CSE 421
Introduction to Algorithms

Lecture 27: Dealing with NP-completeness:

LP Approximation
Local Search
Exponential-time Algorithms

New Orleans for New Year's?

Reminder/Announcement

- The Final Exam is Monday December 11, 2:30-4:20 pm here
  - I don't think that there is an exam after that in this room.
  - If there is extra time and nobody has a conflict that would prevent them staying longer, I will extend the time available.
  - Email me ASAP if you have a conflict with staying longer

- I will send an email later today with information about the exam and a sample final
  - It will be comprehensive and similar in style to the midterm.

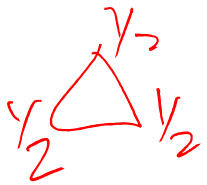
What to do if the problem you want to solve is NP-hard

2nd thing to try if your problem is a minimization or maximization problem

- Try to find a polynomial-time worst-case approximation algorithm
  - For a minimization problem
    - Find a solution with value $\leq K$ times the optimum
  - For a maximization problem
    - Find a solution with value $\geq 1/K$ times the optimum

Want K to be as close to 1 as possible.

Approximation Algorithms using Linear Programming



The generic approach to creating approximation algorithms for **NP**-optimization problems using Linear Programming:

1. Express the original problem as an Integer Program (ILP) or 01-Program (01-LP)
2. Keep same linear constraints but remove the integer requirement to get an LP. (Called the "LP relaxation".)
3. Solve the LP to yield a fractional solution
4. "Round" the fractional solution to an integer solution that satisfies all constraints.

Prove a bound on the ratio of the integer solution to the fractional LP solution

Observation: The LP optimum has at least as good an objective function value as the original problem since the LP allows all the ILP solutions plus some other fractional ones.

Recall: Greedy Approximation for Vertex-Cover

On input $G = (V, E)$

$W \leftarrow \emptyset$

$E' \leftarrow E$

while $E' \neq \emptyset$

  select any $e = (u, v) \in E'$

  $W \leftarrow W \cup \{u, v\}$

  $E' \leftarrow E' \setminus \{\text{edges } e \in E' \text{ that touch } u \text{ or } v\}$

Claim: At most a factor 2 larger than the optimal vertex-cover size.

Proof: Edges selected don't share any vertices so any vertex-cover must choose at least one of u or v each time.

Weighted Vertex Cover

Weighted Vertex Cover:

Given graph $G = (V, E)$ with each vertex v having a weight $w_v \geq 0$.

Find a vertex cover $C \subseteq V$ of G that minimizes $\sum_{v \in C} w_v$.

The greedy approximation approach doesn't work for this weighted version because for each edge, one of the two endpoints might have much larger weight than the other.

Weighted Vertex-Cover as an Integer Program

Variables x_v for $v \in V$

Minimize $\sum_{v \in V} w_v \cdot x_v$

subject to

$x_u + x_v \geq 1$ for each edge $\{u, v\} \in E$

$x_v \in \{0, 1\}$ for each node $v \in V$

The last line is equivalent to:

$0 \leq x_v \leq 1$ for each node $v \in V$

x_v integral for each node $v \in V$

Write OPT for the optimum cover weight

LP relaxation:

Minimize $\sum_{v \in V} w_v \cdot x_v$

subject to

$x_u + x_v \geq 1$ for each edge $\{u, v\} \in E$

$0 \leq x_v \leq 1$ for each node $v \in V$

Write OPT_{LP} for the optimum LP value

How do we round a LP solution achieving this value?

LP-Rounding to Approximate Weighted Vertex Cover

1. Solve the LP Relaxation
   a) Solution gives values $x_v \in [0, 1]$ for each $v \in V$
   b) $x_u + x_v \geq 1$ for each edge (u, v)
2. Round: Define $C \subseteq V$ to be $\{v : x_v \geq 1/2\}$
3. Observe that C is a vertex cover:
   - By 1 b), for each edge (u, v), at least one of $x_u \geq 1/2$ or $x_v \geq 1/2$ is true so either $u \in C$ or $v \in C$. or both
4. Since $x_v \geq 1/2$ for every $v \in C$, the total weight of C is

$$\sum_{v \in C} w_v \leq \sum_{v \in C} w_v \cdot (2x_v)$$

$$= 2 \sum_{v \in C} w_v \cdot x_v \leq 2 \sum_{v \in V} w_v \cdot x_v = 2\, OPT_{LP} \leq 2\, OPT.$$

Factor 2 approximation!

More on LP and Related Approximation Methods

More sophisticated methods for rounding variables $x_i \in [0, 1]$

- Randomized: View each x_i as a probability and independently produce

$$\text{solution } y_i = \begin{cases} 1 & \text{with probability } x_i \\ 0 & \text{with probability } 1 - x_i \end{cases}$$

- Correlated random sampling. Apply the above but "correlate" choices somehow

Instead of LP relaxations, use "Semi-Definite Programming (SDP)" relaxations.

- SDPs generalize LPs. They can also be solved efficiently using Ellipsoid and Interior Point Methods. They are a special case of convex programming.
- Currently yield the best approximations known for many **NP**-hard problems.

What to do if the problem you want to solve is NP-hard

NP-completeness is a worst-case notion...

- Try an algorithm that is provably fast "on average".
  - To even show this one needs a model of what a typical instance is.
  - Typically, people consider "random graphs"
    - e.g. all graphs with a given # of edges are equally likely
    - In this case one can sometimes show that many NP-hard problems are easy
  - Problems:
    - real data doesn't look like the random graphs
    - distributions of real data aren't analyzable

Heuristic Algorithms

These algorithms typically do not have proven bounds on solution quality:

The most important of these methods are based on variants of

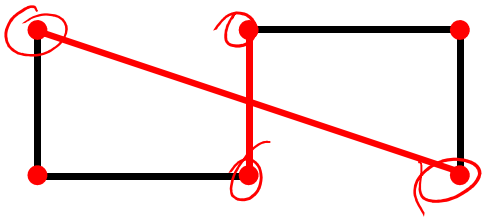
Local search:

- Need a notion of two solutions being neighbors

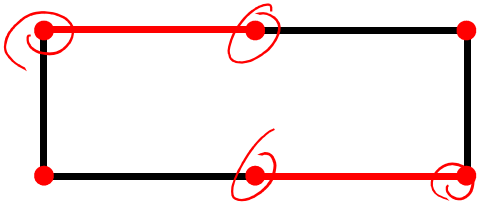
```
Start at an arbitrary solution S
While there is a neighbor T of S that is better than S
    S←T
```

e.g., Neighboring solutions for TSP

Solution S



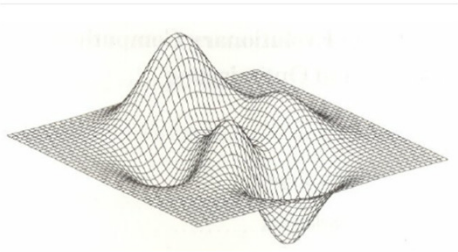
Solution T



Two solutions are neighbors*
iff there is a pair of edges you can swap to transform one to the other

*These are called 2-OPT neighbors. There are other more sophisticated neighbor structures

Variants of Local Search



Basic local search (greedy)

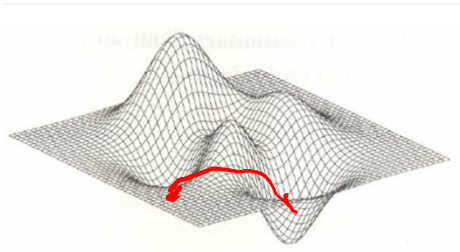
- *Usually fast but often gets stuck in a local optimum that is far from the global optimum*
- *With some notions of neighbor structure even this can take a long time in the worst case*

Randomized local search:

Start local search several times from random starting points and take the best answer found overall.

- *More expensive than plain local search but usually much better answers. It is usual easy to control the time spent so this is almost always better to do.*

Variants of Local Search



Metropolis Algorithm

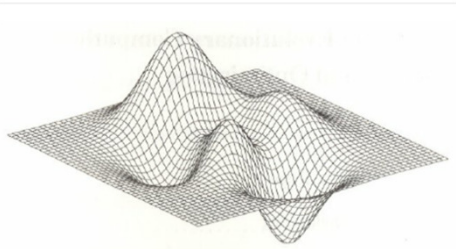
Like randomized local search except that at each step one always chooses a random neighbor but doesn't always move to it:

e.g. Always move to the neighbor if it is better but move to a worse neighbor with some fixed probability depending on how much worse it is. (Fixed inverse temperature.) cf. CSE 312 Markov Chain Knapsack assignment.

Advantage: If local optima are not too deep/steep, will not get stuck there. However can still get stuck

Often used in practice. Drawback: Each run can be much longer than local search but one can hope to try to make it up with solution quality. A good option to compare with randomized local search. It is unclear which will be better in a given circumstance.

Variants of Local Search



Simulated Annealing

Like Metropolis algorithm but probability of going to a worse neighbor is set to decrease with time on a "cooling schedule" as, presumably, solution is closer to optimal

(analogy with slow cooling to get to lowest energy state in a crystal (or in forging a metal)

Much slower to converge than Metropolis.

M*ost improvement occurs at some fixed temperature.*

Answers usually not much better than Metropolis, if at all, so not generally worth the extra compute time.

What to do if the problem you want to solve is NP-hard

Maybe you only need to solve it if the solution size is small...

- What if you only need find cliques or vertex covers of constant size?
- For both **Clique** and **Vertex Cover**, the obvious brute force algorithm would have time $\Theta(n^k)$: try all subsets of size k.
- For **Clique** the best algorithms known are all $n^{\Omega(k)}$
- However, **Vertex Cover** has a much better algorithm with

The theory of **fixed parameter tractability** looks at **NP** problems using a second parameter k in addition to input size n and seeks algorithms with running times $f(k) \cdot n^{O(1)}$ where f might be exponential.

- More later

What to do if the problem you want to solve is NP-hard

$2^{n/1000}$

Try to make an exponential-time solution as efficient as possible.

e.g. Try to search the space of possible hints/certificates in a more efficient way and hope that it is quick enough.

Backtracking search

e.g., for SAT, search through the 2^n possible truth assignments...

...but set the truth values one-by-one so we can able to figure out whole parts of the space to avoid,

e.g. Given $F = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_4 \vee \neg x_3) \wedge (x_1 \vee x_4)$

after setting $x_1 = 1$ and $x_2 = 0$ we don't even need to set x_3 or x_4 to know that it won't satisfy F.

Next Class: Much more clever backtracking search for SAT solutions

Exponential-Time Algorithms

Branch-and-bound search for optimization problems:

- **Branch**: Use backtracking search through a tree representing partial solutions
- **Bound**: In addition to keeping track of the best full solution found so far, at each step produce a bound on the quality of the best possible completion of the current partial solution
  - If that best possible completion is worse than the best full solution found so far, prune the search and backtrack instead.

Example: In backtracking search for **MetricTSP** one can use linear programming to provide lower bounds

Note: An excellent exact solver for **MetricTSP** called **Concorde** combines branch-and-bound and LP/ILP methods and will solve problems involving thousands of cities.

Other Heuristic Algorithms you might hear about

Genetic algorithms:

- View each solution as a **string** (analogy with DNA)
- Maintain a **population of good solutions**
- Allow **random mutations** of single characters of individual solutions
- **Combine two solutions** by taking part of one and part of another (analogy with crossover in sexual reproduction)
- Get rid of solutions that have the worst values and make multiple copies of solutions that have the best values (analogy with natural selection -- survival of the fittest).

*Usually very slow. In the rare cases when they produce answers with better objective function values than other methods they tend to produce very **brittle** solutions – that are very bad with respect to small changes to the requirements.*

Deep Neural Nets and NP-hardness?

- **Artificial neural networks**
  - based on very elementary model of human neurons
  - **Set up a circuit of artificial neurons**
    - each artificial neuron is an analog circuit gate whose computation depends on a set of **connection strengths**
  - **Train the circuit**
    - Adjust the connection strengths of the neurons by giving many positive & negative training examples and seeing if it behaves correctly
  - **The network is now ready to use**

Despite their wide array of applications, they have not been shown to be useful for NP-hard problems.

Quantum Computing and NP-hardness?

Use physical processes at the quantum level to implement "weird" kinds of circuit gates based on unitary transformations

- Quantum objects can be in a "superposition" of many pure states at once
  - Can have n objects together in a superposition of 2^n states
- Each quantum circuit gate operates on the whole superposition of states at once
  - Inherent parallelism but classical randomized algorithms have a similar parallelism: *not enough on its own*
  - Advantage over classical: copies interfere with each other.
- Exciting direction - theoretically able to factor efficiently.
  Major practical problems wrt errors, decoherence to be overcome.
- *Small brute force improvement but unlikely to produce exponential advantage for NP.*