```
Maximum weight matching in bipartite graphs: implementation notes.

General strategy:  Input: A complete bipartite graph with |Outer| = |Inner|.

1. Find a feasible labeling: (Outer: max weight of incident edges, Inner: 0)
   For each outer node u do L(u) <-- max(w(u,v)) over (u,v) in u.Adj.
   For each inner node v do L(v) <-- 0.

2. Compute zero graph Z and a maximum matching of Z:
   Create a graph Z with the vertex set of G.
   for each edge (u,v) in G, if L(u) + L(v) = w(u,v), add (u,v) to Z.
   Find a maximum cardinality matching M in Z.
   If M is a perfect matching, output it as a maximum weight matching of G.

3. If M is not a perfect matching, do the following:
   Take a tree T from the Hungarian forest output by maximum matching.
   T has more outer nodes than inner nodes.  Adjust labels by decreasing
   labels of T’s outer nodes by Delta and increasing labels of its
   inner nodes by Delta.  For edges of G (not Z) that connect an outer node
   of T to a node outside T, the slack of these edges, which is
   the sum of labels at the ends of an edge minus its weight,
   is decreasing.  Choose Delta to be minimum of the slacks of such edges.
   Adjust the labels of the nodes of T and go back to Step 2.

   Details:
   for (u,v) in E with u=outer node in T, v=inner node not in T:
        slack(u,v) = L(u) + L(v) - w(u,v)
   Let Delta = min(slack(e)), over all such edges.
   for each outer node u in T do L(u) -= Delta
   for each inner node v in T do L(v) += Delta

   The zero graph Z changes after the labels are updated. Go back to Step 2.

4. Stop when the maximum matching of Z satisfies the exit criterion
   (that it is a perfect matching).
```

```
Problems with the general strategy:

For small graphs, this algorithm works well.  There are 2 things that the
algorithm needs that make it unusable when |V| is large.  First, it needs
unnecessary padding of extra nodes to ensure that the number of nodes
on each side is equal.  Second, missing edges are added to G (with zero
weight) to make it a complete bipartite graph.  So, it has |V|^2/4 edges.
For |V| = 2 million, it has 10^12 (one trillion) edges.  Allocating tera
bytes of memory is infeasible.

How to implement the algorithm without increasing the size of G:
In the following discussion, graph G is an arbitrary bipartite graph.
The number of inner nodes may not be equal to the number of outer nodes.

1. During the algorithm, we need to ensure that L(u), the label of u,
   is nonnegative for every node u in V.  Therefore, Delta is not only
   restricted by the slack of some edges, but it is also limited to the
   minimum label of an outer node of T.  Reducing its label by a larger
   amount will make its label negative, which is not allowed.

2. At the end of the algorithm, instead of getting a perfect matching,
   we can stop if all free outer nodes have zero weight.

3. When growing alternate trees, we normally start from free nodes.
   Now, we start growing the tree only if a node is free and it has
   positive label, i.e., u.mate = null and L(u) > 0.

4. Normally, augmenting path is found only when a free inner node is
   added to the alternating tree.  Now, we will also claim an augmenting
   path if an outer node u with zero label is added to the tree.
   Such a path will have even length and exchanging its matched and free
   edges will yield a matching of the same size, but in the new matching,
   u is not matched and since L(u)=0, it does not have to be matched.

5. Start growing alternating tree from all free outer nodes (outer nodes
   that are not matched, having positive labels) at the same time and
   augment a set of disjoint augmenting paths obtained in one iteration.
   This step saves a factor of |V| from the running time.
```