Approximation Algorithms

March Madness is (NP-)Hard (draft)

David Liben-Nowell, Moses Liskov, Chris Peikert, Abhi Shelat, Adam Smith, and Grant Wang

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
{dln,mliskov,cpeikert,abhi,asmith,gjw}@theory.lcs.mit.edu

Abstract. We formally define the MARCH-MADNESS decision problem (inspired by popular betting pools for the NCAA basketball tournament), and prove it NP-complete.

1 Introduction

The National Collegiate Athletic Association (NCAA) Basketball Tournament[3], held every March, is a tournament among the top 64 (or more recently, 65) collegiate basketball teams in the United States. The tournament is set up in single-elimination form: to start, each team occupies the leaf of a complete binary tree. Two teams occupying sibling nodes play against each other, and the winner moves up to occupy its parent node, while the loser is out of the tournament. This process continues until only one team (occupying the root node) remains, and that team is crowned the national champion.

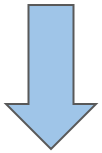
Popular distractions during the tournament are so-called "March Madness" pools, often organized among friends and co-workers. Each participant in a pool

NP-Completeness Retrospective

Skills learned:

- Recognizing provably "hard" problems (save time by not trying to find a fast algorithm)
- Converting a problem into a different problem (useful not only for hardness proofs, but also algorithm design)

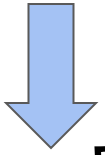
TODAY



Search Problems

vs.

"Size" Problems



vs.

Decision Problems

See lecture 14:
Intro to Complexity
(Used binary search)

Find a max clique

What is the size of a max clique?

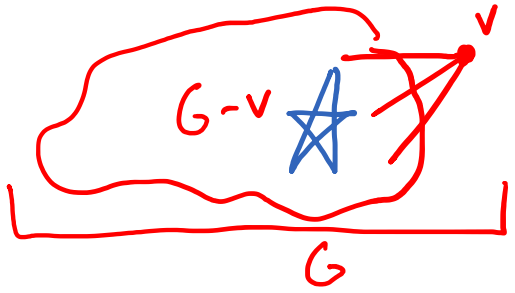
Is there a clique of size ≥k?

For all NP-complete problems,
if the decision version is in time T(n),
then the search version is in poly(T(n)) time.

(we won't prove, but we'll see examples)

Goal: Given an algorithm **size-clique** that returns the size of a max clique in time T(n), devise a poly(T(n))-time algorithm **find-clique** that returns a max clique.

Common Strategy: Go through each vertex and consider whether removing it changes the size of the solution.



Idea of **find-clique(G)**:

1. Call **size-clique(G)**
2. Pick an arbitrary vertex **v** and remove it (and its incident edges) to get G-v.
3. Call **size-clique(G-v)**
    a. If the answer <u>stayed the same</u>:

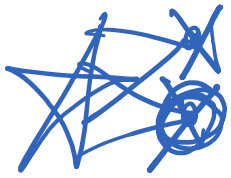
    There exists a max clique without v ⇒ **don't include v** in our clique

    a. If the answer <u>decreased by 1</u>:

    Every max clique contains v ⇒ **include v** in our clique

Goal: Given an algorithm **size-clique** that returns the size of a max clique in time T(n), devise a poly(T(n))-time algorithm **find-clique** that returns a max clique.

Common Strategy: Go through each vertex and consider whether removing it changes the size of the solution.



$O(n \cdot T(n)) + m$

Final Algorithm:

```
find-clique(G):
    k = size-clique(G)
    for each vertex v in G:
        if size-clique(G-v) = k:
            G = G - v        // permanently delete v and its incident edges
    return G
```

Goal: Given an algorithm **size-VC** that returns the size of a min VC in time T(n), devise a poly(T(n))-time algorithm **find-VC** that returns a min VC.

Common Strategy: Go through each vertex and consider whether removing it changes the size of the solution.

Idea of **find-VC(G)**:

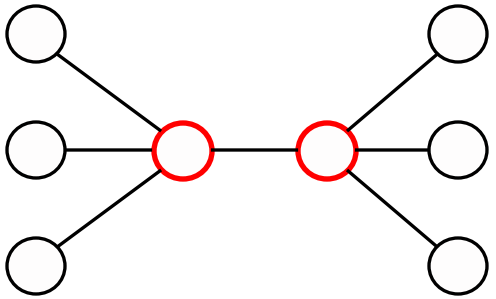
1. Call **size-VC(G)**
2. Pick an arbitrary vertex **v** and remove it (and its incident edges) to get G-v.
3. Call **size-VC(G-v)**
   a. If the answer stayed the same:

No min-VC solution contains v ⇒ do not add v to our solution.

   b. If the answer decreased by 1:

There exists a min-VC solution containing v ⇒ safely add v to our solution

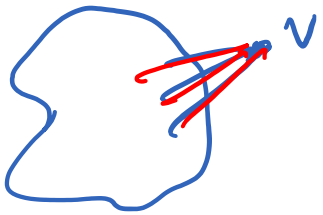
Reminder of VC: set S of vertices so that every edge has at least one endpoint in S



Goal: Given an algorithm **size-VC** that returns the size of a min VC in time T(n), devise a poly(T(n))-time algorithm **find-VC** that returns a min VC.

Common Strategy: Go through each vertex and consider whether removing it changes the size of the solution.

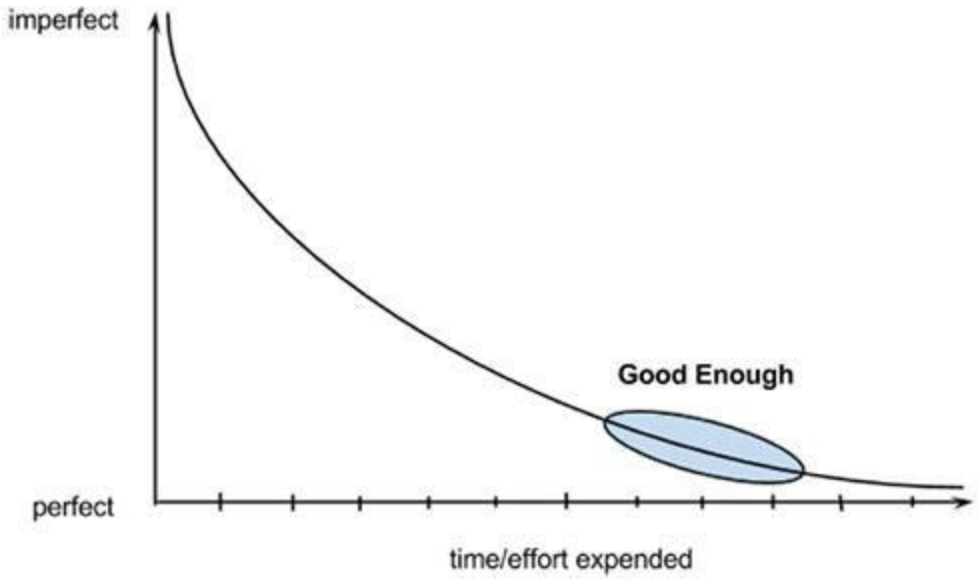
Final Algorithm:




```
find-VC(G):
    k = size-VC(G)
    for each vertex v in G:
        if size-VC(G-v) = k-1:
            add v to our VC
            G = G - v          // permanently delete v and its incident edges
            k = k-1            // need to add k-1 more vertices to VC
            if k = 0:  terminate
```

When we know we will not add v to the VC, why don't we permanently delete v?

Now on to Approximation Algorithms…



Approximating Minimum VC

An algorithm is an **α-*approximation*** for the VC problem if it returns a VC that contains *at most* α times as many vertices as a min VC.

$$OPT \le ALG \le \alpha \cdot OPT, \quad \alpha > 1$$

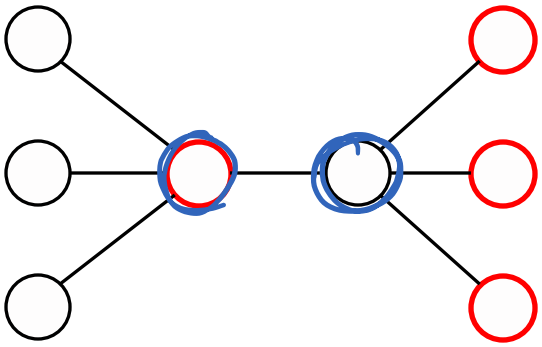
OPT: Optimal solution size

ALG: Solution size returned by our algorithm

α is called the ***approximation ratio*** (smaller is better here).

We will show that VC has a polynomial-time 2-approximation.

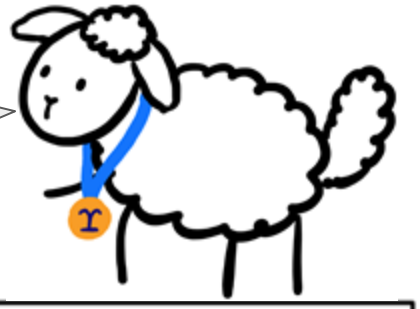
A 2-approx min-VC (optimum = 2)



Coffee Shop CEO said: "I'm ok with building at most twice as many stores as is optimal."

Approximating Minimum VC

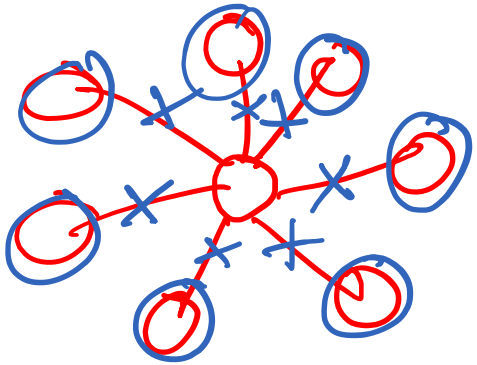
Check out my algorithm!
Pick an arbitrary vertex covering at least one edge, delete it, and repeat!



cover-and-remove(graph G):

1. $C \leftarrow \emptyset$
2. **while** G has an edge:
3. pick a vertex v covering *at least one edge*
4. $G \leftarrow G - v$; $C \leftarrow C \cup \{v\}$ // delete/add it to cover
5. **return** C

2-approx



Your task: How bad is the approximation ratio of the cover-and-remove algorithm?

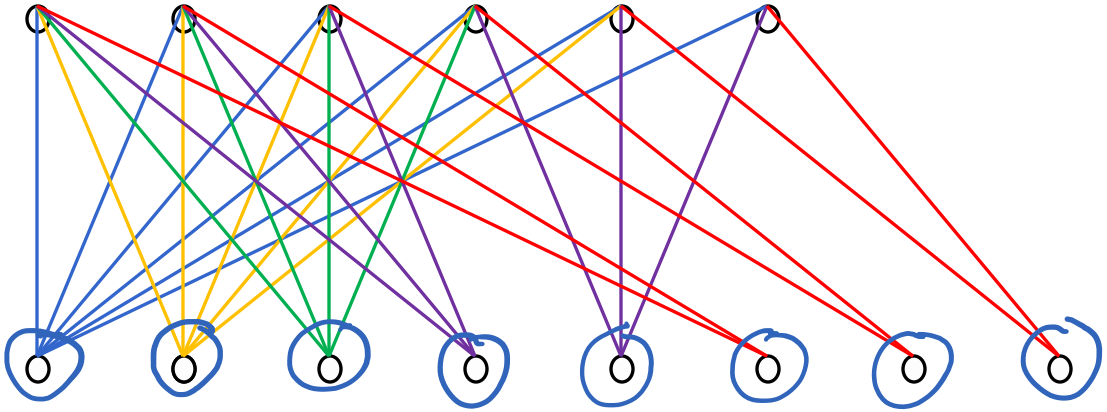
Construct a graph to support your claim

Approximating Minimum VC

I have another idea!
Pick the vertex covering the most edges!

greedy-cover-and-remove(graph G):

1. $C \leftarrow \emptyset$
2. **while** G has an edge:
3. pick a vertex v covering *the most edges*
4. $G \leftarrow G - v$; $C \leftarrow C \cup \{v\}$ // delete/add it to cover
5. **return** C



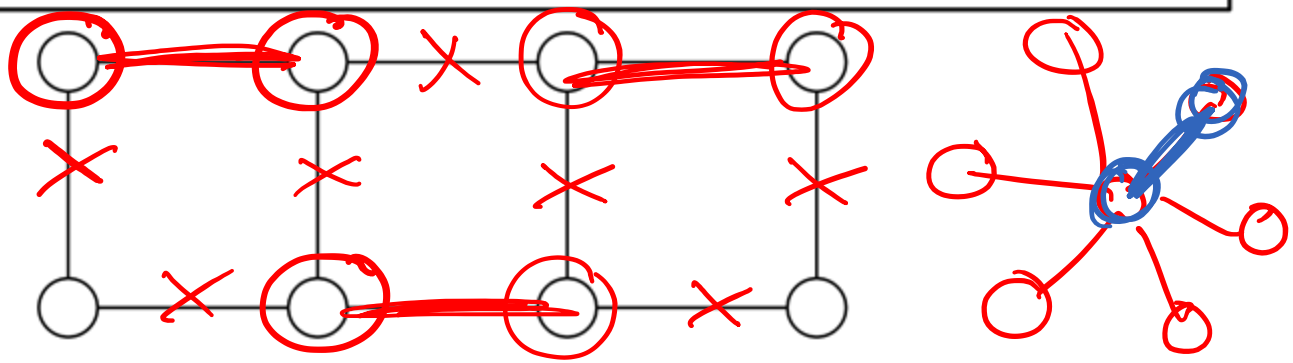
An extension of this idea shows that the approximation ratio is $\alpha = \Omega(\log n)$
(we won't prove)

Approximating Minimum VC

A seemingly-terrible-but-actually-good idea: Choose an arbitrary edge, add *both* endpoints to the VC, and delete endpoints (and incident edges).

double-cover(graph G):

1. $C \leftarrow \emptyset$
2. **while** G has an edge:
3. pick an edge $e = \{u, v\}$
4. $G \leftarrow G - \{u, v\}$; $C \leftarrow C \cup \{u, v\}$ // delete/add both endpoints
5. **return** C

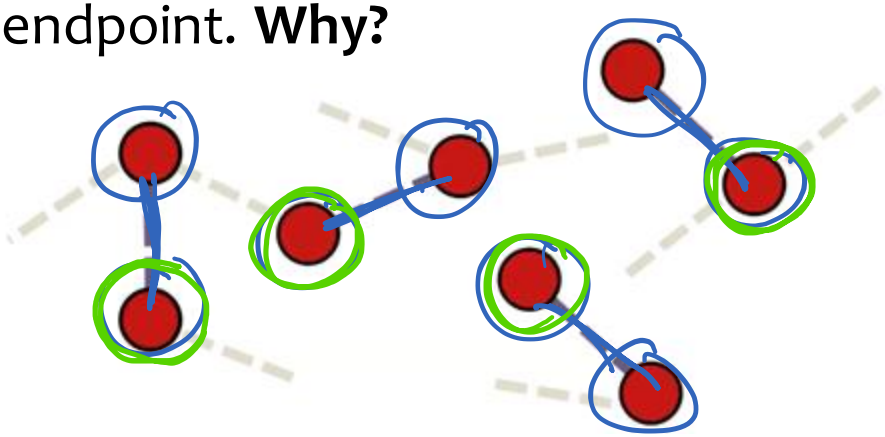


Theorem: double-cover obtains a 2-approx!

Approximating Minimum VC

A seemingly-terrible-but-actually-good idea: Choose an arbitrary edge, add *both* endpoints to the VC, and delete endpoints (and incident edges).

Observation about double-cover algorithm: None of the edges we choose share an endpoint. **Why?**



$O(m)$

Observation: ALG = **2** · (#edges chosen) $\leq 2 \cdot OPT$

Observation: OPT must circle at least one endpoint of each of our chosen edges. **Why?**

$\Rightarrow$ (# edges chosen) $\leq$ **OPT**

So **ALG** $\leq$ **2** · **OPT**

Can we do better?

- Can you get 1.99999-approximation in poly-time?
  - [Khot-Regev 03] Assuming some complexity hypothesis (Unique Games Conjecture), $(2-\epsilon)$-approximation NP-hard for any constant $\epsilon > 0$.
  - [Khot-Minzer-Safra 18] $(\sqrt{2}-\epsilon)$-approximation NP-hard.

Doesn't this mean every NP-complete problem has a 2-approximation since they're all reducible to each other?

No!

2 reasons:

1. Some problems are minimization, some are maximization, and some are neither.

   Minimization: $OPT \leq ALG \leq \alpha \cdot OPT$, $\alpha > 1$

   Maximization: $OPT \geq ALG \geq \alpha \cdot OPT$, $\alpha < 1$

   α is the *approximation ratio*

2. Reductions don't necessarily imply **anything** about approximation

   Consider the following example…

Last time we showed: An n-vertex graph G has a **VC of size ≤k** if and only if G has an **IS of size ≥n-k**.

(This can be used to show both VC ≤p IS and IS ≤p VC. Most reductions cannot be immediately reversed, but this one can since the graph doesn't change.)

E.g. Consider a graph G with **max-IS size n/2** and **min-VC size n/2**.

Running our **2-approx for VC** on G gives a **VC of size ≤n**, which translates to an **IS of size ≥n-n=0**.

So **IS-OPT = n/2**, **IS-ALG ≥ 0**, and the approximation ratio **α is infinite**.

Conclusion: Even though there's a poly-time mapping reduction showing IS ≤p VC, the 2-approximation algorithm for VC doesn't imply anything about approximation algorithms for IS.

NP-complete problems come in many types:

- Some can be approximated to within a constant factor
  - e.g. VC
- Some can only be approximated to within a larger factor
  - e.g. Set Cover has an O(log n)-approximation and there's no better approximation ratio unless P = NP
- Some have no non-trivial approximation at all unless P = NP
  - e.g. Clique and Independent Set
- Some can be approximated *arbitrarily* well
  - e.g. Knapsack