Indian Institute of Engineering Science and Technology, Shibpur

# Implementation of Edmond's Blossom Algorithm for Maximum Matching in a Graph

Suddhabrato Ghosh, Rahul Sharma & Bikramjit Saha

May 19, 2022

*Abstract*

The blossom algorithm is an algorithm in graph theory for constructing maximum matchings on graphs. The algorithm was developed by Jack Edmonds in 1961, and published in 1965. This polynomial time algorithm is used in several applications including the assignment problem, the marriage problem, and the Hamiltonian cycle and path problems (i.e., Traveling Salesman Problem). Through this project, we aim to implement the blossom algorithm and visualize the steps leading to the maximum matching by using a graphical interface. The user creates the graph on screen by virtue of mouse inputs and obtains the maximum matching for the given input graph. The interface also allows the user to control the steps which show the implementation of the algorithm. The inputs and outputs are also stored in text files for future reference and reuse.

# 1 | Blossom algorithm

The blossom algorithm is an algorithm in graph theory for constructing maximum matching on graphs. The algorithm was developed by Jack Edmond's in 1961, and published in 1965.

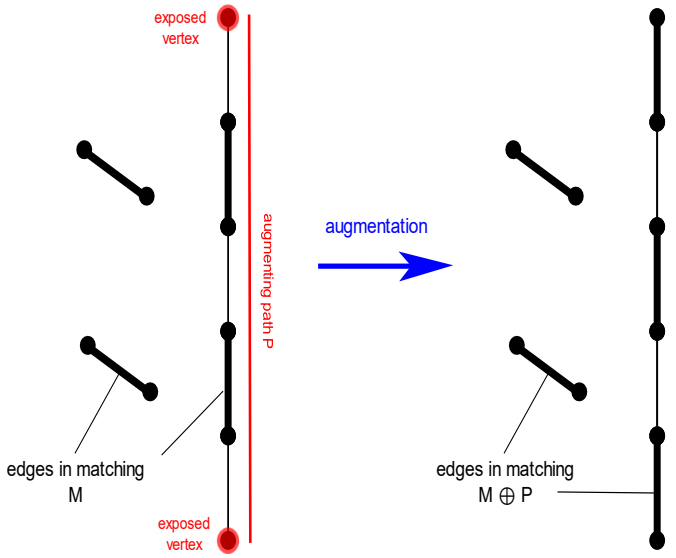
Given a general graph $G = (V, E)$, the algorithm finds a matching $M$ such that each vertex in $V$ is incident with at most one edge in $M$ and $|M|$ is maximized. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph. Unlike bipartite matching, the key new idea is that an odd-length cycle in the graph (blossom) is contracted to a single vertex, with the search continuing iteratively in the contracted graph.

The algorithm runs in time $O(|E||V|^2)$, where $|E|$ is the number of edges of the graph and $|V|$ is its number of vertices. A better running time of $O(|E||V|^{1/2})$ for the same task can be achieved with the much more complex algorithm of Micali and Vazirani.

A major reason that the blossom algorithm is important is that it gave the first proof that a maximum-size matching could be found using a polynomial amount of computation time.

## 1. 1 Augmenting paths

Given $G = (V, E)$ and a matching $M$ of $G$, a vertex $v$ is *exposed* if no edge of $M$ is incident with $v$. A path in $G$ is an *alternating path*, if its edges are alternately not in $M$ and in $M$ (or in $M$ and not in $M$). An *augmenting path* $P$ is an alternating path that starts and ends at two distinct exposed vertices. Note that the number of unmatched edges in an augmenting path is greater by one than the number of matched edges, and hence the total number of edges in an augmenting path is odd. A *matching augmentation* along an augmenting path $P$ is the operation of replacing $M$ with a new $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$.




By Berge's Theorem, matching $M$ is maximum if and only if there is no $M$-augmenting path in $G$. Hence, either a matching is maximum, or it can be augmented. Thus, starting from an initial matching, we can compute a maximum matching by augmenting the current matching with augmenting paths as long as we can find them, and return whenever no augmenting paths are left. We can formalize the algorithm as follows:

```
INPUT:  Graph G, initial matching M on G
   OUTPUT: maximum matching M* on G
A1 function find_maximum_matching(G, M) : M*
A2     P ← find_augmenting_path(G, M)
A3     if P is non-empty then
A4         return find_maximum_matching(G, augment M along P)
A5     else
A6         return M
A7     end if
A8 end function
```

We still have to describe how augmenting paths can be found efficiently. The subroutine to find them uses blossoms and contractions.

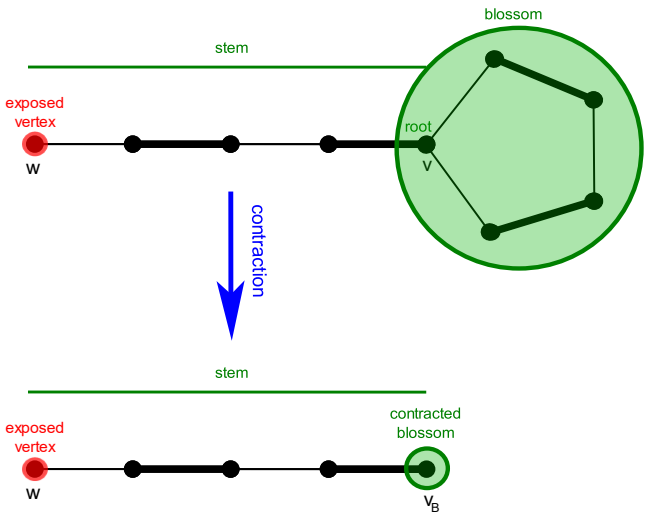
## 1.2 Blossoms and contractions

Given $G = (V, E)$ and a matching $M$ of $G$, a *blossom* $B$ is a cycle in $G$ consisting of $2k+1$ edges of which exactly $k$ belongs to $M$, and where one of the vertices $v$ of the cycle (the *base*) is such that there exists an alternating path of even length (the *stem*) from $v$ to an exposed vertex $w$.

### *Finding Blossoms:*

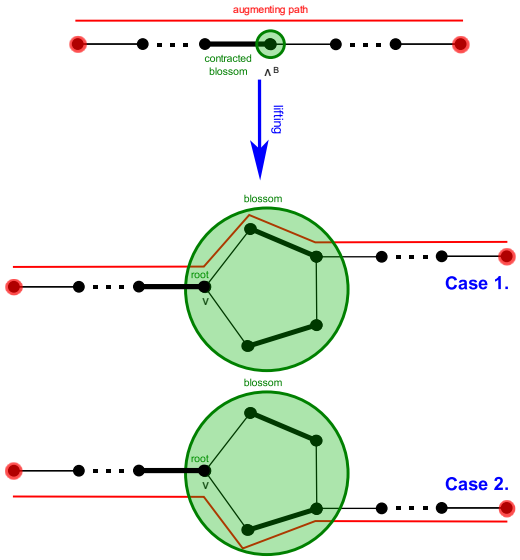
- Traverse the graph starting from an exposed vertex.
- Starting from that vertex, label it as an outer vertex "***o***".
- Alternate the labelling between vertices being inner "***i***" and outer "***o***" such that no two adjacent vertices have the same label.
- If we end up with two adjacent vertices labelled as outer "***o***" then we have an odd-length cycle and hence a blossom.

Define the *contracted graph G'* as the graph obtained from $G$ by contracting every edge of $B$, and define the *contracted matching M'* as the matching of *G'* corresponding to $M$.

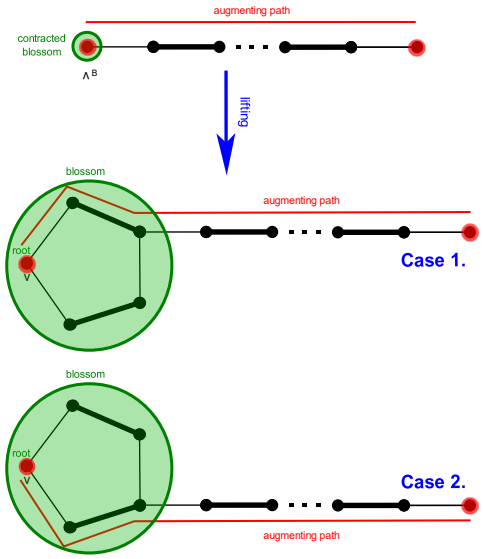


*G'* has an *M'*-augmenting path if and only if *G* has an *M*-augmenting path, and that any *M'*-augmenting path *P'* in *G'* can be *lifted* to an *M*-augmenting path in *G* by undoing the contraction by *B* so that the segment of *P'* (if any) traversing through $v_B$ is replaced by an appropriate segment traversing through *B*. In more detail:

- if *P'* traverses through a segment $u \rightarrow v_B \rightarrow w$ in *G'*, then this segment is replaced with the segment $u \rightarrow (u' \rightarrow \ldots \rightarrow w') \rightarrow w$ in *G*, where blossom vertices *u'* and *w'* and the side of *B*, $(u' \rightarrow \ldots \rightarrow w')$, going from *u'* to *w'* are chosen to ensure that the new path is still alternating (*u'* is exposed with respect to $M \cap B \{w',w\} \in E \setminus M$).




- if *P'* has an endpoint $v_B$, then the path segment $u \rightarrow v_B$ in *G'* is replaced with the segment $u \rightarrow (u' \rightarrow \ldots \rightarrow v')$ in *G*, where blossom vertices *u'* and *v'* and the side of *B*, $(u' \rightarrow \ldots \rightarrow v')$, going from *u'* to *v'* are chosen to ensure that the path is alternating (*v'* is exposed, $\{u',u\} \in E \setminus M$).




Thus, blossoms can be contracted and search performed in the contracted graphs. This reduction is at the heart of Edmonds' algorithm.

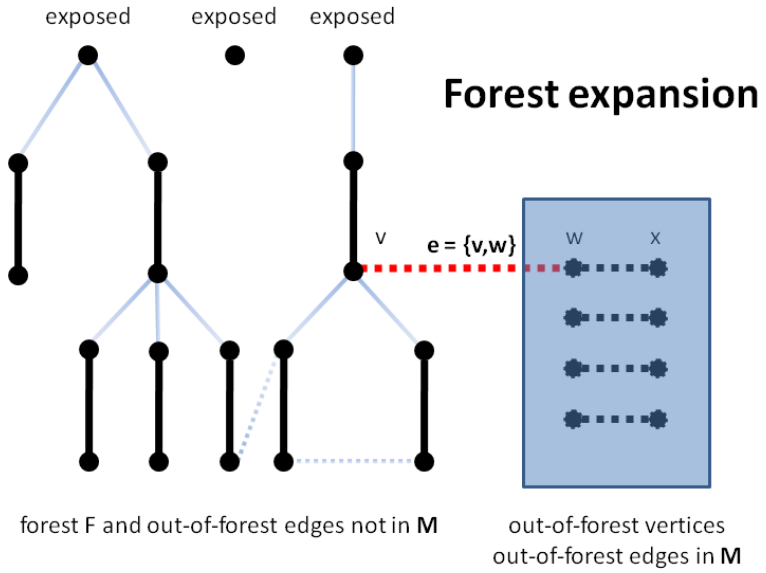
## 1.3 Finding an augmenting path

The search for an augmenting path uses an auxiliary data structure consisting of a forest *F* whose individual trees correspond to specific portions of the graph *G*. In fact, the forest *F* is the same that would be used to find maximum matchings in bipartite graph (without need for shrinking blossoms). In each iteration the algorithm either (1) finds an augmenting path, (2) finds a blossom and recurses onto the corresponding contracted graph, or (3) concludes there are no augmenting paths. The auxiliary structure is built by an incremental procedure discussed next. The construction procedure considers vertices *v* and edges *e* in *G* and incrementally updates *F* as appropriate. If *v* is in a tree *T* of the forest, we let *root(v)* denote the root of *T*. If both *u* and *v* are in the same tree *T* in *F*, we let *distance(u,v)* denote the length of the unique path from *u* to *v* in *T*.

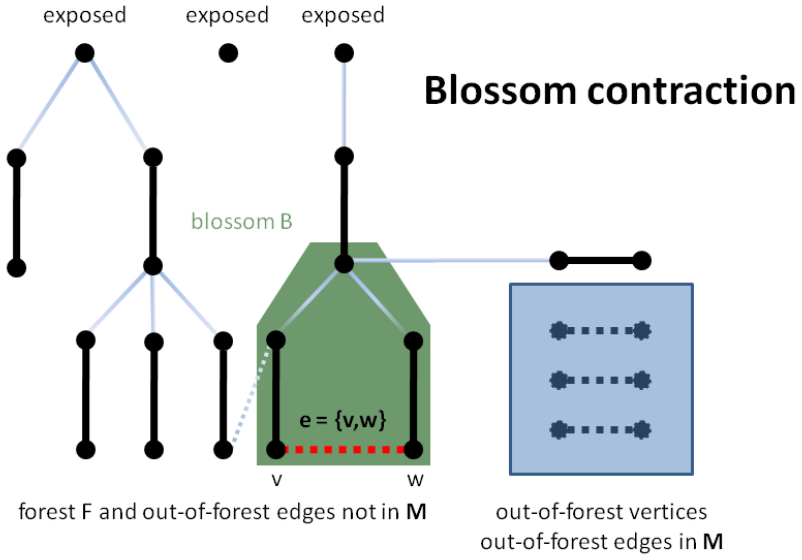
```
INPUT:  Graph G, matching M on G
    OUTPUT: augmenting path P in G or empty path if none found
B01 function find_augmenting_path(G, M) : P
B02    F ← empty forest
B03    unmark all vertices and edges in G, mark all edges of M
B05    for each exposed vertex v do
B06        create a singleton tree { v } and add the tree to F
B07    end for
B08    while there is an unmarked vertex v in F with distance(v, root(v)) even do
B09        while there exists an unmarked edge e = { v, w } do
B10            if w is not in F then
                   // w is matched, so add e and w's matched edge to F
B11                x ← vertex matched to w in M
B12                add edges { v, w } and { w, x } to the tree of v
B13            else
B14                if distance(w, root(w)) is odd then
                       // Do nothing.
B15                else
B16                    if root(v) ≠ root(w) then
                           // Report an augmenting path in F U { e }.
B17                        P ← path (root(v) → ... → v) → (w → ... → root(w))
B18                        return P
B19                    else
                           // Contract a blossom in G and look for the path in the
contracted graph.
B20                        B ← blossom formed by e and edges on the path v → w in T
B21                        G', M' ← contract G and M by B
B22                        P' ← find_augmenting_path(G', M')
B23                        P ← lift P' to G
B24                        return P
B25                    end if
B26                end if
B27            end if
B28            mark edge e
B29        end while
B30        mark vertex v
B31    end while
B32    return empty path
B33 end function
```

### *1.3.1 Examples*

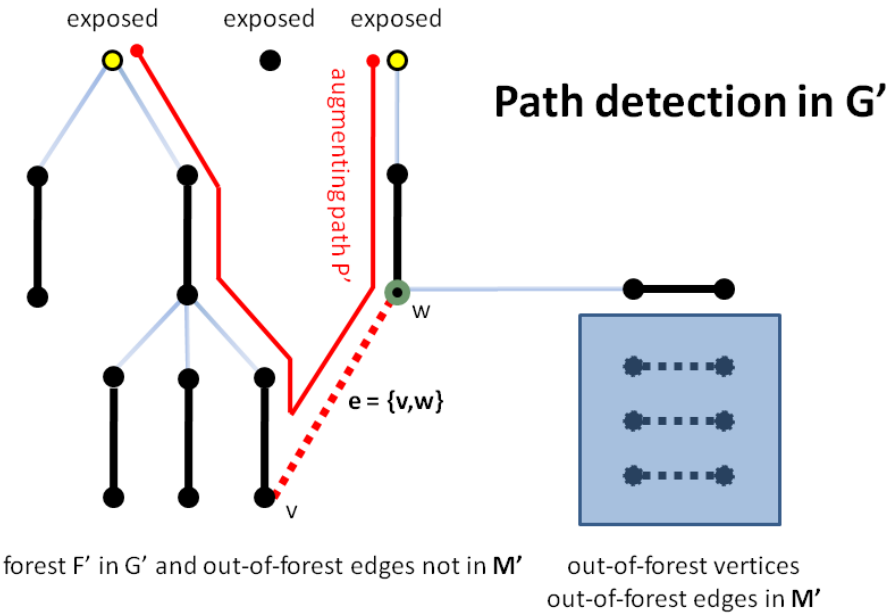
The following four figures illustrate the execution of the algorithm. Dashed lines indicate edges that are currently not present in the forest. First, the algorithm processes an out-of-forest edge that causes the expansion of the current forest (lines B10 – B12).

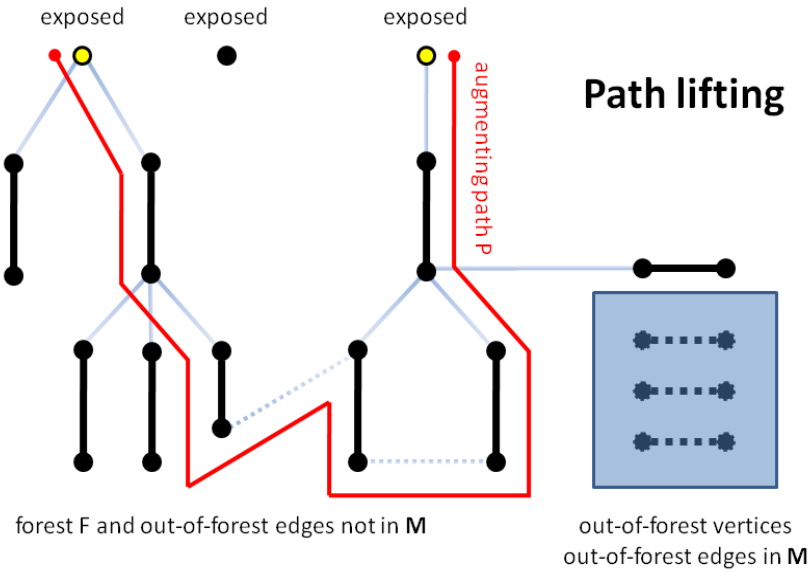



Next, it detects a blossom and contracts the graph (lines B20 – B21).




Finally, it locates an augmenting path P′ in the contracted graph (line B22) and lifts it to the original graph (line B23). Note that the ability of the algorithm to contract blossoms is crucial here; the algorithm cannot find *P* in the original graph directly because only out-of-forest edges between vertices at even distances from the roots are considered on line B17 of the algorithm.


exposed
exposed
exposed
augmenting path P'
Path detection in G'
w
e = {v,w}
v
forest F' in G' and out-of-forest edges not in M'
out-of-forest vertices
out-of-forest edges in M'


exposed
exposed
exposed
augmenting path P
Path lifting
forest F and out-of-forest edges not in M
out-of-forest vertices
out-of-forest edges in M

### *1.3.2 Analysis*

The forest *F* constructed by the *find_augmenting_path()* function is an alternating forest.

- a tree *T* in *G* is an *alternating tree* with respect to *M*, if
    - *T* contains exactly one exposed vertex *r* called the tree root
    - every vertex at an odd distance from the root has exactly two incident edges in *T*, and
    - all paths from *r* to leaves in *T* have even lengths, their odd edges are not in *M* and their even edges are in *M*.
- a forest *F* in *G* is an *alternating forest* with respect to *M*, if
    - its connected components are alternating trees, and
    - every exposed vertex in *G* is a root of an alternating tree in *F*.

Each iteration of the loop starting at line B09 either adds to a tree *T* in *F* (line B10) or finds an augmenting path (line B17) or finds a blossom (line B20). It is easy to see that the running time is $O(|E||V|^2)$.

### *1.3.3 Bipartite matching*

When *G* is bipartite, there are no odd cycles in *G*. In that case, blossoms will never be found and one can simply remove lines B20 – B24 of the algorithm. The algorithm thus reduces to the standard algorithm to construct maximum cardinality matchings in bipartite graphs where we repeatedly search for an augmenting path by a simple graph traversal: this is for instance the case of the Ford-Fulkerson Algorithm.

### *1.3.4 Weighted matching*

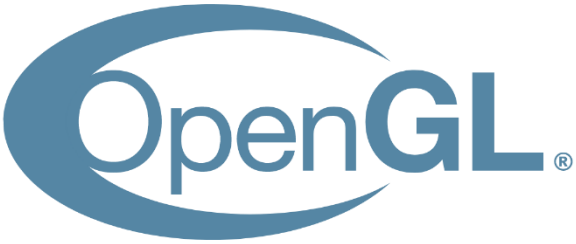
The matching problem can be generalized by assigning weights to edges in *G* and asking for a set *M* that produces a matching of maximum (minimum) total weight: this is the maximum weight matching problem. This problem can be solved by a combinatorial algorithm that uses the unweighted Edmonds's algorithm as a subroutine.

# 2 | Tools & Technologies used for the implementation

*The following tools and technologies have been used in this project for the implementation of the blossom's algorithm in a graphical interface. VS Code was used as a code-editor, C++ was the language used for programming and the OpenGL API was used to design the graphical interface. Here is a brief introduction on each:*

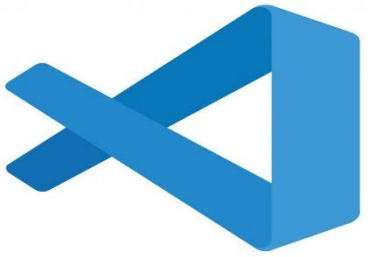
## *2.1 OpenGL*

*OpenGL (Open Graphics Library)* is a cross-language, cross-platform Application Programming Interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a Graphics Processing Unit (GPU), to achieve hardware-accelerated rendering.




## *2.2 Visual Studio Code*

*Visual Studio Code*, also commonly referred to as *VS Code*, is a source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality.



## *2.3 C++*

***C++*** is a general-purpose programming language created by Danish computer scientist *Bjarne Stroustrup* as an extension of the C programming language, or "*C with Classes*". The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM so, it is available on many platforms.

# 3 | Implementation with a Graphical Interface

*To visualize how the algorithm works to find the maximum matching we have used the OpenGL API to take inputs via mouse clicks to create the graph and then keyboard inputs to follow the steps as the algorithm proceeds.*

- We use a single buffer RGBA colour profile.
- The Input window is set to a resolution of 1280 x 720 pixels, positioned at (0,0), i.e., at the top-left corner of the computer screen and titled "Edmond's Blossom Algorithm".
- The Background colour is set to white by *glClearColor (1.0, 1.0, 1.0, 1.0).*
- *gpMouse* is the callback for the *glutMouseFunc()* and manages the user inputs to create and operate on the graph.
- In the main function a file *"Input.txt"* is opened with write permissions to store the vertices and edges for future use.
- In the code snippet give below, we see the main function and the *gpInit()* function which initialises the plot session. The description of the various parameters and instructions have been mentioned above.

```
290  int main(int argc, char *argv[])
291  {
292      fp = fopen("input.txt", "w+");
293      glutInit(&argc, argv);
294      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
295      glutInitWindowSize(WW, WH);
296      glutInitWindowPosition(0, 0);
297      glutCreateWindow("Edmond's Blossom Algorithm");
298      glutMouseFunc(gpMouse);
299      gpInit();
300      glutMainLoop();
301      return 0;
302  }
303
304  void gpInit()
305  {
306      glClearColor(1.0, 1.0, 1.0, 1.0);
307      glColor4f(0.0, 0.0, 0.0, 1.0);
308      glMatrixMode(GL_PROJECTION);
309      glLoadIdentity();
310      gluOrtho2D(0, WW, 0, WH);
311      glClear(GL_COLOR_BUFFER_BIT);
312      glFlush();
313  }
```

# 3.1| Utility functions for plotting points and lines

*Before we actually see how the mouse inputs are used to create vertices and edges, we need to understand certain utility functions which will later be used for drawing points and lines.*

## 3.1.1 Drawing points

*There are 3 types of points: regular, blossom and small points. They are discussed below.*

### 3.1.1.1 Drawing regular points

The *drawpoint()* function takes in two integer parameters *x* and *y*, the respective x and y coordinates of the point where the user clicked on the screen and displays it in the colour as passed into the function as the string parameter *'col'*.

- The 4 global variables [*r, g, b, a*] are respectively set to the values of red, green, blue and alpha that will be taken for a given colour as specified by the string *"col"*. This is done by the multiple if-statement blocks some of which are shown collapsed in the code snippet below.
- The colours are set by *glColor4f (r, g, b, a).*
- The point size is set to "*radius*" which is also globally declared to be equal to 20 pixels, so that a vertex appears as a small circle or a thick point.
- The point is also set to be of smooth nature by *glEnable(GL_POINT_SMOOTH).*
- The actual point is plotted by *glVertex2i(x, y)* placed between *glBegin(GL_POINTS)* and *glEnd().*

```
393  void drawPoint(int x, int y, string col)
394  {
395      if (col == "light")···
402      if (col == "red")···
409      if (col == "black")···
416      if (col == "gray")···
423      if (col == "blue")···
430      if (col == "green")
431      {
432          r = 0.0;
433          g = 1.0;
434          b = 0.0;
435          a = 1.0;
436      }
437      glColor4f(r, g, b, a);
438      glPointSize((double)radius);
439      glEnable(GL_POINT_SMOOTH);
440      glBegin(GL_POINTS);
441      glVertex2i(x, y);
442      glEnd();
443  }
444
```

### 3.1.1.2 Drawing Blossom points

The *drawblossompoint()* function takes in two integer parameters *x* and *y*, the respective x and y coordinates of the vertex which is a part of a blossom which is to be contracted and highlights that vertex with a yellow semi-transparent colour [*glColor4f(1.0, 0.5, 0.0, 0.5)*]. The thickness of the point is determined by the argument *"rad"* which is passed to the function. Usually, this point is made larger than a regular-sized vertex and appears as a glow over a regular-sized vertex. The other properties of the point are similar to what we had in the function *drawpoint()*.

```
444
445  void drawblossompoint(int x, int y, int rad)
446  {
447      glColor4f(1.0, 1.0, 0.0, 0.5);
448      glPointSize((double)rad);
449      glEnable(GL_POINT_SMOOTH);
450      glBegin(GL_POINTS);
451      glVertex2i(x, y);
452      glEnd();
453  }
454
```

### 3.1.1.3 Drawing Small points

The *drawsmallpoint()* function takes in two integer parameters *x* and *y*, the respective x and y coordinates of the vertex which is a father of the *activeVertex* and highlights that vertex with an orange semi-transparent colour [*glColor4f(1.0, 1.0, 0.0, 0.5)*]. The thickness of the point is determined by the argument *"rad"* which is passed to the function. Usually, this point is made smaller than a regular-sized vertex and appears as a dot inside a regular-sized vertex. The other properties of the point are similar to what we had in the function *drawpoint()*.

```
454
455  void drawsmallpoint(int x, int y, int rad)
456  {
457      glColor4f(1.0, 0.5, 0.0, 0.5);
458      glPointSize((double)rad);
459      glEnable(GL_POINT_SMOOTH);
460      glBegin(GL_POINTS);
461      glVertex2i(x, y);
462      glEnd();
463  }
464
```

### 3.1.2 Drawing Lines

The *drawLine()* function takes in four integer parameters *(xa, ya)* and *(xb, yb)*, the x and y coordinates of the initial and final points respectively of the line to be drawn representing and edge between two vertices displays it in the colour as passed into the function as the string parameter *'col'*.

- The 4 global variables [*r, g, b, a*] are respectively set to the values of red, green, blue and alpha that will be taken for a given colour as specified by the string *"col"*. This is done by the multiple if-statement blocks some of which are shown collapsed in the code snippet below.
- The colours are set by *glColor4f (r, g, b, a).*
- The line width is set to be 5 pixels so that the edge appears visibly thick.
- The line is also set to be of smooth nature by *glEnable(GL_LINE_SMOOTH).*
- Within *glBegin(GL_LINES)* and *glEnd()*, the starting and ending points of the line to be plotted are set by *glVertex2i(xa, ya)* and *glVertex2i(xb, yb).*

```
464
465   void drawLine(int xa, int ya, int xb, int yb, string col)
466   {
467 >     if (col == "red")···
474 >     if (col == "black")···
481 >     if (col == "gray")···
488 >     if (col == "blue")···
495       if (col == "green")
496       {
497           r = 0.0;
498           g = 1.0;
499           b = 0.0;
500           a = 1.0;
501       }
502       glColor4f(r, g, b, a);
503       glLineWidth(5);
504       glEnable(GL_LINE_SMOOTH);
505       glBegin(GL_LINES);
506       glVertex2i(xa, ya);
507       glVertex2i(xb, yb);
508       glEnd();
509   }
```

# 3.2 | Mouse function for graph creation

*Let us take a look at some of the important data structures that will be required to create the graph by mouse inputs.*

- The map "*vertices*" is used to store every vertex input from the user and it is mapped to a pair of integers which are the x and y coordinates of the point where the vertex is located on the screen.
- The vector of pairs "*edges*" stores two integers in a pair where each is a vertex and a pair denotes an edge between those two vertices.

```
34    map<int, pair<int, int>> vertices;
35    vector<pair<int, int>> edges;
```

*Some more variables to be observed are:*

- *bool inputphase*: Initially set to *true*. Once the inputs from user are taken, it is set to *false* to avoid any more inputs by stray mouse clicks on the screen while the blossom algorithm is being implemented.
- *int click*: Initialised to zero, this is used to maintain the number of left clicks made by the user.

## 3.2.1 Utility function for an already existing vertex

The function *int existVertex(int xp, int yp)* is used to check if the points *(xp, yp)* passed to the function are the coordinates of an existing vertex already input by the user previously or not. If yes, then the number of that vertex is returned which *(xp, yp)* points to. If no, then -1 is returned and this point *(xp, yp)* is treated as a new point to be plotted in the mouse function.

We traverse the map *"vertices"* and check for every existing vertex whether *(xp, yp)* belongs inside the circle centred at *xcentre* and *ycentre* with radius equal to *"radius"*, where *xcentre* and *ycentre* are the x and y coordinates of the centres of that vertex. If such a point exists inside the circle, that vertex is returned by the function. After the search for all vertices is completed, the control reaches the end of the function only when no vertex could be found to exist at the point *(xp, yp)*. We then return -1, indicating a new vertex is found.

```
315   int existVertex(int xp, int yp)
316   {
317       for (auto q : vertices)
318       {
319           int xcentre = q.second.first;
320           int ycentre = q.second.second;
321           if (((xcentre - xp) * (xcentre - xp) + (ycentre - yp) * (ycentre - yp)) < radius * radius)
322               return q.first;
323       }
324       return -1;
325   }
```

### 3.2.2 The gpMouse() function

The function has 4 parameters: *button, state, x, y*. The *button* parameter specifies the button pressed left or right. The *state* parameter specifies the state of the button whether pressed or released. The parameters *x* and *y* contain the coordinates of the point where the click occurred on the screen.

- When a left-click is encountered, the click value increments and if we are still in the input phase, we proceed to the next steps, else we do nothing.
- We follow a convention that in every odd-numbered click, we set a starting vertex for an edge and in an even-numbered click, we set the ending vertex for that edge.
- In both these cases, we check if the point where the click occurred was an existing vertex. If it was then, we reset the values of x and y to that of the existing vertex and we set the current vertex denoted by 'u' or 'v' to be this existing vertex.
- Otherwise, if it is not an existing vertex, we increment '*Vnum*' as a new vertex is to be added. We set 'u' or 'v' to be '*Vnum*' and we add the pair {x,y} to the map '*vertices*' at '*Vnum*'. Then, we plot this new point (x, y) on the screen in gray colour using *drawpoint()* function.

```
327  void gpMouse(int button, int state, int x, int y)
328  {
329      y = glutGet(GLUT_WINDOW_HEIGHT) - y;
330      if ((button == GLUT_LEFT_BUTTON) && (state == GLUT_DOWN))
331      {
332          click++;
333          if (inputphase)
334          {
335              if (click % 2 != 0)
336              {
337
338                  int val = existVertex(x, y);
339                  if (val == -1)
340                  {
341                      Vnum++;
342                      u = Vnum;
343                      vertices[Vnum] = {x, y};
344                      drawPoint(x, y, "gray");
345                  }
346                  else
347                  {
348                      u = val;
349                      x = vertices[val].first;
350                      y = vertices[val].second;
351                  }
352                  xa = x;
353                  ya = y;
354              }
355              else
```

- In case of odd-numbered clicks, *(xa, ya)* is set to *(x, y)*.
- In case of even-numbered clicks, *(xb, yb)* is set to *(x, y)* and then the edge is drawn from *(xa, ya)* to *(xb, yb)* in black colour using the *drawLine*() function. Also, the vertices are plotted again for neat presentation. Finally, the pair of vertices forming this edge – {u,v} is added to the vector '*edges*'.
- When a right-click is encountered, it marks the end of the input phase and hence the variable *inputphase* is set to false and the *display()* function is called which shall proceed with the execution of the Edmond's Blossom Algorithm for the graph input in the input phase.

```
            else
            {
                int val = existVertex(x, y);
                if (val == -1)
                {
                    Vnum++;
                    v = Vnum;
                    vertices[Vnum] = {x, y};
                    drawPoint(x, y, "gray");
                }
                else
                {
                    x = vertices[val].first;
                    y = vertices[val].second;
                    v = val;
                }
                xb = x;
                yb = y;
                drawLine(xa, ya, xb, yb, "black");
                drawPoint(xa, ya, "gray");
                drawPoint(xb, yb, "gray");
                edges.push_back({u, v});
            }
        }
    }
    if ((button == GLUT_RIGHT_BUTTON) && (state == GLUT_DOWN))
    {
        if (inputphase)
        {
            inputphase = false;
            glFlush();
            display();
        }
    }
    glFlush();
    return;
}
```

### 3.3.1 The display() function

The *display* function performs two types of tasks. Firstly, it sorts the '*edges*' vector for a neat output into the *"input.txt"* file. The first line of *"input.txt"* contains the number of vertices and edges in the graph input. The following lines list all the edges as pairs of vertices in 1-based indexing separated by spaces. Secondly and finally, it calls the *edmondcall()* function.

```
280  void display()
281  {
282      fprintf(fp, "%d %d\n", vertices.size(), edges.size());
283      sort(edges.begin(), edges.end());
284      for (auto x : edges)
285          fprintf(fp, "%d %d\n", x.first, x.second);
286      fclose(fp);
287      edmondcall();
288  }
```

### 3.3.2 The edmondcall() function

The *edmondcall* function opens *"input.txt"* in *read* mode and *"output.txt"* in *write* mode. It scans the number of vertices and edges and stores them in global variables *'V'* and *'E'*. It creates an object of class *Blossom* named *'bm'* and initialises it with values for *'V'* and *'E'*. It then scans the next *E* lines from the input file and passes them to the *addEdge()* function in 0-based indexing. It then calls the *edmondsBlossomAlgorithm()* function to calculate the maximum matching and then prints it to *"output.txt"* using the *printMatching()* function.

```
255  void edmondcall()
256  {
257      fpin = fopen("input.txt", "r+");
258      fpout = fopen("output.txt", "w+");
259      int u, v;
260      int V, E;
261      rewind(fpin);
262      fscanf(fpin, "%d %d", &V, &E);
263      Blossom bm(V, E);
264      while (E--)
265      {
266          fscanf(fpin, "%d %d", &u, &v);
267          bm.addEdge(u - 1, v - 1);
268      }
269      int res = bm.edmondsBlossomAlgorithm();
270      bm.printMatching(res);
271      fclose(fpin);
272      fclose(fpout);
273      cout << "Press Enter thrice to quit\n";
274      flag = getc(stdin);flag = getc(stdin);flag = getc(stdin);
275      exit(0);
276  }
```

## 3.4 | The Class Blossom

*Some data structures and variables to understand before we explore the functions in this class are mentioned as follows:*

- *struct StructEdge*: This struct stores all pairs of vertices for all edges. The integer variable *vertex* stores the number of the vertex and the pointer *next* stores the next sibling of the vertex. *Edge* is a type-defined pointer for *StructEdge*.

```
struct StructEdge
{
    int vertex;
    StructEdge *next;
};

typedef StructEdge *Edge;
```

- A *pool* array is created of type *StructEdge* to store all the vertices from the input file. Pointers of *StructEdge* type are created: "*top*" which points to the top of the input stack in *pool*; and *adj[M]*, an array which stores the address of the edge of last child of the respective indices in *adj[]*.
- Integers *V*, *E* store the number of vertices and edges respectively.
- Integers *q_front* and *q_rear* store the index of the front and rear of the *bfs_queue* array which is used for BFS Traversal in finding augmenting paths.
- Integer array *match[]* stores the matching for the edges. If a matching exists between *u* and *v*, *match[u]* will be equal to *v*, and *match[v]* will be equal to *u*. It is initialised to -1 and remains so for vertices which are unmatched.
- Integer array *father[]* stores the parent vertex from which we reached the current vertex during the BFS traversal.
- Integer array *base[]* stores the base for every vertex. Initially, it is set to its index value for every vertex, but in case a blossom is encountered, all vertices in the blossom will have their base values equal to one vertex in the blossom at which it shall be assumed to be contracted.
- Boolean array *ed[][]* is an adjacency matrix used to avoid duplicate entries in graph. Boolean arrays *inq[]* and *inb[]* are used to check if a particular vertex is in the BFS queue or if it is in a blossom respectively.

```
class Blossom
{
    StructEdge pool[M * M * 2];
    Edge top = pool, adj[M];
    int V, E, q_front, q_rear;
    int match[M], bfs_queue[M], father[M], base[M];
    bool inq[M], inb[M], ed[M][M];

public:
    Blossom(int V, int E) : V(V), E(E) {}
```

### 3.4.1 The initializer list

The initializer list in blossom class initialises the number of vertices *V* and number of edges *E* to the integers *V* and *E* passed to the constructor.

```
public:
    Blossom(int V, int E) : V(V), E(E) {}
```

### 3.4.2 The addEdge() function

This function takes in two vertices *u* and *v* in 0-based indexing. The if-condition checks for duplicates and self-loops in inputs. If no such condition arises the vertices *v* and *u* are added to the *pool* and the respective next values for each are assigned the other's *adj* array value, i.e., for the *StructEdge* element for vertex *v*, next stores the pointer for *adj[u]* and vice-versa. Lastly, the vertices *u* and *v* are added to the adjacency matrix *ed[][]*.

```
void addEdge(int u, int v)
{
    if (!ed[u][v] && u != v)
    {
        top->vertex = v;
        top->next = adj[u];
        adj[u] = top++;

        top->vertex = u;
        top->next = adj[v];
        adj[v] = top++;

        ed[u][v] = ed[v][u] = true;
    }
}
```

### 3.4.3 The pausefunc() function

This function is paired with the *updateGraph()* function to display the updated graph and then halt the program execution till the user allows it to proceed to observe every intermediate step leading to the maximum matching.

```
void pausefunc()
{
        flag = getc(stdin);
}
```

### 3.4.4 The updateGraph() function

- First, the screen is cleared and the background colour is set to white.
- Next for all edges, we check whether it is a matched edge or not. We represent matched edge with red colour and unmatched edges with black. This is done with the help of *drawLine()* function.
- Next for every vertex, we check for the following:
  - If it is the *root* of a blossom, it is highlighted with a *large yellow* circle.
  - If it is not a root but a part of the blossom it is highlighted in *yellow* with a *large but smaller* than the previous circle. Both these tasks are accomplished by *drawblossompoint()* function.
  - If the vertex is the *originalVertex* i.e., the vertex for which we are trying to improve the matching, then it is displayed in *green* colour.
  - If the vertex is the *startVertex* i.e., the vertex at the front of the BFS queue, whose children will be accessed, then it is displayed in *blue* colour.
  - If the vertex is the *activeVertex* i.e., the current vertex we are exploring, it is then displayed in *red* colour.
  - Otherwise, if it is none of the above, it is a regular vertex and we denote it by *gray* colour. All these steps from *originalVertex* to regular vertex are accomplished by *drawpoint()* function.
- Lastly, the *father* of the *activeVertex* is denoted by a *smaller orange* point drawn using *drawsmallpoint()* function.

```
void updateGraph()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    for (auto checkedge : edges)
    {
        int v1 = checkedge.first;
        int v2 = checkedge.second;
        xa = vertices[v1].first;
        ya = vertices[v1].second;
        xb = vertices[v2].first;
        yb = vertices[v2].second;
        if (match[v1 - 1] == (v2 - 1) && match[v2 - 1] == (v1 - 1))
            drawLine(xa, ya, xb, yb, "red");
        else
            drawLine(xa, ya, xb, yb, "black");
    }
    for (auto checkv : vertices)
    {
        int ve = checkv.first;
        if (bmnode != -1 && bmnode == (ve - 1))
            drawblossompoint(vertices[ve].first, vertices[ve].second, radius + 20);
        if (inb[ve - 1])
            drawblossompoint(vertices[ve].first, vertices[ve].second, radius + 10);
        if (originalVertex != -1 && (ve - 1) == originalVertex)
            drawPoint(vertices[ve].first, vertices[ve].second, "green");
        if (startVertex != -1 && (ve - 1) == startVertex)
            drawPoint(vertices[ve].first, vertices[ve].second, "blue");
        if (activeVertex != -1 && (ve - 1) == activeVertex)
            drawPoint(vertices[ve].first, vertices[ve].second, "red");
        if (((ve - 1) != originalVertex) && ((ve - 1) != startVertex) && ((ve - 1) != activeVertex))
            drawPoint(vertices[ve].first, vertices[ve].second, "gray");
    }
    if (activeVertex != -1 && father[activeVertex] != -1)
        drawsmallpoint(vertices[father[activeVertex] + 1].first, vertices[father[activeVertex] + 1].second, radius / 2);
    glFlush();
}
```

### 3.4.5 Finding and displaying maximum matching

The *edmondsBlossomAlgorithm* function calculates the maximum number of matched edges and stores it in *match_counts* and returns this value.

- It initialises the *match[]* array to have all values -1.
- For every vertex which is unmatched, we find an augmenting path for that vertex *u* and we augment the path from *u* to *v*, where *v* is the vertex returned by *find_augmenting_path(u)*. If the path could be augmented, the *match_counts* is incremented by 1.
- Finally, after maximum matching has been found all variables *activeVertex*, *startVertex*, *originalVertex*, *bmnode* and the *inb[]* array is reset to their initial values so that only the final matching is displayed on the screen.

The *printMatching* function takes in the maximum number of edges matched and prints it to the output file, followed by all the matched edges in 1-based indexing. For all vertices, we check if the value in *match[]* array for that vertex exists and is greater than that vertex, to avoid printing same edge twice. If it is so, we print the vertex *i* and the *match[i]*.

```
int edmondsBlossomAlgorithm()
{
    int match_counts = 0;
    memset(match, -1, sizeof(match));
    for (int u = 0; u < V; u++)
        if (match[u] == -1)
            match_counts += augment_path(u, find_augmenting_path(u));

    activeVertex = originalVertex = startVertex = -1, bmnode = -1;
    memset(inb, 0, sizeof(inb));
    updateGraph();
    return match_counts;
}
void printMatching(int maxMatch)
{
    rewind(fpout);
    fprintf(fpout, "%d\n", maxMatch);
    for (int i = 0; i < V; i++)
        if (i < match[i])
            fprintf(fpout, "%d %d\n", i + 1, match[i] + 1);
}
```

### 3.4.6 Finding Augmenting Paths

*The find_augmenting_path function accepts a vertex s and returns the last vertex in the augmenting path beginning from s. It returns -1 if no augmenting path was found.*

- Initially the *inq[]* and *father[]* arrays are initialised to *false* and -1 respectively and *base* of every vertex is set to itself.
- We begin by inserting *s* into *bfs_queue* and while the queue is not empty, we perform BFS on the graph.
- We traverse all children *v* of *u*, where *u* is the *q_front*. Only if the *base* of both *u* and *v* are different and there is no matching existing between *u* and *v*, we proceed.
- If *v* becomes equal to *s* or a matching exists for *v* with a *father* present, we have found an *odd-length cycle* and we *contract the blossom*.
- Otherwise, if *v* doesn't have a *father*, we set *father* of *v* equal to *u*.
- If *v* doesn't have a matching we return this value to *augment_path()* function as we have found an augmenting path.
- If matching exists for *v*, but the matched vertex is absent in current BFS, we add it to the *rear* of *bfs_queue*.

```
int find_augmenting_path(int s)
{
    originalVertex = s;startVertex = -1;activeVertex = -1;
    updateGraph(); pausefunc();
    memset(inq, 0, sizeof(inq));
    memset(father, -1, sizeof(father));
    for (int i = 0; i < V; i++)
        base[i] = i;
    inq[bfs_queue[q_front = q_rear = 0] = s] = true;
    while (q_front <= q_rear)
    {
        int u = bfs_queue[q_front++];
        startVertex = u; activeVertex = -1;
        updateGraph(); pausefunc();
        for (Edge e = adj[u]; e; e = e->next)
        {
            int v = e->vertex;
            activeVertex = v;
            updateGraph(); pausefunc();
            if (base[u] != base[v] && match[u] != v)
                if ((v == s) || (match[v] != -1 && father[match[v]] != -1))
                {
                    blossom_contraction(s, u, v);
                    updateGraph(); pausefunc();
                }
                else if (father[v] == -1)
                {
                    father[v] = u;
                    updateGraph(); pausefunc();
                    if (match[v] == -1)
                        return v;
                    else if (!inq[match[v]])
                        inq[bfs_queue[++q_rear] = match[v]] = true;
                }
        }
    }
```

### 3.4.7 Augmenting a path

The *augment_path* function takes in two parameters *s* and *t* where *t* is the exposed vertex returned by the *find_augmenting_path* function. If there was an augmenting path found, i.e., *t* was not -1, *augment_path()* returns 1 as the matching will be improved by 1.

- The algorithm begins from the exposed vertex *u=t* finds its *father v* and makes a matching between *u* and *v* and then *u* moves to the previous matching of *v*.
- At any time if there ceases to be a matching for *v*, then *u* will become -1 and hence loop will terminate thus having created the maximum matching for that augmenting path starting from *s* and ending at *t*.

```
int augment_path(int s, int t)
{
    int u = t, v, w;
    while (u != -1)
    {
        v = father[u];
        w = match[v];
        match[v] = u;
        match[u] = v;
        u = w;
        updateGraph(); pausefunc();
    }
    return t != -1;
}
```

### 3.4.8 Finding lowest common ancestor for blossoms

This function is used to find the lowest common ancestor for two vertices u and v in a blossom with respect to a root vertex. Firstly, we find a path from u to the root and then we start from v and return the first vertex that was present in the previous path.

```
int LCA(int root, int u, int v)
{
    static bool inp[M];
    memset(inp, 0, sizeof(inp));
    while (1)
    {
        inp[u = base[u]] = true;
        if (u == root)
            break;
        u = father[match[u]];
    }
    while (1)
    {
        if (inp[v = base[v]])
            return v;
        else
            v = father[match[v]];
    }
}
```

### 3.4.9 Marking and Contracting Blossoms

The *mark_blossom* function is used to mark and add to the blossom all vertices from vertex *u* to the vertex *lca* which are accepted as parameters.

The *blossom_contraction* function finds the *LCA* of *u* and *v* with root *s*. Initialises *inb[]* array to all *false* values, then *marks* all vertices in blossom from *u* to *lca* and then from *v* to *lca* using *mark_blossom()*. Finally, for all vertices in the blossom, it changes their *base* to the *lca* value found and adds all vertices to the *rear* of *bfs_queue* if not previously present. This is done so that, all vertices in the blossom can be treated to be contracted to the *LCA* vertex and when we try to find an augmenting path in the graph with a *contracted blossom*, we treat every matching to be created with a blossom vertex to be supposedly with the *LCA* vertex.

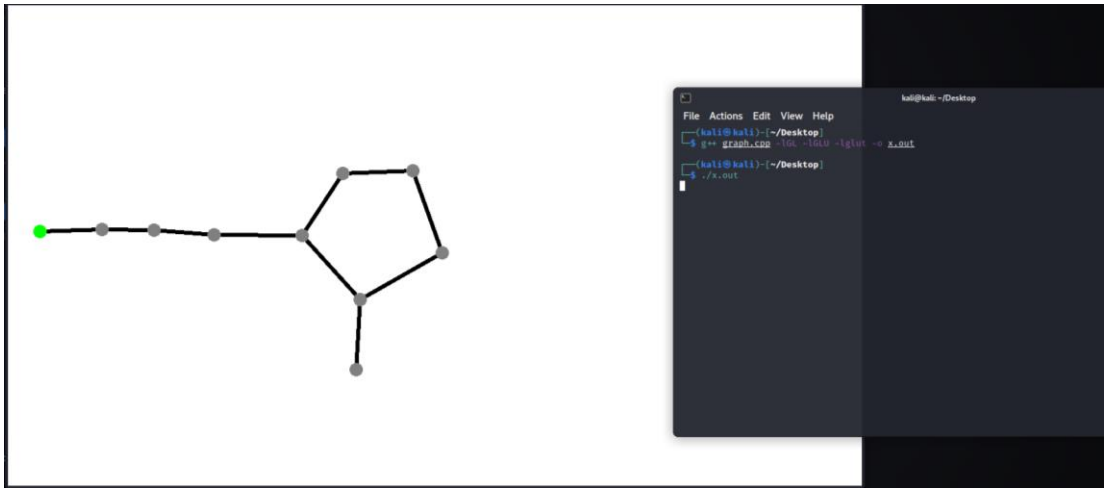
```
void mark_blossom(int lca, int u)
{
    while (base[u] != lca)
    {
        int v = match[u];
        inb[base[u]] = inb[base[v]] = true;
        u = father[v];
        if (base[u] != lca)
            father[u] = v;
        updateGraph(); pausefunc();
    }
}

void blossom_contraction(int s, int u, int v)
{
    int lca = LCA(s, u, v);
    bmnode = lca;
    updateGraph(); pausefunc();
    memset(inb, 0, sizeof(inb));
    mark_blossom(lca, u);
    mark_blossom(lca, v);
    if (base[u] != lca)
        father[u] = v;
    if (base[v] != lca)
        father[v] = u;
    for (int u = 0; u < V; u++)
        if (inb[base[u]])
        {
            base[u] = lca;
            if (!inq[u])
                inq[bfs_queue[++q_rear] = u] = true;
        }
}
```

# 4 | Executing the code

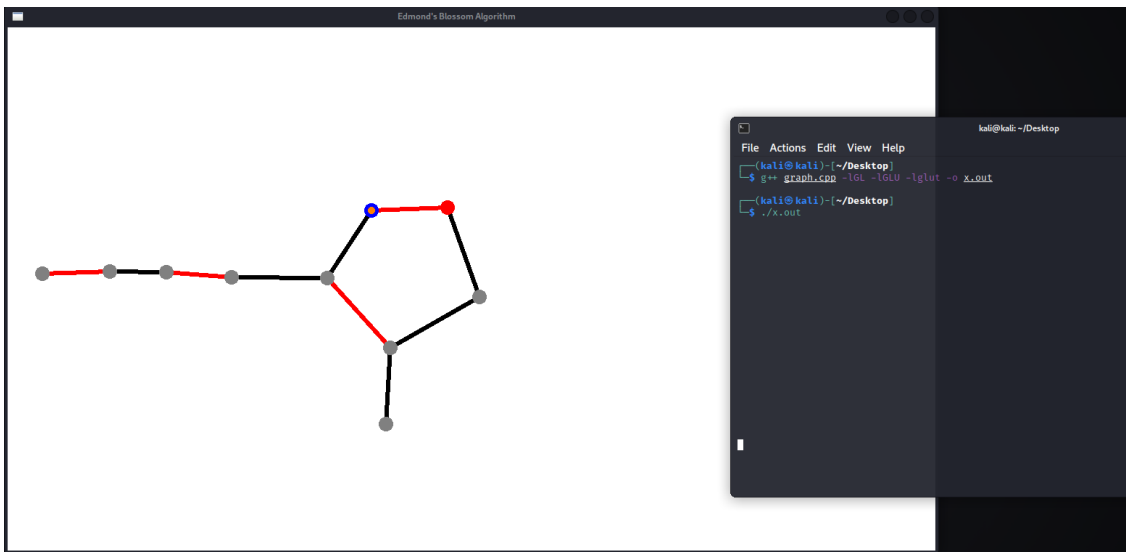
*The code works in both Linux and Windows environments but the system must have OpenGL and GLUT libraries installed in it. For this sample run, we will be operating in a Linux environment.*

- First, we compile the code with the following headers in the terminal:
  *g++ graph.cpp -lGL -lGLU -lglut -o x.out*

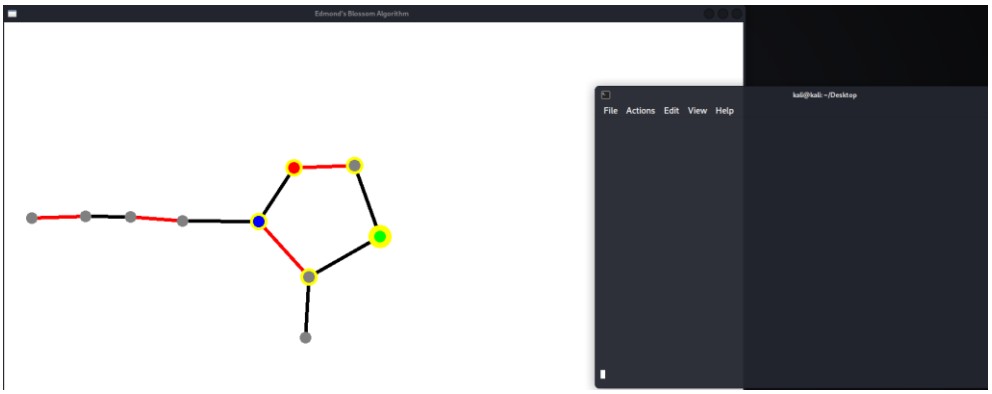
- Now run the binary file *x.out* by typing *"./x.out*
- Plot the graph in the window by clicking with left-mouse button and once all inputs are done, right-click the mouse once to terminate the input phase. To make an edge the first left-click sets the starting vertex for edge and the next left-click completes the edge by setting the other vertex for that edge.



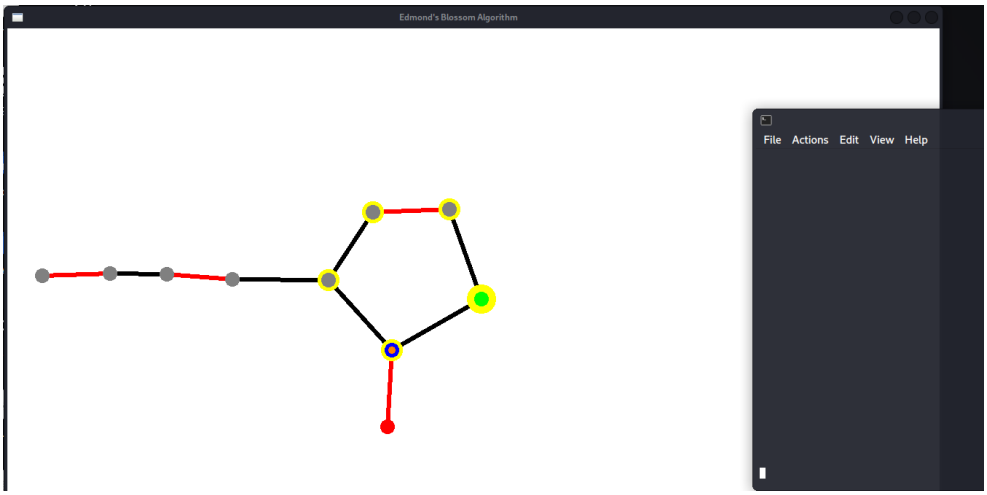

- Now move over to the terminal and keep pressing "Enter" to proceed with the algorithm. As the algorithm proceeds, we see the current vertex denoted by red finding a father denoted by an orange dot and then forming matchings denoted by red lines.



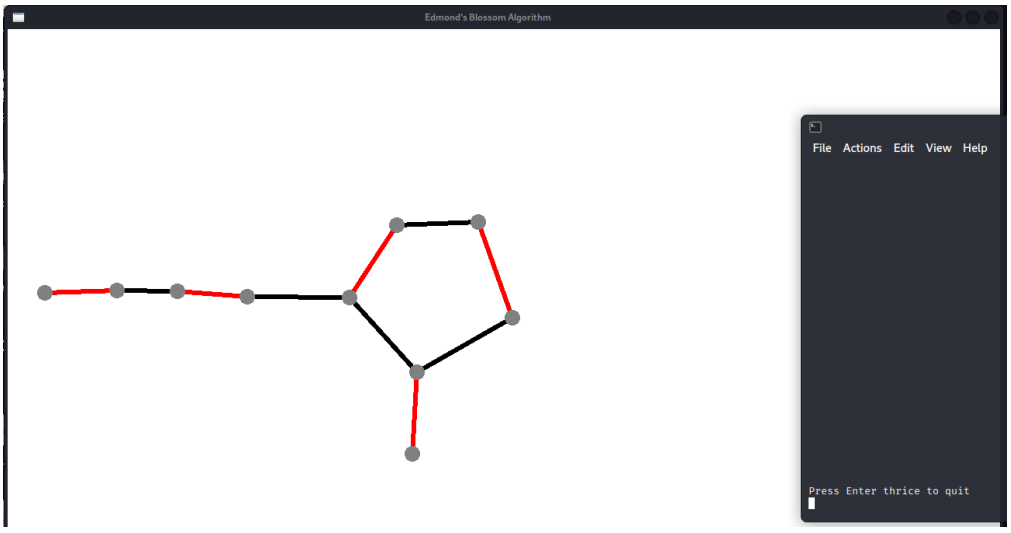
- The algorithm continues and finds a blossom some steps later. Now, it finds the LCA of the red and blue vertex with respect to green root vertex and also marks the blossom vertices. Thus, the blossom has now been contracted.




- The algorithm finds an external node to the blossom and the augmented path is adjusted with respect to that.




- Finally, the augmented path inside the blossom is completed and the maximum matching is obtained.



# 5 | A simple implementation in a real-life problem

## Problem Statement

*Suppose we are given a large collection of people and we are asked to match them into the pairs such that people who are known to each other are preferred to form a pair between them. We have to find the maximum matching number of pairs possible. Also, it is to be noted that every person can belong only to a single pair.*

## Solution approach

Let us assume each person to be a vertex in a graph and the acquaintances to be an edge in a graph between the vertices representing those people. Thus, we create an unweighted, un-directed graph $G = (V, E)$ on which we can apply the Blossom Algorithm to find the maximum matching. We can map each person to a vertex and once the matching is maximised, we can again map the results back to the names of the people and display which are the pairs which will result in maximum matching. We use blossom algorithm for this because it finds the maximum matching in general graphs in polynomial-time. In addition to finding maximum matching, at last we form the unmatched vertices amongst themselves in pairs in random fashion as there is no edge between any of them. Thus, the problem is solved.

# 6 | Implementation in the Battle of Britain (1940)

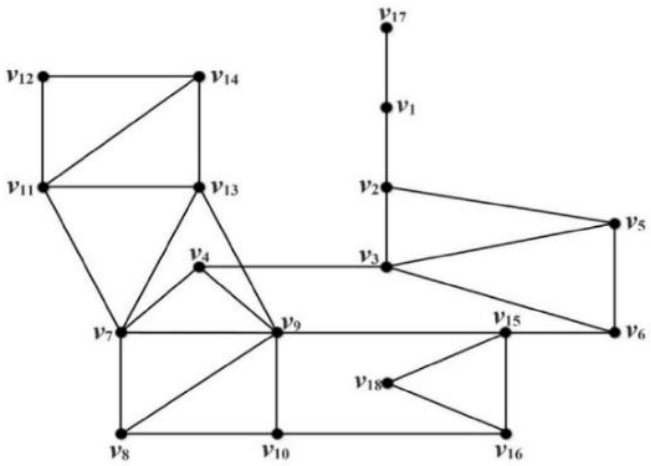
In the Battle of Britain (1940), The Royal Air Force provided several aircrafts to thwart the German air force attack. The aircrafts provided by the Royal Air Force could be flown only by two people as a pilot and co-pilot. To get the best results in the mission to thwart German air force attack, the Royal Air Force brought some of the best pilots from all regions in Britain. However, there was a problem to solve by the Royal Air Force: among all the pilots that were called, some of them could not fly together in one aircraft because there were differences in terms of language, the ability of flying techniques, as well as in terms of flying experience. Suppose the Royal Air Force obtained 18 best pilots from all regions in Britain that were ready to fly the aircraft. Based on the problem, the Royal Air Force wished to pair the 18 pilots where each pair consisted of two pilots in such a way that the number of all the possible pairs is the highest number. In order to solve such problem, the Royal Air Force conducted a series of tests to the 18 pilots including language test, flying technique test, and psychological test to determine the suitability of the 18 pilots to fly together in one aircraft as a pilot and co-pilot.

## *Table of pairing between each aircraft pilot*

| The pilot's number (i=1,2, 3,..,18) | The result of pairing between each aircraft pilot |
|---|---|
| 1st Pilot | Pilot: 2nd and 17th |
| 2nd Pilot | Pilot: 1st, 3rd, and 5th |
| 3rd Pilot | Pilot: 2nd, 4th, 5th, and 6th |
| 4th Pilot | Pilot: 3rd, 7th, and 9th |
| 5th Pilot | Pilot: 2nd, 3rd, and 6th |
| 6th Pilot | Pilot: 3rd, 5th, and 15th |
| 7th Pilot | Pilot: 4th, 8th, 9th, 11th, and 13th |
| 8th Pilot | Pilot: 7th, 9th, and 10th |
| 9th Pilot | Pilot: 4th, 7th, 8th, 10th, 13th, and 15th |
| 10th Pilot | Pilot: 8th, 9th, and 16th |
| 11th Pilot | Pilot: 7th, 12th, 13th, and 14th |
| 12th Pilot | Pilot: 11th and 14th |
| 13th Pilot | Pilot: 7th, 9th, 11th, and 14th |
| 14th Pilot | Pilot: 11th, 12th, and 13th |
| 15th Pilot | Pilot: 6th, 9th, 16th, and 18th |
| 16th Pilot | Pilot: 10th, 15th, and 18th |
| 17th Pilot | Pilot 1th |
| 18th Pilot | Pilot: 15th and 16th |

*By transforming the problem into the form of graph, where the pilots are represented as a vertex and the corresponding relationship between each pilot is represented as an edge, the following graph is obtained:*

$G_2$:

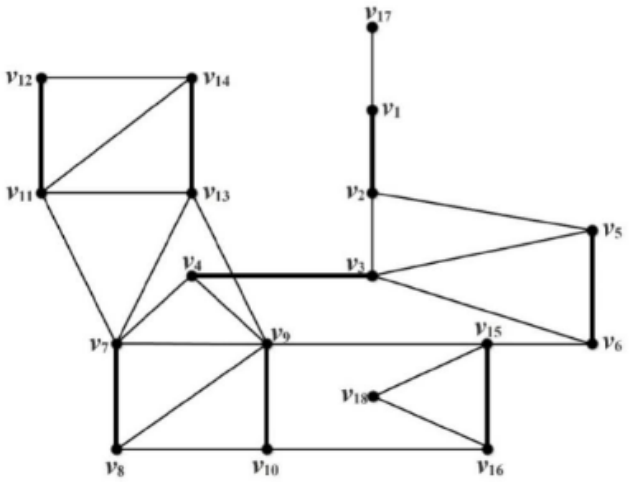



By following the steps from Edmonds' cardinality matching algorithm, we obtain maximum matching on graph $G_2$ as follows:

Step 1 (initialization):

By doing the process of matching initialization to graph $G_2$, we obtain graph $(G_2, M)$ with $M = \{v_1v_2, v_3v_4, v_5v_6, v_7v_8, v_9v_{10}, v_{11}v_{12}, v_{13}v_{14}, v_{15}v_{16}\}$ as follows:

$(G_2, M)$:

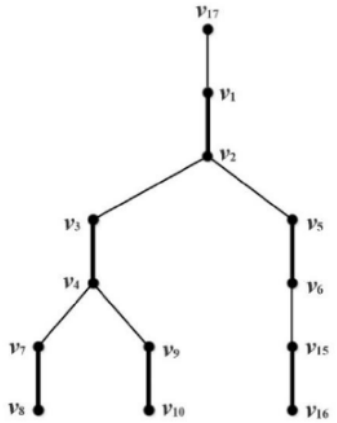



Since there are still two exposed vertices left that are $v_{17}$ and $v_{18}$, then go to step 2.3

Step 2.3 :

Vertex $v_{17}$ is selected as the root, by using BFS we obtain an alternating tree $T_1$ that corresponds to graph $(G_2, M)$ as follows:

Tree $T_1$:

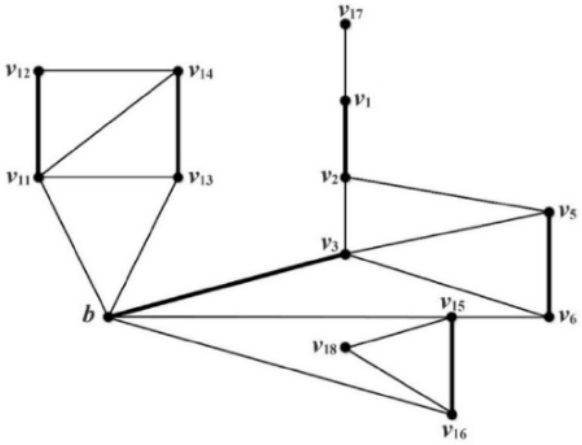



Since blossom $B = \{v_4, v_8, v_9, v_{10}\}$ has been encountered, then go to step 4.

Step 4:

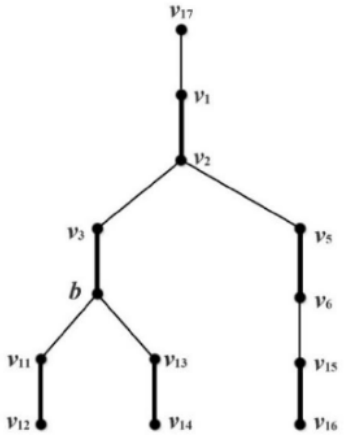
Shrink blossom $B = \{v_4, v_8, v_9, v_{10}\}$ to obtain a graph as follows:

$(G_2', M_1)$:




By continuing the constructing process of alternating tree that corresponds to graph $(G_2', M_1)$, we obtain:

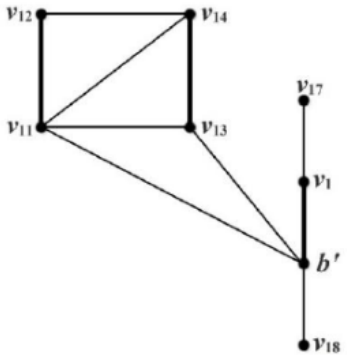
Tree $T_2$:



Since blossom $B' = \{b, v_2, v_3, v_5, v_6, v_{15}, v_{16}\}$ has been encountered, then go to Step 4:

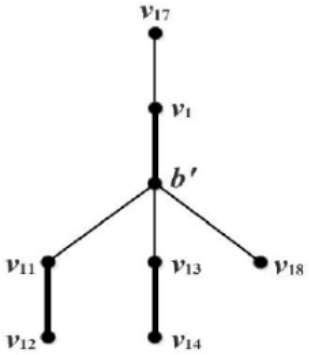
Shrink blossom $B' = \{b, v_2, v_3, v_5, v_6, v_{15}, v_{16}\}$ to obtain a graph as follows:

$(G_2'', M_2)$:




By continuing the constructing process of alternating tree that corresponds to graph $(G_2'', M_2)$, we obtain:

Tree $T_3$:




Since an augmenting path $P'': v_{18} - b' - v_1 - v_{17}$ that contains *pseudovertex* $b'$ has been obtained, then go to step 3.1

Step 3.1:

By unshrinking the pseudovertex $b'$, we obtain augmenting path $P': v_{18} - v_{15} - v_{16} - b - v_3 - v_2 - v_1 - v_{17}$ that contains pseudovertex $b$, then go to step 3.1

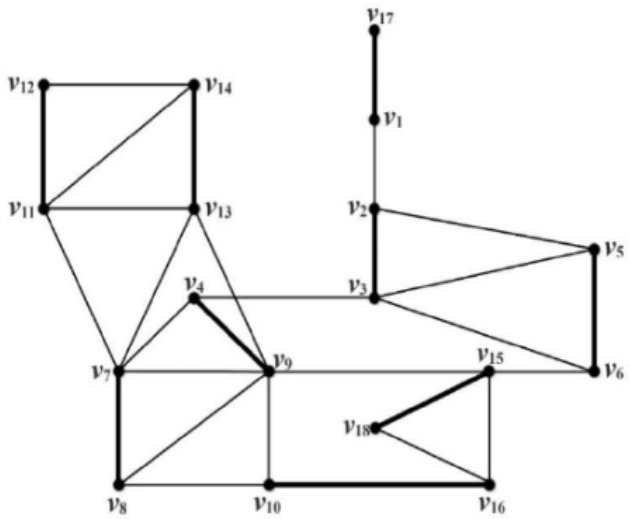
Step 3.1:

By usnhrinking the pseudovertex $b$, we obtain augmenting path $P: v_{18} - v_{15} - v_{16} - v_{10} - v_9 - v_4 - v_3 - v_2 - v_1 - v_{17}$ which does not contain any pseudovertex , then go to step 3.2

Step 3.2:

Augment the initial matching $M$ using augmenting path $P$, we obtain a new matching in $(G_2, M)$ as follows:

$(G_2, M')$:




Since all vertices in $(G_2, M')$ are incident with a matching edge, then go to step 5.

Step 5:

Matching $M'$ in graph $G_2$ is maximum, therefore the finding process is discontinued.

From the steps that have been done, we obtain a maximum matching $M'$ in $G_2$ with cardinality of matching $M'$ that is $|M'| = 9$. Matching $M'$ that has been obtained is the solution of the problem of pairing the 18 pilots as described above with details as follows:

1. Pilot number 1 pairs up with pilot number 17
2. Pilot number 2 pairs up with pilot number 3
3. Pilot number 4 pairs up with pilot number 9
4. Pilot number 5 pairs up with pilot number 6
5. Pilot number 7 pairs up with pilot number 8
6. Pilot number 10 pairs up with pilot number 16
7. Pilot number 11 pairs up with pilot number 12
8. Pilot number 13 pairs up with pilot number 14
9. Pilot number 15 pairs up with pilot number 18

And the maximum number of aircrafts that can be flown is as many as 9 aircrafts.

# 7 | Conclusion

Matching $M$ in graph $G$ is a subset of $E(G)$ edge set where there are no two edges in Matching $M$ that meet each other at the same vertex in graph $G$. Generally, Edmonds' maximum matching algorithm is divided into three steps where the first step is the process of matching initialization. The second step is shrinking and unshrinking of blossom $B_i$. The third step is to execute augmenting- $M$ if the augmenting $P$ path has been found. Edmonds' blossom algorithm can be applied to find for maximum matching especially towards non-bipartite graph. The process of matching initialization is the effort to find the first matching to fasten the finding of maximum matching. Alternating $T$ tree is a $T$ graph concept that is used to find an augmenting path to obtain a new matching that has bigger cardinality than the first matching from the process of matching initialization.

# 8 | References

1. *Wikipedia contributors. "Blossom algorithm." Wikipedia, The Free Encyclopedia, 3 Mar. 2016. Web. 5 Jun. 2016.*

2. *Edmonds, Jack (1991), "A glimpse of heaven", in J.K. Lenstra; A.H.G. Rinnooy Kan; A. Schrijver (eds.), History of Mathematical Programming --- A Collection of Personal Reminiscences, CWI, Amsterdam and North-Holland, Amsterdam, pp. 32–54*

3. *Edmonds, Jack (1965). "Paths, trees, and flowers". Can. J. Math. 17: 449–467.*

4. *Micali, Silvio; Vazirani, Vijay (1980). An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs. 21st Annual Symposium on Foundations of Computer Science. IEEE Computer Society Press, New York. pp. 17–27.*

5. *Edmonds, Jack (1965). "Maximum matching and a polyhedron with 0,1-vertices". Journal of Research of the National Bureau of Standards Section B. 69: 125–130.*

6. *Schrijver, Alexander (2003). Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics. Berlin Heidelberg: Springer-Verlag.*

7. *Lovász, László; Plummer, Michael (1986). Matching Theory. Akadémiai Kiadó.*

8. *Karp, Richard, "Edmonds's Non-Bipartite Matching Algorithm", Course Notes. U. C. Berkeley (PDF).*

9. *Tarjan, Robert, "Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching", Course Notes, Department of Computer Science, Princeton University (PDF).*

10. *Kenyon, Claire; Lovász, László, "Algorithmic Discrete Mathematics", Technical Report CS-TR-251-90, Department of Computer Science, Princeton University.*

11. *Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm", Mathematical Programming Computation.*