Hopcroft–Karp Algorithm for Maximum Matching | Set 2 (Implementation)

We strongly recommend to refer below post as a prerequisite.

Hopcroft–Karp Algorithm for Maximum Matching | Set 1 (Introduction)

There are few important things to note before we start implementation.

1. We need to **find an augmenting path** (A path that alternates between matching and not matching edges, and has free vertices as starting and ending points).
2. Once we find alternating path, we need to **add the found path to existing Matching**. Here adding path means, making previous matching edges on this path as not-matching and previous not-matching edges as matching.

The idea is to use BFS (Breadth First Search) to find augmenting paths. Since BFS traverses level by level, it is used to divide the graph in layers of matching and not matching edges. A dummy vertex NIL is added that is connected to all vertices on left side and all vertices on right side. Following arrays are used to find augmenting path. Distance to NIL is initialized as INF (infinite). If we start from dummy vertex and come back to it using alternating path of distinct vertices, then there is an augmenting path.

1. pairU[]: An array of size m+1 where m is number of vertices on left side of Bipartite Graph. pairU[u] stores pair of u on right side if u is matched and NIL otherwise.
2. pairV[]: An array of size n+1 where n is number of vertices on right side of Bipartite Graph. pairU[v] stores pair of v on left side if v is matched and NIL otherwise.
3. dist[]: An array of size m+1 where m is number of vertices on left side of Bipartite Graph. dist[u] is initialized as 0 if u is not matching and INF (infinite) otherwise. dist[] of NIL is also initialized as INF

Once an augmenting path is found, DFS (Depth First Search) is used to add augmenting paths to current matching. DFS simply follows the distance array setup by BFS. It fills values in pairU[u] and pairV[v] if v is next to u in BFS.

Below is C++ implementation of above Hopkroft Karp algorithm.

```
// C++ implementation of Hopcroft Karp algorithm for
// maximum matching
#include<bits/stdc++.h>
using namespace std;
#define NIL 0
#define INF INT_MAX

// A class to represent Bipartite graph for Hopcroft
// Karp implementation
class BipGraph
{
 // m and n are number of vertices on left
 // and right sides of Bipartite Graph
 int m, n;

 // adj[u] stores adjacents of left side
 // vertex 'u'. The value of u ranges from 1 to m.
 // 0 is used for dummy vertex
 list<int> *adj;

 // These are basically pointers to arrays needed
 // for hopcroftKarp()
 int *pairU, *pairV, *dist;

public:
 BipGraph(int m, int n); // Constructor
 void addEdge(int u, int v); // To add edge

 // Returns true if there is an augmenting path
 bool bfs();

 // Adds augmenting path if there is one beginning
```

```
 // with u
 bool dfs(int u);

 // Returns size of maximum matcing
 int hopcroftKarp();
};

// Returns size of maximum matching
int BipGraph::hopcroftKarp()
{
    // pairU[u] stores pair of u in matching where u
    // is a vertex on left side of Bipartite Graph.
    // If u doesn't have any pair, then pairU[u] is NIL
 pairU = new int[m+1];

    // pairV[v] stores pair of v in matching. If v
    // doesn't have any pair, then pairU[v] is NIL
 pairV = new int[n+1];

 // dist[u] stores distance of left side vertices
 // dist[u] is one more than dist[u'] if u is next
 // to u'in augmenting path
 dist = new int[m+1];

 // Initialize NIL as pair of all vertices
 for (int u=0; u<m; u++)
  pairU[u] = NIL;
 for (int v=0; v<n; v++)
  pairV[v] = NIL;

 // Initialize result
 int result = 0;

 // Keep updating the result while there is an
 // augmenting path.
 while (bfs())
 {
  // Find a free vertex
  for (int u=1; u<=m; u++)

   // If current vertex is free and there is
   // an augmenting path from current vertex
   if (pairU[u]==NIL && dfs(u))
    result++;
 }
 return result;
}

// Returns true if there is an augmenting path, else returns
// false
bool BipGraph::bfs()
{
 queue<int> Q; //an integer queue

 // First layer of vertices (set distance as 0)
 for (int u=1; u<=m; u++)
 {
  // If this is a free vertex, add it to queue
  if (pairU[u]==NIL)
  {
   // u is not matched
   dist[u] = 0;
   Q.push(u);
  }

  // Else set distance as infinite so that this vertex
  // is considered next time
  else dist[u] = INF;
 }

 // Initialize distance to NIL as infinite
 dist[NIL] = INF;
```

```
 // Q is going to contain vertices of left side only.
 while (!Q.empty())
 {
  // Dequeue a vertex
  int u = Q.front();
  Q.pop();

  // If this node is not NIL and can provide a shorter path to NIL
  if (dist[u] < dist[NIL])
  {
   // Get all adjacent vertices of the dequeued vertex u
   list<int>::iterator i;
   for (i=adj[u].begin(); i!=adj[u].end(); ++i)
   {
    int v = *i;

    // If pair of v is not considered so far
    // (v, pairV[V]) is not yet explored edge.
    if (dist[pairV[v]] == INF)
    {
     // Consider the pair and add it to queue
     dist[pairV[v]] = dist[u] + 1;
     Q.push(pairV[v]);
    }
   }
  }
 }

 // If we could come back to NIL using alternating path of distinct
 // vertices then there is an augmenting path
 return (dist[NIL] != INF);
}

// Returns true if there is an augmenting path beginning with free vertex u
bool BipGraph::dfs(int u)
{
 if (u != NIL)
 {
  list<int>::iterator i;
  for (i=adj[u].begin(); i!=adj[u].end(); ++i)
  {
   // Adjacent to u
   int v = *i;

   // Follow the distances set by BFS
   if (dist[pairV[v]] == dist[u]+1)
   {
    // If dfs for pair of v also returns
    // true
    if (dfs(pairV[v]) == true)
    {
     pairV[v] = u;
     pairU[u] = v;
     return true;
    }
   }
  }

  // If there is no augmenting path beginning with u.
  dist[u] = INF;
  return false;
 }
 return true;
}

// Constructor
BipGraph::BipGraph(int m, int n)
{
 this->m = m;
 this->n = n;
 adj = new list<int>[m+1];
}
```

```
// To add edge from u to v and v to u
void BipGraph::addEdge(int u, int v)
{
 adj[u].push_back(v); // Add u to v’s list.
 adj[v].push_back(u); // Add u to v’s list.
}

// Driver Program
int main()
{
 BipGraph g(4, 4);
 g.addEdge(1, 2);
 g.addEdge(1, 3);
 g.addEdge(2, 1);
 g.addEdge(3, 2);
 g.addEdge(4, 2);
 g.addEdge(4, 4);

 cout << "Size of maximum matching is " << g.hopcroftKarp();

 return 0;
}
```

Output:

```
Size of maximum matching is 4
```

The above implementation is mainly adopted from the algorithm provided on Wiki page of Hopcroft Karp algorithm.