Algorithms and data structures Project 1

Matthijs Muis, Allart de Kroon

November 2023

1 Introduction

Before discussing our algorithm, we introduce some terminology and notation to explain how we model the problem. Let B denote the set of boxes. If box A fits in Box B, we say $A \prec B$, we see that under this relation $(\mathcal{B}, \prec)$ forms a partially ordered set (henceforth called a *poset*). Note that $\mathcal{B}$ is only partially ordered, since there are boxes which do not fit in each other, so neither $A \prec B$ nor $B \prec A$ holds.

If we have a subset $\mathcal{A}$ of $\mathcal{B}$ such that for all $A, B \in \mathcal{A}$ we have $A \prec B$ or $B \prec A$, we call $\mathcal{C}$ a *chain*.

Suppose we have a collection of boxes which we can put together into a single box. Then for all boxes A en B in this collection we must have either $A \prec B$ or $B \prec A$. So we see that a collection of boxes which fit together corresponds directly to a *chain* our poset.

The problem at hand is to find the minimum amount of boxes necessary to fit all boxes in them. We essentially want to partition $\mathcal{B}$ into several collections, such that every collection forms a chain. This leads us to conclude that we can solve the problem by finding a minimal chain partition of $\mathcal{B}$.

We will use a bipartite matching algorithm to find the minimal chain partition of $\mathcal{B}$. Before we can do this, we must first convert our problem to a graph theory problem. We define our bipartite graph $G = (V, E)$ as follows:

- $V = L \cup R$ where $L = \mathcal{B} \times \{0\}$, $R = \mathcal{B} \times \{1\}$ (the disjoint union of $\mathcal{B}$ with $\mathcal{B}$).
- If $A \in L$ and $B \in R$, we have $(A, B) \in E$ if $A \prec B$.

For an example of such a graph based on the first test case in the assignment, see figure 1.

Condsider such a graph based on $\mathcal{B}$ and suppose $|\mathcal{B}| = n$. Assume we have found a max matching M in this graph with $|M| = m$. We can then construct a family of chains $\mathcal{P}$ by starting with all boxes and putting A and B into the same chain if $(A, B) \in M$. It is clear that $\mathcal{P}$ is then a partition of $\mathcal{B}$ into $n - m$ chains. We claim that a smaller partition is not possible. Suppose we do have a smaller partition $\overline{\mathcal{P}}$ with $\left|\overline{\mathcal{P}}\right| = k < n - m$. We can then construct a matching $\overline{M}$ by saying that $(A, B) \in \overline{M}$ if A and B are in the same chain. This matching then has size

$$\left|\overline{M}\right| = n - k > n - (n - m) = m.$$

Which contradicts our assumption that M is a max matching. So we conclude that for a max matching M and a minimum chain partition $\mathcal{P}$ we have

$$\left|\overline{\mathcal{P}}\right| = |\mathcal{B}| - |M|.$$

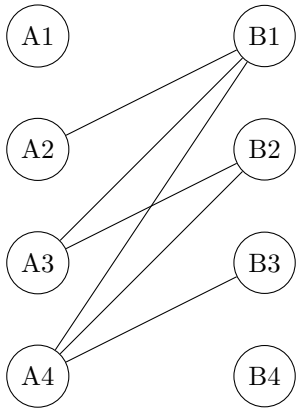

Figure 1: A bipartite graph constructed based on the first test case in the assignment. The number of each node is which box it represents, and the letter represents wether it is on the left or right of the graph.

Before we can discuss the bipartite matching algorithm, we will first introduce some more terminology related to bipartite matching problems.

Let M denote a matching. If $A \in V$ is not an endpoint of an edge in M, we call A a *free vertex*. A path P is called augmenting if its endpoints are free, and it alternates between edges in M and edges in $E \backslash M$.

Let M and N be sets of edges in some graph G, the *symmetric difference* between M and N is defined as

$$(M \cup N) \backslash (M \cap N),$$

or

$$\{(u,v) \mid (u,v) \in M \text{ and } (u,v) \notin N, \text{ or } (u,v) \notin M \text{ and } (u,v) \in N\}.$$

For an example, see figure 2.

2 Algorithm

If we wish to check if box A fits in box B, it is rather inefficient to chek all possible orientations of A until we find an aorientation for which A fits in B. We circumvent this issue as follows: Before running our algorithm, we sort the dimensions of every box. So $(0.9, 0.8, 0.7)$ becomes $(0.7, 0.8, 0.9)$. If we then want to check if $A = (a_1, a_2, a_3)$ fits into $B = (b_1, b_2, b_3)$, we can simply chek if $a_1 < b_1$, $a_2 < b_2$ and $a_3 < b_3$ all hold.

2.1 Bipartite matching

The general idea of the algorithm is to start with some matching M, find a maximal set of shortest augmenting paths such that no two paths share a vertex, and then augment M using these paths. This process repeats until no more augmenting paths can be found. Let M be some matching in G. We transform G into a directed graph by directing each edge in M from left to right, and each edge in $E \backslash M$ from right to left. We start by adding a dummy node to G, a node s connected to

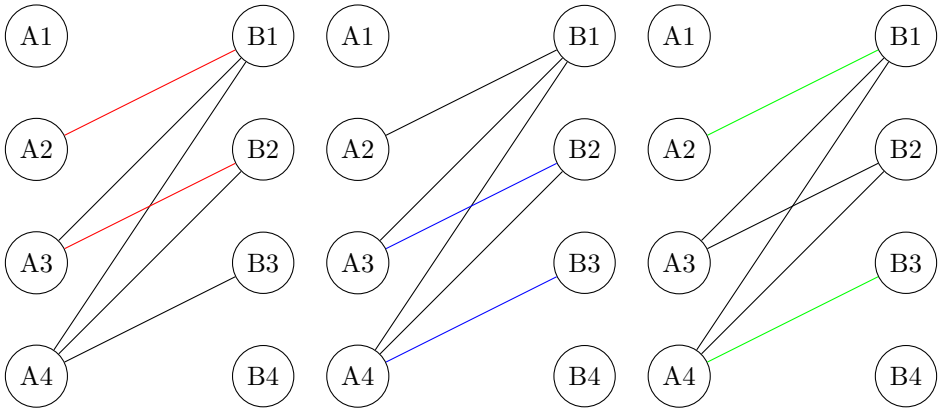

Figure 2: The first two graphs show two sets of edges, the third graph shows their symmetric difference.

every unmatched vertex on the left. We then run a Breadth-First Search (BFS) strarting from s, where we demand that the search always picks a matched edge when going from right to left, and an unmatched edge when going from left to right. The search ends when one or more free vertices on the right hand side are found. We can view this as if we add a vertex t connected to every unmatched vertex on the right and run BFS from s to t. During the search, we keep track of the vertices that precede a vertex (The parents of that vertex) in a dictionary.

This process creates a residual graph $\overline{G}$ consisting of several layers, denoted by L_i. Where each L_i consists of all vertices visited during the i^{th} iteration of BFS. The edges between the layers are constructed as follows: If $v \in L_{i+1}$ is visited from $u \in L_i$ during BFS, we add the edge (u, v) to $\overline{G}$. For an example of such a graph $\overline{G}$, see figure 3.

The algorithm then moves on to finding a maximal set of augmenting paths, such that none of the paths share a vertex. This is done by using a Depth-First Search (DFS). We iterate over the vertices in the final layer of $\overline{G}$ and run DFS from that vertex to s. During the search, we demand DFS only picks vertices from the parents of our current vertex which are marked as unused. When a vertex u is visited, it is immediately marked as used. This is done because if there is no path from u to s at this point in the search, there will not be a path from u to s at any point in the search, so it is not worth visiting. By imposing this limitation, we ensure that DFS has time complexity $\mathcal{O}(|E|)$.

Once such a set of augmenting paths $P_1, \ldots, P_k$ has been found, we create a larger matching $\overline{M}$ by taking the symmetric difference between M and $P_1, \ldots, P_k$:

$$\overline{M} = \{(u, v) | (u, v) \in P_i \textit{ for some } i \textit{ and } (u, v) \notin M, \textit{ or } (u, v) \notin P_i \textit{ for all } i \textit{ and } (u, v) \in M\}.$$

We then repeat the process with the new matching $\overline{M}$ instead of M. Once no more augmenting paths can be found, we have found the max matching and the algorithm ends. For the pseudocode we refer to appendix A.

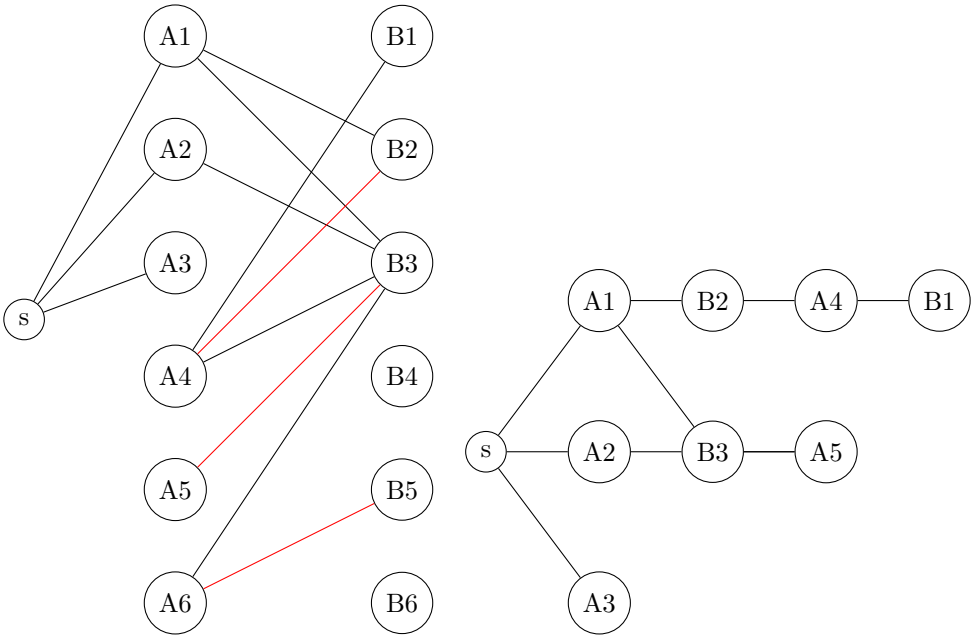

Figure 3: A matching (indicated in red) and the corresponding residual graph, here B1 is the first free vertex reached.

3 Correctness

Since we have already shown that finding a max matching is equivalent to solving the assignment, we wil show here that our algorithm does indeed find a max matching. We start by noting that the paths found by our algorithm are indeed augmenting paths. This is because they start at the final layer of $\overline{G}$ and end at s, which are all free vertices. Meanwhile the paths traverse the layers of $\overline{G}$, so by construction of $\overline{G}$ the edges alternate between matched and unmatched, which means that the paths found by DFS are indeed augmenting paths.

Let M be a matching, P be an augmenting path and let $\overline{M}$ denote their symmetric difference. We will first show that $\overline{M}$ is a matching. Suppose we have $(u, v), (w, v) \in \overline{M}$, There are then several possible cases:

- $(u, v), (w, v) \in M$. This is not possible because M is a matching.
- $(u, v), (w, v) \in P$. This would imply $(u, v) \in M$ or $(w, v) \in M$, which would imply that $(u, v) \in \overline{M}$ or $(w, v) \in \overline{M}$, which contradicts our assumption.
- $(u, v) \in M \backslash P, (w, v) \in P \backslash M$. Since v is not a free vertex, it cannot be the endpoint of P, so there is an edge $(u_0, v) \in P$. Since M is a matching and we already have $(u, v) \in M$, we must have $(u_0, v) \notin M$. However, $(u_0, v) \notin M$ implies that $(w, v) \in M$ because P is an augmenting path, which contradicts our assumption.

All cases lead to a contradiction, so we conclude that $\overline{M}$ is indeed a matching. Since the first and

last edges of P do not lie in M and the rest alternate, we have

$$|P \backslash M| = |p \cap M| + 1.$$

This then yields

$$\begin{aligned} |\overline{M}| &= |(M \cup P) \backslash (M \cap P)| \\ &= |(M \cup P)| - (M \cap P) \\ &= |(M \cup P)| - |P \backslash M| + 1 \\ &= |M| + 1. \end{aligned}$$

So we have shown that each augmenting path of M can be used to increase its size.

Our algorithm stops when no more augmenting paths can be found, so we must show that if there are no augmenting paths, the matching is indeed a max matching. We have already seen that an augmenting path can be used to create a larger matching, so a max matching does not have any augmenting paths. Suppose we have a matching M which is not a max matching $\overline{M}$. Let C denote the symmetric difference of M and $\overline{M}$, since M and $\overline{M}$ are both matchings, every vertex is incident to at most 2 edges (one from M and one from $\overline{M}$) in C. This means that C must consist of cycles of even lenght and paths, whose edges alternate between M and $\overline{M}$. We have assumed that $\overline{M}$ is larger than M, so there must be a path which has more edges in $\overline{M}$ than in M. This must mean that the first and last edge of the path lie in $\overline{M}$, since the rest of the edges alternate. So we have found a path whose edges alternate between being in M and not being in M, where the first and last edge are not in M, this is an augmenting path for M. So we have shown that if a matching is not optimal, there exists an augmenting path for that matching, which means our algorithm will continue searching for a better matching.

4 Time complexity

We claim that the algorithm executes at most $2\sqrt{|V|}$ phases of BFS+DFS. We prove this as follows: During each phase, the algorithm finds a maximal set of augmenting paths of a given lenght. So all remaining augmnting paths must be at least 1 edge longer. This means that after the first $\sqrt{|V|}$ phases the shortest augmenting path has at least lenght $\sqrt{|V|}$. Let M be the matching at this point in the algorithm. The symmetric difference between M and the max matching is then a collection of vertex disjoint augmenting paths and cycles. These augmenting paths in P all have lenght at least $\sqrt{|V|}$ and do not share any vertices. Therefore, there can be at most $\sqrt{|V|}$ of them, since we must have $\sqrt{|V|}|P| = |\cup P| \leq |V|$, where we let $|\cup P|$ denote the total number of vertices in all paths in P. This means that the difference between the size of M and the size of the max matching is at most $\sqrt{|V|}$. Each phase of the algorithm finds one or more augmenting paths and uses them to increase the size of M by at least 1, so there can be at most $\sqrt{|V|}$ phases of the algorithm before the size of M is equal to the size of the max matching.

Each phases of the algorithm performs one BFS and one DFS, this means that one phase has time complexity $\mathcal{O}(|E|)$. Since we have shown that at most $\sqrt{|V|}$ phases are performed, the time complexity of the algorithm is $\mathcal{O}\left(|E|\sqrt{|V|}\right)$. Suppose we have a collection of n boxes. Then our graph has $2n$ vertices. The largest number of edges is formed when the n boxes form a perfect chain, in which case the minimum box has $n-1$ edges to all greater boxes, the second-to-minimum

box has $n-2$ edges, and so forth. This gives an upper bound of $\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \mathcal{O}(n^2)$ edges, so the worst-case time complexity of our algorithm in terms of n is $\mathcal{O}\left(n^2\sqrt{n}\right)$.

A Pseudocode

```
function BFS
    visited ← {s}
    queue ← deque(notMatchedLeft)
    queue[0] ← s
    depth ← []
    depth[s] ← 0
    for l ∈ notMatchedLeft do
        depth[l] ← 1
    end for
    current_d ← 1
    found ← False
    parents ← []
    final ← []
    while queue ≠ ∅ do
        n ← deque(queue)
        d ← depth[n]
        if d > current_d then
            if found = True then
                return True, parents, final
            else
                current_d ← d
            end if
        end if
        if n ∉ G̅ then
            Skip this iteration
        end if
        for q ∈ {x ∈ G̅[n] | x ∉ visitedordepth[q] = d + 1} do
            if q ∈ notMatchedRight and q ∉ visited then
                final ← final.append(q)
                found ← True
            end if
            if q ∉ visited then
                enqueue(queue, q)
            end if
            depth[q] ← d + 1
            visited ← visited.add(q)
            if q ∈ parents then
                parents[q] ← parents[q].append(n)
            else
                parents[q] ← [n]
            end if
        end for
    end while
    return found, parents, final
end function
```

```
function DFS(parents, final)
    used ← ∅
    parent ← []
    for qf ∈ final do
        n ← qf
        while n ≠ s do
            if {x ∈ parents[n] | x ∉ used} ≠ ∅ then
                q ← list ({x ∈ parents[n] | x ∉ used}) [0]
                parent[n] ← q
                if q ≠ s then
                    used ← used.add(q)
                else
                    notMatchedLeft ← notMatchedLeft.remove(n)
                end if
            else
                Break
            end if
            n ← parent[n]
        end while
        if n = s then
            used ← used.add(qf)
        end if
    end for
    return parent, {q ∈ final | ∈ used}
end function
```

```
function augmentPaths(parent, qs)
    for q ∈ qs do
        notMatchedRight ← notMatchedRight.remove(q)
        matches ← matches + 1
        p ← q
        while p ∈ parent do
            G̅[parent[p]] ← G̅[parent[p]].remove(p)
            if p ∉ G̅ then
                G̅[p] ← [parent[p]]
            else
                G̅[p] ← G̅[p].append(parent[p])
            end if
            p = parent[p]
        end while
    end for
end function

function Make_G̅
    G̅ ← []
    for A ∈ ℬ do
        for B ∈ ℬ do
            if A ≺ B then
                G̅[A] ← G̅[A].append(B)
            end if
        end for
    end for
end function

function solveMatching
    matching ← 0
    notMatchedLeft ← ℬ
    notMatchedRight ← ℬ
    Make_G̅
    while True do
        found, parents, qs ← BFS()
        if found then
            parent, qs ← DFS(parents, qs)
            augmentPaths(parent, qs)
        end if
    end while
end function
```