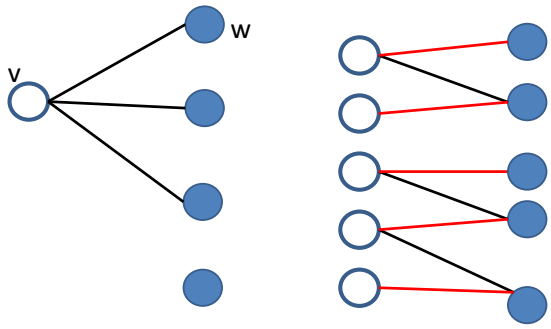
In some situation finding approximately optimal solution to a maximization or minimization problem is good enough in the application. This may be useful if we either lack all the information as we have to make decisions (online algorithms) or we don't have the time to find a fully optimal solution (say as the problem is too hard). In the remaining part of the course, we will explore algorithms for such situations.

1 Online Matching

We start by an online version of the bipartite matching problem. The bipartite matching problem is given by two sets of nodes X and Y, and a set of edges $E \subset X \times Y$, and the problem is to find a matching $M \subset E$ of maximum size. We have seen how to do this in polynomial time using a flow algorithm. In this section, we will consider a version of this problem, where Y is given upfront, nodes $x \in X$ appear one-at-a time, as a node $v \in X$ appears, we are also informed about all its adjacent edges in E, and we need to make a decision to add an edges to M adjacent to v (or not) without knowledge of further nodes that will show up later. So for example, in the graph on the left of the figure below, when the first node v shows up, there seems to be no good way to decide which node to match it on the opposite side. However, once we match v, say using the edge (v, w), there will be no way to change this edge in the matching, we will not be able to use augmenting path to update our matching. A typical application of this style of matching problem, is matching advertisement to web pages. One side of the matching are web users, the other side are advertisements we may want to show the users. When a user v shows up online, we may have an opportunity to show many possible ads, and we need to decide instantly, which add to show the person.



Under these circumstances, we cannot expect to be able to find the true maximum matching. We will be analyzing here the simple greedy algorithm: select a edge adjacent to v that can be added to the matching, whenever such an edge is possible. This greedy algorithm is not optimal. For example, if we use (v, w) in the example above, and the next node is v' with a single edge (v', w) adjacent, then we could have selected a matching of size 2 (had we matched v to a different adjacent node), but unfortunately, our resulting matching is just one edge. However, we can say something positive about the matching the algorithm creates.

We say that the algorithm is a 2-approximation algorithm is the matching it finds has size at least $1/2$ of the maximum possible. Note that for this argument, we need to show a bound on the size of the

maximum matching, and we'll do this without computing the maximum matching explicitly.

Claim. The above online matching algorithm is a 2-approximation, find a matching M is size at least $1/2$ of the maximum possible matching.

Proof. Consider the matching M and the the maximum matching M^*. Let Y be the side that is given, and nodes on the Y side arrive one-at-a-time. Consider the graph just consisting of edges M and M^*, an example of which is shown on the right of the figure, where red edges are edges in M^* and black edges are edges in M. Each component of this graph is a path alternating between M and M^* edges. Consider a node $v \in X$ matched by M^* by an edge (v, w). Either v is also matched in M (to possibly a different node), or w is matched in M, as otherwise the edge (v, w) would have been an option to add when v appeared. So any edge $(v, w) \in M^*$ is adjacent to at least one edge in M, and an edge in M can only be adjacent to at most 2 edges in M^* proving that $|M^*| \leq 2|M|$ as claimed.

2 Knapsack

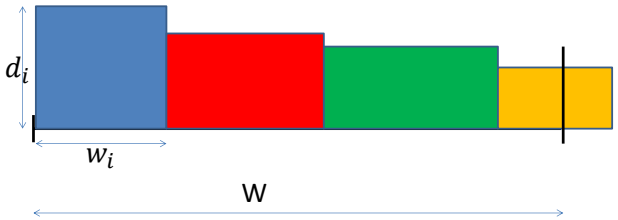
Next we consider the KNAPSACK problem. This problem is given by n items each with a weight w_i and value v_i, and a maximum allowed weight W. The problem is to selected a subset I of items of maximum total value $\sum_{i \in I} v_i$ whose total weight $\sum_{i \in I} w_i \leq W$ is at most the weight given.

First notice that the SUBSET SUM problem is a special case of this problem, when $v_i = w_i$ for all i, and we ask the question if these is a subset with total value (and hence weight) at least W. This shows that KNAPSACK is NP-hard, at least as hard as an NP-complete problem (not NP-complete, as its an optimization problem, and hence not in NP).

Here we give a simple 2-approximation algorithm for the problem. See also Section 11.8 of the book that we'll cover later that gives a much better approximation algorithm. Our simple approximation algorithm will be a form of greedy algorithm. We will consider two greedy ideas. For both algorithms, we first delete all items with weight $w_i > W$ as they cannot be considered by any solution.

(a) Consider items in order of their value, and add to the set I till the weight is not exceeded.

While this algorithm is natural, it is also easy to give a very bad example for this. Say one item has eight $w_n = W$ and value $w_n = W$, and many small items of value $v_i = W - 1$ and weight $v_i = 1$. The item of maximum value is item n which alone fills the knapsack. However, by taking many of the other items, we can take as many as $\min(n - 1, W)$ items each of value $v_i = (W - 1)$ reaching a much higher total value.



This leads us to a different greedy idea: we should consider the density of value in each item $d_i = v_i/w_i$. Our second greedy algorithm considers items in this order

(b) Consider items in order of their value density d_i, and add to the set I till the weight is not exceeded.

The process can we well represented by a figure, if we have the items each represented by a box of length w_i and height d_i where now the area of the box is $w_i d_i = v_i$ is its value. Items are added to the knapsack in the order decreasing density, as suggested by the figure, where the orange item hanging out on the right of the W size knapsack is the first item that didn't fit into the knapsack.

Unfortunately, this greedy algorithm can also be very bad. For an example of this situation, all we need is two items $w_1 = v_1 = W$, while $v_2 = 1 + \epsilon$ and $w_2 = 1$. Now item 1 has density $d_1 + 1$, while item 2 has density $d_2 + 1 + \epsilon$. However, after fitting item 1 in he knapsack, item 2 no longer fits! So we end up with a total value of value $1 + \epsilon$, while value W was also possible.

Interestingly, the better of the two greedy algorithm is a good approximation algorithm.

Claim. Running both (a) and (b) greedy algorithm above, and taking the solution of higher value is a 2-approximation algorithm, finding a solution to the knapsack problem with at least 1/2 of the maximum possible value.

Proof. Consider the two greedy algorithms, and let V_a and V_b the value achieved by greedy algorithms (a) and (b) respectively, and let Opt denote the maximum possible value. For algorithm (b) let I be the set of items packed into the knapsack, and let j be the first item that didn't fit into the knapsack (the orange item in the example). Clearly $\sum_{i \in I} v_i = V_b$ and $v_j \leq V_a$, as algorithm (a) starts by taking the single item of maximum value.

As before, the main issue we need to consider is how we can bound the optimum value, without having to compute it. We'll show below that $\sum_{i \in I} v_i + v_j \geq Opt$, implying that $V_a + V_b \geq Opt$, so the larger of V_a and V_b must be at least 1/2 of OPT.

Claim. Using the notation from algorithm (b) above, $\sum_{i \in I} v_i + v_j \geq Opt$.

Proof. The main observation is that if we could cut a part of the last item so as to exactly fill the knapsack, that would clearly be the optimum solution: it uses all items of density $> d_j$ and fills the remaining part of the knapsack with value density d_j. This shows that the optimum value is bounded by $\sum_{i \in I} v_i$ plus a fraction of the last item.