CS 561, Approximation Algorithms

Jared Saia
University of New Mexico

Outline

- Approximation Algorithms

Vertex Cover

- A *vertex cover* of a graph is a set of vertices that touches every edge in the graph
- The decision version of *Vertex Cover* is: "Does there exist a vertex cover of size k in a graph G?".
- We've proven this problem is NP-Hard by an easy reduction from Independent Set
- The *optimization* version of *Vertex Cover* is: "What is the minimum size vertex cover of a graph G?"
- We can prove this problem is NP-Hard by a reduction from the decision version of Vertex Cover (left as an exercise).

Approximating Vertex Cover

- Even though the optimization version of Vertex Cover is NP-Hard, it's possible to *approximate* the answer efficiently
- In particular, in polynomial time, we can find a vertex cover which is no more than 2 times as large as the minimal vertex cover

Approximation Algorithm

- The approximation algorithm does the following until G has no more edges:
- It chooses an arbitrary edge (u, v) in G and includes both u and v in the cover
- It then removes from G all edges which are incident to either u or v

Approximation Algorithm

```
Approx-Vertex-Cover(G){
  C = {};
  E' = Edges of G;
  while(E' is not empty){
    let (u,v) be an arbitrary edge in E';
    add both u and v to C;
    remove from E' every edge incident to u or v;
  }
  return C;
}
```

Analysis

- If we implement the graph with adjacency lists, each edge need be touched at most once
- Hence the run time of the algorithm will be $O(|V| + |E|)$, which is polynomial time
- First, note that this algorithm does in fact return a vertex cover since it ensures that every edge in G is incident to some vertex in C
- Q: Is the vertex cover actually no more than twice the optimal size?

Analysis

- Let A be the set of edges which are chosen in the first line of the while loop
- Note that no two edges of A share an endpoint
- Thus, *any* vertex cover must contain at least one endpoint of each edge in A
- Thus if $C*$ is an optimal cover then we can say that $|C*| \geq |A|$
- Further, we know that $|C| = 2|A|$
- This implies that $|C| \leq 2|C*|$

Which means that the vertex cover found by the algorithm is no more than twice the size of an optimal vertex cover.

TSP

- An optimization version of the TSP problem is: "Given a weighted graph G, what is the shortest Hamiltonian Cycle of G?"
- This problem is NP-Hard by a reduction from Hamiltonian Cycle
- However, there is a 2-approximation algorithm for this problem if the edge weights obey the *triangle inequality*

Triangle Inequality

- In many practical problems, it's reasonable to make the assumption that the weights, c, of the edges obey the *triangle inequality*
- The triangle inequality says that for all vertices $u, v, w \in V$:

$$c(u, w) \leq c(u, v) + c(v, w)$$

- In other words, the cheapest way to get from u to w is always to just take the edge (u, w)
- In the real world, this is usually a pretty natural assumption. For example it holds if the vertices are points in a plane and the cost of traveling between two vertices is just the euclidean distance between them.

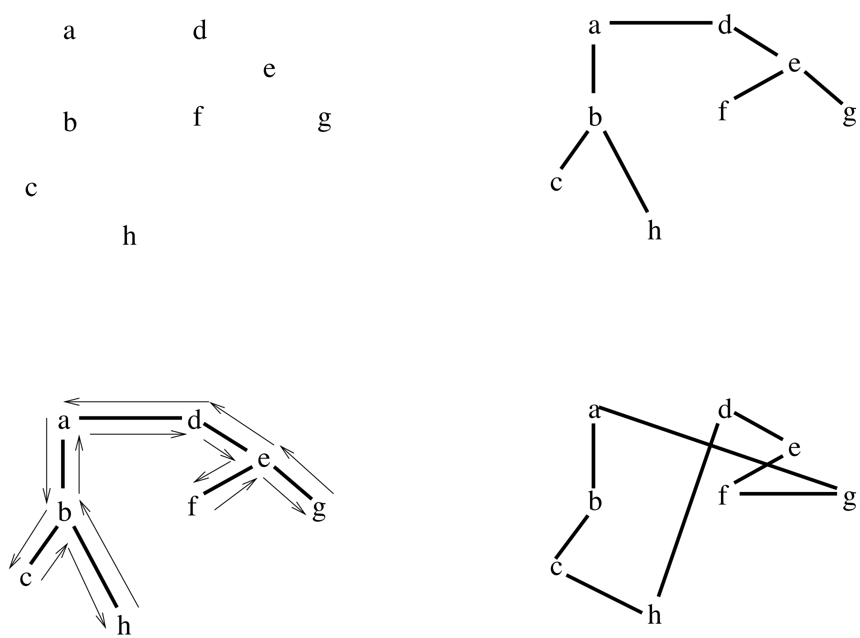
Approximation Algorithm

- Given a weighted graph G, the algorithm first computes a MST for G, T, and then arbitrarily selects a root node r of T.
- It then lets L be the list of the vertices visited in a depth first traversal of T starting at r.
- Finally, it returns the Hamiltonian Cycle, H, that visits the vertices in the order L.

Approximation Algorithm

```
Approx-TSP(G){
  T = MST(G);
  L = the list of vertices visited in a depth first traversal
      of T, starting at some arbitrary node in T;
  H = the Hamiltonian Cycle that visits the vertices in the
      order L;
  return H;
}
```

Example Run



The top left figure shows the graph G (edge weights are just the Euclidean distances between vertices); the top right figure shows the MST T. The bottom left figure shows the depth first walk on T, $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$; the bottom right figure shows the Hamiltonian cycle H obtained by deleting repeat visits from W, $H = (a, b, c, h, d, e, f, g)$.

Analysis

- The first step of the algorithm takes $O(|E| + |V| \log |V|)$ (if we use Prim's algorithm)
- The second step is $O(|V|)$
- The third step is $O(|V|)$.
- Hence the run time of the entire algorithm is polynomial

Analysis

An important fact about this algorithm is that: *the cost of the MST is less than the cost of the shortest Hamiltonian cycle.*

- To see this, let T be the MST and let $H*$ be the shortest Hamiltonian cycle.
- Note that if we remove one edge from $H*$, we have a spanning tree, T'
- Finally, note that $w(H*) \geq w(T') \geq w(T)$
- Hence $w(H*) \geq w(T)$

Analysis

- Now let W be a depth first walk of T which traverses each edge exactly twice (similar to what you did in the hw)
- In our example, $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$
- Note that $c(W) = 2c(T)$
- This implies that $c(W) \leq 2c(H*)$

Analysis

- Unfortunately, W is not a Hamiltonian cycle since it visits some vertices more than once
- However, we can delete a visit to any vertex and the cost will not increase *because of the triangle inequality*. (The path without an intermediate vertex can only be shorter)
- By repeatedly applying this operation, we can remove from W all but the first visit to each vertex, without increasing the cost of W.
- In our example, this will give us the ordering $H = (a, b, c, h, d, e, f, g)$

Analysis

- By the last slide, $c(H) \leq c(W)$.
- So $c(H) \leq c(W) = 2c(T) \leq 2c(H*)$
- Thus, $c(H) \leq 2c(H*)$
- In other words, the Hamiltonian cycle found by the algorithm has cost no more than twice the shortest Hamiltonian cycle.

MAX-SAT

- Imagine that we have some CNF boolean function
- Each clause C_j has some positive variables P_j and some negative variables N_j
- Our goal is to set truth values to the variables in order to maximize the number of satisfied clauses
- IDEA: Solve an LP; Use the settings in this solution to assign probabilities to indicator r.v.'s; Round these r.v.'s.

The Linear Program (LP)

Maximize: $\sum_j z_j$

Subject to:

$z_j \leq \sum_{i \in P_j} y_i + \sum_{i \in N_j} (1 - y_i)$, $\forall C_j$

$0 \leq y_i \leq 1$, $\forall y_i$

$0 \leq z_j \leq 1$, $\forall z_j$

The Algorithm

- Write an LP for the boolean formula as in the previous slide
- Let y_i^* be the settings found in the solution found for the LP
- For each variable i, set i to TRUE with probability y_i^* and FALSE otherwise

Analysis Background

- Convex/Concave Functions
- Arithmetic/Geometric Mean inequality

Convex Functions

- A function, f, is **convex** if for all inputs x and y and for all $t \in [0, 1]$:

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$$

- Key fact: If f has a second derivative, then f is convex iff the second derivative is always non-negative.

Concave Functions

- A concave function is the negative of a convex function
- A function, f, is **concave** if for all inputs x and y and for all $t \in [0,1]$:

$$f(tx + (1-t)y) \geq tf(x) + (1-t)f(y)$$

- Key fact: If f has a second derivative, then f is concave iff the second derivative is always negative.

GM $\leq$ AM

- For any non-negative $x_1, x_2, \ldots, x_k$, the geometric mean is at most equal to the arithmetic mean
- $(x_1 x_2 \ldots x_k)^{1/k} \leq (1/k)(x_1 + x_2 + \ldots + x_k)$
- Easy to see this for 2 variables: $\sqrt{xy} \leq (1/2)(x + y)$

Probability C_j is not satisfied

- Fix some clause C_j and let P_j be the set of positive and N_j be the set of negative variables in C_j
- Then the probability that the clause is not satisfied is

$$
\begin{aligned}
\prod_{i \in P_j} (1 - y_i^*) \prod_{i \in N_j} y_i^* &\leq \left(\frac{1}{k} \left(\sum_{i \in P_j} (1 - y_i^*) + \sum_{i \in N_j} y_i^* \right) \right)^k \\
&= \left(1 - \frac{1}{k} \left(\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} (1 - y_i^*) \right) \right)^k \\
&\leq \left(1 - \frac{z_j^*}{k} \right)^k
\end{aligned}
$$

Using Concavity

- Probability that C_j *i*s satisfied is: $1 - \left(1 - \frac{z_j^*}{k}\right)^k$
- $f(z_j^*) = 1 - \left(1 - \frac{z_j^*}{k}\right)^k$ is concave over $z_j^* \in [0, 1]$
- Hence: For any x and y and all $t \in [0, 1]$:

$$f(tx + (1-t)y) \geq tf(x) + (1-t)f(y)$$

- Specifically if $x = 0$ and $y = 1$, then

$$f((1-t)) \geq (1-t)f(1)$$

- Setting $1 - t$ to be z_j^*, we get that

$$f(z_j^*) \geq z_j^* \left(1 - \left(1 - \frac{1}{k}\right)^k\right)$$

Using Linearity of Expectation

- Probability that C_j is satisfied is $\geq z_j^* \left(1 - \left(1 - \frac{1}{k}\right)^k\right)$
- Let W be the number of clauses satisfied by our algorithm, and let W_j be an indicator r.v. that is 1 iff C_j is satisfied.

$$\begin{aligned}
E(W) &= \sum_j E(W_j) \\
&\geq \sum_j z_j^* \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \\
&\geq \min_k \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \sum_j z_j^* \\
&\geq \min_k \left(1 - \left(1 - \frac{1}{k}\right)^k\right) OPT \\
&\geq (1 - 1/e) OPT \\
&\geq .632 \cdot OPT
\end{aligned}$$

Take Away

- Many real-world problems can be shown to not have an efficient solution unless $P = NP$ (these are the NP-Hard problems)
- However, if a problem is shown to be NP-Hard, all hope is not lost!
- In many cases, we can come up with an provably good approximation algorithm for the NP-Hard problem.