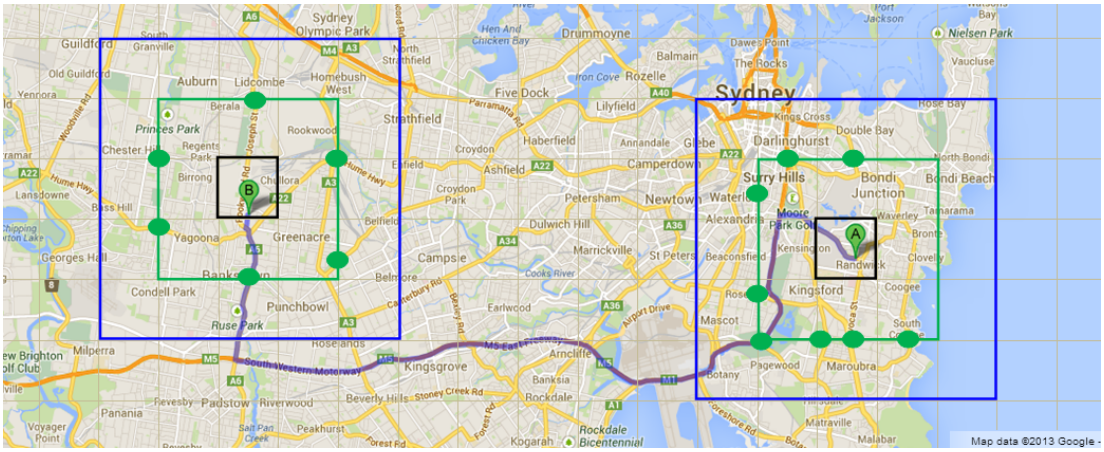

Figure 3.2: Example of Grid-TNR query.

Local Queries

For local queries, we need to resort to some other efficient algorithm. Luckily, (i) there are typically not many local queries (1%-2%) and (ii) source and destination for such queries are close to each other and therefore most of algorithms (see Section 3.5 for more discussion) are relatively fast. In [Lingkun Wu and Zhou, 2012], the authors showed that CH [Geisberger et al., 2008] is often the best choice for dealing with local queries.

3.2.5 Extracting the Shortest Path

Until now, not much work has been done on efficient path extraction using TNR precomputed databases. An intuitive and somewhat naive way would be performing a series of repeated distance queries of the algorithm. In the original paper, the authors suggested first finding the next adjacent node to the source on the shortest path and then iteratively applying a Grid-TNR query from that

node to extract the full path [Bast et al., 2006]. An immediate improvement of this approach would be that we can store the next node of every precomputed shortest path, rather than search for it. Then we apply a similar technique, by simply fetching the next adjacent node of the shortest path. It will require an additional entry of 4 bytes[1], for every precomputed sub-path.

A more sophisticated improvement can be achieved by exploiting a property of the optimal substructure of the shortest path, i.e. any subpath of the shortest path is a shortest path. Using this, we observe that T_{dst} is the correct transit node for any sequential query and we can reuse it, see Figure 3.3. In the example in the Figure 3.3, for all the queries $src \rightsquigarrow dst$, $u \rightsquigarrow dst$, $v \rightsquigarrow dst$, T_{dst} will be always the same access node of the dst. Therefore, for the sequential subpath $u \rightsquigarrow dst$ we know apriori the transit node of the dst and can save time by not searching for T_{dst} and fetching $dist(T_{dst}, dst)$ every iteration, but rather reuse it. Since the query time is quadratic in the number of access nodes, this allow us to reduce all subsequent query times to be linear in the number of access nodes.

[1]The typical size of integer value is 4 bytes, but generally it depends on the platform and the compiler.

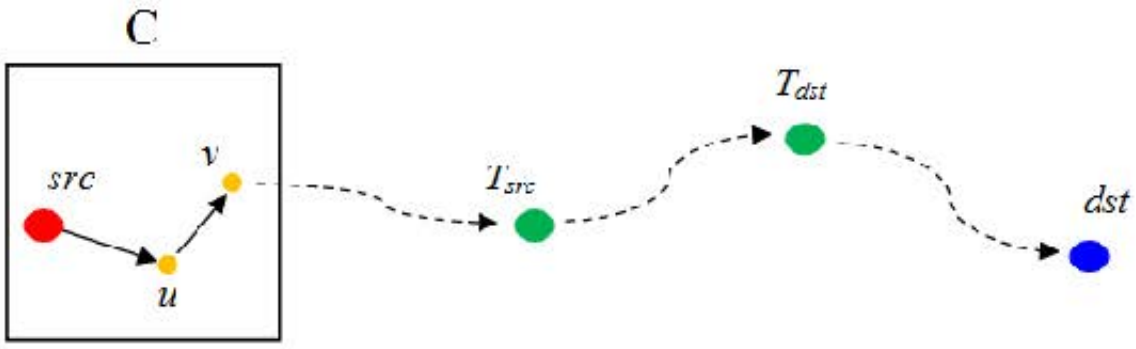

Figure 3.3: Query time as a function of number of access nodes

Moreover, we are also exploiting the fact that access nodes are associated with a cell, therefore any two nodes in the same cell will have exactly same access nodes. For example, in Figure 3.3 nodes src and u have exactly the same set of access nodes. Moreover, the shortest path from u to dst will necessary pass via T_{src}. It means that for any subsequent node v on the path that is in the same cell as its predecessor u (i.e. both u and v are in the same cell C), we know that optimal access node of u, T_u will be also an optimal access node of v. Therefore we don't need to spend time searching for it. This optimization makes the time complexity of every subsequent query constant, i.e. $O(1)$, which in turn makes the whole path extraction time complexity $O(k)$, where k is number of cells which the shortest path is crossing. In section 5.5 we will present in detail an even faster, novel approach for shortest path extraction.

3.2.6 Proof of Correctness

Let *src* and *dst* be source and destination nodes of interest. Assume that the query between *src* and *dst* is global (as defined in 3.2.3). In this case we execute the Grid-TNR query procedure as in 3.2.4. Let P be a path with cost $c(P)$ returned by Grid-TNR. The proof proceeds by contradiction. Suppose there exists another path $\hat{P}$ with cost $c(\hat{P})$, such that $c(\hat{P}) < c(P)$. Now, let $\hat{T}_1$ and $\hat{T}_2$ be transit nodes of $\hat{P}$ of *src* and *dst* respectively. Note, the way we've chosen transit nodes guarantee that $\hat{P}$ passes via some $\hat{T}_1$ and $\hat{T}_2$. Let us denote $\hat{P}_1 = src \rightsquigarrow \hat{T}_1$, $\hat{P}_2 = \hat{T}_1 \rightsquigarrow \hat{T}_2$ and $\hat{P}_3 = \hat{T}_2 \rightsquigarrow dst$ subpaths of $\hat{P}$. The shortest path optimal substructure property [Cormen et al., 2001] (pp. 581-582) says that every subpath of a shortest path is a shortest path. Thus it follows that $\hat{P}_1$, $\hat{P}_2$ are $\hat{P}_3$ are all shortest paths. Similarly, let be $P_1 = src \rightsquigarrow T_1$, $P_2 = T_1 \rightsquigarrow T_2$ and $P_3 = T_2 \rightsquigarrow dst$ be subpaths of P. By assumption (at least) one of the following must be true: $c(P_1) > c(\hat{P}_1)$ or $c(P_2) > c(\hat{P}_2)$ or $c(P_2) > c(\hat{P}_2)$. But this contradicts the fact that Grid-TNR query returns a minimal sum of subpaths from *src* to *dst*. Q.E.D.

3.2.7 Complexity Analysis

In this section we present an approximate asymptotic complexity analysis of TNR. The complexity of the algorithm depends on many factors, such as graph nodes distribution, node connectivity, etc. Intuitively we can see that if we choose the inner square, I to be very small (say containing only one node), then every node will be a transit node and the precomputation will compute all the shortest paths. In this case a query will be a simple lookup in a large precomputed table. On

the other extreme, suppose we choose I to contain all the nodes. In this case, every query is local and we do no precomputation. So we can observe that there is a clear tradeoff between size of the squares, precomputation time, storage and query time. In what follows, we will assume a simplified "grid world" graph layout, where nodes are equally distributed and every node is connected to its four neighbors by a link of unit cost. Let k denote the number of cells and $n = |V|$ denote the number of nodes. Consider cell C. Then $|V_C| = \frac{n}{k}$. Let $Inner$ be a square centred on C consisting of some constant number of cells. In the worst case, for "grid world" graph, the number of transit nodes for cell C equals the number of border nodes of $Inner$, which is $O(\sqrt{\frac{n}{k}})$. Since we have n nodes, the storage space for the *node-to-transit* table will be $O(n\sqrt{n})$ for any choice of k, which may be prohibitive.

In real life networks, not every road has the same travel time. There are highways, major roads, minor roads, etc. In order to make our simplified "grid-world" graph resemble a real life road network we assume that every, say, 10th vertical and horizontal road is a highway. We will model this by assigning significantly smaller cost to such highways. Since every cell C is of bounded size, it follows that only a constant number of highways cross every cell. Consequently, since the "grid world" network is planar and all the long shortest paths will converge to use those highways and the number of transit nodes for every cell is $O(1)$. This gives $O(n)$ storage space for the *node-to-transit* table. Now, the total number of transit nodes is $O(k)$. Therefore, if we choose k to be $O(\sqrt{n})$ we need $O(k^2) = O(n)$ storage space for the *transit-to-transit* table.

Our experiments support such a model as we observed that the storage space did indeed scale linearly.

3.2.8 Speed-Up Techniques

When determining transit nodes in 3.2.2, we need to execute the Dijkstra algorithm from every node on a border of a cell C to every node of a border of the outer square O. In many cases, when the number of border nodes of O is greater than the number of border nodes of C (Figure 3.6), we can speed-up the computation. It is a well known fact that in order to solve single source-many destinations, a.k.a. *one-to-many* shortest path problem, a slight modification of Dijkstra will be more efficient than running many times the basic *one-to-one* Dijkstra algorithm. Similarly, running k times *one-to-m* will be faster than running m times *one-to-k*, where $k < m$. With this observation, when the number of border nodes of O is greater than the number of border nodes of C, we are using the *one-to-many* variation of Dijkstra, by initializing the priority queue with target nodes and considering backward links. This gives a significant improvement in the preprocessing run time.

Another significant improvement is to notice that there is no need to precompute and store distances from/to every node in the network. Without loss of accuracy we can ignore "dead-end" nodes and nodes of degree two. Similar ideas exist in the CH approach [R. Geisberger, 2012], which is equivalent to initial contraction of all nodes that can be contracted without the need to add shortcuts. Given a query where source or destination is one of these nodes we can just simply follow the only path from this node until we encounter the first node whose degree is greater than two. We observed around a 20% reduction in the number of nodes for which we should perform a precomputation stage 3.2.2 on the road network of W. Europe. This gives another significant improvement in

both preprocessing time and storage space.

For directed networks, the set of *outgoing* access nodes may differ from the set of *incoming* access nodes. In this case, we can save computational time and space, by ignoring pairs of access nodes $(t_{incoming}, t_{outgoing})$, simply because this pair will never be realized.

One can notice that the process 3.2.2 of identifying transit nodes for a cell is completely independent of other cells and therefore can be easily parallelized. Similarly we can observe that the precomputation and storage stage 3.2.2 of the three tables (*node-to-transit* (N2T), *transit-to-transit* (T2T), *transit-to-node* (T2N)) is also independent and can be performed in parallel. Of course, while writing to those databases we need to take care of synchronization issues.

In Section 3.2.9 we will discuss more low level engineering consideration for speeding up TNR queries.

3.2.9 Experiments

We implemented the Grid-TNR algorithm in Java and the online query method in C++ compiled (with /Ox optimization) under Microsoft Visual Studio C++ 2012. Online queries were tested on a machine running Windows Server 2012 with 64Gb of DDR3-1333RAM and 6-core Xeon X5650 CPUs at 2.66GHz [Intel]. We tested our implementation on three continental size networks:

Australia	$(\|V\| = 6.1 \times 10^6, \|E\| = 12.4 \times 10^6)$ taken from OSM [OSM]
Western Europe	$(\|V\| = 18 \times 10^6, \|E\| = 42 \times 10^6)$ and
USA	$(\|V\| = 24 \times 10^6, \|E\| = 58 \times 10^6)$

taken from the 9th DIMACS Implementation Challenge [Demetrescu et al., 2009].

We measure the global query time by running 10^7 random queries (picked uniformly in advance).

Low level optimization of the Query

Generally, the query procedure is very simple and can be summarized in pseudocode in Algorithm 1:

```
For every i ∈ 1,...|T_src|
    t_src ← fetch T_src
    d_1 ← dist(src, t_src)
    For every j ∈ 1,...|T_dst|
        t_dst ← fetch T_dst
        d_2 ← dist(t_dst, dst)
        d ← fetch dist(t_src, t_dst)
        if (d_1 + d + d_2 < min_d)
            min_d = d;
return min_d;
```

Algorithm 1: Pseudo code of procedure for finding the shortest path distance

However, while this is correct, that would be not the most cache-efficient way to implement the query. Since access to the main memory is much slower than the CPU clock cycle, memory access becomes a main bottleneck of the query routine. For example on our machine the processor clock run at a speed of 2.66GHz but the memory runs at 1333Mhz, or 1.3GHz, which is considerably slower. The processor essentially has to wait until the memory responds and transfers the data. In order to improve this bottleneck, processor manufacturers have added a small amount of very fast memory close to the processor itself, a.k.a. cache memory. When accessing main memory, the cache is filled with data close to the requested data, under the assumption that it may be needed again, and on

subsequent accesses the data is returned from the very fast cache if its still there. This spares the processor from reaching out to main memory and waiting for it to react and return the data. On modern processors, access to cache memory takes around 7 nano-seconds while accessing main memory (a.k.a. cache miss) may take up to 50 nano-seconds. Similarly in our query, the main bottleneck is a memory access rather than arithmetic operations. In the worst case, it may require $|T_{src}| + |T_{dst}| + |T_{src}||T_{dst}|$ accesses to RAM, which is somewhat wasteful. With a little bit of careful engineering, we can reduce this number to $2|T_{src}| + 2|T_{dst}|$.

```
For every i ∈ 1,...|T_src|
    t_src,i ← fetch T_src
    d_1,i ← dist(src, t_src)
For every j ∈ 1,...|T_dst|
     t_dst,j ← fetch T_dst
     d_2,j ← dist(t_dst, dst)
For every i ∈ 1,...|T_src|
    For every j ∈ 1,...|T_dst|
        d ← fetch dist(t_src,i, t_dst,j)
        if (d_1,i + d + d_2,j < min_d)
           min_d = d;
return min_d;
```

Algorithm 2: Pseudo code of cache efficient procedure for finding the shortest path distance

This implementation, is more cache-efficient because (with *careful engineering*) all access nodes of a *src*/*dst* can be loaded to a cache in one fetch (rather many as before). By *careful engineering* we mean that we need to store related data physically close to each other. For example, when we are precomputing and storing access nodes of a node, it is more cache efficient to store this information sequentially in the memory. This is even more important, when we precomputing

and storing *transit-to-transit* tables.

Query Time

Given two nodes *src* and *dst* that are "far away", the query time depends mostly on the number of *outgoing* access nodes of *src* and the number of *incoming* access nodes of *dst*. Therefore, the average number of access nodes per node is a good quantitative, machine independent measure for the speed of TNR query. Below, in Figure 3.31 we present a graph that plots query time as a function of the average number of access nodes.

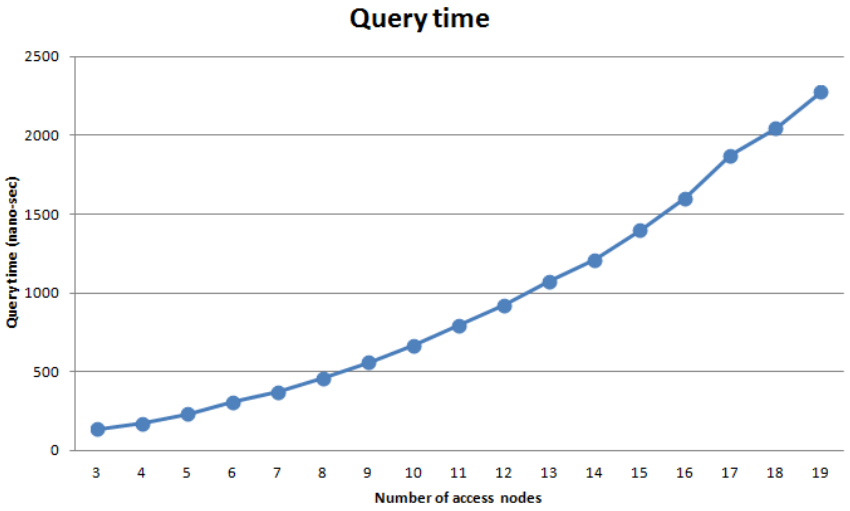

Figure 3.4: Query time as a function of number of access nodes

We fit the runtime to the model Ax^b and found a good fit for b=1.17 with the residual sum of squares (RSS) about 10^5. This suggests that, whilst growth is greater than linear, it is not much greater and is less than quadratic.

Mistake in original Grid-TNR

During very careful examination we have discovered that there is a mistake in the original Grid-TNR [Bast et al., 2006]. In order to speedup the preprocessing authors suggested the *sweep-line algorithm*. The motivation was to reduce the search space of the Dijkstra when identifying *transit* nodes.

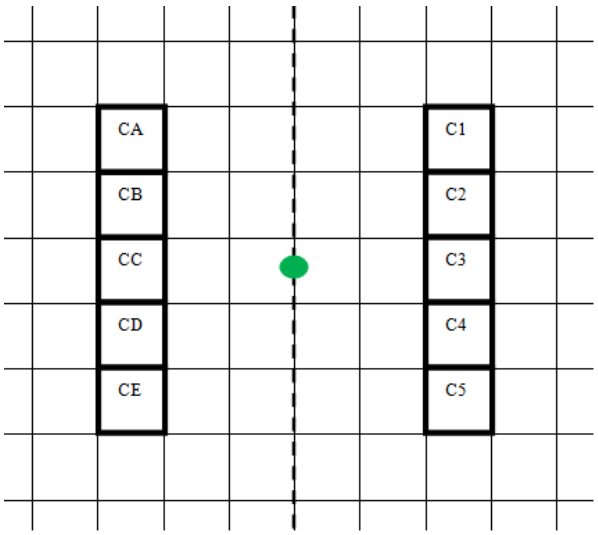

Figure 3.5: Example of *sweep line* algorithm

The *sweep line* algorithm identifies potential *transit* nodes that reside on a vertical (or horizontal) line as follows. In the example in Figure 3.5, for all nodes on a vertical *sweep line* the suggested algorithm executes Dijkstra until all nodes on the border of $C_{left} = \{CA, CB, CC, CD, CE\}$ and $C_{right} = \{C1, C2, C3, C4, C5\}$ are settled (closed). Then, for every pair (v_L, v_R), where v_L is on the boundary of C_{left} and v_R is on the boundary of C_{right}, a node v with minimal $d(v_L, v)+d(v, v_R)$ identified as a transit node. The hidden assumption of the *sweep-line algorithm* is that the transit node on (for example) a horizontal *sweep line* may only appear

on the shortest paths between a node in C and the node on the horizontal border of the outer square O. In reality this may not always be the case. In the example below we schematically present a configuration where the *sweep-line algorithm* will fail to identify v as a *transit* node. Independently, a similar observation was reported in [Lingkun Wu and Zhou, 2012].

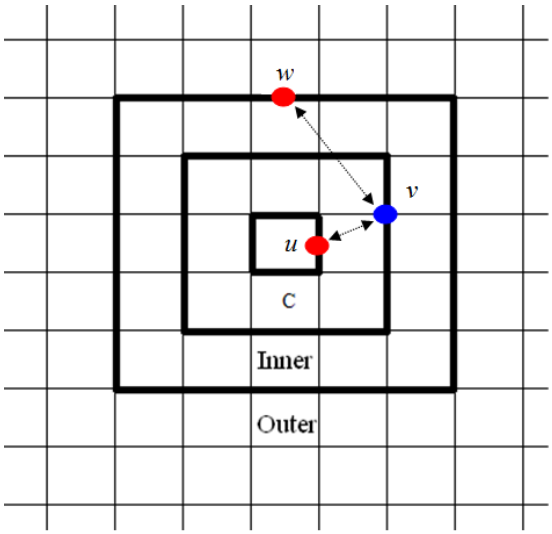

Figure 3.6: Example of the Grid-TNR grid and nodes configuration where the *sweep line algorithm* will fail

This, in turn, results in underestimating the number of *transit* nodes and returns non optimal paths.

In addition, the authors reported average number of access nodes per cell which is somewhat a "skewed" measure. It may be the case (more likely for a finer grid), where some of the cells are empty (e.g. overlay over mountains or forest) or some cells may contain roads where all the nodes are of degree two (for which we do not calculate access nodes). Those type of cells, see Figure 3.7 for example, do

not contribute to the total count of the transit nodes, but reduce the reported average. We report the average number of access nodes per node, for which we actually calculate and store access nodes (i.e. we are excluding "dead-ends and nodes of degree 2). We believe that this is a better quantitative measure that more accurately correlates with global query time.

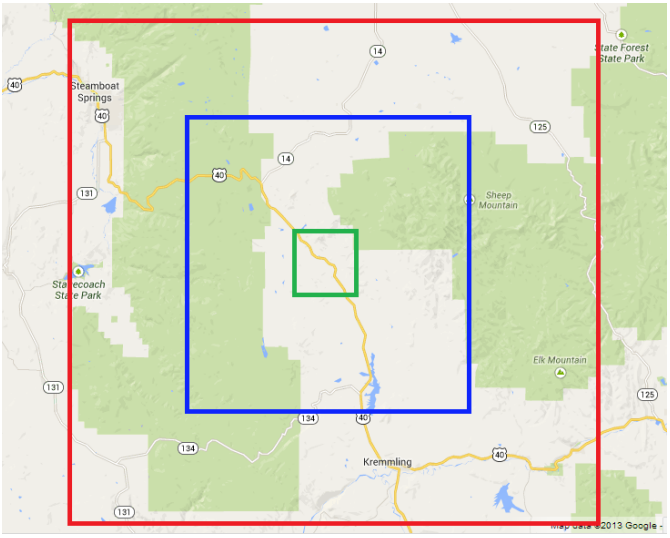

Figure 3.7: Example of a cell with zero transit nodes

Finally, the authors originally used the *time metric* on an undirected version of the US network. This somewhat "favors" shortest paths algorithms, because the time metric induces a better hierarchy between the links, whilst the distance metric does not. Consider the example in the Figure 3.8 below. Assume that all the links are residential (slow) roads with distance 1 and only the link (y, d) is a much faster, but slightly longer highway, say of distance 2. Using the distance metric we will find two shortest paths $s_1 \to x \to d$, $s_2 \to z \to d$ that give us two transit nodes $\{x, w\}$. With the time metric however, it is clearly better to travel

via the faster highway $s_1 \to y \to d$, $s_2 \to y \to d$ that gives us only one transit node y.

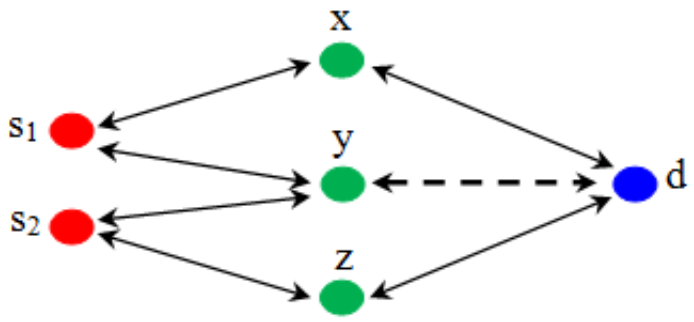

Figure 3.8: Example of an undirected graph, where time metric yields fewer transit nodes than a distance metric

Therefore, not surprisingly, using real distances and not times is more challenging and produces more transit nodes. Below is a table that summarize the difference of running Grid-TNR on undirected Europe and USA networks using the time metric (as was reported originally) vs. using the distance metric (calculated as a great circle distance).

	Europe (time)			Europe (dist)		
Grid Size	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$
256 x 256	41 340	7.7 Gb	11.6	108 963	47.3 Gb	39.3
512 x 512	133 253	68.9 Gb	11.4	276 014	292.2 Gb	30.8
1024 x 1024	366 173	512.8 Gb	10.4	631 108	1.5 Tb	22.8

Table 3.1: Comparison of time metric vs. distance metric for the European network

	USA (time)			USA (dist)		
Grid Size	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$
256 x 256	97 428	38.3 Gb	14.2	213 248	175 Gb	32
512 x 512	340 179	443 Gb	13.1	548 774	1.2 Tb	24.8
1024 x 1024	996 244	3.8 Tb	11.6	1 276 153	6.2 Tb	18.3

Table 3.2: Comparison of time metric vs. distance metric for USA network

In the next Section we present an extensive evaluation of Grid-TNR for three continental size road networks. We report results for both the undirected version (using great circle distance as a metric) and the directed version (using provided travel time as a metric). We report total number of trasnit nodes, additional storage requirements, average number of access nodes (per node), and percentage of global queries out of all queries.

Results

In previous section we have discussed the query time of Grid-TNR. In what follows we present other Grid-TNR critical performance measures. We report total number $|\mathfrak{T}|$ of transit nodes, additional space requirements, average number $|A|$ of access nodes, percentage of global queries for distance metric (undirected version) and travel time metric (directed version).

	Australia (time)			Australia (dist)			
Grid Size	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	% Global
64 x 64	1568	0.4 Gb	8.4	2141	0.4 Gb	14.4	80.7%
128 x 128	4414	0.5 Gb	8.1	5710	0.5 Gb	14.1	89%
256 x 256	11 781	0.9 Gb	8	14 299	1.1 Gb	13.1	93.8%
512 x 512	29 874	3.8 Gb	7.9	34 425	4.8 Gb	13.2	96.6%
1024 x 1024	71 603	19.9 Gb	7.9	77 971	23.5 Gb	12.8	98.4%

Table 3.3: Results of Grid-TNR for Australia network.

	USA (time)			USA (dist)			
Grid Size	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	% Global
64 x 64	7472	2.7 Gb	16.7	25 431	5.8 Gb	46.5	92.5%
128 x 128	26 468	4.9 Gb	15.5	76 280	25.1 Gb	39.4	97.7%
256 x 256	97 428	38.3 Gb	14.2	213 248	175 Gb	32	99.3%
512 x 512	340 179	443 Gb	13.1	548 774	1.2 Tb	24.8	99.8%
1024x1024	996 244	3.8 Tb	11.6	1 276 153	6.2 Tb	18.3	99.9%

Table 3.4: Results of Grid-TNR for USA network.

	Europe (time)			Europe (dist)			
Grid Size	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	% Global
64 x 64	3710	1.2 Gb	11.5	12 726	3.3 Gb	52.9	84 %
128 x 128	12 049	1.7 Gb	11.4	38 968	8.2 Gb	47.5	94.9 %
256 x 256	41 340	7.7 Gb	11.6	108 963	47.3 Gb	39.3	98.4%
512 x 512	133 253	68.9 Gb	11.4	276 014	292.2 Gb	30.8	99.5%
1024x1024	366 173	512.8 Gb	10.4	631 108	1.5 Tb	22.8	99.8%

Table 3.5: Results of Grid-TNR for Europe network.

Directed vs. Undirected Graphs

We notice that in undirected graphs (e.g. using distance metric) the average number of access nodes is larger than in a directed graphs (e.g. using time metric), but the total number of access nodes in undirected graphs is smaller. This "phenomenon" can be explained by the following observation. Consider graphs in Figures 3.9 and 3.10 below. Assume all the link weights are 10 and only the link between y and d_3 has weight of 1.

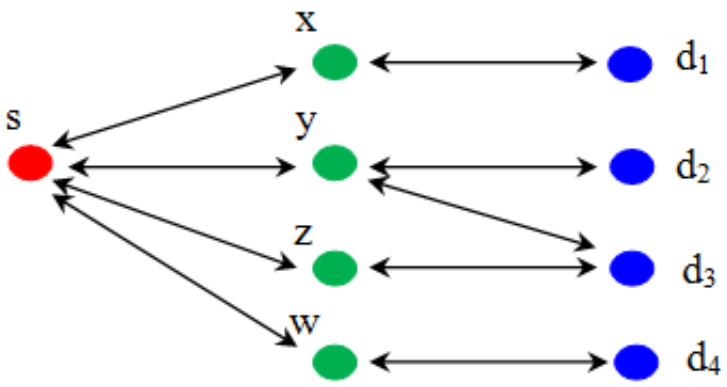

Figure 3.9: Example of undirected graphs, where we have 3 access nodes

In the undirected version we have 4 paths in total that connect s with $\{d_1, d_2, d_3, d_4\}$ in both directions: $(s \leftrightarrow x \leftrightarrow d_1)$ $(s \leftrightarrow y \leftrightarrow d_2)$ $(s \leftrightarrow y \leftrightarrow d_3)$ $(s \leftrightarrow w \leftrightarrow d_4)$ and 3 access nodes $\{x, y, w\}$ in total.

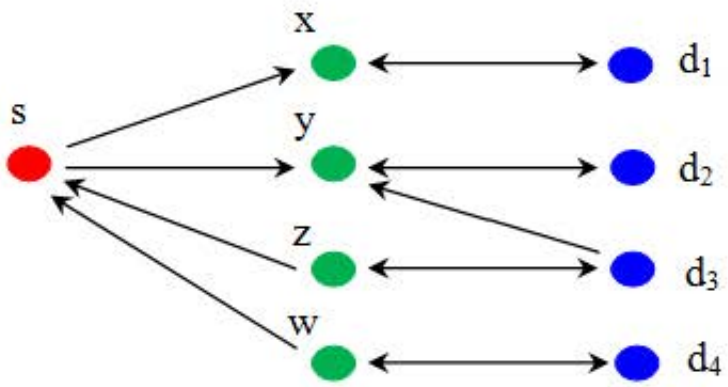

Figure 3.10: Example of directed graphs, where we have 2 *outgoing* and 2 *incoming* access nodes and totally 4 access nodes

In the directed version, on the other hand, we have 2 paths that connect s with $\{d_1, d_2\}$, $(s \to x \to d_1)$ $(s \to y \to d_2)$ and 2 paths that connect $\{d3, d4\}$ with s $(d_3 \to y \to s)$ $(d_4 \to w \to s)$. Therefore, we have 2 *outgoing* access nodes $\{x, y\}$ and 2 *incoming* access nodes $\{z, w\}$. So in total we have 4 access nodes. However, for the sake of queries that start (or finish) at s, we are interested only in 2 *outgoing* (or *incoming*) access nodes $\{x, y\}$. So, in this example, for directed graphs we have a smaller number of *outgoing* (or *incoming*) access nodes, but more access nodes in total.

3.2.10 Minimizing Number of Access Nodes

We recall that Grid-TNR identifies access nodes on a border of the inner square (see 3.2.2). Since the query time is quadratic in the number of access nodes, it is highly beneficial to minimize this set. Lets take a closer, more careful look on how we choose those border nodes.

A border node of an inner square is one of the endpoints of the link that crosses this square. When deciding what endpoint of the crossing link to choose, Bast et al. [Bast et al., 2006] suggested to choose the node with the minimum index[1]. This is a good strategy, because it will prevent choosing unnecessary transit nodes that essentially are endpoints of the same link. However, if we take a closer look, we may notice that in some cases there is a room for improvement, and we can actually reduce the number of access nodes of a cell. Consider an example in Figure 3.11 below.

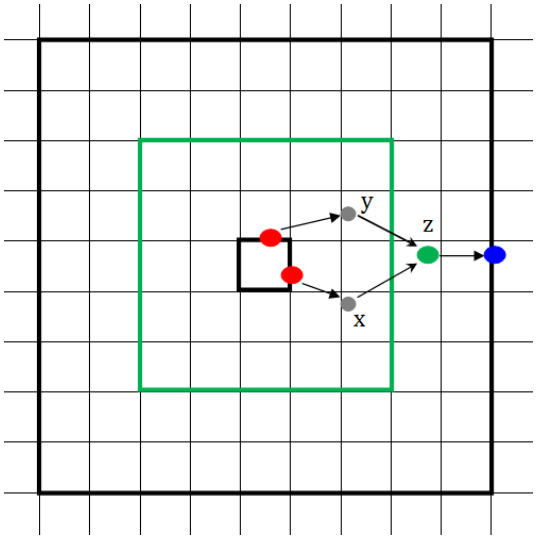

Figure 3.11: Example of a non optimal choice of transit nodes

[1]Index of a node can be any unique random number. For example it can be its address in the memory or the index can be assigned when creating a graph.

If indices of x and y are less than index of z, we may identify x and y as two access nodes. Clearly, choosing z in this case would be a better choice, because we will have only one access node instead of two. In the worst case, a bad choice of the end-point of the crossing link may unnecessarily increase the number of access nodes, also making the query times significantly slower.

Our solution is to search for access nodes from a *minimum vertex cover* (see 2.5) of the *bipartite graph* (see 2.4) where the left nodes of the graph are inside the outer square and the right nodes of the bipartite graph are outside the inner square. For this we have an additional step of finding a minimum vertex cover of all the crossing links of the inner square. From König theorem 2.4 it follows that finding a minimum vertex cover in a bipartite graph reduces to a maximal matching problem. This problem can be efficiently solved by finding a maximum flow in a corresponding unit capacity flow network [Cormen et al., 2001] (pp. 664-667). In practise we saw a reduction up to 35% in the number of access nodes by this method. However, this method is not discussed any further, because it is superseded by a method described in the next paragraph.

Next we show how to find the minimum set of access nodes, which will guarantee us the fastest possible queries Grid-TNR can achieve. For the sake of simplicity, the following discussion will talk about set of *outgoing* access nodes. Exactly the same holds for *incoming* access nodes. Consider the example in Figure 3.12. Intuitively, the access nodes of a cell C can be anywhere[1] on the shortest path between the cell C and outer square O without compromising the

[1] access node can also appear outside the outer square, as long it is on a shortest path that connects C with O

correctness of the algorithm.

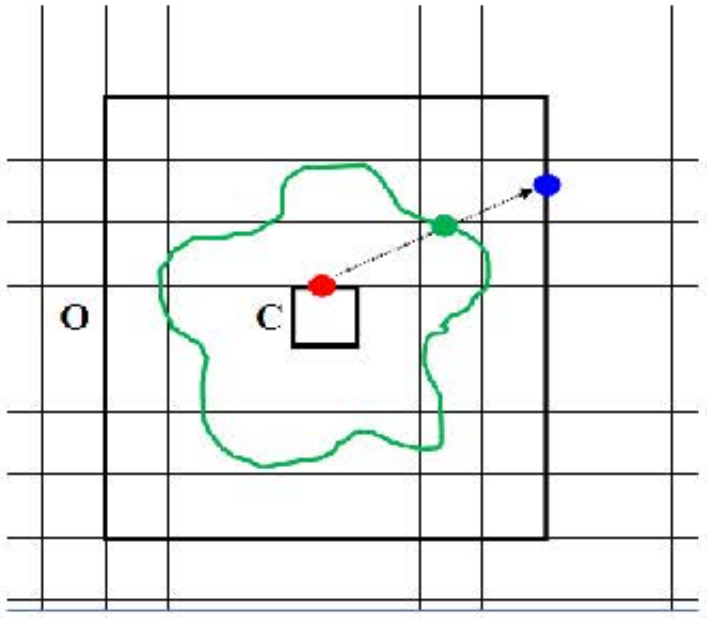

Figure 3.12: Example of the cell and outer square, where the access nodes can appear anywhere between them.

With this observation, we are looking to minimize the number of access nodes in order to have faster queries. One of weaknesses of Grid-TNR is that the overlayed grid is rigid and does not take into account the underlying topology of the network. There is no guarantee that the inner square will fall exactly in the "good" spot (for example, the entrance to a bridge or a tunnel).

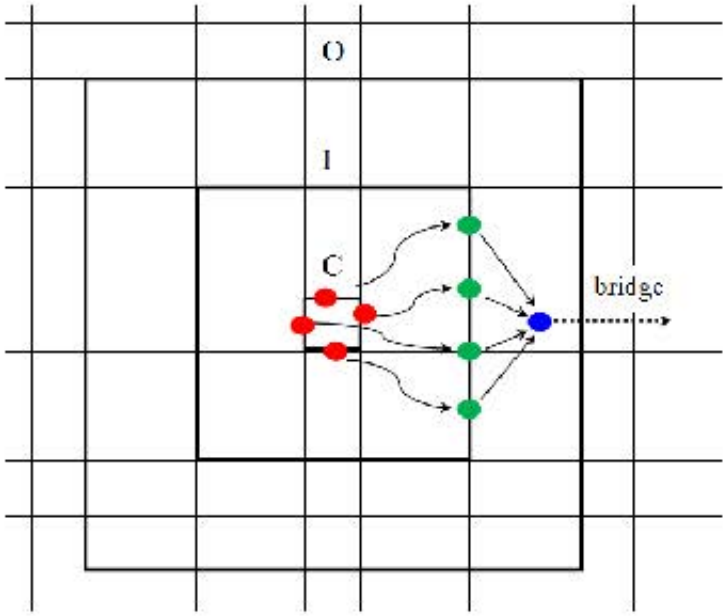

Figure 3.13: Example of more efficient choice of transit nodes.

In the example in Figure 3.13 above, Grid-TNR would find 4 transit nodes (that reside on the border of the inner square), where intuitively we can see that it is more efficient to choose only one transit node (which is not necessarily on the border of the inner square).

Clearly, the minimum set of access nodes, A, should contain at least one node from any shortest path from V_C to V_O. Therefore, one of the most natural candidates to search the access nodes from (instead as originally on the border of the inner square), is the minimum separation vertex set (see Definition 2.14) that disconnects all the shortest paths from V_C to V_O. Please notice, that we do not require to completely disconnect V_C from V_O, but only to disconnect all the shortest paths.

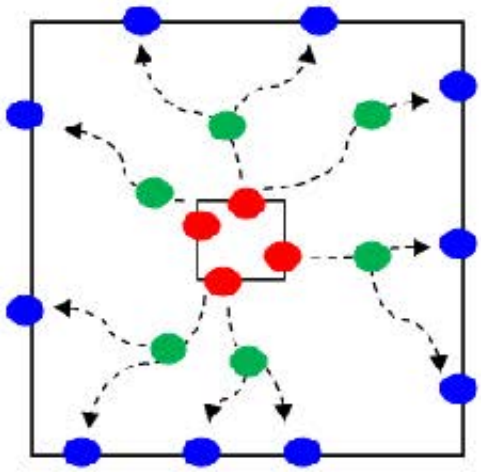

Figure 3.14: Schematic representation of the alternative set of access nodes, depicted in green. Red nodes are the border nodes of the cell C, V_C and blue nodes are the border nodes of the square O, V_O.

Definition 3.1. Let $G = (V, E)$ be a graph, where E is all the edges that lie on all the paths between V_C and V_O and V is set of nodes of those paths. We define a subgraph $G_{C \rightsquigarrow O} = (V_{C \rightsquigarrow O}, E_{C \rightsquigarrow O})$ that contains only edges on the shortest paths[1] between V_C and V_O and nodes of those edges. We define a new set of access nodes A to be the minimum $V_C - V_O$ vertex separation set in $G_{C \rightsquigarrow O}$. Please notice, by the definition of the separation set, we guaranteed that A won't be empty (unless C and O are disconnected in the first place).

Claim 3.1. *For every cell C, the set of access nodes A defined as in 3.1, is the minimum set of access nodes over all other possible choices of access nodes set.*

[1] w.l.og. we can assume that the shortest path between any pair of nodes is unique, otherwise we can add very small random noise to break the symmetries 5.4

Proof. Assume, by contradiction that there exist another, minimum set of access nodes $\bar{A}$, such that $|\bar{A}| < |A|$. Clearly, $\bar{A} \subseteq V_{C \rightsquigarrow O}$ (i.e. all the nodes of $\bar{A}$ lie on some shortest path), because otherwise we could find and remove node $x \in \bar{A}$ that is not on a shortest path between C and O, contradicting minimality of $\bar{A}$. In addition, since $\bar{A}$ contains (at least one) node from all the shortest paths from C to O, removal of nodes of $\bar{A}$ from $G_{C \rightsquigarrow O}$ will necessarily disconnect all the shortest paths from C to O, cause otherwise $\bar{A}$ wouldn't be a complete set of access nodes and there would be a shortest path that $\bar{A}$ does not cover. This in turn contradicts minimality of A which is defined to be a minimum vertex separation set. □

We are only left to show how we find the minimum separating vertex set of $G_{C \rightsquigarrow O}$. For that we define a flow network N as follows. We introduce two virtual nodes s and t and connect s to V_C (border nodes of C) and V_O (border nodes of O) connected to t. We set the capacity of all the links to one. The resulting network N is depicted in Figure 3.15 below. Clearly, an $s - t$ separating vertex set in N is a $V_C - V_O$ separating vertex set in G.

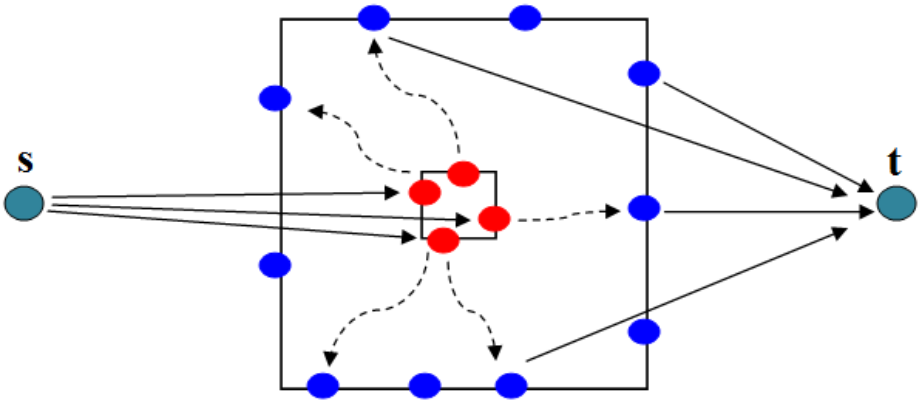

Figure 3.15: Schematic representation of the flow network N. Red nodes are the border nodes of the cell C, V_C and blue nodes are the border nodes of the square O, V_O.

Menger's theorem (see 2.5) suggests to us that in order to find a minimum $s-t$ separating set we need to find the maximum number of vertex disjoint paths in N. We can reduce this problem into a problem of finding the maximal number of edge disjoint paths in a modified network N^* that defined as follows [Marcon, 2012] (pp. 137-141):

(1) Replace each vertex $x \neq s, t$, by two vertices x_1, x_2

(2) Add a directed edge (x_1, x_2) with capacity 1

(3) Replace each directed edge (w, x) in N by a directed edge (w, x_1)

(4) Replace each directed edge (x, w) in N by a directed edge (x_2, w)

Similar to N, all capacities are set to be one.

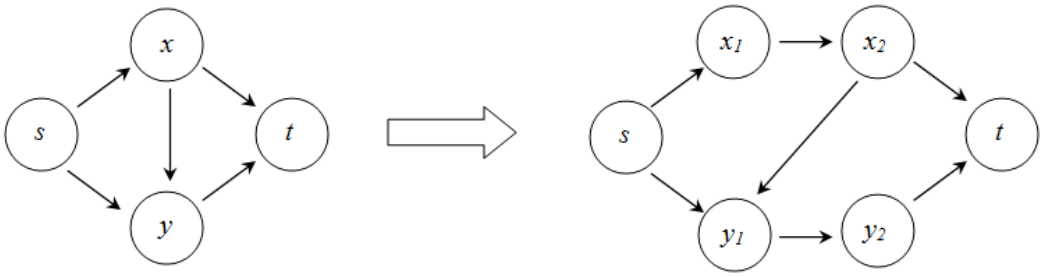

Figure 3.16: Example of a network N^* obtained from N.

Therefore, the problem of finding a minimum set of access nodes for a cell C reduces to a problem of finding a minimum cut in N^*. There are several ways to find a minimum cut in a network [Pothen et al., 1990]. We will follow the *max-flow min-cut* approach. We find a maximum flow in the network N^*, using Ford-Fulerson-Edmond-Karp (FFEK) algorithm [Cormen et al., 2001] (p. 660). Then we identify all the reachable nodes in the residual network N_f^* (Figure 2.9). The reachable nodes are the endpoints of the minimum cut and are exactly the minimum set of access nodes we are after.

However, the query speed-up comes at the price of a larger total number of transit nodes, consequently larger memory requirements and precomputation time. For every cell, the only additional overhead over original Grid-TNR is executing FFEK, whose complexity is

$$O(|V_{C \rightsquigarrow O}||E_{C \rightsquigarrow O}|^2) = O(|V_{C \rightsquigarrow O}|^3) = O\left(\frac{n^3}{k^3}\right),$$

which is relatively fast in practise. The main drawback of this approach is a large number of total transit nodes. It can be easily explained looking at the Figures

3.17 and 3.18. Since, in the original Grid-TNR access nodes could only reside on a grid, many cells could *share* the same transit node. In new settings, this is less likely, because every cell has its own optimal set of access nodes, which can reside anywhere between the cell and its outer square

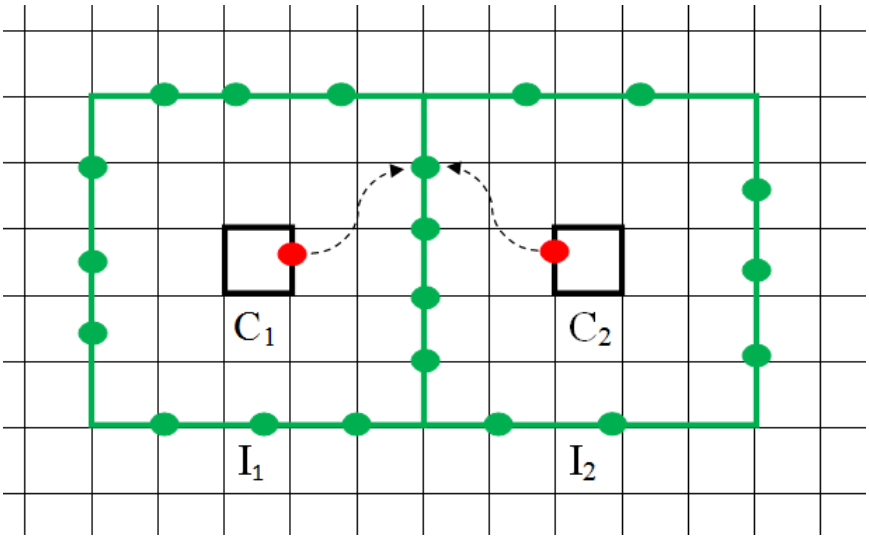

Figure 3.17: Schematic representation of the flow network N. Red nodes are the border nodes of the cell C, V_C and blue nodes are the border nodes of the square O, V_O.

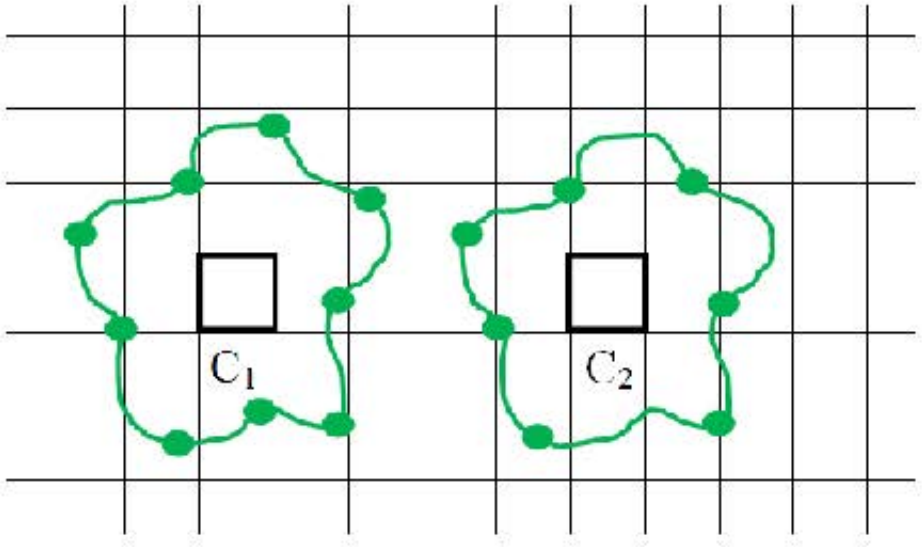

Figure 3.18: Schematic representation of the flow network N. Red nodes are the border nodes of the cell C, V_C and blue nodes are the border nodes of the square O, V_O.

We could achieve a very small number of access nodes (∼5 per node on average), making our queries very fast (measured in nano-seconds), but at a price of prohibitively large additional space. In Section 3.4 we will describe a novel algorithm that will successfully address this issue, without compromising query times.

3.3 Incremental Updating

In this section, we present a novel and efficient way for updating Grid-TNR precomputed databases when the weight of the links, as is typically happens, increases [Antsfeld and Walsh, 2012c]. This addresses one of the weaknesses of the Grid-TNR algorithm which assumes that the underlying network is static, i.e. the edge weights do not change. In reality the road conditions change quite frequently, due to planned (e.g. road repairs, special events) or unforseen events (e.g. car accident).

3.3.1 The Motivation

To provide an intuition for our update algorithm, consider the example below. Lets assume that the road (depicted in red) in the cell in the middle is blocked.

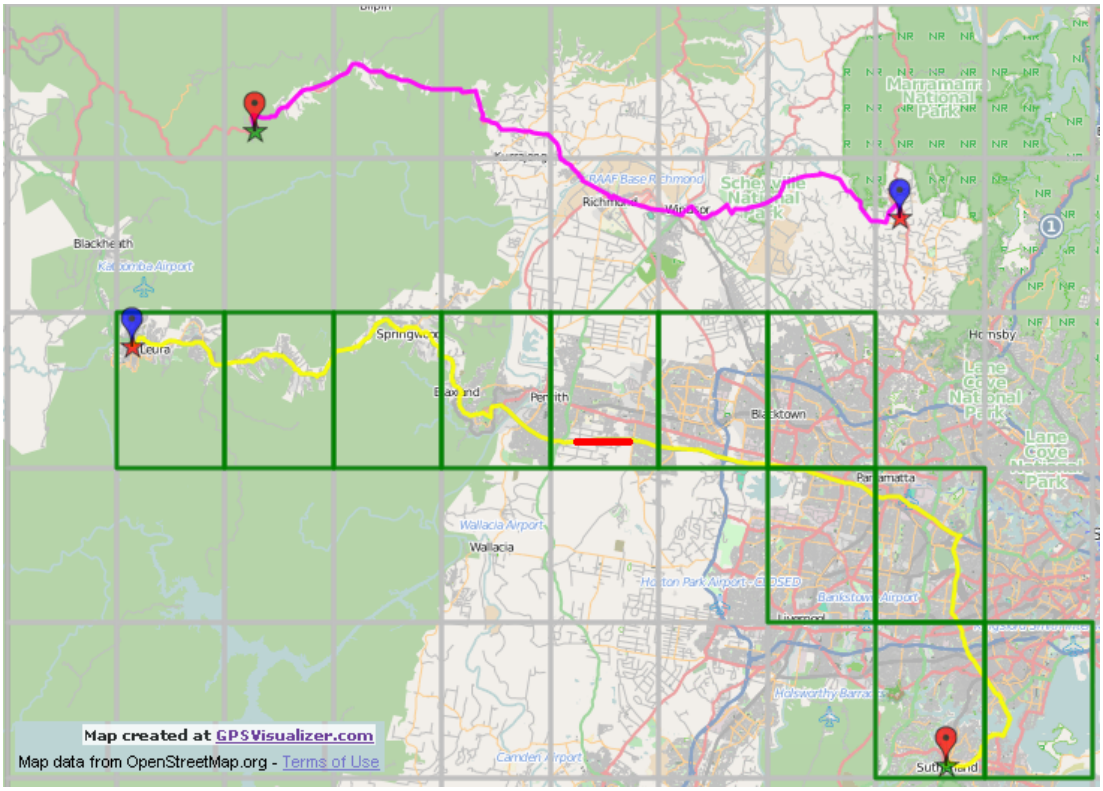

Figure 3.19: Example of the grid and two shortest paths, only one of which going via a blocked road (colored in red).

The traditional approach would be to precompute everything from scratch. As we intuitively understand and also the example shows, there are many shortest paths (including the path above) that are not affected by this change. Any efficient recomputation could ignore these unaffected shortest paths, and save this computational effort. In what follows we present an update algorithm that significantly reduces the amount of the required recomputation.

3.3.2 The Algorithm

We suppose that travel time over a link e, as it usually happens, increases (say because of road works). If only we knew all the pairs of nodes whose shortest path contains e, we could recompute and update the tables *node-to-transit* (N2T), *transit-to-transit* (T2T) and *transit-to-node* (T2N) (see 3.2.2) only for these specific pairs. In this case the recomputation would be minimal.

It is actually quite easy to "hack" Grid-TNR and obtain this information. During precomputation of the N2T, T2T, T2N tables, we can also extract the whole path and store it along with the distance. However, we would now need to store all the edges along the shortest path between pairs of nodes, rather than just the distance. For every link e, we store a set $\mathcal{P}_e = \{(src, dst)\}$ such that e is on a shortest path between every pair in $\mathcal{P}_e$. In the worst case, the asymptotical storage requirement for this additional information is $O(mn)$, where m is a number of links and n is the number of nodes in the graph. Unfortunately, this can be prohibitively large in practice.

In order to reduce the storage requirements, our second attempt was to store this information for pairs of cells rather than pairs of nodes. In other words, for every link e, we store a set $\mathcal{P}_e = \{(C_o, C_d)\}$ such that e is on a shortest path between a node in C_o and a node in C_d. The worst case storage for this case is $O(mk^2)$, where k is the number of cells in the grid. Although better in practice, this was still prohibitively large for real world road networks.

Finally, in order to reduce the storage requirements even more, instead of link e, we considered the cell C containing this link. In other words, for every cell C we store a set $\mathcal{C}_e = \{(C_o, C_d)\}$, such that for every link e in cell C there exist

a pair of nodes src in C_o and dst in C_d, such that e is on the shortest path between src and dst. In this case the asymptotical worst case memory requirement reduced to $O(k^3)$ which was practical to implement.

3.3.3 Experiments

We tested our algorithm on the New South Wales road network with 200,000 nodes and 470,000 edges using different grid sizes. We randomly "blocked" roads and then executed the precomputation stage using the additional information stored during the initial precomputation. Since local, unimportant roads lie on significantly less shortest paths, we "blocked" the most important roads in our experiments. Results are summarized in the table below.

Grid size	Original precomputation time	New precomputation time	Original storage	Additional storage	Average update time
50 x 50	6 min	8 min	172 Mb	185 Mb	4 min
75 x 75	11 min	16 min	315 Mb	940 Mb	3 min
100 X 100	22 min	50 min	580 Mb	3.1 Gb	5 min
125 X 125	31 min	160 min	955 Mb	6.9 Gb	7.5 min

Table 3.6: Results of incremental updating of Grid-TNR for NSW network.

While those results are reasonably feasible, more careful examination showed us that majority of recomputed pairs of nodes didn't pass through the blocked link but only through the cell containing this link (which may contain other, not affected links). Therefore, there is still plenty room for improvement for more economical incremental updating. Considering the fact that many times, not only one, but several roads in a region can be affected, one such update should take care of all of them in one run, rather than running it for every edge separately.

3.4 CHAT

In this section we present a novel algorithm, we named CHAT (for **C**luster, **H**ierarchify **a**nd Hi**t**). It can be seen as instantiation of the more general TNR framework [Schultes, 2008]. CHAT was inspired by the Grid-TNR, but significantly outperforms it, addressing some of Grid-TNR weaknesses. We have observed that CHAT is completive with the latest state of the art algorithms [Abraham et al., 2011; Arz et al., 2013].

One of the drawbacks of Grid-TNR is that the algorithm does not exploit any topological information of the underlying network, like bottlenecks. Consider the example in Figure 3.20 below.

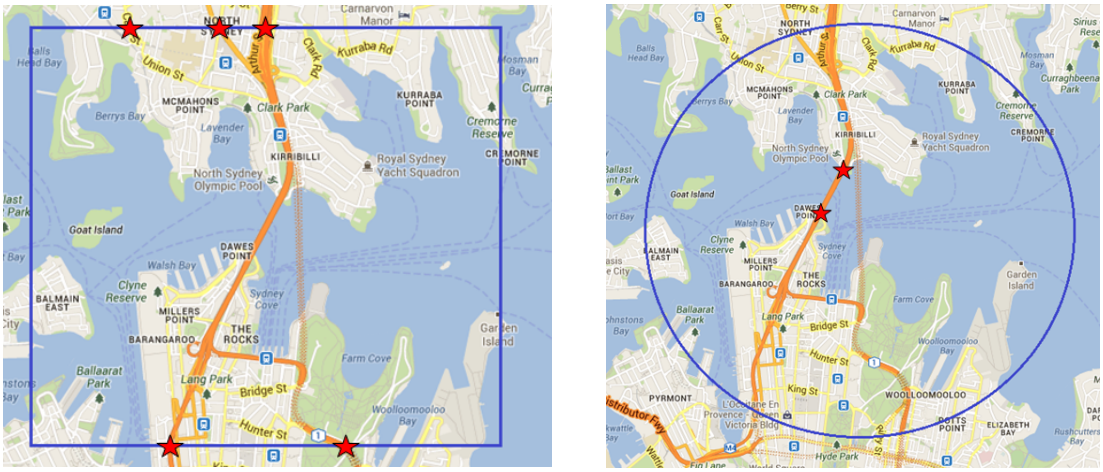

Figure 3.20: Five *transit* nodes identified by Grid-TNR vs two *access* nodes identified by CHAT that cover eventually the same "long" shortest paths.

Since Grid-TNR is identifying *transit* nodes strictly on the border of inner cells, in this example, we would identify five transit nodes. Intuitively, we can clearly see, that two nodes, strategically chosen as entranse/exit points into the

bridge, would be enough to cover all "long" shortest paths (longer than some predefined radius). Also, since the grid is artificial and rigid, there is no guarantee that a *transit* node is indeed an *important* node, e.g. entrance to a highway, bridge or a tunnel. In addition it is not clear, a priori, what grid size we should choose, since it directly affects the tradeoff between precomputation time, storage space and a final query time. Finally, when overlaying a grid, there is no guarantee as for the number of nodes in every cell. Some of the cells may contain very large number of nodes (in the city for example) and some cells can be almost empty (remote areas). We address all these issues.

Intuitively, when we drive "far away", at the beginning we use residential roads, at some stage we usually enter a highway network and eventually somewhere close to our destination we leave the highway network and use residential roads again. In addition we observe that many close nodes will have exactly the same set of access nodes. For example, it is most likely that most of the nodes in the same neighborhood will have exactly the same *access* nodes, when traveling "far away". In what follows we will explain how we successfully exploit those observations.

3.4.1 The Algorithm

For clarity, we will start with the basic, non-optimized version. The basic idea of our algorithm is for every node v to find a small set $\mathcal{A}_v$ of ***important***[1] *access* nodes, such that every "long" shortest path from (or to) v will necessarily pass through one of the nodes in $\mathcal{A}_v$. Then, for a query, given a source *src* and

[1]The idea of using *important* nodes to reduce the search space is also presented in [Schultes and Sanders, 2007]. However, here, the authors used *Highway Hierarchies* to define this set.

destination dst, we will be able to find the shortest path similar to Grid-TNR, simply by trying all possible combinations of subpaths: $src \rightsquigarrow A_{src}$, $A_{src} \rightsquigarrow A_{dst}$, $A_{dst} \rightsquigarrow dst$, where $A_{src} \in \mathcal{A}_{src}$ and $A_{dst} \in \mathcal{A}_{dst}$.

Given that every road has a known type[1] and a traveling speed associated with it, we exploit the inherent hierarchy of the road network. We introduce two levels of hierarchy - highways (usually roads with speed limit above 70km/h) and the rest, residential roads. Initially we identify all the nodes which have at least one endpoint as a highway, we call them *major* (a.k.a. important) nodes. In the pseudo code below, we call this step "*Hierarchify*".

Next, for every node v we want to know how far we can travel (no matter what direction we are going) without passing a major node. Assume that the furthest node we can reach is w and its distance from v is D_v. It means that for every node outside ball B_v of radius D_v with v in its center, we will have to pass via at least one major node (otherwise D_v wouldn't be the maximum distance). In order to find D_v, for every node v (which is not a *major* node), we grow a Dijkstra tree until every branch of the tree hits at least one major node. Theoretically, this can take $O(n \log n)$, practically this is performed much faster, since the actual nodes that we explored are only a very small fraction of n. $\mathcal{A}_v$ consists of major nodes of this tree, such that any node in $\mathcal{A}_v$ doesn't have any other major nodes as its parent. We do the same using incoming links, in order to find *incoming* access nodes when traveling to v. Notice, that $\mathcal{A}_v$ already complies with the desirable property that any shortest path that ends outside B_v will necessary pass via at least one node from $\mathcal{A}_v$. At this stage, we can already

[1] if type of the road is not explicitly given, we can implicitly deduce it using the average travel speed.

successfully apply our query routine using the $\mathcal{A}_v$ and are able to find the shortest path. Practically, the size of $\mathcal{A}_v$ can be prohibitively large (more than a hundred nodes), which significantly impairs the query time.

3.4.2 Reducing the number of access nodes

We describe how to reduce the size of $\mathcal{A}_v$ significantly. We notice that when traveling "far away" and passing via one of the *major* nodes we usually also pass via other *major* nodes. Observing this, bring us to the idea that essentially we are interested to hit all "long" shortest paths that start at v with as few *major* nodes as possible. This is equivalent to the *Minimal Hitting Set Problem* (see 2.7). Finding a minimal hitting set is a NP-hard problem, nevertheless good approximation algorithms exist [Cormen et al., 2001]. Note that we don't need to have the minimal hitting set in order to be able to find a shortest path; any hitting set will do. In our experiments, a greedy approximation algorithm 2.7 produced very good results. In summary, the set of access nodes of a node v is defined as a hitting set of major nodes along all shortest path between v and the border of B_v.

3.4.3 How far is "far away" ?

In Section 3.4.1 we have defined B_v, a ball with a center in v of radius D_v, such that in order to get to any node outside B_v we necessary have to pass via at least one node in $\mathcal{A}_v$. Intuitively we observe that the further we travel, the smaller the hitting set from the previous step is. Experimentally we learnt that defining "far away" radius,$R_v = 1.5 \cdot D_v$ yields the best compromise between preprocessing

time, number of access nodes (consequently required memory size) and the final query time. Choosing $R_v < 1.5 \cdot D_v$ will make precomputation somewhat faster, but will increase size of $\mathcal{A}_v$. Choosing $R_v > 1.5 \cdot D_v$ will insignificantly decrease size of $\mathcal{A}_v$, while making preprocessing longer.

Given a map $G = (V, E)$, in what follows we present a pseudocode for finding a set of *access* nodes.

$M \leftarrow$ `Hierarchify nodes` V
`For every node` $v \in V \setminus M$
    $D_v \leftarrow$ `Dijkstra Tree(`$v,\ M$`)`
    $\{P_i\} \leftarrow$ `Dijkstra(`$v,\ R_v$`)`
$\mathcal{A}_v \leftarrow$ `Hitting Set(`$\{P_i\}$`)`

Algorithm 3: Pseudo code for finding a set of access nodes. M is the set of *major* nodes and P_i set of shortest paths that starts at v and ends at radius R_v

Having successfully reduced the size of $\mathcal{A}_v$, we encountered another problem. For every node v, $\mathcal{A}_v$ are mostly disjoint, i.e. in most of the cases $\mathcal{A}_v \cap \mathcal{A}_u = \phi$. This results in a large total number of access nodes and consequently prohibitively large precomputed *access-to-access* table. In next section we describe how to efficiently resolve this.

3.4.4 Clustering

As we have mentioned before, we observe that when we start a journey from a neighborhood, it is most likely that many starting point in this area will have some *access* nodes in common. Therefore it looks natural to partition the network into clusters and deal with clusters rather than with every individual node separately.

There are many different clustering techniques. The two most intuitive are

KD-Tree and K-Means clustering. KD-Tree clustering partitions the map into rectangles, such that each rectangle contains the same number of nodes. K-Means, clusters the node, such that every node in a created cluster is closer to clusters' center of mass than to any other cluster center of mass. The advantage of KD-Tree is that it is fast and creates balanced cells, the drawback is that it may create large rectangles, which consequently increase the local search radius. K-Means heuristic handles this problem very well, but it is slow and in addition creates unbalanced clusters. In our implementation we combined two techniques. In the first step, we created KD-Tree clusters. Next, we refine this, by applying a single iteration of K-Means clustering, where the initial centers were chosen as centroid of clusters' convex hulls. Our experiments showed that this combination outperformed (in terms of preprocessing time, required storage space and final query time) each of those techniques applied separately. Below is the example of partitioning Sydney area into 64 clusters using the three clustering techniques

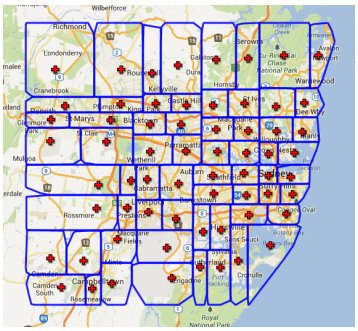

Figure 3.21: KD-Tree Clustering

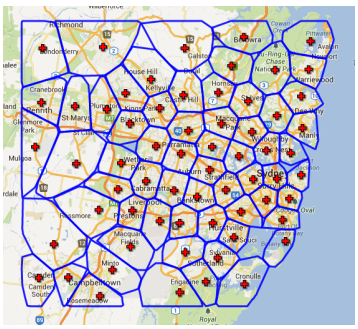

Figure 3.22: K-Means Clustering

Figure 3.23: KD-Tree refined with K-Means Clustering

From this point, the algorithm is essentially the same as was described in previous section. The only difference, that we perform the Algorithm 3 not for every individual node, but for only nodes that are on a border of the cluster

C, i.e. nodes on the convex hull of C. The "far away" radius R_C defined as $R_C = 1.5 \cdot D_C$, where D_C is the radius of the cluster C measured from the center of mass of its convex hull.

$\mathcal{C} \leftarrow$ `Cluster nodes` V
$M \leftarrow$ `Hierarchify nodes` V

`For every cluster` $C \in \mathcal{C}$
    $R_C \leftarrow$ `Dijkstra Tree(`$border(C),\ M$`)`
    $\{P_i\} \leftarrow$ `Dijkstra(`$border(C),\ 1.5 * R_C$`)`
$\mathcal{A}_C \leftarrow$ `Hitting Set(`$\{M_i\}$`)`

Algorithm 4: Pseudo code for finding a set of access nodes. $\mathcal{C}$ is a set of clusters, M is the set of *major* nodes and P_i is a set of shortest paths that starts at v and ends at radius R_C

Finally, similar to Grid-TNR, we are only left to precompute and store the distances between every node and its associated access nodes (in both directions) and between all the access node.

3.4.5 Query

The query is performed similar to the original TNR. For a query between src and dst which is global, i.e. $dist(src, dst) > max(R_{src}, R_{dst})$, we fetch *access* nodes associated with their clusters and find the path with minimal sum of the three subpaths: $src \rightsquigarrow A_{src}$, $A_{src} \rightsquigarrow A_{dst}$, $A_{dst} \rightsquigarrow dst$.

The test whether the query is global or local is also know as a locality filter. A locality filter is a function $\mathcal{L} : \mathcal{V} \times \mathcal{V} \rightarrow \{true, false\}$. In our case $\mathcal{L}(s, t) = true$ (i.e. the query is local) if $dist(src, dst) \leq max(R_{src}, R_{dst})$, otherwise the query is global.

Please notice in Figure 3.24 below how access nodes are located in "*strategically*"

important locations and not restricted to reside on somewhat random grid lines.

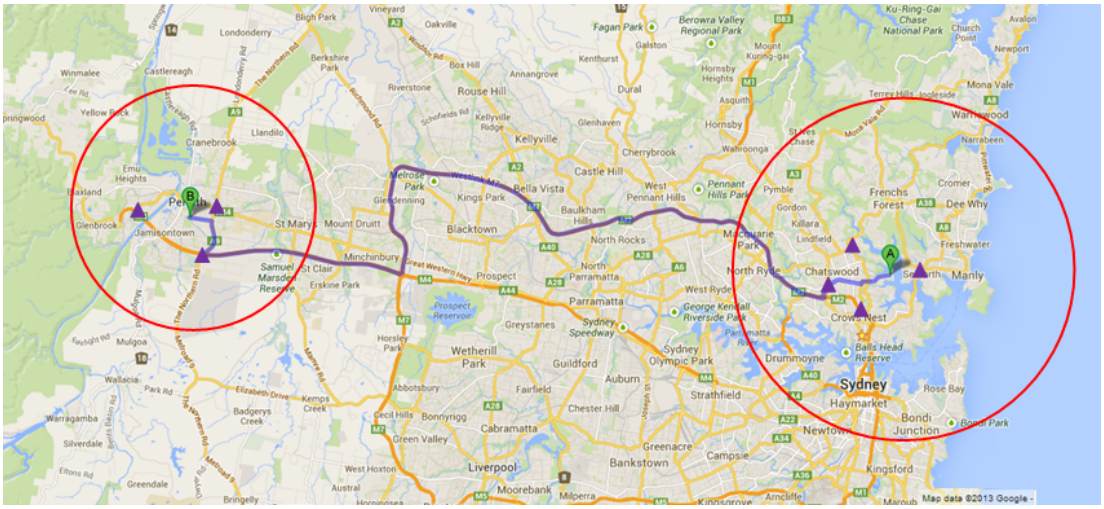

Figure 3.24: Example of CHAT query, where the triangles are access nodes.

For all other, local queries, as was suggested by [Bast et al., 2006] we can use any other efficient local search technique. According to [Lingkun Wu and Zhou, 2012] CH (Contraction Hierarchies) [R. Geisberger, 2012] is the preferable choice.

3.4.6 Proof of Correctness

Before proving the correctness of CHAT algorithm we will prove the following lemma, which may be not completely clear due to the non-trivial way we define access nodes.

Lemma 3.2. *Any global shortest path P between src and dst will always pass via A_{src}, access node of the scr and A_{dst}, access node of the dst, i.e. it will be of the form $P = src \rightsquigarrow A_{src} \rightsquigarrow A_{dst} \rightsquigarrow dst$.*

Proof. Let P be a global shortest path between src and dst. Let D_{src} be a local

radius of the *src*. Since P is global, at some stage P crosses B_{src}, the ball of radius D_{src} with *src* in the center. Lets denote this subpath as P_{src}. Clearly P_{src} is a shortest path by itself. On one hand, by definition, P_{src} contains at least one major node. On the other hand, we remember that the set of access nodes of *src* is defined as a *hitting set* of major nodes along all shortest path between *src* and the border of B_{src} (including P_{src}). Therefore by definition of the *hitting set* P_{src} necessarily contains at least one access node of the *src*. Exactly the same arguments hold for the *dst*. □

We are ready now to prove correctness of CHAT (which is similar to 3.2.6).

Theorem 3.3. *CHAT finds optimal queries.*

Proof. Let *src* and *dst* be source and destination nodes of interest. Assume that the query between *src* and *dst* is global (as defined in 3.4.5). In this case we execute the CHAT query procedure as in 3.4.5. Let P be a path with cost $c(P)$ returned by CHAT. The proof proceeds by contradiction. Suppose there exists another path $\hat{P}$ with cost $c(\hat{P})$, such that $c(\hat{P}) < c(P)$. Now, let $\hat{A}_1$ and $\hat{A}_2$ be access nodes of $\hat{P}$ of *src* and *dst* respectively. From Lemma 3.2 we are guaranteed that $\hat{P}$ passes via some $\hat{A}_1$ and $\hat{A}_2$. Let us denote $\hat{P}_1 = src \rightsquigarrow \hat{A}_1$, $\hat{P}_2 = \hat{A}_1 \rightsquigarrow \hat{A}_2$ and $\hat{P}_3 = \hat{A}_2 \rightsquigarrow dst$ subpaths of $\hat{P}$. The shortest path optimal substructure property [Cormen et al., 2001] (pp. 581-582) says that every subpath of a shortest path is a shortest path. Thus it follows that $\hat{P}_1$, $\hat{P}_2$ are $\hat{P}_3$ are all shortest paths. Similarly, let be $P_1 = src \rightsquigarrow A_1$, $P_2 = A_1 \rightsquigarrow A_2$ and $P_3 = A_2 \rightsquigarrow dst$ be subpaths of P. By assumption (at least) one of the following must be true: $c(P_1) > c(\hat{P}_1)$ or $c(P_2) > c(\hat{P}_2)$ or $c(P_2) > c(\hat{P}_2)$. But this contradicts the fact that Grid-TNR

query returns a minimal sum of subpaths from *src* to *dst*. For local queries, we are using CH [Geisberger et al., 2008], which is known to be correct. Q.E.D. □

3.4.7 Data storage

In this section we describe an efficient way to store the precomputed data. We will start with *outgoing* access nodes (where exactly the same we do for *incoming* access nodes). For every node v, we need to store array of distances from v to its all *outgoing* access nodes, say $\{A_1, A_2, ..., A_q\}$. For this we need to allocate an array of size $(q+1)$. A node v will point to the first element of this array, which will indicate the number of the access nodes, then we sequentially store all the distances, as in Figure 3.26

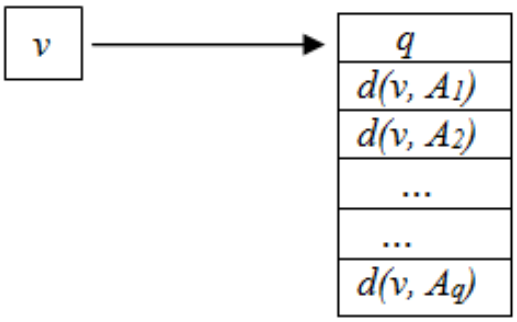

Figure 3.25: Storage of the shortest distances between a node and its access nodes

Assuming that nodes are integers represented in 4 bytes[1] , for the node v we in total we will require $4 + 4(q+1) = 4q + 8$ bytes. Exactly the same true for incoming access nodes. Assuming we have n nodes (that do have access nodes,

[1] 4 bytes is a typical size of an integer, however in general, it depends on the platform and the compiler.

see 3.2.8), we will require $2n(4q + 8) = 8n(q + 2)$ bytes to store distances from (to) every node to (from) its access nodes.

Now, lets take a look on the table that stores distances between all the pairs of access nodes. Since this is the most expensive and the most frequent operation during the query, this table should be implemented very efficiently. Generally, a hash table allows us to fetch an element in $O(1)$ of time complexity, assuming there are no collisions. For this, we first re-index all the access nodes from 0 to $(N - 1)$, where N is the total number of access nodes. A function $xN + y$, where x and y are indexes of two access nodes and is *one to one* and will guarantee no collisions between any two pairs of access nodes.

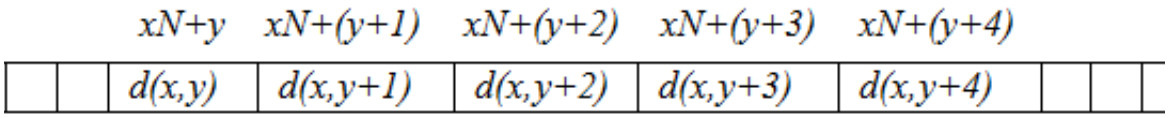

Figure 3.26: Storage of the shortest distances between all transit nodes

In order to store distances between all the pairs of access nodes we need $4N^2$ bytes, plus 4 bytes to store a pointer to the hash table. Summarizing, the total additional storage for CHAT is $8n(q + 2) + 4N^2 + 4$. Exactly the same analysis is also true for Grid-TNR. Thus, for a typical example when we have 10^6 nodes, with 10 access nodes per node on average and totaly 10^5 access nodes, we will require 9.7Gb. The Figure 3.27 below summarizing memory requirements as a function of total number of access nodes and average number of access nodes per node. Of course, for the undirected case, the storage requirements are halved.

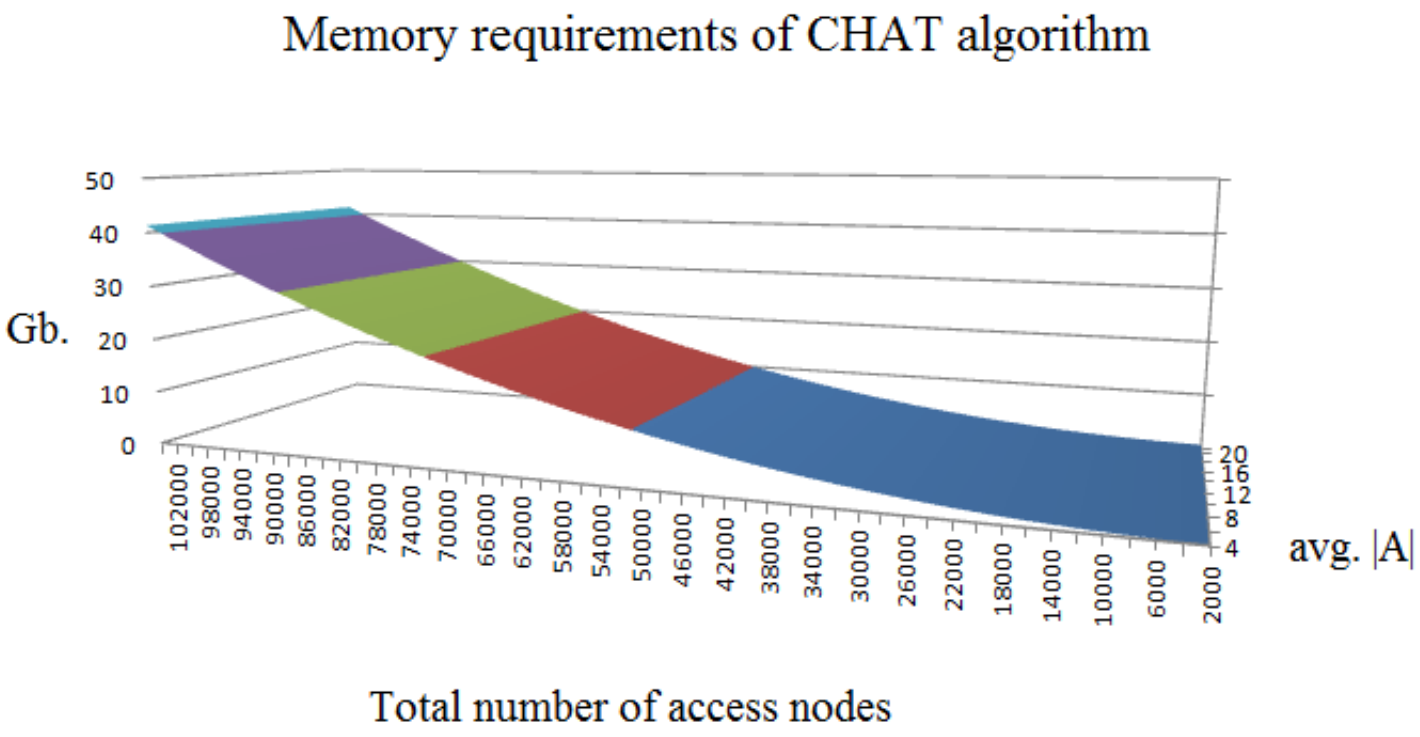

Figure 3.27: CHAT/TNR storage requirements

Space reduction

We can notice, that eventually we do not need to precompute the distances between all pair of access nodes, but only between all *outgoing* to all *incoming* access nodes (see 3.2.8). Assume we totally have $|N_{out}|$ *outgoing* access nodes indexed $\{x, x+1, x+2, x+3, ...\}$ and $|N_{in}|$ *incoming* access nodes indexed $\{y, y+1, y+2, y+3....\}$. For every *outgoing* access nodes x we store the distances only between x and all other *incoming* access nodes, as in Figure 3.28

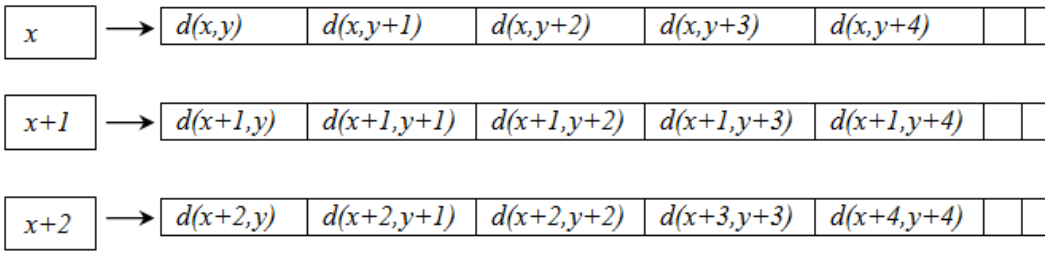

Figure 3.28: More efficient storage of distances between all access nodes.

In this case the space requirement becomes $4|N_{out}| + 4|N_{out}||N_{in}|$. Assuming that $|N_{out}| \approx |N_{in}| \approx \frac{N}{2}$, the total space requirement becomes $8n(q+2)+2N^2+2N$. The total memory requirements are plotted in the Figure 3.29 below. We can see up to factor of 4 space reduction by these optimisations.

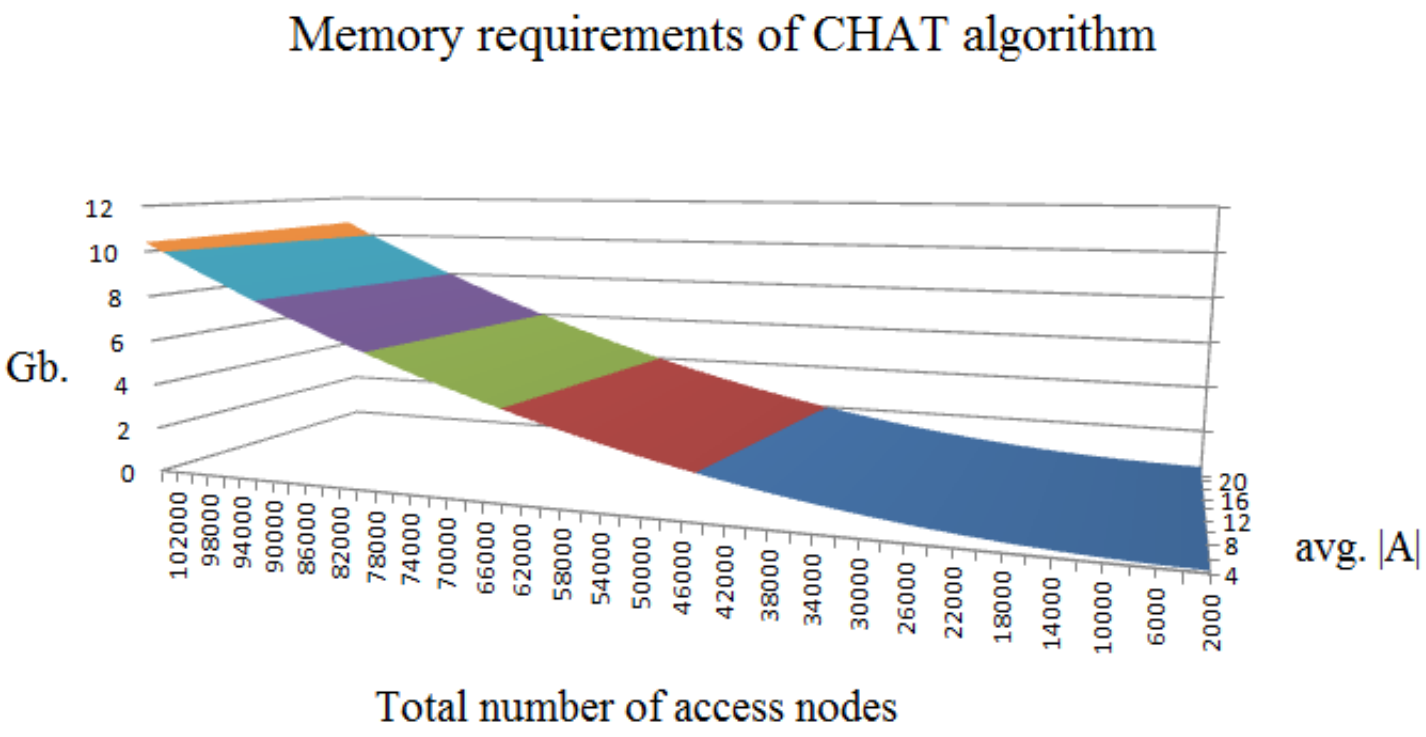

Figure 3.29: More efficient CHAT/TNR storage requirements

For some practical purposes, 2 bytes may be enough to store the fastest time. If time is measured in seconds, 2 bytes = 16 bits will allow us to store maximum time length of 2^{16} seconds, which is slightly more than 18 hours. For maps, where we know that the longest shortest path is faster than 18 hours (France or Germany for example), we can store time using only 2 bytes. In addition, for many practical purposes, precision of minutes is good enough. In this case we can store time in minutes rather than seconds and then the shortest paths of length up to 2^{16} minutes, which is more than 1092 hours can be expressed. This observation will reduce additional storage by almost factor of 2.

More compression can be achieved, by noticing, that in the vast majority of cases the distance from a node to its access node and the distance from an access node to a node is the same, i.e. $d(v, A) = d(A, v)$. In this case we can store it only once, thus significantly reducing the size of the *access-to-node* table. The idea of compressing edge weights and some other compression techniques are presented in [Sanders et al., 2008].

3.4.8 Experiments

We implemented our CHAT algorithm in Java and the online query in C++ compiled (with /O2 optimization) under Microsoft Visual Studio C++ 2012. Online queries were tested on a machine running Windows Server 2012 with 64Gb of DDR3-1333RAM and 6-core Xeon X5650 CPUs at 2.66GHz Intel. Similar to Grid-TNR (3.2.9) we tested our implementation on three continental size networks:

Australia	($\|V\| = 6.1 \times 10^6, \|E\| = 12.4 \times 10^6$) taken from OSM [OSM]
Western Europe	($\|V\| = 18 \times 10^6, \|E\| = 42 \times 10^6$) and
USA	($\|V\| = 24 \times 10^6, \|E\| = 58 \times 10^6$)

taken the from 9th DIMACS Implementation Challenge [Demetrescu et al., 2009]. We measure global query time by running 10^7 random queries (picked uniformly between all possible pairs). In addition, in order to verify correctness of CHAT implementation, extensive comparison with the basic Dijkstra algorithm was performed. All 10^7 tests returned the same value as Dijkstra's algorithm.

Results

In what follows we present CHAT's critical performance measures. We report total number $|\mathfrak{T}|$ of transit nodes, additional space requirements, average number $|A|$ of access nodes, percentage of global queries for distance metric (undirected version) and travel time metric (directed version).

	Australia (time)				Australia (dist)			
No. of Clusters	$\|\mathfrak{T}\|$	Db. Size	avg. $\|A\|$	% Global	$\|\mathfrak{T}\|$	Db. Size	avg. $\|A\|$	% Global
512	4282	1.1 Gb	6	93.8%	3699	0.5 Gb	8.1	91.2%
1024	7877	1.1 Gb	5.6	96.5%	6939	0.6 Gb	7.7	94.4%
2048	13 700	1.1 Gb	5.1	97.5%	12 084	0.6 Gb	7.1	95.4%
4096	22 754	1.4 Gb	4.7	98.2%	20 379	0.8 Gb	6.5	96.5%
8192	36 199	2.1 Gb	4.2	98.6%	33 311	1.5 Gb	6	97.2%
16384	54 510	3.6 Gb	3.8	98.9%	51 262	2.9 Gb	5.5	97.7%
32768	79 143	6.7 Gb	3.5	99.1%	76 481	5.9 Gb	5.1	98%

Table 3.7: Results of CHAT for Australia network.

	USA (time)				USA (dist)			
No. of Clusters	$\|\mathfrak{T}\|$	Db. Size	avg. $\|A\|$	% Global	$\|\mathfrak{T}\|$	Db. Size	avg. $\|A\|$	% Global
512	4422	1.7 Gb	9.5	97.4%	8096	2.9 Gb	17.7	96.1%
1024	8119	1.6 Gb	8.7	98.5%	14 825	2.9 Gb	16.5	97.4%
2048	13 802	1.6 Gb	7.9	98.9%	24 926	3.1 Gb	15.3	98%
4096	24 361	1.9 Gb	7.1	99.3%	41 787	4 Gb	14	98.4%
8192	39 822	2.7 Gb	6.4	99.4%	65 539	6.3 Gb	13.1	98.5%
16384	65 198	5.2 Gb	5.7	99.5%	99 902	11 Gb	12.1	98.7%
32768	101 423	10.9 Gb	5.3	99.6 %	143 988	21.7 Gb	11.4	98.8%

Table 3.8: Results of CHAT for USA network.

	Europe (time)				Europe (dist)			
No. of Clusters	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	% Global	$\lvert\mathfrak{T}\rvert$	Db. Size	avg. $\lvert A\rvert$	% Global
512	5551	1 Gb	6.8	96.3%	7023	1.8 Gb	15.4	87.8%
1024	9604	1 Gb	6.3	97.5%	12 495	1.8 Gb	14.5	91.9%
2048	15 926	1.1 Gb	5.8	98.2%	21 300	2 Gb	13.6	93.3%
4096	25 293	1.4 Gb	5.3	98.5%	35 297	2.7 Gb	12.8	94.8%
8192	37 837	2.1 Gb	4.9	98.7%	54 379	4.3 Gb	12	95.4%
16384	53 424	3.5 Gb	4.5	98.9%	80 144	7.5 Gb	11.4	96%
32768	125 554	15.8 Gb	4.7	99.6%	111 057	13.1 Gb	10.8	96.4%

Table 3.9: Results of CHAT for Europe network.

As we can observe, CHAT is less efficient for the distance metric. This is because of the nature of how we chose *major* nodes. It's easy to think about scenario where the "long" shortest path can avoid all the highways all together. This, in turn will increase the local search radius, which causes the large number of *false negative* global queries. Consider the example in a Figure 3.30 below.

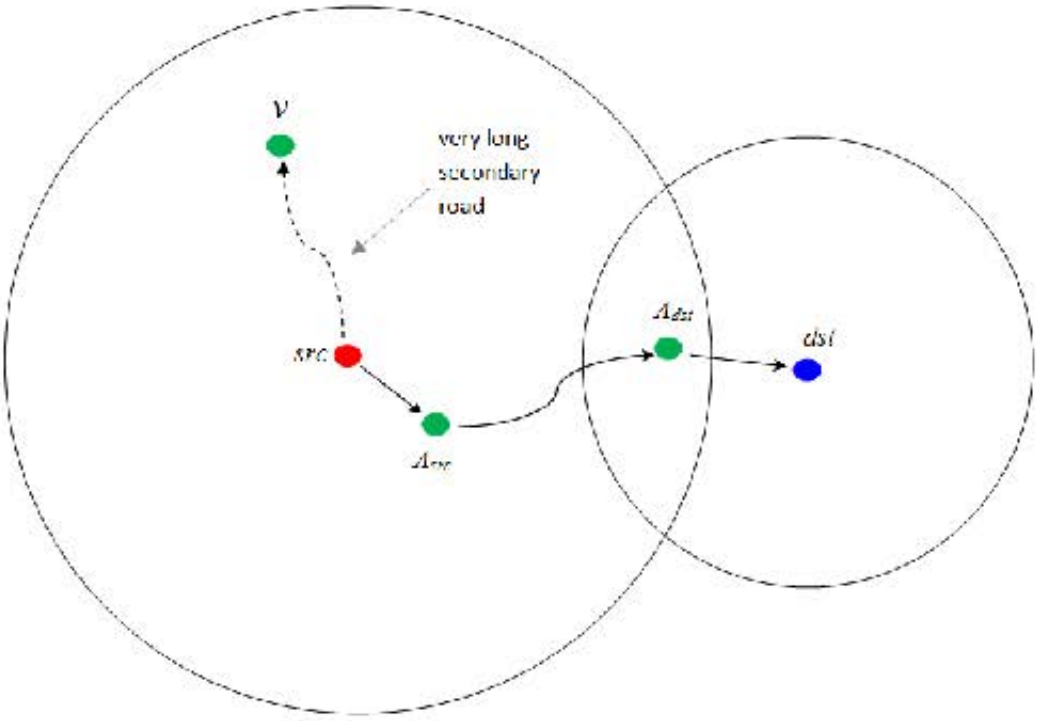

Figure 3.30: Example of potential global query being identified as a local.

The path $src \rightsquigarrow A_{src} \rightsquigarrow A_{dst} \rightsquigarrow dst$ will be identified as a local path, because we needed to travel a very long distance until we met v, a major node at the other side. It may be an interesting research direction, for the distance metric to choose *major* nodes using a different approach.

Given the clustering, CHAT's precomputation time is essentially the same as of Grid-TNR, with the addition of finding a hitting set of the major nodes. In practice, it is very fast, because we are dealing with relatively small sets. Usually $5 - 8$ iterations of the greedy frequency based algorithm is enough. In addition CHAT is highly parallelizable, which makes the precomputation fast when we have many cores. In addition, CHAT requires a much smaller number of clusters than the number of cells required by Grid-TNR to achieve a similar percentage

of global queries.

Global query times are summarized in the chart in Figure 3.31 below. Our global query times are comparable with the state-of-the-art HL algorithm Abraham et al. [2011], while being tested on a somewhat slower machine. On a similar machine, XEON 5680, which is 25% faster, we would expect additional performance improvement.

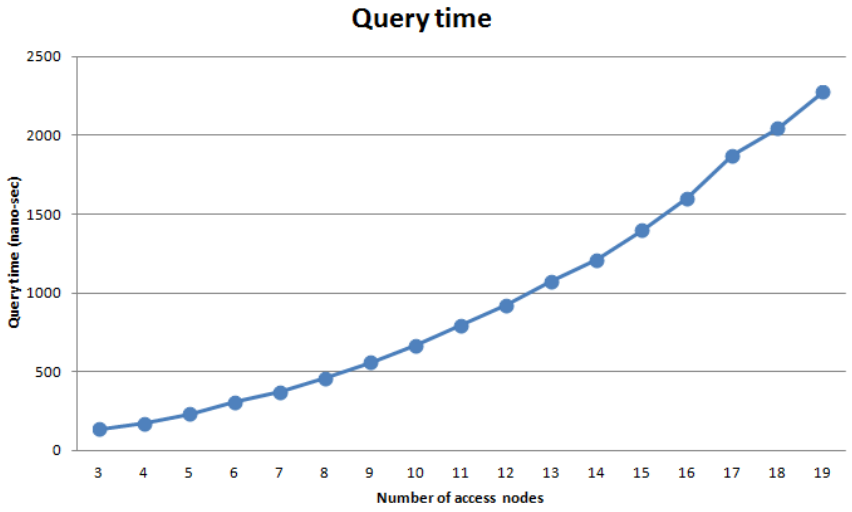

Figure 3.31: Query time as a function of number of access nodes

3.4.9 Discussion

We have presented a novel algorithm, CHAT, for very fast "long" shortest path queries. It uses local *Hitting Set* to identify a small set of access nodes among already provided major nodes. The idea of using a global *Hitting Set* to empirically find a lower bound on the set of transit nodes was used by Eisner and Funke [Eisner and Funke, 2012]. They showed that the number of transit nodes

in Grid-TNR [Bast et al., 2006] and HH-TNR algorithms [Sanders and Schultes, 2006a] was actually close to optimal. Though, authors did not recommend to use their approach for practical purposes as it was very time and space consuming. Another theoretical algorithm for finding an upper bound on the set of transit nodes was introduced by Abraham et al. [Abraham et al., 2010] by using global *Shortest Path Covers* (SPC), which is essentially a global *Hitting Set*. Computing such covers is NP-hard, but the authors suggest a greedy algorithm to achieve a logarithmic approximation factor in polynomial time. However, the authors suggested the algorithm is not practical, since the precomputation time will take several times and the unpractical theoretical bound.

When comparing CHAT with recent state-of-the-art CH-TNR [Arz et al., 2013] we notice two major differences in their approaches - transit nodes selection and locality filter. CH-TNR defines transit nodes as the k highest nodes from the CH data structure and the locality filter is the relatively expensive operation of intersection of two precomputed and stored search spaces. Clearly, CH-TNR fast preprocessing time (measured in minutes) stands out as a clear advantage over CHAT (whose preprocessing time is measured in hours). However, CH-TNR's additional storage is larger. While a query in both algorithms boils down to a few table lookups, the locality filter of CH-TNR's is more complex to calculate, which impairs its final query time. Both approaches are using CH as a fall back algorithm for local queries, so we will compare only speed of the global queries. For the European road network with time metrics, CH-TNR reports global query time $\sim 1.3\mu$s, whilst CHAT's global query time is between $200 - 300$ns for various choices of number of clusters, which is almost 5-6 times faster. Considering all types of queries, on average, CHAT's query time is still very competitive, see

table 3.10 below.

Comparing CHAT with another recent state-of-the-art algorithm *Hub Labeling* (HL) [Abraham et al., 2011] that has the fastest known queries, will show us a similar trade-off. While HL preprocessing is faster (under one hour), with the query times in a range of $250 - 560$ns, its additional storage space can be prohibitive for practical use (up to 18Gb). An improvement *Hub Label Compression* (HLC) [Delling et al., 2013], reduces storage space by an order of magnitude, but at the price of an order of magnitude slower query times. Another noticeable difference is that HL (and its variants) answers all types of queries, where CHAT needs to use a fall back algorithm for local queries. Below we present the table that summarized the additional storage and query times[1] of the discussed algorithms [Bast et al., 2014]. For the sake of completion, for the local queries we are using CH, and adding 0.4Gb. to count for CH data structure to CHAT storage requirements.

Algorithm	Source	Storage Gb.	Global Query (μs)	Avg. Query (μs)
CH	[Bast et al., 2014]	0.4	-	110
CH-TNR	[Bast et al., 2014]	2.5	1.2	1.25
HL	[Bast et al., 2014]	18.8	-	0.56
HL-∞	[Bast et al., 2014]	17.7	-	0.25
HLC	[Bast et al., 2014]	1.8	-	2.55
CHAT-2084		1.5	0.295	2.27
CHAT-4096		1.8	0.256	1.9
CHAT-8192		2.5	0.227	1.65
CHAT-16384		3.9	0.204	1.41
CHAT-32768		16.2	0.215	0.65

Table 3.10: Comparison of storage requirements and query times of various algorithms on Western Europe.

The measured preprocessing times for the above datasets varied between 1 and 12 hours. However, it won't be correct to directly compare CHAT preprocessing times with the other algorithms, for several reasons. First of all, CHAT was implemented in Java (unlike other algorithms that were implemented in C++). Secondly our precomputation were performed on shared resources (a.k.a. cluster) with significantly slower machines (Intel Xeon E5405 @ 2.00GHz vs. Intel Xeon X5680 @ 3.33GHz) than reported state-of-the-art algorithms. When run on a dedicated machine and normalized to the faster, Xeon X5680, we believe CHAT preprocessing times should be at least 2 times faster.

In conclusion, we see that when dealing with long range queries, as for example, in a case of national size fleet logistics transportation problems, CHAT has a clear advantage over other approaches. On average, it is also very competitive with

[1]The times of CH, CH-TNR, HL, HL-∞ and HLC are reported using Intel X5680 3.33Ghz CPU. Our queries were performed on somewhat slower Intel X5650 @2.67Ghz machine

recent state-of-the-art algorithms.

Another clear advantage of CHAT, while not quantitative, but still non insignificant, is that it is very intuitive and relatively simple to implement.

3.5 Conclusions and Future Work

We have extended Grid-TNR to directed graphs. Then we showed how to reduce the number of access nodes and eventually obtained fast possible queries using Grid framework. We formally prove that the vertex separating set between border nodes of the cell and its outer square is the minimum set of access nodes. In addition, we have presented two new algorithms: the first for Incremental Updating Grid-TNR database and CHAT - a new instantiation of TNR framework. While the results are very promising, there are interesting research questions and further improvements that can be pursued. Below we list some directions for potential research.

Incremental TNR

The space reduction from $O(mn)$ to $O(k^3)$ comes with the cost of additional, somewhat redundant, recomputation work that has to be done during the update stage. Cells (C_o, C_d) associated with C may also often contain pairs of nodes that do not involve the blocked road e in the shortest path between them. It is an open problem how we can further reduce these redundant computations.

In the presented algorithm we considered a scenario when the link weight increased. This is the most common scenario in road networks, when dealing with travel times, which are usually get longer due to morning congestion for example.

It would also be interesting to consider how to update efficiently the tables when travel times of a link decrease.

Combining real time live traffic updates into TNR framework is another interesting and challenging problem. It will require faster decision making and may even require completely different approach.

CHAT

An immediate speed-up for CHAT queries is to boost it with Arc Flags (AF) [Möhring et al., 2007]. As it was shown in [Bauer et al., 2010b] CH-TNR queries were almost two times faster when combined with AF. It would be interesting to see how much AF can speed-up CHAT queries.

The second direction is to improve the locality filter and eliminate "long" local queries. For examples, assume there are two locations on opposite sides of a bay (quite common in Sydney). While they are geographically close to each other, it may require quite a long journey to get from one side of the bay to another. Clearly it can be more efficient treat this as a global query.

The example in Figure 3.30 may happen not only when using the distance metric, but also when using a time metric, in particular in remote areas, where we need to travel relatively far until we reach another highway. It seems like an interesting and challenging problem, to find for every node its minimum local search radius without compromising database size and query time. This will reduce the number of local queries (which are much slower than global), will make them faster (because of the reduced search space) and will increase number of global queries (that are very fast).

Finally, it would be interesting to see the effect of different clustering techniques.

Chapter 4

Public Transportation Networks

In this Chapter we present an algorithm to find optimal paths in a public transportation network. Nowadays there are numerous systems that provide users with transit [1] information, e.g. NSW TranportInfo [NSW], Google Transit [Google]. Such systems are often available as a web service or smart phone application. The application asks a user to input an origin, destination and expected departure or arrival time and provides the user with recommended travel routes. Such systems are useful in encouraging people to switch from their private cars to use public transport services, thus reducing congestion, CO_2 emission and providing travelers with a better experience.

Our algorithm extends the Transit algorithm [Bast et al., 2006, 2007] using a novel time expanded graph of a multi-modal public network. As we've discussed in 3.1 the original Grid-TNR algorithm is one of the best methods for finding shortest paths in very large road networks, but was previously limited to a single mode of transport and static and undirected graphs. The nature of public network is

[1] the textual similarity between *transit networks* (a.k.a. public transportation networks) and *transit node routing* is accidental

very different from road network in many ways, e.g., links are directional, time dependent, etc. In addition, the number of nodes is very large, especially if, as is the usual case, we deal with time dependency by constructing a time expanded graph. In practice, simply applying Grid-TNR to a time expanded graph does not scale. To deal with this problem, we will apply it on a two-layers model of the public transportation network. We start with a single objective problem and show how to extend it to multi-objective criteria, such as travel time, tickets cost and hassle of interchanges to accommodate various user preferences.

4.1 Related Work

Modeling

The information of public transport availability is usually provided in a timetable, where each service in the timetable contains a list of stations with its arrival and departure times.

Station	Arrival	Departure
A		8:00
B	8:05	8:06
C	8:10	8:11
D	8:20	8:21
...	...	...
...	...	...
Z	9:00	

Table 4.1: Example of timetable information for a service that travels from A to Z.

The basic idea is to convert those timetables into a graph on which we can

then compute the shortest paths. There are two main ways to model the problem, specifically *time expanded* [Müller-Hannemann and Weihe, 2001; Schulz et al., 2000] and *time dependent* [Brodal and Jacob, 2004; Nachtigall, 1995; Orda and Rom, 1990] models.

In the *time dependent* model the node set of the graph is the set of all the stations and there is a link between two nodes, if and only if there is at least one service between two stations. The cost of each link is time dependent. In order to model realistic transfer times, for every station node, for each route that passes via this station, we introduce an auxiliary *route node*. The route nodes are then connected to their respective station nodes, with the cost reflecting the transfer time.

In the *time expanded* model each node of the graph correspond to an event rather than a physical station. In a simple model, there are two types of events: arrival and departure. In order to model realistic transfer times, we introduce a new *transfer node* which is connected to a *departure node* with a link which cost reflects the transfer time. The main difference between the models are the link weights, which are functions in the first instance and are constant in the later. For an exhaustive description of the models and existing techniques we address the readers to [Müller-Hannemann et al., 2007; Pyrga et al., 2004].

In [Pyrga et al., 2004] the authors experimentally compared those two models on several real-world data sets using variations of Dijkstra. The experiments revealed that *time-dependant* model performed better on a simplified version of the earliest arrival problem, while *time-expanded* model is more suitable to model more complex, realistic scenarios.

There are numerous work on trying to apply techniques from road networks

onto public transportation networks [Delling et al., 2009b; Müller-Hannemann et al., 2007]. Below we review the most recent results.

Car or Public Transport ?

While in recent years many algorithms have been developed that use precomputed information to speed-up shortest path queries in a road network (see 3.1), less progress has been made for public transportation networks. In [Bast, 2009], the author discusses why finding shortest paths in public transportation networks is not as straight forward as in road networks. There are several issues that arise in public networks, which are not encountered in road networks. For example, the start and end of the journey are usually geographical locations (e.g. home or work), and we need to walk to some nearby station first. It is unclear, a priori, what station should we start our journey from, so here we have a set of source and target stations. Other issues we need to consider for example, are *walking* between different services, *transfer time safety buffers*, *tickets cost*, *operating days*, etc.

Pareto Search

In [Müller-Hannemann and Schnee, 2007], the authors presented an efficient algorithm to provide attractive alternatives for users of public railroad systems using *time-expanded* model. They considered three criteria: *travel time*, *fare* and *number of train changes*. Essentially the proposed algorithm is a "Pareto version" of Dijkstra's algorithm using multi-dimensional labels. In the previous work [Müller-Hannemann and Weihe, 2001] the authors showed that in practice the number

of Pareto optimal paths (for the German railroad network) is a relatively small constant, though in theory this number could be much larger.

In order to speed up queries, they used goal directed search (similar to A*), using a *station graph* to calculate a lower bound to the destination. In addition, the author noticed that while there are some solutions that are theoretically sub-optimal (and therefore would be omitted by the algorithm), they are still attractive from a practical point of view. In order to "pick up" those solutions they defined a notion of *relaxed Pareto dominance*. This allowed to find and return sub-optimal, yet reasonably attractive solutions. To compare their results with the current Deutsche Bahn System, the authors introduced a linear utility function to reflect preferences of three different types of travelers: *businessman* (who is interested in the fastest journey), *handicapped person* (interested in the smallest number of changes), *student* (interested in the cheapest journey).

In [Disser et al., 2008] the authors considered a similar problem, i.e. finding all Pareto optimal paths, but this time using a *time dependent* model. They showed that this approach is very competitive with the previous, *time-expanded* one. Still the authors suggested that in order to make this algorithm practical its performance needs to be improved.

Accelerating is Harder Than Expected

In [Berger et al., 2009] the authors investigated the effect of applying two well known speed-up techniques **Arc Flags** and **SHARC** (see 3.1). They describe how those techniques should be adopted to work well with public transportation networks. Eventually, they report only marginal speed-ups (compared to the ob-

served speedups in the road networks) and come to a conclusion that the classical extensions of arc-flags and contraction does not work well.

Access Node Routing

In [Delling et al., 2009a] D. Delling et al. tackled a multi-model problem when in addition to a public transport, travelers are also allowed to use a car at the beginning or at the end of their journey. The authors introduced an *Access Node Routing* (ANR), by separating road and public transport networks and treating each of them separately. D. Delling et al. adopted some ideas from Transit node Routing (TNR) in the sense that access nodes into the public transport network can be seen as transit nodes of the combined (road + public transport) networks. The authors reports preprocessing time in range of if several hours and relatively fast query times in range of 2.3-5.8 ms.

Contraction Hierarchies (CH)

In [Geisberger, 2009] CH was applied to timetable networks. As it was impractical to apply CH directly to the time-dependent model, the author developed a new *station graph* model. For the simplified version, without transfer times, it is exactly as the *time dependent* model. For a more realistic scenario that considers transfer times, the main difference is that the *station graph* still keeps one node per station and does not have parallel edges. The algorithm was tested on real-world data of European railroads. The author reported preprocessing time of a few minutes with query times of half a millisecond.

Google Transit

Until today, this is considered to be a state-of the-art algorithm by H. Bast et al. [Bast et al., 2010] and is currently incorporated in Google Maps (for public transportation networks). They report a time of 10ms for station-to-station query for a North America public transportation network consisting of 338K stations and more than 110M events. The idea is to precompute and store *transfer patterns*. Then the query is performed on a much smaller *query graph* which is constructed from previously precomputed patterns. In order to speed-up precomputation, at the cost of slightly compromising optimality, the authors suggest to perform the precomputation only for the *hub* stations. *Hub* stations are heuristically identified as stations that are on the largest number of shortest paths. Besides being relatively complicated, the main drawback of their algorithm is the large computational resources required for precomputation. The authors report requirements of 20-40 (CPU core) hours per 1 million of nodes. For the Swiss transit network, which is comparable in size to the Sydney network, the reported precomputation time was between $560 - 635$ hours.

User-Constrained Routing (UCCH)

In [Dibbelt et al., 2012] the authors presented a fast multi-modal speedup technique that can handle different multi-modal user preferences at *query time* (and not during the preprocessing). Essentially the authors suggested to merge two networks: road (time independent) and public transport (time dependent) and apply Contraction Hierarchy on the combined graph. Practically, contraction could only be only applied on the road part of the combined network (contract-

ing public transportation led to prohibitively large number of shortcuts). User preferences over the modes of transportation are represented by automata, which is used to constrain Dijkstra search during the query time. For some instances, the authors reported speed-up of over 45000 over multi-modal Dijkstra.

Round-Based Routing (RAPTOR)

In [Daniel Delling and Werneck, 2012], the authors presented a novel algorithm for computing all Pareto-optimal journeys - minimizing the arrival time and the number of transfers. Unlike previous approaches, RAPTOR is not Dijkstra based. It is fully dynamic and does not rely on any preprocessing. Essentially the algorithm falls into the category of dynamic programming. It operates in rounds, where at each round, every route is traversed at least once. Round k computes the fastest way of getting to every other stop with at most $k-1$ transfers. RAPTOR can be easily parallelized and extended to deal with more criteria. While it works very well on relatively small (city size) instances it is still not clear how well it will perform on continental sized networks.

Fuzzy Domination

In [Delling et al., 2012] the authors present an algorithm for finding multi-modal journeys, including unrestricted walking, driving, cycling, and schedule-based public transportation. Initially the algorithm computes the full Pareto set (using adjusted multimodal multicriteria RAPTOR) and then scores all the solutions using techniques from fuzzy logic. This step eliminates less relevant journeys and leaves only the most significant ones. In order to deal with unrestricted walking,

the authors use contraction [R. Geisberger, 2012].

Result Diversity

Recently, in [Bast et al., 2013] proposed a new way *Types aNd Thresholds (TNT)* to present to the traveler a small, still representative and meaningful subset of Pareto optimal results. The TNT concept comprise of several steps: discretization of the Pareto optimal results, filtering unreasonable results (considering the relative duration of each mode) and finally applying threshold for relative durations of each mode. The approach was evaluated on the New York transit network and while the basic algorithm yields infeasible query times, using an (almost optimal) heuristic allowed to reduce the query time to $1sec$.

4.2 Contribution

The original Grid-TNR algorithm [Bast et al., 2006, 2007] remains one of the best methods for finding shortest paths in very large road networks, but is limited to a single mode of transport and static and undirected graphs. One of the advantages of the TNR is that the final query time is completely independent of the path length and only depends on the number of the access nodes of the origin and destination. The public transportation network, somewhat resembles the structure of road network, therefore it looked natural to extend and adjust the Grid-TNR to public transportation network. In this chapter we present a novel time expanded graph of the multi-modal public network and successfully extend Grid-TNR to work on this graph. We elaborate on those results in the following sections. Part of the work in this chapter was presented in [Antsfeld

and Walsh, 2012a,b].

4.3 Modeling

In our new model, we eliminate *transfer nodes* and all the links from *transfer nodes* to *departure nodes*. Instead we connect *arrival* and *departure* nodes directly. The model consist of two layers: *station graph* and *events graph*. The *events graph* nodes are arrival and departure events of a station and are interconnected by four types of links: *departure links*, *continue links*, *changing links*, *waiting links*. The *station graph* nodes are the stations and it has two types of links *station links* and *walking links*. In our experiments, we connect two stations with a *walking link*, if they are within 10 minutes of walking distance from each other. The described graph is illustrated below.

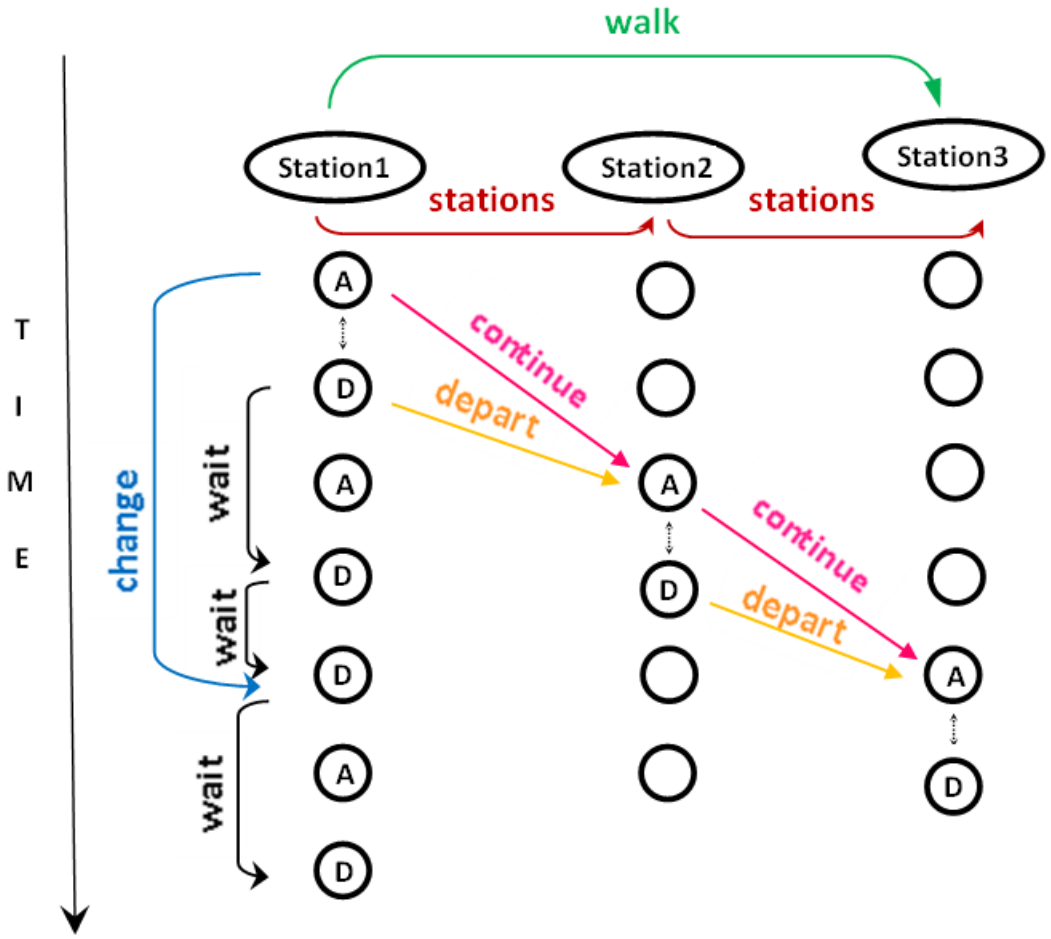

Figure 4.1: The two layered, time expanded graph with three stations.

For the Sydney public transport network containing 9.3×10^3 stations and 4×10^6 events, comparing to the traditional *expanded model* the space reduces by 20%. In addition to the storage saving this modification as we will see, will also speed up our precomputation time.

4.4 Grid-TNR

Assume for a moment that our world is a public transportation network, i.e. we always start and finish our journey at *event nodes*. We start with solving a single objective problem, e.g. we are interested to find the fastest way to get from station A to station B starting our journey at time t. Being consistent with [Bast et al., 2010] we denote this problem as $A@t \rightarrow B$. This problem is equivalent to finding an earliest arrival time, given the departure time. In reality the user may be interested in finding a route with the latest departure time, given an arrival time, which we will denote as $A \rightarrow B@t$. This query can be answered efficiently by simply applying exactly the same algorithm but using backward Dijkstra. Similarly to the original Grid-TNR algorithm [Bast et al., 2006] our algorithm also consists of two phases - precomputation and query.

4.4.1 Precomputation

In order to identify *transit nodes* we exploit the fact that the public transport stations have geographical coordinates associated with them. It allows us to overlay the network with a rectangular grid as we did for a road network (see Section 3.2).

Identifying Transit Nodes

At first stage of this phase, we identify a set of *transit nodes*. Since the time expanded graph is too large, both precomputation time and storage space will be prohibitive. In order to overcome this issue we define transit nodes to be subset of stations rather than events and determine them on the *stations graph* layer. This

approach very successfully solves the scalability issue. Since the *station graph* is relatively small (10K-40K nodes), this is done relatively fast (and of course can be always executed in parallel).

Computation and Storage of Distances

The next stage is performed on the *events graph* layer of our network. We precompute the following shortest routes and store them in three tables:
(i) *node-to-transit*: for every station node S, from every departure event of S to every transit station node of S,
(ii) *transit-to-transit*: from every departure/arrival event of every *transit* station, to every other transit station node and
(iii) *transit-to-node*: for every station node S, from every departure/arrival event of its associated transit station node to S.

4.5 Grid-TNR with Hubs

In what follows we'll describe a significant improvement of the previous precomputation stage. We will show how we can significantly reduce precomputation (time and storage) without loosing optimality of the final queries. Consider an example in Figure 4.2 below.

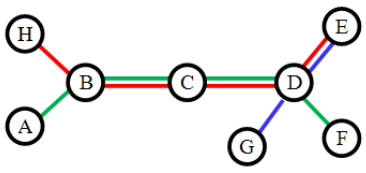

Figure 4.2: Example of hub and non-hub nodes

In the example there are three bus (or e.g. train) services: $A \to B \to C \to D \to F$, $H \to B \to C \to D \to E$ and $G \to D \to E$. We want to travel from A to E. Clearly, at some point we will have to change a service and it can be either at B, C or D. Now, there is no particular reason for us to change a service at C. On the other hand, potentially we may change a line at B or D. Therefore, we will identify B and D as *hub stations* and will refer to C as a *non hub station*. Intuitively, a *hub station*[1] is a station where we could potentially change a service, where all others *non hub stations* have no particular reason to be used for transfers. We observed that in reality (for Sydney and NSW public transportation networks), only a relatively small portion of all stations, $15\% - 20\%$, are hubs. More formally, let $G_S = (V_S, E_S)$ to be a station graph, as described in Section 4.3. Denote S_i be a set of all services that depart from station $s_i \in V_S$. Then set of hub nodes is defined as

$$H = \{s_i \in V_S \mid \exists s_j.\ s.t.\ (s_i, s_j) \in E_S \ \wedge \ S_j \subset S_i\} \bigcup$$

$$\{s_j \in V_S \mid \exists s_i.\ s.t.\ (s_i, s_j) \in E_S \ \wedge \ S_i \subset S_j\} \bigcup$$

$$\{s_i, s_j \in V_S \mid (s_i, s_j) \in E_S \ \wedge \ S_j \neq S_i\}$$

Pseudo code for identifying hub-nodes

Using the notations from previous Section, we present a pseudo code for determining *hub nodes*.

We notice that if we know the optimal route between any hub station to any

[1] Our hub stations are different from the hub stations in [Bast et al., 2010], see 4.1. They are likely to be the same, but may be different

```
for(s_i ∈ V_S)
  for((s_i, s_j) ∈ E)
     if (S_j ⊂ S_i)
          H := H ∖ s_j
          H := s_i
     else if (S_i ⊂ S_j)
          H := H ∖ s_i
          H := s_j
     else if (S_i ≠ S_j )
           H := s_i
           H := s_j
```

Algorithm 5: Pseudo code for identifying hub-nodes

other hub station, then a shortest route between any two nodes (not necessarily hubs) can be found immediately as described in the next Section.

4.5.1 Extracting the optimal path

We make an important observation that for every two "far away" non hub stations A and B, the following schema holds.

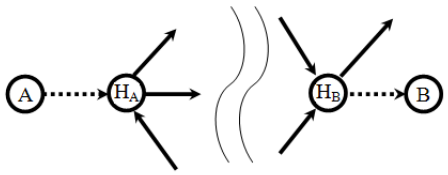

Figure 4.3: Example of query between two non hub stations

Generally, if a station s_i is not a hub node, there is only one succeeding station s_j, such that there is a service between s_i to s_j, i.e. $(s_i, s_j) \in E_S$. Otherwise s_i would be a hub node itself. Similar holds for S_j, etc... Therefore, referring to Figure 4.3, there is only one direct way to follow from any non hub node A to its first hub node H_A and from H_B to B. In other words, any service that departs

from A will certainly arrive to H_A. Similarly any service that arrives to B will certainly depart from H_B (because otherwise B would be a hub as well). It brings us to the following idea. If we only knew the shortest path from any hub node to any other hub node, it would give us very fast, simple and intuitive algorithm for finding shortest path between any two stations.

Given a query between A and B:

(i) Start from A and follow to it's first (outgoing) hub node H_A

(ii) Traverse backwards from B find it's first (incoming) hub node H_B.

(iii) Fetch from the precomputed database the optimal route $H_A \rightsquigarrow H_B$.

(iv) Combine all three segments together to obtain an optimal path .

Since eventually public transportation networks are inherently time dependent, we are interested in a query $A@t \rightarrow B$. Adding a time dimension to the algorithm above is relatively simple as well:

(i) Start with first service that departs at time $t_1 > t$ form A and follow until the first hub node H_A arriving there at time t_2.

(ii) Fetch from the precomputed database the optimal route $H_A@t_2 \rightsquigarrow H_B$

(ii) Continue to B with a direct service that departs from H_B at time $t_3 > t_2$

(iv) Combine all three segments together to obtain an optimal path .

Although the obtained path, in theory will be the fastest, in practice it may not be very convenient and involve unnecessary waiting (for example if services departing from H_A are infrequent) or unnecessary change between services at the hub nodes. In order to address this we do some "after-analysis" of the obtained route: if in the obtained route there is a relatively long waiting (say more than 10 mins) at H_A, we check if we can leave A later and still eventually to arrive at

the same time. We will discuss in a Section 4.8 in more details how to apply the obtained results in the real world.

From Section 4.5.1 we know that it is enough to know the optimal route only between pair of hubs in order to easily reconstruct the optimal route between any two nodes. Therefore our aim will be to efficiently answer the queries between any two hubs. For this we will use the modified Grid-TNR algorithm.

4.5.2 Precomputation

The stage of ***identifying transit nodes*** is performed exactly as in 4.4.1. Note, it wouldn't be correct simply to disregard non-hub nodes now. This is because in order to get from one hub to another, we still may need to change a service at a non hub node, therefore we can not disregard them at this stage.

The ***computation and storage of distances*** stage is performed similarly as in 4.4.1 with the only major difference, that now we simply disregard all non-hub nodes and precompute tables only for hub nodes. More specifically, we precompute the following shortest routes and store them in three tables:

(i) *hub-to-transit*: for every hub station H, from every departure event of H to every transit station node of H

(ii) *transit-to-transit*: from every departure event of transit station node to every other transit station node and

(iii) *transit-to-hub*: for every hub station H, from every departure event of its associated transit station node to H.

4.5.3 Query (time only)

Given a global query[1] $A@t \rightarrow B$, between two hub nodes A and B, we find the fastest journey time similarly as in the original Grid-TNR as follows. We fetch the transit station nodes of A and B, T_A and T_B accordingly. Let $\tau_A \in T_A$ and $\tau_B \in T_B$ be transit nodes of A and B. For every τ_A we fetch $c_1 = cost(A@t, \tau_A)$. Let's assume that this route arrived to τ_A at time t_1. Then we fetch $c_2 = cost(\tau_A@t_1, \tau_B)$. Finally, assuming that the fastest route from τ_A at time t_1 arrived to τ_B at time t_2, we fetch $c_3 = cost(\tau_B@t_2, B)$. Eventually the total travel time will be $c_1 + c_2 + c_3$. Iterating over all $\tau_A \in T_A$ the cost of the fastest route will be the one that yields minimal $c_1 + c_2 + c_3$. In case $A@t \rightarrow B$ is a local query, we just apply any efficient search algorithm, A* for example.

4.5.4 Query (itinerary)

Unlike road networks, where for many practical applications returning the time (or distance) is sufficient, in public transport networks users will be usually interested in concrete route directions. More precisely, besides knowing on which service to board at the start of their journey, we need to provide the whole itinerary, i.e. instructions where they should change services. Note, that unlike in road networks, we don't have to provide all intermediate stops of the journey, but only the stations where transfers between services occur. Luckily, the vast majority of our journeys consist of a very small number of transfers, 2-3 (4 and more on very rare occasions). Observing this, during precomputation phase for every precomputed pair we store the instructions where to change services (will

[1] global query is defined exactly as for Grid-TNR in Section 3.2.3

require additional 4 bytes of storage for every change). Alternatively we can store only the first change (similarly to storing the first link in the road networks) and reconstruct the whole journey by iteratively applying the *Query* from the next transfer station. Again the number of iterations should be very small.

4.5.5 Local queries

Similar to Grid-TNR we still need to address non-global, a.k.a. local queries. Luckily, many of the local queries are *direct connections*, i.e. do not require transfers and can be efficiently obtained as in [Bast et al., 2010]. The authors reported query time of $10\mu sec$. For all other queries we will have to fall back to some other method, A* for example, which should be relatively fast in practise because of the proximity between *src* and *dst*.

4.6 Practical Speed-up Techniques

One significant improvement was achieved by noticing that there is no need to precompute and store distances from (or to) every node in the network. Without loss of optimality we can safely ignore stations that are visited only by one service. Given a query where source or destination one of this nodes we can just simply follow the only service from this node until we encounter the first station node that is visited by more than one service.We observed about a 40% reduction in the number of nodes for which we should perform a precomputation stage 4.4.1. Another significant speed up was achieved by noticing that the same service running, say 10 minutes later, will eventually take us exactly to the same stations just 10 minutes later. Since there is no point for the user to wait on the sta-

tion for exactly the same service which will arrive 10 minutes later, during the precomputation stage, when running Dijkstra this whole branch can be pruned. Consider the example in Figure 4.4 below. Assume we have two identical services that operate with say 10 minutes difference. During the precomputation stage, when performing Dijkstra from the first departure node we can safely prune the whole branch that starts at the later departure node of exactly the same service.

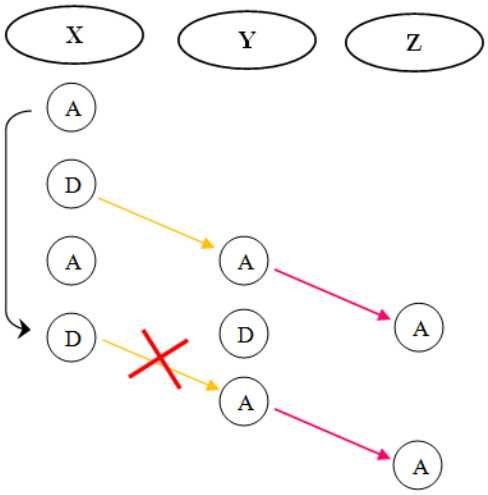

Figure 4.4: Example of pruning identical service.

The branches that can be pruned are identified and marked during the initial creation of the public transport network. This gave us about 20% speed up in the precomputation time.

Similarly, we notice that a service from station A to a station B, say at 9am and 9:30am has actually exactly the same pattern[1]. On the other hand traveling from A to B, say at 6pm may have a different time pattern. In order to capture this, we group similar services during the day according to their time

[1]exactly the same sequence of stops, but only with shifted time.

pattern. Luckily the number of these patterns is not large, for example it may take for some service 20minutes at night, 30minutes in the morning, 25minutes during the day and 35minutes in the evening. Essentially this is the same service, but it may be affected and take different time due to city traffic patterns. Then we precompute the tables only for a single representative service of every pattern. A more simplistic approach would be to divide the day into segments (e.g. 6am-9am, 9am-12pm, 12pm-4pm, 4pm-7pm, 7pm-10pm, 10pm-6am) and then similarly perform precomputation only for a single representative of every segment. This short-cut may sacrifice a small amount of optimality, but makes a significant improvement in precomputation time.

Another 'trick' to make precomputation twice as fast is to precompute the tables only for departure events. During the query time, if we need to continue on the same service through one of the transit stations, the departure event associated with the arrival event of these service can be momentarily accessed by an auxiliary link connecting between those two events. It will cost of additional 4 bytes of storage for every pair *arrival-departure*.

Finally, similarly to Grid-TNR we notice that the process of identifying transit nodes for one cell is completely independent of other cells and therefore can be parallelized. Analogously, we observe that second phase of the precomputation and storage of the three database (*node-to-transit*, *transit-to-transit*, *transit-to-node*) tables is also independent and can be performed in parallel.

For public transportation networks, in order to provide a plethora of results we need to execute several (independent) queries between different source and destination stations. Clearly this can be easily parallelized, allowing us to achieve *location-to-location* query time essentially almost as fast as single *node-to-node*

query (as long we have enough available cores).

4.7 Dealing with Multi-Objective Queries

In addition to finding the fastest connection between two points, the user may consider also other criteria, such as the cost of tickets, hassle of interchanging between services, etc. Moreover, different users can have different preferences over these criteria. For example a businessmen may try to optimize his travel time, a student may try to minimize his costs and a visitor may wish to avoid changing services in order not to get lost. There are several ways to deal with multi-objective optimization Müller-Hannemann et al. [2007]. In this work we choose the normalization approach, by introducing a linear utility function. We will precompute the tables for different values of the linear coefficients that reflect different user preferences. This approach reduces the multi-criteria problem to a single-criterion optimization, which we can solve as described.

4.8 Providing multiple results in the real world

Until now we have assumed that our world is a public transport network, i.e. the start and end point of the journey is always a station node and we are departing exactly at time t. The result we obtain is theoretically optimal, but of course in the real world most of the time this is not the case. We may need to walk from our home to our first station and from the last station to our final destination. Moreover, in real life we would prefer to wait a little bit now if we know that eventually we may arrive earlier, for example to wait for an express bus. In

addition we all like choices, therefore the system will be more user friendly if it could provide multiple attractive alternatives to the user. In order to cover those real life scenarios and provide the user several attractive suggestions we will run our Query starting from different stations around the user's starting location and different times around his specified departure/arrival time t. From those we are choosing the best five. Experimentally we could see that this heuristics provide us with the most reasonable results that a person would make.

Another important aspect we consider is robustness of the provided solutions. In practice, public transport often runs late due to traffic congestion or accidents or other unpredicted events. Missing a connection by one minute may cost us an hour in our total journey time, if the next connection is infrequent and departs say only once an hour. In order to minimize such occurrences and to make the system more reliable and user friendly we identify [1] those trips and warn the user about risky connections. Then the user will choose his preferred trip according to his risk adverseness.

4.9 Implementation and Experiments

The proposed algorithm was implemented using the Java programming language and the experiments were performed on PowerEdge 1950 server, with those specifications: CPU: 2 x 2.00GHz Intel Quad Core Xeon E5405, Memory: 16 GB. The presented results were obtained using the Sydney metropolitan and state of New South Wales (NSW) public transport multimodal network consisting of buses, trains, ferries, lightrail and monorail modes. The graph representing Sydney net-

[1] we post-process each trip and identify risky connection that may cause to a significant total delay

work contains $9.3K$ station nodes, $2.1M$ event nodes and $8.1M$ links. The graph representing the whole NSW network contains $46.3K$ station nodes, $6.7M$ event nodes and $23.2M$ links.

Given a grid size $g \times g$ and sizes of $Inner$ and $Outer$, for every query it is very easy to verify whether it is a global or a local. We just need to check if the two nodes are within a local radius of each other. Therefore by simple sampling of many random queries we can have a good estimate of the percentage of global vs.local queries. It allows us to fine tune the sizes of grid, $Inner$ and $Outer$ to achieve the desirable percentage of global queries. Larger values for $Inner$ and $Outer$ normally yield smaller number of transit nodes, consequently requiring smaller memory requirement, but also covers a smaller number of global queries. We experimented with different sizes of the grid, $Inner$ and $Outer$ and came to a conclusion that for public transportation networks the choice of $Inner = 3 \times 3$ and $Outer = 5 \times 5$ provide better results than traditional choice $Inner = 5 \times 5$ and $Outer = 9 \times 9$ for road networks. Below we present results demonstrating tradeoff between percentage of global queries and memory/time requirements of the offline stage. In addition we present the average query time (in micro seconds) for a single query of different types: hub-to-hub (HH), station-to-station (SS), location-to-location (LL)[1] time complexity..

[1] we store the station nodes in KD-Tree structure, which allow us to find the closest node to a location in $O(log\ n)$

Area	Global queries	Grid size (cells)	$\|T\|$	Storage (Mb.)	Precomp. time (hours)	Query time (μs.)		
						HH	SS	LL
Sydney	70%	70 x 70	1174	321	0.7	91	95	104
Sydney	80%	89 x 89	1505	446	0.9	70	76	85
Sydney	90%	136 x 136	2256	825	1.2	49	55	65
NSW	70%	42 x 42	1023	215	1.5	70	76	85
NSW	80%	59 x 59	1733	375	2.5	89	95	105
NSW	90%	94 x 94	3270	1200	6.5	66	71	80

Table 4.2: Experimental results of applying Hub Grid-TNR on Sydney and NSW public transport network.

Applying the TRANSIT algorithm in a naive way, without including *hub nodes* and any speed ups yields memory requirements up to 45Gb for Sydney and 50Gb for NSW networks with the estimated precomputation time of several days.

4.10 Discussion

In this work we presented a novel approach for finding optimal connections in public transportation network based on Grid-TNR. We compare our algorithm with one of the fastest algorithms for public transportation networks [Bast et al., 2014], Google *Transfer Patterns* (TP) [Bast et al., 2010]. Algorithmically, the two approaches are fundamentally different. TP is based on the observation that many optimal journeys share the same transfer patterns. These transfer patterns are precomputed and stored for all pairs of nodes and departure times. Then, during a query, a small query graph is constructed and relatively fast search is performed on this graph. In order to accelerate the precomputation and to reduce storage space, hub nodes are heuristically selected and then the transfer patterns

are precomputed only for those hubs.

There are several advantages of our approach over *Transfer Patterns* (TP). First of all, TP precomputation requires large computational resources. The authors report requirements of 20-40 (CPU core) hours per 1 million of nodes. For the Swiss transit network, which is comparable in size to the Sydney network, the reported precomputation time is between $560 - 635$ hours, where for Sydney network the precomputation takes slightly more than one hour. The additional precomputed storage is comparable for both algorithms, $\sim$800Mb. An additional advantage of our approach is in a query time. TP requires perform an online search in a small[1] query graph and it is reported a typical query time of 10ms. Grid-TNR approach requires only limited number of table lookups, independently of the size of the underlying network and its global query time is measured in μs. The advantage of TP is that it successfully handles all types of queries, while for local queries, which are not direct connections, we need to fall back for a local search. Nevertheless, considering the case when 10% of the queries are local and take 10ms and the rest 90% of the global queries take 50μs, it gives us an average query time of $\sim$1ms, which is still faster than 2-3ms reported by TP.

4.11 Conclusions and Future Work

In this work we presented a novel approach for finding optimal connections in public transportation network based on Grid-TNR. We presented an experimental results using the Sydney and New South Wales (NSW) timetables. There are several interesting directions that we thing worth pursuing.

[1] typical number of arcs is below 1000

In 4.4.1 we identified transit nodes as a subset of station nodes. This approach very successfully solves the scalability issue, but introduces a new challenge. Between two stops, one path can be a shortest path, say in the morning and another one in the evening. Therefore it could be desirable to make transit nodes time dependent. When querying for a journey a time t, we can fetch only access nodes associated with this time, rather all access nodes as we do today. This in turn will speed-up final query time.

It would be interesting to investigate the hierarchical TNR, introducing several layers of *transit* nodes. This should reduce local queries for which we'll have to invoke a fall back algorithm.

Applying CHAT (Section 3.4) looks like another potential improvement.

In addition we would like to extend this idea to fully realistic inter modal journey planer which includes combination of private transport (e.g. car, motorbike, bicycle) and public transport and incorporates real time updates for both traffic conditions and public transport actual location.

Chapter 5

Grid Networks

Part of the results obtained in this Chapter was a collaborative work with Daniel Harabor. A joint paper appeared in AIIDE'13 [Antsfeld et al., 2012].

5.1 Grid Graphs

Grid maps are widely used in video games and other domains (AI, for example). There are two variants of grid based graphs. In the first instance, every node may be connected only to its nearest horizontal or vertical neighbor with a weight equals 1. In the second instance, diagonal moves are also allowed, therefore additional diagonal links can be present with a weight equals $\sqrt{2}$. Below is an example of typical grid maps from a game "Baldurs Gate II" and artificially created maze.

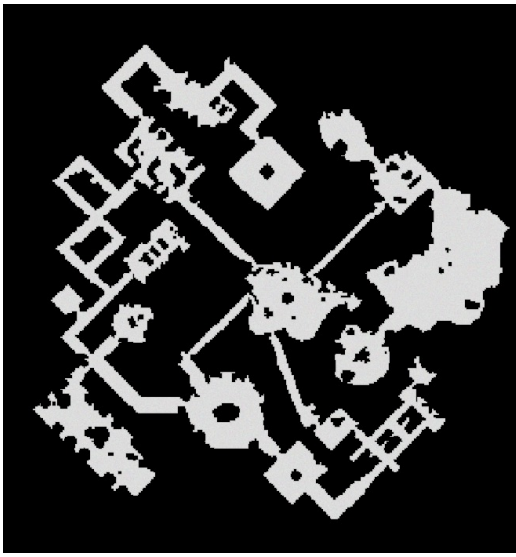

Figure 5.1: Map from Baldurs Gate II

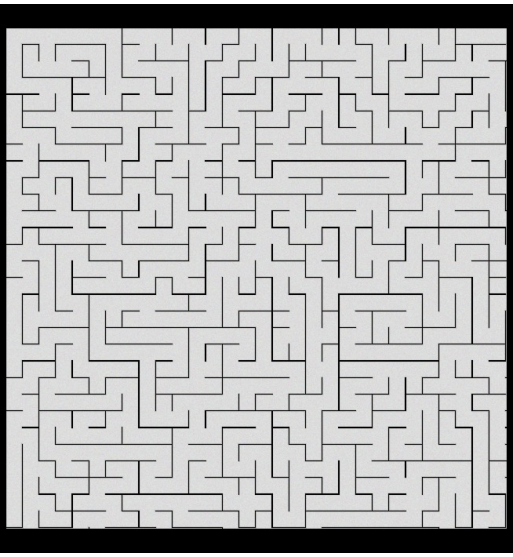

Figure 5.2: Synthetic map of a maze

5.2 Related Work

The AI and Game Development communities have devoted much attention to the study of both exact and approximate techniques that speed up forward state-space search algorithms such as Dijkstra and A*. Below we review the fraction of the work which is the most relevant to our research.

Near-optimal Techniques

In [Botea et al., 2004] the authors presented a Hierarchical Path-Finding A* (HPA*). Initially the grid map is divided into disjunct rectangular clusters and transitions between those clusters are identified. Next, an abstract graph is defined as follows. For each transition between two adjacent clusters, there are two nodes (one for each cluster) and a *inter-edge* link that connects them. Each pair of nodes inside the cluster is connected with an *intra-edge* link. Then, an online query is carried out by A* on the abstracted graph. The authors reported up to

10 times speed-up over highly optimized A*, when the cost of the found paths are within 1% of optimal. Later in [Sturtevant, 2007] this approach was improved by combing it with Path-Refinement A* (PRA*) [Sturtevant and Buro, 2005]. A speed-up of up to 100 times over A* was reported, at the cost of 3% additional storage.

True Distance Heuristics (TDH)

The basic idea is to improve the heuristic used by A* search. Clearly, the *perfect heuristic* will guide A* directly to the goal, but it will require precomputation and storage of *all pairs* shortest paths, which is, of course, in many cases impractical. In [Sturtevant et al., 2009], the authors presented two methods that compute and store only a small part of this information. Experimental results on a number of domains showed a 6-14 fold improvement in search speed over previously known heuristics.

An additional attempt, called *Portal Heuristics* (PH) was introduced in [Goldenberg et al., 2010]. The algorithm is based on partitioning the state space into regions, ideally of similar size with few border states (a.k.a. nodes). It somewhat resembles Precomputed Cluster Distances (PCD) [Maue et al., 2010], with the main difference that here the auxiliary data is used as a heuristic for a following online A* search. For the instances where the grid map could be "nicely" partitioned, (i.e. divided into regions that are connected with each other by small number of edges), PH outperformed previous TDHs.

Compressed Path Databases (CPD)

CPD [Botea, 2011, 2012] is relative new technique for pathfinding without the need of runtime search. The basic idea, similar to Spatial Induced Linkage Cognizance (SILC) [Sankaranarayanan et al., 2005], is that many shortest paths which start at the same location, share the same first move, regardless of the final destination. SILC decomposes the map into quadtree structures and then performs a binary search to find a block that contains a location l. CPD, for every location l decomposes the map into equivalences classes, such that every class contains all destinations reachable from l starting in one of the 8 possible directions. The author report up to 700 times speed-up over A*. Since some of our results intertwine with CPD, we will describe this algorithm in more details in the Section 5.5.1.

Swamps

In [Pochter et al., 2010, 2009], the authors introduced the concept of *swamps* and *swamp hierarchies*. The idea is to use preprocessing to eliminate set of nodes that eventually will not appear on any shortest path. This idea is similar to [Björnsson and Halldórsson, 2006] where the authors use preprocessing to identify *dead ends*. The main difference between those two approaches is that the *swamps* also identify and prune other areas that can be avoided, if there is an alternative shortest path.

Jump Points

In [Harabor and Grastien, 2011], the authors introduced a new online pruning strategy using *jump points*. The idea is, during the search, to eliminate intermediate nodes on a path between two jump points, such that jumping over them does not affect optimality. The authors reported order of magnitude faster queries compared to *Swamps*.

Contraction Hierarchies (CH)

As there is no clear hierarchy among the edges in the grid, it was not that obvious that CH will work "out of the box". The first attempt to apply CH on grid maps was reported in [Sturtevant and Geisberger, 2010]. The authors reported an order of magnitude speed-up over A*. While at that time it considered to be state-of-the-art, those results are superseded today by CPD.

The most recent result [Storandt, 2013] also demonstrates that it is possible to apply CH on grid maps. While no modifications to CH were necessary to run CH on grid maps, the author suggested a speed-up technique, that also allows to extract *canonical* paths, i.e. a paths with a minimum number of turns. The algorithm was tested on some selective instances and showed a speed up of two orders of magnitude over A* with precomputation time of about 10 sec. and additional storage of up to factor of 7.

5.3 Contribution

Our contribution is as follows: We give a first detailed evaluation of Grid-TNR (see Chapter 3.2) on popular grid domains from the AI literature. We find that Grid-TNR is strongly and negatively impacted, in terms of running time and also memory, by uniform-cost path symmetries that are inherent to many grid-based domains but not road networks. We address this with a new general technique for breaking symmetries using small additive ϵ-costs to perturb the weights of edges in the search graph. Our enhancements reduce the number of nodes in the Grid-TNR network by up to 4 times and yield running times up to 4 *orders of magnitude* faster than A* search. We also report on additional benefits derived from tuning various preprocessing parameters, including cell size, border crossings and reductions to the number of queries which are considered global.

We also compare Grid-TNR with CPDs [Botea, 2011]: a recent and very fast database-driven pathfinding approach. Our results indicate the two algorithms have complementary strengths and we suggest an approach by which they could be combined. However, we also identify a class of problems to which Grid-TNR appears uniquely well suited. In these cases we report up to two orders speed improvement vs CPDs using a comparable amount of memory.

Finally, we introduce a new hybrid method combining CPD with Grid-TNR for path extraction. We give an evaluation of our method using popular grid domains from the AI literature Sturtevant [2012a] and find that the synergy of Grid-TNR with CPD outperforms solely Grid-TNR based path extraction by up to more than three orders of magnitude. In addition we observed that the suggested algorithm is very competitive with an original CPD path extraction -

in both memory requirements and the query time, where in addition it answers very fast the distance queries.

5.4 Grid-TNR

In addition to computing shortest paths it is sometimes desirable to efficiently calculate the distance between two units on a map or the distance to an object of interest. Support for such *distance queries* is found in popular game libraries, including Umbra 3[1], where they are used for optimizing game logic and driving scripted events. Distance queries may also be useful for higher level AI; e.g. as described in [Champandard, 2009].

The Grid-TNR algorithm does not appear to have been tested previously on grid-based maps of the type commonly found in game maps. On our first attempt we observed the algorithm often has prohibitively large memory requirements and relatively long query times. This behavior can be traced to a property commonly found in grid maps but rarely in road networks: uniform-cost path symmetries [Harabor and Grastien, 2011]. To overcome this difficulty we propose a novel symmetry breaking technique which we apply during preprocessing and which involves the addition of small positive "noise" to all edge weights. This idea has been previously suggested in the context of symmetry breaking for Integer Linear Programming [Margot, 2009] but the authors note that such perturbation "does not help much and can even be counter-productive". In our case, perturbation of edge weights significantly reduces the number of transit nodes and leads to faster preprocessing times, lower memory requirements and significantly better

[1] http://www.umbrasoftware.com/

query times. To the best of our knowledge, we are the first to successfully apply such a technique to symmetry breaking in pathfinding search. Our idea is generally applicable to any kind of search graph; we exemplify it on the 4-connected grid in Figure 5.3.

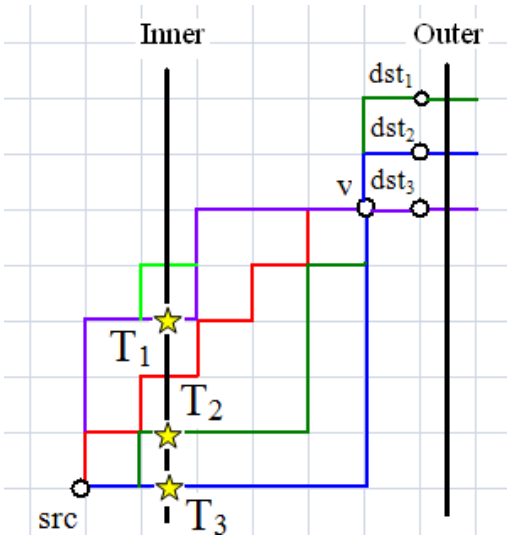

Figure 5.3: (a) Example of many symmetric shortest paths between src and v in 4-connected grid network. (b) Example of shortest paths from src to dst_1, dst_2 and dst_3 that share many common shortest subpaths from src to v.

Assume that during Grid-TNR's preprocessing phase we are calculating shortest paths from the node src to each of dst_1, dst_2 and dst_3 – which all reside on the border of the Outer square O. Notice that each shortest path shares a common symmetric subpath $src \rightsquigarrow v$ and crosses the Inner square I in three different locations. Therefore we identify T_1, T_2 and T_3 as transit nodes (starred locations in Figure 5.3). Our intuition is as follows: if we will add a "small" random ϵ-cost to each edge in the grid, there will be (with high probability) [1] only one shortest

[1] See the Appendix for more detailed analysis

path to node v, e.g. through T_3. We assume here that $src \rightsquigarrow v$ will appear as a common subpath for two or more of dst_1, dst_2, dst_3; although theoretically this is not guaranteed it occurs very often in practice.

We will now show that choosing ϵ sufficiently small will preserve optimality of all shortest paths.

Definition 5.1. Let $G = (V, E)$ be a weighted graph with integer costs (if they're not integer, we just scale them up by a suitable factor so that they are) and let L be the number of links on the longest shortest path in G. Then we define $\epsilon < \frac{1}{L}$.

Note for all practical purposes there is no need to actually calculate the length of the longest shortest path L, but we can use $|V|$ as an upper bound.

Definition 5.2. Let $G(\epsilon)$ to be an exact copy of G with the only difference that for every edge e in $G(\epsilon)$ we add a random number from the interval $(0, \epsilon)$ to its weight.

Lemma 5.1. *For every optimal path π_ϵ in $G(\epsilon)$ there exists a corresponding shortest path π in G that traverses through exactly the same nodes as π_ϵ.*

Proof. By contradiction. Let π_ϵ in $G(\epsilon)$ be an optimal path between two nodes *src* and *dst*. Let π be a path in G which travels through the same nodes as π_ϵ but is not optimal. This means there exists another path π' between *src* and *dst* in G which is strictly shorter than π. Let π'_ϵ be a non-optimal path in G_ϵ which travels through the same nodes as π'. Now we notice that the smallest difference between costs of any two paths in G can be at least 1. From Definition 5.1 this can only happen if π'_ϵ is longer than L, which is impossible. □

A natural value for L in a 4-connected grid map is $\epsilon = \frac{1}{L}$; for 8-connected grids we define $\epsilon = \frac{2-\sqrt{2}}{L}$.

Corollary 2. *The number of transit nodes identified in $G(\epsilon)$ is no greater than in G*

A direct conclusion from the latest discussion is that during the identification of transit nodes stage, we can safely substitute G with $G(\epsilon)$ and in practice eliminate all symmetric path segments. This will reduce the number of transit nodes, precomputation time and storage space as well as final query time.

Notice that perturbation of the graph weights in the manner described above is not specific to grid maps or indeed to any implementation of the Grid-TNR algorithm. It is a general technique for reducing symmetries in pathfinding search and could be used in other settings: e.g. Dijkstra's algorithm.

5.5 Path Extraction

In the domain of grid networks, in most of the cases we are required to find the shortest path itself, rather than simply its cost. In this section we present a new idea of extracting the shortest path by combining Grid-TNR with CPD. A brief overview of CPDs is necessary to understand how the idea is integrated into our architecture. For a more detailed description we point the reader to the original publications [Botea, 2011, 2012].

5.5.1 CPD

In the pre-processing phase, All Pairs Shortest Paths (APSP) data are computed as an iterative procedure. There is one iteration for every map node n. An invocation of the Dijkstra algorithm allows computing a first-move table T_n with

optimal moves move(n, t) from n towards any reachable target t on the map.[1] The table's two-dimensional size mirrors the size of the map, with cell $T_n[x, y]$ storing move(n, t) for the target t with map coordinates (x, y).

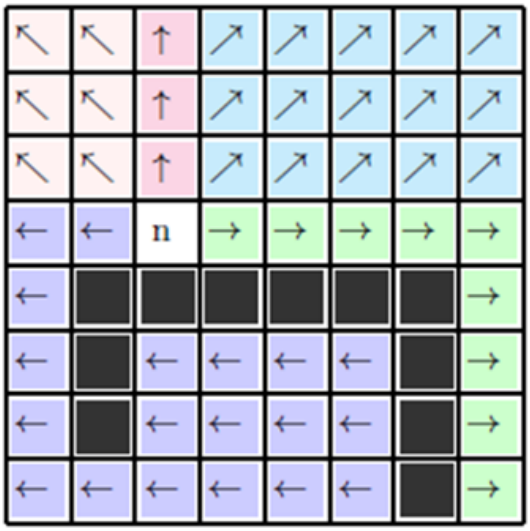

Figure 5.4: Example of the first-move table[2]

The second half of an iteration is the compression, when a first-move table is decomposed into a list L_n of disjoint so-called homogeneous rectangles. All locations t within such a rectangle have the same move label move(n, t). The fewer the rectangles, the better the compression. Part of these can be removed, without any information loss, with a list trimming step, which allows removing from a list rectangles with the same move label d, called a default move [Botea, 2012].

[1] For example, move(n, t) can be the id of an outgoing edge from n.
[2] The drawing by Adi Botea.

Query

At runtime, given a current node n and a target t, we need to retrieve an optimal move from n towards t. The program searches in the list of rectangles in L_n for the rectangle that contains t. The move label of that rectangle is the sought optimal move. Ordering rectangles in L_n decreasingly by their size improves the move fetching time. If no rectangle containing t is found in L_n, it means that the rectangle has been removed as a result of trimming. Thus, the rectangle must have the move label d, which is the move to return as an optimal step from n towards t.

The offline pre-processing can be paralellized, since it consists of a series of independent iterations. The pre-processing time can be critical, for instance in cases such as a dynamically-changing environment, which would require us occasionally to re-compute a CPD. Despite the ability of CPDs to compress All Pairs Shortest Paths (APSP) data by hundreds of times without any information loss, their size can sometimes be a performance bottleneck, depending on the map size, the topology, and the amount of RAM available.

5.5.2 Grid-TNR with CPD

The basic idea of our approach is to break a global $src \rightsquigarrow dst$ query into number of shorter, local subpaths:

$$src = T_0 \rightsquigarrow T_1 \rightsquigarrow T_2 ... \rightsquigarrow ... T_{k-1} \rightsquigarrow T_k = dst.$$

Then we reconstruct our $src \rightsquigarrow dst$ path by sequentially extracting local subpaths $T_i \rightsquigarrow T_{i+1}, i = 0...k$ from the CPD.

5.5.2.1 Precomputation

We immediately notice that all the queries from the CPD are only local, therefore we can precompute CPDs only for pairs of nodes which are within a local search distance of each other. This is a major saving in both precomputation time and memory requirements of CPD.

5.5.2.2 Query

We start with invoking the basic Grid-TNR query. We remember that every global shortest path, (that is longer than a local search radius) is of a form: $src \rightsquigarrow T_{src} \rightsquigarrow T_{dst} \rightsquigarrow dst$. By definition, subpath $src \rightsquigarrow T_{src}$ is local, therefore we are applying a CPD query to extract the shortest path from src to T_{src}. Next, if $T_{src} \rightsquigarrow dst$ is a local query, we extract this subpath from CPDs as before and we are done. If $T_{src} \rightsquigarrow dst$ is a global query, then we will repeat the above procedure but this time using T_{src} as our new src. We can improve this even more, by noticing that when running sequential queries with Grid-TNR T_{dst} and the distance from T_{dst} to dst are not changing (see Figure 3.3). We exploit this fact, by reusing T_{dst}. This speeds up the time complexity of subsequent Grid-TNR queries from quadratic to linear in the number of the transit nodes. Below we present the pseudocode of the described algorithm.

One of the advantages of this approach over the techniques mentioned earlier is that we advance from src to dst by "chunks" of local subpaths, rather than by a single link. In addition for every such "chunk" we exploit the strength of the CPD algorithm that is very efficient for local shortest path queries. Actually we

```
(T_src, T_dst) ← Grid-TNR.query(src, dst);
CPD.getShortestPath(src, T_src);
while ((isLocalQuery(T_src, dst) == false))
    (T̄_src, T̄_dst) ← Grid-TNR.query(T_src, dst);
    CPD.getShortestPath(T_src, T̄_src);
    T_src := T̄_src

CPD.getShortestPath(T_src, dst);
```

Algorithm 6: Pseudo code for extracting the shortest path between *src* and *dst*

have observed that for some instances the described algorithm performs slightly better than pure CPD extraction.

In the next section we will present a comparative analysis of this algorithm.

5.6 Experiments

We evaluate Grid-TNR on a subset (Table 5.1) of Sturtevant's popular and freely available [1] grid map benchmarks [Sturtevant, 2012a].

Benchmark	Map	Size	States
BG	AR0602SR	299 × 308	23 314
	AR0700SR	320 × 320	51 586
DAO	orz100d	395 × 412	99 626
	orz900d	656 × 1491	96 603
Mazes-1	maze512-1-0	512 × 512	131 071
Random-10	random512-10-0	512 × 512	235 900
Rooms-32	32room_000	256 × 256	240 671

Table 5.1: Grid maps used for evaluation of Grid-TNR.

[1] http://movingai.com/benchmarks

These problem sets have appeared extensively in the literature; for example in [Björnsson and Halldórsson, 2006; Botea et al., 2004; Felner and Sturtevant, 2009; Goldenberg et al., 2010; Harabor and Grastien, 2011; Pochter et al., 2009; Sturtevant, 2007]. Some, such as BG and DAO, are taken from real video games. The others, comprising Rooms-32 and Mazes-1, are synthetic. We selected from each benchmark set maps we considered to be the most challenging; either due to the presence of extensive uniform-cost path symmetries (as discussed in the preceding section) or due to topographic features (e.g. narrow corridors, dead-ends etc.) which are likely to induce significant error for standard heuristics such as Manhattan Distance (can move to 4 neighbors) and Octile Distance (can move to 8 neighbors). For each map we generate 10K valid problems from across all possible problem lengths. Our implementation is written entirely in Java. We perform all experiments on an Intel Core2Duo with 8GB RAM. For comparative purposes we include results for a similar set of instances using Compressed Path Databases (CPD). This algorithm is originally described in [Botea, 2011]; its source code was kindly made available to us by the original author.

Efficiently Approximating Network Size

Grid-TNR's performance strongly depends on a set of preprocessing parameters: (i) the size of each grid cell C and (ii) sizes of the Inner square I and Outer square O. It is not clear apriori how to choose those parameters. Finding a set of parameters that is effective for a given input map can be costly, requiring many invocations of Grid-TNR's preprocessing phase.

In this section we propose a very simple heuristic that we found useful for

choosing those parameters and quickly estimating the size of the final preprocessing data as well as percentage of global vs. local queries. For a given overlay grid of size $m \times m$, we count the number of edges crossing each horizontal and vertical line of the grid. This number gives us an upper bound of the number of transit nodes and allows us to tune the grid size to match available computing resources and application requirements. Having selected a grid size S and sizes of I and O, for every query it is very easy to verify whether it is global or local: we just need to check if the two nodes at hand are within a local radius of each other. Using simple random sampling we can build a good estimate of the percentage of global vs.local queries and we can adjust our parameter values until we achieve the desired result. In our experience, larger values for S, I and O normally yield a smaller number of transit nodes and require less memory overall but also cover a smaller number of global queries.

Results

We evaluate Grid-TNR on each of our input maps and measure its performance in terms of: total number of transit nodes (T), database size (DB) and global query time. To provide a common point of reference we report the latter in terms of *speedup* which we define as relative improvement vs. standard A* search. The exception is Table 5.2 where we report times in μs. Database size is always in MB.

Map (* = no diag.)	(S, I, O)	G TN	G QT	G_ϵ TN	G_ϵ QT
32room_000	(32, 5, 9)	2482	14	1922	8
32room_000*	(32, 5, 9)	8858	183	1837	8
AR0602SR	(45, 5, 9)	4251	61	3618	38
AR0602SR*	(45, 5, 9)	4273	89	2173	14
AR0700SR	(40, 5, 9)	10 173	205	9035	122
AR0700SR*	(40, 5, 9)	8769	268	4724	38
maze512-1-0	(38, 5, 9)	1707	2	1707	2
maze512-1-0*	(38, 5, 9)	1707	2	1707	2
orz100d	(43, 5, 9)	18 852	643	16 934	419
orz100d*	(43, 5, 9)	16 189	844	10 192	141
orz900d	(57, 5, 9)	4886	105	3732	74
orz900d*	(57, 5, 9)	2303	74	1177	29

Table 5.2: Effect of adding random ϵ-costs to edge weights. G is the original graph and G_ϵ is the graph with perturbed edge weights. TN = total transit nodes. QT = global query time (μs), S = grid size, I = *Inner* cell size, O = *Outer* cell size. Note that there are two versions of each map: one which allows diagonal transitions and the other which does not.

5.6.1 Symmetry Reduction

In Table 5.2 we give results for our ϵ-based symmetry breaking approach. We run Grid-TNR on two variants of each input map: one where diagonal transitions are allowed and the other where they are not. Both are common in games and often studied in the literature.

In the case where diagonal transitions are disallowed the addition of random ϵ-costs to edge-weights has a dramatic effect, reducing the number of identified transit nodes by a factor of between 2-4 and reducing global query times by anywhere between 2.5 times to over one order of magnitude. When diagonal transitions are allowed (which is always the case in the remainder of this section) the improvement is less dramatic but remains strongly positive: we reduce the

number of transit nodes by between 10-25% and improve global query times by up to a factor of 2.

5.6.2 Distance Queries

We compare Grid-TNR with Compressed Path Databases [Botea, 2011] (CPD) on each map in our test set [1]. CPD is one of the latest and fastest algorithms for shortest path queries without state-space search [Sturtevant, 2012b]. Only a linear number of lookups into a compact database are required (the number is equal to the individual steps on the path). By comparison Grid-TNR performs a single lookup operation which compares an up to a quadratic number of local transit nodes. Figure 5.5 summarises our findings. Note that values along the y-axis are logs in base 10.

We plot in most cases three curves for Grid-TNR; each one represents a different set of preprocessing parameters including different abstract grid size (S) and different sizes for the Inner square I and Outer square O (we measure both of these in terms of cells in the abstract grid). We also give the average number of transit nodes (T) for each cell in the abstract graph and the percentage of queries which are global (GQ) and do not require any state-space search. Remaining local queries are omitted (these paths are usually short and can be solved using any available search algorithm; for example Jump Point Search [Harabor and Grastien, 2011]).

[1] diagonals are always allowed here.

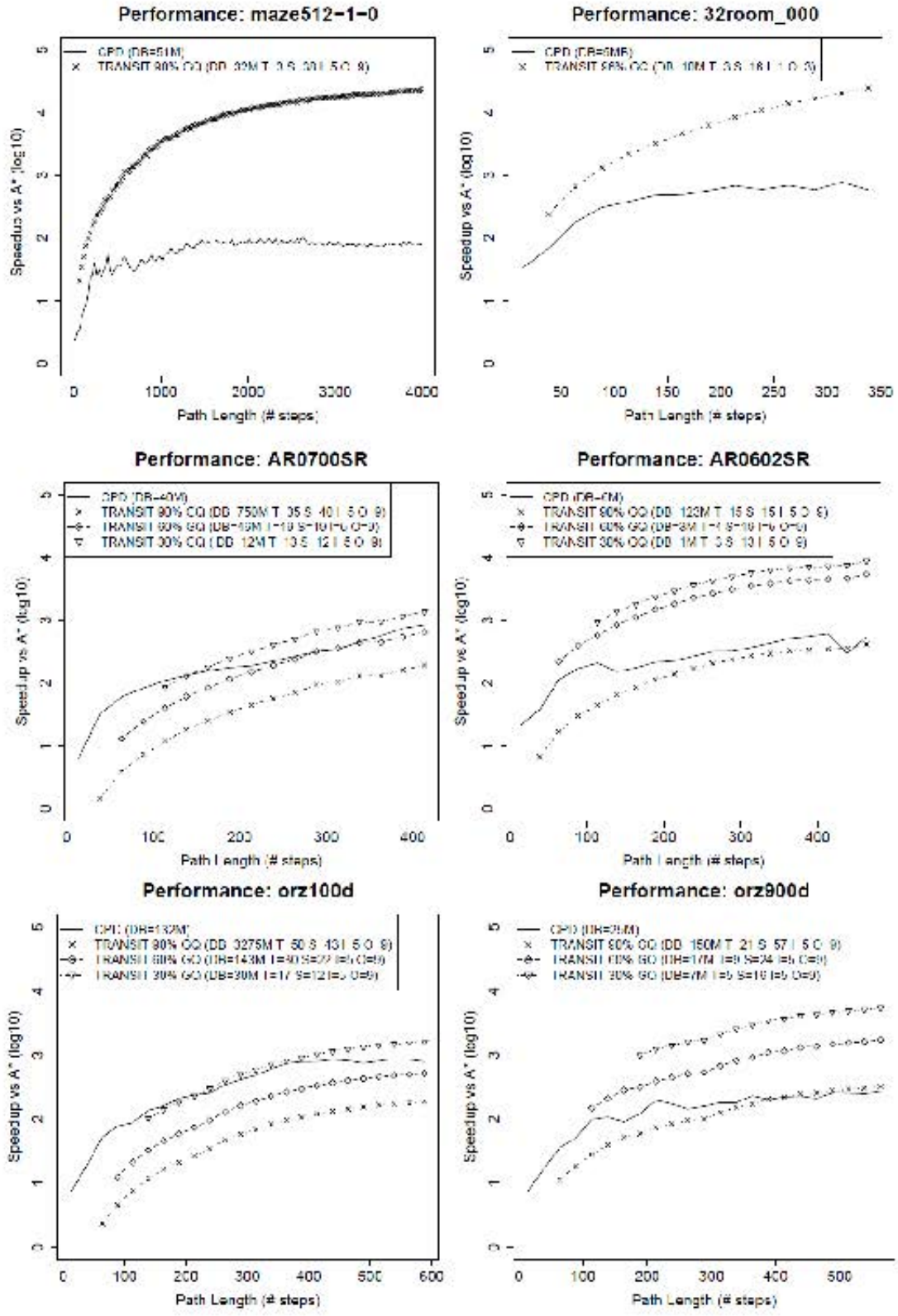

Figure 5.5: Search time speedup (i.e. relative improvement) of Grid-TNR and CPDs vs. A*. Note the log10 scale on the y-axis.

We observe that on domains containing no symmetries (Mazes) or only short symmetric path segments (Rooms) Grid-TNR outperforms CPDs by between one and two orders of magnitude (and up to 4 orders improvement over A* search). Grid-TNR requires a moderate amount of storage (32MB and 10MB respectively) and covers 90% of all queries without search (the remaining 10% are local). CPDs require 51MB and 5MB respectively and cover all queries. Notice that we store only a very small number of transit nodes per cell. We experimented with different preprocessing parameters beyond those given in Figure 5.5 but were unable to reduce this number further for additional speed gains.

The remaining domains, particularly orz100d and AR0700SR, are characterized by large open areas and long symmetric path segments that often span several cells in the abstract grid. The effect of our symmetry breaking techniques using ϵ-costs is less significant for finer grids, because the *Outer* square induced by the fine grid will be naturally smaller and contain fewer symmetrical paths in the first place. As a consequence, for 90% coverage, its performance is dominated by CPDs; both in terms of time and database size. We ran additional experiments on these problems using two smaller values for global query coverage: 60% and 30%. We used a coarser abstract grid in these cases, having larger (absolute) values for I and O. 60% global query coverage omits all paths of lengths between 50-125 (depending on the domain). 30% coverage can omit in some cases all paths up to length 175. In return for this tradeoff, we see a dramatic reduction in the average number of access nodes per cell (T) and a corresponding improvement in performance: Grid-TNR is shown to be up to an order of magnitude faster

than CPDs using 30% global query coverage and requires substantially less memory; in some cases just a few MB. For 60% global query coverage, Grid-TNR is comparable with CPDs, both in terms of performance and database size.

5.6.3 Path Extraction

As we have discussed earlier Grid-TNR and CPD have complementary strengths and weaknesses. In terms of distance query time, Grid-TNR is very fast for global queries and relies on other algorithms for local queries. CPD on other hand is very fast for short queries and becomes slower as the shortest path becomes longer. In terms of memory requirements, as was discussed in [Antsfeld et al., 2012], there is a clear tradeoff between memory requirements and final query time for Grid-TNR. In order to take advantage of the strengths of both algorithms we choose the Grid-TNR parameters that will guarantee us at least 30% of the queries to be global [Antsfeld et al., 2012]. Those are usually the longest and the most challenging queries for CPD.

Map	**CPD**	**CPD$_L$**	**Grid-TNR**	**Combined**
AR0602SR	6.3	3.7	1.2	4.9
AR0700SR	41.7	22.8	12.3	35.1
orz100d	138.2	78.2	31.5	109.7
orz900d	25.8	25.8	9.3	35.13
maze512-1-0	53.2	28.7	7.1	35.8
32room_000	215.4	171.7	40	211.7

Table 5.3: Sizes (in Mb) of the Grid-TNR database, CPD, Local CPD (CPD$_L$) and the combined method.

We evaluate path extraction using three methods (Grid-TNR, CPD and Grid-TNR+CPD) on each of our input maps and measure its performance in terms of: memory requirements (Mb.) and global query time (μs). Figure 5.3 summarizes the additional storage requirements, where CPD_L denotes the CPD database restricted only to local queries.

Figure 5.6 below demonstrates the time required for full path extraction as a function of the shortest path length (measured in number of links).

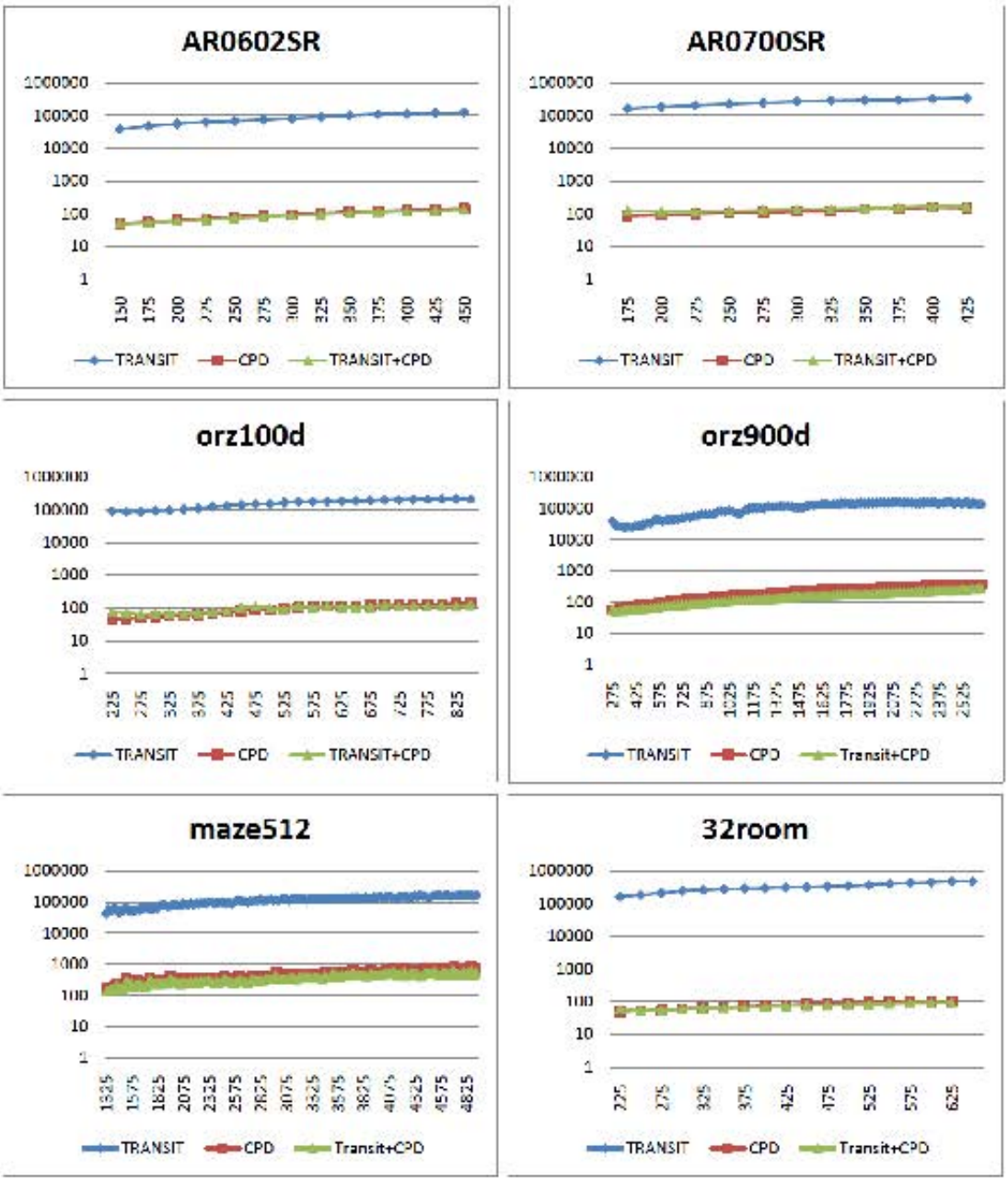

Figure 5.6: Path extraction time (μsec.) as function of the shortest path length (♯ of links). Note the log10 scale on the y-axis.

We observe that our new approach outperforms by 2-3 orders of magnitude the state of the art approach that uses purely Grid-TNR database. Moreover for some of the instances, the combined method performs slightly better than pure CPD path extraction.

Discussion

We show in the preceding section that Grid-TNR is able to compete with and even outperform CPDs for a certain class of distance query: those where the start and goal are not in close proximity. Such problems are usually considered difficult for state-space search algorithms but also for CPDs because more lookups are required and each lookup has an associated, and often linear-time, cost. On one hand, CPDs are attractive because they perform well in practice and offer complete coverage of all queries without resorting to any localized state-space search (as is usually the case for Grid-TNR), as well naturally produce an actual path. On the other hand, CPDs preprocessing time is longer and in case of any change in the network the precomputation has to be made again from scratch. By comparison, recent variants of Grid-TNR can incrementally update the precomputed costs database [Antsfeld and Walsh, 2012c], when the link weight increases.

We find that the two methods have quite different strengths and characteristics and believe them to be orthogonal and easily combined. For example: we could compute a small Grid-TNR database covering queries longer than some minimum length. To cover all remaining queries, we could compute a CPD that contains only paths of lengths less than this minimum: i.e. during the all-pairs shortest path computation, do not generate any successors beyond the predefined

limit. This is a much smaller subset of nodes than CPDs usually consider and we expect it will require proportionally less space to store and potentially less time to perform lookups. Once preprocessing is complete, any given distance query is either global for Grid-TNR, and we can extract it very fast, or we invoke Grid-TNR's local query algorithm and extract the length from our local CPD. Using a similar procedure we can also very quickly extract the actual shortest path for any given distance query.

5.7 Conclusions and Future Work

We have reported the first results for Grid-TNR [Bast et al., 2006] route-finding to grid-based benchmarks from video games. We find that on such domains the basic algorithm is impacted, in a strongly negative way, by the presence of uniform-cost path symmetries. To address this, we give a new general symmetry breaking technique involving the random perturbation of edges in the input graph with small ϵ-costs. We prove this technique is optimality preserving and show that it can reduce Grid-TNR's memory overhead by several factors and improve performance by up to two orders. We undertake an extensive empirical analysis of Grid-TNR on a range of popular grid-based pathfinding benchmarks taken from video games and give a first comparison of Grid-TNR with CPDs [Botea, 2011]. We find the two have complementary strengths and identify a class of problems to which Grid-TNR appears better suited: distance queries involving start and goal locations that are not in close proximity, i.e global queries of the Grid-TNR. For path extraction we proposed a hybrid method that combines Grid-TNR with CPD. We found that the proposed method extracts the complete shortest path

2-3 orders of magnitude faster that any known method based solely on Grid-TNR database. Also we found that our method in some cases performs slightly better than pure CPD while in the most cases require less memory than full-scale CPDs. One obvious direction for future work is to apply Grid-TNR on a sparse graph created by Jump Point Search [Harabor and Grastien, 2011]. Another interesting direction is to reduce the number of the transit nodes, by applying CHAT (see 3.4). Finally, future work would be to extend CPD idea for road networks. While Grid-TNR is already well known technique for roads, there are no reported results for CPD on road networks.

Chapter 6

Concluding Remarks

6.1 Summary

In this work, we have investigated the problem of pathfinding between two points in large scale graphs in three major domains: road networks, pubic transportation networks and finally grid maps. While the basic shortest path problem was solved by Dijkstra already in 1956, there are many practical applications where this solution would be too slow. In addition, some domains impose significant challenges that were not addressed by Dijkstra. Consequently, there is an ongoing extensive research in the area of path finding.

In this thesis we addressed some of those challenges and presented efficient algorithms, based on Grid-TNR for routing in large scale networks.

Road Networks

Initially we adjusted original Grid-TNR [Bast et al., 2006] to work on realistic, directional networks. We formally proved Grid-TNR correctness and per-

formed an approximate complexity analysis. Next, we extended Grid-TNR to deal with dynamic travel time changes (traffic jams for example) and developed a new mechanism to allow incrementally updates, rather than performing full re-computation. Finally, we presented a novel algorithm, CHAT which address some of the main weaknesses of the Grid-TNR. Our experiment show that in some cases, we outperform state-of-the-art algorithms in this domain.

Public Transportation Networks

First of all, we developed a novel, fully realistic model of public transportation networks. Then, we successfully applied Grid-TNR on our new model. We presented a significant improvement, by introducing *hub* stations. We believe that this idea can be also integrated with other previous techniques. In addition, we presented numerous practical speed-up techniques. Our experiment show that applying Grid-TNR to the public transportation domain is not only feasibly, but actually very competitive with current state-of-the-art algorithms.

Grid Maps

We were the first to investigate and apply Grid-TNR algorithm in this domain. Many similar paths, a.k.a. symmetries, was a challenge that made Grid-TNR perform relatively poorly. We addressed this problem, introducing a new technique of symmetry breaking, by adding a small *random noise*. For distance queries, we outperformed state of-the-art, CPD algorithm by an order of magnitude. For complete path extraction, we showed how to efficiently combine Grid-TNR with CPD.

6.2 Future directions

In concluding section of each chapter, we have mentioned possible improvements and future research directions. In addition to those concrete suggestions, the real world application impose new challenges.

Currently Grid-TNR (and CHAT) support static networks, i.e. where the weights of the links are constant and time-independent and single objective criteria. It might be interesting to extends those techniques to deal with time-dependent and multi-objective scenarios. We presented the first attempt of network *dynamization* by introducing incremental updates of the TNR database, where we dealt with multi-objective cases using linear utility function.

One of the weaknesses of TNR is a fall-back for another algorithm for non-global queries. While, most likely, it wouldn't be impossible to eliminate local queries completely, it would be desirable to minimize their number and provide some guarantee for the local query performance.

Minimizing TNR database is another challenge. While our new algorithm, CHAT, could reduce TNR database significantly, we believe there is still room for improvement. Finding the smallest number of access nodes looks like a key to address this issue.

6.3 Outlook

The basic technique of preprocessing and storing auxiliary data that can be used to speed-up later computations is a powerful idea. As usual, there is always a *time-memory tradeoff* and the challenge is to find a fine balance between those

two concepts.

For some optimization problems, when very fast querying of a lot of shortest paths are required, Transit Node Routing (TNR) algorithm is a perfect candidate for this task. This is because of its very simplistic query mechanism - a few precomputed table lookups and a few basic mathematical operations. While this is still a challenging and open problem, we believe that adopting TNR to deal with fully dynamic networks will result in considerably faster query times than current methods.

Appendix

In this section we want to study the effect of adding a random noise to the link weights as we suggested in 5.4. More specifically, we would like to estimate the probability that two symmetric paths will have exactly the same total weight. Assume we have c bits to represent a number. Then $p = 2^c$ is the number of possible integer values we can represent with c bits. We are interested to answer the following question: what is the probability that two random sets of the same size chosen uniformly from $\{0, 1, 2, 3, ..., p\}$ will sum up to the same value? More precisely:

Let X_i be Identically Independent Distributed (I.I.D.) Discrete Uniform Random Variables from $\{0, 1, 2, 3, ..., p\}$. Then $\mu = \frac{p}{2}$ and $\sigma^2 = \frac{(p^2-1)}{12}$

Let $S_n = \sum_{i=1}^{n} X_i$. Then the probability that two random sums are equal is:

$$P(S_n^1 = S_n^2) = \sum_{k=0}^{np} P(S_n^1 = k \wedge S_n^2 = k) = \sum_{k=0}^{np} P(S_n^1 = k)P(S_n^2 = k) = \sum_{k=0}^{np} P(S_n = k)^2$$

Using *standardization* and *continuity correction* we can write:

$$P(S_n = k) = P(S_n^* = \frac{k - n\mu}{\sqrt{n\sigma^2}}) \approx P\left(\frac{k - n\mu - \frac{1}{2}}{\sqrt{n\sigma^2}} \leq S_n^* \leq \frac{k - n\mu + \frac{1}{2}}{\sqrt{n\sigma^2}}\right)$$

where S_n^* is standardized S_n with $\mu = 0$ and $\sigma^2 = 1$.

Substituting $\mu = \frac{p}{2}$ and $\sigma^2 = \frac{(p^2-1)}{12}$ gives us:

$$\frac{k - n\mu \pm \frac{1}{2}}{\sqrt{n\sigma^2}} = \frac{k - \frac{np}{2} \pm \frac{1}{2}}{\sqrt{n\frac{p^2-1}{12}}} = \frac{\sqrt{3}(2k - np \pm 1)}{\sqrt{n(p^2-1)}}$$

For n large enough (as in our case, $n > 30$) we can use Central Limit Theorem (C.L.T.) to approximate S_n^*. Therefore, combining previous substitution we have:

$$P\left(\frac{k - n\mu - \frac{1}{2}}{\sqrt{n\sigma^2}} \leq S_n^* \leq \frac{k - n\mu + \frac{1}{2}}{\sqrt{n\sigma^2}}\right) = P\left(\frac{\sqrt{3}(2k - np - 1)}{\sqrt{n(p^2-1)}} \leq S_n^* \leq \frac{\sqrt{3}(2k - np + 1)}{\sqrt{n(p^2-1)}}\right) \approx$$

$$\approx NA\left(\frac{\sqrt{3}(2k - np - 1)}{\sqrt{n(p^2-1)}}, \frac{\sqrt{3}(2k - np + 1)}{\sqrt{n(p^2-1)}}\right)$$

where $NA(a, b)$ denotes an area under the graph of the standard normal density between a and b. Rewriting the original sum gives us:

$$P(S_n^1 = S_n^2) = \sum_{k=0}^{np} P(S_n = k)^2 \approx \sum_{k=0}^{np} NA\left(\frac{\sqrt{3}(2k - np - 1)}{\sqrt{n(p^2-1)}}, \frac{\sqrt{3}(2k - np + 1)}{\sqrt{n(p^2-1)}}\right)^2$$

We notice that the following is always true:

$$NA(a-\Delta, a+\Delta) < 2\Delta \max(\phi(x)) \ = 2\Delta \frac{1}{\sqrt{2\pi}}$$

where $\phi(x)$ is standard normal density function.

In our case $\Delta = \frac{\sqrt{3}}{\sqrt{n(p^2-1)}}$, therefore

$$NA\left(\frac{\sqrt{3}(2k-np-1)}{\sqrt{n(p^2-1)}}, \frac{\sqrt{3}(2k-np+1)}{\sqrt{n(p^2-1)}}\right) < 2\frac{\sqrt{3}}{\sqrt{n(p^2-1)}}\frac{1}{\sqrt{2\pi}}$$

Since $\sum_{k=0}^{np} NA\left(\frac{\sqrt{3}(2k-np-1)}{\sqrt{n(p^2-1)}}, \frac{\sqrt{3}(2k-np+1)}{\sqrt{n(p^2-1)}}\right) < 1$ we have:

$$\sum_{k=0}^{np} NA\left(\frac{\sqrt{3}(2k-np-1)}{\sqrt{n(p^2-1)}}, \frac{\sqrt{3}(2k-np+1)}{\sqrt{n(p^2-1)}}\right)^2 < 2\frac{\sqrt{3}}{\sqrt{n(p^2-1)}}\frac{1}{\sqrt{2\pi}} = \frac{\sqrt{6}}{\sqrt{\pi n(p^2-1)}} \approx$$

$$\approx \frac{\sqrt{6}}{p\sqrt{\pi n}}$$

Summarizing:

$$P(S_n^1 = S_n^2) < \frac{\sqrt{6}}{p\sqrt{\pi n}}$$

As we can see, the probability of having two symmetric paths of the same length, approaches zero as n, the number of links of the path grows. For typical values of $p = 2^{52}$ and $n = 30$, $P(S_n^1 = S_n^2) < 6 \times 10^{-17}$.

References

Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 782–793, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics. 28, 85

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 230–241, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20661-0. URL http://dl.acm.org/citation.cfm?id=2008623.2008645. 3, 18, 27, 30, 66, 84, 86

Ittai Abraham, Daniel Delling, AndrewV. Goldberg, and RenatoF. Werneck. Hierarchical hub labelings for shortest paths. In Leah Epstein and Paolo Ferragina, editors, *Algorithms – ESA 2012*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33089-6. doi: 10.1007/978-3-642-33090-2_4. URL http://dx.doi.org/10.1007/978-3-642-33090-2_4. 28

Leonid Antsfeld. Shortest paths in networks. *The International Conference on Automated Planning and Scheduling (ICAPS, Doctoral Consortium)*, 2013. 18

Leonid Antsfeld and Toby Walsh. Finding optimal paths in multi-modal public transportation networks using hub nodes and transit algorithm. *3rd workshop on Artificial Intelligence and Logistics (AILOG)*, pages 7–11, 2012a. 99

Leonid Antsfeld and Toby Walsh. Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm. *19th world Congress on Intelligent Transport Systems, Vienna*, 2012b. 100

Leonid Antsfeld and Toby Walsh. Incremental updating of the transit algorithm. *Vehicle Routing and Logistics Optimization (VeRoLog)*, page 13, 2012c. 18, 62, 141

Leonid Antsfeld, Daniel Harabour, Phil Kilby, and Toby Walsh. Transit routing on video game maps. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, page 2, 2012. 118, 138

Leonid Antsfeld, Phil Kilby, Andrew Verden, and Toby Walsh. Fast shortest path queries in road networks. In *European Conference on Operational Research (EURO)*, 2013. 18

Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. *CoRR*, abs/1302.5611, 2013. 27, 29, 30, 66, 85

Hannah Bast. Car or public transport - two worlds. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 355–367. 2009. ISBN

978-3-642-03455-8. doi: 10.1007/978-3-642-03456-5_24. URL http://dx.doi.org/10.1007/978-3-642-03456-5_24. 94

Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th annual European conference on Algorithms: Part I*, ESA'10, pages 290–301, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15774-2, 978-3-642-15774-5. URL http://portal.acm.org/citation.cfm?id=1888935.1888969. 97, 102, 104, 109, 115

Hannah Bast, Mirko Brodesser, and Sabine Storandt. Result diversity for multimodal route planning. *ATMOS - 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pages 123–136, 2013. 99

Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Mller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato Werneck. Route planning in transportation networks. *Microsoft, Technical Report*, 2014. 86, 87, 115

Holger Bast, Stefan Funke, and Domagoj Matijevic. Transit - ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge*, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.7223. 2, 3, 18, 25, 29, 35, 44, 52, 73, 85, 91, 99, 102, 142, 144

Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. *Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2007. 26, 91, 99

Reinhard Bauer and Daniel Delling. Sharc: Fast and robust unidirectional routing. *J. Exp. Algorithmics*, 14:2.4–2.29, January 2010. ISSN 1084-6654. doi: 10.1145/1498698.1537599. URL http://doi.acm.org/10.1145/1498698.1537599. 24

Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing speed-up techniques is hard. In Tiziana Calamoneri and Josep Diaz, editors, *Algorithms and Complexity*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer Berlin Heidelberg, 2010a. ISBN 978-3-642-13072-4. doi: 10.1007/978-3-642-13073-1_32. URL http://dx.doi.org/10.1007/978-3-642-13073-1_32. 24

Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *J. Exp. Algorithmics*, 15:2.3, March 2010b. ISSN 1084-6654. doi: 10.1145/1671970.1671976. URL http://doi.acm.org/10.1145/1671970.1671976. 24, 27, 29, 89

Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating time-dependent multi-criteria timetable information is harder than expected. In Jens Clausen and Gabriele Di Stefano,

editors, *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, volume 12 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-11-8. doi: http://dx.doi.org/10.4230/OASIcs.ATMOS.2009.2148. URL http://drops.dagstuhl.de/opus/volltexte/2009/2148. 95

Yngvi Björnsson and Kári Halldórsson. Improved heuristics for optimal path-finding on game maps. In *AIIDE*, pages 9–14, 2006. 121, 132

A Botea, M Müller, and J Schaeffer. Near optimal hierarchical path-finding. *J. Game Dev.*, 1(1):7–28, 2004. 119, 132

Adi Botea. Ultra-fast optimal pathfinding without runtime search. page 122, 2011. 121, 123, 127, 132, 135, 142

Adi Botea. Fast, optimal pathfinding with compressed path databases. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *Symposium on Combinatorial Search (SOCS)*, page 204. AAAI Press, 2012. 121, 127, 128

Gerth Stlting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92(0):3 – 15, 2004. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2003.12.019. URL http://www.sciencedirect.com/science/article/pii/S1571066104000040. 93

Alex Champandard. Modern pathfinding techniques. 2009. URL http://aigamedev.com/. 124

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001. ISBN 0-262-03293-7. 9, 12, 14, 16, 19, 37, 53, 59, 69, 74

Thomas Pajor Daniel Delling and Renato F. Werneck. Round-based public transit routing. *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140, 2012. 98

Daniel Delling and RenatoF. Werneck. Faster customization of road networks. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38526-1. doi: 10.1007/978-3-642-38527-8_5. URL http://dx.doi.org/10.1007/978-3-642-38527-8_5. 25

Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating multi-modal route planning by access-nodes. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA09), Lecture Notes in Computer Science*, pages 587–598. Springer, 2009a. 96

Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In Jrgen Lerner, Dorothea Wagner, and KatharinaA. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer Berlin

Heidelberg, 2009b. ISBN 978-3-642-02093-3. doi: 10.1007/978-3-642-02094-0_7. URL http://dx.doi.org/10.1007/978-3-642-02094-0_7. 18, 94

Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, , and Renato F. Werneck. Graph partitioning with natural cuts. Technical report, Microsoft, 2010. 25

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 376–387, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20661-0. URL http://dl.acm.org/citation.cfm?id=2008623.2008657. 25

Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing and evaluating multimodal journeys. Technical Report 2012-20, Faculty of Informatics, Karlsruhe Institut of Technology, 2012. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000030757. 98

Daniel Delling, AndrewV. Goldberg, and RenatoF. Werneck. Hub label compression. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 18–29. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38526-1. doi: 10.1007/978-3-642-38527-8_4. URL http://dx.doi.org/10.1007/978-3-642-38527-8_4. 28, 86

Camil Demetrescu, Andrew V Goldberg, Johnson, and David S. The shortest path problem. *9th DIMACS implementation challenge*, 2009. URL http://www.dis.uniroma1.it/challenge9/. 40, 80

Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-constrained multi-modal route planning. In *ALENEX*, pages 118–129, 2012. 97

Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. URL http://jmvidal.cse.sc.edu/library/dijkstra59a.pdf. 19

Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 347–361, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-68548-0, 978-3-540-68548-7. URL http://portal.acm.org/citation.cfm?id=1788888.1788914. 95

Jochen Eisner and Stefan Funke. Transit nodes - lower bounds and refined construction. *ALENEX*, pages 141–149, 2012. 29, 84

Ariel Felner and Nathan R. Sturtevant. Abstraction-based heuristics with true distance computations. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2009. 132

W. Fernandez de la Vega, V.Th. Paschos, and R. Saad. Average case analysis of a greedy algorithm for the minimum hitting set problem. In Imre Simon, editor, *LATIN '92*, volume 583 of *Lecture Notes in Computer Science*, pages 130–138. Springer Berlin Heidelberg, 1992. ISBN 978-3-540-55284-0. doi: 10.1007/BFb0023824. URL http://dx.doi.org/10.1007/BFb0023824. 16

Robert Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Diplomarbeit am Karlsruher Institut fr Technologie*, 2008. 24

Robert Geisberger. Contraction of timetable networks with realistic transfers. In *Workshop on Experimental and Efficient Algorithms*, pages 71–82, 2009. doi: 10.1007/978-3-642-13193-6_7. 96

Robert Geisberger. Advanced route planning in transportation networks. *Dissertation am Karlsruher Institut fr Technologie*, 2011. 20

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Workshop on Experimental and Efficient Algorithms*, pages 319–333, 2008. URL http://algo2.iti.kit.edu/schultes/hwy/contract.pdf. 24, 34, 75

Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. ISBN 0-89871-585-7. URL http://portal.acm.org/citation.cfm?id=1070432.1070455. 21

Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40, 2005. 21

Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better landmarks within reach. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 38–51, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72844-3. URL http://dl.acm.org/citation.cfm?id=1768570.1768576. 22

A.V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *ALENEX*, pages 129–143, 2006. 22

Meir Goldenberg, Ariel Felner, Nathan Sturtevant, and JonathanSchaeffer. Portal-based true-distance heuristics for path finding. In *SoCS*, 2010. 120, 132

Transit Google. *http://www.google.com/intl/en/landing/transit*. 91

R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *In Proc. 6th International Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004. 22

D. Harabor and Al. Grastien. Online graph pruning for pathfinding on grid maps. In *25th Conference on Artificial Intelligence (AAAI-11)*, 2011. 122, 124, 132, 135, 143

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *SIGART Bull.*, pages 28–29, December 1972. ISSN 0163-5719. doi: http://doi.acm.org/10.1145/1056777.1056779. URL http://doi.acm.org/10.1145/1056777.1056779. 19

M. Hilger. Accelerating point-to-point shortest path computations in large scale networks. *Diploma Thesis, Technische Universitt Berlin,*, 2007. 22

Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. In CelsoC. Ribeiro and SimoneL. Martins, editors, *Experimental and Efficient Algorithms*, volume 3059 of *Lecture Notes in Computer Science*, pages 269–284. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22067-1. doi: 10.1007/978-3-540-24838-5_20. URL http://dx.doi.org/10.1007/978-3-540-24838-5_20. 20

Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *J. Exp. Algorithmics*, 13:5:2.5–5:2.26, February 2009. ISSN 1084-6654. doi: 10.1145/1412228.1412239. URL http://doi.acm.org/10.1145/1412228.1412239. 25

Xeon5650 Intel. URL http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI. 18, 40, 80

Denes Konig. Grafok es matrixok. *Matematikai es Fizikai Lapok*, 38:116–119, 1931. 15

Dingxiong Deng Gao Cong Diwen Zhu Lingkun Wu, Xiaokui Xiao and Shuigeng Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *The 38th International Conference on Very Large Data Bases (PVLDB)*, pages 406–417, 2012. 34, 45, 73

Alister Justin Marcon. Flows in networks: An algorithmic approach. *Master Thesis*, 2012. 58

F. Margot. Symmetry in integer linear programming. In M. Jümnger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, editors, *50 Years of Integer Programming*, pages 647–681. Springer, 2009. 124

Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal directed shortest path queries using precomputed cluster distances. In Carme lvarez and Mara Serna, editors, *Experimental Algorithms*, volume 4007 of *Lecture Notes in Computer Science*, pages 316–327. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-34597-8. doi: 10.1007/11764298_29. URL http://dx.doi.org/10.1007/11764298_29. 23

Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *J. Exp. Algorithmics*, Volume 14: 3.2–3.27, January 2010. ISSN 1084-6654. doi: 10.1145/1498698.1564502. URL http://doi.acm.org/10.1145/1498698.1564502. 23, 120

Karl Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–115, 1927. 15

Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup Dijkstra's algorithm. *J. Exp. Algorithmics*, 11, February 2007. ISSN 1084-6654. doi: http://doi.acm.org/10.1145/1187436.1216585. URL http://doi.acm.org/10.1145/1187436.1216585. 21, 89

Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria pareto search. In *Proceedings of the 4th international Dagstuhl, ATMOS conference on Algorithmic approaches for transportation modeling, optimization, and systems*, ATMOS'04, pages 246–263, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74245-X, 978-3-540-74245-6. URL http://portal.acm.org/citation.cfm?id=1961366.1961381. 94

Matthias Müller-Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, WAE '01, pages 185–198, London, UK, 2001. Springer-Verlag. ISBN 3-540-42500-4. URL http://portal.acm.org/citation.cfm?id=647258.720925. 93, 94

Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: models and algorithms. In *Proceedings of the 4th international Dagstuhl, ATMOS conference on Algorithmic approaches for transportation modeling, optimization, and systems*, ATMOS'04, pages 67–90, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74245-X, 978-3-540-74245-6. URL http://portal.acm.org/citation.cfm?id=1961366.1961370. 93, 94, 112

K. Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154 – 166, 1995. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/0377-2217(94)E0349-G. URL http://www.sciencedirect.com/science/article/pii/0377221794E0349G. 93

TransportInfo NSW. *http://www.131500.com.au/*. 91

Ariel Orda and Raphael Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *J. ACM*, 37(3):607–625, July 1990. ISSN 0004-5411. doi: 10.1145/79147.214078. URL http://doi.acm.org/10.1145/79147.214078. 93

OSM. Project OSRM. URL http://project-osrm.org/. 40, 80

N. Pochter, A. Zohar andJ. S. Rosenschein, and A. Felner. Search space reduction using swamp hierarchies. In *AAAI*, 2010. 121

Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Using swamps to improve optimal pathfinding. In *AAMAS*, pages 1163–1164, 2009. 121, 132

Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, May 1990. ISSN 0895-4798. doi: 10.1137/0611030. URL http://dx.doi.org/10.1137/0611030. 59

Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Experimental comparison of shortest path approaches for timetable information. In *Algorithm Engineering and Experimentation*, pages 88–99, 2004. 93

D. Schultes C. Vetter R. Geisberger, P. Sanders. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, Volume 46, Issue 3:388–404, 2012. 18, 24, 39, 73, 99

Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, pages 568–579, 2005. doi: 10.1007/11561071_51. 22, 23

Peter Sanders and Dominik Schultes. Robust, almost constant time shortest-path queries on road networks. In *9th DIMACS Implementaional Challenge*, 2006a. 18, 26, 29, 85

Peter Sanders and Dominik Schultes. Engineering highway hierarchies. In *European Symposium on Algorithms (ESA)*, pages 804–816, 2006b. doi: 10.1007/11841036_71. 22, 23

Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile route planning. In Dan Halperin and Kurt Mehlhorn, editors, *European Symposium on Algorithms (ESA)*, volume 5193 of *Lecture Notes in Computer Science*, pages 732–743. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87743-1. doi: 10.1007/978-3-540-87744-8_61. URL http://dx.doi.org/10.1007/978-3-540-87744-8_61. 80

Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. Efficient query processing on spatial networks. In *ACM international symposium on Advances in Geographic Information Systems (GIS)*, pages 200–209, 2005. 121

D. Schultes. Route planning in road networks. *PhD thesis, University Karlsruhe*, 2008. 23, 29, 66

Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In *In Proc.*

6th Workshop on Experimental and Efficient Algorithms. LNCS, pages 66–79. Springer, 2007. 67

Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *J. Exp. Algorithmics*, 5:110–123, December 2000. ISSN 1084-6654. doi: http://doi.acm.org/10.1145/351827.384254. URL http://doi.acm.org/10.1145/351827.384254. 93

Sabine Storandt. Contraction hierarchies on grid graphs. In IngoJ. Timm and Matthias Thimm, editors, *KI 2013: Advances in Artificial Intelligence*, volume 8077 of *Lecture Notes in Computer Science*, pages 236–247. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40941-7. doi: 10.1007/978-3-642-40942-4_21. URL http://dx.doi.org/10.1007/978-3-642-40942-4_21. 122

N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012a. URL http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf. 123, 131

Nathan Sturtevant. Grid-based path planning competition results. 2012b. URL http://www.movingai.com/GPPC/GPPC.pdf. 135

Nathan R. Sturtevant. Memory-efficient abstractions for pathfinding. In *AIIDE*, pages 31–36, 2007. 120, 132

Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, pages 1392–1397, 2005. 120

Nathan R. Sturtevant and Robert Geisberger. A comparison of high-level approaches for speeding pathfinding. In *AIIDE*, pages 76–82, 2010. 122

Nathan R. Sturtevant, Ariel Felner, Max Barrer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In *IJCAI*, pages 609–614, 2009. 120

Dorothea Wagner and Thomas Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *European Symposium on Algorithms*, pages 776–787, 2003. 20

Dorothea Wagner and Thomas Willhalm. Speed-up techniques for shortest-path computations. In Wolfgang Thomas and Pascal Weil, editors, *STACS 2007*, volume 4393 of *Lecture Notes in Computer Science*, pages 23–36. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-70917-6. doi: 10.1007/978-3-540-70918-3_3. 18