# Edmonds's Blossom Algorithm

A Maximum Matching Algorithm for any Graph

Matthew Kusner

## Problem Description

Consider any graph G, with vertex set V and edge set E. We define a matching as any set of disjoint edges M in G, disjoint in the sense that no two edges share a vertex.
A *maximal matching* is defined as a matching in which no edge in G can be added to the matching.
A *maximum matching* is defined as a matching for which M has the greatest possible size for a particular G.

Why might we want to know about the maximum matching for a particular graph? It turns out that representing a problem as a graph and finding a maximum matching in said graph can be used to solve many interesting problems such as:
- Matching students to classes
- Matching couples at a dance
- Matching double bonds in aromatic compounds. Aromatic compounds consist of carbon rings of alternating single and double bonds. We can represent aromatic compounds as graphs with carbon atoms as vertices, single bonds as unmatched edges, and double bonds as matched edges. Trinajsti■ et al. defines a Kekulé structure as a portion of an aromatic compound in which a perfect matching of carbon atoms places the location of double bonds in the compound.

**History of Matching Algorithms**
Harold Kuhn developed the Hungarian algorithm in 1955 for finding a maximum matching in any bipartite graph (Kuhn, 1955). The algorithm works primarily by attempting to build off of the current matching M to find a larger one by finding an M-augmenting path. Kuhn's main theorem is that a graph matching is maximum if no augmenting path exists in the graph. For bipartite graphs the algorithm is quite fast. However, on non-bipartite graphs the algorithm runs into problems finding existing augmenting paths.

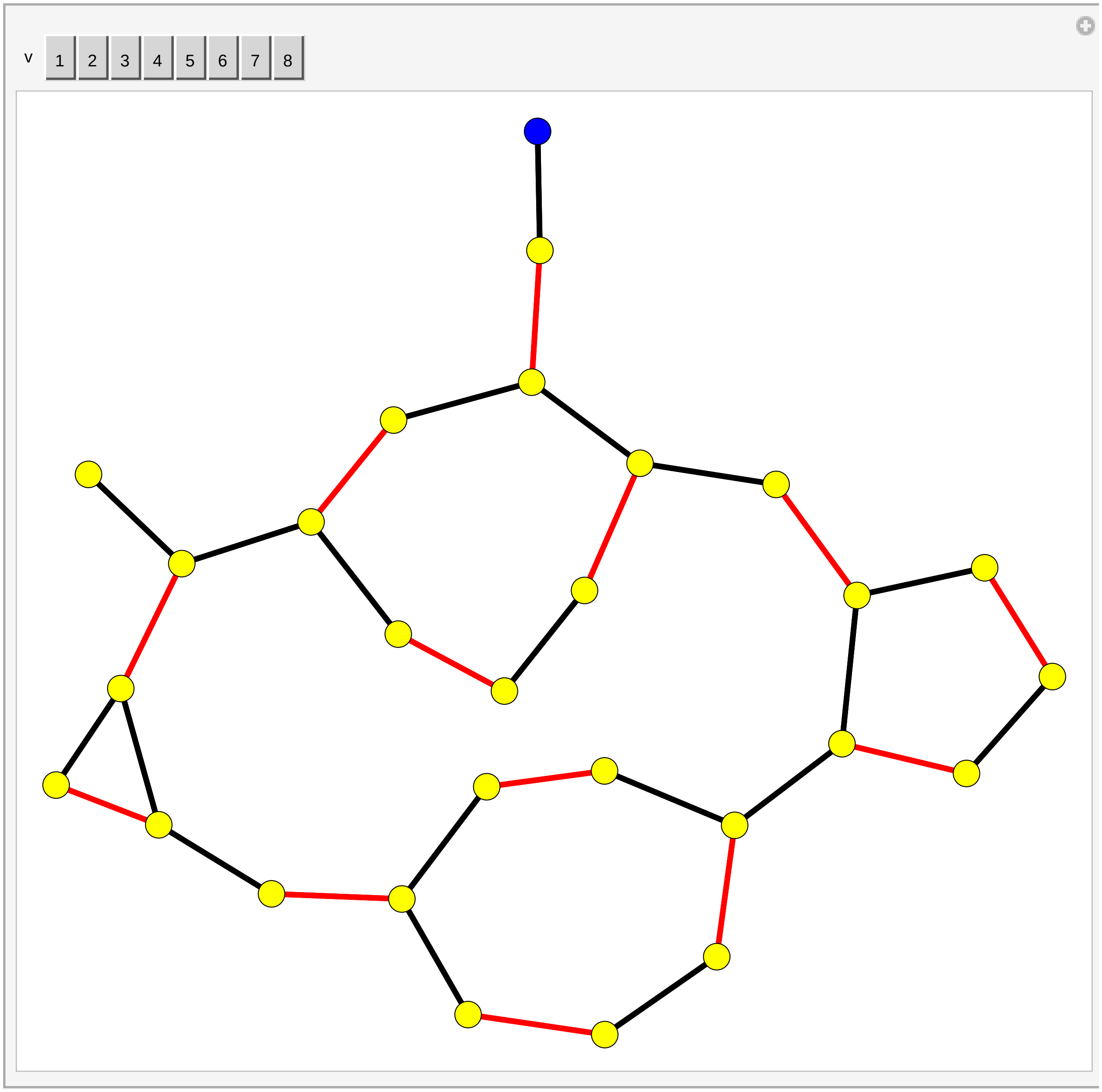
In 1965, Jack Edmonds published the Blossom algorithm for finding a maximum matching in any graph (Edmonds, 1965). In effect, the algorithm deals with a specific insufficiency in the Hungarian algorithm. The main idea is that by shrinking particular cycles in a graph we reveal, in effect, augmenting paths to the Hungarian algorithm.

## Why not Hungarian?

Now, we're told the Hungarian algorithm won't work on every graph, but why is this? Let's try out the Hungarian algorithm on this graph and see how the algorithm does.

```
In[20]:= Module[{gAdjTotal, graph, hungarianFound},
          gAdjTotal = {{2}, {1, 3}, {2, 4, 6}, {3, 5}, {4, 8, 10}, {3, 7, 14},
            {6, 9}, {5, 9}, {7, 8}, {5, 11, 28}, {10, 12, 13}, {11, 13}, {11, 12, 16},
            {6, 15}, {14, 18, 20}, {13, 17}, {16, 22, 24}, {15, 19, 26}, {18, 21}, {15, 21},
            {19, 20}, {17, 23}, {22, 26}, {17, 25}, {24, 27}, {18, 23, 27}, {25, 26}, {10}};
          graph = FromAdjacencyLists[gAdjTotal];
          matching = {{2, 3}, {4, 5}, {6, 7}, {8, 9}, {10, 11}, {12, 13},
            {14, 15}, {16, 17}, {18, 19}, {20, 21}, {22, 23}, {24, 25}, {26, 27}};
          hungarianFound = {1, 2, 3, {4, 6}, {5, 7}, {8, 9, 5, 10}, 11, {12, 13}};
          Manipulate[
           GraphPlot[graph, Method → "SpringEmbedding",
            VertexLabeling → True, VertexRenderingFunction →
             ({If[MemberQ[Flatten[hungarianFound[[1 ;; v]]], #2], Blue, Yellow],
                PointSize[.02], EdgeForm[Black], Disk[#1, .1]} &), EdgeRenderingFunction →
             ({If[MemberQ[matching, #2] || MemberQ[matching, Reverse[#2]], Red, Black],
                AbsoluteThickness[3], Line[#1]} &), PlotRangePadding → 0, ImageSize → 550],
           {v, 1, Length[hungarianFound], 1}, ControlType → Setter, TrackedSymbols → True,
           SaveDefinitions → True]]
```

Out[20]=

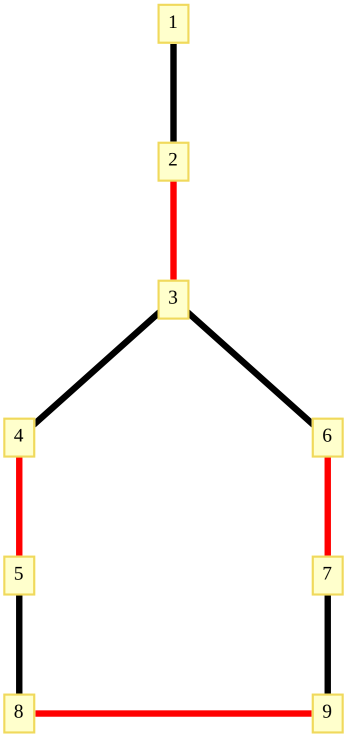


What went wrong? We seem to be getting stuck in cycles. Let's try to quantify the problem.

## Terminology

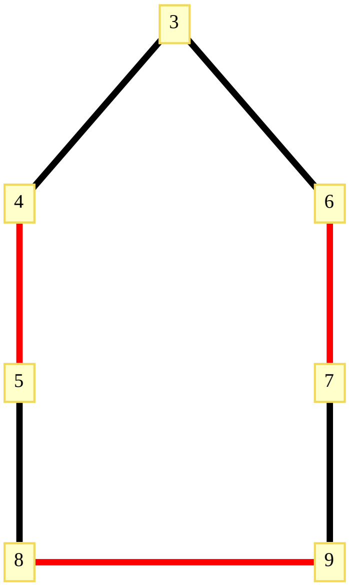
### ■ Flower

A flower is composed of two primary sub-objects: a **blossom**, and a **stem**.



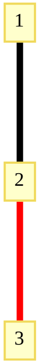
## Blossom

Define a **blossom** B as any odd length cycle that alternates on the current matching. Further, define the base of the blossom b as the only vertex on the blossom that is not covered *by the matching on the blossom*. It is also the only vertex in the blossom that is able to be covered by a matching outside of the blossom. In the blossom below, vertex 3 is the blossom base.



### Stem

Define a **stem** S as an even length alternating path on the current matching, starting at a vertex not covered by the matching to the base of the blossom.



## Blossom Shrinking and Expansion

### Shrinking: Looking for an augmenting path

Edmonds introduced the concept of shrinking blossoms in order to reveal augmenting paths. Consider a blossom B. At a high level when a blossom is shrunken one may imagine that all of the vertices in B aggregate within the base of B, creating a new vertex which represents B in the new graph, refered to hereafter as the shrunken graph, $G_B$.

### Expansion: Arriving at final augmenting path

Edmonds also makes use of the idea of expanding shrunken blossoms in order to get an augmenting path in our unshrunken graph G. Again consider a blossom B. Expansion is the direct opposite of shrinking, the vertices of B that are hidden in the vertex representing B in $G_B$ are reinstated back to their original form before shrinking. Note that, it must be the case that a blossom is shrunk before it may be expanded, expansion only occurs on shrunken blossoms.

## Proof of Maximum

We now prove how the Blossom algorithm always finds a maximal matching using Zwick's and Tarjan's notes as a guide (Zwick, 2009 & Tarjan, 2002). This turns out to be equivalent to not being able to find an augmenting path, as the theorem below points out.

### Augmenting Path Theorem

Theorem:
Provided a matching M and a graph G with vertex set V and edge set E, M is maximum iff there is no M-augmenting path.

Define Q Δ W as the symmetric difference of sets Q and W. Specifically, the set Q Δ W contains the objects which occur within Q and W but not both.

**Proof:**
Suppose there is an M-augmenting path P between nodes v and w. Then N = E(P) Δ M is a matching which covers all of the nodes in P, which is two more than in M. Therefore M is not a maximum matching.

### Theorems for Finding Augmenting Paths

Let us define $G_B$ as resulting graph after blossom B is shrunken. Now, to prove that the Blossom algorithm always finds an augmenting path appropriately we need to prove two things:

Theorem 1:
If there is an augmenting path, AP in G, then there is an augmenting path, $\mathrm{AP}_B$ in $G_B$.

Theorem 2:
If an AP does not exist in G, then an $\mathrm{AP}_B$ does not exist in $G_B$. We note that this is logically equivalent to saying that if there is an $\mathrm{AP}_B$ in $G_B$ then there is an AP in G.

### ▪ Proof of Theorem 1 - AP in G -> AP$_B$ in $G_B$

Again let $v_B$ be the base of the blossom B. There are three cases:

**Case 1: Augmenting path does not go through** $v_B$
If AP doesn't pass through any vertex of B, then AP = AP$_B$, and we are done.

**Case 2:** $v_B$ **is M-covered**
Define S as the stem of the flower that includes B. S must exist because $v_B$ is M-covered and based on the fact that $v_B$ must be labeled EVEN Let M' = M Δ S. $| M | = | M' |$, because S is an even-length alternating path. As well, because $v_B$ is M-covered then $v_B$ is not M'-covered. Replace M by M' and continue to Case 3.

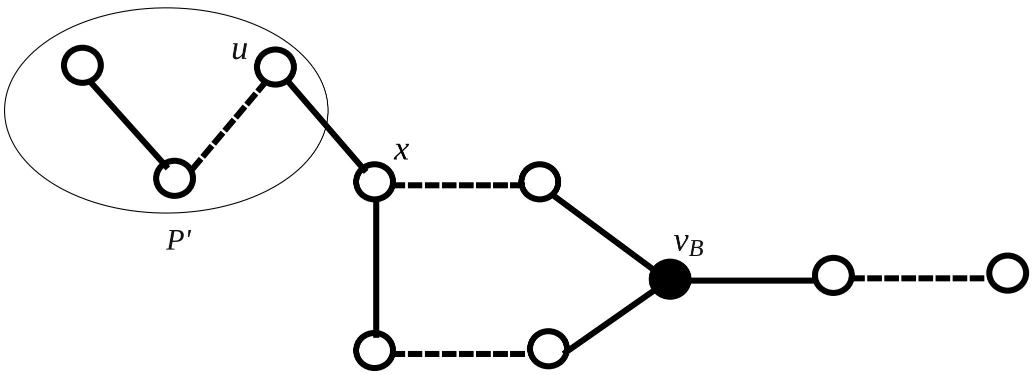
**Case 3:** $v_B$ **is not M-covered**
If AP does pass through a vertex of B, it must be the case that one end of the augmenting path ends at $v_B$. This is because, based on the definition of a blossom as an odd-length alternating path, all of the other vertices in B must be M-covered.
We define P' as the part of P starting at the M-uncovered vertex not in B and ending right before reaching B. Define the edge (u, x) as the edge connecting P' to B, where x ∈ B and u ∈ P'. In $G_B$ vertex u is attached to $v_B$. The path, AP$_B$ = P' $\bigcup$ $v_B$ is an augmenting path in $G_B$. This is because first, $v_B$ is M-uncovered, and second, because (u, $v_B$) is unmatched, because all of vertices in B are M-covered except $v_B$.

In the figures below, dashed lines represent matched edges while solid lines represent unmatched edges.

*Case 3*

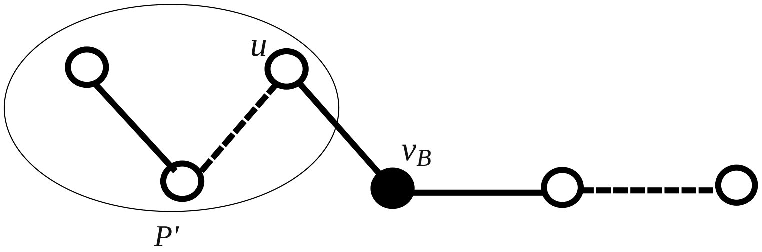
$v_B$ is not $M$ – covered

$\boldsymbol{G}$



$\boldsymbol{G_B}$

$\mathrm{AP}_B = P' \bigcup v_B$



### ■ Proof of Theorem 2 - AP$_B$ in $G_B$ -> AP in G

Define vertex $v_B$ as the vertex representing the blossom in $G_B$. Also define, $M_B$ as the matching in $G_B$ and M as the matching in G.

There are three possible cases:

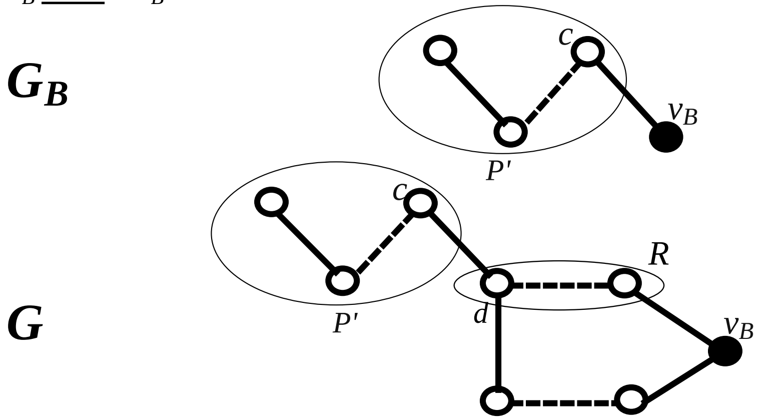
**Case 1: Augmenting path doesn't go through** $v_B$
Then the augmenting path in $G_B$ is also and augmenting path in G and we are done.

**Case 2: Augmenting path starts at** $v_B$
Let ($v_B$, c) be the first edge in AP$_B$, and define P' as the part of AP$_B$ that begins at c and goes all the way to the other unmatched vertex that is not $v_B$. The edge ($v_B$, c) cannot be $M_B$-covered because the first or last edge in an augmenting path must be unmatched. Now, there must be an edge (d, c) in G, where d $\in$ B and (d, c) is unmatched, otherwise c would not be connected to $v_B$ in $G_B$. If d = $v_B$ then, set R = {}. Otherwise, starting at d in G, begin by following a matched edge in B and continue until just before reaching $v_B$. Add these vertices to R. Now, unless R is empty, R must be an alternating path, based on the definition of a blossom. The path AP = $v_B \bigcup$ R $\bigcup$ P' is an augmenting path in G, and we are done.

**Case 3: Augmenting path goes through** $v_B$
Define the edges (e, $v_B$) and ($v_B$, f) as the edges in AP$_B$ which enter and leave B, respectively. Let's say that (e, $v_B$) $\in M_B$ and ($v_B$, f) $\notin M_B$ (the opposite case would yield the same results). Because the only matched edge that enters the blossom must enter the $v_B$ in G, the edge (e, $v_B$) also exists in G. Define $P_e$ as the alternating path going from the end of AP$_B$ to vertex e, such that $P_e$ doesn't include $v_B$. Define $P_f$ as the alternating path going from vertex f to the other end of AP$_B$. Define (f, h) as an edge in G, such that h $\in$ B. Further, define R as the path from h until just before $v_B$, moving around B, first following a matching edge. Based on the definition of a blossom R will be an M-alternating path. The path AP = $P_e \bigcup v_B \bigcup$ R $\bigcup P_f$, is an augmenting path in G, and we are done.

*Case 2*

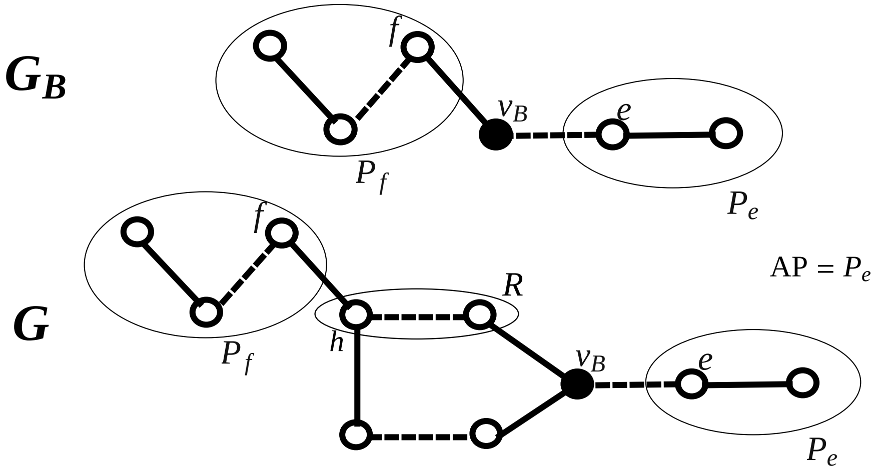
$\text{AP}_B$ starts at $v_B$



$$\text{AP} = v_B \bigcup R \bigcup P'$$

*Case 3*

$\text{AP}_B$ goes through $v_B$



$$\text{AP} = P_e \bigcup v_B \bigcup R \bigcup P_f$$

## The Algorithm

### Algorithm Background

At a high level, the Blossom algorithm works by running the Hungarian algorithm until we run into a blossom, B. When we do, we shrink the blossom. After shrinking, we begin the Hungarian algorithm once again. If another blossom is found, we shrink the blossom and start the Hungarian yet again on the doubly-shrunken graph. If at any point an augmenting path, $AP_B$, is found, we expand any blossoms that were shrunk in the reverse order of shrinking, keeping track of any changes in $AP_B$.

Expansion occurs until the original graph is restored and the augmenting path in the orignial graph, AP, is gotten. Once this occurs, the matched edges of AP are turned into non-matched edges while the non-matched edges are turned into matched edges. The result is an overall matching that is greater than the previous matching by one edge.

If instead we ever become stuck (i.e., we can't shrink a blossom or add any more edges to our tree in the Hungarian algorithm) then the algorithm is done and a maximum matching is achieved.

It should be noted that our implementation of the Blossom algorithm differs from the majority of implementations in that we make use of recursion to shrink blossoms we find while running the Hungarian algorithm. Below we describe our implementation at a more detailed level.

### Blossom Pseudocode

This pseudocode is inspired by the work of Cook and Zwick, (Cook, 1998 & Zwick, 2009).

Start with a maximal matching M and a queue Q, holding a single unmatched vertex $r_1$ in graph G. Label $r_1$ **EVEN**.

```
MAIN ROUTINE

If Q isn't empty,
Take vertex v off the head of the queue.
    If v is labeled EVEN:
         A) Using breadth-first search, move along all of the unmatched edges emanating from v. Call the set of vertices on the opposite end of such edges W.
        B) Add the vertices in W to the queue.
            For each, $w \in W$
                If w is not M-covered:
                    BLOSSOM EXPANSION[w]
                    Restart entire routine.
                If w is M-covered and is labeled EVEN:
                        BLOSSOM SHRINKING[w]
                If w is M-covered and is unlabeled:
                    label w ODD
            Take v off of the queue.
    If v is labeled ODD:
        A) Move along matched edge to vertex h.
```

```
                    If h is labeled ODD:
                        BLOSSOM SHRINKING[h]
                    If h is unlabeled:
                        label h EVEN.
                B) Add h to the queue.
If Q is empty, add a vertex to Q which is unlabeled and not M-covered $r_2$, if it exists. Go to 💡.
If no such vertex exists, then end.
```

BLOSSOM SHRINKING[z]

Shrink the blossom, whose vertices are the symmetric difference of $P_1$, the path from v to $r_i$, the initial M-uncovered vertex, and $P_2$, the path from z to $r_i$. To shrink the blossom:

1. Remove all edges in the blossom.
2. Reattach edges originally attached to the blossom vertices to $v_B$

Recurse into MAIN ROUTINE, starting at 💡.

BLOSSOM EXPANSION[y]

Set AP$_B$ equal to the path from y to $r_1$.

Expand blossoms in the reverse order in which they were shrunk, propagating the augmenting path through the expansion steps.

To expand a blossom:

1. Restore the edges within the blossom.
2. Reattached edges currently attached to $v_B$ back to the vertices in the blossom to which they were originially attached before shrinking.

Extend AP through expanded blossom.

Recurse out of MAIN ROUTINE, propogating AP$_B$ through each expanded blossom.

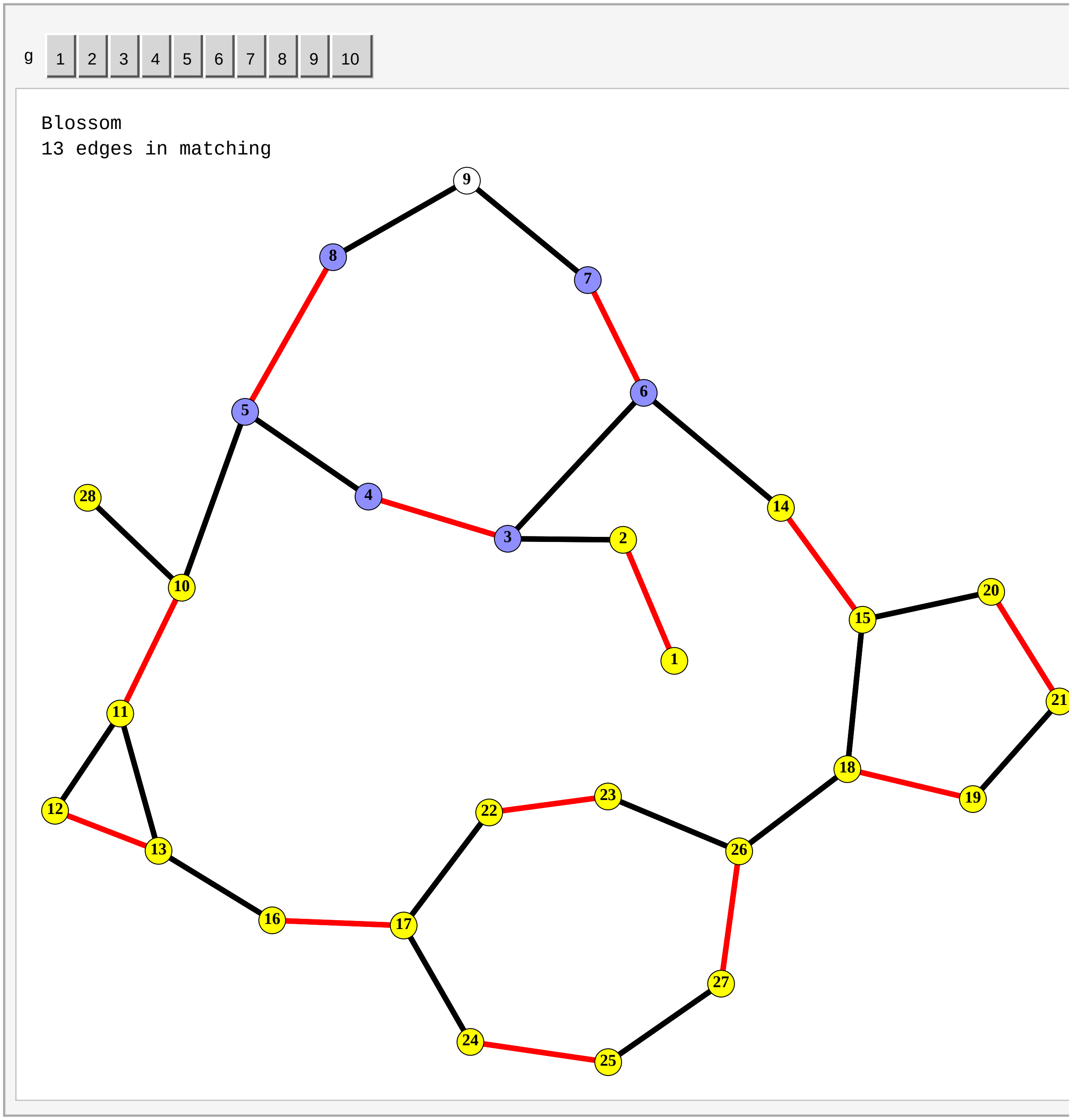
Upon arriving at G, and concurrently AP, flip the matching of AP. Go to 💡.

## Visualization

### Zwick Graph

```
In[21]:= Module[{theGraph, GraphToShow, matching, event,
           listOfInterest, color, blosOrAP, im, locs, blosBase, gAdjTotal, allGraph},
         gAdjTotal = {{2}, {1, 3}, {2, 4, 6}, {3, 5}, {4, 8, 10}, {3, 7, 14}, {6, 9},
           {5, 9}, {7, 8}, {5, 11, 28}, {10, 12, 13}, {11, 13}, {11, 12, 16}, {6, 15},
           {14, 18, 20}, {13, 17}, {16, 22, 24}, {15, 19, 26}, {18, 21}, {15, 21}, {19, 20},
           {17, 23}, {22, 26}, {17, 25}, {24, 27}, {18, 23, 27}, {25, 26}, {10}};
         allGraph = FromAdjacencyLists[gAdjTotal];
         MaximumMatchingMK[allGraph, TraceQ → True];
         im = GraphPlot[allGraph, Method → "SpringEmbedding", VertexLabeling → True];
         locs =
          VertexCoordinateRules /. Cases[InputForm[ im], HoldPattern[VertexCoordinateRules → _], ∞];
         locs = ReplacePart[locs, {8 → locs[[8]] + {-.5, 3.}, 3 → locs[[3]] + {-0.2, -1.0},
            9 → locs[[9]] + {-.3, 4.}, 7 → locs[[7]] + {0, 2.5}, 4 → locs[[4]] + {-0.2, -0.4},
            6 → locs[[6]] + {0, 0.7}, 2 → locs[[2]] + {0.6, -2.0},
            1 → locs[[1]] + {1.0, -3.8}, 5 → locs[[5]] + {-0.5, 1.}}];
         Manipulate[
          SeedRandom[7];
          theGraph = listOfGraphs[[g]];
          matching = listOfMatchings[[g]];
          event = listOfEvents[[g]];
          blosOrAP = listOfBlosAndAPs[[g]];
          blosBase = blossomBases[[g]];
          GraphToShow = FromAdjacencyLists[theGraph[[1 ;; -2]]];
          If[event == "Blossom", listOfInterest = blosOrAP[[2]]; color = Lighter@Lighter@Blue];
          If[event == "AugmentingPath", listOfInterest = blosOrAP; color = Yellow];
          If[event == "Stuck", listOfInterest = {}; color = Yellow];
          Column[{event, StringForm["`` edges in matching", Length[Select[matching, #1 ≠ 0 &]] / 2],
            Show[
             GraphPlot[GraphToShow, VertexCoordinateRules → Thread[Range[V[allGraph]] → locs],
              EdgeRenderingFunction → ({If[event == "AugmentingPath" &&
                     (MemberQ[listOfInterest, #2] || MemberQ[listOfInterest, Reverse[#2]]),
                    Lighter@Green, White], AbsoluteThickness[8], Line[#1]} &)],
             GraphPlot[GraphToShow, VertexCoordinateRules → Thread[Range[V[allGraph]] → locs],
              VertexCoordinateRules →
               Thread[Range[V[GraphToShow]] → RandomReal[2, {V[GraphToShow], 2}]]
               (*GraphToShow[[2]]]*), EdgeRenderingFunction →
               ({If[matching[[#2[[1]]]] == #2[[2]], Red, Black], AbsoluteThickness[3], Line[#1]} &),
              VertexRenderingFunction →
               ({If[#2 == blosBase && (event == "Blossom" || event == "AugmentingPath"),
                   White, If[MemberQ[listOfInterest, #2], color, Yellow]],
                  (*Which[event=="Blossom",If[listOfInterest[[1]]==#2,Lighter@Lighter@Red],
                   event=="AugmentinPath",If[MemberQ[listOfInterest,#2],Lighter@Green]];*)
                  PointSize[.02], EdgeForm[Black], Disk[#1, .1], Black,
                  Text[Style[#2, Bold], #1]} &)], PlotRangePadding → 0, ImageSize → 570]}],
          {g, 1, Length[listOfGraphs], 1}, ControlType → Setter, TrackedSymbols → True,
          SaveDefinitions → True]]

Total number of recursion calls: 6
```

Out[21]=

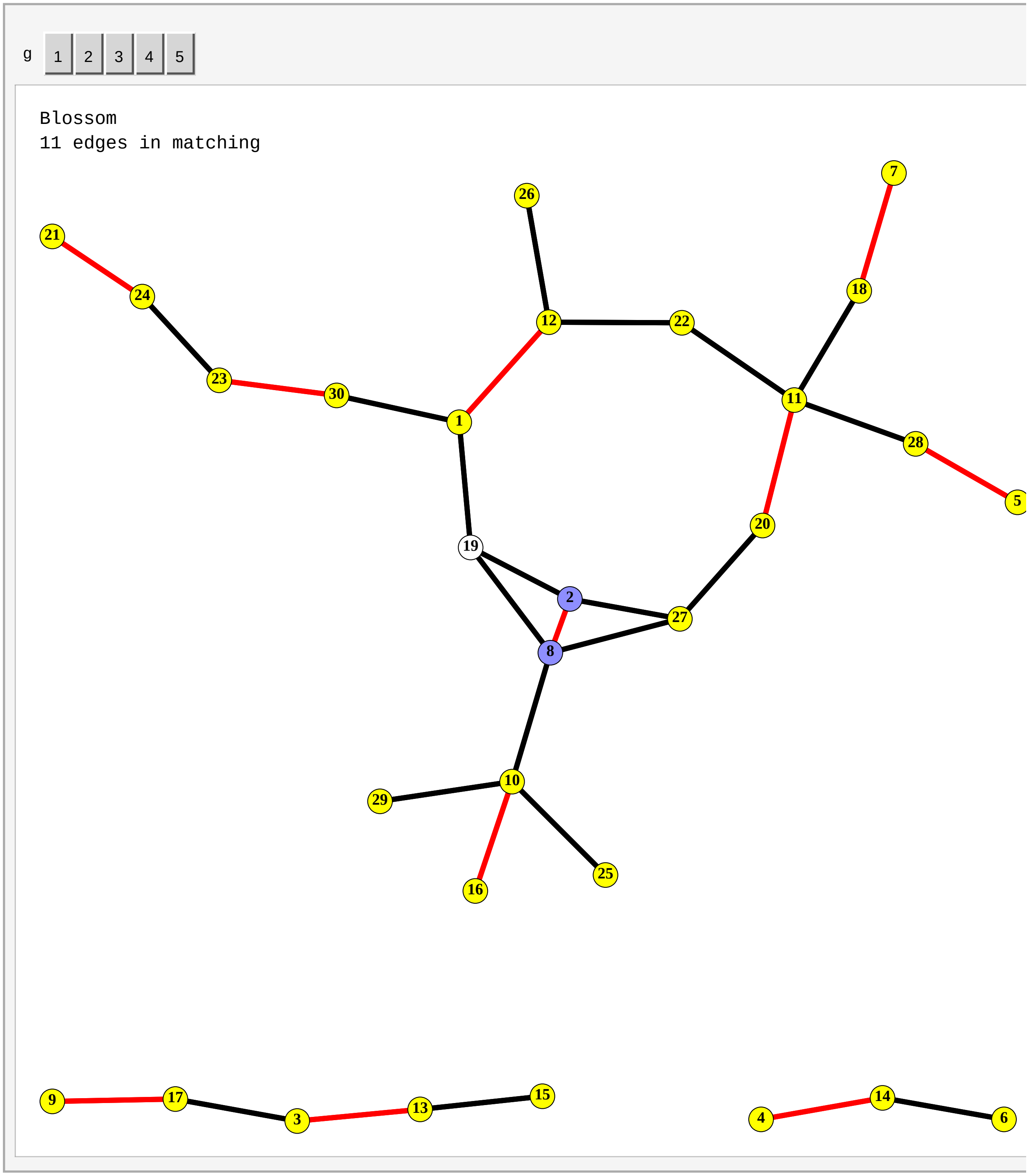


```
Blossom
13 edges in matching
```

## ■ Random Graph

```
In[22]:= Module[{theGraph, GraphToShow, matching, event,
           listOfInterest, color, blosOrAP, im, locs, blosBase, gAdjTotal, allGraph},
         SeedRandom[9];
         allGraph = RandomGraph[30, .07];
         MaximumMatchingMK[allGraph, TraceQ → True];
         im = GraphPlot[allGraph, Method → "SpringEmbedding", VertexLabeling → True];
         locs =
          VertexCoordinateRules /. Cases[InputForm[ im], HoldPattern[VertexCoordinateRules → _], ∞];
         (*locs = ReplacePart[locs, {8→ locs[[8]]+{-.5, 3.},
             3→ locs[[3]]+{-0.2, -1.0},9→ locs[[9]]+{-.3, 4.},7→ locs[[7]]+{0, 2.5},
             4→ locs[[4]]+{-0.2, -0.4},6→ locs[[6]]+{0, 0.7},2→ locs[[2]]+{0.6, -2.0},
             1→ locs[[1]]+{1.0, -3.8},5→ locs[[5]]+{-0.5, 1.}}];*)
         Manipulate[
          SeedRandom[7];
          theGraph = listOfGraphs[[g]];
          matching = listOfMatchings[[g]];
          event = listOfEvents[[g]];
          blosOrAP = listOfBlosAndAPs[[g]];
          blosBase = blossomBases[[g]];
          GraphToShow = FromAdjacencyLists[theGraph[[1 ;; -2]]];
          If[event == "Blossom", listOfInterest = blosOrAP[[2]]; color = Lighter@Lighter@Blue];
          If[event == "AugmentingPath", listOfInterest = blosOrAP; color = Yellow];
          If[event == "Stuck", listOfInterest = {}; color = Yellow];
          Column[{event, StringForm["`` edges in matching", Length[Select[matching, #1 ≠ 0 &]] / 2],
            Show[
             GraphPlot[GraphToShow, VertexCoordinateRules → Thread[Range[V[allGraph]] → locs],
              EdgeRenderingFunction → ({If[event == "AugmentingPath" &&
                   (MemberQ[listOfInterest, #2] || MemberQ[listOfInterest, Reverse[#2]]),
                  Lighter@Green, White], AbsoluteThickness[8], Line[#1]} &)],
             GraphPlot[GraphToShow, VertexCoordinateRules → Thread[Range[V[allGraph]] → locs],
              VertexCoordinateRules →
               Thread[Range[V[GraphToShow]] → RandomReal[2, {V[GraphToShow], 2}]]
               (*GraphToShow[[2]]*), EdgeRenderingFunction →
               ({If[matching[[#2[[1]]]] == #2[[2]], Red, Black], AbsoluteThickness[3], Line[#1]} &),
              VertexRenderingFunction →
               ({If[#2 == blosBase && (event == "Blossom" || event == "AugmentingPath"),
                  White, If[MemberQ[listOfInterest, #2], color, Yellow]],
                 (*Which[event=="Blossom",If[listOfInterest[[1]]==#2,Lighter@Lighter@Red],
                   event=="AugmentinPath",If[MemberQ[listOfInterest,#2],Lighter@Green]];*)
                 PointSize[.02], EdgeForm[Black], Disk[#1, .1], Black,
                 Text[Style[#2, Bold], #1]} &)], PlotRangePadding → 0, ImageSize → 570]}],
          {g, 1, Length[listOfGraphs], 1}, ControlType → Setter, TrackedSymbols → True,
          SaveDefinitions → True]]

Total number of recursion calls: 4
```

Out[22]=

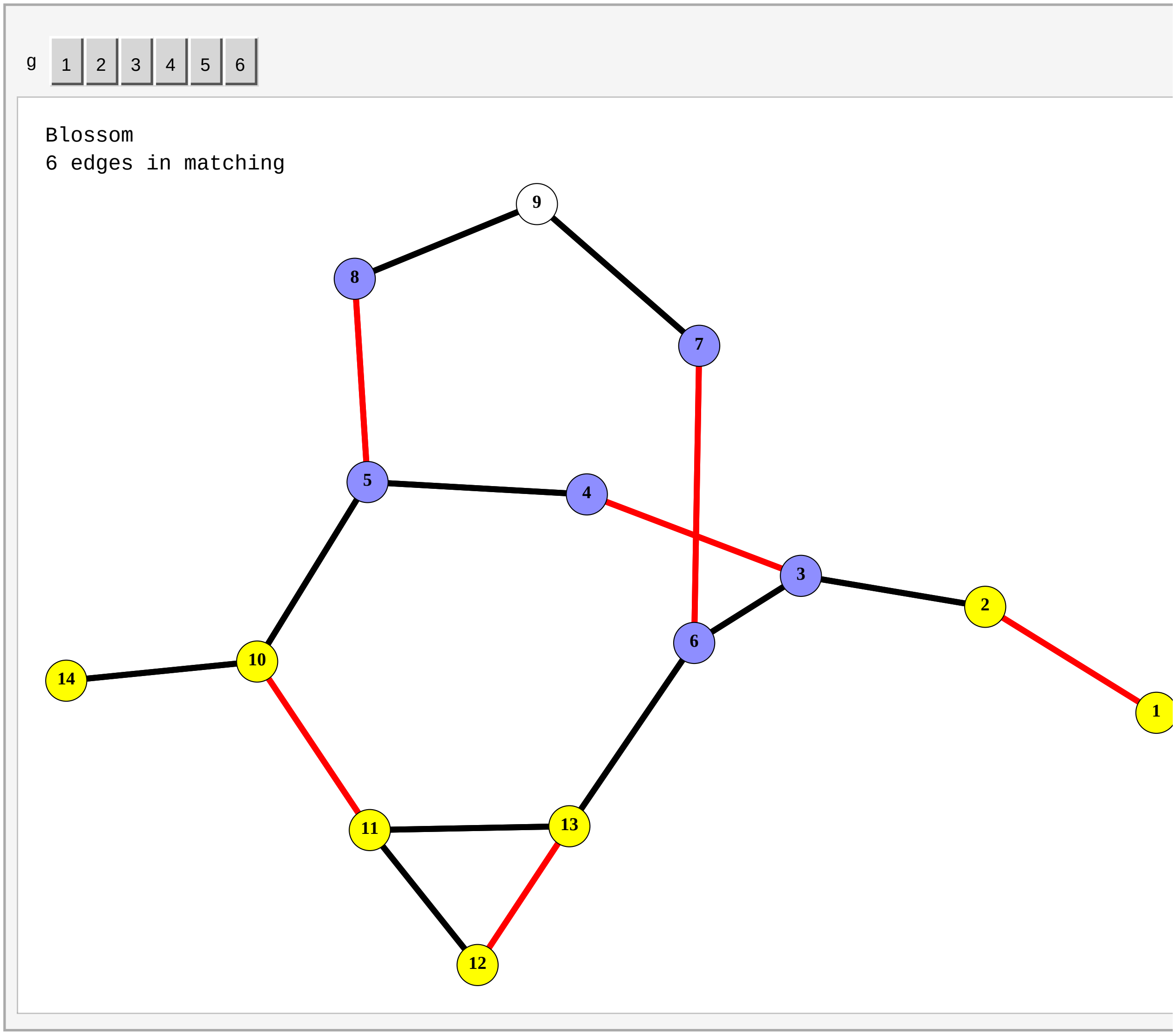


```
Blossom
11 edges in matching
```

### ▪ 5 Minute Presentation Graph

```
In[23]:= Module[{theGraph, GraphToShow, matching, event,
           listOfInterest, color, blosOrAP, im, locs, blosBase, gAdjTotal, allGraph},
         gAdjTotal = {{2}, {1, 3}, {2, 4, 6}, {3, 5}, {4, 8, 10}, {3, 7, 13}, {6, 9},
            {5, 9}, {7, 8}, {5, 11, 14}, {10, 12, 13}, {11, 13}, {6, 11, 12}, {10}};
         allGraph = FromAdjacencyLists[gAdjTotal];
         MaximumMatchingMK[allGraph, TraceQ → True];
         im = GraphPlot[allGraph, Method → "SpringEmbedding", VertexLabeling → True];
         locs =
          VertexCoordinateRules /. Cases[InputForm[ im], HoldPattern[VertexCoordinateRules → _], ∞];
         (*locs = ReplacePart[locs, {8→ locs[[8]]+{-.5, 3.},
             3→ locs[[3]]+{-0.2, -1.0},9→ locs[[9]]+{-.3, 4.},7→ locs[[7]]+{0, 2.5},
             4→ locs[[4]]+{-0.2, -0.4},6→ locs[[6]]+{0, 0.7},2→ locs[[2]]+{0.6, -2.0},
             1→ locs[[1]]+{1.0, -3.8},5→ locs[[5]]+{-0.5, 1.}}];*)
         Manipulate[
          SeedRandom[7];
          theGraph = listOfGraphs[[g]];
          matching = listOfMatchings[[g]];
          event = listOfEvents[[g]];
          blosOrAP = listOfBlosAndAPs[[g]];
          blosBase = blossomBases[[g]];
          GraphToShow = FromAdjacencyLists[theGraph[[1 ;; -2]]];
          If[event ⩵ "Blossom", listOfInterest = blosOrAP[[2]]; color = Lighter@Lighter@Blue];
          If[event ⩵ "AugmentingPath", listOfInterest = blosOrAP; color = Yellow];
          If[event ⩵ "Stuck", listOfInterest = {}; color = Yellow];
          Column[{event, StringForm["`` edges in matching", Length[Select[matching, #1 ≠ 0 &]] / 2],
            Show[
             GraphPlot[GraphToShow, VertexCoordinateRules → Thread[Range[V[allGraph]] → locs],
              EdgeRenderingFunction → ({If[event ⩵ "AugmentingPath" &&
                   (MemberQ[listOfInterest, #2] || MemberQ[listOfInterest, Reverse[#2]]),
                  Lighter@Green, White], AbsoluteThickness[8], Line[#1]} &)],
             GraphPlot[GraphToShow, VertexCoordinateRules → Thread[Range[V[allGraph]] → locs],
              VertexCoordinateRules →
               Thread[Range[V[GraphToShow]] → RandomReal[2, {V[GraphToShow], 2}]]
               (*GraphToShow[[2]]]*), EdgeRenderingFunction →
               ({If[matching[[#2[[1]]]] == #2[[2]], Red, Black], AbsoluteThickness[3], Line[#1]} &),
              VertexRenderingFunction →
               ({If[#2 ⩵ blosBase && (event ⩵ "Blossom" || event ⩵ "AugmentingPath"),
                   White, If[MemberQ[listOfInterest, #2], color, Yellow]],
                  (*Which[event⩵"Blossom",If[listOfInterest[[1]]⩵#2,Lighter@Lighter@Red],
                    event⩵"AugmentinPath",If[MemberQ[listOfInterest,#2],Lighter@Green]];*)
                  PointSize[.02], EdgeForm[Black], Disk[#1, .1], Black,
                 Text[Style[#2, Bold], #1]} &)], PlotRangePadding → 0, ImageSize → 570]}],
          {g, 1, Length[listOfGraphs], 1}, ControlType → Setter, TrackedSymbols → True,
          SaveDefinitions → True]]

Total number of recursion calls: 4
```

Out[23]=



```
Blossom
6 edges in matching
```

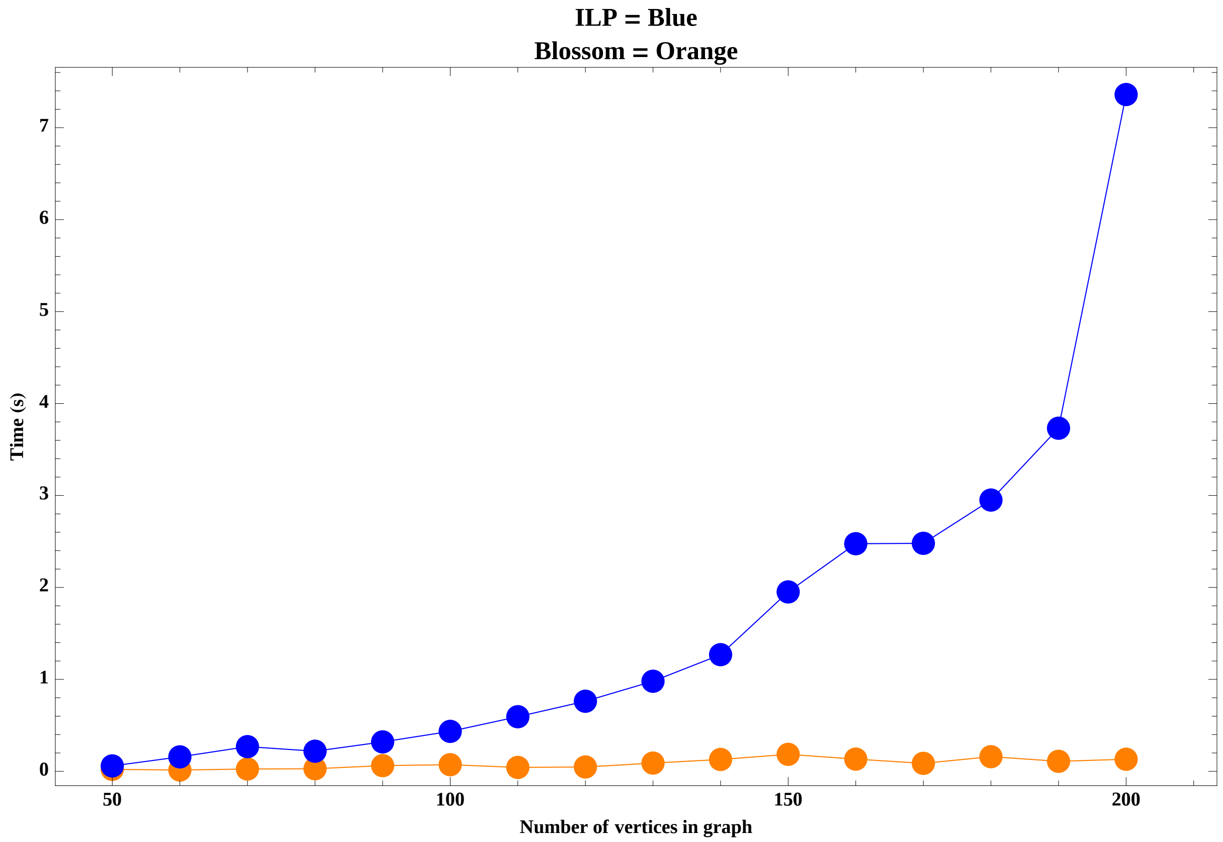
## Speed Comparison

### ■ A set of random graphs

Here we test the Blossom algorithm against a maximum matching ILP algorithm on random graphs having between 50 and 200 vertices.

```
SeedRandom[5];
ILPTimings = Table[g = RandomGraph[i, 0.2];
   {V[g], MaximumMatchingILP[g, TraceQ → False] // Timing // First}, {i, 50, 200, 10}];
SeedRandom[5];
BlossomTimings = Table[g = RandomGraph[i, 0.2];
   {V[g], MaximumMatchingMK[g, TraceQ → False] // Timing // First}, {i, 50, 200, 10}];
```

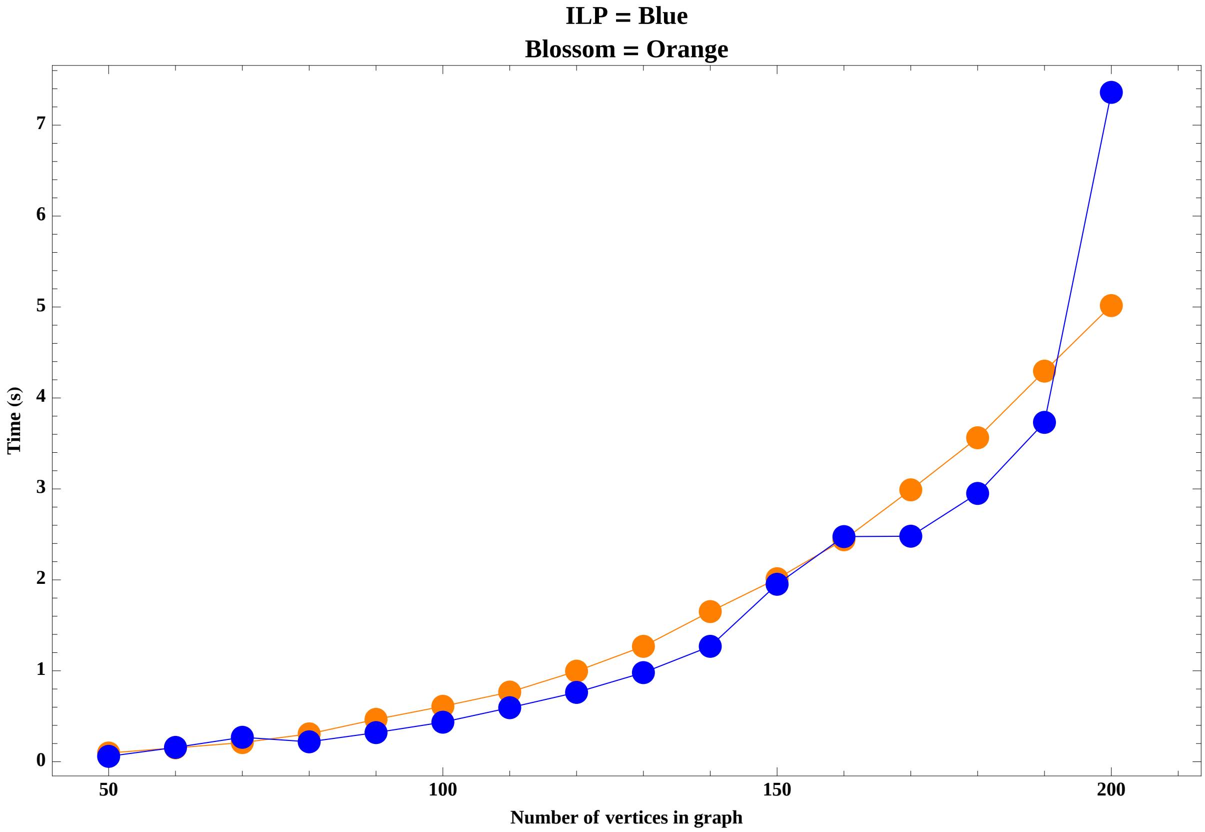
## ■ List Plot of ILP vs Blossom



Ah, we note that ILP seems to exponentially increase in time as the number of graph edges increases. However, the timing of the Blossom algorithm does not increase noticeably for these graphs. One potential reason for this is that we essentially start the Blossom algorithm with a maximal matching, whereas ILP starts with no sort of matching. To make a fairer comparison of the algorithms, let's remove the maximal matching lines in the Blossom code and plot our results again.

## ■ List Plot of ILP vs Blossom (without initial maximal matching)

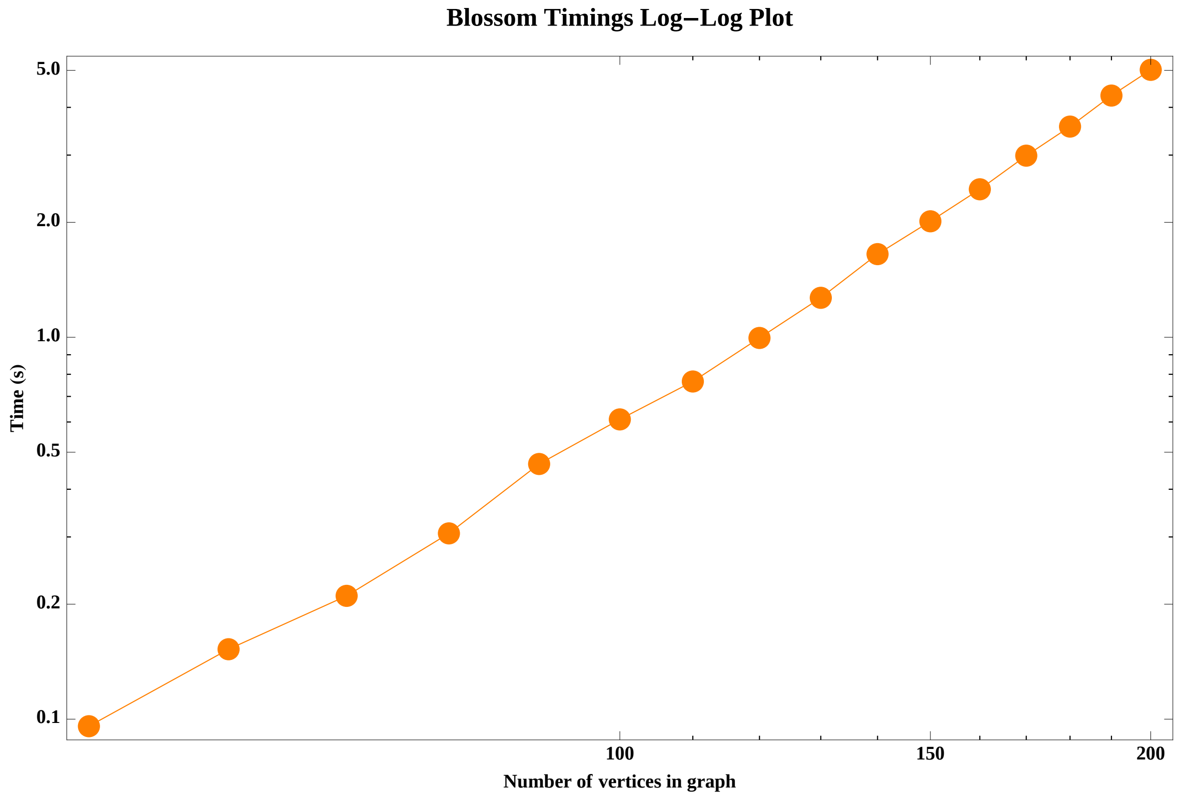
```
(* Before running, comment out the lines from matchingList=MaximalMatchingSW[Graph];
up until Do[matching[[e]]=Reverse[e],{e,matchingList}] in MaximumMatchingMK *)
SeedRandom[5];
BlossomTimingsNoMaximal = Table[g = RandomGraph[i, 0.2];
   {V[g], MaximumMatchingMK[g, TraceQ → False] // Timing // First}, {i, 50, 200, 10}];
```



Aha! We see that when we do not use a maximal matching, the timings of the Blossom algorithm are quite similar to those of ILP matching, save for the last data point. In fact, if we were to extrapolate to graphs having 300 vertices we would see a dramatic difference between the timings of the Blossom algorithm and those of ILP.

To get a better idea of the complexity of the blossom algorithm, do a log-log plot of its timings.

### ■ Log-Log Plot of Blossom Results



We note that the complexity appears to be polynomial. We are interested in the degree of this polynomial, which we can find by taking the slope of our log-log plot:

```
(Log[BlossomTimingsNoMaximal[[-1, 2]]] - Log[BlossomTimingsNoMaximal[[1, 2]]]) /
 (Log[BlossomTimingsNoMaximal[[-1, 1]]] - Log[BlossomTimingsNoMaximal[[1, 1]]])
```

```
2.855
```

Wikipedia notes that the complexity of the blossom algorithm is O(V^4), whereas our complexity is somewhat closer to O(V^3).

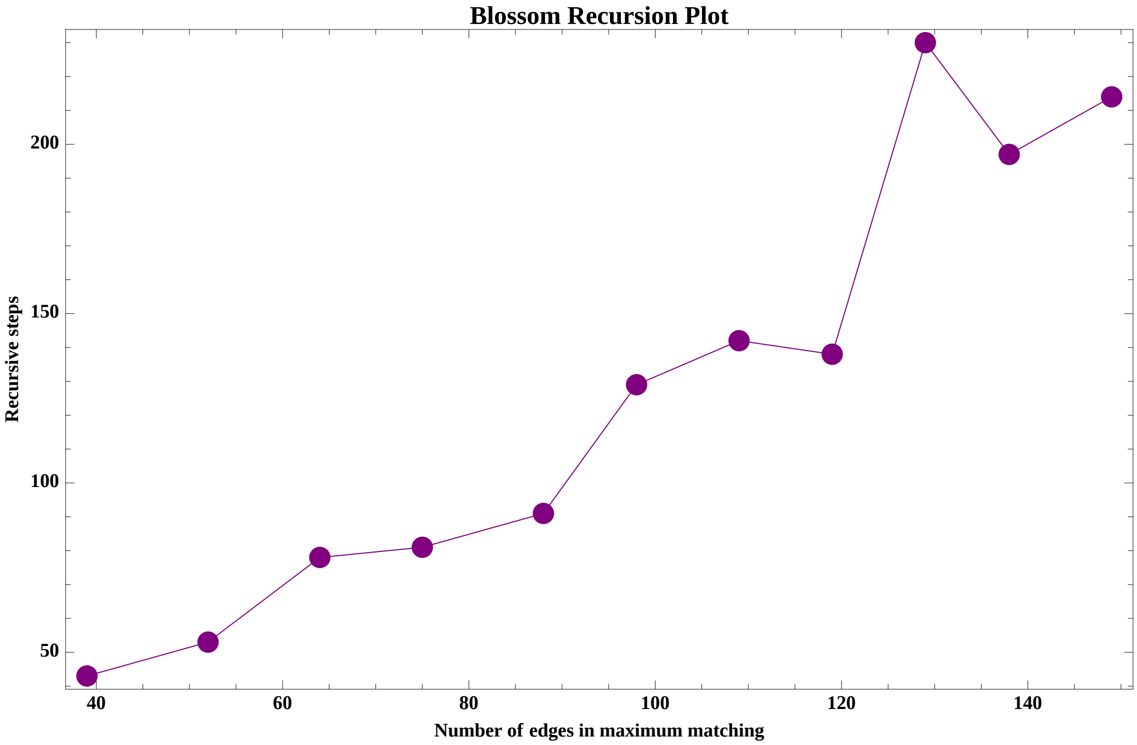
## Recursion Count

### ■ A set of random graphs

To get an idea of the number of recursive steps the Blossom algorithm lets take a look at a set of random graphs having between 100 and 300 vertices. As above, we will remove the finding of the maximum matching within the Blossom code to get a better idea of the amount of recursion necessary for the Blossom algorithm.

```
(* Before running, comment out the lines from matchingList=MaximalMatchingSW[Graph];
up until Do[matching[[e]]=Reverse[e],{e,matchingList}] in MaximumMatchingMK *)
SeedRandom[20];
Table[g = RandomGraph[i, 0.02];
 MaximumMatchingMK[g, TraceQ → True] // Length, {i, 100, 300, 20}]
```

```
Total number of recursion calls: 43

Total number of recursion calls: 53

Total number of recursion calls: 78

Total number of recursion calls: 81

Total number of recursion calls: 91

Total number of recursion calls: 129

Total number of recursion calls: 142

Total number of recursion calls: 138

Total number of recursion calls: 230

Total number of recursion calls: 197

Total number of recursion calls: 214

      {39, 52, 64, 75, 88, 98, 109, 119, 129, 138, 149}
```



We note a general upward trend in the number of recursive steps necessary as the number of edges in the maximum matching increases. If we calculate the slope using the first and last points on the graph we get a general relationship:

Recursive steps ≈ 1.55*(Number of edges in maximum matching)

## References

- Cook, William et al. Combinatorial optimization, Wiley, 1998.
- Edmonds, Jack. "Paths, trees, and flowers". Canad. J. Math. 17: 449–467. 1965.

Kuhn, Harold W. The Hungarian Method for the assignment problem. Naval Research Logistics Quarterly, 2:83–97, 1955.

- Tarjan, Robert. Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching, 2002. (http://www.cs.dartmouth.edu/~ac/Teach/CS105-Winter05/Handouts/tarjan-blossom.pdf)
- Trinajsti▮, Nenad; Klein, Douglas J.; Randi▮, Milan, On some solved and unsolved problems of chemical graph theory, International Journal of Quantum Chemistry 30 (S20): 699–742, 1986.
- Zwick, Uri. Lecture notes on: Maximum matching in bipartite and non-bipartite graphs, December 2009. (http://www.cs.tau.ac.il/~zwick/grad-algo-0910/match.pdf)
- Code for blossom shrinking, tree construction and graph placement, ideas about recursion, and code organization received from Stan Wagon.

## Code Appendix

### Blossom Code

```
In[6]:= << Combinatorica`

In[7]:= Options[MaximalMatchingSW] = {StartEdgeIndex -> Automatic};

        MaximalMatchingSW[g_Graph, opts___] := Module[
        {match = {}, ee = Edges[g]}, st = StartEdgeIndex /. {opts} /. Options[MaximalMatchingSW];
           st = st /. Automatic -> 1; match = ee[[st]];
        Clear[seen]; seen[_] := False;
        seen[match[[1]]] = True;
        seen[match[[2]]] = True;
        Do[If[! seen[e[[1]]] && ! seen[e[[2]]], match = Join[match, e];
             seen[e[[1]]] = seen[e[[2]]] = True], {e, Delete[ee, st]}]; Sort /@ Partition[match, 2]];



In[9]:= DoublePathToFlower[doublePath_] :=
          Module[{timesOccurred = Tally[doublePath], twice, locations},
           twice = Select[timesOccurred, #[[2]] == 2 &];
           If[twice == {}, Return[{{}, doublePath}]];
           twice = twice[[-1, 1]]; (* Get the first part of the last list *)
           locations = Flatten[Position[doublePath, twice]];
           {Take[doublePath, locations[[1]] - 1], Most[Take[doublePath, locations]]}];

        FindAugOrBlossom[Gadj_, _] := {"Stuck", {}} /; Length[Gadj] ≤ 2;
        FindAugOrBlossom[Gadj_, MV_] := Module[{parents, n, vertices, treeRoots, head, tail,
            Q, neighbors, novelNeighbors, vw = {}, evenQ, uncoveredNode = 0, v, doublePath},
           n = Length[Gadj] - 1;
           vertices = Range[n];
           parents = Table[0, {n}];
           treeRoots = Select[vertices, MV[[#]] == 0 &];
           (*Print["Tree roots"];
           Print[treeRoots];*)
           head = tail = 0;
           (*Q=Table[0,{n}];*)
           (*evenQ=Table[False,{n}];*)
```

```
(* Workhorse right here *)
While[
 If[head == tail && uncoveredNode == 0 && vw == {} && treeRoots ≠ {},
  Q[tail] = parents[[treeRoots[[1]]]] = treeRoots[[1]];
  tail++;
  evenQ[treeRoots[[1]]] = True;
  treeRoots = Rest[treeRoots]];
 head < tail && treeRoots ≠ {} && vw == {},
 v = Q[head];
 head++;
 neighbors = Gadj[[v]];
 If[evenQ[v] == True,
  (* check for possibility of:
    1. augmenting path
     2. extending tree
     3. blossom
  *)
  (* NOTE: This goes through all neighbors, pretty sweet *)
  Do[
   Which[ parents[[w]] == 0 && MV[[w]] == 0
     (* found augmenting path *), parents[[w]] = v; uncoveredNode = w; Break[],
    parents[[w]] == 0 && MV[[w]] ≠ 0 (* extend tree *), parents[[w]] = v;
    Q[tail] = w; tail++; evenQ[w] = False,
    parents[[w]] ≠ 0 && evenQ[w], vw = {v, w}; Break[]],
   {w, neighbors}],

  (* check for the matching edge *)
  (* NOTE: This goes through all neighbors, pretty sweet *)
  Do[
   If[MV[[v]] == w && parents[[w]] == 0,
    parents[[w]] = v; Q[tail] = w; tail++; evenQ[w] = True; Break[]];
   (* Found odd-odd blossom *)
   If[MV[[v]] == w && parents[[w]] ≠ 0,
    If[! evenQ[w],
     vw = {MV[[w]], w}; Break[]]]
   , {w, neighbors}]
 ]
];

Which[uncoveredNode ≠ 0,
 {"AugmentingPath", Reverse[Most[FixedPointList[parents[[#]] &, uncoveredNode]]]
  (* FixedPointList:
       - Goes through parents until finding a fixed point, the root
   Most:
        - Gets all but the last, because FixedPointList displays
           the fixed point twice
   Reverse:
        - Starts the list at the blossom/stem, going to the uncovered node
  *)
```

```
    },
    vw ≠ {},
    {"Blossom", doublePath = Join[Reverse[Most[FixedPointList[parents[[#]] &, vw[[1]]]]],
       Most[Most[FixedPointList[parents[[#]] &, vw[[2]]]]]];
     DoublePathToFlower[doublePath]},
    True (* Default *), {"Stuck", {}}]
  ];

ShrinkFlower[Gadj_, MV_, {stem_, blossom_}] :=
  Module[{Gadj1, mate, ns, stemBase, blossomNeighbors, blossomNeighborConnections},
   Gadj1 = Gadj;
   mate = MV;
   stemBase = blossom[[1]];
   (* This gets all of the neighbors
    of the blossom not in the blossom! The novel neighbors *)
   blossomNeighbors = Complement[Union @@ Gadj1[[blossom]], blossom];

   (* This sets all of the blossom vertices adjacencies to {}, except the base! *)
   Gadj1[[Rest[blossom]]] = {};

   (* This sets the adjacency of the stem to be the novel neighbors of the blossom! *)
   Gadj1[[stemBase]] = blossomNeighbors;

   (************* Added by MK ************)
   Do[If[! MemberQ[Gadj1[[bN]], stemBase],
     PrependTo[Gadj1[[bN]], stemBase]], {bN, blossomNeighbors}];

   (* This does exactly what we want, avoiding a while loop. Here we remove the vertices
    in the adjacency list which have been shrunk and removed in the blossom*)
   Gadj1 = DeleteCases[Gadj1, Alternatives @@ Rest[blossom], ∞];

   (* Attaches things not in the blossom*)
   blossomNeighborConnections = Table[{b, Complement[Gadj[[b]], blossom]}, {b, blossom}];

   (* This is key. This attaches the stem and blossom at the
    end of the adjacency list. But don't we want to append or join it?*)
   Gadj1[[-1]] = {stem, blossomNeighborConnections};

   (* This takes the matching function and removes the matching in the blossom *)
   (*mate[[blossom]]=0;*)

   ns = Length[stem];

   (* This flips the matching on the stem! ****Look at this more**** *)
   (* Killed by Zwick *)
   (*Do[mate[stem[[i]]]=stem[[i-1]];mate[stem[[i-1]]]=stem[[i]],{i,ns,2,-1}];*)

   {Gadj1, mate}];

ExpandBlossomsAndChangeMatching[Gadj1_, MV_, augPath_] :=
  Module[{Gadj, mate, stem, blossomConnectData, stemBase,
 place, blossom, blossomNeighbors, b, v, c, pathToAdd, matchingFlag,
 currVert, augmentingPath, position, nextInBlossom, pB, pC, basePositions},
Gadj = Gadj1;
mate = MV;
augmentingPath = augPath;

If[augmentingPath == {},
 Return[{}]];
If[Gadj[[-1]] == {},
 Return[augmentingPath]];


{stem, blossomConnectData} = Gadj[[-1]];


b = blossomConnectData[[1, 1]];
(* OUTER IF START *)
(* Aug path goes through blossom *)
If[MemberQ[augmentingPath, b],
 (* OUTER IF CASE 1 *)
 (* Restore the blossom *)
 blossom = Table[First[i], {i, blossomConnectData}];

 Do[
  (* Get position in blossom of vertex *)
  place = Position[blossom, First[i], 1, 1][[1, 1]];
  (* Connect blossom vertices with each other *)
  Which[
   place == 1, Gadj[[First[i]]] = {blossom[[place + 1]], blossom[[-1]]},
   place == Length[blossom], Gadj[[First[i]]] = {blossom[[1]], blossom[[place - 1]]},
   True, Gadj[[First[i]]] = {blossom[[place + 1]], blossom[[place - 1]]}];
  (* Add non-blossom vertices to blossom adj positions *)
  Gadj[[First[i]]] = Join[Gadj[[First[i]]], Last[i]];
  (* Add blossom vertices to non-blossom adj positions *)
  If[Last[i] ≠ {},
   Do[
    AppendTo[Gadj[[j]], First[i]];
    basePositions = Position[Gadj[[j]], b, 1, 1];
    Gadj[[j]] = Delete[Gadj[[j]], basePositions] , {j, Last[i]}]
  ],
  {i, blossomConnectData}];

 (* Restore matching of blossom *)
 (*Do[mate[[blossom[[i]]]]=blossom[[i+1]];
   mate[[blossom[[i+1]]]]=blossom[[i]]
   ,{i,2,Length[blossom],2}];*)
 (* If structure below can perhaps be simplified
  further by just checking where matching enters blossom, if it does *)

 (* INNER IF START *)
 (* Aug path ends at blossom *)
```

```
If[augmentingPath[[1]] == b,
 (* INNER IF CASE 1 *)
 (* From part right before blossom, get where it connects to blossom *)
 c = augmentingPath[[2]];
 (* Need to insert blossom path after vertex connected to blossom *)
 (* Aug path goes through blossom *)
 ,
 (* INNER IF CASE 2 *)
 (*mate[[b]]=stem[[-1]];*)
 position = Flatten[Position[augmentingPath, b]][[1]];
 If[augmentingPath[[position + 1]] == stem[[-1]],
  c = augmentingPath[[position - 1]],
  c = augmentingPath[[position + 1]]
 ]
];
(* INNER IF END *)
(* What if c connected to multiple parts of blossom???,
   Right now I just pick one *)
v = Select[blossom, MemberQ[Gadj[[c]], #] &][[1]];
pathToAdd = {v};

matchingFlag = True;
currVert = v;
(* Follow blossom to b alternating the matching *)
While[currVert ≠ b,
 If[matchingFlag,
  AppendTo[pathToAdd, mate[[currVert]]]; matchingFlag = False,
  (* Else *)
  nextInBlossom =
   Select[blossom, MemberQ[Complement[Gadj[[currVert]], pathToAdd], #] &][[1]];
  AppendTo[pathToAdd, nextInBlossom];
  matchingFlag = True;
 ];
 currVert = pathToAdd[[-1]];
];

pathToAdd = Most[pathToAdd];
(* Reverse path if b before c in augPath *)
(* Definitely redundant, recheck this to be sure *)
pB = Flatten[Position[augmentingPath, b]][[1]];
pC = Flatten[Position[augmentingPath, c]][[1]];

If[pB < pC,
 pathToAdd = Reverse[pathToAdd];
 augmentingPath = Join[augmentingPath[[1 ;; pB]], pathToAdd, augmentingPath[[pC ;; -1]]]
 ,
 augmentingPath = Join[augmentingPath[[1 ;; pC]], pathToAdd, augmentingPath[[pB ;; -1]]]
];
Return[augmentingPath]

,
    (* OUTER IF CASE 2 *)
    (* Aug path doesn't go through blossom, just return aug path *)
    Return[augmentingPath]
   ]
   (* OUTER IF END *)

  ];


(* This function takes a graph object,
a matching as an edge list and forms an adjacency list and a mate list *)

$RecursionLimit = ∞;
Options[AugmentingPathRecursive] = {TraceQ → False, MaxRecursion → ∞};

AugmentingPath[G_, mateList_, opts___] := Module[{result},
   result = AugmentingPathRecursive[Append[ToAdjacencyLists[G], {}], mateList, opts];
   result
  ];



AugmentingPathRecursive[GadjAug_, mate_, opts___] :=
  Module[{shrunken, augOrBlossom, Hadj, MM, n = Length[GadjAug], newPath, newMate, trQ},
   (* Something with options *)
   {maxrec, trQ} = {MaxRecursion, TraceQ} /. {opts} /. Options[AugmentingPathRecursive];

   augOrBlossom = FindAugOrBlossom[GadjAug, mate];
   count++; mPrint[count]; If[count > maxrec, Return["Exceeded MaxRecursion"]];
   (* If[trQ,Print[StringForm["Found ``,  ``; entering recursive call number ``",
       augOrBlossom[[1]],augOrBlossom[[2]], count]]]; *)
   If[trQ, AppendTo[listOfGraphs, GadjAug];
    AppendTo[listOfMatchings, mate];
    AppendTo[listOfEvents, augOrBlossom[[1]]];
    AppendTo[listOfExtraBlossoms, {}];
    If[augOrBlossom[[1]] === "AugmentingPath",
     AppendTo[listOfBlosAndAPs, Partition[augOrBlossom[[2]], 2, 1]],
     AppendTo[listOfBlosAndAPs, augOrBlossom[[2]]]];
    ]
   ];
   Which[
    (* This is the base case *)
    augOrBlossom[[1]] === "AugmentingPath",
    If[trQ, AppendTo[blossomBases, 0]];
    newPath = ExpandBlossomsAndChangeMatching[GadjAug, mate, augOrBlossom[[2]]],
    augOrBlossom[[1]] === "Stuck",
    newPath = {}; If[trQ, AppendTo[blossomBases, 0]],
    augOrBlossom[[1]] === "Blossom",
    If[trQ,
     AppendTo[blossomBases, augOrBlossom[[2, 2, 1]]];
    ];
    shrunken = ShrinkFlower[GadjAug, mate, augOrBlossom[[2]]];
    newPath = AugmentingPathRecursive[shrunken[[1]], shrunken[[2]], opts];
    newPath = ExpandBlossomsAndChangeMatching[shrunken[[1]], shrunken[[2]], newPath];

    If[trQ,
     If[newPath ≠ {},
      AppendTo[listOfGraphs, GadjAug];
      AppendTo[listOfEvents, "AugmentingPath"];
      AppendTo[listOfBlosAndAPs, Partition[newPath, 2, 1]];
      AppendTo[blossomBases, augOrBlossom[[2, 2, 1]]];
      AppendTo[listOfExtraBlossoms, augOrBlossom[[2, 2]]];
      AppendTo[listOfMatchings, mate]
     ]
    ]
   ];
   newPath
  ];


Options[MaximumMatchingMK] = {TraceQ → False, MaxRecursion → ∞};




(* Primary Function *)
MaximumMatchingMK[G_, opts___] := Module[
   {Graph, matching = Table[0, {V[G]}], augPath = {}, oldMatching = {}, matchingList, trQ},
   Graph = G;
   count = 0;
   {maxrec, trQ} = {MaxRecursion, TraceQ} /. {opts} /. Options[MaximumMatchingMK];
   $RecursionLimit = ∞;
   (* Use this for visualization *)
   If[trQ,
    Clear[listOfGraphs, listOfMatchings, listOfEvents,
     listOfBlosAndAPs, blossomBases, numAugPaths, listOfExtraBlossoms];
    listOfGraphs = listOfMatchings = listOfEvents =
       listOfBlosAndAPs = listOfExtraBlossoms = blossomBases = {};
    numAugPaths = 0
   ];

   (* Comment the lines from matchingList=MaximalMatchingSW[Graph];
   to Do[matching[[e]]=Reverse[e],{e,matchingList}];
   below to remove the initial maximal matching *)
   matchingList = MaximalMatchingSW[Graph];
   If[2 Length[matchingList] + 1 >= V[G],
    If[trQ, Print[StringForm["Total number of recursion calls: ``", count]]];
    Return[matchingList]];
   mPrint[Length[matchingList]];
   Do[matching[[e]] = Reverse[e], {e, matchingList}];
   mPrint["startingAugmentation"];
   augPath = AugmentingPath[Graph, matching, opts];
```

```
    mPrint[{"first augmentation done", 1 / 2 Count[matching, x_ /; x > 0]}];

    If[augPath == {},
     Do[If[i ≠ 0,
       If[! MemberQ[oldMatching, {matching[[i]], i}],
         AppendTo[oldMatching, {i, matching[[i]]}]]];
      ], {i, matching}];
     If[trQ, Print[StringForm["Total number of recursion calls: ``", count]]];
     Return[oldMatching],
     Do[If[EvenQ[i],
       matching[[augPath[[i]]]] = augPath[[i - 1]];
       matching[[augPath[[i - 1]]]] = augPath[[i]]] , {i, 1, Length[augPath]}]
    ];
    While[True,
     mPrint[{"startingAugmentation", matching}];
     augPath = AugmentingPath[Graph, matching, opts];

     (* If we get stuck, end *)
     If[augPath == {},
      Break[]
     ];

     (* Change matching*)
     Do[If[EvenQ[i],
       matching[[augPath[[i]]]] = augPath[[i - 1]];
       matching[[augPath[[i - 1]]]] = augPath[[i]]] , {i, 1, Length[augPath]}]
    ];

    (* Check if matching is maximum, if it is, return it *)
    If[Count[matching, r_ /; r > 0] + 1 >= V[G],
     If[trQ, Print[StringForm["Total number of recursion calls: ``", count]]];
     Return[DeleteDuplicates[
       Sort /@ DeleteCases[Table[{i, matching[[i]]}, {i, V[G]}], {_, 0}]]]]];
 mPrint[{"augmentation done", augPath, matching, 1 / 2 Count[matching, x_ /; x > 0]}];

    Do[If[i ≠ 0,
      If[! MemberQ[oldMatching, {matching[[i]], i}],
        AppendTo[oldMatching, {i, matching[[i]]}]]];
     ], {i, matching}];
    If[trQ, Print[StringForm["Total number of recursion calls: ``", count]]];
    $RecursionLimit = 256;
    oldMatching
   ];
```

- **Blossom Plot**

```
(* NOTE: To run, the option TraceQ → True must be included *)
BlossomPlot[G_, opts___] := Module[{aGraph, graphToShow, matching, event,
    listOfInterest, color, blosOrAP, im, locs, blosBase, gAdjTotal, extraBlossom},
  MaximumMatchingMK[G, TraceQ → True];
  If[count == 0, Return["Cannot show graph, no recursive calls"]];
  (*im=GraphPlot[G, Method→"SpringEmbedding", VertexLabeling→True];
  locs=VertexCoordinateRules/.
    Cases[InputForm[ im], HoldPattern[VertexCoordinateRules→ _],∞];*)
  locs = First /@ G[[2]];
  im = GraphPlot[G, VertexCoordinateRules → locs, VertexLabeling → True];
  Manipulate[
   aGraph = listOfGraphs[[g]];
   matching = listOfMatchings[[g]];
   event = listOfEvents[[g]];
   blosOrAP = listOfBlosAndAPs[[g]];
   blosBase = blossomBases[[g]];
   extraBlossom = listOfExtraBlossoms[[g]];
   graphToShow = FromAdjacencyLists[aGraph[[1 ;; -2]]];
   If[event == "Blossom", listOfInterest = blosOrAP[[2]]; color = Lighter@Lighter@Blue];
   If[event == "AugmentingPath", listOfInterest = blosOrAP; color = Yellow];
   If[event == "Stuck", listOfInterest = {}; color = Yellow];
   Column[{Style[StringForm["Event = ``, `` edges in matching",
        event, Length[Select[matching, #1 ≠ 0 &]] / 2], FontFamily → Times],
      Show[
       (*GraphPlot[graphToShow,
        VertexCoordinateRules→ Thread[Range[V[G]] → locs],EdgeRenderingFunction→
         ({If[event=="AugmentingPath"&&(MemberQ[listOfInterest,#2]||MemberQ[listOfInterest,
                Reverse[#2]]),Lighter@Green,White],AbsoluteThickness[8],Line[#1]}&)],*)
       Graphics[{If[event === "Blossom", {EdgeForm[{Thickness[.01], Darker@Blue}],
           FaceForm[Blue], Polygon[locs[[listOfInterest]]]}, {}]}],
       Graphics[{If[event === "AugmentingPath", {EdgeForm[{Thickness[.01], Darker@Blue}],
           FaceForm[Lighter@Blue], Polygon[locs[[extraBlossom]]]}, {}]}],
       GraphPlot[graphToShow, VertexCoordinateRules → Thread[Range[V[G]] → locs],
        EdgeRenderingFunction →

         ({If[event === "AugmentingPath" &&
              (MemberQ[listOfInterest, #2] || MemberQ[listOfInterest, Reverse[#2]]),
             {Lighter@Green, Thickness[.02], Line[#1]}, {}]} &)],
       GraphPlot[graphToShow,
        VertexCoordinateRules → Thread[Range[V[G]] → locs],
        EdgeRenderingFunction → ({If[matching[[#2[[1]]]] == #2[[2]],
             If[event === "Stuck", Orange, Red], Black], AbsoluteThickness[3], Line[#1]} &),
        VertexRenderingFunction → ({
            If[#2 == blosBase && (event === "Blossom" || event === "AugmentingPath"),
             {PointSize[Large], EdgeForm[{Black, Thickness[0.0004]}],
              FaceForm[White; Yellow], Disk[#1, .032], Text[Style["B", Bold], #1]},
             If[MemberQ[listOfInterest, #2],

              {PointSize[Large], EdgeForm[{Black, Thickness[0.0004]}],
               FaceForm@color, Disk[#1, .035], Text[Style["B", Bold], #1]}; {},

              {PointSize[Large], EdgeForm[{Black, Thickness[0.0004]}],
               FaceForm@Yellow, Disk[#1, .035], Text[Style["B", Bold], #1]}; {}]]
            } &)], Sequence @@ FilterRules[{opts}, Options[Graphics]],
        PlotLabel → None, PlotRangePadding → 0,
       Frame → ! True, FrameTicks → True, ImageSize → 570]}],
   {g, 1, Length[listOfGraphs], 1, ControlType → Automatic, Appearance -> "Open"},
   TrackedSymbols → True, SaveDefinitions → ! True]]
```