CS 406 : Algorithmic Graph Theory

Maximum Matching in Bipartite and General Graphs

Bhargey Mehta
201701074

Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar

Chapter 1

Introduction & Motivation

1.1 Blossom Algorithm

The blossom algorithm was developed by Jack Edmonds in 1965. It works on general unweighted graphs. The algorithm is named blossom because the odd length cycles are called blossoms.

The algorithm finds an application in the development of chemistry development kit. In organic chemistry a lot of focus is present on how bonds interact with each other since 2 compounds having the same molecular formula but different bond structure behave differently. While doing computational chemistry one often needs to assign the double bonds to specific places so that the valency of the carbon atoms is satisfied.

This can be posed as a problem in graph theory such that each carbon atom is a vertex and each single bond is an edge. Now we need to assign double bonds to certain singly bonded atoms. These double bonds cannot occur at a single vertex so a single vertex can belong to at maximum 1 double bond. This leads to the fact that the double bonds form a matching in the graph generated by a compound. This leads to a need to solve the maximum matching problem algorithmically in polynomial time.

1.2 Hopcroft-Karp Algorithm

The Hopcroft Karp Algorithm was developed by John Hopcroft and Richard Karp in 1973. It solves a specialised case of the assignment problem when the edge weights are equal i.e. we don't have a preference for any agent or job. This algorithm is of interest since it is faster than the Hungarian algorithm. Applications include matching organ donors to compatible patients, matching (same preference) job applicants to companies in a job fair etc.

1.3 Hungarian Algorithm

The Hungarian algorithm was developed by Harold Kuhn in 1955 and reviewed by James Munkres in 1957 to be polynomial time. Kuhn had developed this algorithm mainly on the basis of work done by Hungarian mathematicians Dénes Kőnig and Jenő Egerváry so he named the algorithm the "Hungarian" algorithm.

The algorithm works on a complete weighted bipartite graph. This is of interest because a well known combinatorial problem called the assignment problem can be posed as finding the maximum or minimum weighted matching in this bipartite setting. We are given a list of agents and a list of tasks. These lists of agents and tasks are nothing but the bipartitions of the graph. Each agent does the listed tasks for some cost which is the edge weight. We want to assign an agent to a single task so that the overall cost is maximum or minimum. This translates to finding a maximum weighted matching in the graph. Applications include assigning machines to factories, bidding and contract assignments etc.

Chapter 2

Blossom Algorithm

2.1 Algorithm

Note that Blossom is a cycle of odd length and unsaturated vertex v (wrt a matching M of a graph G) is a node v belongs to the graph G but not present in matching M

Algorithm 1: FindMaxMatching

```
Data: G, M
Result: M*
P ← FindAugmentingPath(G, M)
if P is empty then
    return M
end
M+ ← increased matching of M by adding alternating edges of P
return FindMaxMatching(G, M+)
```

Algorithm 2: FindAugmentingPath

```
Data: G, M
Result: P
F ← empty forest
rootOfNode ← empty node-to-node mapping
terminals ← unsaturated vertices of G
for each node v in terminals do
    Add v to F
    rootOfNode(v) ← v
mark all edges of M in G
for each node v in terminals do
    for each unmarked edge (v, w) adjacent to v do
        if w ∉ F then
            AddToForest(M, F, v, w, terminals, rootOfNode)
        else
            p_w ← rootOfNode(w)
            if d(w, p_w) is even then
                p_v ← rootOfNode(v)
                if p_v == p_w then
                    P ← BlossomRecursion(G, M, F, v, w)
                else
                    P ← ConstructPath(F, v, w, rootOfNode)
                return P
        mark edge (v, w)
return empty path
```

Algorithm 5: ConstructPath

```
Data: F, v, w, rootOfNode
Result: P
p_v ← rootOfNode(v)
p_w ← rootOfNode(w)
P_1 ← Path from v to p_v in forest F
P_2 ← Path from v to p_w in forest F
return P_1 + P_2
```

Algorithm 3: Blossom Recursion

Data: G, M, F, v, w
Result: P
$B \leftarrow$ nodes in path from $v \rightarrow w$ in F
$G^+ \leftarrow G$ with all nodes of B contracted into w
$M^+ \leftarrow M$ with all nodes of B contracted into w
$P^+ \leftarrow \text{findAugmentingPath}(G^+, M^+)$
if $w \in P^+$ **then**
  $P \leftarrow \text{liftBlossom}(B, P^+, M)$
  return P
return P^+

Algorithm 4: AddToForest

Data: M, F, v, w, terminals, rootOfNode
Result: Null (Utility Function)
$x \leftarrow$ node adjacent to w in M
add edges (v, w) and (w, x) in F
add node x in terminals
$\text{rootOfNode}(w) \leftarrow \text{rootOfNode}(v)$
$\text{rootOfNode}(x) \leftarrow \text{rootOfNode}(v)$

Algorithm 6: LiftBlossom

Data: B, P^+, M
Result: P
$v \leftarrow$ node in B such that it is incident on 2 edges not present in M
$x \leftarrow$ node in B such that $(x, x') \in P^+$ but $(x, x') \notin M$
if *no such x exists* **then**
  return empty path
$P_1 \leftarrow$ subpath of P^+ ending at v
$P_2 \leftarrow$ subpath of P^+ ending at x
$P_B \leftarrow$ path which begins at (x, y) such that $(x, y) \in M$ and ends at v
$P \leftarrow P_1 + P_B + P_2$
return P

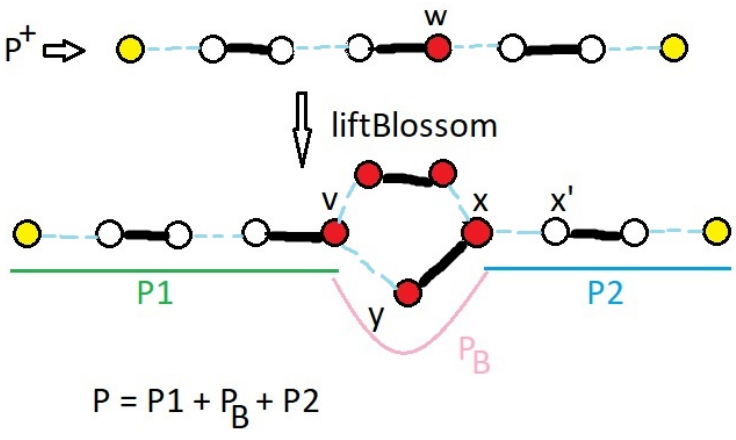

Figure 2.1: Lifting a Blossom

2.2 Proof of Correctness

2.2.1 Lemmas

Augmenting Path Lemma

Given a graph G, a matching M is maximum if and only if there is no augmenting path in G.

This lemma has been proved in lecture so we use it as it is.

Identification Lemma

Identification Lemma: This algorithm identifies Blossoms correctly.

The Blossom Recursion subroutine is executed only when the following conditions hold. This is because the conditions we impose before executing the said subroutine.

1. Edge (v, w) is unmarked.
2. The parent of v and w is same i.e. $p_v = p_w$. Hence both are reachable from each other.
3. Node w is at an even distance from it's root p_v.

Now there are 2 cases.

- Case A: $v \in M$. In this case v must have been added to the list of terminal nodes by some other node and due to the structure of addToForest subroutine, we always add nodes which are at even distance from some original root of forest F.
- Case B: $v \notin M$ or equivalently $v \in F$. In this case v itself is a root and it's distance is 0, hence even.

Hence both v and w are at an even distance from p_v and so the path that exists between them in F is of even length. This path together with the edge (v, w) forms a blossom.

Base Lemma

Since this blossom is formed of edges alternating in M and length of the cycle is odd, there is exactly one node in blossom which is incident on 2 edges which are both absent in M but present in the blossom. We call this vertex as the base of the blossom.

Stem Lemma: If the base is saturated by M, then the augmenting path P^+ which passes through the contracted blossom, passes through it such that it passes through the base.

The above lemma is true since, if it is not the case then it would imply that the vertex (which is not a base) is incident on 2 edges which are present in M which contradicts the definition of M.

Note that in the figure above, node v is the base of the blossom of size 5.

2.2.2 Blossom Lemma

Blossom Lemma: G contains an augmenting path if and only if G^+ (the graph after contraction of blossom) contains an augmenting path.

Note that if the augmenting path in G or G^+ doesn't pass through the blossom, then the proof is trivial since the path in both of them is exactly equal.

Forward Proof

There are 2 cases depending upon whether or not the blossom is internal to the augmenting path in G^+ or is an endpoint.

- Case A: Blossom is internal to the augmenting path.

  Observation A_1: Due to the base lemma proved above, there is one node which is the base (defined in lemma above) and hence the edge which enters into the blossom is in the matching.

Observation A_2: Any other node is in the matching M such that it's corresponding edge belongs to the matching and hence the edge which exits the blossom to the other end is not in the matching.

From the above 2 observations, when the blossom is contracted into a single node, these 2 edges are alternating edges and hence the corresponding path in G^+ is augmenting.

- Case B: Blossom is an endpoint.

  Observation A_2 holds for this case also.

  Now since, the blossom is internal, the blossom becomes an unsaturated vertex since there is no edge incoming to this contracted blossom.

  Hence we can say that the augmenting path in G^+ is aagain an augmenting path.

Backward Proof

Now since the blossom is of odd length, (WLOG) if we consider the base as the entry point and the other vertex as the exit point in the blossom, we can see that there are 2 paths (one travelling the upper path and the other travelling the lower path) between them.

Independent of the position of the exit point wrt the entry point (base vertex), there is always a path of even length and a path of odd length.

If we traverse from the base on the even length portion of the blossom, the first edge is not in the matching and the subsequent edges alternate. And since the path is of even length, the last edge is necessarily in matching.

Now again there are 2 cases.

- Case A: Blossom is internal.

  According to the base lemma, the subpath outside the blossom attaching the unsaturated vertex to the base will end with an edge (incident on base) which is in the matching.

  For the edge connecting the exit point with the other unsaturated vertex, it is necessarily not in matching since the corresponding edge in matching M belongs to the blossom.

  Hence in this case, the paths described above in union with the even length path forms an augmenting path in G.

- Case B: Blossom is and endpoint.

  For this case the first path in case A is an empty path and rest of the reasoning remains the same.

  Hence in this case also, the union of these paths forms an augmenting path in G.

Hence from the above Blossom Lemma in combination with the augmenting path lemma and the identification lemma, we can say that our algorithm is correct.

2.3 Complexity Analysis

- The addToForest subroutine takes O(1) time since all 6 steps complete in constant time.
- The constructPath subroutine takes O(N) time since the maximum length of a path can be N.
- The liftBlossom subroutine takes O(N) time since the maximum length of the path can be N and the decision to decide the subportion of the blossom to include in lifting the path is done in constant time.
- The findMaxMatching function is called recursively at max $\frac{N}{2}$ times since the size of the matching increases by 2 in every recursion call. Hence the time complexity is O($N*$ cost of findAugmentingPath subroutine)
- Each call to the blossomRecursion subroutine will take at max O($N + M$) time barring the recursion since in each call the cost is equal to the size of the blossom found.

- In each call to the augmenting path finding algorithm, we either find a blossom and recurse or we find an augmenting path and return. We call the work done (traversing the edges and contraction of blossom) of $O(N+M)$ in the algorithm except the recursion work as a sub phase. Hence each recursion is a sub phase. There will be at most $O(N)$ sub phases. Hence the cost is $O(NM)$

Finally plug the above result in time complexity of findMaxMatching function. Hence overall time complexity of the entire algorithm is $O(N * O(M * N)) = O(N^2M)$.

2.4 Parallel Algorithm

Now in the core subroutine given below, the outer for loop which iterates over vertices v can't be parallelised since after each iteration, the state of the tree changes and hence there is an inherent dependency. However the inner loop which iterates over the adjacent edges of some vertex v can be done in parallel. However in the sequential version, after processing one edge, we may either find a path (without the need of a Blossom Recursion) or we find no augmenting path and there is some blossom or there is no augmenting path. So if we allow each iteration to modify the forest in parallel, the state of the forest becomes inconsistent if one iteration finds a path and other doesn't. Hence we have to delay the actual modification of the forest until all edges have been processed. Note that this implies that we can't use the AddToForest subroutine.

Algorithm 7: FindAugmentingPath

```
Data: G, M
Result: P
F ← empty forest
rootOfNode ← empty node-to-node mapping
terminals ← unsaturated vertices of G
for each node v in terminals do
    Add v to F
    rootOfNode(v) ← v
mark all edges of M in G
for each node v in terminals do
    temp ← empty map
    for each unmarked edge (v, w) adjacent to v in PARALLEL do
        if w ∉ F then
            x ← node adjacent to w in M
            temp(w) ← x
        else
            p_w ← rootOfNode(w)
            if d(w, p_w) is even then
                p_v ← rootOfNode(v)
                if p_v == p_w then
                    P ← BlossomRecursion(G, M, F, v, w)
                else
                    P ← ConstructPath(F, v, w, rootOfNode)
                return P
        mark edge (v, w)
    for each node w in temp in PARALLEL do
        if w == temp(temp(w)) then
            P ← BlossomRecursion(G, M, F, v, w)
            return P
        else
            add edges (v, w) and (w, temp(w) = x) in F
            add node x = temp(w) to terminals
return empty path
```

The maximum matching wrapper depends on the previous matching and so can't be parallelised however augmenting a matching can be done in parallel.

Algorithm 8: FindMaxMatching

$P \leftarrow$ FindAugmentingPath(G, M)
if P *is empty* **then**
  return M
end
$M^+ \leftarrow$ increased matching of M by adding alternating edges of P in PARALLEL
return FindMaxMatching(G, M^+)

The construct path algorithm is sequential by nature and hence can't be parallelised. The best we can do is find P_1&P_2 in parallel. This doesn't scale as the number of processors increases.

Algorithm 9: ConstructPath

Data: F, v, w, rootOfNode
Result: P
$p_v \leftarrow$ rootOfNode(v)
$p_w \leftarrow$ rootOfNode(w)
$P_1 \leftarrow$ Path from v to p_v in forest F in PARALLEL with P_2
$P_2 \leftarrow$ Path from v to p_w in forest F in PARALLEL with P_1
return $P_1 + P_2$

In the below algorithm, the once the blossom is found, the contraction of the blossom can be done in parallel since the presence of each edge in the blossom is independent from other edges. The lifting algorithm remains unchanged due to dependencies in the path finding process.

Algorithm 10: Blossom Recursion

Data: G, M, F, v, w
Result: P
$B \leftarrow$ nodes in path from $v \rightarrow w$ in F
$G^+ \leftarrow G$ with all nodes of B contracted into w in PARALLEL
$M^+ \leftarrow M$ with all nodes of B contracted into w in PARALLEL
$P^+ \leftarrow$ findAugmentingPath(G^+, M^+)
if $w \in P^+$ **then**
  $P \leftarrow$ liftBlossom(B, P^+, M)
  return P
return P^+

2.4.1 Proof of Correctness

we will consider the parallelism seen in the augmenting path finding algorithm since we're deviating from the serial counterpart.

When extending the forest in parallel, we are only considering the effect of adding one unmarked edge at a time. Since when processed in isolation, if blossoms are formed due to addition of this single edge then our algorithm is no different from the serial counterpart in terms of behaviour. Hence, to prove the correctness our algorithm, we only need to consider the case when a blossom is formed due to addition of edges from the same node v.

Suppose two edges (v, w_i) and (v, w_j) are some two edges. Their corresponding marked edges are (w_i, x_i) and (w_j, x_j). Note that x_i and x_j are BOTH not present in the forest at this moment since if one of them were present then we would found a blossom while considering the other, before reaching at this situation. Both (v, w_i) and (v, w_j) are unmarked so v is necessarily the base of the blossom. Now consider the following 2 cases:

- Blossom is of size 5 or greater: If it is blossom of size 5 or more then the remaining part of the blossom should already be present in the forest since we have just added 3 edges and discovered a blossom. But this leads to a contradiction to our original assumption that x_i and x_j were both absent in the forest.
- Blossom is of size 3: From above, we can see that the blossom formed would always be a size 3 blossom. We are specifically checking for size 3 blossom in our algorithm after the edges are handled in parallel.

2.4.2 Time Complexity

- There will be at max O($|V|$) iterations of the main algorithm since a matching of size at max $\frac{|V|}{2}$ is possible.
- We now again examine the work done in each sub phase of the algorithm. We have to iterate over all the vertices and then their edges in parallel. This means work of O($|V|\frac{|V|}{p}$).
- In the same sub phase, the work done by the blossom recursion is O($|V| + \frac{|E|}{p}$) since the blossom identification cannot be parallelised.
- Each sub phase thus takes O($|V| + |V|\frac{|V|}{p} + \frac{|E|}{p}$) task. There can be at most O(V) of these sub phases. Hence total time of the augmenting path finding algorithm takes O($|V|^2 + \frac{|V|^3}{p} + \frac{|V||E|}{p}$)

Hence the overall time complexity would be O($|V| \times (|V|^2 + \frac{|V|^3}{p} + \frac{|V||E|}{p})$) = O($|V|^3 + \frac{|V|^4}{p} + \frac{|V|^2|E|}{|p|^2}$).

2.5 Distributed Algorithm

As discussed earlier, due to the sequential nature of the algorithm, it is very difficult to parallelise the blossom algorithm. In the blossom contraction, several edges have to be collapsed into a single node. This requires that the graph be modified and further recursions be applied upon this modified graph.

In a distributed setting this requires some central authority that handles this contraction. We will be using 2 algorithms A_1 and A_2. A_1 rebuilds the graph locally be obtaining the information from the network and then applies A_2 locally to produce the required result.

The algorithm A_2 would be the blossom algorithm in our case. We are assuming that a single vertex can contain the information of the entire network and there is a authority vertex which drives the collection of the network information and then runs the blossom algorithm locally.

2.5.1 Correctness and Message Complexity

The algorithm works by sending a discovery message to all adjacent neighbours. Neighbours which have not been discovered would then reply their parents with an accepted response. This way a parent-child relationship is established which forms a DFS tree in the network.

Now the vertex itself sends discovery message to its neighbours to determine its children and so on. Any new discovery message is responded with a discovery reject response.

In this DFS network, the leaf nodes will first send their adjacency info to their parents since their child set C would be empty. The vertices which are internal will wait for all children to send them adjacency information. This would contain the the network information of all the children, grand-children etc of that vertex. This vertex now unions this information with its own adjacency list and then forwards it to their parent. In this way the entire network is now accumulated at the driver node v_a which then runs the blossom algorithm locally.

Along each edge we have 3 messages- discovery, discovery result and adjacency information hence the message complexity of the algorithm would be O($3|E|$).

Algorithm 11: A_1: Build network

```
Data: authority vertex id v_a and own id v
C ← empty set
if v == v_a then
    p ← NULL
    for each adjacent w do
        send DISCOVER msg
    for each adjacent w do
        receive DISCOVERY_RESULT
        if DISCOVERY_RESULT is accepted then
            add w to C
    for each adjacent w do
        receive DISCOVER msg
        send DISCOVERY_REJECTED to w
    for each c_i ∈ C do
        receive ADJACENCY_INFO from c_i
    run the BLOSSOM ALGORITHM locally
    print the result and shut down
else
    receive 1 DISCOVER msg from some w
    send DISCOVERY_ACCEPTED to w
    p ← w
    for each adjacent w except p do
        send DISCOVER msg
    for each adjacent w except p do
        receive DISCOVERY_RESULT
        if DISCOVERY_RESULT is accepted then
            add w to C
    for each adjacent w except p do
        receive DISCOVER msg
        send DISCOVERY_REJECTED to w
    subTreeInfo ← NULL
    for each c_i ∈ C do
        receive ADJACENCY_INFO from c_i
        subTreeInfo ← subTreeInfo ∪ ADJACENCY_INFO
    send subTreeInfo to p and shut down
```

2.6 References

- Blossom Algorithm Analysis by Amy Shoemaker & Sagar Vare
- Wikipedia Blossom Algorithm

Chapter 3

Hopcroft-Karp Algorithm

3.1 Algorithm

Note that X and Y are the bipartitions of the bipartite graph G and M is some matching of G.

Algorithm 12: BFS

```
Data: X, Y, M
Result: G_BFS
G_BFS ← empty graph
visited ← empty Set
Q ← empty Queue
for each node v in B_1 do
    if v ∉ M then
        enqueue v in Q
        add v to visited
while Q is NOT empty do
    v ← dequeue from Q
    if v ∈ X then
        for each edge (v, w) adjacent to v such that (v, w) ∉ M do
            add (v, w) to G_BFS
            if w ∈ visited then
                skip edge
            enqueue w in Q
            add w to visited
    else
        for each edge (v, w) adjacent to v such that (v, w) ∈ M do
            add (v, w) to G_BFS
            if w ∈ visited then
                skip edge
            enqueue w in Q
            add w to visited
return G_BFS
```

Algorithm 13: ActualDFS

```
Data: G_BFS, M, v, visited, P
Result: Boolean
if v in visited then
    return False
add v to visited
for all edges (v, w) adjacent to v in G_BFS do
    if (v ∈ X and (v, w) ∉ M) or (v ∈ Y and (v, w) ∈ M) then
        skip edge
    append (v, w) to P
    if w ∉ M then
        add w to visited return True
    foundPath ← actualDFS(G_BFS, M, w, visited, P)
    if foundPath is True then
        return True
    remove (v, w) from P
return False
```

Algorithm 14: DFS (wrapper)

```
Data: G_BFS, M
Result: set of disjoint paths
visited ← empty set
disjointPaths ← empty set
for each node v in Y do
    if v ∉ M then
        P ← empty path
        ActualDFS(G_BFS, M, v, visited, P)
        add P to disjointPaths if P is NOT empty
return disjointPaths
```

Algorithm 15: HopcroftKarp

```
Data: X, Y, G
Result: M*
M ← empty matching
do
    G_BFS ← BFS(X, Y, M)
    disjointPaths ← DFS(G_BFS, M)
    augment M with each P in disjointPaths
while disjointPaths is NOT empty;
return M
```

3.2 Proof of Correctness

3.2.1 Augmenting Path Lemma

Augmenting Path Lemma: A matching M of G is maximum if and only if there is no augmenting path in G.

This lemma has been proved in the lecture, so we use it as it is.

3.2.2 Disjoint Paths Lemma

Disjoint Path Lemma: The paths returned by the DFS subroutine are vertex disjoint and are augmenting paths of M wrt G.

The graph formed by the BFS has 2 types of nodes, belonging to X and belonging to Y. They can be imagined as stacks of layers going from top to bottom as shown in the figure. It is easy to see that the algorithm selects edges going from $X \to Y$ only when the edge is NOT present in the matching and selects edges going from $Y \to X$ only when the edge IS present in the matching.

Hence when we apply DFS from a node $v \in Y$, we are guaranteed to find a node $w \in X$ and $w \in M$ such that $(v, w) \notin M$. Similarly when we move from a node $v \in X$ and $v \in M$, we are guaranteed to find a node $w \in Y$ such that $(v, w) \in M$.

Also see that the DFS returns a path which begins and ends at nodes NOT present in the matching. Further the path contains edges alternate between X and Y such that all edges from $X \to Y \in M$ and all edges from $Y \to X \notin M$. Hence this path is an augmenting path.

Moreover this path is also disjoint since after using a node we immediately mark it as used. Hence it is effectively removed from the graph G_{BFS} and cannot be shared with any other path found by further iterations of the DFS algorithm.

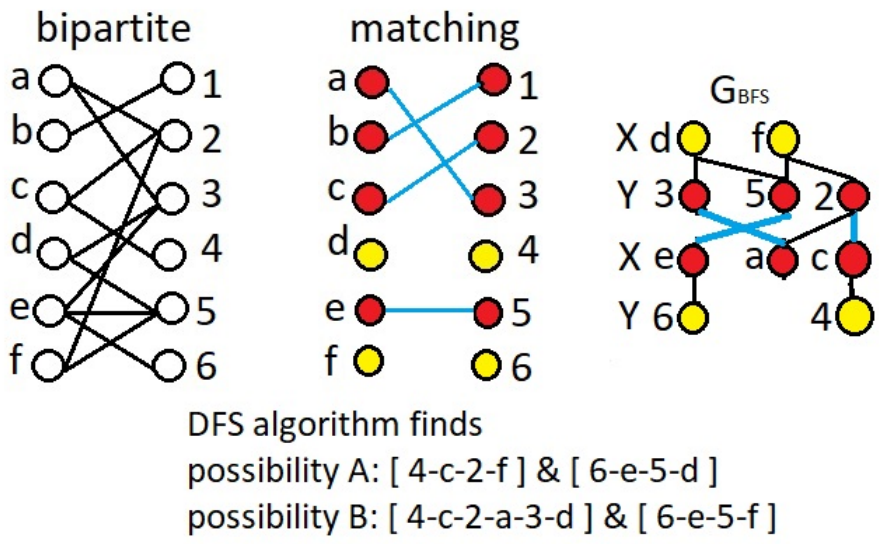

Figure 3.1: Finding Augmenting Paths

3.3 Time Complexity

3.3.1 Claim 1

We make a claim that for a maximum matching M^* and some matching matching M, the symmetric difference of M^* and $M = M \Delta M^*$ has $|M^*| - |M|$ augmenting paths and these paths are vertex disjoint.

We know that the presence of an augmenting path increases the size of M by 1. So there are exactly $|M^*| - |M|$ augmenting paths. Also since the degree of each node is at max 2, the paths are vertex-disjoint.

3.3.2 Claim 2

We make a claim that there are at max $2\sqrt{N}$ iterations in the main loop.

Notice that each iteration of the main algorithm increases the size of the matching by at least 1. Hence the shortest augmenting path is of length $\sqrt{N}$ after $\sqrt{N}$ iterations.

Since these paths will be vertex disjoint, we can say that there are at maximum $\sqrt{N}$ augmenting paths. Each of these paths increases the size of M by 1 and hence $|M^*| - |M|$ is at max $\sqrt{N}$. Hence optimal matching M^* will be achieved after at max $\sqrt{N}$ more iterations of the main algorithm.

3.3.3 Calculating Time Complexity

- Each iteration involves BFS, DFS and augmenting M with vertex disjoint paths.
- Both BFS and DFS take $O(|E|)$ time.
- Since the paths used to augment M are vertex disjoint, the entire process takes $O(|V|)$ time.
- Hence each iteration of the main algorithm takes $O(|V| + |E|) = O(|E|)$ time.
- From claim 2, we can see that there are at max $O(\sqrt{|V|})$ iterations.

So the overall time complexity is $O(|E|\sqrt{|V|})$.

3.4 Parallel Algorithm

The main 2 components of the Hopcroft-Karp Algorithm are augmenting the matching and disjoint path finding in the bipartite graph. The augmentation of the matching is has a dependency i.e. to augment some matching, we need the matching itself.

The second component i.e. disjoint path finding again consists of a modified BFS and DFS. Attempting the BFS in a prallel manner results in the need of synchronisation between the processors at such a high level that the parallelism is negligible even when ignoring overheads. Similarly is the case for DFS. To make the paths vertex disjoint we have to make use of mutual exclusion when considering the neighbours of a vertex. This again results in overheads (checking all neighbours of a vertex before performing the recursion) in the algorithm that actually increase the time complexity.

However experiments have shown that using these techniques, an experimental speedup is obtained even with a poorer time complexity than the serial algorithm.

3.5 Distributed Algorithm

The algorithm can be distributed with the help of a central authority driving the phases of the algorithm performing the BFS in a synchronised manner (achieved with the help of the leader taking charge of this) and then a distributed alternating DFS while implementing look ahead strategy so that different vertices do not claim a single vertex to be part of two different augmenting paths since the paths needs to be vertex disjoint.

3.6 References

- Wikipedia article on the Hopcroft-Karp Algorithm
- A Parallel Tree Grafting Algorithm for Maximum Cardinality Matching in Bipartite Graphs by Ariful Azad, Aydin Buluc, Alex Pothen
- Michael M. Wu; Michael C. Loui (1990). An efficient distributed algorithm for maximum matching in general graphs

Chapter 4

Hungarian Algorithm

4.1 Introduction

We are given a complete bipartite graph G with $|X| = |Y|$ such that X denotes the jobs and Y denotes the agents. The edge weight $w(x, y)$ of edge $e(x, y)$ denotes the cost of doing job y when assigned to agent x.

We will first clarify some terminologies related to the algorithm.

4.1.1 Alternating Tree

A sub graph of G which is a tree having an unsaturated root wrt some matching M having the edges alternate between being in M and not being in M is called an alternating tree. Hence we can see that if a leaf node is unsaturated in this alternating tree, then the path from root to this leaf node is an augmenting path.

4.1.2 Labelling Function

A labelling function is a mapping $l : V \to Z$ such that it assigns an integer to each node in G. Hence $l(v)$ denotes the label of node $v \in G$.

A labelling is called feasible if for every edge $e(x, y)$ we have,

$$l(x) + l(y) \geq w(x, y)$$

.

4.1.3 Equality Graph

Let E_l denote the set of edges $e(x, y)$ such that wrt some labelling function l, the equality $l(x)+l(y) = w(x, y)$ holds.

The graph G_l formed by the vertices of G and the edge set E_l is called the equality graph G_l wrt the labelling function.

4.2 Lemmas

4.2.1 Kuhn-Munkres Lemma

The Kuhn-Munkres Lemma (abbreviated as KM Lemma from now on) is at the heart of this algorithm. It states that,

KM Lemma: If l is a feasible labelling and M is a perfect matching in G_l then M is maximum weight matching in G.

Let M' be any perfect matching in G not necessarily maximum and not necessarily in G_l. We have the weight $w(M') = \Sigma w(x, y)$ where $e(x, y) \in M'$.

Since M' is a perfect matching, each vertex appears only once in this summation. Since l is feasible, we have $w(x, y) \leq l(x) + l(y)$, we get the following result.

$$w(M') = \Sigma w(x, y) \leq \Sigma l(v)$$

.

Now consider a perfect matching M in G_l. Note that by definition of G_l we have $w(x, y) = l(x) + l(y)$. Hence we get the following result.

$$w(M) = \Sigma w(x, y) = \Sigma l(v)$$

Combining these results we get $W(M) \geq W(M')$ and hence M is maximum weight matching in G.

4.2.2 Relabelling Lemma

We will now look at a relabelling lemma which will aid is in modifying a given feasible l into another feasible labelling l'.

If $S \subseteq X$ then let $N_l(S)$ denote the set of vertices $v \in Y$ such that for some $u \in X$, $e(u, v) \in E_l$.

Relabelling Lemma: Let $S \subseteq X$ and let $T = N_l(S) \neq Y$, we have

$$\alpha_l = \min(\{l(x) + l(y) - w(x, y) : x \in S, y \notin T\})$$

$$l'(v) = \begin{cases} l(v) - \alpha_l & v \in S \\ l(v) + \alpha_l & v \in T \\ l(v) & \text{otherwise} \end{cases}$$

is a feasible labelling such that

1. If $e(x, y) \in L$ such that $x \in S, y \in T$, then $e(x, y) \in E_{l'}$.
2. If $e(x, y) \in L$ such that $x \notin S, y \notin T$, then $e(x, y) \in E_{l'}$.
3. $\exists$ edge $e(x, y)$ with $x \in S, y \notin T$ such that $e(x, y) \in E_{l'}$

Proof

1. $x \in S, y \in T$: The same amount is added on one side and subtracted on another so the sum remains the same. If $e(x, y) \in E_l$, then $l(x) + l(y) = w(x, y) = (l(x) - \alpha_l) + (l(y) + \alpha_l)$.
2. $x \in S, y \notin T$: In this case, for each edge, we have some slack about how much we can reduce $l(x)$, since α_l is the minimum of that slack, the labelling remains feasible. In addition the edges which correspond to the minimum slack are now present in $E_{l'}$.
3. $x \notin S, y \in T$: In this case the label of y is being increased by α_l hence l' remains feasible.
4. $x \notin S, y \notin T$: The labelling remains the same, hence it is feasible.

4.3 Algorithm

Algorithm 16: HungarianMatching

```
Data: G
Result: M*
M ← empty matching
for all x ∈ X do
    l(x) ← max(w(x, y))
for all y ∈ Y do
    l(y) ← 0
while M is not perfect matching do
    P, l ← FindAugmentingPath(G, l, M)
    M ← augment M with P
return M
```

Algorithm 17: FindAugmentingPath

Data: G, l, M
Result: P, l^*
pick unsaturated vertex $u \in X$
$S \leftarrow \{u\}$ and $T \leftarrow \phi$
$P \leftarrow$ NULL
while *P is NULL (path not found)* **do**
  if $N_l(S) = T$ **then**
    $\alpha_l \leftarrow \min(\{l(x) + l(y) - w(x,y) : x \in S, y \notin T\})$
    $$l^*(v) \leftarrow \begin{cases} l(v) - \alpha_l & v \in S \\ l(v) + \alpha_l & v \in T \\ l(v) & \text{otherwise} \end{cases}$$
  $l \leftarrow l^*$
  pick one v from $\in N_l(S) - T$
  if *v is unsaturated* **then**
    $P \leftarrow$ path from v to root u
  else
    $z \leftarrow$ the matched vertex of v by edge $e(v,z) \in M$
    $S \leftarrow S \cup \{z\}$ and $T \leftarrow T \cup \{v\}$
return P, l^*

4.4 Proof of Correctness

It is easy to see that the initial labelling function is feasible.

We claim that the alternating tree T_u rooted u in the augmenting path finding algorithm is always contained in the labelling even after we modify the labelling according to the relabelling lemma. This claim can be proved by induction as below.

- Base Case: T_u only contains u. It is trivial to see that the claim holds.
- Assume that the edges of T_u are in E_l. Note that this is equivalent to saying $T_u \cap X = S$ and $T_u \cap Y = T$. But this means that every edge in T_u is such that one of its end points is in S and another in T. Hence by reason #1 of the relabelling lemma explained in section 2, all these edges are present in E_{l^*} as well.
- We also add new edges but those are already according to the updated labelling and hence present T_u according to E_{l^*} as well.

Hence we prove that our alternating tree T_u always increases in size. And since M was not perfect, we are guaranteed to find an unsaturated vertex z eventually which would provide us the augmenting path P..

Note that we are always selecting edges from the set E_l and by definition these edges are present in G_l. We have also proved in the relabelling lemma that the modified labelling is always feasible and have also proved by induction that the edges are always present in the modified labelling.

This means that the matching eventually becomes a perfect matching in G_l and by the KM Lemma proved earlier in section 2, the resultant matching will be a maximum weight matching in G.

4.5 Time Complexity

1. In the augmenting path finding algorithm, the augmenting path would be found after at max O($|V|$) iterations of the outer loop.
2. The minimum of the slacks i.e. α_l can be found in O($|V|^2$) time and all the new labelling can be obtained after O($|V|$) updates. Adding v to T and z to S is constant time task Hence one single iteration takes O($|V|$) time.

3. The overall time complexity of the path finding algorithm is $O(|V|^3)$ since there are $O(|V|)$ iterations each costing $O(|V|^2)$ time.

4. The Hungarian algorithm augments the matching. A single augmentation increases the size of the matching by 1 and hence there will be $O(|V|)$ iterations of the path finding algorithm.

Thus the overall time complexity of the Hungarian algorithm would be $O(|V|^4)$.

4.6 Parallel Algorithm

Like in the above algorithms, Hungarian algorithm's path augmenting step cannot be parallelised due to the matching dependency. When considering the augmenting path algorithm, it contains finding minimum and updates of slacks.

The updates of slacks can be done in parallel. Finding the minimum of the slacks in parallel can take $O(\frac{|V|^2}{p} + p)$. Thus the time complexity could then be reduced to $O(|V|^2(\frac{|V|^2}{p} + p)) = O(p|V|^3 + \frac{|V|^4}{p})$.

4.7 Distributed Algorithm

The Hungarian algorithm can be distributed with the help of a central authority which can store information of $O(|V|)$ which it anyways has to do since the degree of a vertex in a complete bipartite graph is $\frac{|V|}{2} - 1$. The intermediate matching is stored in the central vertex. It then directs other vertices to calculate their slacks and send it back. Selecting the minimum slack, it then proceeds to assign new slacks and updates the alternating tree which again can be stored in $O(|V|)$ space.

4.8 References

- Notes by Dr. Mordecai J Golin

Chapter 5

Related Algorithms

5.1 Unweighted Maximum Matching in Bipartite Graphs

- 1973: J Hopcroft, R Karp; An $N^{\frac{5}{2}}$ Algorithm for Maximum Matchings in Bipartite Graphs
- 1991: H Alt, N Blum, K Mehlhorn, M Paul; Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}m \log n)$
- 1995: T Feder, R Motwani; Clique Partitions, Graph Compression and Speeding-Up Algorithms

5.2 Weighted Maximum Matching in Bipartite Graphs

- 1955: H Kuhn ; The Hungarian method for the assignment problem
- 1960: M Iri; A New Method for Solving Transportation-Network Problems
- 1969: E Dinic, M Konrod; An Algorithm for the Solution of the Assignment Problem
- 1985: H Gabow; Scaling algorithms for network problems
- 1987: M Fredman, R Tarjan; Fibonacci heaps and their uses in improved network optimization algorithms
- 1989: H Gabow, R Tarjan; Faster Scaling Algorithms for Network Problems
- 1999: M Kao, T Lam, W Sung, H Ting; A Decomposition Theorem for MaximumWeight Bipartite Matchings with Applications to Evolutionary Trees
- 2001: M Kao, T Lam, W Sung, H Ting; A Decomposition Theorem for Maximum Weight Bipartite Matchings
- 2014: S Das, K Kapoor; Fine-Tuning Decomposition Theorem for Maximum Weight Bipartite Matching

5.3 Unweighted Maximum Matching in General Graphs

- 1965: J Edmonds; Paths, Trees and Flowers
- 1972: H Gabow; An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs
- 1974: T Kameda, I Munro; A O($|V||E|$) algorithm for maximum matching of graphs
- 1975: S Even, O Kariv; An O($N^{2.5}$) algorithm for maximum matching in general graphs
- 1980: S Micali, V Vazirani; An O($\sqrt{|V|}|E|$) algoithm for finding maximum matching in general graphs

5.4 Completely Distributed Algorithms

It is interesting to note that completely distributed maximum matching algorithms which have no sequential counterpart were not found. One of the reason could be that all these algorithms make use of the fact that a matching would have an augmenting path in case it is not maximum which is a property of the graph that turns up at a global level.

References

All the references used to create this report are listed below. The references are hyperlinks.

1. Ariful Azad, Aydin Buluc, Alex Pothen (2015); A Parallel Tree Grafting Algorithm for Maximum Cardinality Matching in Bipartite Graphs
2. Michael M. Wu; Michael C. Loui (1990). An efficient distributed algorithm for maximum matching in general graphs
3. Wikipedia Article on the Hopcroft-Karp Algorithm
4. Wikipedia Article on the Blossom Algorithm
5. Amy Shoemaker & Sagar Vare (2016); Edmond's Blossom Algorithm
6. Dr. Mordecai J Golin; Notes on the Hungarian Algorithm
7. Wikipedia Article on the Hungarian Algorithm
8. Noel O'Boyle, John Mayfield (2017); ACS Talk
9. R Duan, S Pettie, H Su (2014); Scaling Algorithms for Weighted Matching in General Graphs
10. S Das (2020); A Modified Decomposition Algorithm for Maximum Weight Bipartite Matching and Its Experimental Evaluation