CMPSCI 611 Advanced Algorithms **Fall 2001**

Lecture 24: December 5

Lecturer: Micah Adler *Scribe: Vanessa Gaudin*

24.1 Finishing TSP

An approximation for MTSP uses a Eulerian tour.

Definition 24.1 *An Eulerian tour is a path that traverses every edge of the graph exactly once and returns back to the initial vertex (it may visit vertices multiple times).*

A graph G has an Eulerian tour if and only if G is connected and every vertex has even degree. We also know that if an Eulerian tour exists we can find it in polynomial time.

1. Construct a graph representing MTSP such that vertices represent the cities, all edges are present, and the edge weights are defined by the distance function.

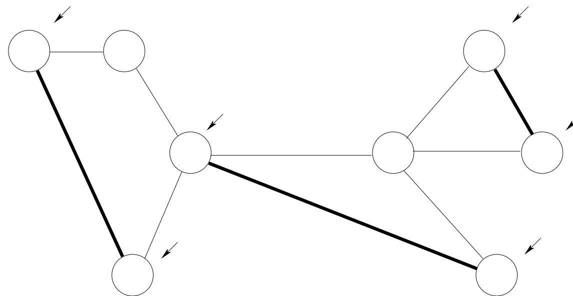

Figure 24.1: MTSP Minimum spanning tree plus matchings.

2. Find the Minimum Spanning Tree, T (represented by the thin lines in Figure 24.1).

3. Add edges to T so that every vertex has even degree as follows:

   (a) Find the set D of vertices of T whose degree is odd, (these are denoted by the arrows in Figure 24.1). $|D|$ will always be even for every graph. This is because every edge touches two vertices. The sum of the degree of all vertices is an even number (twice the number of edges). Therefore the sum of the degrees of the odd degree vertices is even, because the only way to get an even number from the sum of n odd numbers is if n is even.

   (b) Add to T a minimum weight perfect matching on the vertices in D, (these edges are shown by thick lines in Figure 24.1). A perfect matching must exist because D has an even number of vertices. If it happens that any of the added edges duplicate an existing edge, that doesn't affect the algorithm. We just have that edge twice.

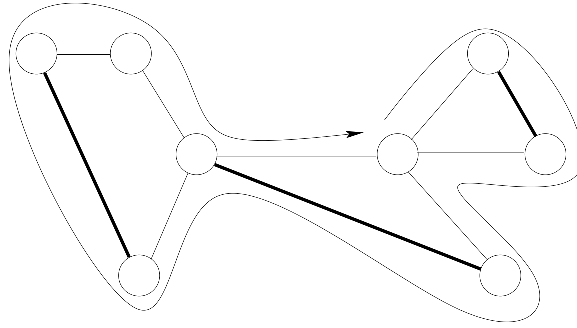

Figure 24.2: An Eulerian tour.

4. Find an Eulerian tour on the resulting graph. We know that every vertex must be in the tour (because we started with a MST) and that each vertex is visited at least once (because this is an Eulerian tour). The tour is shown in Figure 24.2.

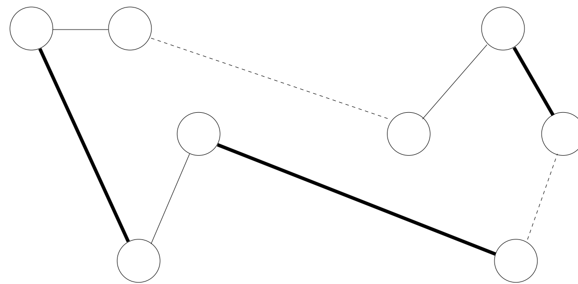

Figure 24.3: The Final Approximation to the MTSP.

5. Visit all the vertices in the order of the Eulerian tour, short-cutting any repeated vertices (using shortcuts as in the first approximation algorithm from lecture 23). The tour with shortcuts is shown in Figure 24.3, where the shortcuts are represented by dotted lines.

Claim 24.2 *This is a $\frac{3}{2}$ approximation.*

Proof: We know, from the triangle inequality, that the short-cutting does not increase the length of the tour. So the resulting tour has a cost no larger than that of the Eulerian tour.

$$\text{(cost of our tour)} \leq \text{(cost of the Eulerian tour)}$$

The Eulerian tour is composed of minimum spanning tree T and the D-matching.

$$\text{(cost of the Eulerian tour)} \leq \text{(cost of } T) + \text{(cost of the matching)}$$

Therefore,

$$\text{(cost of our tour)} \leq \text{(cost of } T) + \text{(cost of the matching)}$$

Calculating the cost of the matching:
The traveling salesman tour is a tour on all vertices in the graph. If we restrict our tour to just D, we can take the tour that visits all of the vertices, short-cutting any that are not in D. So,

$$\text{(cost of the tour on } D) \leq \text{(cost of the optimal tour)}$$

If we have an optimal tour on the vertices in D, we can divide it into two matchings (since D has an even number of vertices). This implies that there exists a matching with weight at most $\frac{1}{2}$ the cost of the optimal tour on D. So,

$$\text{(cost of matching)} \leq \frac{1}{2} \cdot \text{(cost of the optimal tour on } D\text{)}$$

and

$$\text{(cost of the optimal tour on } D\text{)} \leq \text{(cost of the optimal tour)}$$

Since we required a minimal matching on D, its cost must also be at most $\frac{1}{2}$ of the cost of the optimal tour:

$$\text{(cost of matching)} \leq \frac{1}{2} \cdot \text{(cost of the optimal tour)}$$

Then,

$$\text{(cost of our tour)} \leq \text{(cost of } T\text{)} + \text{(cost of the matching)}$$

Since T is a minimum spanning tree:

$$\text{(cost of } T\text{)} \leq \text{(cost of the optimal tour)}$$

And we have:

$$\begin{aligned} \text{(cost of our tour)} &\leq \text{(cost of the optimal tour)} + \frac{1}{2} \cdot \text{(cost of the optimal tour)} \\ &\leq \frac{3}{2} \cdot \text{(cost of optimal tour)} \end{aligned}$$

Thus we have a $\frac{3}{2}$-approximation. ■

Christofides [C76]: Proved that the Metric version of TSP (in which the distance function obeys the triangle inequality) has a 3/2-approximation.

Arora [A96]: Attempted to prove that the 3/2-approximation was the best possible one, and found a *better* approximation for a more restricted version of the problem.

Euclidean version of TSP:
The cities are restricted to a plane and the distances between them are distances on the plane. This problem is **NP**-complete, though the proof is not given here.

For all $\epsilon > 0$, a $(1+\epsilon)$-approximation exists and can be found in $n^{O(1+1/\epsilon)}$ time. This is powerful because we're saying that no matter what ϵ we choose we can get within a $(1+\epsilon)$ factor of the correct answer.

If ϵ is considered to be constant, this is a polynomial time algorithm in n. However, if ϵ is very small, the exponent in the polynomial may be very large. For example, $\epsilon = 0.01$ yields an exponent of $n^{O(101)}$.

24.2 Polynomial Time Approximation Scheme

Definition 24.3 *A problem has a* polynomial time approximation scheme *(PTAS) if and only if* $\forall \epsilon > 0$ *it has a polynomial time* $(1+\epsilon)$*-approximation.*

This is not the best we can do in general. The reason is that these schemes are exponentially dependent on ϵ. There are problems for which we can have a polynomial dependence. If we let ϵ get arbitrarily small, the running time for a PTAS gets slower and slower, but with *fully* polynomial time approximation schemes (FPTAS) the running time remains polynomial.

Definition 24.4 *A PTAS is called a fully polynomial time approximation scheme (FPTAS) if the runtime of the* $(1+\epsilon)$*-approximation is bounded by a polynomial in* $\frac{1}{\epsilon}$ *(as well as the size of the input).*

24.2.1 KNAPSACK

KNAPSACK has a FPTAS algorithm. The last time we looked at the Knapsack problem, we looked at a simplified version where the weight equaled the value. Now we look at the full version.

The KNAPSACK problem:

Input: A set of items numbered as $1, 2, \ldots, n$;
a weight w_i for each item i;
a value v_i for each item i;
and a knapsack capacity, C.

Output: A subset, B, of the items with the maximum total value such that $\sum_{i \in B} w_i \leq C$.

24.2.2 Previous Knapsack Algorithm

In a previous lecture we introduced a dynamic programming algorithm to solve the KNAPSACK problem.

Define $knap(i, w)$ to be the maximum value obtained using items 1 to i and capacity w_1 to w.

$Knap(i+1, w) = \max(Knap(i, w), Knap(i, w - w_{i+1}) + v_{i+1})$

Computing $knap(i, w)$ row by row in a table of size $n \cdot C$ we can get the solution $knap(n, C)$.

The runtime of this algorithm is $O(n \cdot C)$, and it will always give an exact solution. (Notice that C is exponential to its input size – $\log(C)$.) This algorithm does not lend itself to FPTAS.

24.2.3 New Knapsack Algorithm

Each entry in our dynamic programming table will be the minimum weight for a given value. Instead of taking the maximum of a subset of items and capacities, we define $Vknap(i, v)$ to be the minimum weight required to achieve a value of at least V using items $1, 2, \ldots, i$. We want a way of computing these values adding items one at a time:

$$Vknap(i+1, v) = \min\{ Vknap(i, v),\ Vknap(i, v - v_{i+1}) + w_{i+1}\}$$

and, for the first row in the table:

$$Vknap(1, v) = \begin{cases} w_1 & \text{if } v \leq v_1 \\ \infty & \text{if } v > v_1 \end{cases}$$

We want the minimum weight to achieve a value at least V, we need the minimum weight required to achieve $v - v_{i+1}$, and to that we just have to add the weight required for the $i^{th} + 1$ item. In the old version the

number of columns was the total capacity. In this version the number of columns is the sum of the values. To find the optimum solution (the maximum value that doesn't exceed the knapsack capacity), we will examine the last row, starting with the largest value, and scan left until we find the value that can be achieved without exceeding the capacity. In other words we will find the largest column, v_{max}, such that $Vknap(n, v_{max}) \leq C$. (Notice that $Vknap(n, v)$ is non-decreasing as v increases.)

To analyze the runtime of this algorithm, we define $V = \max_i(v_i)$. The maximum solution achievable is nV (the bottom right-hand corner of the table). The size of the table is n^2V (n rows by $n \cdot V$ columns, with every entry computed in constant time). We can see the runtime of this algorithm is $O(n^2V)$. This algorithm is slower than our original dynamic programming algorithm which was $O(nV)$. And it turns out this new algorithm is not really a polynomial time algorithm because V could be exponential with respect to the input size. We can design a (faster) polynomial time algorithm by approximating V.

24.2.4 Knapsack Approximation

The original version of the algorithm looked at the weights. We might be tempted to round down the weights. The problem then is that we could end up with a subset of items whose weights exceed the knapsack's capacity. If we round up the weights we get a valid solution but we might not have included an item that would have maximized the value in the knapsack.

For an approximation of V we use the binary representation of V and take the k least significant bits and make them equal to 0. This doesn't affect the value of V too much because the lower order bits don't add up to very much compared to the higher order bits, but reducing the number of bits reduces the amount of time to process them, and thus improves the running time of the algorithm.

An approximation for KNAPSACK: Change the k lowest order bits of every v_i to 0. The tradeoff here is that when k is small, our approximation is good (ϵ is small); while when k is large, the runtime is reduced. The solution is still valid because we haven't changed the weights, and the value won't vary from the optimal value by very much.

For FPTAS we need a $(1+\epsilon)$-approximation. This will give an upper bound on k, we would like the largest k within this bound in order to minimize the running time of the algorithm, while still observing our accuracy constraints.

Let $v'_i = v_i$ with the k lowest order bits set to zero.
Let C_{alg} be the optimal answer to the modified problem.
Let C_{opt} be the optimal answer to the original problem.
Let B' be the optimal subset of items in the modified problem.
Let B be the optimal subset of items in the original problem.

Since this is a maximization problem, we need to find k such that $\frac{C_{opt}}{C_{alg}} \leq (1+\epsilon)$, for a given ϵ.

$$C_{alg} = \sum_{i \in B'} v'_i$$

Since the set B' represents the maximum solution to the modified problem, the solution formed by the set B must be less than or equivalent to it (in the modified problem). Therefore:

$$C_{alg} = \sum_{i \in B'} v'_i \geq \sum_{i \in B} v'_i$$

Since each v_i' has the k smallest bits set to 0, $v_i' > v_i - 2^k$ for all v_i, and

$$\sum_{i \in B} v_i' > \sum_{i \in B} v_i - 2^k \geq C_{opt} - n \cdot 2^k$$

Therefore:

$$\begin{aligned}
C_{alg} &\geq C_{opt} - n \cdot 2^k \\
\frac{C_{opt}}{C_{alg}} &\leq \frac{C_{opt}}{C_{opt} - n \cdot 2^k} \\
&= \frac{C_{opt} - n \cdot 2^k + n \cdot 2^k}{C_{opt} - n \cdot 2^k} \\
&= 1 + \frac{n2^k}{C_{opt} - n2^k} \\
&\leq 1 + \frac{n2^k}{V - n2^k}
\end{aligned}$$

Where V is the maximum v_i, an upper bound on C_{opt}, assuming no items exist that cannot fit into the knapsack. For now we assume $V \geq 2n2^k$. At the end we verify that this assumption holds.

Assuming that $V \geq 2n2^k$, we have

$$1 + \frac{n2^k}{V - n2^k} \leq 1 + \frac{2n2^k}{V}$$

Since we are looking for a $(1+\epsilon)$-approximation, we want a value of k such that $\frac{2n2^k}{V}$ will be no larger than ϵ. Thus,

$$\begin{aligned}
\frac{2n2^k}{V} &\leq \epsilon \\
k &\leq \lg(\frac{\epsilon V}{2n})
\end{aligned}$$

To satisfy our previous assumption, it is sufficient that:

$$\begin{aligned}
V &\geq 2n2^k \\
&= 2n\frac{\epsilon V}{2n} \\
\epsilon &\leq 1
\end{aligned}$$

This assumption holds if we make a restriction on the algorithm that $\epsilon \leq 1$, which is fine since we are interested in having a small ϵ. With a FPTAS we are given a value of ϵ and we want to compute the value of k that will give us a $(1+\epsilon)$-approximation.

24.2.5 To verify it is fully polynomial

The running time is still the size of the table, with each table entry taking constant time. Assume that the algorithm removed bits, rather than simply setting them to 0. Each value would then be shifted k bits, and the new maximum value is $\frac{V}{2^k}$. We still have n rows, but now we have $\frac{nV}{2^k}$ columns, so the table is size $n \cdot \frac{nV}{2^k}$. Solving for k, $k \leq \lg(\frac{\epsilon V}{2n})$ Thus the run time is $O(\frac{n^2 V}{2^k}) = O(\frac{n^2 V}{\frac{\epsilon V}{2n}}) = O(\frac{n^3}{\epsilon})$.

which is linear in $1/\epsilon$, and polynomially dependent on ϵ.

24.2.6 Quick Review of KNAPSACK

FPTAS for Knapsack problem:

Given a problem instance and an approximation ratio $(1+\epsilon)$:

1. Set the value of k, i.e., let $k = \lfloor \log(\frac{\epsilon V}{2n}) \rfloor$;
2. Truncate the last k bits from values of all items;
3. Use this as input to the new version of the dynamic programming algorithm to get the subset of items.
4. Return the solution that subset of items gives us.

24.3 Problems without good approximation algorithms

24.3.1 Review of approximation algorithms

Thus far we have seen, in order of increasing quality of approximation:

- Vertex-Cover/Max-Cut has a 2-approximation,
- Metric TSP has a 3/2-approximation,
- TSP-Euclidean has a PTAS solution, and
- Knapsack has a FPTAS solution.

Is there a FPTAS for every problem?

It is believed that there is no good approximation solution for many problems, however, this is difficult to prove.

If we can prove that there is no FPTAS for any **NP**-complete problem, we can show that $\mathbf{P} \neq \mathbf{NP}$, since any polynomial time algorithm provides an exact solution in polynomial time, and would therefore also provide an FPTAS solution to the problem.

However, we also know that certain problems can't be approximated well unless $\mathbf{P} = \mathbf{NP}$.

24.3.2 Types of Approximation Algorithms

In order of decreasing strength:

1. FPTAS - (the best we can hope for) Example: Knapsack
2. PTAS - Example: Euclidean TSP
3. Constant - worse, but still respectable. Example: vertex cover
4. $\log n$
5. n^ϵ, for constant ϵ - Example: clique problem.

References

A96 S. Arora, Polynomial time approximation schemes for Euclidean TSP and other geometric problems, *37th Annual Symposium on Foundations of Computer Science, IEEE Comput. Soc. press.*

C76 N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, TR CS-93-13, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, 1976.

CLR90 T. Cormen and C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 1990

FD2000 F. Diaz, Scribe notes, Lecture 21, Advanced Algorithms, Fall 2000.

GW2000 Z. Ge and A. Wolfe, Scribe notes, Lecture 22, Advanced Algorithms, Fall 2000.