Aviad Rubinstein
Date: December 4, 2018

1 Coping with NP-hardness

Today we consider two problems (Max-Cut and Vertex Cover) that are NP-hard. We do not formally cover NP-hard in this course (see CS103, CS154, etc.); for our purposes it suffices that these are problems for which we are extremely unlikely to find an efficient algorithm.

At a high level, there are three approaches for dealing with NP-hardness (and other hardness):

Beyond worst-case Even if worst case inputs of the problems are NP-hard, we can try to design algorithms that work on some plausible subclass of inputs. When we talked about dynamic programming we saw an example where Independent Set (which is NP-hard in general) can be solved efficiently on trees. The Max-Cut problem can be solved efficiently on graphs that are "planar"[1].

Approximation algorithms While finding an optimum to an optimization problem like Max-Cut or Vertex Cover can be NP-hard, we will see in this lecture how to efficiently find solutions that are only moderately worse than the optimum.

Exponential time algorithms Sometimes, it is reasonable to have an algorithm that runs in exponential time, provided that the parameter in the exponent is relatively small. In the dynamic programming lecture, we saw algorithms for the (0/1 and unbounded) knapsack problem, which is NP-hard; our algorithms ran in time proportional to the capacity of the knapsack which may be exponential in the length of the input, but is often not prohibitively large.

1.1 Approximation algorithms

Suppose that we have a maximization problem P of the form $\max_x f(x)$, where x is subject to some constraints. Suppose that P has an optimal solution x^* with value $OPT = f(x^*)$. Then, an α-approximation algorithm for P has to return a feasible solution y with value $ALG = f(y) \geq \alpha \cdot OPT$. (Note that wlog we always have $f(y) \leq f(x^*)$ because we assume that x^* is optimum.)

Similarly, if P is a minimization problem $\max_x f(x)$with optimum solution x^*, we require that an α-approximation algorithm returns y such that $f(y) \leq \alpha f(x^*)$.

Note that for maximization problems we always have approximation factor $\alpha \leq 1$, and for minimization problems $\alpha \geq 1$; for both, we want α to be as close to 1 (exact algorithm) as possible.

2 Max-Cut

The input to the Max-Cut problem is an undirected graph $G = (V, E)$. The output is a subset of vertices $S^* \subset V$ that maximizes the number of edges from S^* to the rest of the graph $(V \setminus S^*)$:

$$S^* = \arg\max_{S \subseteq V} |E \cap (S \times (V \setminus S))|.$$

We will see two approximation algorithms for this problem: greedy and random.

[1] A graph is *planar* if it can be drawn such that edges intersect only at their endpoints; for example, a road network without bridges.

2.1 Greedy Max-Cut

The greedy algorithm for Max-Cut iteratively grows two disjoint sets $S, T \subseteq V$ until $S \cup T = V$. At each iteration, we consider a new vertex, and add it to the set that would maximize the number of new edges from S to T.

Algorithm 1: GreedyMaxCut(G)

$S, T \leftarrow \emptyset$.
foreach $v \in V$ **do**
  if *(# of edges from v to S) > (# of edges from v to T)* **then**
    Add v to T
  else
    Add v to S
Return (S, T).

2.1.1 Analysis of Greedy Max-Cut

Since the Max-Cut problem is NP-hard, we don't expect the algorithm to be exactly correct. But we will prove that it is approximately correct:

Claim 1. *Algorithm GreedyMaxCut is a $1/2$-approximation algorithm for the Max-Cut problem.*

Proof. Let S_v, T_v denote the sets S, T held by the algorithm when it considers vertex v; let E_v denote the set of edges between v and $S_v \cup T_v$. Notice that every edge in the graph belongs to exactly one E_v, hence $|E| = \sum_v |E_v|$.

At each iteration, the size of the cut increases by at least $|E_v|/2$. Therefore the final size of the cut is at least $\sum_v |E_v|/2 = |E|/2$. In particular, the size of the cut returned by GreedyMaxCut is at least half of the maximum cut. □

2.1.2 Running time

$O(|V| + |E|)$

Can we have an even faster approximation algorithm? (Notice that just reading the input already takes $O(|V| + |E|)$ time...)

2.2 Randomized Max-Cut

It turns out that there is an even faster and even simpler approximation algorithm: choose a uniformly random cut! If we're unlucky, the cut will be really small; but we will show that in expectation the cut contains exactly half of the edges.

Algorithm 2: RandomCut(G)

$S, T \leftarrow \emptyset$
foreach $v \in V$ **do**
  $b \leftarrow$ uniformly random bit **if** $b = 1$ **then**
    Add v to T
  else
    Add v to S
Return (S, T).

2.2.1 Analysis of Randomized Max-Cut

Claim 2. *Algorithm RandomCut is a 1/2-approximation-in-expectation algorithm for the Max-Cut problem.*

Proof. For each edge $(u, v) \in E$, there are two ways that (u, v) can cross the cut: $u \in S, v \in T$ and $u \in T, v \in S$; and there are two ways that can not cross the cut: $u, v \in S$ and $u, v \in T$. Thus the probability that (u, v) crosses the cut is $1/2$. Therefore the expected number of edges that cross the cut is $|E|/2$. □

3 Vertex Cover

The input to the Vertex Cover problem is an undirected graph $G = (V, E)$. The output is a subset of vertices $S \subset V$ such that every edge $(u, v) \in E$ is *covered* by S, i.e. at least one of u, v is in S. Of course, $S = V$ is a valid solution, but we would like the algorithm to output a minimum one.

Vertex Cover is also NP-hard, so we will look for a good approximation algorithm.

3.1 Attempt 1: Random-Greedy Vertex Cover

It is tempting to try to use a greedy algorithm to solve Vertex Cover: iteratively pick a vertex v, add it to S, and then remove v together with all the edges covered by v from the graph.

The above description is really more of a template or a meta-algorithm, because we didn't specify how to pick the vertex v at each iteration. Let's first consider what happens when we pick the vertices at random.

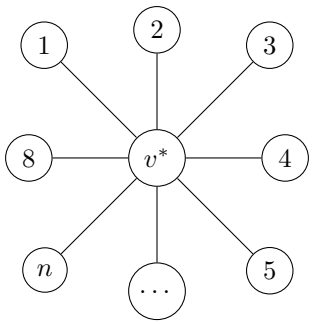
Algorithm 3: RandomGreedyVC(G)

```
S ← ∅.
while E ≠ ∅ do
    v ∼ V (uniform random vertex)
    foreach (v, u) ∈ E do
        E ← E \ {(v, u)}
    S ← S ∪ {v*}
    V ← V \ {v}
Return S.
```

Claim 3. *Algorithm RandomGreedyVC is an $\Omega(n)$-in-expectation-approximation algorithm.*

This is really bad - approximately as bad as just taking all the vertices! Since this is a worst-case approximation guarantee, to prove the claim it suffices to construct one bad example.

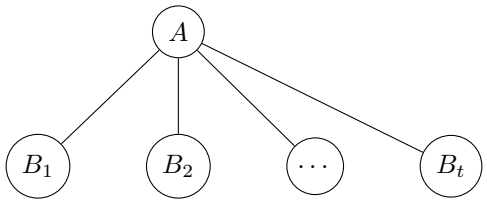
Example 1. Consider the $n + 1$-vertex star graph: it has a special vertex v^* that is connected to n other vertices, and there are no edges between the other vertices.



The optimal vertex cover is $S = \{v^*\}$, which has only one vertex.

In contrast, the greedy algorithm will, in expectation, see $n/2$ other vertices before reaching v^*, but it will only terminate after adding v^*. Hence in expectation it will have to use $n/2 + 1$ vertices in the cover.

Figure 1: High level overview of Example 2.



3.2 Attempt 2: Degree-Vertex Cover

Let's modify our above template to make more intelligent choices when picking the vertex v. When the algorithm considers v, it removes all the incident edges from the graph. Thus we make the most progress if we pick a vertex with maximal degree. So let's do that.

Algorithm 4: DegreeGreedyVC(G)

```
S ← ∅.
while E ≠ ∅ do
    v* ← arg max_{v∈V} deg(v)
    foreach (v*, u) ∈ E do
        E ← E \ {(v*, u)}
    S ← S ∪ {v*}
    V ← V \ {v*}
Return S.
```

How good is the approximation now? It turns out this algorithm gives a much better approximation than RandomGreedyVC, but it's still not as good as we'd like:

Claim 4. *Algorithm DegreeGreedyVC is an* $\Omega(\log(n))$*-approximation algorithm.*

This ratio is actually tight, but here will only prove that it is no better than $\log(n)$, and again we'll do that by example.

Example 2. We'll construct a bipartite graph, the vertices are partitioned into sets A and B, and there are no edges with two endpoints from A (respectively two endpoints from B). Notice that A and B are both valid solutions to the Vertex Cover problem. We will construct an instance where A is much smaller than B, but the Algorithm DegreeGreedyVC selects B.

Set A has 2^t vertices; each vertex in A will have degree $2^t - 2$. We further partition set B into t subsets $B_1, \ldots, B_t$, each of $2^{t-1} = 2^t/2$ vertices. Notice that in total $|B| = t \cdot 2^t/2 = \frac{t}{2}|A|$.

The vertices in set B_i have degree 2^i; The vertices in B_t are connected to all the vertices in A. The first half of the vertices in B_{t-1} are connected to the first half of the vertices in A, and the second half are connected to the second half. $B_{t-2}, B_{t-3}, \ldots$ are defined analogously, until the B_1 that are each connected to two vertices from A.

Let's consider the run of DegreeGreedyVC on our example: At first, it will pick all the vertices from B_t that have maximal degree 2^t. After processing all the vertices from B_t and removing their edges, each vertex in A lost 2^{t-1} edges; its new degree is now $2^{t-1} - 2$. Hence now the algorithm will proceed to the vertices in B_{t-1}, etc.

3.3 Attempt 3: the wasteful algorithm

The moral from the previous example is that when we look at an edge (u, v) we know that one of its endpoints should belong to the vertex cover, but its really hard to know which one.

Figure 2: Example 2: only A and B_1.

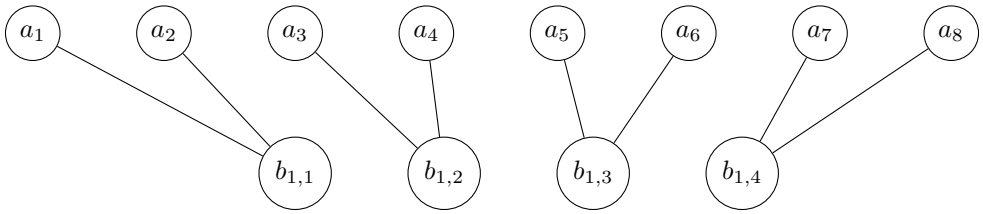
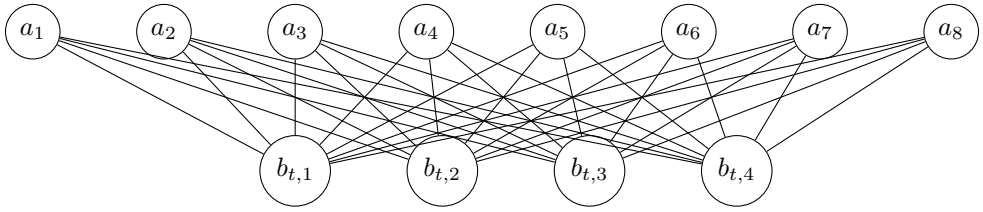


Figure 3: Example 2: only A and B_t.



Here is an idea: since we don't know which endpoint to take, take both!

Algorithm 5: BothEndpoints(G)

$S \leftarrow \emptyset$.
while $\exists (u, v) \in E$ **do**
  foreach $(u, w) \in E$ **do**
    $E \leftarrow E \setminus \{(u, w)\}$
  foreach $(v, w) \in E$ **do**
    $E \leftarrow E \setminus \{(v, w)\}$
  $S \leftarrow S \cup \{u, v\}$
  $V \leftarrow V \setminus \{u, v\}$
Return S.

Taking both endpoints seems wasteful, but it turns out that Algorithm BothEndpoints is the best approximation algorithm we know for Vertex Cover (moreover, it is conjectured to be optimal!)

Claim 5. *Algorithm BothEndpoints is a* 2*-approximation for Vertex Cover*

Proof. Suppose the algorithm finishes after s iterations, and let $(u_1, v_1), (u_2, v_2), \ldots (u_s, v_s)$ be the sequence of edges considered by the algorithm. Any vertex cover for the entire graph in particular has to cover those edges. Since every time we consider an edge we remove both of its endpoints, none of those edges share any endpoints. Therefore any vertex cover for those edges has to include at least one of U_i, v_i for each i. So any vertex cover has at least s vertices. But the set S returned by the algorithm has only $2s$ vertices, hence it is a 2-approximation to the optimum. □