

Exploring Graph Partitioning for Shortest Path Queries on Road Networks

Theodoros Chondrogiannis
Free University of Bozen-Bolzano
tchond@inf.unibz.it

Johann Gamper
Free University of Bozen-Bolzano
gamper@inf.unibz.it

ABSTRACT

Computing the shortest path between two locations in a road network is an important problem that has found numerous applications. The classic solution for the problem is *Dijkstra's* algorithm [1]. Although simple and elegant, the algorithm has proven to be inefficient for very large road networks. To address this deficiency of Dijkstra's algorithm, a plethora of techniques that introduce some preprocessing to reduce the query time have been proposed. In this paper, we propose *Partition-based Shortcuts* (PbS), a technique based on graph-partitioning which offers fast query processing and supports efficient edge weight updates. We present a shortcut computation scheme, which exploits the traits of a graph partition. We also present a modified version of the bidirectional search [2], which uses the precomputed shortcuts to efficiently answer shortest path queries. Moreover, we introduce the Corridor Matrix (CM), a partition-based structure which is exploited to reduce the search space during the processing of shortest path queries when the source and the target point are close. Finally, we evaluate the performance of our modified algorithm in terms of preprocessing cost and query runtime for various graph partitioning configurations.

Keywords

Shortest path, road networks, graph partitioning

1. INTRODUCTION

Computing the shortest path between two locations in a road network is a fundamental problem and has found numerous applications. The problem can be formally defined as follows. Let $G(V, E)$ be a directed weighted graph with vertices V and edges E . For each edge $e \in E$, a weight $l(e)$ is assigned, which usually represents the length of e or the time required to cross e . A path p between two vertices $s, t \in V$ is a sequence of connected edges, $p(s, t) = \langle (s, v_1), (v_1, v_2), \dots, (v_k, v_t) \rangle$ where $(v_k, v_{k+1}) \in E$, that connects s and t . The shortest path between two vertices s and t is the path $p(s, t)$ that has the shortest distance among all paths that connect s and t .

The classic solution for the shortest path problem is Dijkstra's algorithm [1]. Given a source s and a destination t in a road network G , Dijkstra's algorithm traverses the vertices in G in ascending order of their distances to s . However, Dijkstra's algorithm comes with a major shortcoming. When the distance between the source and the target vertex is high, the algorithm has to expand a very large subset of the vertices in the graph. To address this shortcoming, several techniques have been proposed over the last few decades [3]. Such techniques require a high start-up cost, but in terms of query processing they outperform Dijkstra's algorithm by orders of magnitude.

Although most of the proposed techniques offer fast query processing, the preprocessing is always performed under the assumption that the weights of a road network remain unchanged over time. Moreover, the preprocessing is metric-specific, thus for different metrics the preprocessing needs to be performed for each metric. The recently proposed *Customizable Route Planning* [4] applies preprocessing for various metrics, i.e., distance, time, turn cost and fuel consumption. Such an approach allows a fast computation of shortest path queries using any metric desired by the user, at the cost of some extra space. Moreover, the update cost for the weights is low since the structure is designed such that only a small part of the preprocessed information has to be recomputed. In this paper, our aim is to develop an approach which offers even faster query processing, while keeping the update cost of the preprocessed information low. This is particularly important in dynamic networks, where edge weights might frequently change, e.g., due to traffic jams.

The contributions of this paper can be summarized as follows:

- We present *Partitioned-based Shortcuts* (PbS), a preprocessing method which is based on Customizable Route Planning (CRP), but computes more shortcuts in order to reduce the query processing time.
- We propose the *Corridor Matrix* (CM), a pruning technique which can be used for shortest path queries when the source and the target are very close and the precomputed shortcuts cannot be exploited.
- We run experiments for several different partition configurations and we evaluate our approach in terms of both preprocessing and query processing cost.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we describe in detail the preprocessing phase of our method. In Section 5, we present a modified version of the bidirectional search algorithm. In Section 6, we show preliminary results of an empirical evaluation. Section 7 concludes the paper and points to future research directions.

2. RELATED WORK

The preprocessing based techniques that have been proposed in order to reduce the time required for processing shortest path queries can be classified into different categories [3]. *Goal-directed techniques* use either heuristics or precomputed information in order to limit the search space by excluding vertices that are not in the direction of the target. For example, A^* [5] search uses the Euclidean distance as a lower bound. *ALT* [6] uses precomputed shortest path distances to a carefully selected set of landmarks and produces the lower bound using the triangle inequality. Some goal-directed techniques exploit graph partitioning in order to prune the search space and speed-up queries. *Precomputed Cluster Distances* (PCD) [7] partitions the graph into k components, computes the distance between all pairs of components and uses the distances between components to compute lower bounds. Arc Flags [8] maintains a vector of k bits for each edge, where the i -th bit is set if the arc lies on a shortest path to some vertex of component i . Otherwise, all edges of component i are pruned by the search algorithm.

Path Coherent techniques take advantage of the fact that shortest paths in road networks are often spatially coherent. To illustrate the concept of spatial coherence, let us consider four locations s, s', t and t' in a road network. If s is close to s' and t is close to t' , the shortest path from s to t is likely to share vertices with the shortest path from s' to t' . Spatial coherence methods precompute all shortest paths and use then some data structures to index the paths and answer queries efficiently. For example, Spatially Induced Linkage Cognizance (SILC) [9] use a quad-tree [10] to store the paths. Path-Coherent Pairs Decomposition (PCPD) [11] computes unique path coherent pairs and retrieves any shortest path recursively in almost linear time to the size of the path.

Bounded-hop techniques aim to reduce a shortest path query to a number of look-ups. Transit Node Routing (TNR) [12] is an indexing method that imposes a grid on the road network and re-computes the shortest paths from within each grid cell C to a set of vertices that are deemed important for C (so-called access nodes of C). More approaches are based on the theory of 2-hop labeling [13]. During preprocessing, a label $L(u)$ is computed for each vertex u of the graph such that for any pair u, v of vertices, the distance $dist(u, v)$ can be determined by only looking at the labels $L(u)$ and $L(v)$. A natural special case of this approach is Hub Labeling (HL) [14], in which the label $L(u)$ associated with vertex u consists of a set of vertices (the hubs of u), together with their distances from u .

Finally, *Hierarchical techniques* aim to impose a total order on the nodes as they deem nodes that are crossed by many shortest paths as more important. *Highway Hierarchies* (HH) [15] and its direct descendant *Contraction Hierarchies* (CH) organize the nodes in the road network into a hierarchy based on their relative importance, and create shortcuts among vertices at the same level of the hierarchy. *Arterial Hierarchies* (AH) [16] are inspired by CH, but produce shortcuts by imposing a grid on the graph. AH outperform CH in terms of both asymptotic and practical performance [17]. Some hierarchical approaches exploit graph partitioning to create shortcuts. *HEPV* [18] and *HiTi* [19] are techniques that pre-computes the distance between any two boundary vertices and create a new overlay graph. By partitioning the overlay graph and repeating the process several times, a hierarchy of partitions is created, which is used to process shortest path queries.

The recent *Customizable Route Planning* (CRP) [4] is the closest work to our own. CRP is able to handle various arbitrary metrics and can also handle dynamic edge weight updates. CRP uses PUNCH [20], a graph partitioning algorithm tailored to road networks. CRP pre-computes distances between boundary vertices

in each component and then CRP applies a modified bidirectional search algorithm which expands only the shortcuts and the edges in the source or the target component. The main difference between our approach and CRP is that, instead of computing only shortcuts between border nodes in each component, we compute shortcuts from every node of a component to the border nodes of the same component. The extra shortcuts enable the bidirectional algorithm to start directly from the border nodes, while CRP has to scan the original edges of the source and the target component.

3. PBS PREPROCESSING

The *Partition-based Shortcuts (PbS)* method we propose exploits graph partitioning to produce shortcuts in a preprocessing phase, which during the query phase are used to efficiently compute shortest path queries. The idea is similar to the concept of transit nodes [12]. Every shortest path between two nodes located in different partitions (also termed components) can be expressed as a combination of three smaller shortest paths. Consider the graph in Figure 1 and a query $q(s, t)$, where $s \in C_1$ and $t \in C_5$. The shortest path from s to t can be expressed as $p(s, b_s) + p(b_s, b_t) + p(b_t, t)$, where $b_s \in \{b_1, b_2\}$ and $b_t \in \{b_3, b_4, b_5\}$. Before PbS is able to process shortest path queries, a preprocessing phase is required, which consists of three steps: graph partitioning, in-component shortcut computation and shortcut graph construction.

3.1 Graph Partitioning

The first step in the pre-processing phase is the graph partitioning. Let $G(V, E)$ be a graph with vertices V and edges E . A partition of G is a set $P(G) = \{C_1, \dots, C_k\}$ of connected sub-graphs C_i of G , also referred to as *components* of G . For the set $P(G)$, all components must be disjoint, i.e., $C_1 \cap \dots \cap C_k = \emptyset$. Moreover, let $V_1, \dots, V_{|P(G)|}$ be the sets of vertices of each component. The vertex sets of all components must cover the vertex set of the graph, i.e., $V_1 \cup \dots \cup V_{|P(G)|} = V$. We assign a tag to each node of the original graph, which indicates the component the node is located in. The set of *connecting edges*, $E_C \subseteq E$, is the set of all edges in the graph for which the source and target nodes belong to different components, i.e., $(n, n') \in E$ such that $n \in C_i, n' \in C_j$ and $C_i \neq C_j$. Finally, we define the *border nodes* of a component C . A node $n \in C$ is a border node of C if there exists a connecting edge $e = (n, n')$ or $e = (n', n)$, i.e., n' is not in C . If $e = (n, n')$, n is called *outgoing border node* of C , whereas if $e = (n', n)$, n is called *incoming border node* of C . The set of all border nodes of a graph is referred to as B . Figure 1 illustrates a graph partitioned into five components. The filled nodes are the border nodes. Note that for ease of exposition we use only undirected graphs in the examples.

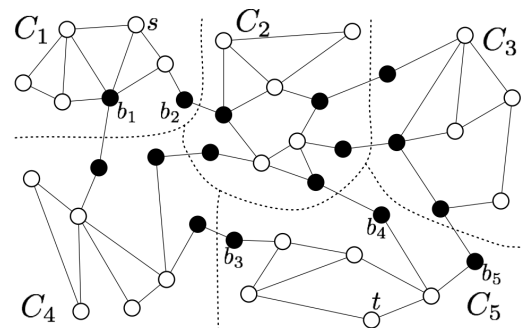


Figure 1: Partitioned graph into five components.

We characterize a graph partition as good if it minimizes the number of connecting edges between the components. However, graph partitioning is an *NP*-hard problem, thus an optimal solution is out of the question [21]. A popular approach is *multilevel graph partitioning* (MGP), which can be found in many software libraries, such as METIS [22]. Algorithms such as PUNCH [20] and *Spatial Partition Clustering* (SPC) [23] take advantage of road network characteristics in order to provide a more efficient graph partitioning. We use METIS for graph partitioning since it is the most efficient approach out of all available ones [24]. METIS requires only the number of components as an argument in order to perform the partitioning. The number of components influences both the number of the in-component shortcuts and the size of the shortcut graph.

3.2 In-component Shortcuts

The second step of the preprocessing phase is the computation of the in-component shortcuts. For each node n in the original graph, we compute the shortest path from the node to every outgoing border node of the component in which n is located. Then we create outgoing shortcuts which abstract the shortest path from n to each outgoing border node. The incoming shortcuts are computed in a similar fashion. Thus, the total number of in-component shortcuts, S , is

$$S = \sum_{i=1}^k N_i \times (|B_{i_{inc}}| + |B_{i_{out}}|),$$

where N_i is the number of nodes in component C_i and $B_{i_{inc}}$, $B_{i_{out}}$ are the incoming and outgoing border nodes of C_i , respectively. Figure 2 shows the in-component shortcuts for a node located in component C_2 .

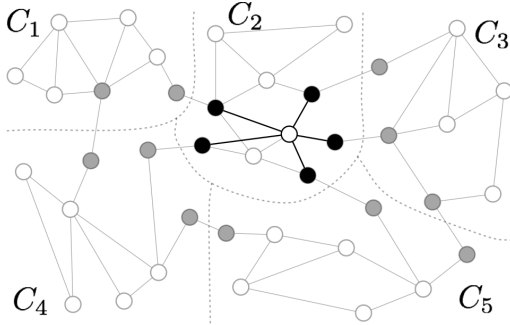


Figure 2: In-component shortcuts for a given node.

For each border node in a component, $b \in C$, we execute Dijkstra's algorithm with b as source and all other nodes (including border nodes) in C as targets. Depending on the type of the source node, the expansion strategy is different. When an incoming border node is the source, forward edges are expanded; vice versa, when an outgoing border node is the source, incoming edges are expanded. This strategy ensures that the maximum number of node expansions is at most twice the number of border nodes of G .

3.3 Shortcut Graph Construction

The third step of the preprocessing phase of our approach is the construction of the shortcut graph. Given a graph G , the shortcut graph of G is a graph $G_{sc}(B, E_{sc})$, where B is the set of border nodes of G and $E_{sc} = E_C \cup S_G$ is the union of the connecting edges, E_C , of G and the shortcuts, S_G , from every incoming border node to every outgoing border node of the same component.

Thus, the number of vertices and edges in the shortcut graph is, respectively,

$$|B| = \sum_{i=1}^k |B_{i_{inc}} \cup B_{i_{out}}| \text{ and}$$

$$|E_{sc}| = \sum_{i=1}^k (|B_{i_{inc}}| \times |B_{i_{out}}|) + E_C.$$

Figure 3 shows the shortcut graph of our running example. Notice that only border nodes are vertices of the shortcut graph. The set of edges consists of connecting edges and the in-component shortcuts between the border nodes of the same component. Note that there is no need for extra computations in order to populate the shortcut graph.

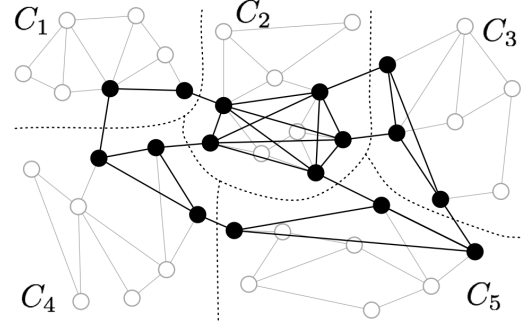


Figure 3: Shortcut Graph illustrated over the original.

4. CORRIDOR MATRIX

In Section 3 we presented how PbS creates shortcuts in order to answer queries when the source and the target points are in different components. However, when the source and the target points of a query are located in the same component, the shortest path may lie entirely inside the component. Therefore, the search algorithm will never reach the border nodes and the shortcuts will not be expanded. In such a case, the common approach is to use bidirectional search to return the shortest path. However, if the components of the partitioned graph are large, the query processing can be quite slow. In order to improve the processing time of such queries, we partition each component again into sub-components, and for each component, we compute its Corridor Matrix (CM). In general, given a partition of a graph G in k components, the *Corridor Matrix* (CM) of G is a $k \times k$ matrix, where each cell $C(i, j)$ of CM contains a list of components that are crossed by some shortest path from a node $s \in C_i$ to a node $t \in C_j$. We call such a list the *corridor* from C_i to C_j . The concept of the CM is similar to Arc-Flags [8], but the CM requires much less space. The space complexity of the CM is $O(k^3)$, where k is the number of components in the partition, while the space complexity of Arc-Flags is $|E| \times k^2$, where $|E|$ is the number of edges in the original graph.

	C_1	C_2	C_3	C_4	C_5
C_1	\emptyset				
C_2		\emptyset			
C_3			\emptyset		
C_4				\emptyset	
C_5					\emptyset

↗ $\{C_2, C_3\}$

Figure 4: Corridor Matrix example.

To optimize the look-up time in CM, we implemented each component list using a bitmap of length k . Therefore, the space complexity of the CM in the worst case is $O(k^3)$. The actual space occupied by the CM is smaller, since we do not allocate space for bitmaps when the component list is empty. For the computation of the Corridor Matrix, we generate the Shortcut Graph in the same way as described in Section 3.3. To compute the distances between all pairs of vertices, we use the Floyd-Warshall algorithm [25], which is specifically designed to compute the all-pair shortest path distance efficiently. After having computed the distances between the nodes, instead of retrieving each shortest path, we retrieve only the components that are crossed by each path, and we update the CM accordingly.

5. SHORTEST PATH ALGORITHM

In order to process a shortest path query from a source point s to a target point t , we first determine the components of the graph the nodes $s \in C_s$ and $t \in C_t$ are located in. If $C_s = C_t$, we execute a modified bidirectional search from s to t . Note that the shortcuts are not used for processing queries for which the source and target are located in the same component C . Instead, we retrieve the appropriate corridor from the CM of C , which contains a list of sub-components. Then, we apply bidirectional search and prune all nodes that belong to sub-components which are not in the retrieved corridor.

In the case that the points s and t are not located in the same component, we exploit the pre-computed shortcuts. First, we retrieve the lengths of the in-component outgoing shortcuts from s to all the outgoing borders of C_s and the length of the in-component incoming shortcuts from all the incoming borders of C_t to t . Then we apply a many-to-many bidirectional search in the overlay graph from all the outgoing borders of C_s to all the incoming borders of C_t . We use the length of the in-component shortcuts (retrieved in the first step) as initial weights for the source and target nodes of the bidirectional search in the Shortcut Graph. The list of edges consisting the path is a set of connecting edges of the original graph and in-component shortcuts. For each shortcut we retrieve the pre-computed set of the original edges. The cost to retrieve the original path is linear to the size of the path. After the retrieval we replace the shortcuts with the list of edges in the original graph and we return the new edge list, which is the shortest path from s to t in the original graph.

6. PRELIMINARY RESULTS

In this section, we compare our PbS method with CRP, the method our own approach is based on, and CH, a lightweight yet very efficient state-of-the-art approach for shortest path queries in road networks [17]. CRP can handle arbitrary metrics and edge weight updates, while CH is a technique with fast pre-processing and relatively low query processing time. We implemented in Java the basic version of CRP and PbS. The CH algorithm in the experiments is from *Graphhopper Route Planner* [26]. Due to the different implementations of the graph models between ours and CH, we do not measure the runtime. Instead, for preprocessing we count the extra shortcuts created by each algorithm, while for query processing we count the number of expanded nodes.

For the experiments we follow the same evaluation setting as in [17]. We use 5 publicly available datasets [27], four of which are a part of the US road network, and the smallest one represents the road network of Rome. We present the characteristics of each dataset in Table 1. In order to compare our PbS approach and CRP with CH, we run our experiments over 5 query sets Q_1 – Q_5 , which

Name	Region	# Vertices	# Edges
CAL	California/Nevada	1,890,815	4,657,742
FLA	Florida	1,070,376	2,712,798
BAY	SF Bay Area	321,270	800,172
NY	New York City	264,346	733,846
ROME	Center of Rome	3353	8,859

Table 1: Dataset characteristics.

contain 1000 queries each. We make sure that the distance of every query in set Q_i is smaller than the distance of every query in set Q_{i+1} . We also evaluate the CM separately by comparing our CM implementation against Arc Flags and the original bidirectional search for a set of 1000 random queries in the ROME dataset. We use a small dataset in order to simulate in-component query processing.

6.1 Preprocessing and Space Overhead

Figures 5 and 6 show a series of measurements for the preprocessing cost of our approach in comparison to CRP and CH over the four largest datasets. Figure 5 shows how many shortcuts are created by each approach. The extra shortcuts can be translated into the space overhead required in order to speed-up shortest path queries. CH uses shortcuts which represent only two edges, while the shortcuts in PbS and CRP are composed of much longer sequences. The difference between the shortcuts produced by CRP and CH is much less. In short, PbS produces about two orders of magnitude more shortcuts than CRP and CH. Moreover, we can observe that the number of shortcuts produced by PbS is getting lower as the number of components is increasing.

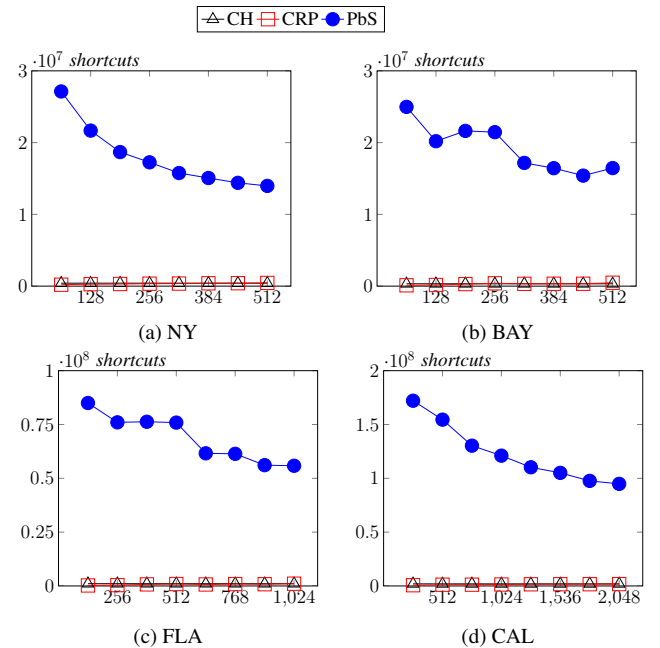


Figure 5: Preprocessing: # of shortcuts vs. # of components.

The same tendency as observed for the number of shortcuts can be observed for the preprocessing time. In Figure 6, we can see that PbS requires much more time than CRP and CH in order to create shortcuts. However, we should also notice that the update

cost for CRP and PbS is only a small portion of the preprocessing cost. When an edge weight changes, we need to update only the shortcuts that contains that particular edge. In contrast, for CH the update cost is the same as the preprocessing cost since a change in a single weight can influence the entire hierarchy.

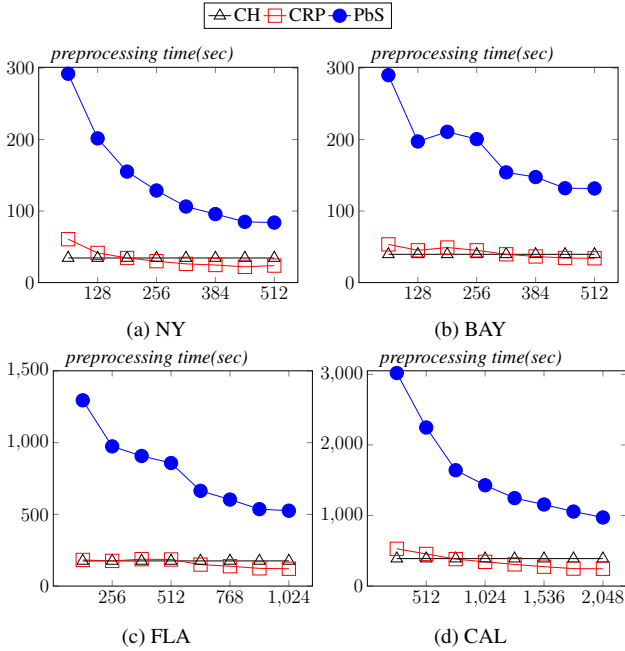


Figure 6: Preprocessing: time vs. # of components.

6.2 Query Processing

Figure 7 shows a series of measurements of the performance of CRP and PbS. We evaluate both techniques for different partitions and various numbers of components. An important observation is the tendency of the performance for CRP and PbS. The performance of CRP gets worse for partitions with many components while the opposite happens for PbS. The reason is that for partitions with few components, PbS manages to process many queries with two look-ups (the case where the source and the target are in adjacent components).

In Figure 8 we compare CH with CRP (we choose the best result) and two configurations of PbS: PbS-BT, which is the configuration that leads to the best performance, and PbS-AVG, which is the average performance of PbS among all configurations. We can see that PbS outperforms CRP in all datasets from Q_1 to Q_5 . However, CH is faster in terms of query processing than our PbS approach. CH is more suitable for static networks as the constructed hierarchy of shortcuts enables the shortest path algorithm to expand much fewer nodes.

6.3 In-component Queries

In Figure 9, we compare the performance of our bidirectional algorithm using the proposed CM, the original bidirectional search and the bidirectional algorithm using Arc Flags. We observe that the bidirectional search is the slowest since no pruning is applied. Between Arc Flags and CM, the Arc Flags provide slightly better pruning thus fewer expanded nodes by the bidirectional search. On the other hand, the preprocessing time required to compute the Arc Flags is significantly higher than the time required to compute the CM.

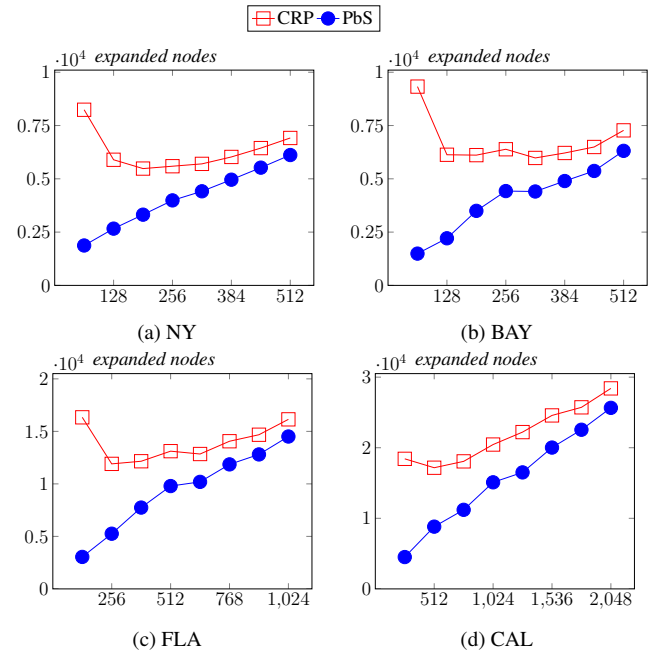


Figure 7: Performance of shortest path queries vs. # of components.

7. CONCLUSION

In this paper we presented PbS, an approach which uses graph partitioning in order to compute shortcuts and speed-up shortest path queries in road networks. Our aim was a solution which supports efficient and incremental updates of edge weights, yet is efficient enough in many real-world applications. In the evaluation, we showed that our PbS approach outperforms CRP. PbS supports edge weight updates as any change in the weight of an edge can influence only shortcuts in a single component. On the other hand, CH is faster than our PbS approach. However, CH cannot handle well edge weight updates as almost the entire hierarchy of shortcuts has to be recomputed every time a single weight changes. For queries where the source and the target are in the same component, we introduced the CM. The efficiency of the CM in query processing approaches the efficiency of Arc Flags, while consuming much less space.

In future work, we plan to extend our approach to support multi-modal transportation networks, where the computation has to consider a time schedule, and dynamic and traffic aware networks, where the weights of the edges change over time. We will also improve the preprocessing phase of our approach both in terms of time overhead, by using parallel processing, and space overhead, by using compression techniques or storing some of the precomputed information on the disk.

8. REFERENCES

- [1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.
- [2] I. S. Pohl. *Bi-directional and Heuristic Search in Path Problems*. PhD thesis, Stanford, CA, USA, 1969. AAI7001588.
- [3] H. Bast, D. Delling, A. Goldberg, M. Müller, T. Pajor, P. Sanders, D. Wagner, and R. Werneck. Route planning in transportation networks. (MSR-TR-2014-4), January 2014.

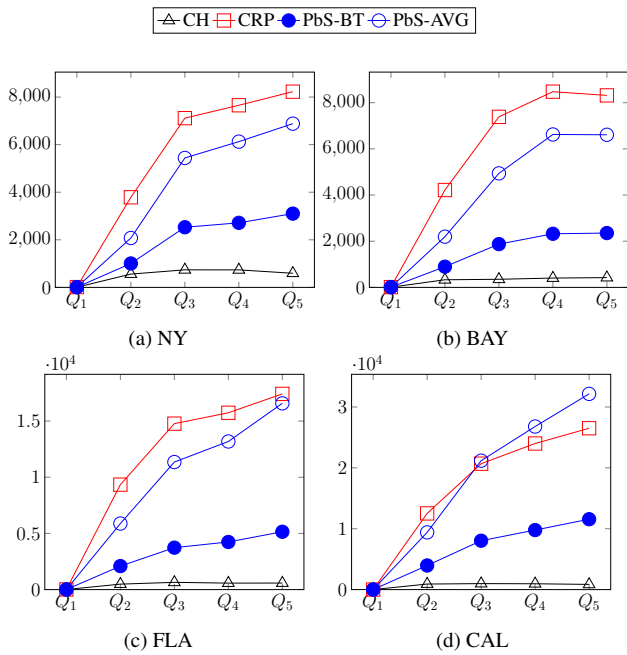


Figure 8: Performance of shortest path queries vs. query sets.

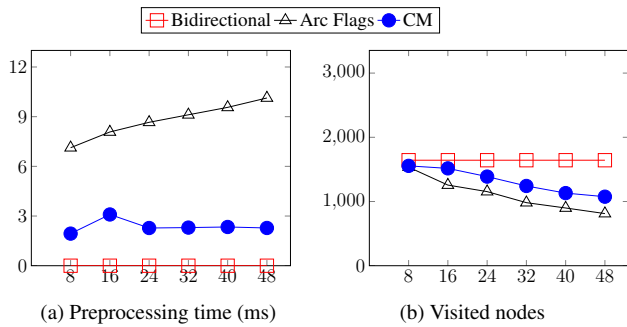


Figure 9: Evaluation of Arc Flags & CM using ROME dataset.

[4] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Proc. of the 10th Int. Symposium on Experimental Algorithms (SEA)*, pages 376–387, 2011.

[5] P. Hart, N. Nilsson, and B. Raphael. Formal Basis for the Heuristic Determination of Minimum Cost PATHS. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] A. V. Goldberg and C. Harrelson. Computing the Shortest Path : A * Search Meets Graph Theory. In *Proc. of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 156–165, 2005.

[7] J. Maue, P. Sanders, and D. Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *Journal on Experimental Algorithms*, 14:2:3.2–2:3.27, January 2010.

[8] E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *Proc. of the 9th DIMACS Implementation Challenge*, 2006.

[9] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *Proc. of the 2005*

Int. Workshop on Geographic Information Systems (GIS), page 200, 2005.

[10] R.A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[11] J. Sankaranarayanan and H. Samet, H. and Alborzi. Path Oracles for Spatial Networks. In *Proc. of the 35th VLDB Conf.*, pages 1210–1221, 2009.

[12] H. Bast, S. Funke, D Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *Proc. of the Workshop on Algorithm Engineering and Experiments*, pages 45–59, 2007.

[13] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 937–946, 2002.

[14] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proc. of the 10th Int. Symposium on Experimental Algorithms*, pages 230–241, 2011.

[15] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proc. of the 13th European Conf. on Algorithms (ESA)*, pages 568–579, 2005.

[16] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice. In *Proc. of the 32nd SIGMOD Conf.*, pages 857–868, 2013.

[17] L. Wu, X. Xiao, D. Deng, G. Cong, and A. D. Zhu. Shortest Path and Distance Queries on Road Networks : An Experimental Evaluation. In *Proc. of the 39th VLDB Conf.*, pages 406–417, 2012.

[18] Y. W. Huang, N. Jing, and E. A. Rundensteiner. Hierarchical path views : A model based on fragmentation and transportation road types. In *Proc. of the 3rd ACM Workshop Geographic Information Systems (GIS)*, 1995.

[19] S. Jung and S. Pramanik. Hiti graph model of topographical roadmaps in navigation systems. In *Proc. of the 12th ICDE Conf.*, pages 76–84, 1996.

[20] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proc. of the 35th Int. Parallel & Distributed Processing Symposium (IPDPS)*, pages 1135–1146, 2011.

[21] A. E. Feldmann and L/ Foschini. Balanced Partitions of Trees and Applications. In *29th Symp. on Theoretical Aspects of Computer Science*, volume 14, pages 100–111, Paris, France, 2012.

[22] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[23] Y. W. Huang, N. Jing, and E. Rundensteiner. Effective Graph Clustering for Path Queries in Digital Map Databases. In *Proc. of the 5th Int. Conf. on Information and Knowledge Management*, pages 215–222, 1996.

[24] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali. Parallel graph partitioning on multicore architectures. In *Proc. of the 23rd Int. Conf. on Languages and Compilers for Parallel Computing*, pages 246–260, 2011.

[25] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962.

[26] <https://graphhopper.com>.

[27] <http://www.dis.uniroma1.it/challenge9/>.