Bachelorarbeit

Dynamic All-Pairs Reachability on Partitioned Graphs

Verfasser

Alwin Stockinger, B.Sc.

angestrebter akademischer Grad

Bachelor of Science (B.Sc.)

Wien, 2019

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Scientific Computing
Betreuerin:	Dr. Kathrin Hanauer, M.SC. B.Sc.
Co-Betreuer:	Dipl.-Math. Dipl.-Inform. Dr. Christian Schulz

Abstract

We performed an experimental study of several *single-source reachability* algorithms for the *fully dynamic all-pairs reachability* problem. Not only were the single-source algorithms generalized to all-pairs algorithms, but their graphs were also partitioned to further enhance their performance. Additionally we implemented some simple minded algorithms, since previous studies showed they were the overall best algorithms.
We tested our algorithms on three different real-world graphs and the results show that a simple static *bidirectional breadth-first search*, which was to our knowledge never tested for this specific problem, beats every one of our optimized algorithms. Even with tuning the many parameters of our algorithms, we were not able to beat the static bidirectional search.

Acknowledgments

I would like to thank my two supervisors, Kathrin and Christian, for their many helpful ideas and support. Especially the many meetings and quick responses helped me to progress with the thesis. Also the graph library *Algora*, maintained by Kathrin, that I was allowed to use did save me a lot of implementation time.

Contents

1 Introduction

Graphs and their algorithms can be useful mathematical tools to represent real world applications. Some of those can be represented by the *all-pairs reachability* problem, which asks for a digraph $G = (V, E)$ if a target vertex t is reachable from a source vertex s over a directed path from s to t. There are multiple applications for this problem ranging from biological networks to XML databases[12].
Some of these applications have a dependence on time and therefore their graphs and algorithms also have to be dynamic, meaning that they change over time. Dynamic versions of this problem can either be *incremental*, if only edge insertions are allowed, *decremental*, if only edge deletions are allowed, or if both are allowed it is called *fully dynamic*. This thesis deals with the fully dynamic version of the problem, while also answering the question of reachability(*queries*) between any two vertices. Algorithms designed for this problem face the challenge to efficiently handle dynamic changes to the graph, while also being able to answer queries in a reasonable time frame. Experimental studies on a range of such algorithms have been conducted in [8] and [13], where for most cases involved dynamic algorithms which maintain the *transitive closure*(Information of reachability for all vertex pairs) could not beat simple minded algorithms which computed the reachability for a vertex pair from scratch for every query. Contrary to that a recent experimental study[10] dealt with the similar problem of *single-source reachability*, which asks the question of reachability always from the same fixed source vertex s. The study showed that simple dynamic algorithms which can update themselves for graph changes do show the best performance for the single-source case.

This thesis deals with the generalization of the single-source reachability algorithms from [10] to all-pairs reachability algorithms and investigates if they are viable for the more general problem. Further the algorithms are optimized by partitioning the graph G into k subgraphs with the partitioning program KaHIP[21]. This optimization approach is inspired by [4], where it was used for faster route planning. Multiple different algorithms using the partitioned graphs have been implemented. They vary in their complexity to keep a dynamic structure, from maintaining generalized dynamic *single-source reachability* algorithms with all their partitioned graph structures to just maintaining the partitioned graph structures for a correct partition and using only static algorithms on them. All the algorithms working on partitioned graphs have been tested and evaluated against unpartitioned static algorithms, which showed to be among or even the best in [8] and [13].

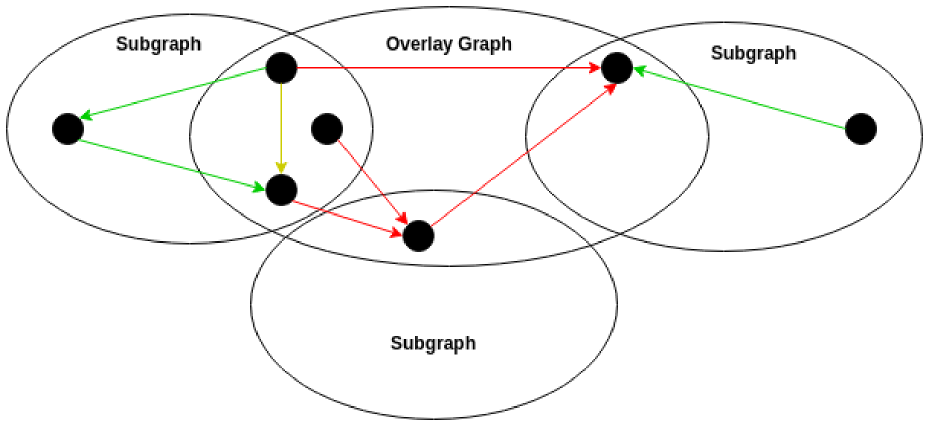


Figure 1: An example of a partitioned graph. Red arrows represent real overlay arcs, yellow arrows are subgraph overlay arcs and green arrows are normal edges inside the subgraphs.

2 Preliminaries

2.1 Concepts

Let $G = (V, E)$ be a digraph with *vertex* set V and *edge* set E. Further let $n = |V|$ and $m = |E|$, with $d = \frac{m}{n}$ being the density of G. Each edge $(u, v) \in E$, also referred to as an *arc*, has a *head* v and a *tail* u. Vertices u which have a edge $(u, v) \in E$ are called *in-neighbours* of v, while vertices v are called *out-neighbours* of u. A *directed path* of edges is a list of edges between two vertices, where the head of the first edge is the tail of the second, the head of the second edge is the tail of the third and so on. A *target vertex* t is *reachable* from a *source vertex* s, if there is a directed path of edges in G from s to t. The *transitive closure* is a digraph $G^+(V, E^+)$ containing the set of all vertices V of $G(V, E)$ and edges $(s, t) \in E^+$ for all vertices $s, t \in V$ if t is reachable from s within $G(V, E)$. [6]

A partitioning of a graph $G(V, E)$ with an integer $k > 1$ is a division of the vertex set V into k *subgraphs* of nearly equal size while also minimizing another objective [20]. The objective for this problem is to minimize the amount of edges running between the subgraphs. Each subgraph $G_s = (V_s, E_s) \subset G$ contains a subset of vertices of G, so $(V_s \subset V)$ and all the arcs $(u, v) \in E_s$ where $u, v \in V_s$. Let $G_v = (V_v, E_v)$ and $G_u = (V_u, E_u)$ be two different subgraphs $(G_v \cap G_u = \emptyset)$. The *overlay graph* $G_o = (V_o, E_o)$ is a graph that contains all arcs (u, v) of the original graph and all tails and heads of such arcs, where $u \in V_u$ and $v \in V_v$. Each set $V_s \cap V_o$ of vertices is referred to as the *border vertices* B_s of the subgraph $G_s = (V_s, E_s)$. This means that the vertex size of the overlay graph n_o is the same as the sum of all border vertices

$$n_o = \sum_s B_s$$

The overlay graph G_o additionally contains an arc (s,t) between border vertices s and t of the same subgraph, if t is reachable from s within that subgraph. Therefore the overlay graph includes the transitive closures of all subgraphs, without the vertices that are not border vertices. The arc size m_o of the overlay graph thus is

$$m_o = m_{o/r} + m_{o/s}$$

with $m_{o/r}$ being the amount of real overlay arcs between different subgraphs and $m_{o/s}$ being the amount of subgraph overlay arcs between border vertices of the same subgraph. Notice that the upper bound for $m_{o/s}$ is

$$m_{o/s} = \mathcal{O}\left(\sum_s B_s^2\right)$$

.

2.2 Related Work

Important theoretical results for the fully dynamic all-pairs reachability problem includes an algorithm with $\mathcal{O}(n^2)$ *amortized* update time and $\mathcal{O}(1)$ worst-case query time [5][18]. Notice that there can't be a better algorithm than $\mathcal{O}(n^2)$ worst-case for updates that answers queries in $\mathcal{O}(1)$, since an update may change $\mathcal{O}(n^2)$ values in the reachability matrix (A matrix representing the transitive closure). There are also dynamic algorithms which can't answer queries in $\mathcal{O}(1)$ time but offer a faster update time [19].

Some of these algorithms and more simple ones have been tested and evaluated in the last decade by [8] and [13]. Their results showed that simple minded algorithms, like Breadth-First-Search, show a very good performance compared to more complex algorithms, for some cases even beating them all. This was especially the case for real-world graphs if the amount of queries was not too high compared to edge deletions and insertions.
Otherwise very similar work has been done in [10], in which single-source reachability algorithms have been evaluated. Their results showed that depending on the scenario a simple minded incremental algorithm or a simplified decremental algorithm that both have been made fully dynamic do show the best performance. The algorithms of [10] will be explained in more detail in chapter 3.1, since this thesis focuses on the all-pairs generalization of their algorithms.

3 Algorithms

3.1 Single Source Reachability Algorithms

All of the unpartitioned all-pairs reachability (AP-Reach) algorithms are based on different Single Source Reachability (SS-Reach) algorithms from [10], except for the later in chapter 3.3 explained Bidirectional BFS algorithm. These SS-Reach algorithms are explained in the following sections.

3.1.1 BFS & DFS Algorithms

Depth-first search(DFS) and *breadth-first search*(BFS) are simple tree search algorithms that were dynamized in two different ways in [10], one with caching, called *caching BFS/DFS*, and the other one with lazy caching, called *lazy BFS/DFS*. There is also a third, most simple algorithm called *static BFS/DFS*, which does not remember any information about the graph. The other algorithms have a cache in which the reachability information of vertices can be stored. The whole cache becomes invalid after an update, if a previously unreachable vertex becomes reachable or an arc from a reachable vertex is removed, then the whole cache has to be built from scratch when a query is made. The lazy algorithms differ from the caching algorithms by caching only the vertices encountered while doing a query and not all of them. When doing a query to a vertex that is not already in the cache, these algorithms can just continue searching at not already cached parts of the graph. If a critical change occurred(one that could change the reachability of cached vertices), the cache of the lazy algorithms is also invalidated and the cache has to be built from scratch for new queries.

3.1.2 Simple Incremental

The *Simple Incremental*(SI) algorithm works in principle like the lazy algorithms but updates its cache immediately when edges are inserted or deleted to answer queries in $\mathcal{O}(1)$ and maintains a reachability tree by storing the reachable parent of each reachable vertex. Insertion updates are handled by updating the previously not, but now reachable vertices. On deletions the algorithm either computes everything from scratch or updates the part of the cache containing all vertices that were known to be reachable because of the removed edge. So all vertices that were in the sub-tree of root v when an edge (u, v) was removed, where u is reachable and the parent of v. The update is done by computing a *reverse* BFS, a BFS that handles arcs as though they were reversed, for those vertices, to find a vertex that is known to be reachable. If such a vertex can be found, it means that the vertex, whose reachability has to be determined, and all its children must be reachable.

3.1.3 Even-Shiloach Trees

The normal *Even-Shiloach*(ES) algorithm refers to the in [7] described decremental connectivity algorithm, which in [11] was discovered to also provide a decremental algorithm for *directed* SS-Reach problems and is made fully dynamic in [10].
When initialising, the algorithm builds a BFS tree from it's source s. For each vertex v encountered, the algorithm saves the lowest possible level(distance from s) $l[v]$ of the vertex in the tree, which will be the level of the first encounter with v, since a BFS tree is used. Also all in-neighbours $N^{-}[v]$ of v are saved and the edges (u, v) of all in-neighbours u are mapped to an index corresponding to the position in $N^{-}[v]$ for fast access.
The algorithms can answer queries in $\mathcal{O}(1)$ by checking the level $l[v]$. When $l[v] = \infty$ it means that v can't be reached, otherwise it has to be reachable.
When an edge (u, v) is inserted, the algorithm builds a BFS tree from v and checks for each vertex (including v) if its level or at least its parent index can be decreased, so that the parent of the vertex is always the in-neighbour with the lowest level and subordinately with the lowest possible parent index. If that's not the case, the algorithm does not have to check the children of the vertex, since then their level will also not be changeable.
When an edge (u, v) is removed and the edge was part of the BFS tree of the source s, the vertex v is added to a FIFO(first in, first out)-queue. For each vertex in the queue a new parent has to be found and perhaps its level $l[v]$ increased. The algorithm checks for each vertex in the queue, if it has an in-neighbour p in $N^{-}[v]$ whose level is $l[v]-1$. If that is the case, the level $l[v]$ does not have to be increased and the new parent of v in the BFS tree is p. When such a parent cannot be found, the level of v is increased by one and all its children and v itself are again inserted into the queue. The children are inserted because they could potentially have another parent that has the same level as v had before the edge removal. If the level of a vertex in the queue increases to n, the vertex can't be reachable from s. So its level is set to ∞ and it will not be inserted back into the queue.
In some cases the queue of vertices that have to be processed becomes so large, that calculating everything from scratch becomes cheaper.
[10] also implemented two slightly changed versions of the algorithm called *Multi-Level ES*(MES) and *Simplified ES*(SES). For deletions the MES algorithm keeps track of the potentially lowest level parent that it encounters while iterating through its in-neighbours and instead of being reinserted into the queue when no parent with level $l[v]-1$ was found, the parent of the vertex is set to be the in-neighbour with the lowest level. If the level of v had to be increased, all its children are added to the queue, because they may have another in-neighbour at the original level of v.
The SES algorithm is a simplified version which does not keep an ordered list of all in-neighbours, instead only a pointer to the parent vertex is stored. If the edge to the parent is deleted the algorithm just iterates arbitrarily through all in-neighbours, instead of iterating through a ordered list and uses the one with

Algorithm	**Insertion Time**	**Deletion Time**	**Query Time**	**Permanent Space**	**Work Space**
SBFS & SDFS	0	0	$\mathcal{O}(n+m)$	0	$\mathcal{O}(m)$
CBFS, CDFS, LBFS & LDFS	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
SI	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
ES & ML-ES	$\mathcal{O}(n+m)$	$\mathcal{O}(n\cdot m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$	$\mathcal{O}(m)$
SES	$\mathcal{O}(n+m)$	$\mathcal{O}(n\cdot m)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$

Table 1: Worst-case time and space requirements for the SS-Reach algorithms

the lowest level as the new parent. If the level had to be increased, all children will be inserted into the queue.

3.2 The All-Pairs Reachability Algorithm based on the Single Source Algorithms

These AP-Reach algorithms use one SS-Reach algorithm for each vertex, so that queries from any vertex s to any vertex t can be answered by using the SS-Reach algorithm with source s and querying it for target t. Therefore the space requirements are all n-times higher than the SS-Reach algorithms they are based upon. The same is also true for the initialization cost, which is just n-times the initialization of the underlying SS-Reach algorithm.

3.2.1 Queries

When a query from vertex s to vertex t is made, the algorithm chooses the SS-Reach algorithm associated with vertex s (e.g. with a map) and queries the algorithm for the reachability of t. Searching for the algorithm associated with vertex s can be done in constant time using a map. Thus the query times are all of the same order as the SS-Reach algorithms they are based upon.

3.2.2 Arc Updates

These updates do only affect the underlying SS-Reach algorithms, since the AP-Reach algorithms themselves do not care about arcs. The AP-Reach algorithm has a worst-case update time of n multiplied with the worst-case time of the SS-Reach algorithms, since there is one SS-Reach algorithm that has to be updated for each vertex in the graph. The work space is the same as for the underlying SS-Reach algorithm since updates for the SS-Reach algorithms are executed sequentially.

3.2.3 Vertex Updates

When a new vertex is added to the graph, the algorithm has to create a new SS-Reach algorithm for the new vertex, therefore the worst-case time for ver-

Based on SS-Reach Algorithm	**Insertion Time**	**Deletion Time**	**Query Time**	**Permanent Space**	**Work Space**
SBFS & SDFS	0	0	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$
CBFS, CDFS, LBFS & LDFS	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n+m)$	$\mathcal{O}(n^2)$	$\mathcal{O}(m)$
SI	$\mathcal{O}(n \cdot (n+m))$	$\mathcal{O}(n \cdot (n+m))$	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(m)$
ES & ML-ES	$\mathcal{O}(n \cdot (n+m))$	$\mathcal{O}(n^2 \cdot m)$	$\mathcal{O}(1)$	$\mathcal{O}(n \cdot (n+m))$	$\mathcal{O}(m)$
SES	$\mathcal{O}(n \cdot (n+m))$	$\mathcal{O}(n^2 \cdot m)$	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(m)$

Table 2: Worst-case time and space requirements for the AP-Reach algorithms

tex updates also depends on the initialization time of the underlying SS-Reach algorithm. When a vertex is removed from the graph, the AP-Reach algorithm has to remove the SS-Reach algorithm belonging to this vertex.

3.2.4 Static Algorithms

When using the static SS-Reach algorithms(SBFS, SDFS) the AP-Reach algorithm can be optimised by using just a single SS-Reach algorithm instead of one for each vertex, because the static algorithms do not save any information belonging to the position of their source in the graph and using one algorithm per vertex would therefore be a waste of memory.

3.3 Bidirectional BFS

We additionally implemented a bidirectional BFS. This algorithm searches from source s with a BFS and from target vertex t with a reverse BFS at the same time by switching between the BFS after a certain amount of steps. If the BFS trees encounter each other, the search can be stopped since then there must be a directed path from s to t. This algorithm has the advantage that it's average query time is significantly lower than that of an ordinary BFS, while its worst-case time stays the same. Graph updates, like with the static BFS/DFS, are completely ignored. Notice that for the static problem this algorithm may be efficient for calculating the reachability of a simple vertex pair, but would not yield any advantages over a normal BFS for calculating the whole reachability matrix. Therefore this optimized algorithm is just useful for the dynamic version of the problem, when single queries are made while the graph undergoes updates.

3.4 The Partitioned Algorithms

The following algorithms are inspired by another algorithm from [4], where a multilevel partitioned graph is used for fast customizable route planning. The algorithms of this thesis, like [4], use partitioning to divide a large graph into multiple smaller subgraphs and an overlay graph that holds all the arcs that run between the different subgraphs. Objective of the partitioning is to reduce the amount of arcs that run between the subgraphs and to keep the size of all

subgraphs almost equal.
$G_s(V_s, E_s)$ and $G_t(V_t, E_t)$ with $s \in V_s$ and $t \in V_t$ will be used to describe the subgraphs of the source s and the target t. Additionally the border vertices of s are named B_s and the ones of t are named B_t.

3.4.1 Queries

For queries the algorithms first check whether the subgraph of the source vertex and the target vertex are the same. If so an algorithm on the subgraph can be used to determine if the vertex is reachable in the subgraph alone. If that's the case the algorithm can return true, else it has to do the standard query.
The general query consists of three parts

- Finding connections from the source to the border vertices of the source subgraph
- Finding connections from the border vertices of the source subgraph to the border vertices of the target subgraph on the overlay graph
- Finding connections from the border vertices of the target subgraph to the target

The Simple Partitioned Algorithm This partitioned algorithm has one AP-Reach algorithm for each subgraph and one for the overlay graph, which are used for reachability queries inside these graphs. First the algorithm does queries to search for a border vertex of the source graph that is reachable from the source vertex. When a connection to a border vertex is found, the algorithm of the overlay graph is used to query for the border vertices of the target graph. When such a connection is found, the algorithm of the target graph is queried for reachability from this border vertex to the target vertex. If it happens to be reachable, the target vertex is also reachable from the source vertex. Otherwise, if no such connection is found, the algorithm continues with the next border vertices. The algorithm can be optimized by removing the just queried vertex from a temporary set of border vertices of the target graph. The set is used as the targets for the overlay queries, so that if other connections to this border vertex exist, they will be ignored and no redundant queries will have to be made from it. When all possible connections over the overlay graph have been tried unsuccessfully, there cannot be any possible path from the start vertex to the end vertex and therefore t is not reachable from s.
The worst-case scenario would be, that the target vertex is not reachable from any vertex of B_t or only reachable from the last vertex in B_t. Also all vertices of B_t and B_s are reachable, but the vertices in B_t are only reachable from the last vertex in B_s. So that for all vertices in B_s, queries have to be done to all vertices in B_t and from the source vertex to all vertices in B_s. This results in the following worst-case query time

$$\mathcal{O}(|B_s| \cdot q_s + |B_s| \cdot |B_t| \cdot q_o + |B_t| \cdot q_t)$$

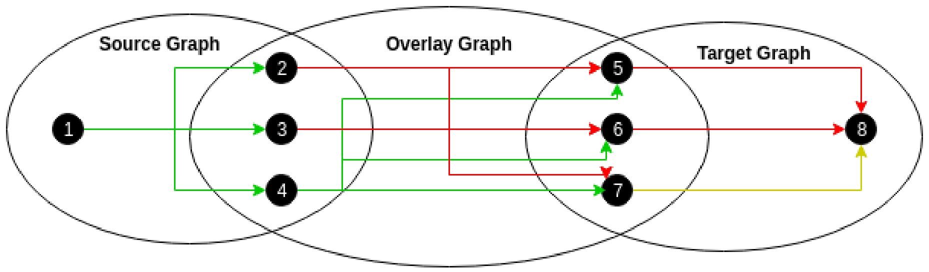


Figure 2: An example of the worst-case for queries with the partitioned algorithm (Green are successful queries, red unsuccessful queries and yellow queries can be either one)

with q_s being the worst-case query time of the source subgraph algorithm, q_o the one of the overlay graph and q_t the one of the target subgraph. Without the optimization of removing already visited vertices from B_t, the worst-case would be

$$\mathcal{O}(|B_s| \cdot q_s + |B_s| \cdot |B_t| \cdot q_o + |B_s| \cdot |B_t| \cdot q_t)$$

An example of the optimized worst-case can be seen in Figure 2. There the border vertices of the source graph(2, 3 and 4) are all reachable from the source vertex(1). After checking if vertex 2 is reachable, queries from vertex 2 are made to all border vertices of the target graph which all fail. Then after checking whether vertex 3 is reachable, the same queries from vertex 3 are made, which also all fail. Then the same queries are made again with vertex 4 as the source and all succeed. Then from vertex 5 and 6 queries are made to the target vertex 8, but both fail. So a query from vertex 7 has to be made. The outcome of 7 is irrelevant for the worst-case query time since it is the last possible query that has to be done. This means that there is one query in the source subgraph for each of its border vertices ($\mathcal{O}(|B_s| \cdot q_s)$), one query from each border vertex of the source subgraph to each border vertex of the target subgraph in the overlay-graph($\mathcal{O}(|B_s| \cdot |B_t| \cdot q_o)$) and one query for each border vertex of the target subgraph in the target subgraph ($\mathcal{O}(|B_t| \cdot q_t)$).

The Partitioned Super Vertex Algorithm The Super Vertex algorithm differs from the above described algorithm only by not using an AP-Reach algorithm for the overlay graph. Instead two additional vertices(from here on called *super vertices*) are added to the overlay graph for each subgraph. One of the two vertices has an incoming arc from each border vertex of the subgraph(from here on called *target super vertex*) and the other vertex(from here on called *source super vertex*) has an arc to all the border vertices of the subgraph. For each source super vertex the algorithm has one SS-Reach algorithm for queries from this vertex. When a query is made, the algorithm first queries the subgraph algorithms of the source and the target for connections to and from the border vertices. For each border vertex that is not reachable from the source, the arc

between the source super vertex and the not reachable border is removed for the query. Similarly if there is no connection between a border vertex of the target subgraph and the target vertex, the arc between this border vertex and the target super vertex of this subgraph is removed. This results in the source super vertex being a representation in the overlay graph for the original source vertex and the target super vertex being a representation for the target vertex. That means that only a single query on the overlay graph with the SS-Reach algorithm of the source super vertex has to be made to the target super vertex to determine the reachability between the original source and target vertices. The algorithm can be optimised by immediately returning false if all the border vertices of the source graph are not reachable or all the border vertices of the target graph can't reach the final target vertex.
Worst-case for the queries thus is

$$\mathcal{O}\left(|B_s| \cdot q_s + |B_t| \cdot q_t + \widetilde{q_o} + \widetilde{a_o} \cdot (|B_s| + |B_t|) + \widetilde{r_o} \cdot (|B_s| + |B_t|)\right)$$

where $\widetilde{q_o}$ refers to the query time of the SS-Reach algorithm used for the overlay graph, $\widetilde{a_o}$ to its addition time and $\widetilde{r_o}$ to its removal time.

The Bidirectional Super Vertex Algorithm We also implemented a less sophisticated version of the Super Vertex algorithm. This algorithm, instead of using SS-Reach algorithms on the overlay graph, uses just a bidirectional BFS on the overlay graph to find if there is a path from the source super vertex to the target super vertex. Since the algorithm on the overlay graph is static, there will be no additional removal and addition time of the arcs for the algorithm.
Worst-case for the queries thus becomes

$$\mathcal{O}\left(|B_s| \cdot q_s + |B_t| \cdot q_t + n_o + m_o\right)$$

The BFS Overlay Algorithm This algorithm uses a simple BFS for queries on the overlay graph and like the previously described partitioned algorithms, a all-pairs reachability algorithm for the subgraphs. With the subgraph algorithms, the algorithm first searches for the border vertices that can be reached from the source and for the border vertices that can reach the target. Then for each reachable border vertex of the source graph, the algorithm does a BFS from that border vertex to find reachable border vertices of the target graph. If a border vertex of the target graph is reachable and that border vertex can also reach the target vertex, the target is reachable from the source.
The algorithm can potentially be enhanced by not doing a BFS from any border vertices already encountered by a previous BFS during this query. Since if a vertex is found in the BFS, also all the children of the found vertex must be in the BFS of the original vertex. Therefore if the BFS from one vertex failed, a BFS from its children also has to fail. But this optimization could potentially also slow the algorithm down, since now for every encountered vertex it has to be checked whether it is a border vertex of the same subgraph, so that it can remember not to query from the encountered vertex.

Since for each source border vertex a query has to be made if no connection is found, the worst-case for queries is

$$\mathcal{O}(|B_s| \cdot q_s + |B_t| \cdot q_t + |B_s| \cdot (n_o + m_o))$$

The Reverse BFS Algorithm This algorithm works like the above mentioned BFS overlay algorithm, but it also uses BFS for queries on its subgraphs instead of all-pairs reachability algorithms. The algorithm does a normal BFS from the source to find all reachable border vertices and a reverse BFS from the target vertex to all its border vertices to find all border vertices which can reach the target. Then like the Overlay BFS algorithm, a BFS is done to search if a reachable source border vertex can reach a target border vertex that can reach the target. The potential optimization of not doing BFS from encountered border vertices also applies.
The worst-case query time for this algorithm is

$$\mathcal{O}(n_s + m_s + n_t + m_t + |B_s| \cdot (n_o + m_o))$$

with n_s/n_t being the amount of vertices of the source/target subgraph and m_s/m_t the amount of arcs of the subgraph.

The Fully Reverse BFS Algorithm This algorithm is the same as the reverse BFS algorithm, but uses a bidirectional search on the overlay graph. The search is done by starting with a BFS that uses all the source border vertices as roots and a reverse BFS that uses all the target border vertices as roots. If the BFS of the source and the reverse BFS of the target intersect at some point, the target must be reachable. The worst-case query stays the same

$$\mathcal{O}(n_s + m_s + n_t + m_t + |B_s| \cdot (n_o + m_o))$$

3.4.2 Arc Updates

If an arc is added to a partitioned algorithm there are two cases to distinguish. In the first case head and tail of the arc belong to two different subgraphs, then the algorithm has to forward the change to the overlay graph and if the head and/or tail are not already present there, also add those vertices to the overlay graph.
In the second case if head and tail belong to the same subgraph, the algorithm can just forward the change to the subgraph, but it then has to check if the reachability from the border vertices of this subgraph to other border vertices of this subgraph has changed. This can be done by checking the reachability from every border vertex to every other border vertex of the same subgraph. If there is such a new connection, a new arc is added to the overlay graph between the reachable borders which represents the reachability inside the subgraph. This is important because vertices could be reachable over a connection inside of a third subgraph which lies in between the target and the source subgraph of the query.

Arc removals are handled similarly. First the algorithm checks to which graph the arc belongs and removes it from the graph. If the graph is the overlay graph, it can happen that a vertex then is no longer relevant in the overlay graph, since the last of its real overlay arcs was removed, then the vertex should also be removed from the overlay graph.
If the graph is a subgraph, the reachability between the border vertices of the subgraph has to be checked, to find if arcs of the overlay graph that run between the border vertices of this subgraph have to be removed.

The worst-case update time for arc changes thus always includes

$$\mathcal{O}(|B_s| \cdot q_{s/all})$$

with B_s being the set of border vertices of the subgraph that is updated and $q_{s/all}$ the query time for finding possible connections to *all* the other border vertices of the subgraph. In the following the different specific update procedures of the partitioned algorithms are described in detail.

The Simple Partitioned Algorithm The worst-case update time for subgraph arc changes is

$$\mathcal{O}(|B_s|^2 \cdot (q_s + u_o) + u_s)$$

with B being the set of border vertices of the subgraph, q_s the query time of the all-pairs subgraph algorithm and u_s the update time of the all-pairs subgraph algorithm and u_o the update time of the all-pairs overlay algorithm. For changes in the overlay graph, this of course only becomes the update time of the overlay algorithm, which is

$$\mathcal{O}(u_o)$$

The Partitioned Super Vertex Algorithm Since the algorithm doesn't use an all-pairs reachability algorithm for its overlay graph and instead only k single-source reachability algorithms, the updates become much cheaper. If only an update in the overlay graph has to be made, the update time becomes

$$\mathcal{O}(k \cdot \widetilde{u_o})$$

with $\widetilde{u_o}$ being the update time of the single-source overlay algorithms. If the update happens in a subgraph, the worst-case time is

$$\mathcal{O}(|B_s|^2 \cdot (q_s + k \cdot \widetilde{u_o}) + u_s)$$

The Bidirectional Super Vertex Algorithm This algorithm has no dynamic overlay algorithm that has to be maintained, therefore the update time becomes

$$\mathcal{O}(|B_s|^2 \cdot q_s + u_s)$$

The BFS Overlay Algorithm Like with the previous algorithm there is no dynamic overlay algorithm that has to be updated, so the worst-case update time is also

$$\mathcal{O}(|B_s|^2 \cdot q_s + u_s)$$

The Reverse BFS Algorithm Updates in the overlay graph also do not concern this algorithm. If an update in a subgraph has to be made, the algorithm now has to do a BFS for new or removed connections between border vertices inside the subgraph to add those as arcs in the overlay graph. This results in a worst-case update time of

$$\mathcal{O}(|B_s| \cdot (n_s + m_s))$$

Otherwise there are no updates to be made, since all used reachability algorithms are static.

The Fully Reverse BFS Algorithm Updates work exactly like for the previous algorithm, therefore the worst-case update time is

$$\mathcal{O}(|B_s| \cdot (n_s + m_s))$$

3.4.3 Vertex Updates

Newly added vertices will not be immediately added to the algorithms partition structure, instead adding the new vertex is delayed until the first arc of this vertex is added. When an arc of this new vertex is added, the new vertex will be added to the subgraph of the other vertex of the new arc. This is done to ensure that the new vertex is placed inside a subgraph where it will get probably a lot of neighbours, so that the overlay graph stays smaller. To ensure queries to and from a not already added vertex are still answered correctly, the algorithms all check first if the source and target vertex are the same and returns true if that is the case. If not, the algorithm checks whether both vertices are already in the partition structure and if not returns false, since a not already added vertex can't have an arc, it can only be reachable from itself.
Alternatively the algorithms can also add new vertices to random subgraphs, to ensure the partition keeps its balance.

When removing vertices, the algorithm has to remove the vertex from the overlay graph, if present there, and from its subgraph.

3.4.4 Space Requirements

The space requirement for these algorithms does depend on the size of the subgraphs and the overlay graph. Notice that for a higher k the subgraphs will be smaller, since $n_s \approx \frac{n}{k}$ and therefore the space requirements of the algorithms will

Algorithm	**Query Time**
Simple Partitioned	$\mathcal{O}\left(\lvert B_s\rvert \cdot q_s + \lvert B_t\rvert \cdot q_t + \lvert B_s\rvert \cdot \lvert B_t\rvert \cdot q_o\right)$
Super Vertex	$\mathcal{O}\left(\lvert B_s\rvert \cdot q_s + \lvert B_t\rvert \cdot q_t + \widetilde{q_o} + \widetilde{a_o}\left(\lvert B_s\rvert + \lvert B_t\rvert\right) + \widetilde{r_o}\left(\lvert B_s\rvert + \lvert B_t\rvert\right)\right)$
Bidirectional Super Vertex	$\mathcal{O}\left(\lvert B_s\rvert \cdot q_s + \lvert B_t\rvert \cdot q_t + n_o + m_o\right)$
Overlay BFS	$\mathcal{O}(\lvert B_s\rvert \cdot q_s + \lvert B_t\rvert \cdot q_t + \lvert B_s\rvert \cdot (n_o + m_o))$
Reverse BFS	$\mathcal{O}(n_s + m_s + n_t + m_t + \lvert B_s\rvert \cdot (n_o + m_o))$
Fully Reverse BFS	$\mathcal{O}(n_s + m_s + n_t + m_t + \lvert B_s\rvert \cdot (n_o + m_o))$

Algorithm	**Update Time**
Simple Partitioned	$\mathcal{O}\left(\lvert B_s\rvert^2 \cdot (q_s + u_o) + u_s\right)$
Super Vertex	$\mathcal{O}(\lvert B_s\rvert^2 \cdot (q_s + k \cdot \widetilde{u_o}) + u_s)$
Bidirectional Super Vertex	$\mathcal{O}(\lvert B_s\rvert^2 \cdot q_s + u_s)$
Overlay BFS	$\mathcal{O}(\lvert B_s\rvert^2 \cdot q_s + u_s)$
Reverse BFS	$\mathcal{O}(\lvert B_s\rvert \cdot (n_s + m_s))$
Fully Reverse BFS	$\mathcal{O}(\lvert B_s\rvert \cdot (n_s + m_s))$

Table 3: Worst-case times for the partitioned algorithms (Letters with tilde refer to SS-Reach algorithms)

not be necessarily larger if there are more subgraphs. The space requirements are of the order

$$S(n,m) = \mathcal{O}\left(\sum_s S_s\left(n_s, m_s\right) + S_o\left(\sum_s \lvert B_s\rvert \;,\; \sum_s \lvert B_s\rvert^2 + m_{o/r}\right)\right)$$

where S_s refers to all algorithms and structures required for a subgraph and S_o to all algorithms and structures required for the overlay graph.

4 Implementation

The algorithms are all implemented in C++ as classes and inherit from the class *DiGraphAlgorithm* from Algora[1]. The DiGraphAlgorithm class of Algora is implemented as an observer to a graph, so that algorithms can automatically be notified if the graph changes. The graph implementation, the SS-Reach algorithms and some other used data structures and algorithms are also from the Algora library.

4.1 All-Pairs Reachability Algorithm based on SS-Reach Algorithms

This algorithm uses the Algora[1] SS-Reach algorithms for queries, which also all inherit from the *DiGraphAlgorithm* class and therefore use the observer functionality, which automatically notifies the algorithms when their graph is updated. The AP-Reach algorithm has one SS-Reach algorithm for each vertex of the graph, which is used as the source in the SS-Reach algorithm. The pairs of vertices and algorithms are saved in a *FastPropertyMap*, which has a worst case lookup time of $\mathcal{O}(1)$. The SS-Reach algorithm that shall be used, can be

specified by a template parameter.
If a query is made, the algorithm looks up the algorithm of the source vertex via the map and then queries the found SS-Reach algorithm for the target vertex and returns its result. Arc changes do not concern this algorithm, since the same graph is also known by the SS-Reach algorithms, which are automatically notified if their graph changes. New vertices will be added to the map with a new algorithm and when a vertex is deleted its algorithm will also be removed from the map.
The optimization mentioned in 3.2.4 is done by using template specialisation. The specialization uses a single SS-Reach algorithm, for which the source is changed to the queried source vertex for every query.

4.2 Partitioned All-Pairs Reachability Algorithms

The implementation of this algorithm uses a pointer to a function which can partition a DiGraph from Algora. The function takes a DiGraph and returns a FastPropertyMap which maps every vertex to an id, indicating to which subgraph the vertex belongs. This allows one to easily swap out the function used to partition with another one, by setting the pointer to the new function.
When the algorithm is initialised, it first uses the function to get a partitioning of its graph. It then proceeds to build a set of new digraphs, the subgraphs, from the original digraph and the returned map. Each new digraph gets all vertices belonging to the same id and the arcs that run between them. Then the overlay graph is build, containing all the original edges that run between vertices of different subgraphs including their tail and head vertices. Also all subgraphs are mapped to the set of their border vertices so that they can be quickly accessed. Then queries are done on all subgraphs to find connections between border vertices, which, if they exist, are added to the overlay graph as arcs.

4.3 Hierarchical Structure of the Algorithm Classes

The AP-Reach algorithms all inherit from a base class called *DynamicAllPairsReachabilityAlgorithm*, which offers queries from a source to a destination and implements, together with its child classes, the composite pattern [9]. The reason for this is, to make it easy to use multiple levels of partitioned algorithms, so that one can use a partitioned graph, which is part of a larger graph, which again could be a subgraph of an even larger graph and so on. Depending on the partitioned algorithm, the pattern also allows one to use different AP-Reach algorithms for the overlay and the subgraphs or to use completely new kinds of AP-Reach algorithms, that are not currently implemented. A simplified class diagram(UML) of the relevant classes can be seen in Figure 3.

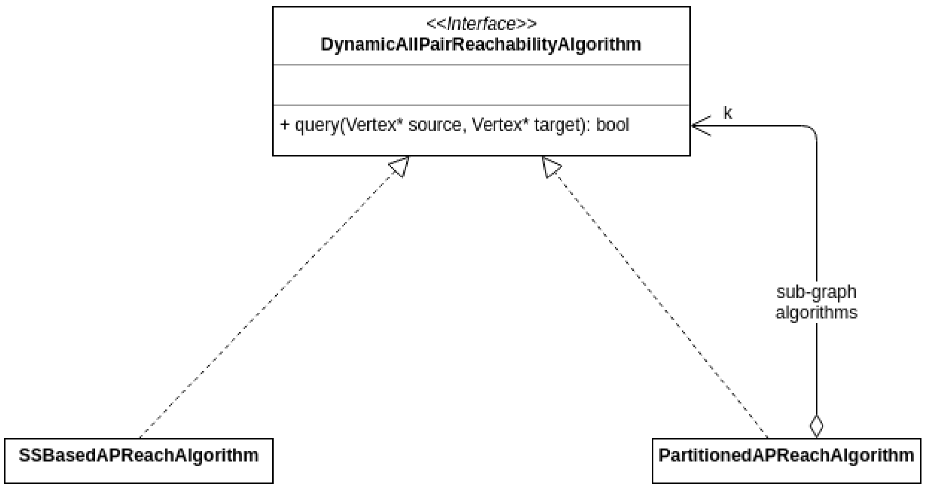

<<Interface>>
DynamicAllPairReachabilityAlgorithm
+ query(Vertex* source, Vertex* target): bool
k
sub-graph algorithms
SSBasedAPReachAlgorithm
PartitionedAPReachAlgorithm

Figure 3: A simple class diagram of the implemented composite pattern

4.4 Partitioning

The partitioning is done with the program KaHIP[21]. Since KaHIP can only deal with undirected graphs, which have no more than one edge between two vertices, the digraphs are converted to undirected ones with weights corresponding to the amount of arcs running in any direction between two vertices, so that KaHIP can respect the importance of certain connections.
KaHIP offers different preconfigurations that vary in their speed and quality. The used preconfigurations are *ecosocial* (Balance between quality and speed), *fastsocial* (Fast but with a worse quality) and *strongsocial* (Slow but with a better quality).

5 Evaluation and Discussion

5.1 Testing Details

All experiments were run on a virtual machine with an *Intel Xeon E5-2650 v4* and 504 GB of memory under *Ubuntu 18.04.2 LTS*. All tests were done on the same single core. To ensure all algorithms get the same partition, the same seed(877) was used for KaHIP. Different partition sizes all with k being a power of 2 have been tested. Queries were generated with the *InstanceProvider* from Algora[1] with the same seed(183788608).
Tables 4 and 5 show the abbreviations of the algorithms and their variables. When SS-Reach subgraph algorithms were used, they are mentioned after the colon. If additionally SS-Reach algorithms are used on the overlay graph, they are written in the bracers after the subgraph algorithm. All results shown have been achieved with the preconfiguration *fastsocial* since it showed the best

Algorithm	Abbreviation	Colour	Partitioned?
Single-Source based Algorithms	*Single-Source Name*	Blue	no
Bidirectional BFS	Bidi BFS	Green	no
Bidirectional Super Vertex	Bidi SV	Red	yes
Fully Reverse BFS	FRBFS	Orange	yes
Overlay BFS	OBFS	Purple	yes
Simple Partitioned	SP	Cyan	yes

Table 4: Algorithm abbreviations

Variable	Abbreviation
Number of partitions	k
Repartition threshold	t
Random vertex additions	r
Don't query from already visited vertices on the overlay graph	A
Step size for the bidirectional BFS	s

Table 5: Algorithm variables

performance by far, especially when repartitioning was taken into account. The stronger partitions did only work insignificantly better for queries and updates.

5.2 Instances

The algorithms were tested on three different sized graphs. The graphs were all squashed to a density of 2, so that the partitioning works better. For each graph update the algorithm also had to answer one query. We always used the same timeout of 10 minutes. The instances include two graphs representing the links between pages on Wikipedia of different languages from the site KONECT[14] and the graph *answers* from [10]. The Wikipedia NL graph was only processed to the year 2004 to reduce its size, so that the many different algorithm variables could be tested out. The *answers* instance of [10] was generated by them using the SNAP software library [16] and estimated initiator matrices [15] which corresponds to a real-world network. The graph was dynamized by generating multiple snapshots of the graph and applying the changes in random order. A table of the instances and their attributes can be seen in Table 6.

Graph	n	m	N	M	$\overline{n}$	$\overline{m}$	$\overline{d}$	δ	δ_+	success
Answers Shuffled[10][16][15]	9.8k	21.8k	15.4k	21.8k	14.0k	21.8k	1.6	390k	50%	22.5%
Reduced Wikipedia NL[2][17]	6.8k	13.5k	64.4k	243k	41.0k	139k	3.2	319k	86%	11.3%
Wikipedia Simple[3][17]	6.8k	13.8k	100k	747k	62.6k	404k	6.1	1.6m	73%	40.6%

Table 6: The attributes of the three tested dynamic graphs. The lowercase letters denote the sizes at the start and the uppercase letters the sizes at the end. δ is the amount of operations, δ_+ the portion of arc additions of the operations and *success* the percentage of successful queries

5.3 Results

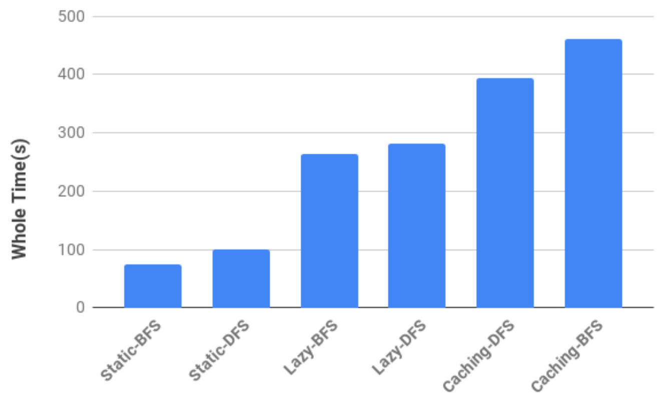
5.3.1 Generalized Single Source Algorithms

We first compare the unpartitioned generalized SS-Reach algorithms, which can be seen in Figure 4. The static, lazy and caching BFS/DFS algorithms were the only algorithms able to finish under the timeout. The more sophisticated SS-Reach algorithms(SI, ES, SES, MES) did have a timeout, which was a result of their high update cost. The completely static algorithms did beat all the dynamic versions of the algorithms, which is in line with the results of [13] and [8] and shows that the best approaches to the SS-Reach problem cannot be generalized so easily for the AP-Reach problem. Looking at the update times in Figure 4b one can see that the update operations are very expensive for even the most simple dynamized algorithms. This is probably due to the large amount of SS-Reach algorithms that have to be notified for the dynamic algorithms, while the static algorithms can completely ignore the updates. The query times, which can be seen in Figure 4c, also favour the static algorithms. This is interesting, since one would probably expect the algorithms which cache reachability information to be faster. But this conundrum can be explained when considering the probability that exactly one of the source vertices, that has also cached the target vertex, is picked, which is quite low. The normal caching algorithms therefore often have to search for the entire reachability information, only to be soon invalidated by an update and probably never queried again at this source vertex. The lazy algorithms have to cache each vertex encountered, but are very unlikely to use the information before its invalidated by an update, which explains why they are slightly slower than the static algorithms in the query time.

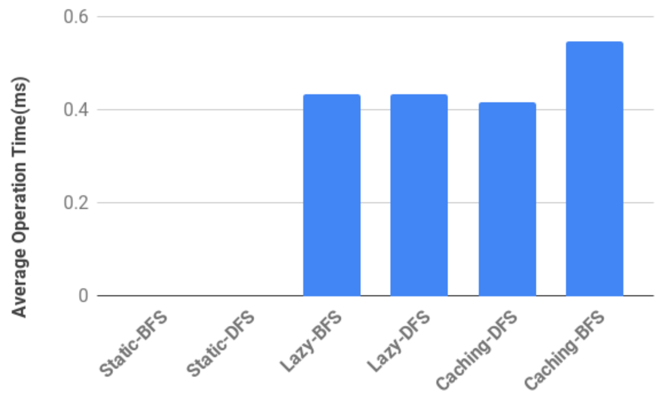
5.3.2 Partitioned Algorithms

A comparison between the partitioned algorithms (Figure 5) shows that a bidirectional approach on the overlay graph is clearly superior to all other approaches. The not bidirectional version of the Super Vertex and the RBFS algorithms are not shown, since their bidirectional versions beat them by a very large margin. Looking at the query times(Figure 5b), one can see that the vast difference arises from the queries. There the Bidirectional Super Vertex and the Fully Reverse BFS algorithm clearly show the best performance. The approach to use dynamic algorithms on the overlay graph did not yield any good results. This can partly be explained when considering how many updates have to be done on the overlay graph. When a new arc is added to a subgraph, potentially multiple new arcs have to be added to the overlay graph, which is problematic for dynamic algorithms with a high update cost. Also for such high values of k, as the here presented fastest algorithms, the overlay graph becomes quite large and therefore the algorithms with SS-Reach algorithms on the overlay graph will suffer from the same problems as the unpartitioned algorithms based on SS-Reach algorithms. Comparing the two fastest partitioned algorithms, the Bidirectional Super Vertex and the Fully Reverse BFS algorithm, shows that

(a) Whole Time



(b) Average Arc Operation Times



(c) Average Query Times

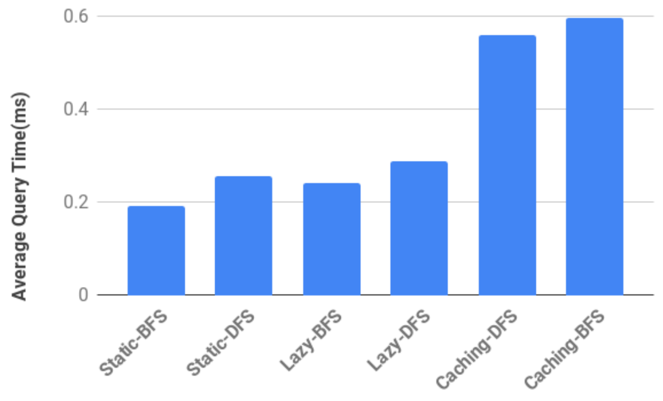
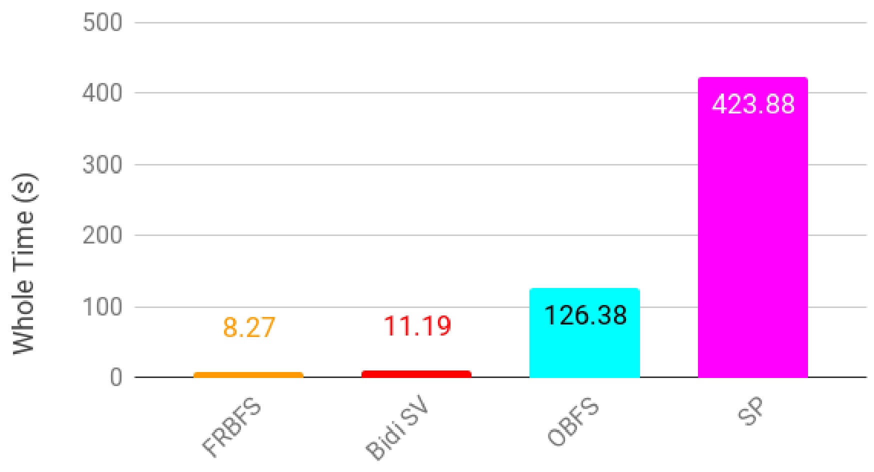


Figure 4: A comparison of the unpartitioned generalised SS-Reach algorithms on the answers graph

(a) Whole Time



(b) Average Query Time

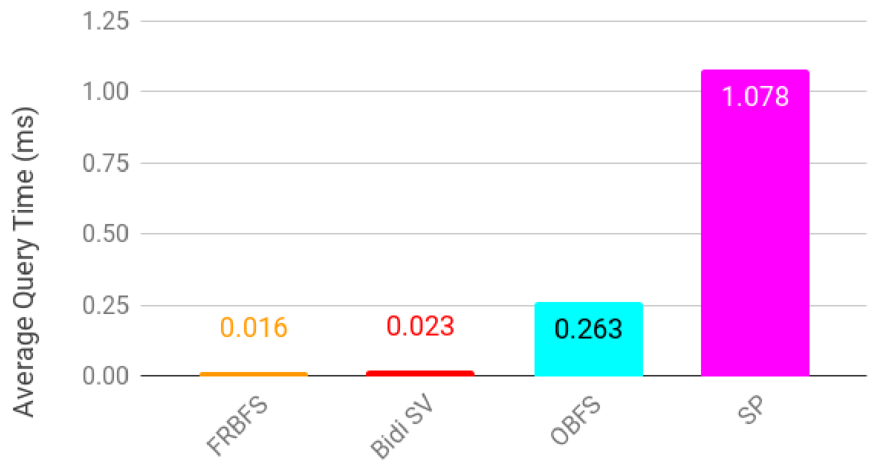


Figure 5: A comparison of the the fastest partitioned algorithms of each category on the answers graph. (Algorithms: FRBFS($k = 4096/t = \infty/r = 0/s = 2$); Bidi SV($k = 4096/t = \infty/r = 0/s = 5$): SI; OBFS($k = 1024/t = 20000/A = 1/r = 0$): SI; SP($k = 8192/t = \infty/r = 0$): Static-DFS (Static-BFS)

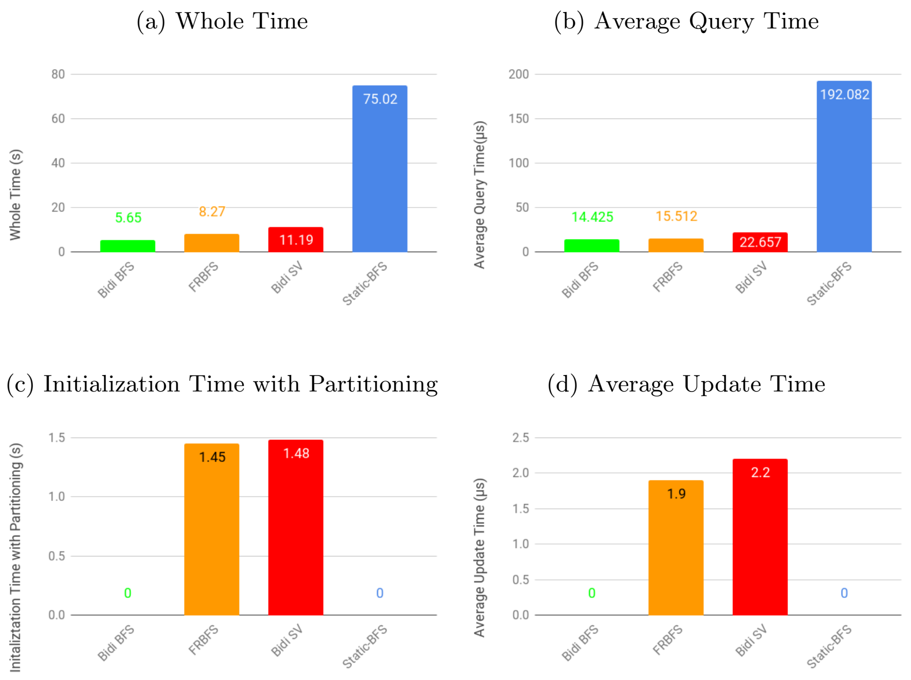


Figure 6: A comparison of the the fastest algorithms of each category on the answers graph. (Algorithms: Bidi BFS($s = 1$), FRBFS($k = 4096/t = \infty/r = 0/s = 2$), Bidi SV($k = 4096/t = \infty/r = 0/s = 5$): SI)

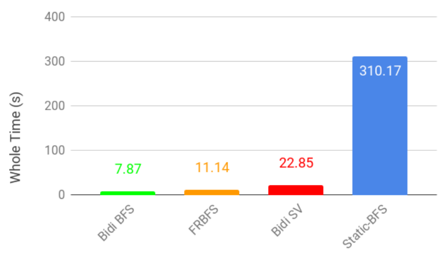
the approach of using generalized SS-Reach algorithms in the subgraphs also doesn't improve the results, which can be explained when considering the high optimal k, as will be explained later in chapter 5.3.4.

5.3.3 The Best Algorithms

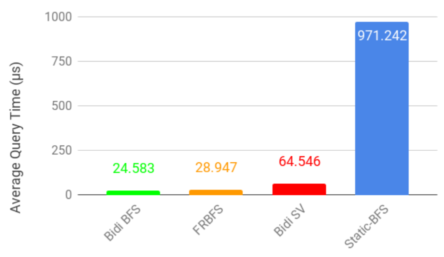
A comparison of the fastest algorithms from different approaches on the answers graph can be seen in Figure 6. There it is clearly visible that the Static-BFS, which showed to be among the best in [8] and [13], can easily be beaten with some partitioned approaches, but the only slightly more complex approach of a static bidirectional BFS can't be beaten by even the best partitioned algorithms. As can be seen in figure 6b, the query times of the Fully Reverse BFS are almost on par with the static approach, but the time it takes to partition and maintain the partition significantly slows down the algorithm as can be seen in Figure 6c.

Similar results can be seen for the Wikipedia graphs in Figure 7 and 8. The optimal step size for the bidirectional BFS(indicated by the s parameter) seems to vary with the graph size, which can partly be explained by taking into account

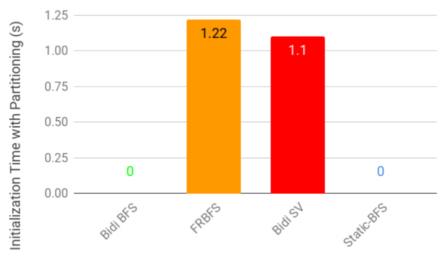
(a) Whole Time



(b) Average Query Time



(c) Initialization Time with Partitioning



(d) Average Update Time

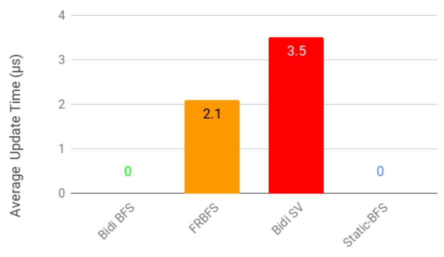
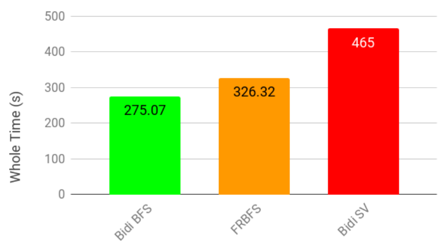
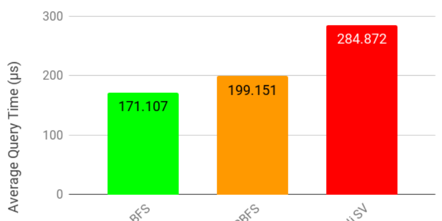


Figure 7: A comparison of the the fastest algorithms of each category on the reduced Wikipedia NL graph. (Algorithms: Bidi BFS($s = 2$), FRBFS($k = 32768/t = \infty/r = 1/s = 4$), Bidi SV($k = 16384/t = \infty/r = 1/s = 3$): SI)

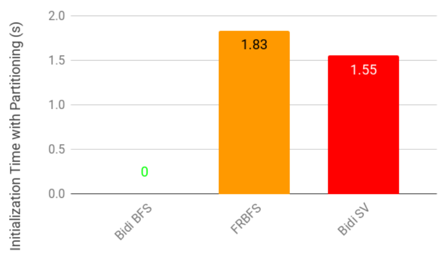
(a) Whole Time



(b) Average Query Time



(c) Initialization Time with Partitioning



(d) Average Update Time

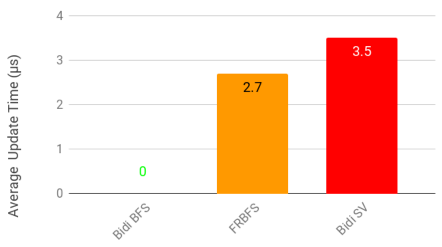


Figure 8: A comparison of the the fastest algorithms of each category on the Wikipedia Simple graph. (Algorithms: Bidi BFS($s = 5$), FRBFS($k = 65536/t = \infty/r = 1/s = 3$), Bidi SV($k = 32768/t = \infty/r = 1/s = 3$): SI)
The Static-BFS did not finish under the timeout of 10 minutes.

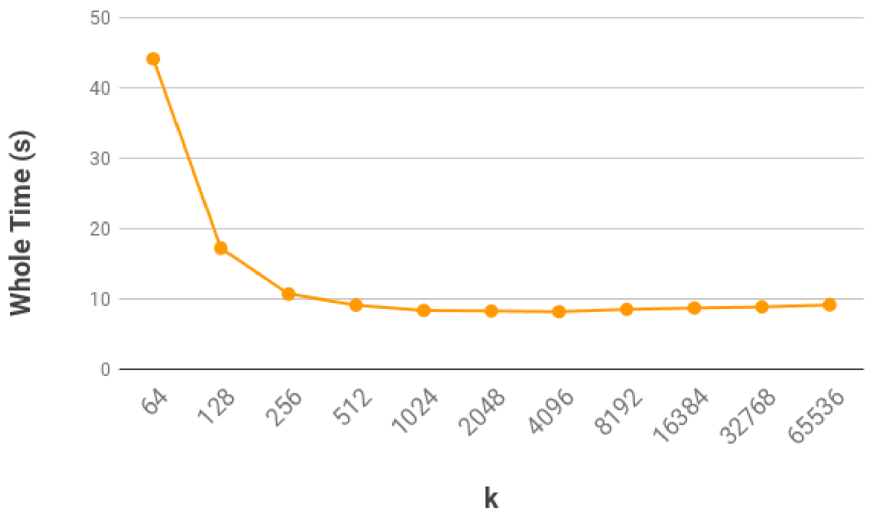


Figure 9: The relation between overall run-time and the amount of subgraphs of the fastest algorithm(FRBFS($k = x/t = \infty/r = 0/s = 2$)) on the answers graph

that switching between the two searches also takes some time and for large graphs the cost of doing to much steps is outweighed by the cost of switching between the searches. Although the step parameter seems to be somewhat random when looking at the partitioned algorithms. The differences in run-time for step sizes close to the shown one were generally very small even when taking the median of multiple measurements. So the strange step parameters could just be a result of run-time fluctuations.
One can can also see that the larger the graph becomes, the higher the k gets, but its always close to n. The Super Vertex algorithm generally preferred smaller or at least equal values for k compared to the FRBFS algorithm, which can be explained when considering that the Super Vertex has additionally $2 \cdot k$ arcs in the overlay graph, so a large k leads to an even larger overlay graph for the super vertex algorithm compared to the overlay graph of the other partitioned algorithms.

5.3.4 Problems with the partition

Finding the best k Surprisingly the best values for k were extremely high. A figure showing the dependence of the overall run-time on the value of k can be seen in Figure 9. A closer investigation of the partition reveals why that is the case. First of all, the amount of border vertices per subgraph becomes smaller with higher values for k, which is an important parameter for the worst-case run-time of all the partitioned algorithms. A plot depicting the dependence of

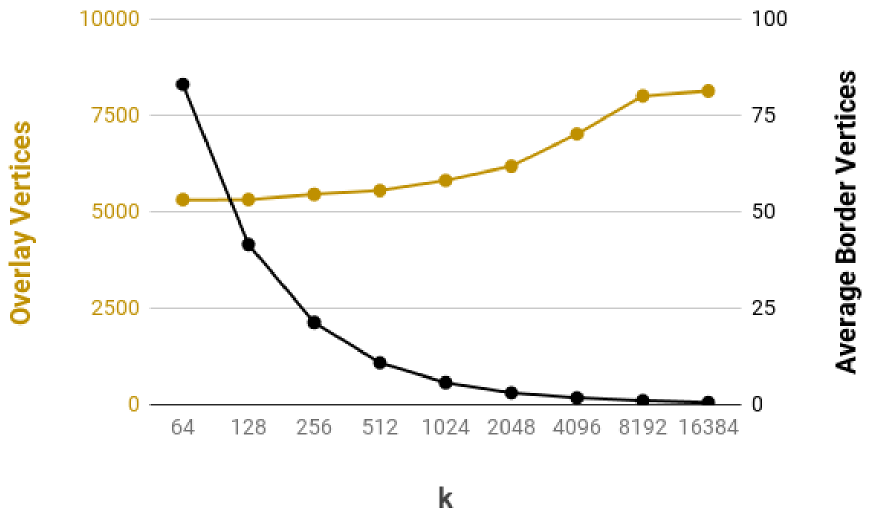


Figure 10: The relation between k and the vertices on the overlay graph for the answers graph at the beginning

the border vertices on the amount of subgraphs can be seen in figure 10. A second problem for low values of k is the high amount of overlay arcs due to subgraph connections. As explained in the preliminaries, the upper bound for these arcs does depend quadratically on the border vertices per subgraph and therefore also on k. This means that for low k there are a large amount of additional arcs that make the overlay-graph in terms of arcs even larger than the original graph, but the subgraphs are also larger for low values of k. This conundrum results in the partitioned algorithms having to have a high k, so that their overlay-graphs keep a manageable amount of arcs. But then the algorithms have to deal with a overlay graph that is almost as large as the original graph, not only in its arc size but also in its vertex size, which diminishes the original purpose of the partitioning to deal with smaller graphs. A plot depicting the arcs due to subgraph connections, real overlay arcs and all the overlay arcs can be seen in Figure 11.

Random Vertex Additions vs Smart Vertex Additions As can be seen in Figure 12, the best approach to adding vertices to the partition does depend on the graph. This is probably due to the effect that the answers graph does not grow as much as the Wikipedia graphs. When comparing the growth of the largest subgraph over time, the largest subgraph in the Wikipedia graphs grows very fast and therefore becomes quite large which in turn leads to an unbalanced partition and slows down the algorithm. With repartitioning this problem can be somewhat mitigated but then the repartitioning becomes an expensive factor

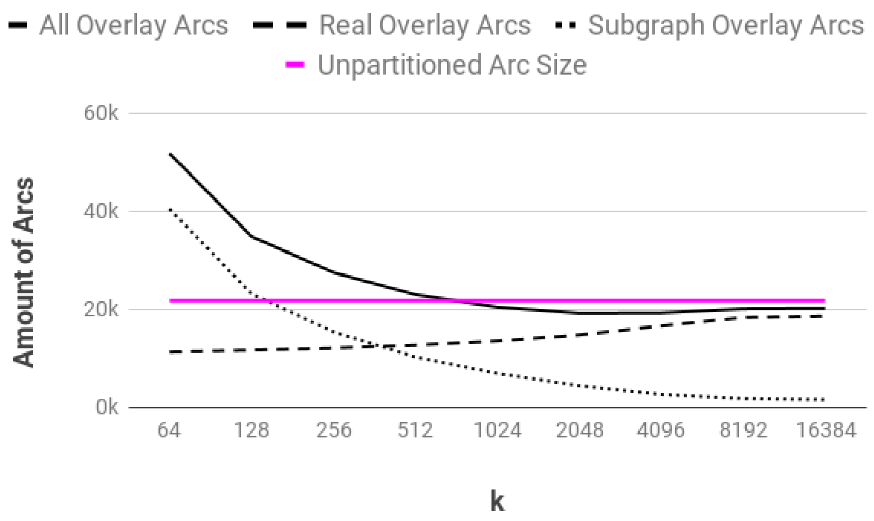


Figure 11: The relation between k and the amount of arcs in the overlay graph at the beginning

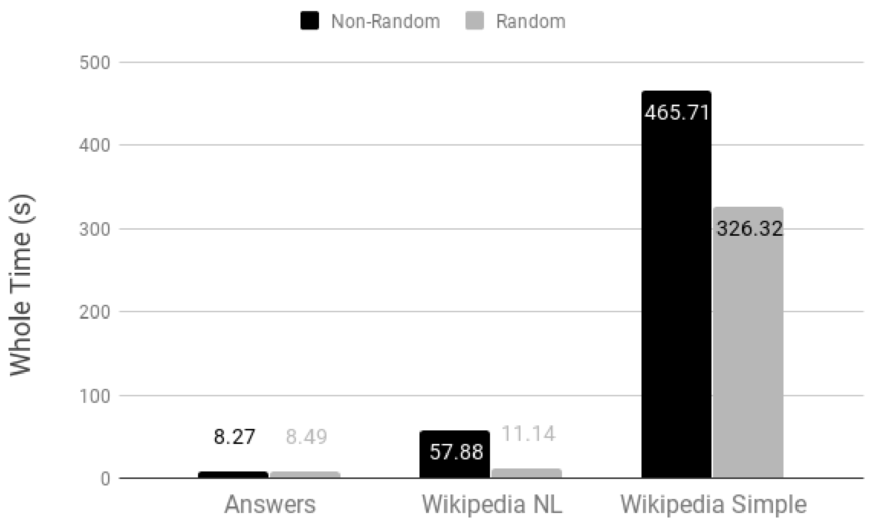


Figure 12: The fastest algorithms with random vertex additions vs the fastest algorithms with a ”smart” vertex addition on all three graphs. The algorithms using the ”smart” repartition on the Wikipedia graphs all required repartitioning to finish under the timeout

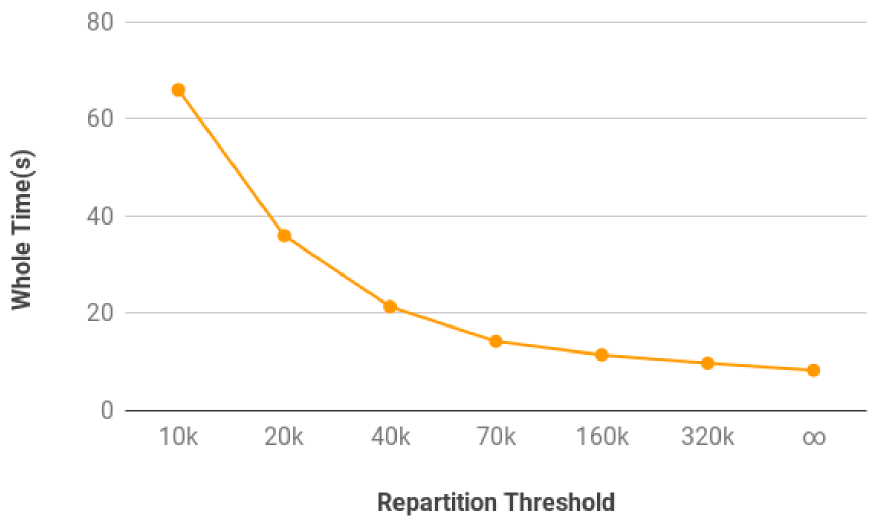


Figure 13: The relation between overall run-time and the repartition threshold of the fastest algorithm(FRBFS(k=4096/t=x/r=0/S=2)) on the answers graph

in the overall run-time of the algorithm.

Repartitioning Repartitioning the graph after a certain amount of operations did only show performance improvements when using non-random vertex additions on the Wikipedia graphs, as just explained it mainly improved the balance of the partition. On the answers graph repartitioning resulted in a worse run-time as can be seen in Figure 13, because the additional partition time could not be mitigated by faster operation and query times.

Advanced Overlay BFS The option to not do a BFS from source border vertices already visited by another overlay BFS during the same query did show to improve the results a lot. A comparison can be seen in Figure 14. The difference becomes less with a higher k, which makes sense, since the average border vertices of a subgraph also decrease with a higher k, which means the performance improvement can't be as large.

Subgraph Algorithms The only tested competitive algorithm that uses SS-Reach algorithms is the Bidirectional Super Vertex algorithm. As can be seen in Figure 15, the choice of the SS-Reach algorithm does not really matter. The differences are so low that with some fluctuations in run-time one can not really say which algorithm is the best. This is a result of the optimal k being so large, which means the subgraphs are extremely small and therefore updates and query costs are nearly the same. That also explains why algorithms that

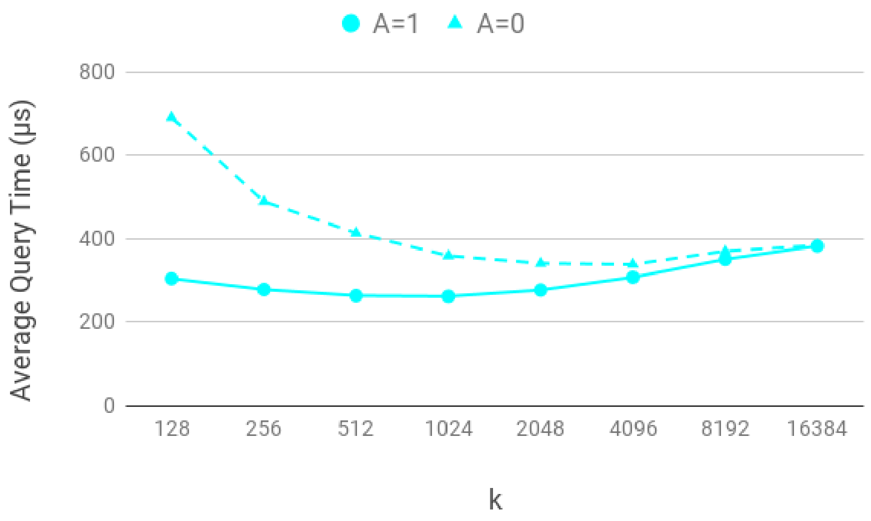


Figure 14: A comparison of the average query time of the algorithm OBFS($k = x/t = 20000/A = ?/r = 0$) when doing queries from already visited vertices on the overlay graph versus not doing queries from already visited vertices.

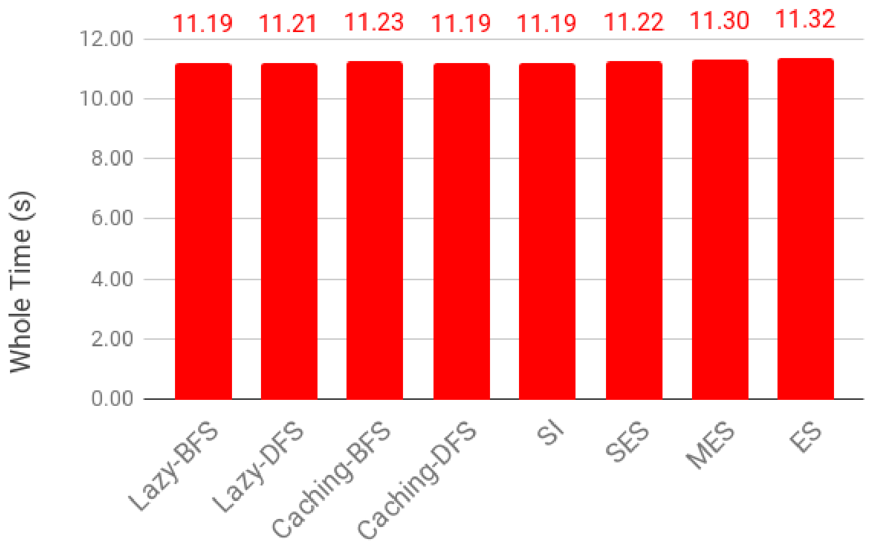


Figure 15: A comparison between different subgraph algorithms on the answers graph with the partitioned algorithm Bidi SV($k = 4096/t = \infty/r = 0/s = 5$)

use static searches on the subgraphs like the FRBFS are faster, since they don't have the overhead of administering subgraph algorithms and picking the right algorithms for queries.

5.4 Memory Requirements

Some of the SS-Reach based algorithms could not be tested on the Wikipedia NL and Wikipedia Simple graph, because they required to much memory. But we expect these algorithms to perform very poorly because of their large memory requirements and their results on the smaller answers graph.
The partitioned algorithms generally did not require much more space than the static algorithms, which is probably a result of the high k and the therefore small subgraphs, which meant that even the highly memory demanding algorithms didn't use much space.

6 Conclusions and Future Work

The approach of using partitioning and or SS-Reach algorithms for a faster AP-Reach algorithm was not successful. Simple minded static algorithms still were the best algorithms compared to the partitioned algorithms tested. These algorithms have a large amount of variables that can be tuned, which makes finding the best algorithms hard and somewhat impractical. One has to tune the variables depending on the graph. Many of the variables also have a dependence on each other which makes finding the optimal values for the variables even harder. For example, the optimal repartition threshold t and the amount of subgraphs k have an obvious dependence on each other. A lower k means faster partition times, which means more repartitions can be done.
All the tuning for the algorithms could not improve their query-times significantly over the best static algorithms such that their additional time of partitioning and maintaining the partition could be mitigated. The fastest partitioned algorithms were not even able to beat the static algorithms in their query time which is probably a result of two factors. First, the overhead of doing not just a query on the overlay graph, but also queries on the subgraphs. Second, the partitioning does not seem to work well for this reachability scenario, since a lot of additional overlay arcs have to be added for correct overlay queries, which slows down the queries on the overlay graph.
We therefore cannot recommend using partitioned and/or single-source algorithms for the all-pairs reachability problem. Instead a simple bidirectional BFS should be used, which is much easier to implement, has only one parameter to tune, does not use much memory and has showed to have the best performance by far compared to all our other implemented algorithms.

But there is much additional work that could be done to improve the partitioned algorithms. The easiest performance gains could probably be achieved by parallelizing the partitioned algorithms. Since there are so many sub-queries to be done for operations and general queries, one could easily just split the many queries in subgraphs among multiple threads. For example checking for new subgraph connections that have to be added to the overlay graph, could be done by executing searches from each border in parallel.
On the algorithm side one could significantly improve the arc add and remove times by using SS-Reach algorithms that can notify the main algorithm if their reachability information changed. Because if such SS-Reach algorithms would be used, the main algorithm would not have to do $|B_s|^2$ queries for every border vertex when a subgraph change is done. Then one could just update the overlay graph by adding/removing the arcs to the vertices of which the SS-Reach algorithms did just report that they now can/can't reach.
It would also be interesting to test other AP-Reach algorithms on on the partitioned graphs than generalized SS-Reach algorithms. Some of the algorithms in [8] and [13] could perhaps work better as subgraph algorithms where answering queries in constant time becomes more important. One could also try to use multi-source/-target reachability algorithms instead of AP-/SS-Reach algorithms, which would yield the advantage of not having to do so many subqueries.
Other future work could include more tuning on the partition side. Balancing the amount of edges in the subgraphs could for example yield some performance improvements. One could also partition the graph with the objective of minimizing the border vertices instead of the arcs between them, since it seems from the results that the border vertices are the major factor in the run-time. But the partitions would probably be similar to the current partitions, since the amount of overlay arcs and the amount of border vertices are dependent on each other.
The method of adding vertices could also be improved by introducing limits to how large a subgraph is allowed to get, before a vertex is added to a random subgraph, instead of adding it to the subgraph of its neighbour.

In summary there is much work that can be done to potentially improve the algorithms, but it remains questionable if they could beat the static bidirectional search, since all our attempts and the attempts of [8] and [13] failed to clearly beat simple static algorithms with complex ones.

References

[1] Algora. `https://gitlab.com/libalgora`.

[2] Wikipedia, nl (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-nlwiki`, Apr. 2017.

[3] Wikipedia, simple en (dynamic) network dataset – KONECT. `http://konect.uni-koblenz.de/networks/link-dynamic-simplewiki`, Apr. 2017.

[4] Delling, D., Goldberg, A. V., Pajor, T., and Werneck, R. F. Customizable route planning. In *International Symposium on Experimental Algorithms* (2011), Springer, pp. 376–387.

[5] Demetrescu, C., and Italiano, G. F. Fully dynamic transitive closure: breaking through the $\mathcal{O}(n^2)$ barrier. In *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000), IEEE, pp. 381–389.

[6] Deo, N. *Graph theory with applications to engineering and computer science*. Courier Dover Publications, 2017.

[7] Even, S., Shiloa, Y., et al. An on-line edge-deletion problem. Tech. rep., Computer Science Department, Technion, 1979.

[8] Frigioni, D., Miller, T., Nanni, U., and Zaroliagis, C. An experimental study of dynamic algorithms for transitive closure. *Journal of Experimental Algorithmics (JEA) 6* (2001), 9.

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[10] Hanauer, K., Henzinger, M., and Schulz, C. Fully dynamic single-source reachability in practice: An experimental study. *CoRR abs/1905.01216* (2019).

[11] Henzinger, M. R., and King, V. Fully dynamic biconnectivity and transitive closure. In *Proceedings of IEEE 36th Annual Foundations of Computer Science* (1995), IEEE, pp. 664–672.

[12] Jin, R., Xiang, Y., Ruan, N., and Wang, H. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 595–608.

[13] Krommidas, I., and Zaroliagis, C. An experimental study of algorithms for fully dynamic transitive closure. *Journal of Experimental Algorithmics (JEA) 12* (2008), 16.

[14] Kunegis, J. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web* (2013), ACM, pp. 1343–1350.

[15] Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., and Ghahramani, Z. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research 11*, Feb (2010), 985–1042.

[16] Leskovec, J., and Sosič, R. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST) 8*, 1 (2016), 1.

[17] Preusse, J., Kunegis, J., Thimm, M., Gottron, T., and Staab, S. Structural dynamics of knowledge networks. In *Proc. Int. Conf. on Weblogs and Social Media* (2013).

[18] Roditty, L. A faster and simpler fully dynamic transitive closure. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (2003), Society for Industrial and Applied Mathematics, pp. 404–412.

[19] Roditty, L., and Zwick, U. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing 45*, 3 (2016), 712–733.

[20] Sanders, P., and Schulz, C. Kahip v2.0–karlsruhe high quality partitioning–user guide. *arXiv preprint arXiv:1311.1714* (2013).

[21] Sanders, P., and Schulz, C. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)* (2013), vol. 7933 of *LNCS*, Springer, pp. 164–175.