Arc-Flag Compression
Student thesis

Andreas Gemsa
Supervising tutor: Prof. Dr. Dorothea Wagner, Dipl. Inf. Daniel Delling

December 19, 2008

Contents

1 Introduction **5**

2 Preliminaries **5**
2.1 Shortest Path . . . 5
2.2 Solving the Shortest Path Problem . . . 6
2.2.1 Dijkstra's Algorithm . . . 6

3 Speed-up techniques **6**
3.1 Arc-Flags . . . 7
3.1.1 Definitions . . . 8
3.2 SHARC . . . 8

4 Bloom Filter **9**
4.1 Fundamentals . . . 9
4.2 Application to Arc-Flags/SHARC . . . 11
4.2.1 Simple . . . 12
4.2.2 A little more elaborate . . . 12
4.3 Reducing the Number of Bits Set . . . 12
4.3.1 Anchor . . . 12
4.3.2 Chains . . . 13
4.3.3 Consider only some Flags for the Bloom Filter . . . 14

5 Compression **14**
5.1 Basic principle . . . 15
5.2 Application . . . 15
5.2.1 The cost function . . . 16
5.2.2 Finding the best flags to remap . . . 17
5.3 Finding the best flag . . . 17
5.3.1 Simple version . . . 17
5.3.2 Slightly enhanced version . . . 18
5.3.3 Fast version . . . 18
5.4 The cost function revisited . . . 20

6 Experiments **20**
6.1 Bloom Filter . . . 21
6.1.1 Basic version . . . 22
6.1.2 Extended version . . . 24
6.1.3 Discussion . . . 25
6.2 Compression . . . 25
6.2.1 Confirming the initial Assumption . . . 25
6.2.2 Comparison of good cost functions . . . 26
6.2.3 Other road networks . . . 29
6.2.4 Artificial Graphs . . . 33
6.2.5 Detailed comparison and analysis . . . 35
6.2.6 Best results . . . 37

7 Conclusion **41**

Acknowledgement

I want to thank Prof. Dr. Dorothea Wagner and Daniel Delling for providing me with the possibility to work on this very interesting subject. Especially Daniel Delling for the advice and encouragement given throughout the work on this thesis. Finally, I want to thank my siblings as well as my parents for their support.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet habe.

Karlsruhe, 19. Dezember 2008

1 Introduction

Motivation. In recent times the use of speed-up techniques for Dijkstra's algorithm to find an optimal route in a road network for private as well as professional use has become increasingly popular. The computing environments for such a problem range from server grids to small hand held devices with limited storage and computing power. Although there are several techniques to solve the problem of finding an optimal route in an adequate time have been developed improvements are still possible. As most speed-up techniques for shortest path queries use additional overhead the space consumption can be significantly higher than without those information. One speed-up technique, SHARC (c.f. [BD09]), uses arc-flags (or simply flags), to determine during a shortest path query, which roads may be and which roads may not be part of the shortest path. Unfortunately, the number of unique flags can pose a significant overhead. This paper deals with way to reduce the space needed for those flags. A space reduction may be very useful for handheld devices which normally lack a large RAM and a huge hard disk. Furthermore, a better performance of the algorithm may be achieved due to better cache utilization.

Main Contribution. The work in this paper is mainly based on SHARC and the overhead it introduces. One of the overhead factors are the arc-flags, the other one is caused by the shortcuts. In this paper we examine two different methods which are able to reduce the space needed for storing the unique flags used by SHARC. An explanation of the fundamentals of both approaches is provided. Finally, both techniques are evaluated through numerous different experiments and the results are discussed.

Overview. This work is organized as follows. In the next section we give an overview over the basics of the shortest path problem. Dijkstra's algorithm is discussed briefly. Then we give an outline of two techniques which solve the shortest path problem but yield a considerable speed up compared to Dijkstra's algorithm. The first method which is able to reduce the space for the arc-flags is discussed in Section 4. It uses the probabilistic data structure called *bloom filter*. The second method is introduced in Section 5. Its basic idea is completely different to the method before. Through a mechanism unique flags are removed from and all references pointing to them are changed to point to another flag. Several different algorithms to determine the flags which are removed are discussed in this section. In Section 6 experiments are conducted and the results are discussed. Some characteristic for the parameters which yield good results are established. We conclude our paper in Section 7.

2 Preliminaries

To solve any problem with a computer a mathematical model of that problem is necessary. The standard model for the search for the optimal route between two points in a road-network includes a directed graph $G = (V, E)$, which consists of the set of nodes V and the set of the edges E. A certain metric (i.e. a cost function) has to be applied to the graph to weight the edges. In the standard case for road networks the weights are positive and a lower weight means a better or preferable connection between two nodes. Generally, the traveling time is used as a cost metric, although others might be used in different approaches.

2.1 Shortest Path

First, we define what a path in a given directed Graph $G = (V, E)$ is. A *path* between two nodes v_s, v_t in a given directed graph $G = (V, E)$ is a set of edges $E' = (e_1, \ldots, e_n) \subset E$ with $e_1 = (v_s, v_1), e_n = (v_n, v_t), \forall_{i \in \{2,\ldots,n-1\}} : e_i = \{v_i, v_{i+1}\}$. The *shortest path* in a given directed Graph $G = (V, E)$ with a cost function $c : E \rightarrow \mathbb{R}$ between two nodes $s, t \in V$ is the path p between s and t where $\sum_{e \in P} c(e)$ is minimal among all paths between s and t.

2.2 Solving the Shortest Path Problem

There are various algorithms which can solve the shortest path problem. Because many of them are well known and have been extensively studied (cf. [CLRS01]), we only introduce one of them briefly. This is Dijkstra's algorithm which is the classic algorithm for solving the shortest path problem for any given directed graph. It is also the basis for the speed-up techniques introduced in the later sections.

2.2.1 Dijkstra's Algorithm

In the case of a cost function c which only maps to non-negative values Dijkstra's algorithm solves the shortest path problem for any given directed Graph $G = (V, E)$ (cf. [Dij59]). The algorithm uses a set S which contains all the nodes whose shortest path, from the source s have already been determined and a priority queue Q, which manages the remaining nodes $V \setminus S$ by their key. The key is the minimal known cost for a path from s. There is an array d, in which all known distances from s to all other nodes are kept, and an array π which maintains the predecessor on the shortest path from s for all nodes, are needed.

During the execution of the algorithm, the node u with the smallest key is chosen and added to the set S, and all outgoing edges of V are relaxed. That means, if any node contained in Q can be reached through u by one of its outgoing edges and with a smaller cost than currently known minimal cost, the key in Q for that particular node is updated to the smaller cost. That process is repeated until the extracted node u from Q is the target node. The shortest path can be obtained through π, and its cost is $d[t] = d[u]$.

The principle of Dijkstra's algorithm is shown in Algorithm 1. Its worst case running time is in $O(|E| + |V| \log |V|)$ (cf. [CLRS01]).

Algorithm 1: Dijkstra's algorithm

input : Directed graph $G = (V, E)$, source node s, target node t
output: Shortest path from s to t

```
1  Initialize Q, d and π;
2  S = ∅;
3  Fill Q with V \ {s};
4  while Q ≠ ∅ and u ≠ t do
5      u = Q.extractMin();
6      S = S ∪ {u};
7      foreach Outgoing edge e = (u, v) do
8          if d[u] + c(e) < d[v] then
9              d[v] = d[u] + c(e);
10             π[v] = u;
11             Q.updateKey(v, d[v]);
```

3 Speed-up techniques

Although Dijkstra's Algorithm solves the shortest path problem its running time in large graphs is relatively high. Before the algorithm reaches the target node it visits all nodes which are closer to the source node than the target node. In large graphs this yields a very high running time. Thus, speed-up techniques have to be devised. They all share a common idea. In an *offline* phase the graph on which the shortest path query is performed is analyzed and enriched with additional information. This information helps in the *online* phase to speed-up a shortest path query. However, the ideas used in both phases may differ from one speed-up technique to another. An overview of some of them can be found in [DSSW09]. We briefly explain two of them.

3.1 Arc-Flags

The basic idea behind the conventional Arc-Flags approach (described in [Lau04], [MSS+06] and [HSWW06]) is to use Dijkstra's Algorithm (see Section 2.2.1) and try to minimize the number of edges which are considered by the algorithm. It does so by including some preprocessing of the graph data.

In the preprocessing phase the set of nodes V of the directed graph $G = (V, E)$ is partitioned into several *cells* $C := \{C_1, \ldots, C_n\}$, so that $\bigcup_{i=1}^{|C|} = V$ and $\forall_{C_i} \forall j \neq i : C_i \cap C_j = \emptyset$. That means every node of V belongs to exactly one cell C_l, and there is a function $f_V : V \mapsto C$, which maps the nodes to the Cells they are contained in. A bit vector f_e, called *flag*, with the size $|C|$ is added for all edges $e \in E$. Every bit corresponds to a cell C_i, and $f_e(i)$ is set to 1 if the edge e the flag belongs to is contained in at least one shortest path to a node within that cell. Initially all bits are set to zero except the bit corresponding to the cell the edge belongs to. The next step in the preprocessing phase is to determine for every edge for every cell C_i if it is included in at least one shortest path starting from any node to another node which lies within C_i. This can be achieved by calculating the *border nodes* or *boundary nodes*, nodes that have at least one neighbouring node which belongs to a different cell, and computing shortest path trees beginning at those boundary nodes to all nodes outside this cell to the border nodes. If an edge e is part of a shortest path to a border node of a cell C_i the bit of the flag belonging to the edge $f_e(i)$ is set to one. Some details on how the shortest path trees can be computed with a centralized shortest path algorithm efficiently are demonstrated in [HKMS08]. An important factor which affects the preprocessing time as well as the time needed to determine a shortest path between two nodes with the arc-flag approach is the way the partitioning is chosen. There are very simple ideas like dividing a graph geographically into rectangles or more sophisticated techniques like the multi-way arc separator. Several possible ways to partition a graph are discussed in [KMS05] and in [MSS+05]. The authors of both papers come to the conclusion that the multi-way arc separator yields very good results. Among the different multi-way arc separators evaluated are those provided by METIS (c.f. [Lab07]).

After the preprocessing phase has finished, a shortest path query can be implemented with Dijkstra's algorithm and only one additional line where the algorithm considers only those edges (cf. Algorithm 2 l. 8) with the flag-bit set to one that corresponds to the cell the target belongs to.

Algorithm 2: Arc-Flag algorithm

input : Directed graph $G = (V, E)$, source node s, target node t, set of Flags F
output: Shortest path from s to t

```
1  Initialize Q, d and π;
2  S = ∅;
3  Fill Q with V \ {s};
4  while Q ≠ ∅ and u ≠ t do
5      u = Q.extractMin();
6      S = S ∪ {u};
7      foreach Outgoing edge e = (u, v) do
8          if cellBit(t) is set in F(e) then
9              if d[u] + c(e) < d[v] then
10                 d[v] = d[u] + c(e);
11                 π[v] = u;
12                 Q.updateKey(v, d[v]);
```

Because the main concern of this work is dedicated to the flags we want to illustrate two points. The most obvious way for storing the flags would be to save each flag separately with the edges they belong to. But most likely it should be avoided. Furthermore, many flags appear more than once and depending on the size of each individual flag (i.e. the number of cells) it is more feasible to save all flags at one location and to save only an index pointing to the appropriate flag with every edge. All

the ideas in this paper are based on the latter approach. The second point we want to stress is that if the bit at position i is set in a flag and this bit is flipped to zero, the result of a shortest path to a target node within cell C_i may become erroneous. If on the other hand a bit on position i is flipped to one, a shortest path query to a target node within C_i is always correct. Only the time needed for the computation may rise. This is proven in the following theorem.

Theorem 3.1

Let $G = (V, E)$ be a directed graph and F its set of flags which is created with the preprocessing phase of the Arc-Flags algorithm. If a bit of a flag $f \in F$ is changed from zero to one the Arc-Flags algorithm still delivers the correct results for a shortest path query between two arbitrary nodes s and t in G.

Proof: As already mentioned, the difference between the Arc-Flags algorithm and Dijkstra's algorithm is only one added line (cf. Algorithm 2 l. 8). It checks if the cell bit of the target node is set in the flag the algorithm considers. If it is, it means the considered edge may be part of the shortest path from s to t. If not, the edge cannot be part of the shortest path. That means if a bit of a flag is changed from zero to one the Arc-Flags algorithm may consider it a candidate for a shortest path. But it always turns out that either this edge does not lead into the target cell or any path containing this edge has larger costs than the actual shortest path. If this would not be the case, the Arc-Flags algorithm would not solve the shortest path problem correctly. But in [HKMS08] the correctness of the Arc-Flags algorithm could be proven. Thus, the theorem follows. ■

3.1.1 Definitions

Now that the term flag has been established it is useful to introduce some definitions. These definitions are used in the following sections.

Definition 3.1 (*Population count*)

The population count of a flag is the number of bits set to one. For a flag f it may be referred to as $|f|$.

Definition 3.2 (*Subset*)

A flag f_1 is called subset of a flag f_2 if f_2 has at least the same bits set to one as f_1. The flag f_1 must not contain bits set to one that are not set in f_2. This property may be indicated with

$$f_1 \subset f_2$$

Definition 3.3 (*One-flag*)

The flag with all bits set to 1 is called ***one-flag****.*

Definition 3.4 (*Zero-flag*)

The flag with all bits set to 0 is called ***zero-flag****.*

3.2 SHARC

The general idea behind SHARC Routing [BD09] is to use Arc-Flags and apply Shortcuts, a technique developed for Highway Hierarchies [SS06], to achieve very small query times. It also aims to overcome two major disadvantages of Arc-Flags. One of the disadvantages is that the search does not gain any speedup within one cell. Furthermore, if a search approaches its target cell, the number of edges which have to be considered rises significantly. SHARC Routing solves these problems by using *multilevel partitions*, which is a family of partitions $\Phi = \{P^0, \dots, P^l\}$ with $\forall_{i<j} C_k^i \in P^i \, \exists C_m^{i+1} \in P^{i+1} : C_k^i \subseteq C_m^{i+1}$ instead of only one partition of cells $P := \{C_1, \dots C_n\}$. The cell C_m^{i+1} for which $C_k^i \subseteq C_m^{i+1}$ holds true is called *super cell*. Note that due to the multilevel partitions $|P^l|$ bits of the arc-flags represent

the cells of the highest *level*. Exactly $|P^{l-1}|$ bits represent the cells of the next lower level and so on. This is an important fact used in the following sections.

As mentioned before, SHARC Routing uses Shortcuts, which are introduced by *contracting* the graph. That means nodes are *bypassed* by removing and adding edges that represent the distances between the remaining nodes. These new edges skip the node n, which is to be bypassed, and are connected to all nodes which n has outgoing edges to. The length of a new edge $e_{new} = \{u, v\}$ is the sum of lengths of the edge $e_1 = \{u, n\}$ and of the edge $e_2 = \{n, v\}$. If there are more than one edge from u to v, only the shorter one is kept. Through a parameter c it can be determined if a node n should be contracted or bypassed. The total number of new edges are called #shortcuts. Should the following condition hold, the node n is contracted: #shortcuts $\leq c \cdot (deg_{in}(n) + deg_{out}(n)$. Of course not all nodes can be contracted easily. Boundary nodes have the important property that they border their cell and are connected to another node of another cell. Although bypassing boundary nodes may lead to an increase in boundary nodes it is possible to add additional boundary shortcuts to improve performance by decreasing the hop count from source to target node.

Because SHARC-Routing uses arc-flags all edges have to be assigned a flag, even the removed ones. The flags of the non-removed edges can be computed by the same mechanism as described in Section 3.1 with the exception that the shortest path trees are grown from the boundary nodes only until all shortest paths from every node of the supercell are determined. The flags for all other edges (i.e. the removed edges) are either set to zero on all positions except for their own cell (called *own-cell bit*) or they are all set to one. The former is done if the node the edge originates from is bypassed. The latter is done otherwise. All shortcuts get their own cell bit set to zero, because there is no gain in speedup in relaxing shortcuts. However, this procedure does not yield optimal results and there are some possible refinements possible as described in [BD09], which may result in an additional speed-up.

The resulting graph is different from the original one in that it contains additional shortcuts. A shortest path query using SHARC uses multi-level Arc-Flags Dijkstra and the graph constructed by the preprocessing of SHARC. The modifications of the original Dijkstra are that, when settling a node u, i.e. taking it out of the priority queue and adding it to the set S (cf. Algorithm 1 lines 5-6), the lowest level i for which u and the target node t are in the same super cell is calculated. When an outgoing edge is relaxed (cf. Algorithm 1 lines 7-12), only those edges with arc-flag set for level i are considered.

Although the differences to the conventional Arc-Flags mechanism are quite extensive, the property mentioned in Section 3.1 in Theorem 3.1 still holds true. If any bit of any flag is flipped from zero to one the query still calculates the shortest path correctly. This is important because all of the following approaches to minimize the space needed for the flags are based on that fact.

4 Bloom Filter

A bloom filter is a probabilistic data structure that was introduced in [Blo70] by Burton Bloom. It uses hash functions for a set membership test. This test returns `true` if the element has been inserted into the bloom filter. Due to the probabilistic nature of this data structure a membership query may return `true` even if the element has not been inserted. Although it may not seem obvious how this can help in reducing space needed for the arc-flags, it is possible and is explained in the following. But first the bloom filter is introduced.

4.1 Fundamentals

As already mentioned, a bloom filter supports membership queries for elements of a set. The query guarantees to return `true` if the element has been inserted into the bloom filter and may return `true` or `false`, depending on a false positive rate, if the element has not been inserted. The false positive rate depends on three factors. The length l of the bit vector used by the bloom filter, the number of elements d which are inserted into the bloom filter and the number of hash functions k. An element is inserted into a bloom filter by calculating k hashes by the k hash functions. These hashes represent positions in the bit vector and are all set to one. If a membership query is carried out, the bloom filter calculates the k hashes again and tests whether all bits at the positions corresponding to hashes

in the bit vector are set or not. If all are set to one, it returns `true` (i.e. meaning the element is probably included in the set) and `false` if one or more positions are set to zero. The pseudo code of inserting and testing elements is given in Algorithms 3 and 4.

Algorithm 3: Bloom filter insert function

```
  input: Element e, number of hash functions k, bit vector size l
1 for i = 1 to k do
2     hash_value = hashfunction_i(e);
3     hash_value = hash_value modulo l;
4     bitvector[hash_value] = 1;
```

Algorithm 4: Bloom filter contains function

```
  input  : Element e, number of hash functions k, bit vector size l
1 for i = 1 to k do
2     hash_value = hashfunction_i(e);
3     hash_value = hash_value modulo l;
4     if bitvector[hash_value] == 0 then
5         return false;
6 return true;
```

The number of hash functions used is important as it can significantly improve or deteriorate the false positive rate. We give a short overview on how the optimal number of the hash functions can be chosen and how to calculate the false positive probability.

If one element is inserted into the bloom filter and only one hash function is used, the probability that any given bit is not set to one is

$$1 - \frac{1}{l}$$

and thus if k hash functions are used the probability is

$$\left(1 - \frac{1}{l}\right)^k.$$

After inserting d elements the probability that a bit has remained zero is:

$$\left(1 - \frac{1}{l}\right)^{kd}.$$

That means that the probability that a bit is set to one is

$$1 - \left(1 - \frac{1}{l}\right)^{kd}.$$

To calculate the false positive rate for a membership query, recall that the bloom filter for an element that has not been inserted returns `true` iff all bits at positions calculated by the hash functions have to be set to one. Thus the false positive rate (FPR) is

$$FPR(l, d, k) = \left(1 - \left(1 - \frac{1}{l}\right)^{kd}\right)^k \tag{1}$$

It can be shown (cf. [Mit02]) that for optimal results the value for k can be obtained by the following equation, but remember that k is the number of hash functions and hence has to be integer:

$$k = \frac{l}{d} \ln 2$$

l/d	FPR
1	61.85%
2	38.25%
3	23.66%
5	9.05%
8	2.14%
10	0.82%
15	0.07%
20	0.006%

Table 1: False positive rate depending on l and d with optimal k

Assuming the optimal value for k has been chosen and the parameters l and d are known, the FPR can be computed either by Equation (1) or approximated by

$$FPR_{approx}(l,d) = 0.6185^{l/d} \tag{2}$$

Some examples of l/d values and their approximate false positive rates are shown in Table 1. These results suggest that if the ratio of bits in the bit vector to the number of inserted elements is 10:1, the false positive rate is below 1%, which is most likely a very good result, but if we have a worse ratio like 3:1, the false positive rate is around 23%.

4.2 Application to Arc-Flags/SHARC

The general idea behind all of the following approaches is very similar in that the bloom filter is used to save flags and restore them with a certain degree of correctness. It is important to note is that a bit of a restored flag must never be erroneously set to zero. As mentioned in Section 3.1 in Theorem 3.1 any bit of any flag may be set to one. The algorithm still determines the correct shortest path. This property is exploited as follows:

Remember that the bloom filter is used to evaluate set membership tests. To store a flag with the help of a bloom filter every bit of every flag has to be assigned a unique number or identifier. This identifier represents the element for the set membership query.

Inserting a flag f into a bloom filter is very simple. For every bit set to one in f its unique identifier is computed and inserted into the bloom filter. Restoring a flag f works intuitively. All unique identifiers for all bits of f are created and the *contains* function (cf. Algorithm 4) is called for every element. If the element has been inserted into the bloom filter (i.e. the bit of f has been set to one), it returns `true` and the bit of the reconstructed flag $\tilde{f}$ is also set to one (we denote the reconstructed flag for a flag f as $\tilde{f}$). Note that $\tilde{f}$ has all bits set that are set in f but may also contain some bits set that are not set in f due to the probabilistic nature of the bloom filter. This guarantees correct results but may imply a higher computational time.

It is necessary to modify the query algorithm of SHARC, so that the bloom filter is used to reconstruct the flags. Note that it suffices to only restore the cell bit which has to be tested. That implies the complexity of a bloom filter membership query is independent from the size of the flags. However, to simplify matters, we always call that process the reconstruction of flags.

A potential problem for the running time of a query is that it is necessary to execute at least one hash computation for every outgoing edge, of all nodes that are settled. Due the the complexity of hash computations it is very likely, that there is a penalty regarding the running time of a shortest path query. The complexity of the hash functions used may heavily influence the time needed for processing a shortest path query. Although this is not part of the analysis presented in this paper it must be considered if implemented. A more detailed analysis of some of the problems bloom filter suffer from and ways to minimize their effects is presented in [PSS07].

The indices at the edges originally pointing to the flag may seem unnecessary, but they are needed to create the unique id of the flag which is required for the bloom filter rebuilding process. Thus

they cannot be eliminated and the only opportunity for space consumption improvement is possible by reducing the space needed for saving the flags.

4.2.1 Simple

The most basic approach is to take all flags, calculate all unique identifiers for all the bits set to one and insert them into the bloom filter. However, this approach may be problematic because many flags may have set many bits to one, which would significantly worsen the false positive rate. Note that even with no improvement in space consumption the false positive rate can be unacceptably high. For example, if the average amount of bits set to one is 1/3 of the size of the flag for all flags, the false positive rate is about 23.66% (cf. Table 1).

4.2.2 A little more elaborate

Due to the multilevel partition used in SHARC some bits represent the cells of the highest level, some bits represent the cells of the next lower and so on. That means if a bit is changed from zero to one that represents one of the cells of the highest level, the number of edges the algorithm unnecessarily considers may rise significantly. However, if a bit is changed from zero to one that represents one of the cells of the lowest level the impact on the search scope (i.e. the edges the algorithm considers) is most likely much lower. All this implies that it may be a good approach not to use only one bloom filter but many of them and to use the parameters l and k to tune that bloom filter for optimal results. In SHARC most of the query is carried out on the topmost level. Only when the search approaches its target the lower levels are important. Hence, it may be useful to "steal" bits from the bit vector of bloom filters which represent low level cells and add those bits to the bit vector of a bloom filter which represents high level cells. Although d is not tunable because it represents the number of elements inserted into the bloom filter, it is known at the construction time. So, it can be used to influence l and k, which makes it possible to adjust the false positive rate. However, this approach comes with certain unavoidable overhead. Instead of storing only one set of parameters l and k for one bloom filter as it would be necessary for the approaches in the previous section, it is now mandatory to save these combinations for every bloom filter. Depending on the number of bloom filters this may be too expensive.

Although the false positive rate for certain bits can be improved, the same drawback as described in Section 4.2.1 still applies. Therefore, it may be desirable to reduce the number of bits set to one in all the flags.

4.3 Reducing the Number of Bits Set

4.3.1 Anchor

In an effort to reduce the number of bits set to one before inserting them into the bloom filter the following approach can be used: A certain set A of flags, so called *anchor-flags*, is chosen that remains unchanged. For any other flag f a new flag f_x is computed by choosing a flag $f_a \in A$ and calculating:

$$f_x = f \; XOR \; f_a \tag{3}$$

The newly created flag f_x is inserted into the bloom filter instead of f. The reconstruction of the flag f resulting in $\tilde{f}$ suffers from some drawbacks. First of all we need to reconstruct f_a and f_x as described in earlier sections, which results in the flags $\tilde{f}_a$ and $\tilde{f}_x$. The obvious approach would be to calculate $\tilde{f}_a$ XOR $\tilde{f}_x$ and assign $\tilde{f}$ the result, but this may lead to wrong results of the shortest path query. An example, why for a reconstructed flag $\tilde{f}$ a bit may be set to zero where it should be one, is illustrated in Figure 1. Therefore, the only way to reconstruct f safely is to use the following equation

$$\tilde{f} = \tilde{f}_x \; OR \; \tilde{f}_a \tag{4}$$

It is easy to see that this way of recovering a flag cannot lead to a bit mistakenly set to zero, albeit it may lead some bits unnecessarily set to one in the reconstruction process of $\tilde{f}$. The probability of

the false positive rate for any bit of f rises from that described in Section 1 to

$$FPR_a(l,d,k) = 1 - \left(1 - \left(1 - \frac{1}{l}\right)^{kd}\right)^{2k}$$

According to the approximation given in Equation (2) the following holds. The probability that one bit is not set to one is:

$$1 - FPR_{approx}(l,d) = 1 - 0.6185^{l/d}$$

Hence the probability that two bits are not set is $(1 - 0.6185^{l/d})^2$ which leads to the false positive rate approximation of a bit of a reconstructed flag of:

$$FPR_{approx_a}(l,d) = 1 - (1 - 0.6185^{l/d})^2 \tag{5}$$

Because f and its anchor flag f_a cannot be the same flag, there must be bits set either in f or in f_a that are not set in the respective other flag. If a bit is set in f_a and not in f, the bit (cf. Equation (4)) is set in $\tilde{f}$, due to the reconstruction, even though it is not set in the original f. To avoid that, it may seem useful to ensure that the flag f_a, which is an anchor to another flag f, has no bit set that is not set in f. But this may limit the number of possible anchor candidates to choose from.

With this idea some additional overhead is created, because every flag has to save a reference to its own anchor flag (if it is itself an anchor, it must be obvious from that reference).

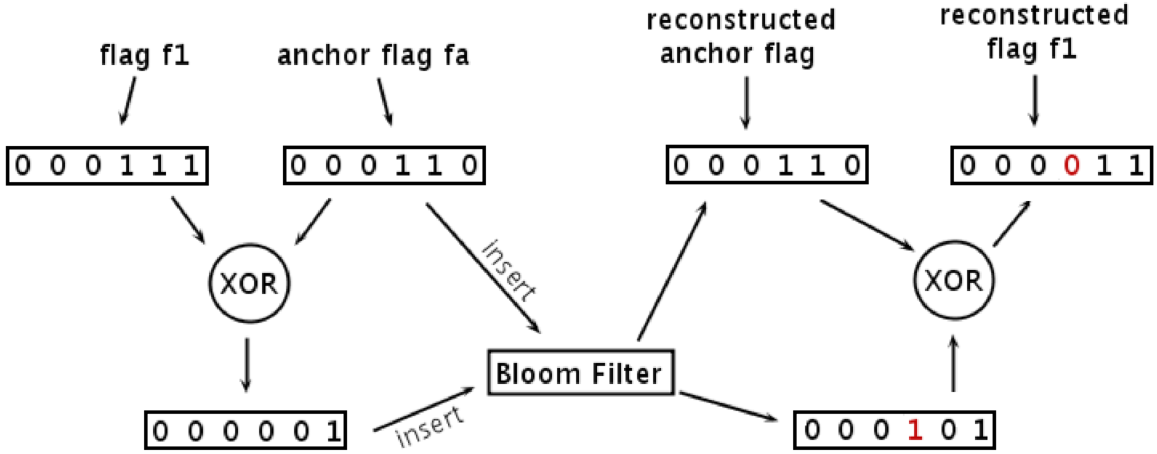

Figure 1: Example how a reconstruction using an anchor flag and XOR through a bloom filter can lead to a flag with a bit erroneously set to zero

4.3.2 Chains

The following idea takes the anchor approach a step further. Instead of applying Equation (3) to a flag f_1 and its anchor-flag f_a, it is applied to f_1 and another non-anchor flag f_2. That process is repeated with the flag f_2 and another flag f_3 and so on until an anchor flag is reached. The flags have now created a *chain* $(f_1, f_2, f_3, \ldots, f_{a1})$ in which flag f_1 is only reconstructable (cf. Equation (4)) if flag f_2 is known and so on. Chains may vary in length. Due to the restrictions mentioned in Section 4.3.1 it is recommended to choose the flags of the chain in such a way that for a flag f_i the flag f_{i+1} has no bit set that is not set in f_i. This idea has a great potential for reducing the number of bits set throughout all flags. Unfortunately, there are two major drawbacks. The more obvious one is that if the first flag f_c of a chain is requested, the flag f_2 has to be reconstructed, which requires a reconstruction of flag f_3 and so on until the anchor flag is reached. This requires many bloom filter accesses and may be unsuitable for an actual implementation. The other, even more serious problem, is a very high false positive rate which is the result of the decoding process. To illustrate this, imagine

l/d	c	FPR
3	1	23.66%
	2	41.72%
	5	74.07%
	10	93.28%
5	1	23.66%
	2	17.28%
	5	37.77%
	10	61.28%
10	1	0.82%
	2	1.63%
	5	4.03%
	10	7.90%
	20	15.17%

Table 2: False positive rate depending on l, d and c with optimal k

that a flag f_i needs to be reconstructed. Every flag in the chain which is between f_i and the anchor flag f_a has to be rebuilt. If only one of those recovered flags f_j (including the flag f_a) has a bit set at a position which was in its original state not set, it becomes set for all reconstructed flags f_c with $c < j$. That means even if for a single bit the false positive rate is relatively small, it may become very high for bits of flags which are distanced from the anchor-flag. The false positive rate for a bit of a flag f_i depending on the number c of flags which have to be reconstructed before f_i can be calculated with:

$$FPR_{approx_a}(l, d, c) = 1 - (1 - 0.6185^{l/d})^c \tag{6}$$

Some examples of combinations of l, d and c and their false positive rate can be seen in Table 2. Even if the false positive rate for a single bit is less than 1%, the probability that a bit of a flag is erroneously set to one may be significantly higher.

In general the drawbacks mentioned are so severe that this approach should most likely not be considered for implementation.

4.3.3 Consider only some Flags for the Bloom Filter

As mentioned throughout the previous sections, the ratio of bits set in all flags to the number of bits in the bit vector of the bloom filter is the main factor that influences the false positive rate. If the ratio is high, the number of bits erroneously set to one is small and increases if the ratio decreases. Due to the fact that some flags are less populated with ones than others it may be good to insert only flags with a certain maximum *population count*, which is the number of bits set to one, into the bloom filter. The flags not inserted into the bloom filter remain untouched and the conventional approach is used. Of course this may heavily limit the number of flags that are inserted into the bloom filter and, hence, may limit the space consumption improvement. However, it can guarantee a low false positive rate. If for example only flags with at most 1/15 bits of the flags length are set to one and space is reduced by one half of the required size for storing them in the conventional approach, the false positive rate is about 2.41% (cf. Table 1).

One thing to keep in mind when using this approach is that during the modified Dijkstra of SHARC for every flag it must be decided if the conventional approach or the bloom filter is used to determine if a certain bit of a flag is set. This branching may be a drawback.

5 Compression

Some of the drawbacks of the bloom filter approach mentioned in Section 4 are the modification of the algorithm to include the bloom filter and the possible high complexity of the numerous hash function

calls during a shortest path query. It may seem better to approach the problem from a different perspective and create a less invasive procedure.

5.1 Basic principle

As already mentioned, the flags are stored at one location and only an index is stored at the edges. That means no two flags are stored twice in the array (which is exactly the advantage over saving the flags directly with the edges). That does not mean that there are no flags which are similar, i.e., they share a common set of bits set to one. If there is a flag f_1, which differs in only one or a few bits to a flag f_2, it might be possible to remove flag f_1 from the array with only a small impact on the time needed for a shortest path query. Of course, all indices of edges pointing to f_1 have to be changed so that they point to f_2. Note that to ensure correctness, f_1 has to be a subset of f_2 (cf. Definition 3.2). The combination of removing flag f_1 and changing all indices as explained is in the following referred to as *remapping* of f_1 to f_2 and f_1 may be called *mapping source* and f_2 *mapping target*. If this process is repeated, the number of unique flags and thus the space needed for storing them can drastically decrease. Note that it is practically guaranteed that this process can be repeated until only one flag is left (the one-flag). All flags can be *mapped* to that particular flag and the shortest path query would still deliver the correct result. The algorithm would behave like the unmodified version of Dijkstra's algorithm. If the number of flags can be reduced significantly it may be possible to reduce the size of the index stored at the edges pointing to their flag. This may yield a significant space consumption improvement, as this concerns every edge of the graph.

Figure 2 shows two examples for the remapping process. At the top are flags consisting of 6 bits. They represent the mapping source. At the bottom are two other flags which represent the mapping targets. The example on the left is a valid pair of mapping source and mapping target the other one is not.

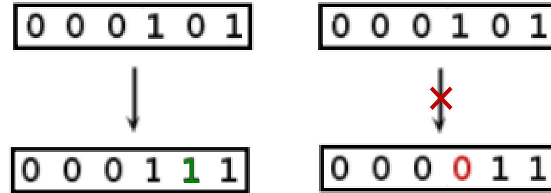

Figure 2: Mapping example. The example on the left represents a valid mapping. The other one is not valid.

One of the most appealing factors of this idea is that there are no changes needed to the SHARC algorithm itself as only flags are removed and the indices of the edges are changed accordingly. All of that is done in the preprocessing phase and does not require any modifications of the query algorithm.

5.2 Application

Let F be a set of flags. The main problem of this approach is to find a pair of flags (f_1, f_2) with $f_1, f_2 \in F$ and the property that if f_1 is remapped, the running time of the shortest path query is affected minimal under all possible combinations of flags (f_i, f_j) with $f_i, f_j \in F$. Even for very small graphs testing all possible combinations is not feasible, hence another way of selecting two flags has to be devised. Three factors that influence the quality of a pair (f_1, f_2) can easily be obtained and most likely influences the running time of shortest path query.

1. The first one is the number of times a certain flag is referenced by the edges of the graph. If this number is high and the bits which are set to one in f_2 and not in f_1 have a much greater impact on the query time, because these bits of these flags are tested much more frequently than it would be the case if f_1 is referenced seldom. Hence flags which are selected for removing should occur as seldom as possible.

2. Another factor which negatively affects the running time of a SHARC query is the number of bits which are changed from zero to one[1]. The more bits are unnecessary set to one the more edges are considered unnecessarily.

3. Due to the multilevel partition approach of SHARC the position of those bits is important, too. If a bit representing a cell of a high level is changed the number of edges that are considered but cannot be part of the shortest path are much higher than if a bit of a cell of a low level is changed from zero to one.

Summarizing, the flag f_{from} which is remapped should be referenced seldom, the mapping target f_{to} should only differ in few bits from f_{from}. Furthermore, the bits that differ from f_{to} to f_{from} should represent lower cells. An essential requirement is that f_{from} is a subset of f_{to}. Otherwise shortest path queries leads to erroneous results.

Algorithm 5: Remapping of flags

---

input: Set of flags F, cost function *cost*, number of flags to remove n

```
1  foreach flag f ∈ F do
2  │  min_cost[f] = min{cost(f, f_i)};
3  flags_mapped = 0;
4  while flags_mapped < n do
5  │  min_cost_flag = min{min_cost[f]};
6  │  min_cost[f] = ∞;
7  │  if isRemoved[to[f]] then
8  │  │  recalculate min_cost[f];
9  │  else
10 │  │  remap f;
11 │  │  isRemoved[f] = true;
12 │  │  flags_mapped++;
```

5.2.1 The cost function

In the following sections the cost function $c_{f_i}(f_j)$ is always in the form of

$$c_{f_i}(f_j) = \begin{cases} o \cdot (\#references(f_i) + a) + c \cdot bitflip(f_i, f_j) & \text{, if } f_i \subset f_j \\ \infty & , otherwise \end{cases}$$

where the $\#references(f_i)$ is the number of times the flag f_i is referenced by edges of the graph. The variables a, o and c can be chosen arbitrarily and $bitflip(f_i, f_j)$ is a function that assigns certain cost to every bit that is set to one in f_j and is not set in f_i. This cost function (in the following called *bitflip cost function*) may be influenced by the multilevel partition used by SHARC. In Figure 3 two examples of a bitflip cost function are shown. The flags consists of 6 bits and are partitioned into two level (indicated by the vertical line). Below the flags there is the the bitflip cost function indicated. In both cases the cost of changing one bit from zero to one for the two leftmost bits are 1 for each and for the remaining 4 bits the cost per bit is 2. The mapping target is below this indication and at last the total bitflip cost is displayed, and bits representing lower bits may be called *bit of a lower level.*

It seems reasonable that the flipping of bits representing cells of higher level should be more expensive than the flipping of bits representing lower ones. In the following the bits representing a cell of a higher level may be called *bit of a high level.*

[1] Note that this occurs every time a flag is remapped

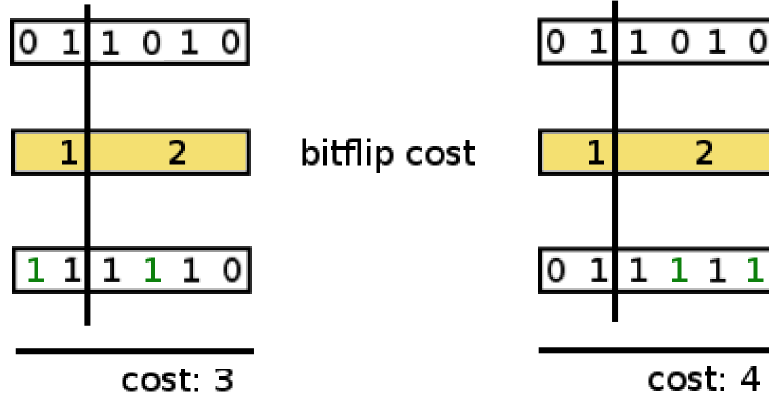

Figure 3: Example of a bitflip cost function

5.2.2 Finding the best flags to remap

Let F be a set of flags and R the set of flags already remapped. In this paper only cost functions of the form explained in Section 5.2.1 or Section 5.4 are used. The cost function helps to determine which flag is remapped next. For every flag f_i another flag f_j is determined with the smallest cost value for $c_{f_i}(f_j)$ for all $f_j \in F \setminus \{f_i\} \cup R$ and is stored at $min_cost[f_i]$ and the flag f_j for which $c_{f_i}(f_j)$ is minimal stored at $to[f_i]$. Note that as long as the one-flag is included in F, there is at least one flag f_j for every flag f_i with $f_i \subset f_j$ (except for the one-flag itself). In the next step the minimum $min_cost[f_l]$ for all $f_l \in F \setminus R$ is determined and f_l is remapped to the flag $to[f_i]$. This process is repeated until a desired number of flags have been removed (i.e. remapped) or there are no candidates for removal left. Observe that if a flag is remapped which has been a mapping target for already removed flags, the edges with the indices to that flag have to be updated appropriately. As it is not feasible to calculate all possible $min_cost[f_i]$ values every time a flag has been removed, we use a minimal binary heap with lazy evaluation which is explained in the following.

First, all $min_cost[f_i]$ values are be calculated and then the flags are inserted into the priority queue with their c_{f_i} value as key. The priority queue maintains the flag with the smallest $min_cost[f_i]$ value at its top. The top element is extracted and remapped, unless its mapping target has been remapped. If the mapping target has already been remapped, the calculated minimal costs are invalid and an new mapping target as well as its mapping costs have to be determined. This is where lazy evaluation is used. Instead of checking after every flag removal, all $to[f_k]$ values if they are valid (i.e. pointing to a non removed flag) this is done only for flags that are extracted from the priority queue. If the extracted flags mapping target has been removed, the minimal costs are recalculated and the element is reinserted into the priority queue. Otherwise the flag is remapped. A simple boolean vector suffices to maintain the state (removed or not removed) for all flags. Its principle is illustrated in Algorithm 5.

5.3 Finding the best flag

The biggest problem with this approach is to find for a flag f_i another flag f_j so that $c_{f_i}(f_j)$ is minimal. It is needed for the first step, the population of the priority queue and later on for recalculating the minimum costs. In the following, we explain three approaches which solve this problem.

5.3.1 Simple version

The simplest method is to determine for every flag f all the candidates which are mapping targets (i.e. the flag f is a subset of those) and calculate the mapping costs to each of those candidates. The minimum mapping cost is determined and every flag is inserted into the priority queue with its minimum mapping costs as key. The subset test of two different flags is done with a simple logical

AND operation. A flag f_i is subset of another flag f_j iff $f_i AND f_j = f_i$ holds true. This approach for filling the priority queue, is illustrated in Algorithm 6. Obviously the running time is in $O(|F|^2)$ and may lead to unacceptably high computation times for graphs with many unique flags.

Algorithm 6: Fill priority queue

```
    input: Priority Queue Q, set of flags F
1   for i = 1 to |F| do
2       min_cost[f] = infinity;
3       for j = 1 to |F| do
4           if i == j then
5               continue;
6           else
7               if F[i] ⊂ F[j] then
8                   if min_cost[i] < cost(F[i], F[j]) then
9                       min_cost[i] < cost(F[i], F[j]);
10      Q.insert(i, min_cost[i]);
```

5.3.2 Slightly enhanced version

The fundamental idea for this principle is that for every two flags f_i and f_j exactly one of the following conditions holds true:

- $f_i \subset f_j$. That means f_i may be mapped to f_j.
- $f_j \subset f_i$. That means f_j may be mapped to f_i.
- No flag is subset of the other one. No flag may be mapped to the other one.

So instead of calculation $f_i AND f_j$ and checking whether this equals f_i and some time later calculation $f_j \, AND \, f_i$ and checking whether this equals f_j only one time $f_i \, AND \, f_j$ is calculated and it is tested whether it equals f_i or f_j. Algorithm 7 displays this algorithm, which populates the priority queue. Although the asymptotic running time is still in $O(|F|^2)$ the number of logical AND operations used is less than in the simple version of this algorithm. Although this leads to a noticeable speed up, a far better approach can be devised if some restrictions for the cost functions are applied.

5.3.3 Fast version

The basic idea behind this approach is to apply some restrictions to the cost function so that it is possible to reduce the search scope for the mapping candidates. To achieve this, the flags are sorted by the cost the bitflip cost function would determine if the zero flag would be mapped to them. Then, it can be proven that for an arbitrary flag f_i the flag f_j with minimal mapping cost is the flag which is the first flag in the sorted order to which f_i is a subset to. This is formulated in Theorem 5.1 and is proven in the following.

Theorem 5.1

Let $c_{from}(f_{to})$ be a cost function like the one explained in Section 5.2.1 with the restriction that the bitflip function only assigns strictly positive values and the mentioned variables a, o and c are independent from the mapping target. Further let f_0 be the zero flag (cf. Definition 3.4) and F a set of flags. Then, for a flag $f_i \in F$ the flag f_j with minimal mapping cost is the flag with the smallest $c_{f_0}(f_j)$ greater than $c_{f_0}(f_i)$ and which satisfies $f_i \subset f_j$.

To Proof this theorem we first prove two simple corollaries.

Algorithm 7: Fill priority queue - small improvement

```
   input: Priority Queue Q, set of flags F
 1 for i = 1 to |F| do
 2   min_cost[f] = infinity;
 3 for i = 1 to |F| do
 4   for j = i + 1 to |F| do
 5     subset_test = F[i] AND F[j];
 6     if subset_test == F[i] then
 7       if min_cost[i] < cost(F[i], F[j]) then
 8         min_cost[i] < cost(F[i], F[j]);
 9     else if subset_test == F[j] then
10       if min_cost[j] < cost(F[j], F[i]) then
11         min_cost[j] < cost(F[j], F[i]);
12   Q.insert(i, min_cost[i]);
```

Corollary 5.1

Let F be a set of flags. If a flag $f_1 \in F$ is mapped to a flag $f_2 \in F$ the population count of f_2 is always greater than that of f_1.

$$|f_1| < |f_2|$$

Proof: This corollary follows directly from the fact that for a mapping source f_1 and a mapping target f_2 the statement $f_1 \subset f_2$ holds true and that $f_1 \neq f_2$. ■

Corollary 5.1 implies that to find a candidate for mapping an arbitrary flag f only the flags with a higher population count than f need to be considered.

Corollary 5.2

Let $c_{from}(f_{to})$ be a cost function like the one explained in Section 5.2.1 with the restriction that the bitflip function only assigns strictly positive values and the mentioned variables a, o and c are independent from the mapping target. Let F be a set of flags and let f_0 be the zero flag. Then, to find for an arbitrary flag $f_1 \in F$ another flag $f_2 \in F$ with $f_1 \subset f_2$ only the flags $f_j \in F$ with $c_{f_0}(f_j) > c_{f_0}(f_i)$ need to be considered.

Proof: As proven in Corollary 5.1 the population count for mapping target f_2 has to be higher than the population count of the mapping source f_1. That means at least one bit in f_2 is set that is not set in f_1. Because the bitflip function assigns only strictly positive values and the equation $c_{f_0}(f_2) > c_{f_0}(f_i)$ has to be true for any possible mapping target f_2, which proves the corollary. ■

Now it is possible to prove Theorem 5.1.

Proof of Theorem 5.1: As we have seen in Corollary 5.2 for to find a mapping target for a flag f_i, only flags with a higher value for $c_{f_0}(f_j)$ need to be considered. Now we need to prove that not only all mapping candidates fulfill that requirement but the flag f_j with the smallest $c_{f_i}(f_k)$ for all $f_k \in F$ is the flag f_j with $f_i \subset f_j$ and the smallest $c_{f_0}(f_j)$ larger than $c_{f_0}(f_i)$.

Let f_{from} be a flag and f_{to} the flag for which $c_{f_{from}}(f_{to})$ is minimal. Further let f_1 and f_2 be flags with $f_{from} \subset f_1$ and $f_{from} \subset f_2$ and $c_{f_0}(f_1) < c_{f_0}(f_2)$. Thus f_1 and f_2 are candidates for f_{to} (the mapping target). It is proven that the mapping costs from f_{from} to f_1 are always less than those from f_{from} to f_2. Remind yourself of the structure of the cost function explained in Section 5.2.1. The number of times an edge is referenced is completely independent from the mapping target and hence is ignored here. Thus, the cost function depends mainly on the bitflip costs. But we are able to calculate these costs, because the bitflip cost for the mapping from f_{from} to f_1 is $c_{f_0}(f_1) - c_{f_0}(f_{from}) =: c_1$ and f_{from} to f_2 equals $c_{f_0}(f_2) - c_{f_0}(f_{from}) =: c_2$. It is easy to see, that $c_1 < c_2$. Combining this and the

insight from Corollary 5.2, that only flags f_j with a higher value for $c_{f_0}(f_j)$ need to be considered the theorem follows. ■

Observe that a conversion from a bitflip function, that assigns not only strict positive values, to one that does is conceivable.

Due to Theorem 5.1 it is now possible to devise an algorithm which outperforms Algorithms 6 and 7. The first thing that has to be done is to compute all values $c_{f_0}(f_i)$ for all $f_i \in F$, sort them in ascending order and store that order in array O. To find for a flag $f_{from} = O[i]$ the flag f_{to} with minimal mapping costs it suffices to look at every value $O[j], j > i$ and stop as soon as a flag $O[j]$ with $f_{from} \subset O[j]$ has been found. This principle is demonstrated in Algorithm 8, which fills the priority queue with the initial values.

Algorithm 8: Fast way of filling the priority queue

```
  input: Priority Queue Q, set of flags F
1 for i = 1 to |F| do
2 └ A[i] = cost(0, F[i]);
3 sort flags in ascending order according to A and store them in O;
4 for i = 1 to |F| do
5 │ j = i + 1;
6 │ while !(O[i] ⊂ O[j]) do
7 │ └ j++;
8 │ min_cost[i] = cost(O[i], O[j]);
9 └ Q.insert(i, min_cost[i]);
```

5.4 The cost function revisited

The cost function mentioned in Section 5.2.1 is unfair. Consider two flags f_1 and f_2. Both have the same #references value and the same bitflip costs if they are mapped. The algorithm chooses one or the other one not considering that for example f_1 may be a mapping target to numerous already removed flags and f_2 may be not a mapping target for any flag. Thus, it seems reasonable to include the flags already mapped to a flag in the cost function. The easiest solution is to add i to an additional #mapped variable of a flag f every time i flags are mapped to f. Then, the bitflip costs can be calculated with $(\#mapped(f_i) + 1) \cdot bitflip$, which means we add the bitflip costs for all flags mapped to f, but only for those bits which differ from f and its target. Those that differ from the mapped flag to f have already been "paid" for. All that leads to the following cost function:

$$\bar{c}_{f_i}(f_j) = \begin{cases} o \cdot (\#references(f_i) + \#mapped(f_i)) + (\#mapped(f_i)) + 1) \cdot bitflip(f_i, f_j) & \text{, if } f_i \subset f_j \\ \infty & , otherwise \end{cases}$$

Note that even though the cost function has been changed, Theorem 5.1 still holds true. The part $o \cdot (\#references(f_i) + \#mapped(f_i))$ is independent of the mapping target. Given that the bitflip cost function itself assigns strictly positive values to the flipping of bits the expression $(\#mapped + 1) \cdot bitflip(f_i, f_j)$ changes only by a constant factor. This constant factor equals the factor c (c.f. Section 5.2.1). Thus, the new cost functions satisfies the precondition of Theorem 5.1.

6 Experiments

In this section all experimental results are shown. To make the experiments independent from the underlying hardware (and implementation of the bloom filter) the quality of the approaches is judged by the number of settled nodes, i.e., the number of nodes taken from the priority queue, by the SHARC

algorithm. To get reliable results 5000 shortest path queries are executed every time a certain number of flags has been remapped or a certain compression level of the bloom filter space has been reached.

The x-axis of all diagrams shown below shows the percentage of flags removed (or space saved). The y-axis shows the average number of nodes which are settled by the SHARC algorithm.

Input. All graphs, which are used in the experiments, are generated with the preprocessing algorithm of SHARC. The graphs are created using the following command:

```
genSHARC -g <graph file> -p <partitioning file> -L 0
```

The SHARC queries are executed by one of the following commands:

```
runSHARC -f <bloom filter file> -g <graph file> -c 2 -m 5000
runSHARC -f <sharc flag   file> -g <graph file> -c 1 -m 5000
```

The parameter `-c 2` indicates that runSHARC has to use a bloom filter. Thus, the first command is used by the method which utilized a bloom filter (Section 6.1). The parameter `-c 1` is used for all experiments in Section 6.2. As already mention, for every experiment 5000 random SHARC queries are executed. This is specified with `-m 5000`.

Setup. An overview of the graphs used in the experiments and some additional information regarding them are shown in Table 3. Especially the number of unique flags and the partition is important. The string in the last row shows how many cells on a given level. The number of cells of the lowest level is denoted by the left most number and the number of cells of the highest level is denoted by the rightmost number. Observe, that for all partitions there are exactly 128 cells. Thus, the size of each flag is 128 bit.

graph	unique flags	nodes	edges	partition
eur2dist	2,304,672	18,010,173	57,590,331	4-4-4-4-8-104
eur2time	948,663	18,010,173	53,375,388	4-4-4-4-8-104
eur2unit	1,151,629	18,010,173	55,505,351	4-4-4-4-8-104
grid 2dim	323,060	250,000	1,684,403	4-4-120
grid 3dim	856,994	250,047	2,138,911	4-4-120
unit disc (degree 5)	78,495	994,980	5,102,577	4-4-8-112
unit disc (degree 7)	760,112	996,394	8,268,245	4-4-8-112

Table 3: Information on the graphs used in the experiments

6.1 Bloom Filter

As mentioned in the Section 4 the number of bits set to one in all flags is an important factor, which influences the false positive rate. Thus, in the following sections only flags with a certain maximum number of bits set to one inserted into a bloom filter. Although the maximum number of bits set in the flags in the experiments are only between 5 and 12 bits (of a total of 128 bits per flag) there is still a significant number of flags, which fall into that category (cf. Table 4). The optimal number of hash functions are always calculated according to Equation (2).

Although mentioned, the ideas for reducing the number of bits set presented in Sections 4.3.1 and 4.3.2 are not implemented. The anchor approach reduced the number of bits set, but could not reduce it to a large degree. The chain variant is not implemented for the reasons stated in Section 4.3.2. So we only evaluate two different approaches. Both of them consider only flags with a certain maximum number of bits set. But the first one uses only one bloom filter and the second one uses one bloom filter for every level.

graph	number of flags with at most 10 bits set	total number of flags
eur2dist	667,819 (28.98%)	2,304,672
eur2time	397,772 (42.06%)	945,663
eur2unit	494,152 (42,91%)	1,151,629

Table 4: Number of flags with at most 10 bits set

6.1.1 Basic version

The first idea tested is based upon Section 4.2.1 and the idea explained in Section 4.3.3. Only flags with a relatively small number of bits set are inserted into the bloom filter. The other flags remain unchanged and stored in their original state.

Setup. The first experiment considers only flags with at most 5 bits set for the bloom filter. The number of flags satisfying this condition is determined and the size of the bit array of the bloom filter is set to the size of space consumption of the flags which are considered. Then, the flags are inserted into the bloom filter. In each iteration the size of the bit array of the bloom filter is repeatedly reduced by 5% of its original size. After every reduction 5000 random shortest path queries are executed by the SHARC algorithms and the average number of settled nodes is determined. This process is repeated with flags with at most 6 bits set, and then with at most 7 bits set, and so on until flags with at most 12 bits set are inserted into a bloom filter. The results of this with the graphs eur2dist, eur2time and eur2unit are depicted in Figures 4, 5 and 6.

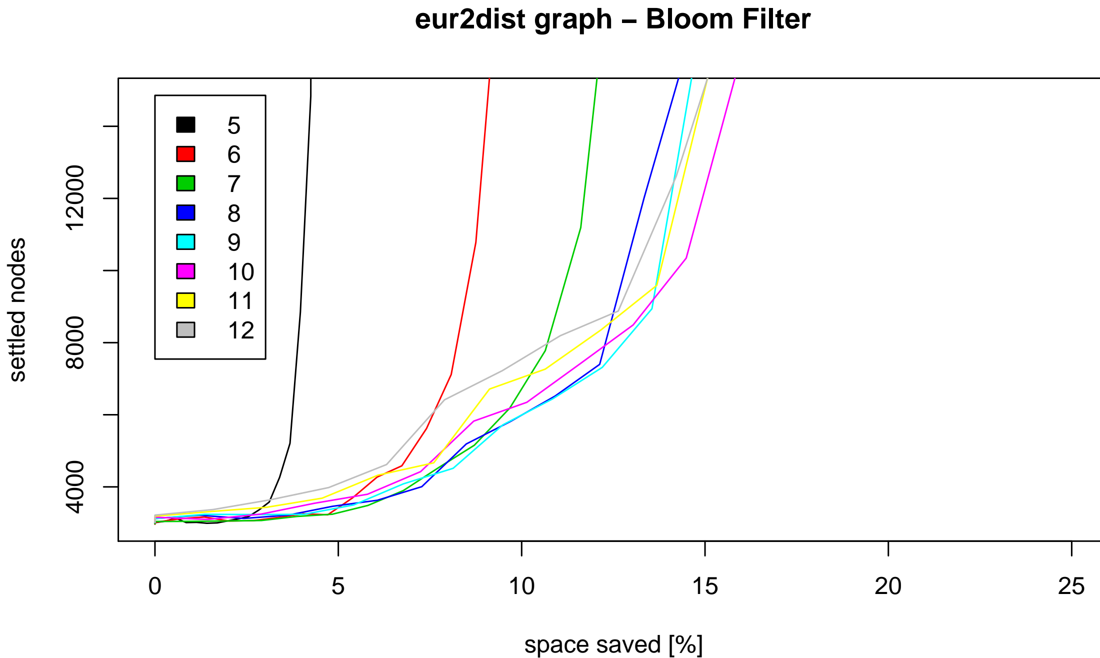

Figure 4: eur2dist graph with simple bloom filter. The numbers in the legend are the number of maximum bits set in the flags which are inserted into the bloom filter.

Results. If only flags with at most 5 bits set are inserted into a bloom filter the total space consumption can be reduced between 2.5% and 5% without a penalty. After that a sharp rise is identifiable.

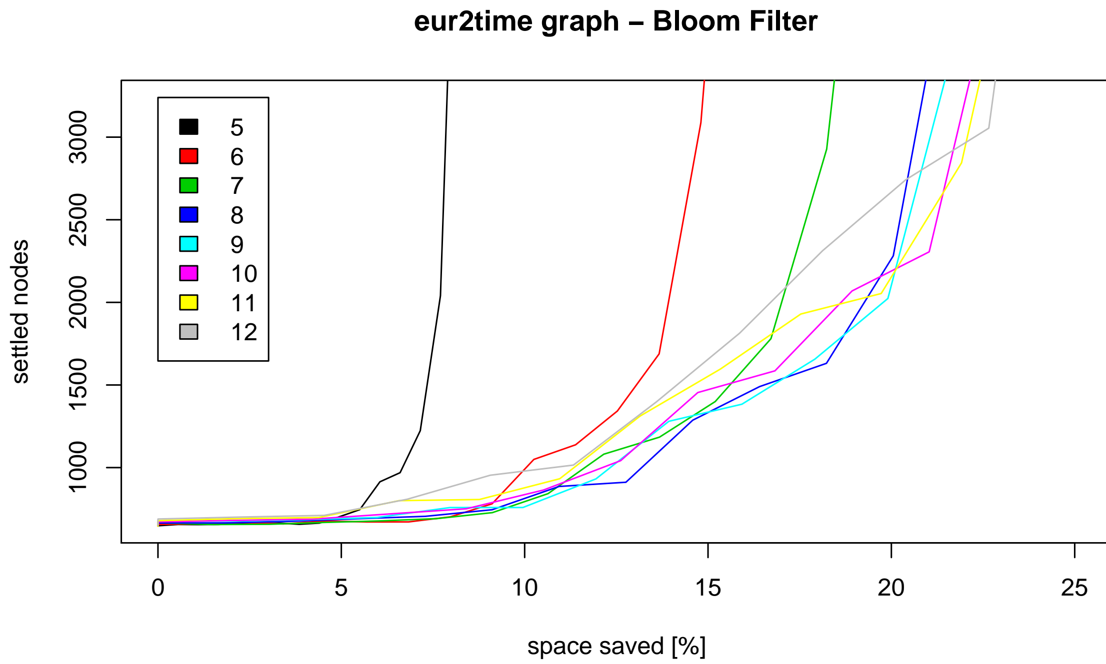

Figure 5: eur2time graph with simple bloom filter. The same properties for the legend as in Figure 4 apply.

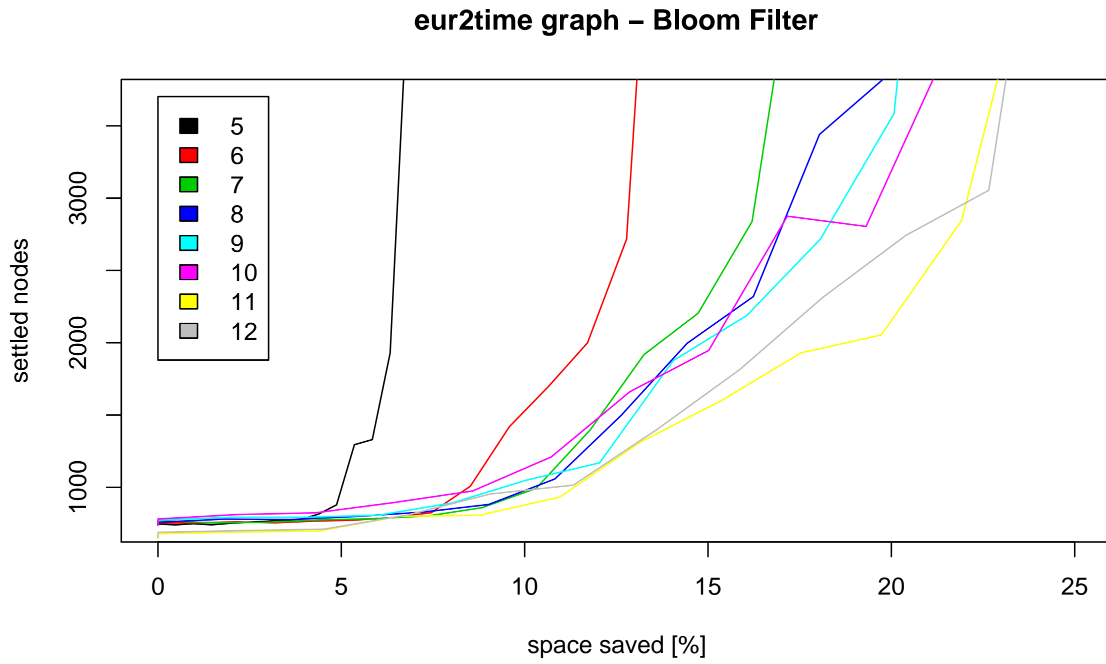

Figure 6: eur2unit graph with simple bloom filter. The same properties for the legend as in Figure 4 apply.

		best results	
graph	**compression**	**max number of bits**	**settled nodes(increase[%])**
	00%	-	3,044 (0%)
	05%	7	3,242 (6.50%)
eur2dist	10%	9	5,720 (87.91%)
	15%	10	10,345 (239.85%)
	20%	10	146,910 (4,726.22%)
	00%	-	642 (0%)
	05%	7	668 (4.05%)
eur2time	10%	9	758 (18.07%)
	15%	9	1,286 (100.31%)
	20%	9	2,023 (215.11%)
	00%	-	739 (0%)
	05%	11	833 (12.72%)
eur2unit	10%	11	1,089 (47.36%)
	15%	11	1,939 (162.38%)
	20%	11	3,760 (408.80%)

Table 5: Results for the basic bloom filter (c.f. 6.1.1) with different degrees of compression and with different graphs.

This is sharp rise follows after a certain threshold of compression, after which the false positive rate of the bloom filter rises significantly. This trend is noticeable with the other bloom filter as well. A higher compression can be reached, if more flags are inserted into the bloom filter. It is possible to reach a 5% compression in the eur2dist and a 10% compression in the eur2time and eur2unit graphs with a rise in the average number of settled nodes by about 10%.

Flags with at most 7-11 bits set seem to be the best candidates for this approach. If flags with more bits set are chosen the initial false positive rate of the bloom filter is already quite high. Although at a certain compression level using flags with a higher number of bits set delivers better results, the penalty for the number of nodes settled is then already relatively high. This can be seen in Figure 5. The approach where flags with at most 12 bis are used performs better than the other approaches only after a compression of about 22%. But at that point the number of nodes settled is about 5 times of its initial value.

The best results of this approach are depicted in Table 5.

6.1.2 Extended version

Setup. As mentioned in Section 4.2.2 it may be a good idea to manage a bloom filter for every level of the partition and "steal" bits from the bit array of the bloom filter for the lower levels and add them to the bit array of the highest level. The general idea of the following experiments is first to reduce the size of the bit array of the bloom filter that represents the lowest level repeatedly by 5% until the size has been reduced to 50% of its original size. After every reduction 5000 shortest path queries are executed by the SHARC algorithm with the resulting bloom filter. After the bit array of the first bloom filter has been reduced by 50% the bit array of the bloom filter representing the bits of the second lowest level is reduced repeatedly by 5% until its size is 50% of the original. This is repeated until the size of all bit arrays of all bloom filters is half of its original size.

Results. Unfortunately, the results are much worse than those of the basic version. A comparison between them and those of the basic version are depicted in Table 6. Those results are caused by an uneven distribution of bits set in the different parts of the flags which represent different cells of different level. As it turns out there are very few bits of the highest level set. The number of bits set of lower levels are set much more often. This leads to a high false positive rate for all bits of lower levels. This may happen even if the space for them is not compressed.

	compression			
	5%	10%	15%	20%
bloom filter basic	833	1089	1939	3760
bloom filter extended	27,649	27,992	30,798	31,987

Table 6: Comparison of both bloom filter variants discussed in Sections 6.1.1 and 6.1.2. Results are the average number of settled nodes of 5000 random SHARC queries - (eur2unit graph)

6.1.3 Discussion

It can be shown that if only one bloom filter is used it is possible to achieve a certain degree of compression (5-10%) of the space needed for the flags with only a small penalty regarding the average number of settled nodes of a SHARC query. Unfortunately, the use of multiple bloom filters, one for each level, and reducing the size of the bit array of the bloom filters of the lower levels proves to be a bad choice. The average number of settled nodes of a SHARC query rises sharply even for a small compression.

Although a certain degree of compression is possible this approach still suffers from some drawbacks. The running time of a shortest path query may be heavily influenced by the repeated hash computations. That and the insight, that the compression without a high penalty for the average number of settled nodes is relatively small leads us to the conclusion that this approach is not very useful.

6.2 Compression

The following sections show the experimental results of the remapping process according to Section 5 with several different graphs and cost functions. To give the reader a better “feeling” on how the different results compare the y-axis of all diagrams shows the interval from the lowest value to three times of that value (except stated otherwise).

6.2.1 Confirming the initial Assumption

In the last paragraph of Section 5.2 three properties are named that seem logical to be a good influence on the cost function. Those three are

1. The number of times a flag is referenced in the graph should be as low as possible
2. The number of bits which have to be changed due to the remapping process should be as low as possible
3. The bits which are changed should represent cells with a low level

To confirm that assumptions 2 and 3 are correct several different bitflip cost functions are tested. Each of those assigned a certain cost to every bit of a flag, which has to be flipped from 0 to 1 due to the remapping. The bitflip cost functions are identified by string of the from “A_B_C_D_E_F” with one letter for every level of the partition. Every letter represents the costs for changing one bit of its corresponding level from zero to one (bitflip costs). The bitflip costs are ordered from left to right from bitflip cost of bits of the lowest level to bitflip cost of bits of the highest level. For the example depicted in Figure 3 this string would be “1_2”.

In the diagrams (Figures 7, 8 and 9) experiments are shown with different bitflip cost functions and a fixed the weighting factor o (c.f. Section 5.4) value of 1. The x-axis depicts the percentage of flags removed and the y-axis depicts the number of nodes settled by the SHARC algorithm. All three diagrams display roughly the same trend. The bitflip cost function “1_2_4_8_16_32” and “1_3_8_27_84_243”, which both assign higher costs for flipping bits of higher levels deliver far better results than all others. Neglecting the importance of the level and assigning every bit the same bit flip cost (“1_1_1_1_1_1”) leads to a rise of the number of nodes settled with a higher number of flags

remapped. Weighting the bits representing the cells of levels which are neither the lowest nor the highest provides worse results. The worst results are achieved by assigning the lowest costs to bits representing the cells of the highest level ("32_16_8_4_2_1"). Thus the initial assumption is confirmed and we only study bitflip cost functions, which are similar to "1_2_4_8_16_32" and "1_3_8_27_84_243". We can now be reasonably sure that the assumptions 2 and 3 are indeed correct. It remains to test if assumption 1 proves to be true.

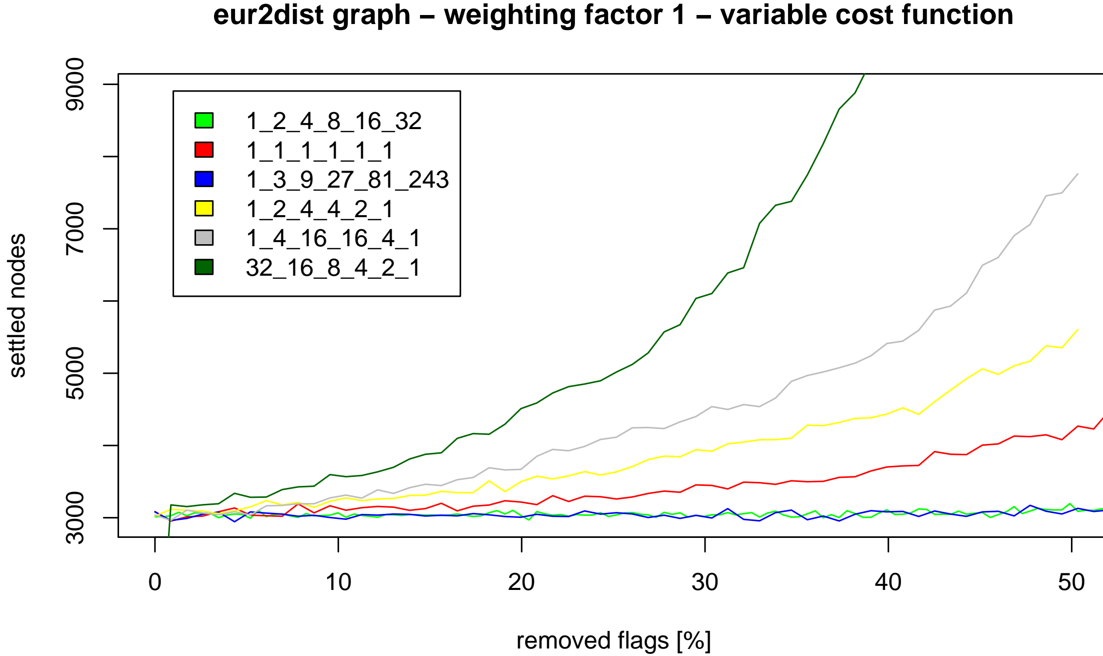

Figure 7: eur2dist graph - The legend depicts the different bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

In Figures 10, 11 and 12 different values for the weighting factor o combined with the bitflip cost function "1_2_4_8_16_32" are depicted. Again we see similar trends in the diagrams. Obviously, the value 0 (i.e. neglecting the number of times a flag is referenced) delivers results not as good as the rest, which confirms assumption 1. A high value for o seems to cause a earlier rise in the number of nodes settled by the SHARC algorithm with a increasing number of flags mapped, than values between 0 and 1 (not including 0). Those seem to be the best choices, but a precise "best" value among them is not easy to identify.

The diagrams (Figures 7, 8 and 9) suggest that this approach may deliver very good results considering that with a "good" cost function and 50% removed flags there is no penalty concerning the number of settled nodes. It remains to be enquired if this is just a random result or if it can be reproduced with other graphs.

6.2.2 Comparison of good cost functions

The Figures 13, 14 and 15 show several good bitflip cost functions with the weighting factor 1. The general trend in all graphs is nearly identical. Until about 60% of all flags have been remapped, the different bitflip cost functions deliver results which are good (i.e. a rise in settled nodes is barely noticeable) and are basically the same. It does not seem to matter which bitflip cost function is chosen. After the 60% mark has been passed the results start to be distinct from one another even though most of them do not differ much. The bitflip cost function "1_2_4_8_16_32" delivers the worst results among all different bitflip cost functions with about 80% flags removed. This is most likely

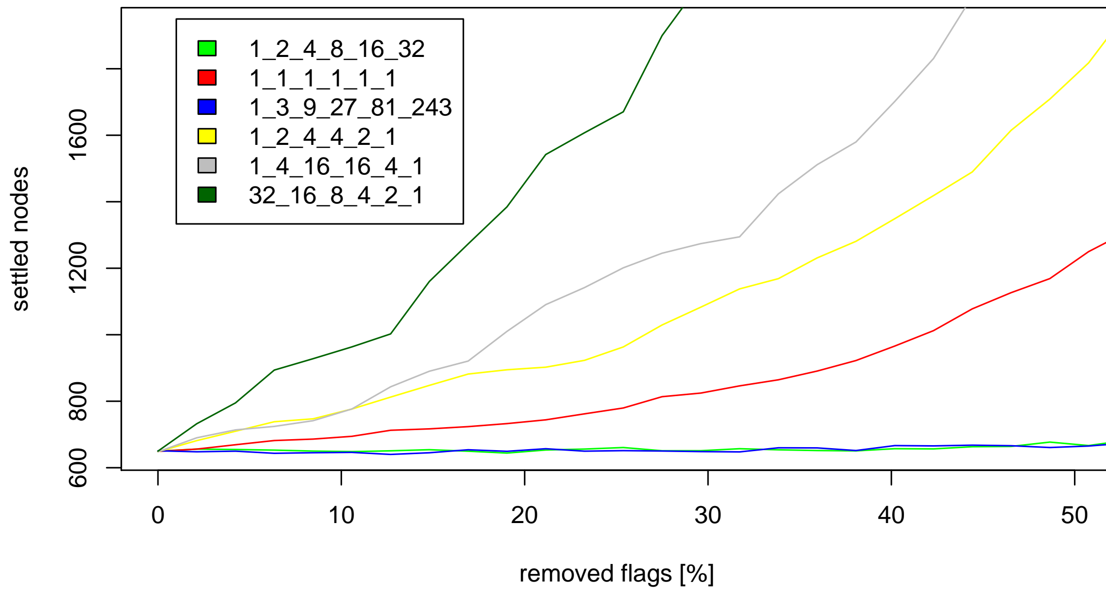

Figure 8: eur2time graph - The legend depicts the different bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

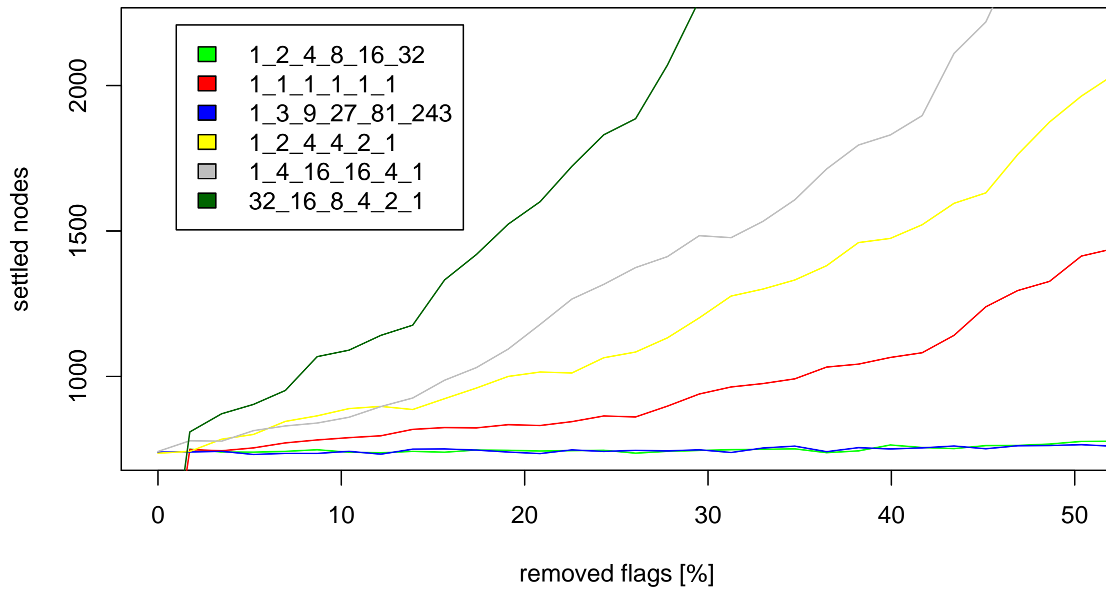

Figure 9: eur2unit graph - The legend depicts the different bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

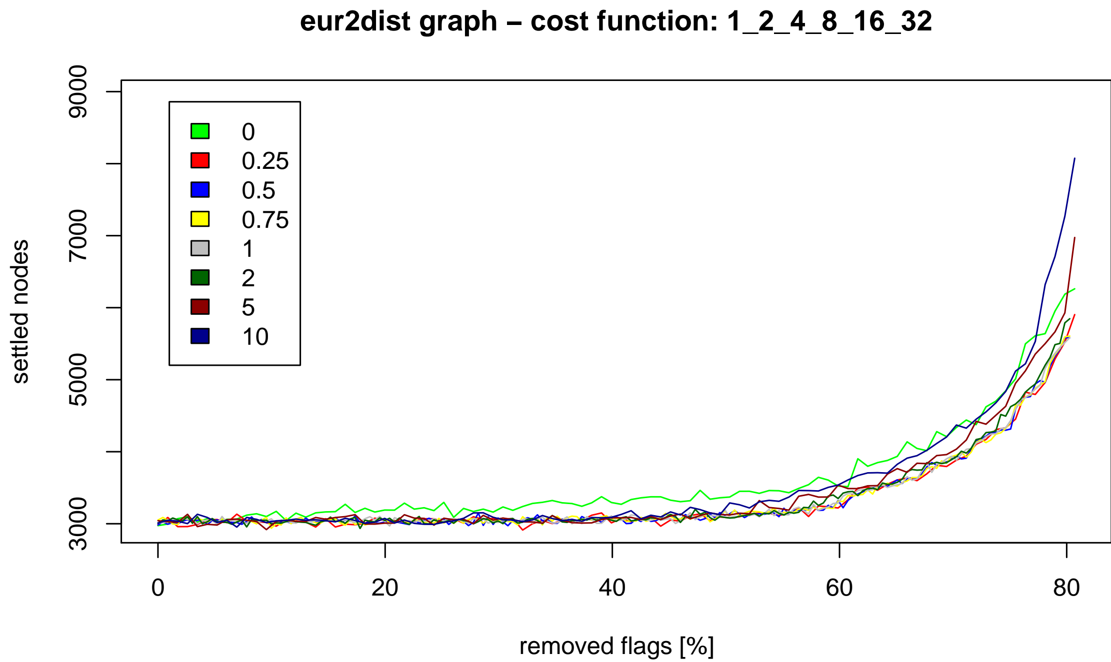

Figure 10: eur2dist graph - The legend depicts the different values used for the weighting. The bitflip cost function is "1_2_4_8_16_32".

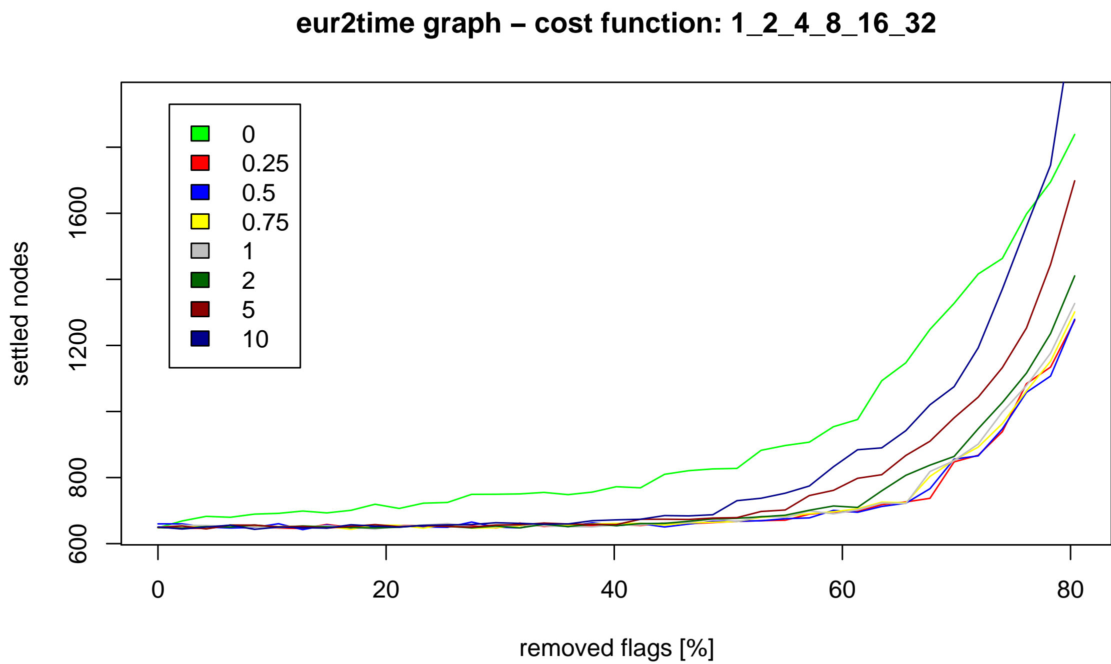

Figure 11: eur2time graph - The legend depicts the different values used for the weighting. The bitflip cost function is "1_2_4_8_16_32".

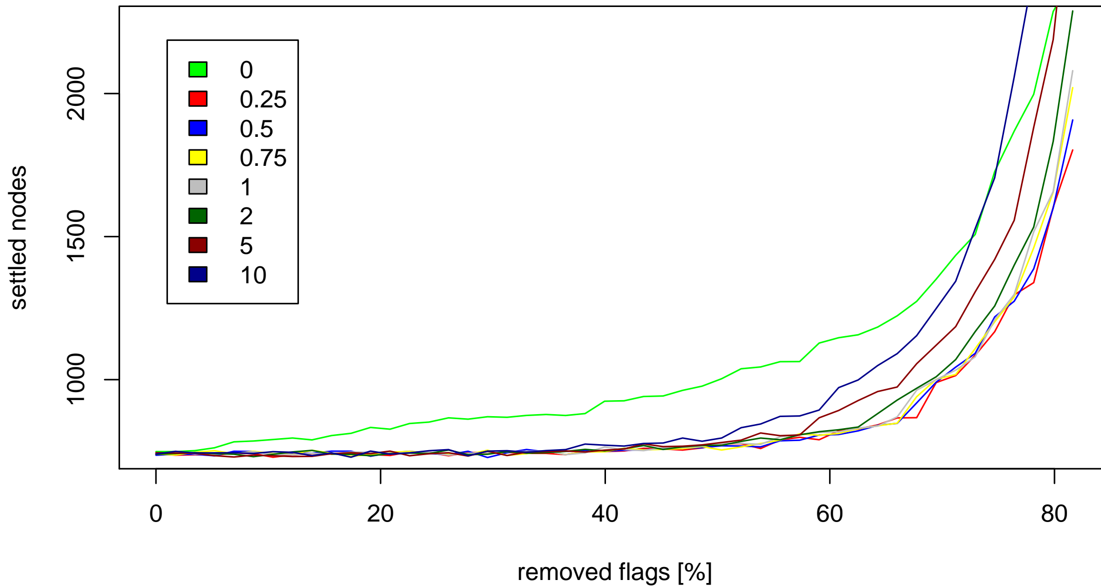

Figure 12: eur2unit graph - The legend depicts the different values used for the weighting. The bitflip cost function is "1_2_4_8_16_32".

due to the fact, that bits which represent cells of the highest level get flipped earlier than in all other cost functions. On the other hand the bitflip cost function "1_2_4_8_16_128", which heavily weights those bits also does not perform that well in comparison to the other functions. The most reasonable explanation for this is, that because flipping a bit representing the highest level is very expensive flags with less of those bits but are referenced more often are much more likely to be remapped. This means a tradeoff between the weighting of bits of higher regions and the weighting of the times a flag is referenced by the edges has to be found. The bitflip cost function "1_3_9_27_81_243" with the weighting factor of 1 delivers consistently good results and may represent such a tradeoff.

6.2.3 Other road networks

As already mentioned, it remains to examine, if good bitflip cost functions illustrated in the previous sections are good bitflip cost functions for other road networks or entirely artificial graphs.

The graphs used for the next experiments are road networks of the united states. All of them have the same 6 level partitioning. Different good bitflip cost functions with the weighting factor of 1 are used. The results are depicted in Figures 16, 17 and 18. The trend resembles those in the diagrams of the european road network. A 60% compression can be reached without a sharp rise of the number of settled nodes. After the 60% mark the results clearly worsen, although not that much compared to the european road network graphs. Especially the usa_dist graph with the bitflip cost function "1_3_9_27_81_243" leads after 80% of all flags have been removed to "only" an increase of settled nodes of about 50%.

All those results suggest, that for a road network the remapping process can lead to a serious reduction of unique flags and thus to a significant reduction in space needed for storing them. It remains to examine if those good results are reproducible for artificial graphs.

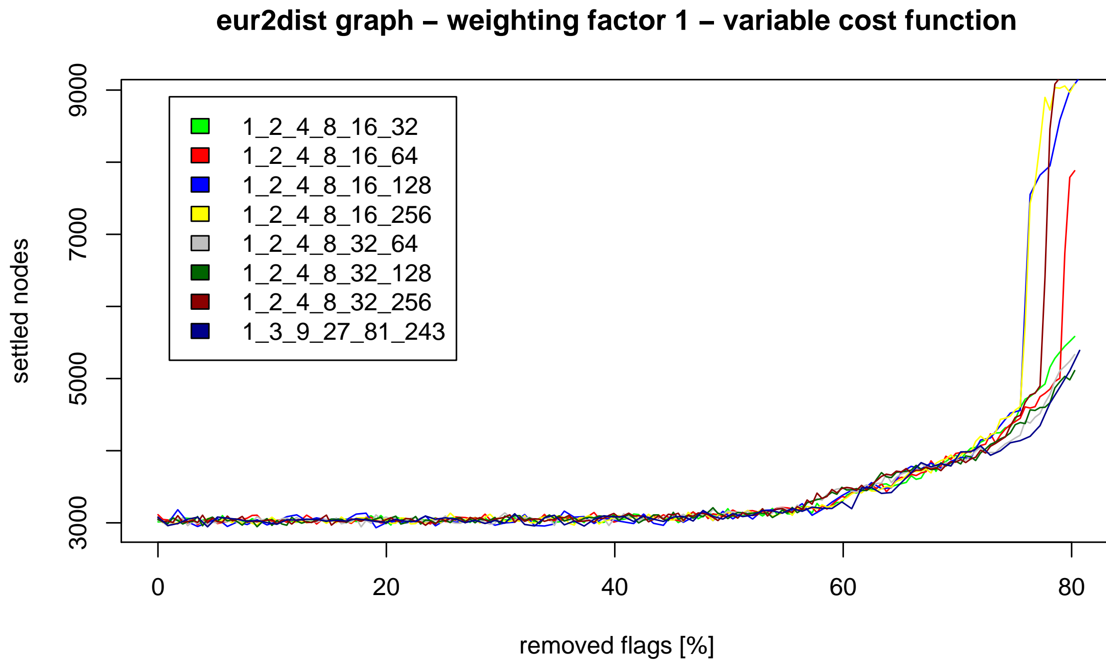

Figure 13: eur2dist graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

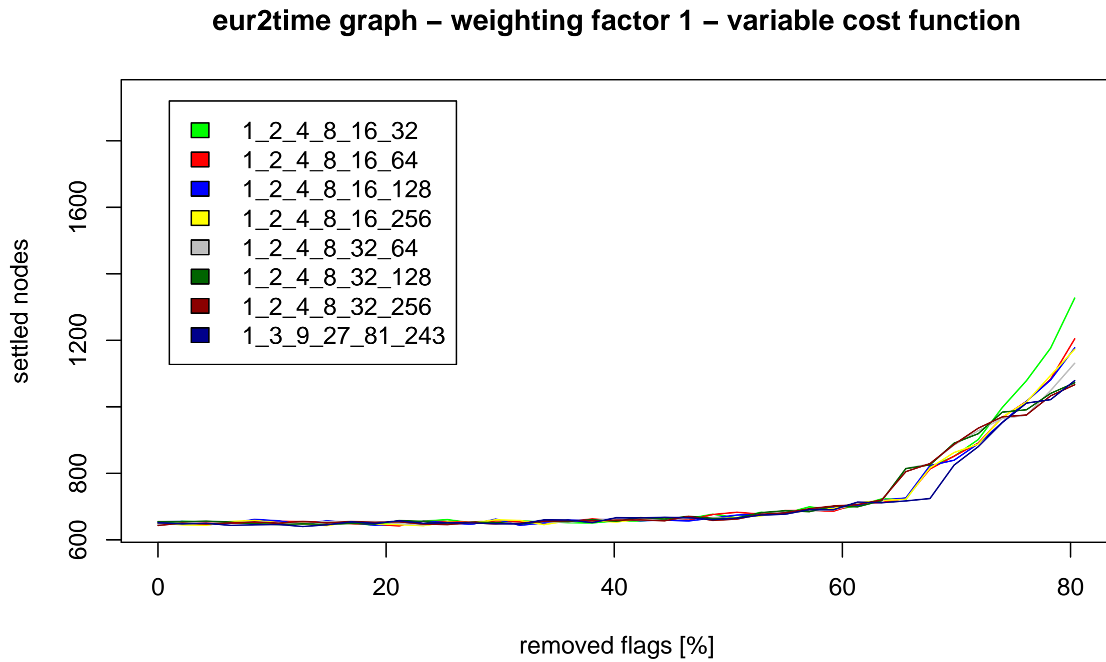

Figure 14: eur2time graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

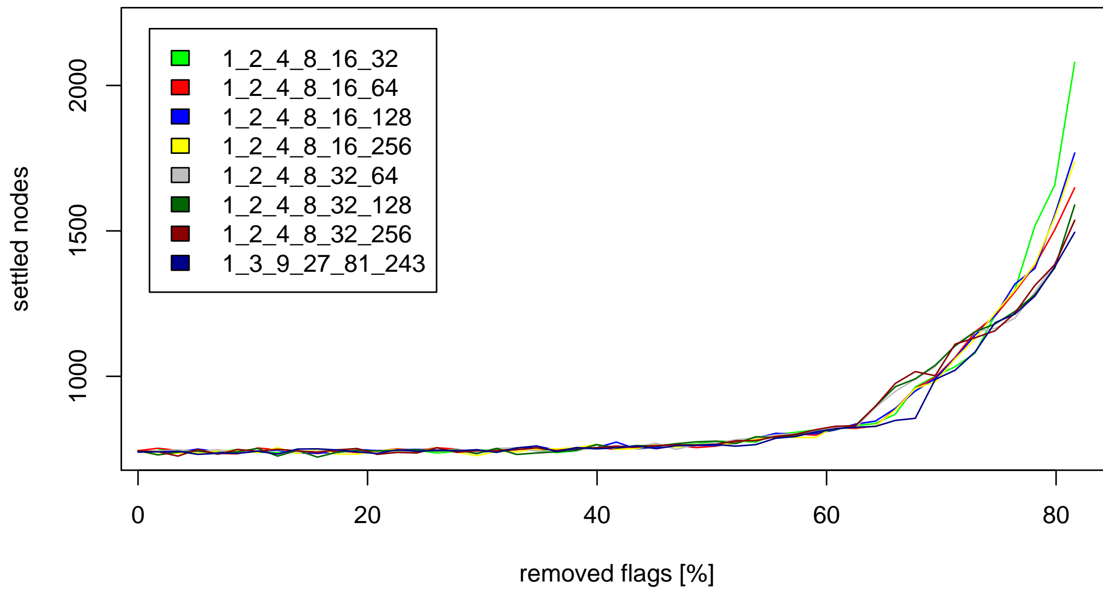

Figure 15: eur2unit graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

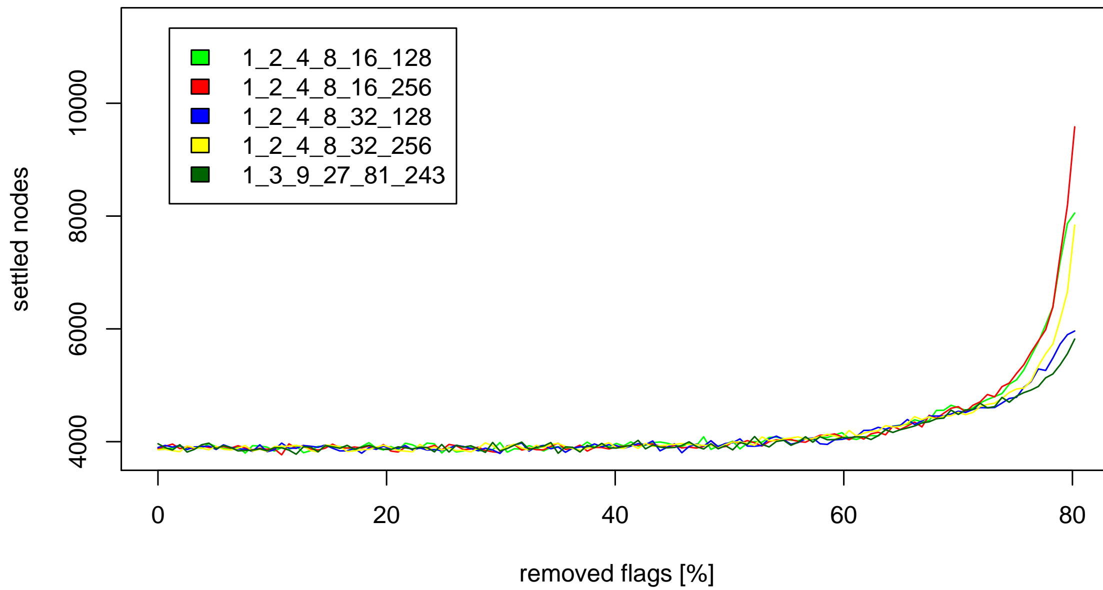

Figure 16: usa dist graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

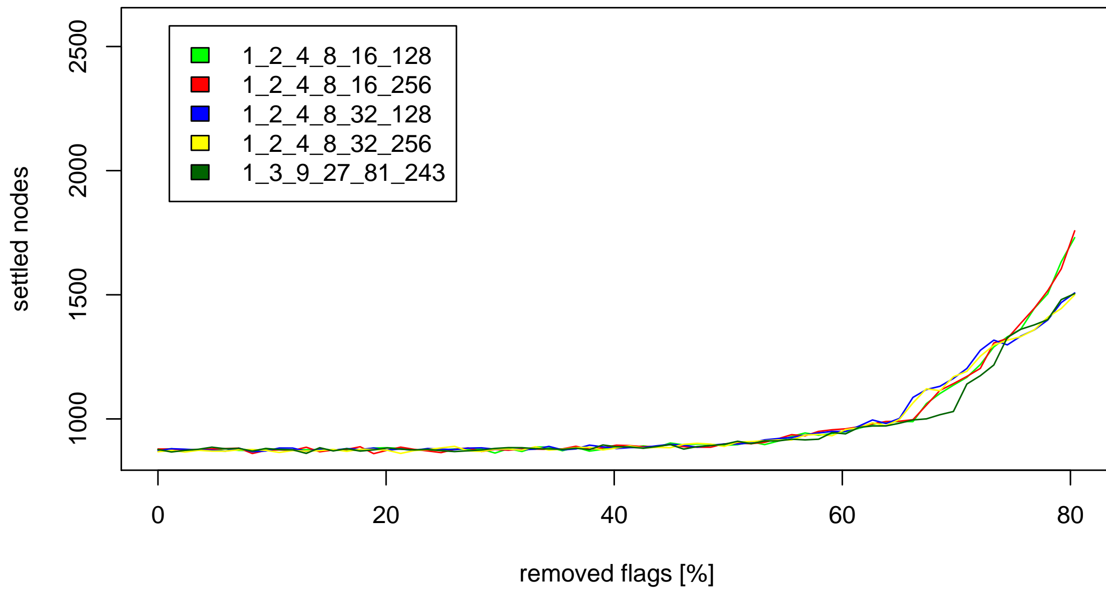

Figure 17: usa time graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

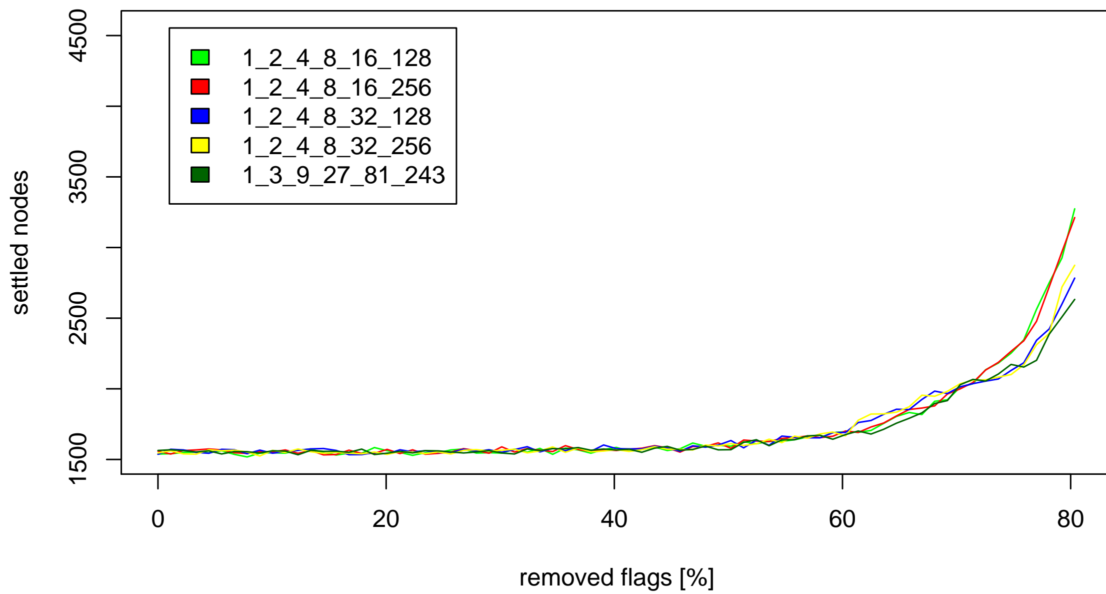

Figure 18: usa unit graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

6.2.4 Artificial Graphs

Four different artificial graphs are tested. Two different unit disc graphs and two different grid graphs are used (cf. Table 3).

The results for the degree 5 unit disc graph (Figure 19) are similar to those of the road network graphs. The results of all different bitflip cost functions are so similar that it does not seem to matter which one is chosen. This might be the case because there are not many unique flags (only 78,495) in this graph and thus there are not many candidates for the remapping process. Illustrated in Figure 20 are the results for the degree 7 unit disc graph. They are not as good as the results of the degree 5 unit disc graph, but about 40% of the flags can be remapped without a penalty. This is still a good result. In this case the results for different bitflip cost functions differs noticeable in the interval between a 55% and a 75% remapping of flags. Before and after that they seem to be produce very similar results. A cause for this is not apparent.

Figure 21 displays the results for the remapping of the unique flags for a 2 dimensional grid with a 3 level partitioning. The results again are very similar to those of the road network graphs, but two bitflip cost functions deliver far worse results than the other three, after the 60% remapping mark has been reached. As both of them ("1_2_32" and "1_4_32") very heavily weight bits of the highest level and the others (i.e. the ones that produce better results) do not this behaviour may be caused by remapping too many flags which are referenced often instead of flags which are referenced far less but are populated with some high level bits.

For the three dimensional grid (again with a 3 level partitioning) the results (Figure 22) of the bitflip cost functions do not seem to produce different results from one another. After 40% of the flags have been remapped a nearly linear increase in the number of settled nodes can be seen. Before the 30% mark a negative effect is barely noticeable.

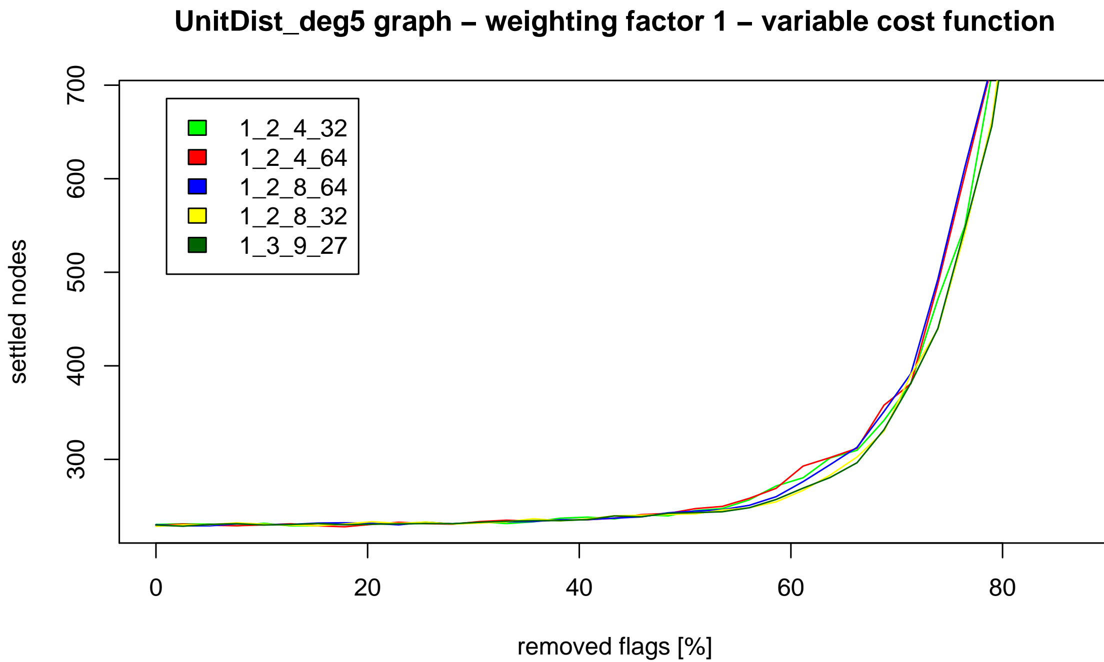

Figure 19: Unit Disc (degree 5) graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

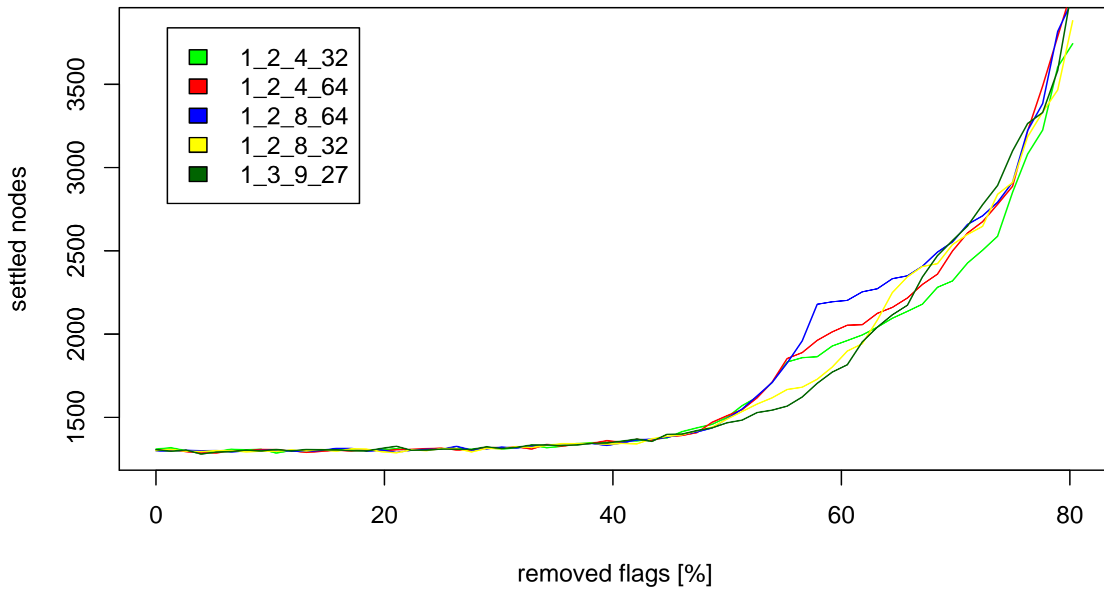

Figure 20: Unit Disc (degree 7) graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

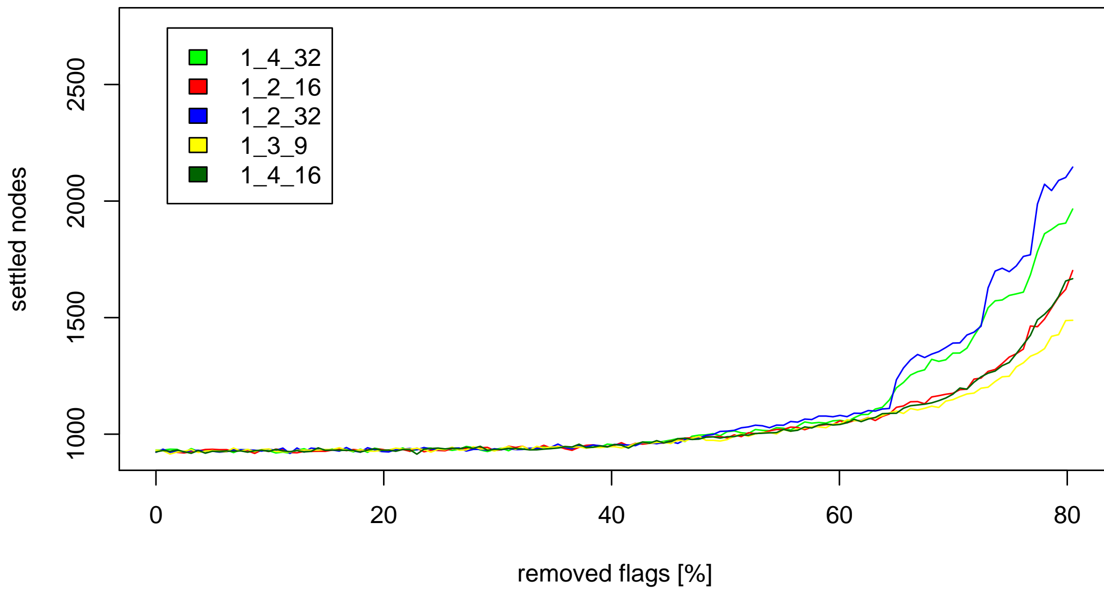

Figure 21: Grid (2dim) graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

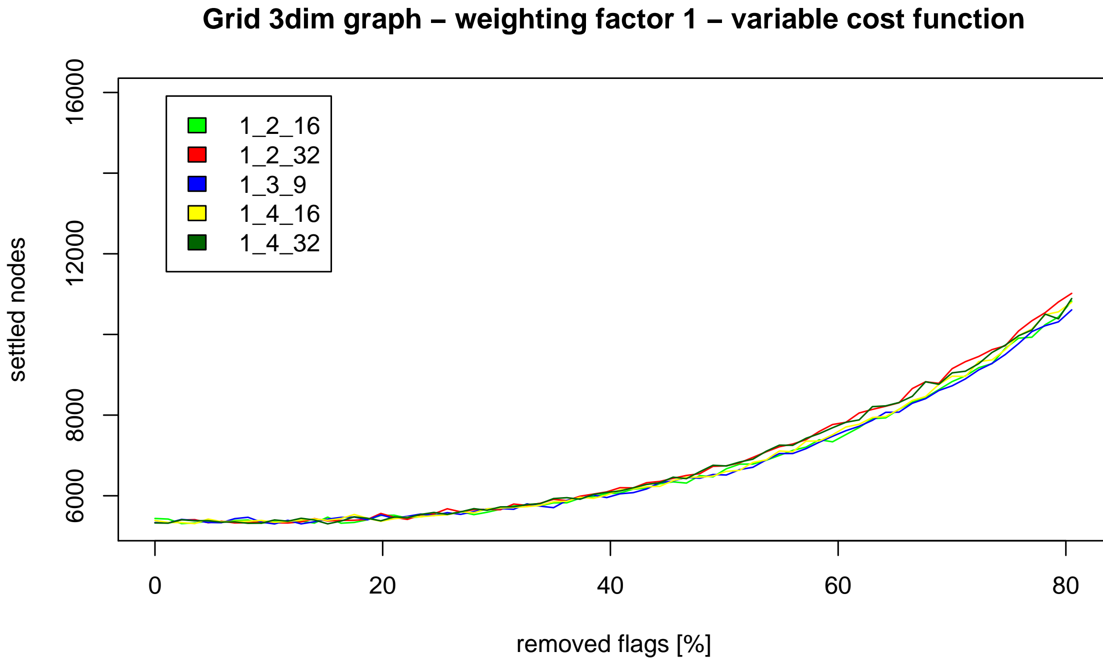

Figure 22: Grid (3dim) graph - The legend depicts the different good bitflip cost functions as explained in Section 6.2.1. The weighting factor of all cost functions is 1.

6.2.5 Detailed comparison and analysis

To understand how certain cost functions perform, a more detailed analysis is needed. In Figure 23 the y-axis depicts the number of bits of a certain level, which are flipped. The x-axis depicts, as before, the percentage of flags remapped. As an example we compare only two different bitflip cost functions. Most of the other good bitflip cost functions behave like one of the two shown. In the diagram (Figure 23) those two different bitflip cost functions with the weighting factor 1 are shown. One is the very good performing "1_3_9_27_81_243" bitflip cost function and the other is the bitflip cost function "1_2_4_8_16_128" (cf. 24), which performs not as good as the one before. The difference is only noticeable in the interval between 60% and 80% remapped flags. The bitflip cost function "1_3_9_27_81_243" does change much more bits from the three lowest levels form 0 to 1 than the other one. The biggest difference can be seen in the number of bits of level 6. Bits of level 6 are flipped much less by that bitflip cost function. This is leads to a decreased number of unnecessary considered edges by the SHARC algorithm (compared to the behaviour of the other bitflip cost function) and is most likely the cause for the better performance of "1_3_9_27_81_243".

If we compare the good performing "1_3_9_27_81_243" with the cost function "1_1_1_1_1_1" which performs far worse, we see the reason why it performs so poorly (Figure 25). As one might expect the number of bits changed from zero to one of the latter bitflip cost function is throughout every level nearly the same. There is no resemblance to the results of the better performing bitflip cost function. Interestingly, after about 35% removed flags, the bits of the highest level get changed far more often than the other ones. This might be due to the uneven distribution of bits in the flags which is mentioned in Section 6.1.2. Thus it may be, that flags differ mostly in the parts representing the lower level and a this causes the high level bits to be flipped more often.

If we want to focus on the weighting factor o, we can see in Figure 26 the bitflip cost function "1_2_4_8_16_64" with three different values for the weighting factor. It is obvious that the weighting factor $o = 1$ delivers the best results. In Figure 27 we can see from what this derives. Applying such a weighting factor, we observe the following: Bits representing upper levels are flipped at a later point,

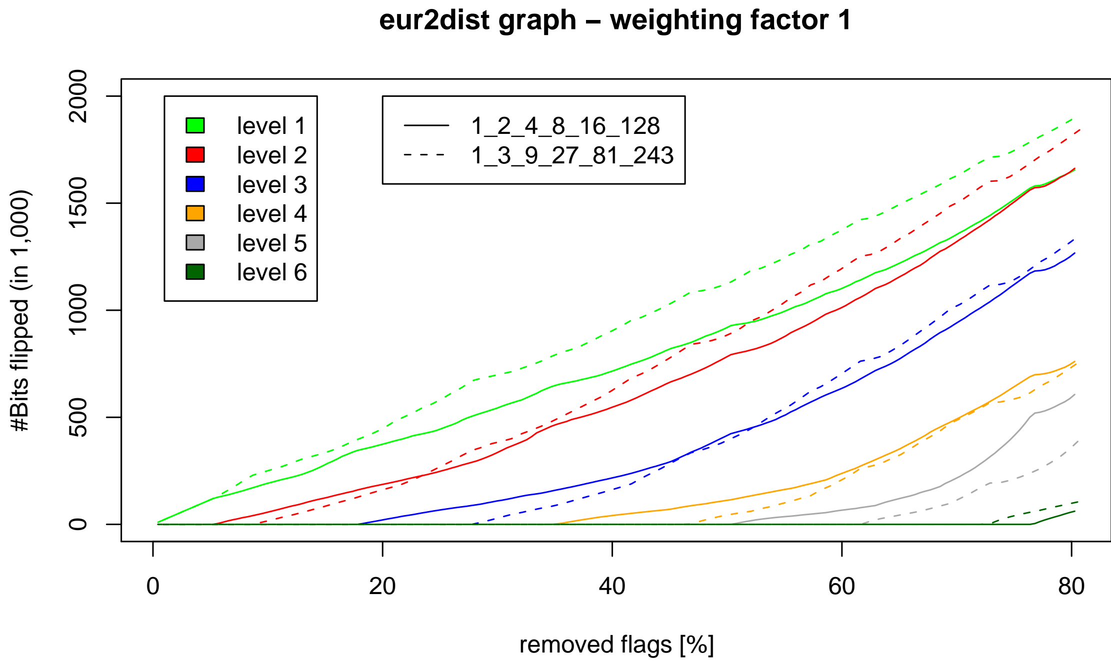

Figure 23: eur2dist - Comparison between two bitflip cost functions

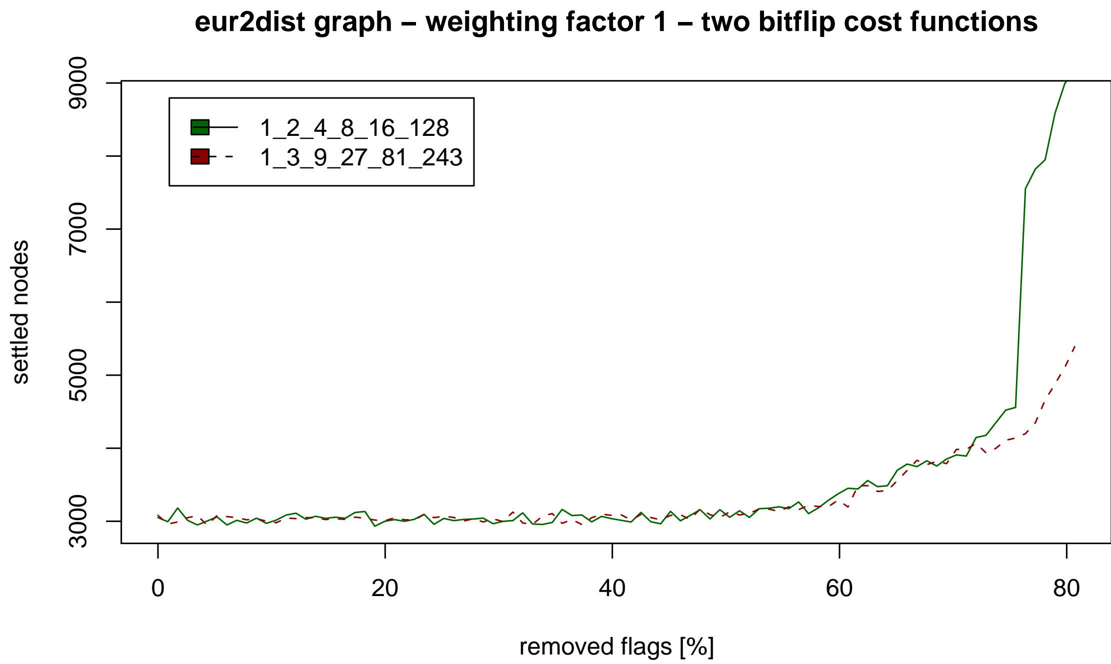

Figure 24: eur2dist - Comparison between two bitflip cost functions regarding the average settled nodes of 5000 random SHARC queries.

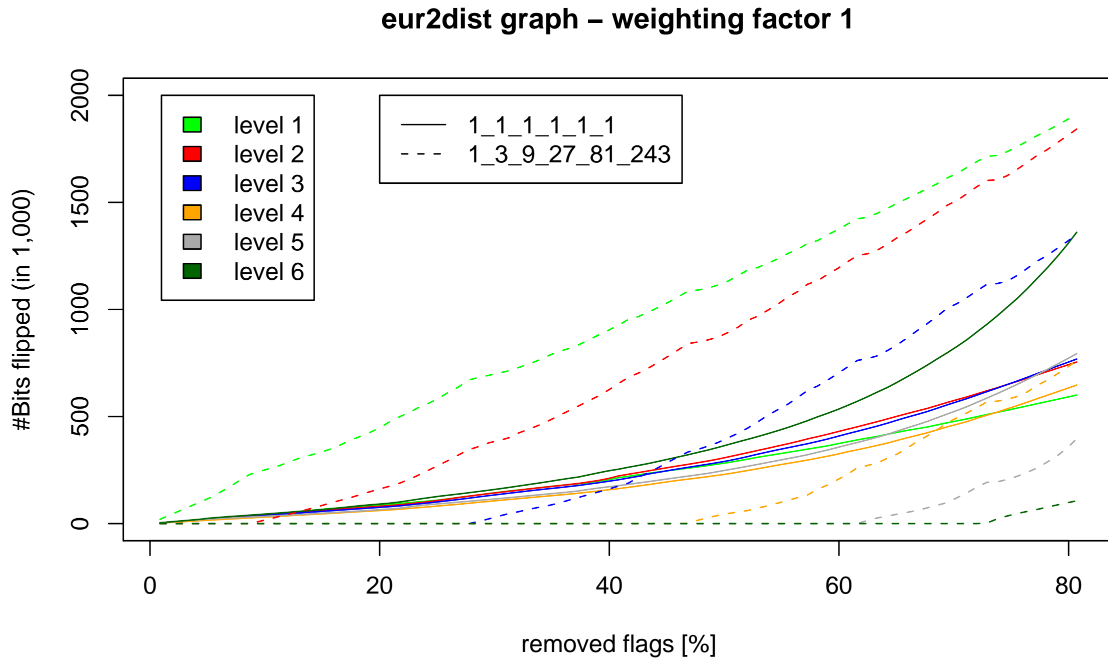

Figure 25: eur2dist graph - Comparison between a good and a bad performing bitflip cost function. It is shown how many bits of which level are flipped by the remapping process.

and less of them are flipped. This is caused by the fact that a high value for o favours the remapping of flags which are referenced rarely but may have even high level bits set. Flags which are referenced more often are not favoured for the remapping process. Flipping high level bits later delivers better results because most of a shortest Path query is carried out at the topmost level.

Altogether this leads to the insight that, as we have seen at the beginning of this section, the number of bits of the lower levels is not very important. It may be even logical to make it even cheaper to flip bits of those levels. But the changing of bits of the highest level should not be much more expensive than the changing of bits of the second highest level. This is done by the bitflip cost function "1_2_4_8_16_128" and delivered not as good results as other bitflip cost functions. The weighting factor, as already mentioned in Section 6.2.1, should not be too high.

6.2.6 Best results

Three comprehensive tables (7, 8 and 9) show best results for every graph with the cost function which produced those results. They suggest, that a 50% removal of flags of graphs representing road networks, does lead to only a very small penalty. Even a removal of 60% of all flags can be achieved with only a small rise (2.67% to 14.71%) in the number of settled nodes by the SHARC algorithm. If more flags are removed the results begin to worsen significantly.

The results of the artificial graphs are not as good but are still remarkable. The increase of settled nodes per search query rises are between 12.03% and 49.81%. Table 9 suggest, that a 80% reduction of the number of unique flags leads to a significant increase (between 59.82% and 708.3%) in the number of average nodes settled by the SHARC algorithm. The results for the artificial graphs show an even higher increase of settled nodes. If 50% of all flags are removed the results are, compared to those of the road networks, not as good but are still exceptional (increase between 4.83% and 22.12%). Although applying the remapping technique delivers better results with graphs of road networks, reducing the number of flags by one half and gaining only such a small penalty is remarkable. If even less flags of

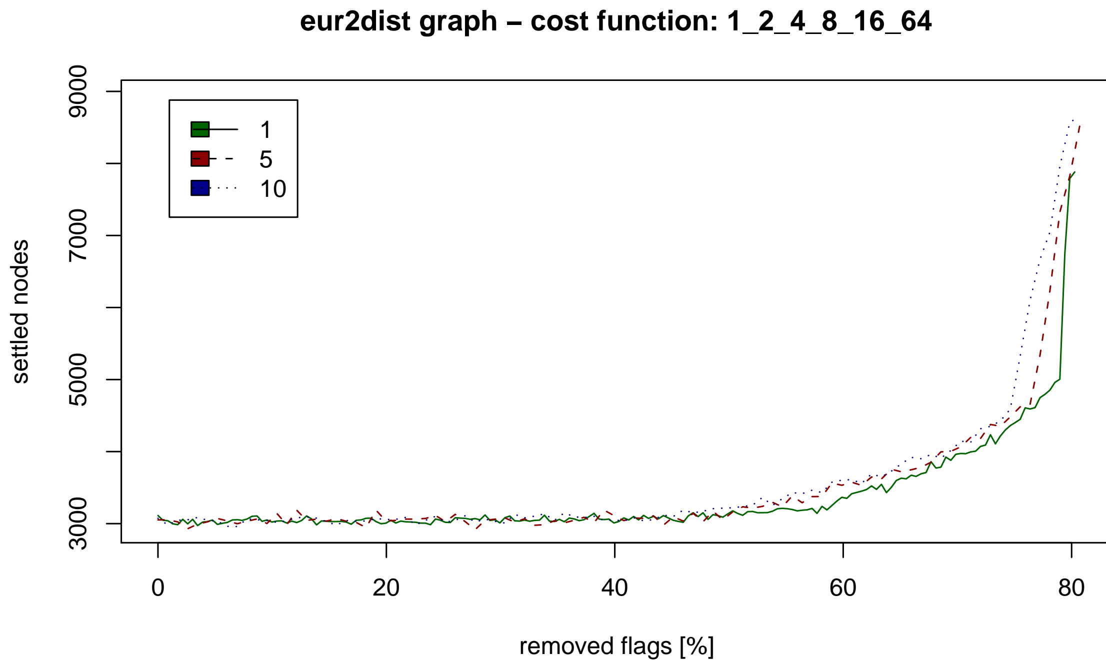

Figure 26: eur2dist graph - A Comparison between different weighting factor regarding the average settled nodes of 5000 random SHARC queries.

the artificial graphs are removed (between 30%-40%) a rise in the number of settled nodes is hardly noticeable. Hence, the running time of a shortest path query differs not from the one that uses all arc-flags. So even for artificial graphs, this approach can lead to a substantial decrease in unique flags without a penalty regarding the running time.

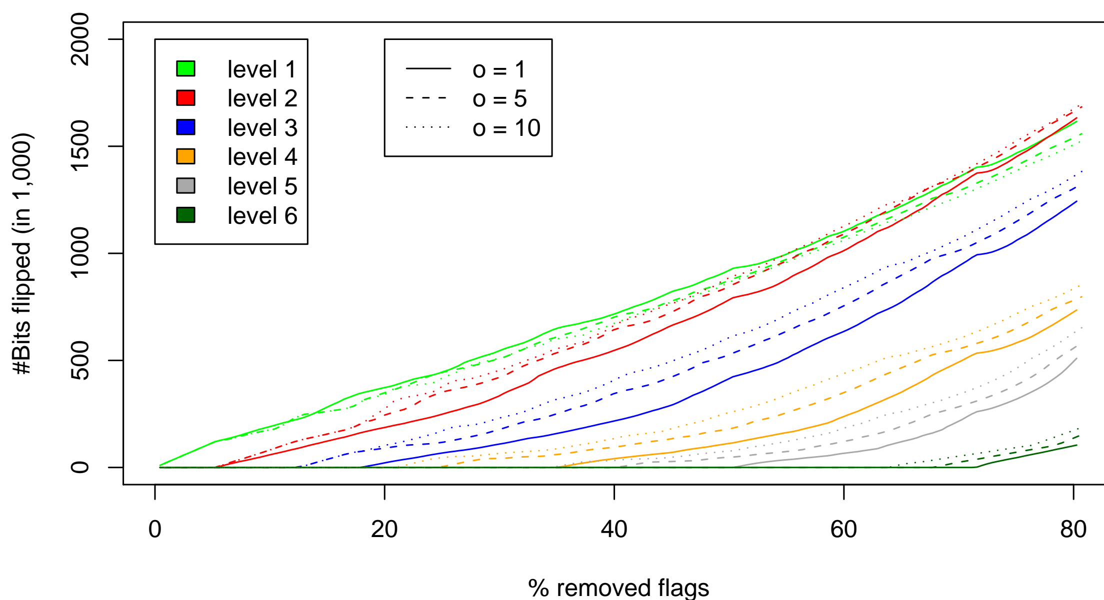

Figure 27: eur2dist graph - Comparison between three different weighting factors for the bitflip cost function “1_2_4_8_16_64”. It is shown how many bits of which level are flipped by the remapping process.

		best results		
graph	**removed flags**	**bitflip**	**weighting**	**settled nodes(increase[%])**
eur2dist	00%	-	-	3044 (0%)
	50%	1_3_9_27_81_243	1	3086 (1.38%)
	60%	1_2_4_8_32_128	1	3492 (14.71%)
	70%	1_2_4_8_32_128	1	3816 (19.12%)
	80%	1_2_4_8_32_128	1	5110 (67.87%)
eur2time	00%	-	-	642(0%)
	50%	1_3_9_27_81_243	1	660 (2.80%)
	60%	1_2_4_8_32_256	1	701 (9.20%)
	70%	1_3_9_27_81_243	1	824 (28.35%)
	80%	1_2_4_8_32_256	1	1066 (66.04%)
eur2unit	00%	-	-	739 (0%)
	50%	1_3_9_27_81_243	1	765 (3.52%)
	60%	1_3_9_27_81_243	0.5	823 (14.71%)
	70%	1_3_9_27_81_243	0.5	1024 (38.57%)
	80%	1_3_9_27_81_243	0.5	1443 (95.26%)

Table 7: Best results for european road network graphs

		best results		
graph	**removed flags**	**bitflip**	**weighting**	**settled nodes(increase[%])**
	00%	-	-	3899 (0%)
	50%	1_3_9_27_81_243	1	3928 (0.74%)
usa dist	60%	1_3_9_27_81_243	1	4070 (4.39%)
	70%	1_3_9_27_81_243	1	4482 (14.95%)
	80%	1_3_9_27_81_243	1	5819 (49.24%)
	00%	-	-	876 (0%)
	50%	1_3_9_27_81_243	1	909 (3.76%)
usa time	60%	1_3_9_27_81_243	1	979 (11.76%)
	70%	1_3_9_27_81_243	1	1029 (17.47%)
	80%	1_3_9_27_81_243	1	1504 (71.69%)
	00%	-	-	1563 (0%)
	50%	1_3_9_27_81_243	1	1569 (0.38%)
usa unit	60%	1_3_9_27_81_243	1	1676 (7.23%)
	70%	1_3_9_27_81_243	1	2028 (29.75%)
	80%	1_3_9_27_81_243	1	2638 (68.78%)

Table 8: Best results for usa road network graphs

		best results		
graph	**removed flags**	**bitflip**	**weighting**	**settled nodes(increase[%])**
	00%	-	-	931 (0%)
	50%	1_3_9	1	976 (4.83%)
grid 2dim	60%	1_3_9	1	1043 (12.03%)
	70%	1_3_9	1	1161 (24.70%)
	80%	1_3_9	1	1488 (59.82%)
	00%	-	-	5334 (0%)
	50%	1_3_9	1	6514 (22.12%)
grid 3dim	60%	1_3_9	1	7613 (42.72%)
	70%	1_3_9	1	8726 (63.60%)
	80%	1_3_9	1	10610 (98.91%)
	00%	-	-	229 (0%)
	50%	1_3_9_27	1	244 (6.55%)
unit disc deg 5	60%	1_3_9_27	1	296 (29.26%)
	70%	1_3_9_27	1	547 (138.86%)
	80%	1_3_9_27	1	1851 (708.30%)
	00%	-	-	1306 (0%)
	50%	1_3_9_27	1	1467 (12.33%)
unit disc deg 7	60%	1_3_9_27	1	1815 (38.97%)
	70%	1_2_4_32	1	2319 (77.57%)
	80%	1_2_4_32	1	3743 (186.60%)

Table 9: Best results for artificial graphs

7 Conclusion

In this paper two different mechanism of reducing space needed for storing arc-flags have been studied. One is based on a probabilistic data structure, the bloom filter. The other approach removes flags and changes all references inside the graph accordingly.

Some ideas on how to use the bloom filter to reduce the space needed for storing the flags have been presented. Unfortunately most of them proved not to be a good choice. Only one idea lead to a positive result. With it, we are able to save up to 10% space, without invoking to much penalty for the running time of a shortest path query using SHARC. But it suffers from two major disadvantages. First, the SHARC query algorithm itself has to be modified to use the bloom filter. Second, bloom filters yield a higher overhead due to the hash operations. Hence, running time increases.

The second approach, the deletion of some arc-flags and changing the indices pointing to them to other arc-flags, proves to be a far better choice. Three general characteristics of good cost functions, which help to determine the best candidates for removal, have been explained. Furthermore, a more detailed analysis of those characteristics has been been presented. With this approach it is possible to remove *50-60%* of all flags of the tested road network graphs with only a small penalty concerning the number of settled nodes and thus the running time. A very appealing factor of this approach is, that it does not suffer from the disadvantages of one using the bloom filter. There is no need to modify any code of the SHARC algorithm and thus there is no additional overhead during a shortest path query. The computational effort lies completely in the preprocessing phase, but for large graphs it may be significant, even though a fairly quick way of finding the mapping target with minimal mapping cost has been devised.

Future Work. SHARC introduces two forms of overhead. One of them are the arc flags. The other overhead factor is caused by the shortcuts. This paper dealt with the former overhead problem and showed one mechanism which yields very high compression rates for the arc-flags with only a very small penalty regarding the running time of a SHARC query. The overhead posed by shortcuts is a problem which might be yield enormous potential for space consumption improvement. Recall that the graph is enriched by additional edges and nodes which can lead to a significant rise in the size of the graph. This is even more problematic for the time dependent SHARC variant, which is not discussed in this paper. Information regarding it can be found in [Del09]. It seems to be a good idea to test if it is possible to determine if there are less important shortcuts in a graph which can be removed without deteriorating the running time of a SHARC query too much.

References

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. Invited submission to a special issue of the ACM Journal of Experimental Algorithmics devoted to the best papers of ALENEX 2008, 2009. 5, 8, 9

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. 9

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. 6

[Del09] Daniel Delling. Time-Dependent SHARC-Routing. Invited submission to a the special issue of Algorithmica devoted to the best papers of ESA 2008, 2009. 41

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. 6

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. To appear. 6

[HKMS08] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2008. To appear. 7, 8

[HSWW06] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10, 2006. 7

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In WEA'05 [WEA05], pages 126–138. 7

[Lab07] Karypis Lab. METIS - Family of Multilevel Partitioning Algorithms, 2007. 7

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. volume 22, pages 219–230. IfGI prints, 2004. 7

[Mit02] Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002. 10

[MSS+05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In WEA'05 [WEA05], pages 189–202. 7

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006. 7

[PSS07] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient bloom filters. In *WEA*, pages 108–121, 2007. 11

[SS06] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006. 8

[WEA05] *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science. Springer, 2005. 42