Approximation Algorithms and Victory Lap

CSE 421 Fall 2023
Lecture 26

A Linear Program

A linear program is defined by:

Real-valued **variables**

Subject to satisfying **everything** in a list of **linear constraints**

A linear constraint is a statement of the form: $\sum a_i x_i \leq c_i$
where a_i are constants, the x_i are variables and c_i is a constant.

Maximizing or minimizing a linear objective function

A linear objective function is a function of the form: $\sum b_i x_i$
where b_i are constants and the x_i are variables.

Solving LPs

A solution (or point) is a setting of all the variables

A **feasible point** is a point that satisfies all the constraints.

An **optimal point** is a point that is feasible and has at least as good of an objective value as every other feasible point.

There are polynomial time algorithms which return an optimal point of an LP in polynomial time.

But the optimal point might have fractional values.

A nicer example

Sometimes we can round fractional solutions into integral ones.

Minimum Weight Vertex Cover

We've seen how to solve the problem with DP on trees.

Let's try it now with linear programming.

A set S of vertices is a vertex cover if for every edge (u, v), u is in S, v is in S or both are in S.

Vertex Cover LP

Write an LP for finding the minimum weight vertex cover

A set S of vertices is a vertex cover if for every edge (u, v), u is in S, v is in S or both are in S.

What are your variables, then how do you constrain them?

Let $w(u)$ be the weight for a vertex u. You can treat $w(u)$ as a constant.

Vertex Cover LP

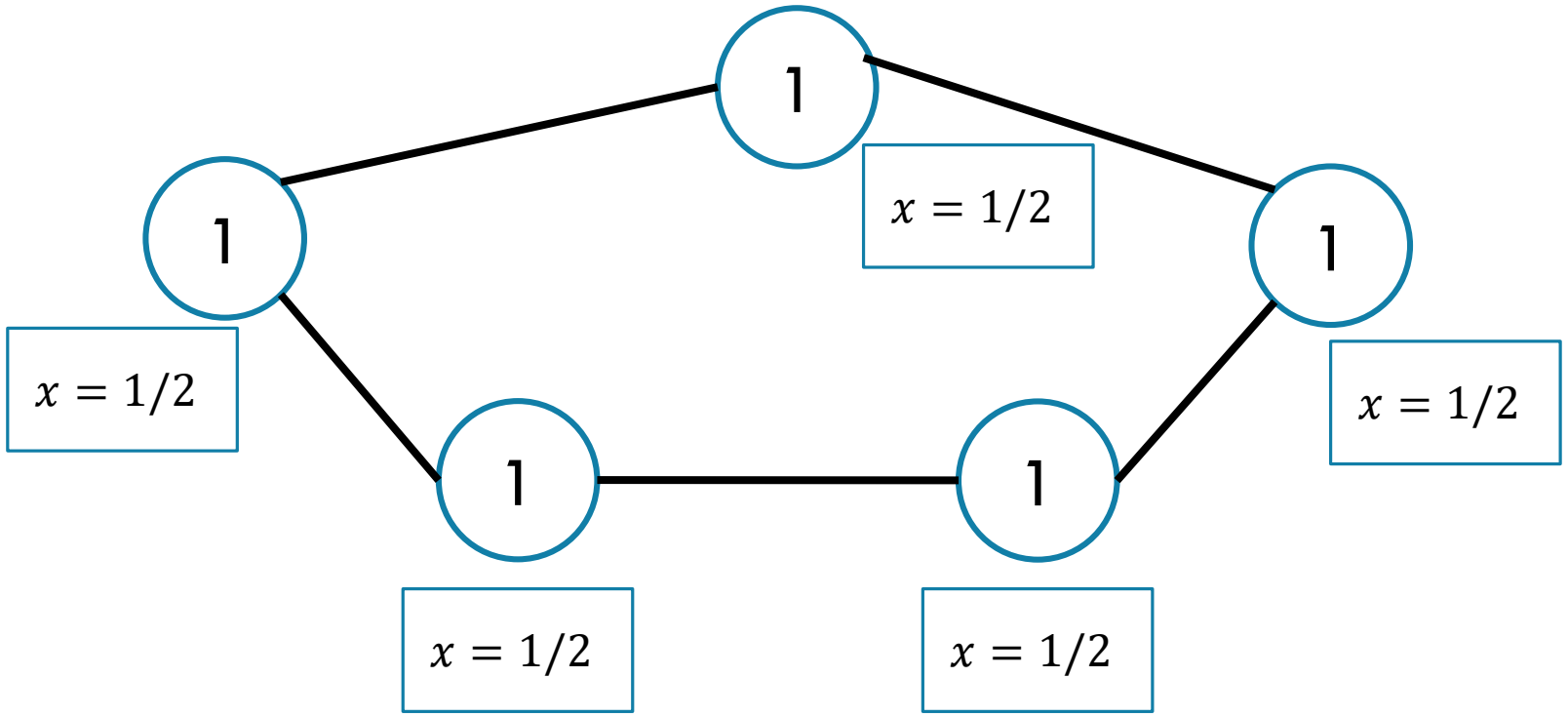
Minimize $\sum w(u) \cdot x_u$

Subject to:

$x_u + x_v \geq 1$ for all $(u, v) \in E$

$0 \leq x_u \leq 1$ for all u.

Non-Bipartite



The LP finds a fractional vertex cover of weight 2.5

There's no "real"/integral VC of weight 2.5. – lightest is weight 3.

There's a "gap" between integral and fractional solutions.

So, what if the graph isn't bipartite?

Big idea:

Just round!

Pollev.com/robbie

If $x_u \geq \frac{1}{2'}$ round up to 1.

If $x_u < \frac{1}{2}$, round down to 0

Minimize $\sum w(u) \cdot x_u$

Subject to:

$x_u + x_v \geq 1$ for all $(u, v) \in E$

$0 \leq x_u \leq 1$ for all u.

Two questions – is it a vertex cover? How far are we from the true minimum?

Is it a vertex cover?

Every edge was covered in the fractional matching

i.e. for every edge (u, v)

$x_u + x_v \geq 1$.

At least one of those is getting rounded up!

So every edge is covered.

And we've rounded to integers, so we have a "real" vertex cover.

How good of an approximation is it?

Well, we might have doubled the value of the LP when we rounded. But we definitely didn't do any more than that.

$2 \cdot LP \geq ALG$

And the value of the LP is definitely not bigger than the true size of the vertex cover (because otherwise the LP would have found that).

$OPT \geq LP$

Combining:

$2 \cdot OPT \geq ALG$

So we're safe in calling this a 2-approximation.

Comparing to the LP value

We did a weird thing on that last slide.

We were supposed to compare the value of our vertex cover to the best vertex cover.

But instead we compared it to the value of the LP…which we know isn't always the value of the vertex cover!

That wasn't laziness, it's a very common technique. We know very little about the true value of the vertex cover (if we knew what it looked like VERY VERY precisely, why couldn't we just write an algorithm to find it? We actually won't know much). So we start with what the algorithm gave us (that we do understand).

Side Note

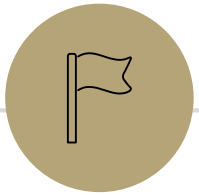
Could we do better?

Not just with the LP.

If you take a graph with n vertices and every possible edge, the LP's minimum is $n/2$, the true minimum vertex cover is size $n - 1$.

The ratio is $2 - 1/n$. So if we don't at least double the value **sometimes** we won't get a vertex cover at all.

Getting a 1.9999999 approximation is an open problem!



Another Approximation Algorithm

Recall: Finding an approximation for VC

For every edge, at least one of u,v is in the minimum vertex cover.

But instead of checking which of u, v a good idea to add, just add them both!

```
While(G still has edges)
     Choose any edge (u,v)
     Add u to VC, and v to VC
     Delete u v and any edges touching them
EndWhile
```

Does it work?

Do we find a vertex cover?

Is it close to the smallest one?

Does it run in polynomial time?

Do we find a vertex cover?

When we delete an edge, it is covered (because we added both u and v. And we only stop the algorithm when every edge has been deleted. So every edge is covered (i.e. we really have a vertex cover).

How big is it?

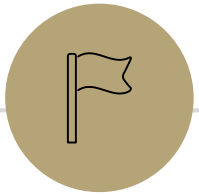
Let OPT be a **minimum** vertex cover.

Key idea: when we add u and v to our vertex cover (in the same step), at least one of u or v is in OPT.

Why? (u, v) was an edge! OPT covers (u, v) so at least one is in OPT.

So how big is our vertex cover? At most twice as big!

This is a 2-approximation for vertex cover!



Hard to approximate

Inapproximability

We can get pretty darn close to finding the best vertex cover.

Other problems, we can't get close at all. And importantly we know why.

For every $\varepsilon > 0$, there is not a poly-time approximation algorithm for MAX-CLIQUE with approximation ratio better than $O(n^{1-\varepsilon})$

Unless $\mathrm{P} = \mathrm{NP}$.

There are reductions from NP-complete problems to "finding a halfway-decent MAX-CLIQUE approximation"

P vs. NP

What does the world look like?

If $\mathbf{P} = \mathbf{NP}$ finding the exact maximum clique is polynomial-time solvable.

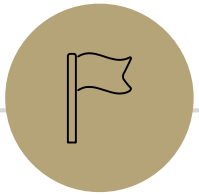
So is finding a minimum vertex cover and 2-coloring. They're all easy.

If $\mathbf{P} \neq \mathbf{NP}$ then these problems are fundamentally different.

Max-clique is REALLY REALLY difficult; don't even expect to get close.

Min Vertex Cover is difficult, but if you're willing to settle for ok, you're in really good shape.

2-coloring can be exactly solved.



Victory Lap

What have we seen this quarter?

- Stable Matchings
- Graph Search
  - BFS/DFS
  - Graph modeling
- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
  - DP on arrays
  - Adding parameters
  - DP on trees
- Network Flow
  - Using Max-Flow
  - Using Min Cuts
  - Assignment problems
- NP-completeness
  - Reductions
  - Showing a problem is NP-hard
  - P vs. NP
- Linear Programming
- Approximation Algorithms

Stable Matchings

Modeling matters!

It's better to be a proposer than a chooser!

Algorithms can be used to prove 'non-computational' facts

Stable Matchings always exist is easiest to prove by saying "here's how to find one."

Reductions

Sometimes there's a clever way to use an existing library (we'll need these a lot later in the quarter).

Graph Search

BFS and DFS search through a graph differently

So you can adapt them to solve different problems!

Use libraries

Finding SCCs and Topological sorting are “almost free” preprocessing

2-Coloring can be performed in linear time.

Greedy

Code is easy; proofs are hard.

Generating examples is **extremely** important.

To frame your thinking for proofs

Greedy stays ahead

Exchange argument

Structural result

Divide and Conquer

Trust the recursion.

Don’t be afraid to change what the recursive call gives you!

Add extra parameters!

State in English what the recursive call gives you.

In your “combine” step, make sure you’re beating baseline!

Dynamic Programming

Focus on solving the problem recursively; everything else is (mostly) formulaic once you've done that.

Write exactly the problem you're solving in English.

It's better to get down a "guess" at the problem and then see where you get stuck.

Don't be afraid to add a second recurrence or extra parameters.

Don't try to cleverly figure out which option is best. Try them all.

The magic of recursion tells you which is best for a particular situation.

DP is very useful on trees!

Network Flow

The value of the maximum flow is always equal to the capacity of the minimum cut.

Ford-Fulkerson finds you both.

Useful for modeling (“assigning” things)

Can the Mariners make the playoffs? (Assign who wins games)

Who does which chores?

And solving other problems

Especially on bipartite graphs, like vertex cover and maximum matching.

NP-completeness

$A \leq_P B$ means that you can use a library that solves problem B as a way to solve problem A

(with only polynomial time and polynomial calls to the library).

3-SAT (and other problems!) are NP-hard, which means for every problem $A \in NP$, $A \leq_P$3-SAT.

Showing a reduction from an already known $\mathbf{NP}$-hard problem to a new problem shows the new problem is also $\mathbf{NP}$-hard.

The order of the reduction is VERY important.

Approximation Algorithms

A way to cope with NP-hardness.

Approximation ratio:

The number (or function) > 1 that represents the ratio between alg and opt.

Minimization problem: for all instances I: $\alpha \cdot OPT(I) \geq ALG(I)$

Maximization problem: for all instances I: $OPT(I) \leq \alpha \cdot ALG(I)$

Rounding LPs is a common strategy

All our tools from the rest of the quarter can be used as well!

Just ask "I want a good answer" instead of "I want the best"

Some problems can be approximated well, others can't (unless $\mathbf{P} = \mathbf{NP}$).

How To Approach Problems

In section, we've made you follow these steps:

1. Read the problem carefully (make sure you know what problem you're actually solving)

2. Make some sample inputs/outputs

3. Set a "baseline."

4. Then try to generate the algorithm.

It's hard to take the time to do these in an exam, but at least make sure you do #1. Solving the wrong problem is not good for test-taking.

PLEASE do these steps in real life.

What have you learned?

Algorithm design techniques/paradigms

Ways to frame your thinking to design a new algorithm

Methods of modeling new problems in terms of familiar ones.

Solve a new problem without (many) new ideas.

How to approach a problem

Techniques to avoid getting stuck or get yourself un-stuck.

Some famous algorithms

You'll know Ford-Fulkerson, Bellman-Ford, etc. when you see them in libraries.

Or at least where to look them up.

What should you do next?

CSE 431 (complexity theory)

What ***can't*** you do? (in polynomial time, at all, or in limited memory)

CSE 422

Toolkit for modern algorithms: algorithmic principles behind modern stats and ML

CSE 426 [490C]

Cryptography: a mix of math, algorithms, and complexity.

CSE 521 and 525

Graduate level courses in algorithms and randomized algorithms.

Look ahead! These courses usually run once-per-year.