Last update: July 17, 2023 Translated From: e-maxx.ru

Kuhn's Algorithm for Maximum Bipartite Matching

Problem

You are given a bipartite graph G containing n vertices and m edges. Find the maximum matching, i.e., select as many edges as possible so that no selected edge shares a vertex with any other selected edge.

Algorithm Description

Required Definitions

- A **matching** M is a set of pairwise non-adjacent edges of a graph (in other words, no more than one edge from the set should be incident to any vertex of the graph M). The **cardinality** of a matching is the number of edges in it. All those vertices that have an adjacent edge from the matching (i.e., which have degree exactly one in the subgraph formed by M) are called **saturated** by this matching.
- A **maximal matching** is a matching M of a graph G that is not a subset of any other matching.
- A **maximum matching** (also known as maximum-cardinality matching) is a matching that contains the largest possible number of edges. Every maximum matching is a maximal matching.
- A **path** of length k here means a *simple* path (i.e. not containing repeated vertices or edges) containing k edges, unless specified otherwise.
- An **alternating path** (in a bipartite graph, with respect to some matching) is a path in which the edges alternately belong / do not belong to the matching.
- An **augmenting path** (in a bipartite graph, with respect to some matching) is an alternating path whose initial and final vertices are unsaturated, i.e., they do not belong in the matching.
- The **symmetric difference** (also known as the **disjunctive union**) of sets A and B, represented by $A \oplus B$, is the set of all elements that belong to exactly one of A or B, but not to both. That is, $A \oplus B = (A - B) \cup (B - A) = (A \cup B) - (A \cap B)$.

Berge's lemma

This lemma was proven by the French mathematician **Claude Berge** in 1957, although it already was observed by the Danish mathematician **Julius Petersen** in 1891 and the Hungarian mathematician **Denés Kőnig** in 1931.

Formulation

A matching M is maximum $\Leftrightarrow$ there is no augmenting path relative to the matching M.

Proof

Both sides of the bi-implication will be proven by contradiction.

1. A matching M is maximum $\Rightarrow$ there is no augmenting path relative to the matching M.

   Let there be an augmenting path P relative to the given maximum matching M. This augmenting path P will necessarily be of odd length, having one more edge not in M than the number of edges it has that are also in M. We create a new matching M' by including all edges in the original matching M except those also in the P, and the edges in P that are not in M. This is a valid matching because the initial and final vertices of P are unsaturated by M, and the rest of the vertices are saturated only by the matching $P \cap M$. This new matching M' will have one more edge than M, and so M could not have been maximum.

   Formally, given an augmenting path P w.r.t. some maximum matching M, the matching $M' = P \oplus M$ is such that $|M'| = |M| + 1$, a contradiction.

2. A matching M is maximum $\Leftarrow$ there is no augmenting path relative to the matching M.

   Let there be a matching M' of greater cardinality than M. We consider the symmetric difference $Q = M \oplus M'$. The subgraph Q is no longer necessarily a matching. Any vertex in Q has a maximum degree of 2, which means that all connected components in it are one of the three -

   - an isolated vertex
   - a (simple) path whose edges are alternately from M and M'
   - a cycle of even length whose edges are alternately from M and M'

   Since M' has a cardinality greater than M, Q has more edges from M' than M. By the Pigeonhole principle, at least one connected component will be a path having more edges from M' than M. Because any such path is alternating, it will have initial and final vertices unsaturated by M, making it an augmenting path for M, which contradicts the premise. $\blacksquare$

Kuhn's algorithm

Kuhn's algorithm is a direct application of Berge's lemma. It is essentially described as follows:

First, we take an empty matching. Then, while the algorithm is able to find an augmenting path, we update the matching by alternating it along this path and repeat the process of finding the augmenting path. As soon as it is not possible to find such a path, we stop the process - the current matching is the maximum.

It remains to detail the way to find augmenting paths. Kuhn's algorithm simply searches for any of these paths using depth-first or breadth-first traversal. The algorithm looks through all the vertices of the graph in turn, starting each traversal from it, trying to find an augmenting path starting at this vertex.

The algorithm is more convenient to describe if we assume that the input graph is already split into two parts (although, in fact, the algorithm can be implemented in such a way that the input graph is not explicitly split into two parts).

The algorithm looks at all the vertices v of the first part of the graph: $v = 1 \ldots n_1$. If the current vertex v is already saturated with the current matching (i.e., some edge adjacent to it has already been selected), then skip this vertex. Otherwise, the algorithm tries to saturate this vertex, for which it starts a search for an augmenting path starting from this vertex.

The search for an augmenting path is carried out using a special depth-first or breadth-first traversal (usually depth-first traversal is used for ease of implementation). Initially, the depth-first traversal is at the current unsaturated vertex v of the first part. Let's look through all edges from this vertex. Let the current edge be an edge (v, to). If the vertex to is not yet saturated with matching, then we have succeeded in finding an augmenting path: it consists of a single edge (v, to); in this case, we simply include this edge in the matching and stop searching for the augmenting path from the vertex v. Otherwise, if to is already saturated with some edge (to, p), then will go along this edge: thus we will try to find an augmenting path passing through the edges $(v, to), (to, p), \ldots$. To do this, simply go to the vertex p in our traversal - now we try to find an augmenting path from this vertex.

So, this traversal, launched from the vertex v, will either find an augmenting path, and thereby saturate the vertex v, or it will not find such an augmenting path (and, therefore, this vertex v cannot be saturated).

After all the vertices $v = 1 \ldots n_1$ have been scanned, the current matching will be maximum.

Running time

Kuhn's algorithm can be thought of as a series of n depth/breadth-first traversal runs on the entire graph. Therefore, the whole algorithm is executed in time $O(nm)$, which in the worst case is $O(n^3)$.

However, this estimate can be improved slightly. It turns out that for Kuhn's algorithm, it is important which part of the graph is chosen as the first and which as the second. Indeed, in the implementation described above, the depth/breadth-first traversal starts only from the vertices of the first part, so the entire algorithm is executed in time $O(n_1 m)$, where n_1 is the number of vertices of the first part. In the worst case, this is $O(n_1^2 n_2)$ (where n_2 is the number of vertices of the second part). This shows that it is more profitable when the first part contains fewer vertices than the second. On very unbalanced graphs (when n_1 and n_2 are very different), this translates into a significant difference in runtimes.

Implementation

Standard implementation

Let us present here an implementation of the above algorithm based on depth-first traversal and accepting a bipartite graph in the form of a graph explicitly split into two parts. This implementation is very concise, and perhaps it should be remembered in this form.

Here n is the number of vertices in the first part, k - in the second part, $g[v]$ is the list of edges from the top of the first part (i.e. the list of numbers of the vertices to which these edges lead from v). The vertices in both parts are numbered independently, i.e. vertices in the first part are numbered $1 \ldots n$, and those in the second are numbered $1 \ldots k$.

Then there are two auxiliary arrays: mt and used. The first - mt - contains information about the current matching. For convenience of programming, this information is contained only for the vertices of the second part: $\text{mt}[i]$ - this is the number of the vertex of the first part connected by an edge with the vertex i of the second part (or -1, if no matching edge comes out of it). The second array is used: the usual array of "visits" to the vertices in the depth-first traversal (it is needed just so that the depth-first traversal does not enter the same vertex twice).

A function try_kuhn is a depth-first traversal. It returns true if it was able to find an augmenting path from the vertex v, and it is considered that this function has already performed the alternation of matching along the found chain.

Inside the function, all the edges outgoing from the vertex v of the first part are scanned, and then the following is checked: if this edge leads to an unsaturated vertex to, or if this vertex to is saturated, but it is possible to find an increasing chain by recursively starting from $\text{mt}[to]$, then we say that we have found an augmenting path, and before returning from the function with the result true, we alternate the current edge: we redirect the edge adjacent to to to the vertex v.

The main program first indicates that the current matching is empty (the list mt is filled with numbers -1). Then the vertex v of the first part is searched by $\mathrm{try\_kuhn}$, and a depth-first traversal is started from it, having previously zeroed the array used.

It is worth noting that the size of the matching is easy to get as the number of calls $\mathrm{try\_kuhn}$ in the main program that returned the result true. The desired maximum matching itself is contained in the array mt.

```
int n, k;
vector<vector<int>> g;
vector<int> mt;
vector<bool> used;

bool try_kuhn(int v) {
    if (used[v])
        return false;
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
            mt[to] = v;
            return true;
        }
    }
    return false;
}

int main() {
    //... reading the graph ...

    mt.assign(k, -1);
    for (int v = 0; v < n; ++v) {
        used.assign(n, false);
        try_kuhn(v);
    }

    for (int i = 0; i < k; ++i)
        if (mt[i] != -1)
            printf("%d %d\n", mt[i] + 1, i + 1);
}
```

We repeat once again that Kuhn's algorithm is easy to implement in such a way that it works on graphs that are known to be bipartite, but their explicit splitting into two parts has not been given. In this case, it will be necessary to abandon the convenient division into two parts, and store all the information for all vertices of the graph. For this, an array of lists g is now specified not only for the vertices of the first part, but for all the vertices of the graph (of course, now the vertices of both parts are numbered in a common numbering - from 1 to n). Arrays mt and are used are now also defined for the vertices of both parts, and, accordingly, they need to be kept in this state.

Improved implementation

Let us modify the algorithm as follows. Before the main loop of the algorithm, we will find an **arbitrary matching** by some simple algorithm (a simple **heuristic algorithm**), and only then we will execute a loop with calls to the $\mathrm{try\_kuhn}()$ function, which will improve this matching. As a result, the algorithm will work noticeably faster on random graphs - because in most graphs, you can easily find a matching of a sufficiently large size using heuristics, and then improve the found matching to the maximum using the usual Kuhn's algorithm. Thus, we will save on launching a depth-first traversal from those vertices that we have already included using the heuristic into the current matching.

For example, you can simply iterate over all the vertices of the first part, and for each of them, find an arbitrary edge that can be added to the matching, and add it. Even such a simple heuristic can speed up Kuhn's algorithm several times.

Please note that the main loop will have to be slightly modified. Since when calling the function $\mathrm{try\_kuhn}$ in the main loop, it is assumed that the current vertex is not yet included in the matching, you need to add an appropriate check.

In the implementation, only the code in the $\mathrm{main}()$ function will change:

```
int main() {
    // ... reading the graph ...

    mt.assign(k, -1);
    vector<bool> used1(n, false);
    for (int v = 0; v < n; ++v) {
        for (int to : g[v]) {
            if (mt[to] == -1) {
                mt[to] = v;
                used1[v] = true;
                break;
            }
        }
    }
    for (int v = 0; v < n; ++v) {
        if (used1[v])
            continue;
        used.assign(n, false);
        try_kuhn(v);
    }

    for (int i = 0; i < k; ++i)
        if (mt[i] != -1)
            printf("%d %d\n", mt[i] + 1, i + 1);
}
```

Another good heuristic is as follows. At each step, it will search for the vertex of the smallest degree (but not isolated), select any edge from it and add it to the matching, then remove both these vertices with all incident edges from the graph. Such greed works very well on random graphs; in many cases it even builds the maximum matching (although there is a test case against it, on which it will find a matching that is much smaller than the maximum).

Notes

- Kuhn's algorithm is a subroutine in the **Hungarian algorithm**, also known as the **Kuhn-Munkres algorithm**.
- Kuhn's algorithm runs in $O(nm)$ time. It is generally simple to implement, however, more efficient algorithms exist for the maximum bipartite matching problem - such as the **Hopcroft-Karp-Karzanov algorithm**, which runs in $O(\sqrt{n}m)$ time.
- The minimum vertex cover problem is NP-hard for general graphs. However, Kőnig's theorem gives that, for bipartite graphs, the cardinality of the maximum matching equals the cardinality of the minimum

vertex cover. Hence, we can use maximum bipartite matching algorithms to solve the minimum vertex cover problem in polynomial time for bipartite graphs.

Practice Problems

- Kattis - Gopher II
- Kattis - Borders