Universitt Dortmund
Wirtschafts- und Sozialwissenschaftliche Fakultt

Diskussionsbeitrge des Fachgebietes
Operations Research und Wirtschaftsinformatik
No. 32 (2006)

On computing maximum-size matchings in graphs

Andreas Schoppmeyer

Universität Dortmund, Operations Research und Wirtschaftsinformatik, Vogelpothsweg 87, D 44 221 Dortmund, Germany, e-mail: andreas.schoppmeyer@uni-dortmund.de

On computing maximum-size matchings in graphs

by Andreas Schoppmeyer

Abstract: The problem of finding a maximum-size matching in a graph appears in many situations in graph theory. Therefore it is crucial to compute such matchings in a fast way. In some cases a hybrid algorithm, consisting of an heuristic to find a start-matching and an exact algorithm to compute the maximum-size matching, appears to be much faster than classical algorithms. We show a way to implement appropriate heuristics, exact algorithms and hybrid algorithms in C++ and compare their performance on different random graphs. To reduce the programming-effort , we used comprehensible techniques. These techniques can be implemented independent of programming languages and the operating systems.

Keywords: Maximum-size matchings, Efficient graph algorithms

1 Introduction

The subject of this paper is the problem of finding maximum-size matchings in graphs. All graphs $G = (V, E)$ are considered to be connected, finite and undirected. A matching M is a subset of E with the property, that no vertex $v \in V$ is incident with more than one edge of M. A vertex is called free, if it is not incident to an edge of M. The set of all vertices adjacent to a vertex v is named neighborhood of v. An alternating path with respect to M is a path, consisting alternating of edges $e \in M$ and edges $e \in E \setminus M$. If both end-vertices of an alternating path are free vertices, the path is called augmenting path. The augmenting path theorem[1] says, that a matching is of maximum-size if no augmenting path can be found in G.

We will give an new approach, based on ideas of Micali and Vazirani[3], but better to implement. Also the possibility of using simple hybrid algorithms to increase the performance is discussed.

2 Heuristics

In order to construct the hybrid algorithms, we need some heuristics, to compute a first "good" matching (not necessary of maximum size). We use one basic frame for all these heuristics. Each heuristic selects a vertex v^* and matches it with a selected vertex u^* from the neighborhood of v^*. After that, the remaining graph is updated. The heuristics differ only in the selection of vertices v^* and u^* in step 1 and step 2 and in the update-procedure for the remaining graph, used in step 2 and step 3 in the greedy-heuristic basic frame. These differences are written in **bold letters** in the following pseudocode:

```
Greedy-heuristic basic frame
Given: graph $G$
Step 0 remaining graph $\widetilde{G} = (\widetilde{V}, \widetilde{E}) := G$, matching $M := \emptyset$
Step 1 Select a vertex $v^* \in \widetilde{V}$
       Go to step 2
Step 2 If the neighborhood of $v^*$ is not empty:
          Select a vertex $u^*$ from the neighborhood of $v^*$
          Go to step 3
       Else remove $\{v^*\}$ from $\widetilde{G}$ and update the remaining graph $\widetilde{G}$
       Go to step 1
Step 3 Set $M = M \cup (v^*, u^*)$
       Remove $\{v^*, u^*\}$ from $\widetilde{G}$ and update the remaining graph $\widetilde{G}$
       If $\widetilde{V} = \emptyset$, then STOP:
          $M$ is a matching
       Else go to step 1
```

The first used heuristic, called Random-Heuristic, is selecting the vertices v^* and u^* randomly. We do not need an update-procedure. The second heuristic is picking the first vertex v^* of minimum degree above all free vertices.

The corresponding vertex u^* is selected from the neighborhood and has also minimum degree above all neighbors. After that, the degrees of all vertices, adjacent to the selected vertices v^* and u^* respectively are updated. This approach we call MinDegree-Heuristic in the following sections. This greedy-heuristic finds better starting matchings than the random heuristic. But we will also find out, that its performance suffers dramatically on dense graphs, because of the update-procedure in steps 2 and 3.

3 Algorithms

Beside the heuristics we need some exact algorithms to construct a hybrid algorithms. In this paper, we used Hopcroft and Karp's algorithm[2] for bipartit graphs, that runs in $O(|V|^{\frac{5}{2}})$, $|V|$ is the number of vertices. The idea behind this approach is to augment the matching along a maximum-cardinality set of minimum-length augmenting pathes.

We also want to process non-bipartit graphs. The fastest known algorithm is given in [3] by Micali and Vazirani. They used Hopcroft and Karp's bound for the number of necessary augmentation steps[2]. The procedure they present turned out, to be too complex for efficient implementation. Because of that, a new algorithm is given.

3.1 The new algorithm

We present an new simplified algorithm, based on the ideas of Micali and Vazirani. The main difference to our approach is that they used a special procedure to treat odd length cycles. By storing information about the predecessors of each vertex in a alternating path, we can avoid the use of such procedures. The second advantage of storing the predecessor information is that we can use a simple breadth-first search in the algorithm. To explain this approach, we need some definitions.

Definition 1. *Let $G = (V, E)$ be a graph, M a matching in G and $v_0 \in V$ a free vertex. A tree $T = (V(T), E(T))$ of G is called* ***rooted alternating tree*** *with* ***root*** *v_0, if*

- *$v_0 \in V(T)$ and*
- *each vertex in $V(T)$ is connected to the root v_0 on an alternating path.*

Definition 2. *Let $G = (V, E)$ be a graph, M a matching in G, $v_0 \in V$ a free vertex and $T = (V(T), E(T))$ a rooted alternating tree with root v_0. The vertices in $V(T)$ are called* ***inner*** *vertices and* ***outer*** *vertices, so that*

- *the root v_0 is* ***outer*** *vertex,*
- *each* ***inner*** *vertex is adjacent in T with an* ***outer*** *vertex and*
- *each* ***inner*** *vertex is incident with two edges from $E(T)$.*

In this approach we start with the construction of rooted alternating trees in each free vertex. These trees are expanded with a breath-first search. The procedure inserts all inner or outer vertices respectively, that can be reached in the existing alternating tree plus one edge.

In an odd-length cycle all vertices can be reached on two different alternating pathes from the root. On one path, each vertex is outer vertex in the tree, and on the other path, each vertex is inner vertex. So the vertices in an odd-length cycle are considered twice on different search-levels. If an augmenting path is found, all vertices of the current search-level are examined. After the augmentation the breath-first search starts with empty trees again.

A special feature of the presented algorithm is the storage of the predecessors. All stored predecessors are outer vertices, because the inner vertices can be identified by the current matching.

Algorithm 1

```
Given:  graph $G = (V, E)$, matching $M$
Step 1  For all vertices $v \in V$ do
           set $innerlevel(v) = outerlevel(v) = \infty$
           set $innerpred(v) = outerpred(v) = -1$
           set $tree(v) = -1$
           set $done(v) = 0$
        set $i = -1$, $aug = 0$
        set $outerlevel(v) = 0$ and $tree(v) = v \, \forall \, v \in V \setminus V(M)$
        Go to step 2
Step 2  $i = i + 1$
        if $\{v \mid outerlevel(v) = i \vee innerlevel(v) = i)\} = \emptyset$ then STOP
        if $i$ is an even number then go to step 3, else go to step 4
Step 3  for all $v$ with $outerlevel(v) = i$ and $done(v) = 0$ do
          for all $(u, v) \in E$ with $innerlevel(u) = i + 1, tree(u) =$
          $tree(v), innerpred(v) = u$ do
              set $innerlevel(u) = 0$
          for all $(u, v) \in E$ with $innerlevel(u) > i + 1, tree(u) \in$
          $\{tree(v), \infty\}$ do
              set $innerlevel(u) = i + 1$
              set $innerpred(u) = v$
              set $tree(u) = tree(v)$
          if $\{w \mid (w, v) \in E, tree(w) \neq tree(v), outerlevel(w) < \infty\} \neq \emptyset$
          then
              Select a vertex $w$ from this set
              identify the augmenting path $P$ from $tree(w)$ to $tree(v)$
              augment the matching $M$ with respect to $P$.
              set $done(u) = 1 \, \forall \, u$ with $tree(u) = tree(v) \vee$
              $tree(u) = tree(w)$
              set $aug = aug + 1$.
        if $aug = 0$ then go to step 2, else go to step 1
```

```
Step 4  for all $v$ with $innerlevel(v) = i$ and $done(v) = 0$ do
            for all $(u, v) \in M$ with $outerlevel(u) > i + 1, tree(u) \in \{tree(v), \infty\}$ do
                set $outerlevel(u) = i + 1$
                set $outerpred(u) = innerpred(v)$
                set $tree(u) = tree(v)$
            if $\{w \mid (w, v) \in M, tree(w) \neq tree(v), done(w) = 0\} \neq \emptyset$ then
                Select a vertex $w$ from this set
                identify the augmenting path $P$ from $tree(w)$ to $tree(v)$
                augment the matching $M$ with respect to $P$.
                set $done(u) = 1 \,\forall\, u$ with $tree(u) = tree(v) \vee$
                $tree(u) = tree(w)$
                set $aug = aug + 1$.
        if $aug = 0$ then go to step 2, else go to step 1
```

Theorem 1. *Let $G = (V, E)$ be a graph and M be a matching not of maximum-size in G. Then* Algorithm 1 *finds a maximum-cardinality set of augmenting pathes of minimum-length among all possible augmenting pathes.*

Proof. An augmenting path musst exist, because of the augmenting path theorem[1]. We can divide all augmenting pathes in two alternating pathes of the same length and a linking edge between those pathes, because every augmenting path has an odd number of edges. The construction of the trees is started in each free vertex (Step 1), so there can only be an augmenting path, consisting of two vertex-disjoint alternating pathes and a linking edge between these pathes. Because of G being connected, there musst be a linking edge, between each pair of alternating pathes. Every vertex contained in one of the pathes (*outer-/innerlevel* $< \infty$) can be reached from a free vertex (the root) on one or two alternating pathes. The only possibility that a vertex can be reached on two different pathes, is the existence of an odd-length cycle. In this case, the vertex is duplicated (*outer*- and *innerlevel* $< \infty$), it is outer and inner vertex on two different search-levels (*outerlevel* $\neq$ *innerlevel*) and is part of two different alternating pathes.

While the construction is in progress, all possible linking edges are examined. There are two cases, when a linking edge is found, that implicate an augmenting path. The first case appears, when two outer vertices from different trees are adjacent (if-clause in Step 3). The second case of a possible augmentation is, when two inner vertices are joined by a matching edge (if-clause in Step 4), because these vertices are reached on a free edge in the tree. Linking edges between an inner vertex and an outer one are of no interest, because the inner vertex must join the outer vertex with a free edge and it is reached with a free edge in the tree.

Because of all linking edges being examined, each augmenting path is found and because of the iterative increasing search level (Step 2) both vertices of a linking edge have the same level. A linking edge between two vertices of a different level is not possible. If the level difference is odd, then an inner vertex is joined to an outer one and no augmenting path can be found. If the level difference is even and two or more, the vertex of higher level must have been integrated to a tree on a lower search-level. Because of this, all linking edges are found on the lowest possible search-level.

When the first augmenting path is found, all vertices on the current search-level are checked. All other augmenting pathes with the same length are found in this case and all pathes are used for augmentation, so the number of augmenting pathes being found is maximum. After that the search is stopped and restarted. □

With theorem 1, the results of Hopcroft and Karp can be applied and the number of restarts for the search is at most $\sqrt{|V|}$. In the worst case, the search examines all edges in the graph, so at most $|E|$ steps are needed for one search, with $|E|$ being the number of edges in the graph. This results in running time of $O(\sqrt{|V|} \cdot |E|)$.

4 Hybrid-algorithms

Beside the new approach, we want to increase the capability to solve large matching problems using hybrid algorithm. These hybrid algorithms are constructed by combining heuristics with exact algorithms. For our test we used both heuristics from section 2 and combined them with both algorithms, that of Hopcroft and Karp and our new one. These algorithms can be started with a first matching, that is improved, while the algorithm is running. When these algorithms are standing alone, we start with an empty matching, but in hybrid algorithms, a heuristic is used for finding the start matching. This is simply realized, by running the heuristic and handing the found matching to the exact algorithm to start with it. In the following section, the hybrid algorithms are labeled with the name of the heuristic "+" the name of the exact algorithm.

5 Computational results

In this section, we test the hybrid algorithms in competition to the exact algorithm standing alone. All test's have taken place on random graphs. The heuristics and algorithms are implemented in C++, with respect to efficient programming.

As we see in table 1, the MinDegree-Heuristic is not suitable for the use in hybrid algorithms. But using the Random-Heuristic in a hybrid algorithm improves the performance, its average time for computing the matching is less than half of the average time, Algorithm 1 needed. In table 2 we see results for a test on bipartit graphs. The algorithm of Hopcroft & Karp can be improved with both heuristics, but the Random-Heuristic performs better. The direct

method	average time	min time	max time
Algorithm 1	63,45	47	79
Random-Heuristic + Algorithm 1	26,86	15	32
MinDegree-Heuristic + Algorithm 1	279,25	250	282

Table 1. Results for a test with 1000 random graphs with $|V| = 8000$

method	average time	min time	max time
Hopcroft & Karp	7523,21	7391	7641
Random-Heuristic + Hopcroft & Karp	82,74	62	94
MinDegree-Heuristic + Hopcroft & Karp	271,57	250	282
Algorithm 1	125,83	109	141
Random-Heuristic + Algorithm 1	99,70	93	110
MinDegree-Heuristic + Algorithm 1	289,66	281	297

Table 2. Results for a test with 1000 random bipartit graphs with $|V| = 9000$

comparison between both algorithms shows, that Algorithm 1 is faster than Hopcroft & Karp's algorithm, but a hybrid algorithm with Hopcroft & Karp's algorithm and the Random-Heuristic is the fastest method in the test.

References

1. Berge, C. (1957) Two theorems in graph theory, Proc. Natl. Acad. Sci. (U.S.A.), 43:842–844
2. Hopcroft, J. E., Karp R. M. (1973) An $n^{\frac{5}{2}}$ Algorithm for Maximum Matchings in Bipartit Graphs, SIAM J. Comput., 2(4):225–231
3. Micali, S., Vazirani, V. V. (1980) An $O(\sqrt{|V|} \cdot |E|)$ Algorithm for Finding Maximum Matching in General Graphs, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 12–27