Project - Advanced Algorithmics - 2021/2022

In this project, we implement the Hopcroft-Karp algorithm that finds a maximum matching in graphs.

1 Introduction

In the real world, we need to pair things up: students with tutors, courses with classrooms, interviews with time slots. Moreover, sometimes not all pairings are possible: TP sessions need to be in classrooms with machines, and an interviewee may only be available in the afternoon.

Such problems can be modeled by *maximal matching in bipartite graphs*. A *bipartite graph* is a *undirected, unweighted* graph G such that its vertex set V can be divided into two parts V_1 and V_2, symbolizing the two types of objects we want to pair up (for example V_1 for courses and V_2 for classroom). Moreover, all the edges in G link one vertex in V_1 to one in V_2. In other words, each edge e in G can be written as $\{v_1, v_2\}$ such that $v_1 \in V_1$, $v_2 \in V_2$. See Figure 1(a) for an example of bipartite graph. The edges express possible pairings between two types of objects (*e.g.*, a course in V_1 is linked to a classroom in V_2 by an edge if we can put this course in this classroom in the planning). In such a bipartite graph G, a planning is thus represented by a *matching* M, which is a subset of edges in G that *do not share any vertices*. This is because we want to avoid double booking. See Figure 1(b) for an example of a matching. We say a node v in V is *matched* if there is some edge e in M containing v, and *unmatched* if it has no edge in M.

In the situation of planning, we usually want to find a planning that makes up as many pairs as possible. In terms of matching, it means that we want to find a matching with the maximal number of edges. See Figure 1 for an example of a maximum matching. This is called the *maximum matching problem*, and can be solved by the Hopcroft-Karp algorithm with time complexity $O(|V|^{1/2}|E|)$.

2 The Hopcroft-Karp algorithm

The core concept in the Hopcroft-Karp algorithm is *augmenting path*, which is a vertex-distinct path of that starts from and ends at unmatched vertices, passing through edges

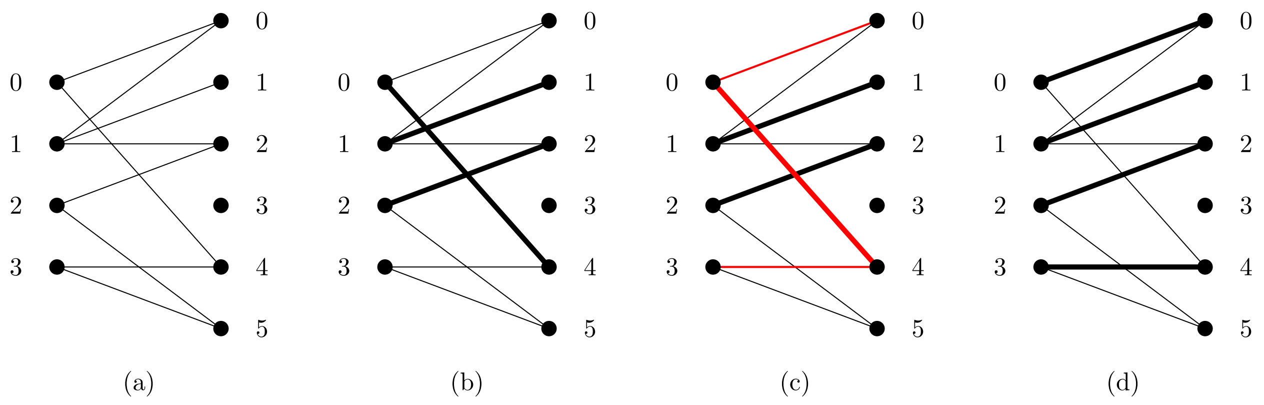

Figure 1: Examples of bipartite graph and matching: (a) a bipartite graph; (b) a matching; (c) an augmenting path in the matching (b); (d) the matching obtained from augmenting the matching in (c) by the augmenting path in (c), it is also a maximum matching.

$e_1, e_2, \ldots, e_k$ such that $e_1, e_3, e_5, \ldots$ are not in the matching M, while $e_2, e_4, e_6, \ldots$ are in the matching M. It is clear that the last edge e_k of an augmenting path cannot be in M, as the end of the path is unmatched. Therefore, the length k of an augmenting path is always odd. Given a matching M and an augmenting path P represented as the set of its edges, we can construct a larger matching M' by taking the symmetric difference $M' = M \oplus P$, that is, removing the edges $e_2, e_4, e_6, \ldots$ from M and adding the edges $e_1, e_3, e_5, \ldots$. As P as an odd number of edges, the new matching M' has one more edge than M, thus larger. This is why P is called an "augmenting path"

Instead of finding an augmenting path each time, the Hopcroft-Karp algorithm finds a maximal set of augmenting paths without sharing vertices to improve the current matching, and repeat until no such augmenting path exists. In this case, we know that no improvement is possible, and the matching is maximum. To find a maximal set of augmenting paths in each iteration, we use a combination of BFS and DFS. Given a bipartite graph $G = (V, E)$ with $V = V_1 \cup V_2$ the two parts, we try to improve an existing matching M by taking the following steps.

- First, we perform a BFS while marking the level of visited vertices, starting from all unmatched vertices in V_1 as level 0, with a special rule: when passing from an even level $2k$ to an odd level $2k+1$, we **only take edges not in** M, and when passing from an odd level $2k-1$ to an even level $2k$, we **only take edges in** M. We stop after having a level containing unmatched vertices in V_2, meaning that we have at least one augmenting path, or after exploring all vertices, meaning that no augmenting path exists. In the latter case, we can terminate the algorithm.

- Second, we perform a DFS starting from unmatched vertices in V_1, but according to the levels computed in the BFS: when we are exploring the neighbors of a vertex v on level k, we can only explore those on level $k+1$. Furthermore, as in the BFS, when passing from an even level $2k$ to an odd level $2k+1$, we **only take edges not in** M, and when passing from an odd level $2k-1$ to an even level $2k$, we **only take edges in** M. Whenever we encounter an unmatched vertex in V_2 during the DFS, we find an augmenting path. We stop the current DFS immediately, improve the matching using this augmenting path, mark all vertices of this augmenting path as visited, and start DFS on another unmatched vertex in V_1. We repeat until having done DFS for all unmatched vertices in V_1.

The BFS in the first step has two purposes: separate vertices in levels so that each path with incrementing levels alternates between edges not in M and those in M; find the length of the shortest augmenting path. Once the levels are separated, the DFS in the second step can find the augmenting paths, while ensuring that these paths do not share vertices, meaning that we can improve the matching by all of them without interference. We give a pseudo-code of the algorithm in Figure 2.

3 Graphs used in the project

The graph files used in this project is a text file with extension `.gr` containing:

- First line: two integers `n1, n2`, which are the sizes of V_1 and V_2.
- Second line: an integer `m`, which is the number of edges.
- The following `m` lines: two integers $v1, v2$ representing an edge between the vertex labeled $v1$ in V_1 and the one labeled $v2$ in V_2. **Note that the labels in V_1 (resp. V_2) starts from `0` and ends at `n1-1` (resp. `n2-1`).**

See the end of the subject for an example of the graph file of the graph in Figure 1(a).

4 What you need to do for the project

For this project, you need to first implement the Hopcroft algorithm in Java, using interfaces and classes for graphs that you have implemented in TP. This implementation will be used to compute a perfect matching of maximal cardinality in any graph given in files in the format indicated in the previous section. Your program should have the following functionalities:

- Reading the text file containing the graph. Your program should receive the filename (without extension) as an argument on command line.

```
levelBFS(G, V1, M, level)
    queue = {vertices in V1 that is unmatched in M}
    newqueue = {}
    initialize the array of level and visited
    clvl = 0                                // current level
    while queue is not empty
        for each v in queue
            for each neighbor v' of v not in visited
                if (clvl even and {v,v'} not in M) or (clvl odd and {v,v'} in M)
                    add v' to newqueue and visited
                    level[v'] = clvl + 1    // current level
        increment clvl, clear queue
        if any vertex in newqueue is unmatched and not in V1
            return true                     // existence of augmenting paths
        exchange queue and newqueue
    return false                            // no augmenting path

levelDFS(G, V1, M, level, visited, v)
    add v into visited
    if v is not in V1 and unmatched in M then
        return true                         // Augmenting path found
    for each neighbor v' of v not visited
        // Augmenting path condition
        if (level[v] even and {v,v'} not in M) or (level[v] odd and {v,v'} in M)
            if level[v'] == level[v]+1 and levelDFS(G, V1, M, level, visited, v')
                // We are on the way back from an augmenting path
                // So we switch the current edge in the matching
                if {v, v'} is in M then remove it from M, otherwise add it to M
                return true
    return false                            // No augmenting path here

Hopcroft_Karp(G, V1, V2)
    M = {}
    initialize the array of level and visited
    while levelBFS(G, V1, M, level)         // Still has augmenting paths
        for each unmatched v in V1
            levelDFS(G, V1, M, level, visited, v)
        clear up the array of level and visited
    return M
```

Figure 2: Pseudo-code for the Hopcroft-Karp algorithm

- Your program should then construct the graph, and compute a perfect matching with maximal number of edges, using your implementation of the Hopcroft-Karp algorithm.
- Your program writes an output file with the same filename but with the extension `.sol` at the end. In the file, **your program should first print the number of edges in the computed perfect matching, then each edge `a b` in the perfect matching, in increasing order in the first node.**
- Your program should also print on the screen the name of the input and the output file, with the size of matching and the number of iterations in the while loop of the Hopcroft-Karp algorithm.

Here are some reminders.

1. The graph is **undirected**. Furthermore, in the graph file, we label the vertices in V_1 and V_2 independently, so we may have some vertex in V_1 that has the same label as some other vertex in V_2. You need to convert the labels of vertices in V_2 so that all vertices have distinct labels.
2. **Your program will be checked automatically against a test set of graphs.** Please make sure that your program produces correct results.
3. You may enrich the interface `Graph` in order to implement the algorithm. You should also choose an appropriate implementation of your interface.
4. Given that the algorithm is already provided, the main work for you is to implement it correctly. Therefore, any error in the implementation will be severely penalized as a lack of effort. You are supposed to thoroughly test your implementation (for instance, it should effectively find out a shortest path when there is one). You are highly encouraged to do such tests on graphs, complicated and simple ones, that you construct by yourself. You should also test for corner cases.
5. You are encouraged to first implement a simpler algorithm that uses DFS to find one augmenting path at each time. It serves two purpose: as the prototype of `levelDFS`, and as a way to check the correctness of your algorithm.

You should submit an archive that contains:

- Full sources of your program, reasonably commented;
- Home-brew test files that you used (not the ones in Section 3), with comments;

- A README file (preferably in plain text, otherwise in pdf), which contains the necessary information to run your program, and extra explanation of your code that you consider helpful for understanding.

Caution: There are many articles on the Internet about the Hopcroft-Karp algorithm, from which you may draw inspirations. However, beware of that some of them may contain errors in the description of the algorithm.

At the end of this subject, we provide an example of the intended execution of your program.

Example of execution on a small graph in the format given in Section 3:

Input: `g1.gr`

```
4 6
9
1 0
2 2
0 4
3 4
2 5
0 0
1 1
1 2
3 5
```

Output file: `g1.sol`

```
4
0 0
1 1
2 2
3 4
```

Screen output:

```
File g1.gr, solution g1.sol
Matching with 4 edge(s)
Using 1 iteration(s)
```