Theory and Applications of Mathematics & Computer Science

(ISSN 2067-2764, EISSN 2247-6202)
http://www.uav.ro/applications/se/journal/index.php/tamcs

Theory and Applications of Mathematics & Computer Science 4 (1) (2014) 1–13

Practical Algorithmic Optimizations for Finding Maximal Matchings in Induced Subgraphs of Grids and Minimum Cost Perfect Matchings in Bipartite Graphs

Mugurel Ionuţ Andreica[a,*]

[a]*Computer Science Department, Politehnica University of Bucharest, Splaiul Independenţei 313, RO-060042, sector 6, Bucharest, Romania.*

Abstract

In this paper we present practical algorithmic optimizations addressing two problems. The first one is concerned with computing a maximal matching in an induced subgraph of a grid graph. For this problem we present a faster sequential algorithm using bit operations and a way of implementing it in a parallel environment. The second problem is concerned with computing minimum cost perfect matchings in bipartite graphs. For this problem we extend the idea behind the Hopcroft-Karp maximum matching algorithm and then we consider a more general situation in which multiple minimum cost perfect matchings need to be computed in the same graph, under certain cost restrictions. We present experimental results for all the proposed optimizations.

Keywords: Minimum cost perfect matching, maximal matching, maximum flow, grid graph.
2010 MSC: 05C21.

1. Introduction

The problem of computing maximum or maximal matchings in bipartite graphs has been considered many times in the scientific literature. Many of the proposed algorithms use the fact that computing a maximum matching in a bipartite graph is equivalent to computing a maximum flow in a slightly modified graph. Thus, results from the theory of network flows can be used for computing maximum matchings. If only a maximal matching is needed, then simpler greedy-type algorithms can be employed. In this paper we present several practical algorithmic improvements for some of the algorithms used for computing maximal matchings in grid graphs and minimum cost perfect matchings in bipartite graphs.

The rest of this paper is structured as follows. In Section 2 we define the main terms and techniques used in this paper. In Section 3 we discuss related work. In Section 4 we present

[*]Corresponding author
Email address: `mugurel.andreica@cs.pub.ro` (Mugurel Ionuţ Andreica)

faster algorithms for computing maximal matchings in induced subgraphs of grid graphs based on algorithms which use bit operations. In Section 5 we extend an idea used for computing maximum matchings in bipartite graphs to the computation of minimum cost perfect matchings. The idea consists of using multiple edge-disjoint augmenting paths per iteration in order to reduce the number of iterations of the algorithm. In Section 6 we consider another perfect matching problem. In this problem we are interested in computing a minimum cost perfect matching in a complete bipartite graph under certain restrictions regarding the cost computation. The cost of the matching is considered to be equal to the sum of the costs of the edges from the matching *except* for the cost of the minimum cost edge from the matching (i.e. the minimum cost edge of the matching is considered to have cost 0 when computing the cost of the matching). In Section 7 we present experimental evaluations of all the algorithms discussed in this paper. Finally, in Section 8 we conclude.

2. Terms and Definitions

A bipartite graph is a graph whose vertices can be split into two sets L (left) and R (right). We consider the vertices to be numbered from 1 to $|L|$ in the left set and from 1 to $|R|$ in the right set (it is acceptable to have vertices with the same number in the graph, because they will be differentiated based on the set L or R to which they belong). Every edge (x, y) of the graph is between a node $x \in L$ and a node $y \in R$. A matching in a bipartite graph is a set of edges such that no two edges in the set have a common vertex. A maximum matching is a matching of maximum cardinality. A maximal matching is a matching to which no more edges can be added (i.e. all the edges outside of the matching have a common vertex with at least one edge from the matching). A perfect matching is a matching in which every node of the graph is an endpoint of an edge from the matching (such a matching may exist only when $|L| = |R|$).

In order to reduce the maximum matching problem to a maximum flow problem we need to construct a directed graph as follows. We will have a special node S called the *source* and another special node T called the *sink*. We will also keep all the nodes from the given bipartite graph. Each edge (x, y) of the original bipartite graph will be replaced by a directed arc from x to y having capacity 1. We will also add capacity 1 arcs from S to every node $x \in L$ and from every node $y \in R$ to T. In case the edges of the bipartite graph have costs these costs are maintained on the directed arcs from the nodes $x \in L$ to the nodes $y \in R$ (we will denote by $c(x, y)$ the cost of the edge between $x \in L$ and $y \in R$). All the arcs having S or T as an endpoint will have cost 0.

One of the best known maximum flow algorithms is the Edmonds-Karp algorithm (Edmonds & Karp, 1972). This algorithm can be summarized as follows: As long as possible find a shortest path from S to T in the residual graph and augment the flow along that path. When arc costs are involved the algorithm can be adjusted in order to find a minimum cost path from S to T in the residual graph. Note that the residual graph may contain negative costs. This version of the Edmonds-Karp algorithm is known as the *successive shortest path* algorithm (Todinov, 2013). A simple breadth-first search algorithm is used for finding a shortest path in the first case (i.e. when edge costs are not involved), while a minimum cost path algorithm needs to be used in the second case (i.e. when edge costs are involved), for instance, Bellman-Ford-Moore (Papaefthymiou & Rodrigue, 1997) or even Dijkstra's algorithm (Todinov, 2013) after modifying the graph's arc costs in order to remove negative costs. Thus, the algorithm consists of multiple iterations, in each of which the flow is increased along a single path. The most time consuming part in each iteration is the traversal of the graph in order to find an augmenting path. In a graph with V vertices and E arcs finding the shortest augmenting path takes $O(V + E)$ time when no costs are involved and $O(V \cdot E)$ time when costs are involved (or $O(V + E \cdot log(V))$ or $O(E + V \cdot log(V))$ time when Dijkstra's algorithm is used on the modified residual graph costs). Then, augmenting the flow along the found path is easy (it takes only $O(V)$ time). In the case of bipartite graphs it is sufficient to find a path from S to an unmatched vertex in R (because this vertex is directly connected to T through an existing arc in the residual graph).

3. Related Work

The best algorithm for computing a maximum matching in sparse bipartite graphs is the Hopcroft-Karp algorithm (Hopcroft & Karp, 1973), which has a time complexity of $O(E \cdot \sqrt{V})$ where V is the number of vertices and E is the number of edges of the graph. For dense bipartite graphs the algorithm proposed in (Alt *et al.*, 1991) has a slightly better time complexity of $O(V^{1.5}\sqrt{\frac{E}{log(V)}})$. Both of these algorithms have a better time complexity than the Edmonds-Karp algorithm for finding a maximum flow presented in the previous section. However, due to its simplicity, the Edmonds-Karp algorithm is used in many practical implementations. Moreover, experimental evaluations showed that for some types of bipartite graphs some modified versions of the Edmonds-Karp algorithm (which use breadth-first search from all the source's neighbors for finding augmenting paths) are faster than the Hopcroft-Karp algorithm, despite having a worse theoretical time complexity (Cherkassky *et al.*, 1998).

Edmonds-Karp is not the only algorithm for computing maximum flows in graphs. In fact, many such algorithms were proposed in the scientific literature. Some of the most popular ones are Dinic's algorithm (Dinic, 1970), Karzanov's algorithm (Karzanov, 1974) and the push-relabel maximum flow algorithm (Goldberg & Tarjan, 1986).

A minimum cost perfect bipartite matching can be computed in $O(V^3)$ time using the Hungarian algoritm (Munkres, 1957). The *successive shortest path* algorithm for minimum cost maximum flows can be implemented in $O(V \cdot (E + V \cdot log(V)))$ time in order to compute a minimum cost maximum matching by using Fibonacci heaps (Fredman & Tarjan, 1987). The algorithm consists of $O(V)$ iterations and each iteration runs in $O(E + V \cdot log(V))$ time. Dynamic versions of the minimum cost perfect bipartite matching problem, in which edge costs can be changed, have also been considered (Mills-Tettey *et al.*, 2007).

Maximum matchings can also be computed in general graphs, not just bipartite graphs (see, for instance, Gabow's algorithm (Gabow, 1976), having an $O(V^3)$ time complexity). Minimum cost perfect matchings have also been considered in some special classes of graphs, e.g. graphs induced by points in the plane (Varadarajan, 1998). Greedy algorithms for maximal matchings, including parallel versions, were presented in (Blelloch *et al.*, 2012). The problem of maintaining maximal matchings in dynamic graphs has been addressed in (Neiman & Solomon, 2013).

4. Faster Algorithm for Maximal Matchings in Induced Subgraphs of Grid Graphs using Bit Operations

We consider an $M \cdot N$ grid graph in which every node has a coordinate (x, y) $(0 \leq x \leq N-1$, $0 \leq y \leq M-1)$ and some nodes are missing. The graph is defined by the implicit adjacency structure of the existing nodes (i.e. two nodes at distance 1 in the grid are neighbors). We are interested in computing a maximal matching in this graph. Note that a maximal matching simply implies that no other edge of the graph can be added to the matching and not that the matching has maximum cardinality. Computing a maximum cardinality matching can be done easily, because the graph is bipartite (we can separate the nodes into two groups based on the parity of their sum of x and y coordinates) and there are many polynomial-time maximum matching algorithms in such graphs (see Section 3).

Computing a maximal matching can be achieved faster, in only $O(M \cdot N)$ time. Let's consider the following Greedy algorithm (Algorithm 1) which traverses the grid graph in increasing order of the y-coordinate and for each y in inceasing order of the x-coordinate.

Algorithm 1 Greedy $O(M \cdot N)$ Algorithm for Finding a Maximal Matching in an Induced Subgraph of a Grid Graph

```
$C = 0$ {At the end of the algorithm $C$ will be the size of the maximal matching.}
for $y = 0$ to $M-1$ do
  for $x = 0$ to $N-1$ do
    if node $(x, y)$ exists in the graph then
      if $y > 0$ and node $(x, y-1)$ exists in the graph and node $(x, y-1)$ is not matched then
        Match the nodes $(x, y)$ and $(x, y-1)$
        $C = C + 1$
      else if $x > 0$ and node $(x-1, y)$ exists in the graph and node $(x-1, y)$ is not matched then
        Match the nodes $(x, y)$ and $(x-1, y)$
        $C = C + 1$
      end if
    end if
  end for
end for
```

We can implement a faster version of the Algorithm 1 by using bit operations. Note that the presented algorithm will only compute the size of the maximal matching and not the matching itself. The speed increase is due to using bit operations and handling multiple nodes at the same time. We will split each row of the grid graph (corresponding to a y-coordinate) into blocks of B bits. Block i of each row contains bits referring to the coordinates $i \cdot B, \ldots, (i+1) \cdot B - 1$. We will denote by $block(y, i)$ the block i of the row corresponding to the coordinate y. We will have bit j of $block(y, i)$ set to 1 if the node $(i \cdot B + j, y)$ exists in the graph, and set to 0 otherwise $(0 \leq j \leq B-1)$. We will traverse the graph from $y = 0$ to $y = M-1$ as in Algorithm 1. During the traversal we will maintain a row of blocks corresponding to the previous row in which 1 bits will correspond to existing unmatched nodes. When considering a new row y, the first step is to perform an **AND** between the current row and the previous row of unmatched nodes. All the 1 bits in the result of this operation will represent nodes from the current row which are matched to nodes from the previous row. After performing this match we will clear the matched 1 bits from the current row. The next step is to match nodes from the current row which are adjacent to each other faster than $O(N)$ time. In order to achieve this we will need to use a preprocessing step. For each sequence S of B bits we will compute $MCnt(S)$ that is the number of pairs of adjacent bits matched in S and $MRes(S)$ the B-bit sequence containing the remaining unmatched 1 bits of S. We will start with $MCnt(S) = 0$ and $MRes(S) = S$. Then we will traverse all the bits j of S from 1 to $B-1$. If $MRes(S)(j) = 1$ and $Mres(S)(j-1) = 1$ then we increase $MCnt(S)$ by 1 and we clear the bits j and $j-1$ in $MRes(S)$. Thus, we can compute $MCnt(S)$ and $MRes(S)$ in $O(B)$ time, obtaining a preprocessing time of $O(2^B \cdot B)$. Within the same time complexity we will also compute for each B-bit sequence S the number of 1 bits in S, $BCnt(S)$.

With these values we can perfom the matching on the current row y. We will consider each block i from 0 to $(N-1)/B$ and we will maintain the state of the current row as a sequence of blocks $crow$. First we copy $block(y, i)$ to $crow(i)$. Then, if $i > 0$ and bit $B-1$ of $crow(i-1)$ is 1 and bit 0 of $crow(i)$ is 1 we match these two bits and then we set them to zero. Afterwards we replace $crow(i)$ by $MRes(crow(i))$. The detailed algorithm is presented in Algorithm 2.

The time complexity of Algorithm 2 is $O(2^B \cdot B + M \cdot N/B)$. The $O(2^B \cdot B)$ term is the time complexity of the preprocessing stage and the $O(M \cdot N/B)$ is the time complexity of the actual algorithm. The presented algorithm can even be implemented in a parallel manner. First of all the preprocessing stage is obviously parallelizable: each of the 2^B values of the tables $MRes$, $MCnt$ and $BCnt$ ca be computed independently. In order to parallelize the actual algorithm we will need to refactor it first. We will first perform all the horizontal matchings on each of the M rows. We can first perform the matching within each block of each row independently in parallel and store the result in a variable specific to each $(row, block)$ pair (this means that we would have such a variable for each block of each row). Then we can handle the matching between bit 0 of odd-numbered blocks and bit $B-1$ of the preceding even-numbered block in parallel, followed by another stage in which we handle the matching between bit 0 of even-numbered blocks and bit $B-1$ of the preceding odd-numbered block in parallel. Then we can handle matchings between nodes on different rows. In order to parallelize this stage we will first consider all the rows corresponding to odd y coordinates being matched to the adjacent row with a smaller and even y coordinate. Obviously, each block of all of these $M/2$ (we consider integer division) rows can be handled independently in parallel. Then we will consider all the rows corresponding to even y coordinates being matched to the adjacent row with a smaller and odd y coordinate. Each block of these $M - M/2$ rows can also be handled independently, in parallel. The parallel algorithm presented here can use up to 2^B processors in the preprocessing stage and up to $M \cdot N/B$ processors in the maximal matching computation stage. Note that the result of the parallel version may differ from the result of the sequential algorithm (Algorithm 2) because a different maximal matching will be computed (due to the different order of performing the vertical and horizontal matches).

Algorithm 2 Greedy $O(2^B \cdot B + M \cdot N/B)$ Algorithm for Finding a Maximal Matching in an Induced Subgraph of a Grid Graph

```
Compute the tables $MRes$, $MCnt$ and $BCnt$.
$C = 0$
$prow(i) = 0$ $(0 \leq i \leq (N-1)/B)$
for $y = 0$ to $M-1$ do
    $crow(i) \leftarrow block(y, i)$ $(0 \leq i \leq (N-1)/B)$
    for $i = 0$ to $(N-1)/B$ do
        $vmatch(i) = crow(i)$ AND $prow(i)$
        $C = C + BCnt(vmatch(i))$
        $crow(i) = crow(i)$ XOR $vmatch(i)$
    end for
    $C = C + MCnt(crow(0))$
    $crow(0) = MRes(crow(0))$
    for $i = 1$ to $(N-1)/B$ do
        if bit $B-1$ of $crow(i-1)$ is 1 and bit 0 of $crow(i)$ is 1 then
            $C = C + 1$
            Clear bit $B-1$ of $crow(i-1)$ and bit 0 of $crow(i)$.
        end if
        $C = C + MCnt(crow(i))$
        $crow(i) = MRes(crow(i))$
    end for
    $prow(i) = crow(i)$ $(0 \leq i \leq (N-1)/B)$
end for
```

5. Using a Maximal Set of Edge-Disjoint Paths for Reducing the Number of Iterations of Minimum Cost Perfect Matching Algorithms

The best algorithm for computing a maximum matching in a sparse bipartite graph is the Hopcroft-Karp algorithm (Hopcroft & Karp, 1973) which has a time complexity of $O(E \cdot \sqrt{V})$. The main idea behind that algorithm is to enhance a standard augmenting path algorithm as follows. After each BFS traversal of the graph in order to find an augmenting path, the matching will not be increased only along one path, but rather along a maximal set of edge-disjoint shortest paths (note that in this case edge-disjoint paths are also vertex-disjoint paths, because they are paths in a shortest path tree; the only common vertex is the source S).

The same idea can be used when computing a minimum cost perfect matching. At each iteration of the *successive shortest path* algorithm (Todinov, 2013) we need to find a minimum cost path in the residual graph. Note that the residual graph may have arcs with negative costs, but does not have negative cycles. Thus, we either need to use a shortest path algorithm which supports negative costs (e.g. Bellman-Ford-Moore (Papaefthymiou & Rodrigue, 1997)) or we need to modify the costs in order to obtain non-negative costs only and, thus, use Dijkstra's algorithm (Todinov, 2013).

No matter what shortest path algorithm we use, at the end of the algorithm we have the minimum cost of a path starting at the source and ending at each node $x \in R$. We can sort all these nodes in ascending order of the cost of the minimum cost path to reach them (ignoring the unreachable nodes, if any). Then, rather than only increasing the matching along the minimum cost augmenting path, we can consider these nodes in sorted order. For each node x we trace back its shortest path to the source. If the matching was already augmented at the current iteration along at least one edge of the path, then we ignore node x and we move on to the next one. If the current path does not intersect with any of the paths along which the matching was augmented at the current iteration then we can augment the matching along this path and mark its edges in order to know that no other shortest path containing (some of) these edges can be used for augmenting the matching at the current iteration.

A direct implementation of this modified matching augmentation algorithm takes $O(V^2)$ time per iteration, because there may be $O(V)$ verified paths and each verification may take $O(V)$ time. On the other hand, we cannot guarantee that the matching will be augmented along more than one path. A scenario in which all the paths have the first edge in common (from the source to a vertex $x \in L$) and the minimum cost path has only this edge in common with the other paths is quite possible. Since $O(V^2)$ may be a higher time complexity than that of computing the minimum cost paths, we may end up increasing the running time of the algorithm instead of decreasing it. Thus, we need to reduce the time complexity of the matching augmentation part. This can be achieved as follows. Let's remember that the minimum cost paths are paths in a shortest path tree (where the length of a path is its cost). We will consider the paths in the same order as before and we will consider all the edges to be initially unmarked. If the last edge of the current path is not marked then we will be able to augment the matching along the current path. After augmenting the matching along the current path, let $x \in L$ be the first vertex on the path (after the source). We will traverse the whole subtree of the shortest path tree rooted at x and we will mark all the edges of this subtree. Augmenting the matching along all the possible paths takes at most $O(V)$ time overall (because the paths are edge-disjoint). Marking the edges of the shortest path tree also takes at most $O(V)$ time overall, because there are $O(V)$ edges in the shortest path tree and each edge is marked at most once. Thus, the matching augmentation algorithm takes only $O(V)$ time plus the time needed for sorting the paths in increasing order of their costs (e.g. $O(V \cdot log(V))$ time).

The first version of the algorithm proposed in this section is described in pseudocode in Algorithm 3. The input to the algorithm consists of two maps: *dist*, containing the cost of the shortest path from S to every vertex $x \in R$ (we consider $dist(x) = +\infty$ if the vertex x is not reachable from S), and *parent*, containing the *parent* in the shortest path tree for each vertex of the graph. Left-set vertices x are denoted as (x, L) in the algorithm and right-set vertices y are denoted as (y, R). In order to maintain the pseudocode simpler, we will mark the graph vertices instead of the edges (because, as mentioned earlier, in this case the edge-disjoint paths are also vertex-disjoint). The second version of the algorithm is described in pseudocode in Algorithm 4. The input to the Algorithm 4 also consists of the same two maps *dist* and *parent*, together with an extra map, *children*, which contains the children in the shortest path tree of each vertex of the graph.

This algorithm basically augments the matching along a maximal set of edge-disjoint paths (in fact, because they are paths of a shortest path tree, the paths are also vertex-disjoint except for the source vertex). It is important. however, to consider these paths in increasing order of their costs,

Algorithm 3 Increasing the Matching Along a Maximal Set of Edge-Disjoint Paths - The $O(V^2)$ Algorithm

```
Input: dist, parent.
  Set all the vertices of the graph as unmarked.
  Sort the vertices $x \in R$ in increasing order of $dist(x)$.
  for $x \in R$ in increasing order of $dist(x)$ such that $dist(x) < +\infty$ do
    $y = (x, R)$, $OK = true$
    while $y \neq S$ and $OK = true$ do
      if vertex $y$ is marked then
        $OK = false$
      else
        $y = parent(y)$
      end if
    end while
    if $OK = true$ then
      Increase the matching along the shortest path from $S$ to $(x, R)$ (the reverse of the path can
      be found by following the parent pointers starting from $(x, R)$).
      $y = (x, R)$
      while $y \neq S$ do
        Mark vertex $y$ as marked.
        $y = parent(y)$
      end while
    end if
  end for
```

Algorithm 4 Increasing the Matching Along a Maximal Set of Edge-Disjoint Paths - The $O(V)$ Algorithm

```
Input: dist, parent, children.
  Set all the vertices of the graph as unmarked.
  Sort the vertices $x \in R$ in increasing order of $dist(x)$.
  for $x \in R$ in increasing order of $dist(x)$ such that $dist(x) < +\infty$ do
    if $(x, R)$ is not marked then
      Increase the matching along the shortest path from $S$ to $(x, R)$ (the reverse of the path can
      be found by following the parent pointers starting from $(x, R)$).
      Let $(y, L)$ be the first node on the path from $S$ to $(x, R)$ after $S$. Recursively mark all the
      vertices located in vertex $(y, L)$'s subtree of the shortest path tree. The children map will
      be used for retrieving for each vertex $v$ the set $children(v)$ that is the set of the shortest path
      tree children of the vertex $v$.
    end if
  end for
```

in order to make sure that the residual graph at the next iteration does not contain negative cycles. The reason for which this optimization works is as follows. In a perfect mtaching every vertex has to be matched. When augmenting the matching along a shortest path to a node x, even if x is not the right-side node with the minimum cost path, the point is that any future minimum cost path to node x (in any future residual graph) will not have a lower cost than the current shortest path to x. So there is no reason for us not to augment the matching along that path, as long as this does not block paths with lower costs along which the matching could have been augmented.

Note that this optimization is not correct in a maximum matching algorithm. It is not correct to augment the matching along a path which does not have the globally minimum cost, because we are not sure if the right side vertex x needs to be in the optimal matching or not. And since vertices added to the matching are never removed by this algorithm, it is possible to make a mistake in this case.

This optimization does not change the theoretical time complexity of the minimum cost perfect matching algorithm, because we cannot provide any extra guarantees regarding the number of augmenting paths per iteration (and, thus, we cannot provide guarantees regarding the reduction in the number of iterations).

6. Minimum Cost Perfect Matching With the Minimum Cost Edge Ignored

In this section we consider the following problem. Given a complete bipartite graph with n nodes on the left side and n nodes on the right side and costs on its edges, we want to find a minimum cost perfect matching in which the cost is defined as the sum of the costs of all the edges in the matching except for the cost of the edge with the smallest cost.

A simple method for solving this problem is to iterate over all the edges (i, j) and fix them as the smallest edge in the matching. Then we would compute a (usual) minimum cost maximum matching in the bipartite graph from which left node i, right node j and all the edges (i', j') with $c(i', j') < c(i, j)$ (or $c(i', j') = c(i, j)$ and the edge (i', j') was considered before the edge (i, j)) are removed. If the maximum matching M has size $n-1$ then we found a potential solution, as follows. The potential solution consists of the $n - 1$ edges of the found matching plus the edge (i, j). The cost of the matching (according to the definition used in this section) is equal to the sum of the costs of the $n - 1$ edges of M. Note that the fixed edge (i, j) is the edge with the minimum cost in the perfect matching (the one whose cost is not considered towards the cost of the matching). Once the edge (i, j) was fixed we needed to minimize the total cost of the other $n - 1$ edges of the perfect matching. Moreover, the other edges of the perfect matching needed to have costs which were larger than or equal to $c(i, j)$. The minimum cost maximum matching M contains the $n - 1$ edges we were looking for, in case its cardinality is $n - 1$. If its cardinality is less than $n - 1$ then we can conclude that there is no perfect matching containing the edge (i, j) as the minimum cost edge.

This solution requires the computation of $O(n^2)$ independent minimum cost maximum matchings. The key to obtain a better solution is to notice that the $O(n^2)$ matchings that we need to compute are not totally independent. We will sort all the n^2 edges first in ascending cost order and then we will consider them in this order. For the first edge we will compute the minimum cost maximum matching from scratch. Let's assume that we reached the edge (i, j). This time we will not compute the new matching from scratch. Instead, let's consider the matching M obtained for the previous edge in the sorted order. We will remove from M any edge with an endpoint at the left node i or the right node j (if any). Then we will remove from M all the edges with a cost smaller than $c(i, j)$ (or equal to $c(i, j)$ but for which the corresponding edge was considered before the current edge (i, j) in the sorted order). All the other edges of M will be maintained. We will start the (usual) minimum cost maximum matching for the edge (i, j) with all the remaining edges from M as part of the matching. Note that the algorithm may replace some of these edges by other edges. This can happen if the reverse of an edge (x, y) ($x \in L$ and $y \in R$) from M is, at some point, part of the shortest path from S to T in the (new) residual graph. When considering the maximum matching problem as a maximum flow problem, the fact that an edge (x, y) is removed from the matching means that the flow is pushed back along that edge (in order to be redirected somewhere else).

By applying the optimizations from the previous paragraph we expect that the number of iterations required for computing each new minimum cost maximum matching will be significantly reduced.

This problem can also be viewed as a dynamic minimum cost perfect matching problem, in which the edge costs can be modified (for instance, instead of removing edges from the graph we can consider that their cost increased to $+\infty$).

7. Experimental Results

We implemented the three optimizations presented in this paper and compared them against their unoptimized versions. All the tests were run on a machine running Windows 7 with an Intel Atom N450 1.66 GHz CPU and 1 GB RAM. All the algorithms were implemented in the C++ programming language and the code was compiled using the G++ compiler version 3.3.1.

First we tested our new algorithm for computing a maximal matching in an induced subgraph of a grid graph. We chose $M = N = 2048$ and we randomly generated the induced subgraph - each point (x, y) ($0 \le x \le N - 1$, $0 \le y \le M - 1$) had an equal probability of being part of the subgraph or not. Thus, each of the tested subgraphs had approximately 50% of the nodes of the full grid graph. We generated 100 subgraphs and ran Algorithm 1 (the unoptimized version) and Algorithm 2 (the optimized version) on each of them. We computed the total running time for all the graphs. Algorithm 1 took 5.3 seconds. For Algorithm 2 we considered two values for B: $B = 16$ and $B = 8$. For $B = 16$ the running time was 2.74 sec and for $B = 8$ it was 1.13 sec. Note that in this case we computed the tables $MCnt$, $Mres$ and $BCnt$ each time (i.e. for each of the 100 tests). However, when running the algorithm on multiple tests with the same value of B, these tables only need to be computed once, in the beginning. Thus, we changed the algorithm in order to compute these tables only once in the very beginning and not for each of the 100 tests. The new running times were 0.92 sec for $B = 16$ and 1.13 sec for $B = 8$. Note that the running time is unchanged for $B = 8$ because the sizes of the tables are small and the time needed to compute them is negligible compared to the time needed to compute the matching. However, for $B = 16$, when the sizes of the tables increase significantly, it is much better to compute the tables in the beginning and reuse them for each test.

Then we repeated the tests for induced subgraphs of grid graphs containing 75% and 100% of the nodes of a full grid graph. For graphs with 75% of the nodes of a full grid graph the running times of our optimized algorithm were: 1.21 sec for $B = 8$ and 0.93 sec for $B = 16$ (note that we only considered the case when the tables are computed just once). The running time of the unoptimized version was 4.69 sec. When considering the full grid graph we obtained the following running times: 0.90 sec for $B = 8$ and 0.55 sec for $B = 16$ for the optimized version and 3.40 sec for the unoptimized version.

We did not test other values of B because the implementation would become less feasible. For $B > 16$ the sizes of the precomputed tables would become too large. For $B = 8$ and $B = 16$ we were able to make use of existing C/C++ data types (*unsigned char* and *unsigned short int*) in order to store a block. For $B \neq 8$ and $B \neq 16$ (and $B \leq 16$) we cannot exactly fit a block into an existing C/C++ data type.

Second we tested the improvement brought by the use of multiple edge-disjoint paths for augmenting the matching in a minimum cost bipartite perfect matching algorithm. The expected improvement consisted in a reduction of the number of iterations. The standard algorithm would use n iterations where $|L| = |R| = n$. We chose $n = 256$ and we generated 100 complete bipartite graphs. The cost of each edge was chosen to be a random integer between 1 and 10000 (inclusive). The unoptimized algorithm we used was the standard *successive shortest path* algorithm with the Bellman-Ford-Moore algorithm for computing minimum cost paths at each iteration. The optimized algorithm simply included the $O(n^2)$ matching augmentation along multiple edge-disjoint minimum paths described in Section 5. We measured both the total running time and the total number of iterations. The total running time (for all the 100 graphs) and the total number of iterations of the standard algorithm were: 23.17 sec and 25600 iterations. With our optimization the total running time was 2.3 sec and the total number of iterations was 1022. We notice that, even with the most basic implementation of our optimization, the running time was reduced 10 times and the number of iterations was reduced 25 times. Although the running time improvements may not translate directly when other minimum cost path computation algorithms are used (e.g. Dijkstra's algorithm) or when the $O(V + V \cdot log(V))$ matching augmentation optimization is used (instead of the $O(V^2)$ version), the improvement in the number of iterations does not depend on these algorithms and, thus, it is applicable to any implementation of the *successive shortest path* minimum cost bipartite perfect matching algorithm.

Then we considered the same testing scenario, except that the edge costs were chosen as random integers between 1 and 2 (inclusive). The total running time of our optimized algorithm was 2.98 sec and the total number of iterations was 5100. The number of iterations of the standard algorithm remained the same (as expected), but its running time dropped to 13.83 sec.

We also considered the complete bipartite graph with the following costs $c(x, y) = min(x, y)$ $(1 \leq x, y \leq n)$ and $n = 256$. In this case our optimization did not reduce the number of iterations at all (due to the special structure of the bipartite graph it was never able to augment the matching along more than one path per iteration). However, the running time with our optimization enabled was almost identical to the unoptimized version. We conclude that our optimization has a great potential for reducing the number of iterations of the *successive shortest path* minimum cost perfect matching algorithm and even in the pathological cases when it cannot reduce the number of iterations, it doesn't cause any significant overhead.

Although there is a large difference between the number of iterations obtained by our optimization in the cases of random complete bipartite graphs and in the case of the specific bipartite graph from the previous paragraph, we did not consider other types of bipartite graphs for testing. Understanding the correlation between the performance of our optimization and the specific properties of the costs of the edges of the bipartite graph is an interesting topic, but we defer its study to a later date, because we feel that this topic is more appropriate for a separate, more experimentally focused, paper.

For the problem presented in Section 6 we tested our optimization of not recomputing the perfect matching from scratch each time. The first minimum cost perfect matching algorithm that we used was the one which contained our matching augmentation optimization presented in Section 5 and tested earlier. We generated 10 bipartite graphs with $n = 128$ and edge costs randomly selected between 1 and 10000 (inclusive). We computed the total execution time and the total number of iterations of the *successive shortest path* algorithm. When the matching was computed from scratch each time ($O(n^2)$ times) the total running time was 805 sec and the total number of iterations was 1305200. When we applied our optimization from Section 6, the total running time was 221 sec and the total number of iterations was 244886. When using the standard *successive shortest path* algorithm in order to compute a minimum cost perfect matching (and not using our optimization of not recomputing the matching from scratch each time), the total running time was 5798 sec and the total number of iterations was 20610157. When we applied our optimization from Section 6 and also used the standard minimum cost perfect matching algorithm the total running time was 232 sec and the total number of iterations was 251789. We can see that our optimization of not recomputing each minimum cost perfect matching from scratch is very effective. When combined with the optimization presented in Section 5, of augmenting the matching along multiple paths at each iteration, we obtained the best results. However, even when just the standard *successive shortest path* minimum cost perfect matching algorithm is used in conjunction with our optimization from Section 6 the improvements over the naive unoptimized version are significant. Nevertheless, more tests may need to be performed in the future in order to understand sufficiently well how good our proposed optimization really is.

8. Conclusions

In this paper we presented three practical algorithmic optimizations addressing problems like computing maximal matchings in induced subgraphs of grid graphs or computing minimum cost perfect matchings in bipartite graphs (under certain restrictions). The proposed optimizations were evaluated experimentally and compared against the unoptimized algorithms. The execution time was significantly reduced in each case, thus proving the validity and effectiveness of our optimizations.

References

Alt, H., N. Blum, K. Mehlhorn and M. Paul (1991). Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{\frac{m}{logn}})$. *Information Processing Letters* **37**(4), 237–240.

Blelloch, G.E., J.T. Fineman and J. Shun (2012). Greedy sequential maximal independent set and matching are parallel on average. In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*. pp. 308–317.

Cherkassky, B.V., A.V. Goldberg, P. Martin, J.C. Setubal and J. Stolfi (1998). Augment or push? a computational study of bipartite matching and unit capacity maximum flow algorithms. *ACM Journal of Experimental Algorithmics*.

Dinic, E.A. (1970). Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady* **11**, 1277–1280.

Edmonds, J. and R.M. Karp (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM* **19**(2), 248–264.

Fredman, M.L. and R.E. Tarjan (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* **34**, 596–615.

Gabow, H.N. (1976). An efficient implementation of edmonds' algorithm for maximum matching on graphs. *Journal of the ACM* **23**(2), 221–234.

Goldberg, A. and R.E. Tarjan (1986). A new approach to the maximum flow problem. In: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*. pp. 136–146.

Hopcroft, J.E. and R.M. Karp (1973). An n5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* **2**(4), 225–231.

Karzanov, A.V. (1974). Determining the maximal flow in a network by the method of preflows. *Soviet Math. Doklady* **15**, 434–437.

Mills-Tettey, G.A., A. Stentz and M.B. Dias (2007). The dynamic hungarian algorithm for the assignment problem with changing costs. Technical report. Robotics Institute.

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics* **5**(1), 32–38.

Neiman, O. and S. Solomon (2013). Simple deterministic algorithms for fully dynamic maximal matching. In: *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*. pp. 745–754.

Papaefthymiou, M. and J. Rodrigue (1997). Implementing parallel shortest-paths algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **30**, 59–68.

Todinov, M.T. (2013). *Flow Networks: Analysis and Optimization of Repairable Flow Networks, Networks with Disturbed Flows, Static Flow Networks and Reliability Networks*. Elsevier.

Varadarajan, K.R. (1998). A divide-and-conquer algorithm for min-cost perfect matching in the plane. In: *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*. pp. 320–329.