Shortest Paths in Networks

Leonid Antsfeld
NICTA and UNSW, Sydney, Australia
leonid.antsfeld@nicta.com.au

Introduction

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. 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 multi-objectiveness, user preferences, etc. Finally we extend our technique to the completely new domain of grid networks, where one of the challenges is to deal with path symmetries.

Road Networks

Finding a shortest path between two nodes in a road network is a classic problem with many real-world applications. The need to find a shortest path arises, for example, in navigation, vehicle routing problems and public transport service queries. The well known Dijkstra algorithm (Dijkstra 1959) will find a shortest path between two nodes in $O(m + n \log m)$, where n is the number of nodes and m is the number of edges in the graph. For road networks, where degree of an edge is bounded by a small constant, the time complexity of finding the shortest path between two nodes becomes $O(n \log n)$. For a large scale network, it may take more than a second to find a solution, which is a prohibitively slow for many real world applications.

Related work

There has been much work on shortest path problem in the literature. We therefore only give a brief overview here of the most prominent results. For a more detailed review, we direct the reader to (Goldberg and Harrelson 2005; Sanders and Schultes 2006b; Abraham et al. 2011). Generally shortest paths algorithms can be divided into two categories - purely search algorithms and those that require some preprocessing in order to speed up the search later. Well known and widely used pure search algorithms include Dijkstra (Dijkstra 1959), A* (Hart, Nilsson, and Raphael 1972) and bidirectional search (Nicholson 1966). For the algorithms that use preprocessing, we highlight the most recent and related results to our work. Contraction Hierarchies (CH) by Geisberger et al. (Geisberger et al. 2008) is a speed up technique solely based on the concept of node contraction. Contracting a node v means replacing shortest paths going through v by shortcuts. This work came as an enhancement of their previous Highway Hierarchies (HH) technique (Sanders and Schultes 2005; 2006a), which exploited inherent hierarchy of the road network. In 2006 H. Bast at all. introduced the concept of transit node routing (Bast, Funke, and Matijevic 2006), a technique where a shortest path query in a road network is replaced by a small number of lookups in some precomputed tables. The transit nodes were determined using a grid and exploiting geographical position of nodes. As was noted by P. Sanders and D. Schultes in (Sanders and Schultes 2006b) initially their CH approach was beaten by the original transit routing (Bast, Funke, and Matijevic 2006) by a factor of 40. Until recently, the most competitive results by P. Sanders and D. Schultes was to combine HH with the transit node routing. In (Sanders and Schultes 2006b) they report query times in the range 5-20 μs for all type of queries, whilst in (Bast, Funke, and Matijevic 2006) they report query times around 10 μs for 99% types of queries, where both results were benchmarked on undirected graphs. Recently, I. Abraham introduced a new labeling algorithm, which by using very fine-tuned implementation may achieve query times measured in ns (Abraham et al. 2011).
We have investigated the TRANSIT approach. Despite its initial relatively good performance, there is still a lot of room for improvement and extensions of the idea. In addition Transit node routing is appealing due to its quite intuitive approach and relatively simple implementation.

TRANSIT routing

The TRANSIT 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 edges that are common to many shortest paths (highways are a natural example). The endpoints of such edges constitute a set of so-called "transit nodes".TRANSIT proceeds in two phases: (i) an offline precomputation phase and (ii) an online query phase.

Offline Precomputation Phase

TRANSIT's offline precomputation phase consists of two main steps. At the first step we identify the aforesaid transit nodes and in the second step we build a database of exact distances between every node and its associated transit

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

nodes, as well as between all transit nodes. We will describe each step in turn.

Identifying Transit Nodes TRANSIT starts by dividing an input map into a grid of equal-sized cells. To achieve this TRANSIT 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 of sizes $i \times i$ and $o \times o$ respectively, having C in the center as depicted in Fig. 1 below.

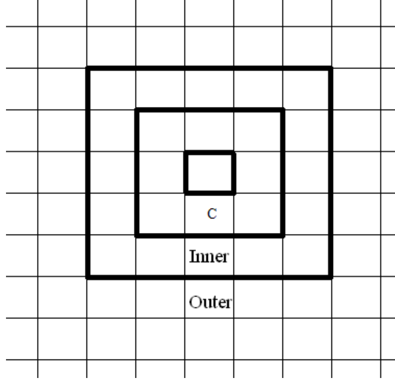

Figure 1: Example of the TRANSIT grid; also cells and inner and outer squares.

The size of the squares C, I and O can be arbitrary without compromising correctness. Their exact values however will directly impact factors such as TRANSIT's preprocessing time, storage requirements and online query times. In (Antsfeld et al. 2012) was discussed how those parameters can be tuned and the tradeoff between precomputation time, memory requirements and finally the query time.

In what follows we will compute shortest paths between nodes that reside on border of C and O and choose as transit nodes one of the endpoints of the edges that cross the border of I. More precisely, 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 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. We associate every node inside C with the set of transit nodes of C. Next, we iterate over all cells and similarly identify transit nodes for every other cell.

Computation and Storage of Distances Once we have identified all transit nodes, we store for every node on the map, the shortest distance from this node to all its associated transit nodes. Recall from the previous section that every such node $v \in V$ is associated with the set of transit nodes that were found for its cell. In addition we also compute and store the shortest distance from each transit node to every other transit node. In an undirected map it is enough to compute and store costs only in one direction.

Local Search Metrics

Two necessary conditions for TRANSIT to be correct are that any source node src and a destination node dst (i) outside outer squares of each other and (ii) their corresponding inner squares do not overlap. Therefore, we define a local search metric to be the size of the inner square I plus the distance from I to the outer square O, which equivalent to $(i + o)/2$. Two nodes for which horizontal or vertical distance is greater than local search metric are considered to be "far away" and a query between them called a *global* query. All other queries are *local*.

Online Query Phase

For every global query from src to dst 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 local queries the inventors of TRANSIT suggested to apply any efficient search algorithm; A* for example (Bast, Funke, and Matijevic 2006). In (Antsfeld et al. 2012) we have presented a new, more efficient technique, using CPDs for dealing with local queries.

Shortest Path Extraction

Until now, not much work have been done on efficient path extraction using TRANSIT precomputed databases. The intuitive and somewhat naive way would be simply to store the paths in addition to the distances. This approach would incur prohibitively large memory requirements. To avoid this we can reconstruct the path by performing a series of repeated distance queries of TRANSIT. In the original paper the authors suggested first finding the next adjacent node to the source on the shortest path and then iteratively applying TRANSIT query from that node to extract the full path (Bast, Funke, and Matijevic 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. Another improvement was suggested in (Bast et al. 2007) by observing that actually the transit node associated with destination, T_{dst}, is not changing for all the sequential queries. This fact was exploit in order to avoid searching for T_{dst} at every iteration, but rather reuse it. More sophisticated improvement can be achieved by noticing that for many sequental queries T_{src} is also staying the same. Therefore the sequential TRANSIT query can avoid searching for optimal T_{src} and T_{dst} all together and immediately fetch the corresponding entry, $src \rightsquigarrow T_{src}$, which will guide us to the next move. In later section we will present in details a novel, more efficient approach for the shortest path extraction.

Complexity analysis

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 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 start assuming 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 I be a square centered on C consisting of some constant number of cells. In the worst case, the number of transit nodes for cell C equals the number of border nodes of square I, 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 zero 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 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.

Incremental Updating

One of the main drawbacks of the TRANSIT algorithm is that it 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). In our research we try to update this precomputed information in an efficient manner when the network changes (Antsfeld and Walsh 2012c).

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

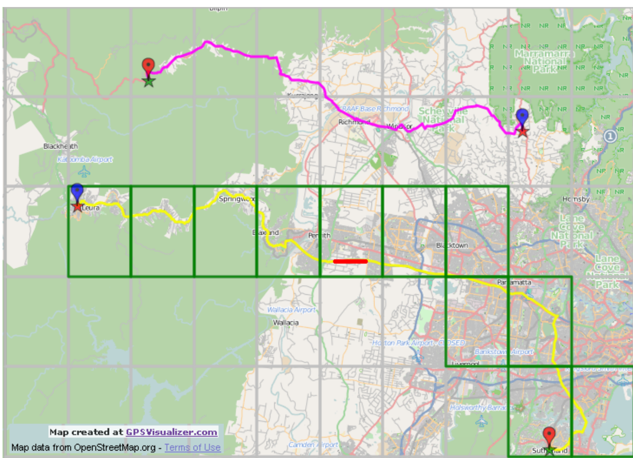

Figure 2: Example of the grid and two shortest paths, only one of which going via a blocked road (colored in red)

The traditional approach would be to precompute everything from scratch. As the example shows, there are many shortest paths (including the path above) that not affected by this change. Obviously, more refined approach could ignore these shortest paths, and save this computational effort.

The Algorithm We suppose that travel time over a link e increases because, say, of roadworks. If we knew all the pairs of nodes whose shortest path contains e, we could recompute and update the precomputed tables only for these specific pairs. In this case the recomputation would be minimal. In the original Transit algorithm, we can store this information during the precomputation stage of the three tables *node-to-transit*, *transit-to-transit* and *transit-to-node*. However, we would now need to store the 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, we stored this information for pairs of cells rather than pairs of nodes. The worst case storage for this 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. Hence, in order to reduce the storage requirements even more, instead of link e, we considered the cell C_e containing this link and stored for every such cell, a list of pairs of cells $\mathcal{C}_e = (C_{src}, C_{dst})$, such that for every pair in $\mathcal{C}_e$ there exist a pair of nodes (src, dst), with src inside cell C_{src} and dst inside cell C_{dst} and e is on a shortest path between src and dst. The asymptotical worst case requirement for this is reduced to $O(k^3)$ which was practical to implement. This reduction comes with the cost of additional, somewhat redundant, recomputation work that we have to do during the update stage.

Public Transportation Networks

We now turn to another type of networks, public transportation networks. On one hand, the public transportation network, somewhat resembles the structure of a road network, therefore it looked natural to extend and adjust the TRANSIT to public transportation network. On the other hand, the nature of public networks is very different from roads in many ways, e.g., links travel times are time/day dependent. 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 TRANSIT to a time expanded graph does not scale. To deal with this problem, we will apply it to a new, two-layers model of the public 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 according to user preferences.

Related work

As we discussed before, in recent years several algorithms have been developed that use precomputed information to obtain a shortest path in a road network in a few microseconds. However there has been less progress for public transport networks. In (Bast 2009) H. Bast discusses why finding shortest paths in public transport 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. First of all public transport network are inherently multi-modal. i.e. the are at least two different means of transport. Other issues we need to consider are *time dimension*, *transfer time safety buffers*, *tickets cost*, *operating days*, etc. The recent and the most prominent result in this area is by H. Bast et al. (Bast et al. 2010). They also use a notion of *hub stations*, but in completely different way. H. Bast et al perform Dijkstra searches from a random sample of source stations and choose as *hubs* stations those that are on the the largest number of shortest paths. 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. 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.
In our research we are trying to solve multi-objective multi-modal shortest path problem using the idea inspired by TRANSIT. In (Antsfeld and Walsh 2012a) we reported an initial attempt to do so, by applying TRANSIT algorithm (Bast, Funke, and Matijevic 2006) to an improved time expanded graph of the multi-modal public network. In (Antsfeld and Walsh 2012b) we described an improvement, which is more intuitive, has much less memory requirements and precomputation time. In addition we showed how to provide multiple results in the real world incorporating user preferences.

Modeling the Network There are two main approaches to model a public transport networks, known as *Time-Dependent* and *Time-Expanded* models. For an exhaustive description of the models and existing techniques we address the readers to (Müller-Hannemann et al. 2007; Pyrga et al. 2004). We suggest a new model which is an enhanced combination of the two. Our model consist of two layers: *station graph* and *events graph*. The *station graph* nodes are the stations and it has two types of links *station links* and *walking links*. In our experiments we assume that we can walk from every station to every other station within a 10 minutes walking radius. 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*. Typically the *Time-Expanded* model has three types of nodes: *arrival node*, *departure node* and *transfer node*. Eliminating *transfer nodes* and all the links from *transfer nodes* to *departure nodes* in the typical *Time-Expanded* model and connecting *arrival* and *departure* nodes directly allowed us to reduce space requirements by 30%. In addition to the storage saving this modification also speeds up precomputation time. The described graph is illustrated in Figure 2. Modeling the public transport in this way allows us to treat all different modes as a single mode.

Hub Nodes Consider an example in Figure 3 below.

Assume 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*. We observed that in reality (for Sydney and NSW public transportation networks), only relatively small portion of all stations, $15\% - 20\%$, are hubs.

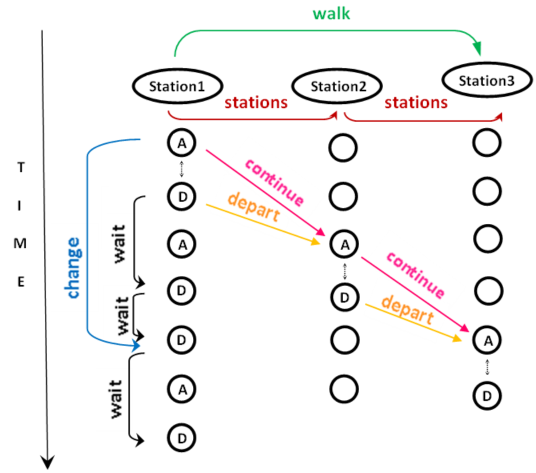

Figure 2: The two layered, time expanded graph. The first layer consist of three stations connected with *station links* and *walking links*. The second layer consist of arrival (A) and departure (D) events connected with *depart links*, *continue links*, *change links* and *wait links*.

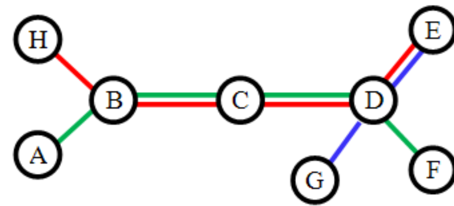

Figure 3: Example of hub and non-hub nodes

We notice that if we know the optimal route between any hub station to any other hub station, then a shortest route between any two nodes (not necessarily hubs) can be found immediately as described in the next Section.

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.

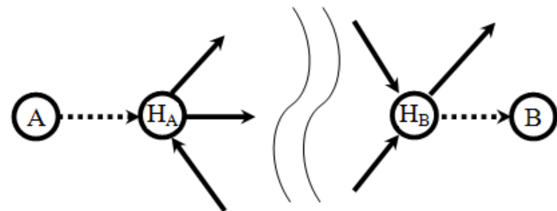

Figure 4: Example of query between two non hub stations

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, 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 (cause 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 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 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) and 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. In (Antsfeld and Walsh 2012b) we describe in details how we deal with real world scenarios, such as multi-objectiveness, route alternatives and user preferences.

Grid Networks

We now turn to a third type of network, a grid network as found in computer video games and robotics. One of the challenges in this domain, was to deal with uniform-cost path symmetries, which is commonly found in grid networks, but rarely in road networks (Harabor, Botea, and Kilby 2011).

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*. These efforts range from: **(i)** abstraction-based near-optimal techniques such as (Botea, Müller, and Schaeffer 2004; Sturtevant 2007) **(ii)** precomputation algorithms for improving heuristic estimates; for example (Sturtevant et al. 2009; Goldenberg et al. 2010) **(iii)** online and offline pruning and symmetry breaking methods such as (Björnsson and Halldórsson 2006; Pochter, Zohar, and Rosenschein 2009; Harabor and Grastien 2011) and **(iv)** compressing the entire set of All-Pairs data, aka CPD, as in (Botea 2011; 2012). Almost all involve a speed vs. memory tradeoff and typically deliver improvements in the range of one or (as in the case of (Botea 2011)) two orders of magnitude.
Our research tries to bridge the gap between very efficient path finding on road networks and still developing path finding in grid networks (Antsfeld et al. 2012). We present a new algorithm for shortest-path extraction, and distance-query answering in grid video-game maps. Our method combines strengths of the TRANSIT (Bast, Funke, and Matijevic 2006) and CPD (Botea 2011) methods.

TRANSIT with CPD

The basic idea of our approach is to break a long shortest path (i.e. $src \rightsquigarrow dst$ is a global query) to 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 CPD.

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

Query We start with invoking the basic TRANSIT query. We remember that every global shortest path, (i.e. it is longer than a local search metric) 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 CPDs query to extract 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 srs. We can improve this even more, by noticing that when running sequential quires of TRANSIT T_{dst} and the distance from T_{dst} to dst are not changing. We can exploit this fact, by reusing T_{dst}. This will speed up the time complexity of subsequent TRANSIT queries from quadratic to linear by the number of the transit nodes.

CHAT

One of the drawbacks of TRANSIT is that the algorithm does not exploit any topological information of the underlying network, like bottlenecks. 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 or a tunnel. In addition it is not clear what grid size we should choose, since it directly affects the tradeoff between precomputation time, storage space and query time.
Our new algorithm, CHAT (**C**luster, **H**ierarchify **a**nd Hi**t**) addresses these issues. CHAT has the same intuition as

TRANSIT, but identifies those key nodes in a more precise manner. 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. Similarly, 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. Unlike TRANSIT our new algorithm successfully identifies those entrance/exit key nodes, which we will refer as *access* nodes (as access to the highways network).

The Algorithm

In the first stage, we identify all the nodes which have at least one endpoint as a highway, we call them *major* (a.k.a. important) nodes. Next, we divide our map to clusters, using the K-means heuristic. Then for every cluster, using BFS, we identify the furthest $major$ node, such that there is no other $major$ node on a shortest path from this cluster. The distance between the cluster and this node will precisely define a notion of "far away" for this particular cluster. Finally, the set of access nodes of a cluster is defined as a hitting set of all $major$ nodes which are on a "far away" shortest paths originating from the convex hull of the cluster.

The query is performed similar to the original TRANSIT. For every src and dst we fetch their cluster *access* nodes and find the path with minimal sum of the three subpaths. In our experiments, the average number of access nodes per cluster is around 5, which translates to a few table lookups.

Experiments

We tested our algorithm on Sydney, New South Wales and Australia road networks using different parameters, comparing the original TRANSIT with our new algorithm. For all instances, the precomputed storage was reduced by $10\% - 30\%$, while the final query time was reduced by $4 - 9$ times. Whilst these results are very promising, there are further interesting questions need to be answered. In particular, it would be interesting to see the effect of different clustering techniques.

Conclusions and Future work

We have reported a number of novel approaches for shortest path finding in three important domains (road, public transport and grid networks) by exploiting the specific properties of the underlying network, using the basic idea of TRANSIT (Bast, Funke, and Matijevic 2006). In addition, we have introduced a novel algorithm, CHAT, for finding shortest paths in road networks, which addresses the drawbacks of TRANSIT. It requires less memory and allows faster queries. An interesting direction would be to extend transit/access node routing to multi-objective problems. In this case, theoretically we will need to precalculate and store all non-dominated solutions, which may not be feasible in practice.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, 230–241. Berlin, Heidelberg: Springer-Verlag.

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

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

Antsfeld, L., and Walsh, T. 2012c. Incremental updating of the transit algorithm. *Vehicle Routing and Logistics Optimization (VeRoLog)*.

Antsfeld, L.; Harabour, D.; Kilby, P.; and Walsh, T. 2012. Transit routing on video game maps. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Bast, H.; Funke, S.; Matijevic, D.; Sanders, P.; and Schultes, D. 2007. In transit to constant time shortest-path queries in road networks. *Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*.

Bast, H.; Carlsson, E.; Eigenwillig, A.; Geisberger, R.; Harrelson, C.; Raychev, V.; and Viger, F. 2010. 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, 290–301. Berlin, Heidelberg: Springer-Verlag.

Bast, H.; Funke, S.; and Matijevic, D. 2006. Transit ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge*.

Bast, H. 2009. Efficient algorithms. Berlin, Heidelberg: Springer-Verlag. chapter Car or Public Transport–Two Worlds, 355–367.

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *AIIDE*, 9–14.

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

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *AIIDE*.

Botea, A. 2012. Fast, optimal pathfinding with compressed path databases. In Borrajo, D.; Felner, A.; Korf, R. E.; Likhachev, M.; López, C. L.; Ruml, W.; and Sturtevant, N. R., eds., *SOCS*. AAAI Press.

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

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Workshop on Experimental and Efficient Algorithms*, 319–333.

Goldberg, A. V., and Harrelson, C. 2005. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, 156–165. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Goldenberg, M.; Felner, A.; Sturtevant, N.; and JonathanSchaeffer. 2010. Portal-based true-distance heuristics for path finding. In *SoCS*.

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

Harabor, D. D.; Botea, A.; and Kilby, P. 2011. Path symmetries in undirected uniform-cost grids. In *SARA*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. A formal basis for the heuristic determination of minimum cost paths. *SIGART Bull.* 28–29.

Müller-Hannemann, M.; Schulz, F.; Wagner, D.; and Zaroliagis, C. 2007. 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, 67–90. Berlin, Heidelberg: Springer-Verlag.

Nicholson, T. A. J. 1966. Finding the shortest route between two points in a network. *Computer.* 9:275–280.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2009. Using swamps to improve optimal pathfinding. In *AAMAS*, 1163–1164.

Pyrga, E.; Schulz, F.; Wagner, D.; and Zaroliagis, C. D. 2004. Experimental comparison of shortest path approaches for timetable information. In *Algorithm Engineering and Experimentation*, 88–99.

Sanders, P., and Schultes, D. 2005. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, 568–579.

Sanders, P., and Schultes, D. 2006a. Engineering highway hierarchies. In *European Symposium on Algorithms*, 804–816.

Sanders, P., and Schultes, D. 2006b. Robust, almost constant time shortest-path queries on road networks. In *9TH DIMACS Implementaional Challenge*.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.