Lecture 10

Dynamic Programming

November 1, 2004
Lecturer: Kamal Jain
Notes: Tobias Holgers

10.1 Knapsack Problem

We are given a set of items $U = \{a_1, a_2, \ldots, a_n\}$. Each item has a weight $w_i \in \mathbb{Z}^+$ and a utility $u_i \in \mathbb{Z}^+$. Our task is to find the most valuable set of items with respect to the utility function under the constraint that the knapsack has a fixed capacity of B. Find $S \subseteq U$ satisfying:

$$\begin{array}{ll} \text{condition} & \sum_{i \in S} w_i \leq B \\ \text{maximum} & \sum_{i \in S} u_i \end{array}$$

10.2 Types of Polynomial Algorithms

This section present three types of polynomial algorithms. Each definition contains an example run time for the Knapsack problem if such an algorithm existed.

Definition 1 *Polynomial algorithm ≡ Running time Poly(Instance size)*

$$Poly(\sum_i \log w_i + \sum_i \log u_i)$$

Definition 2 *Strongly polynomial ≡ Running time Poly(#numbers in instance)*

$$Poly(n)$$

Definition 3 *Pseudo polynomial ≡ Running time Poly(magnitude of numbers). Where numbers are written in unary encoding. Unary encoding for $n \geq 0$ is $n+1$ consecutive ones, e.g. $5 = 111111$.*

$$Poly(\sum_i w_i + \sum_i u_i)$$

10.3 Dynamic Programming

We have an instance I of a problem we like to solve using dynamic programming. Now create sub-instances $\mathcal{S} = \{I_1, I_2, \ldots, I_k\}$. The dynamic program then does:

1. It puts a total (or partial) order on $\mathcal{S}$.
2. First sub-instance is easily solvable. (total order)
   All minimal instances are easily solvable. (partial order)
3. The last instance is your original instance. (total order)
   The maximum instance is your original instance. (partial order)
4. Take any instance $I_m \in \mathcal{S}$
   If you know the solutions of the previous instances then a solution of I_m can be computed easily. For the partial ordering you must know the solutions of the lower instances.

10.4 Dynamic Programming for Knapsack Problem

We will here apply the previous four steps of a dynamic programming algorithm on the knapsack problem. First define a sub-instance $A(i, w)$ which is the maximum utility using only items $\{a_1, a_2, \ldots, a_i\}$ and with a maximum capacity of w.

The second item is simply a question of whether we are able to pick the only item with regard to the maximum capacity, i.e. $A(n, B)$.

$$A(1, w) = \begin{cases} 0 & w < a_1 \\ u_1 & w \geq a_1 \end{cases}$$

Our final solution, which corresponds to item three, is the sub-instance over all items using the original capacity.

We can show item four by induction. We can determine the solution to a sub-instance using only previous/lesser sub-instances.

$$A(i, w) \left| \begin{array}{l} A(i', w') \\ i' \leq i, w' \leq w \text{ where at least } i' < i \text{ or } w' < w \end{array} \right.$$

The addition of one more item is similar to the case where we only had one item. We either add it to the previous sub-instance with room for the new item or we don't pick it at all.

$$A(i, w) = \max \begin{cases} u_i + A(i-1, w - w_i) & w \geq w_i \\ A(i-1, w) & \text{otherwise} \end{cases}$$

The running time for this dynamic programming algorithm is $n \cdot B$. It is also pseudo polynomial in B if all inputs are given in unary.

10.5 Knapsack Minimization Problem

The running time of the previous algorithm depends on B which might be large. We rather want it to be pseudo polynomial in $\sum_i u_i$. Rewrite the sub-instances to be $A(i, U)$ which is the minimum weight using only items $\{a_1, a_2, \ldots, a_i\}$ and a utility of at least U. The recursion then looks like

$$A(i,U) = \min \begin{cases} w_i + A(i-1, U-u_i) & U \geq u_i \\ A(i-1,U) & \text{otherwise} \end{cases}$$

This dynamic programming algorithm has a running time of $n \cdot \sum_{i \in U} u_i$. The exact solutions of the two different recursive formulations of the knapsack problems are equivalent. In the approximation case they are different.

10.6 Approximation Scheme

Definition 4 *An α-approximation algorithm produces a solution satisfying.*

1. *Minimization problem, $\alpha \geq 1$*
   solution $\leq \alpha \cdot \text{OPT}$

2. *Maximization problem, $\alpha \leq 1$*
   solution $\geq \alpha \cdot \text{OPT}$

10.7 An Approximation Scheme

1. Minimization problem
   $1+\epsilon$ approximation algorithm for every $\epsilon > 0$

2. Maximization problem
   $1-\epsilon$ approximation algorithm for every $\epsilon > 0$

A polynomial time approximation scheme (PTAS) is a $(1+\epsilon)$-approximation with a runtime that is polynomial in the instance size. Here follows a couple of examples of run times and whether or not they are algorithms polynomial in the instance size.

$$\begin{array}{ll} \frac{n}{\epsilon}, 2^{\frac{n}{\epsilon}} n, n^{\frac{1}{\epsilon}}, n^{2^{\frac{1}{\epsilon}}} & \text{ok} \\ (\frac{1}{\epsilon})^n & \text{not ok} \end{array}$$

A fully polynomial time approximation scheme (FPTAS) is a PTAS where the run time is polynomial is $\frac{1}{\epsilon}$ too.

Proof 1 *We have a integer valued maximization problem such that* OPT $> 2^{Poly(instance)}$ *for all instances. Now choose $\epsilon = \frac{1}{2^{Poly(instance)+1}}$. Assume there exists a FPTAS, i.e. there exists a $(1-\epsilon)$-approximation algorithm with the running time $Poly(instance, \frac{1}{\epsilon})$.*

$$(1-\epsilon)OPT > OPT - \epsilon 2^{Poly(instance)} = OPT - \frac{2^{Poly(instance)}}{2^{Poly(instance)+1}} > OPT - 1$$

Our approximation is therefore an optimal solution with the running time $Poly(Poly(instance_{unity}))$. The algorithm is hence pseudo polynomial. □

10.8 Approximation of Knapsack

An approximation algorithm can often be created by rounding off the least significant bits of the input numbers. The amount of rounding is determined by $\frac{1}{\epsilon}$ and some error threshold.

Here follows a $(1-\epsilon)$-factor algorithm for the Knapsack problem. The solution $A \geq (1-\epsilon)\text{OPT}$ and the allowed run time is $Poly(n, \frac{1}{\epsilon})$. Now assume that all $u_i \leq B$, if not, throw the item away since it cannot be picked. Then take the largest $u_i \approx \frac{n}{\epsilon}$ and scale all weights by $\frac{1}{\epsilon}$ and turn into an integer.

$$f = \frac{\epsilon \cdot u_{\max}}{n}$$

And the new weight then becomes.

$$u_i' = \left\lfloor \frac{u_i}{f} \right\rfloor$$

And the running time is $n \sum_{i \in U} u_i'$ where each u_i' is bounded by n so we get a pseudo polynomial algorithm.

Proof 2 *Let*

OPT	$\equiv$	*optimal solution*
OPT'	$\equiv$	*value of* OPT *solution with* u_i' *weight function*
A'	$\equiv$	OPT*imum solution to out rounded instance with utility function* u_i'
A	$\equiv$	*Value of out solution with respect to original utilities*

The following holds

$$A' \geq \text{OPT}' \qquad\qquad A \geq fA'$$

From which this follows

$$\begin{aligned} \frac{u_i}{f} &\leq u_i' + 1 \\ u_i &\leq fu_i' + f \\ \text{OPT} &\leq f\text{OPT}' + fn \\ &\leq fA' + fn \\ &\leq A + fn \end{aligned}$$

Which gives us the result

$$\begin{aligned} \text{OPT} - fn &\leq A \\ \text{OPT} - \sum u_{max} &\leq A \end{aligned}$$

And since $u_{max} \leq \text{OPT}$ *we get* $\text{OPT} - \epsilon\text{OPT} \leq A$ *which proves that our approximation is within a factor* $(1-\epsilon)$. □

10.9 Traveling Salesman Problem (TSP)

For som dimension d, we got vertices $a_1, a_2, \ldots, a_n \in \mathbb{R}^d$. Find the shortest length tour visiting all the vertices. The distance function is the vector norm defined as

$$x, y \in \mathbb{R}^d : \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

Lemma 1 *An optimum TSP tour is a non self-intersecting polygon on* $a_1, a_2, \ldots, a_n$.

- *Lets start by showing that the tour is a polygon. The example tour* $a_1, a_2, a_5, a_3, a_2, a_6, \ldots$ *visits the vertex* a_2 *multiple times and since the triangle inequality holds, i.e.*

  $$distance(a_3, a_6) \leq distance(a_3, a_2) + distance(a_2, a_6)$$

  We can remove the vertex a_2 *from the tour and thereby decreasing the total length which means that the tour wasn't optimal to start with.*

  This gives us that the tour is a polygon.

- *What if we have crossing edges? Remove them and reconnect the two components. This will decrease the length of the tour so the optimal solution has no crossings. The decrease in length is given by the triangle inequality which can be applied to the vertices and the midpoint.*

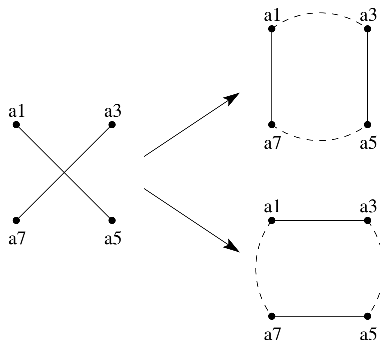

Figure 1: Remove crossing and reconnect components

10.10 Polynomial Time Approximation Scheme for TSP

Create a bounding box for all vertices. That is, create the smallest axis parallel box containing all the points. Now, without loss of generality, assume the width and height are $L = 4n^2$. Assume further, without loss of generality, that $n = 2^k$.

Approximate by creating a unit grid for the bounding box and move points to center of the square.

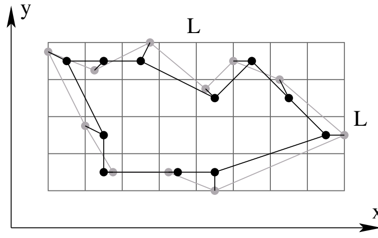

Figure 2: Approximation of TSP tour. Gray tour is exact and black approximated.

Lemma 2 OPT *is the original optimal solution and let* OPT′ *be the optimum solution, i.e. the same tour, for the rounded instance.*

$$\forall \epsilon > 0 : \mathrm{OPT}' \leq (1+\epsilon)\mathrm{OPT}$$

Which holds if n is big enough, $n > \frac{1}{\epsilon}$. Going to the center and going back to the original vertex adds a maximum distance of 4 for each square.

$$\begin{aligned} \mathrm{OPT}' &\leq \mathrm{OPT} + 4n \\ &\leq \mathrm{OPT} + \frac{4n^2}{n} \\ &\leq \mathrm{OPT} + \frac{\mathrm{OPT}}{n} \\ &= (1 + \frac{1}{n})\mathrm{OPT} \end{aligned}$$

Divide the bounding box into four square areas and place in a tree, see figure 3. Keep dividing and adding square areas so that the leaf nodes of the tree are unit squares. The height of the tree will become $\log(4n)$.

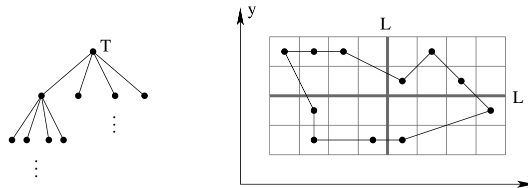

Figure 3: Division into square areas

In each box we solve the problem of minimizing path lengths with the given entry and exit points. The minimum path that covers all points will be non self-intersecting due to the lemma.

Add entry and exit portals to each unit square, see figure 4. Edges will be rounded to enter/exit from the closest portal. Lets create m portals on each

side of the square and have double portals on the corners. Pick $m = O(\frac{\log n}{\epsilon})$. Enumerating all pairings of portals then takes time

$$2^m \equiv 2^{O(\frac{\log n}{\epsilon})} \equiv n^{O(\frac{1}{\epsilon})}$$

Which is polynomial in n.

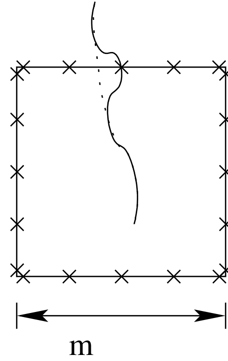

Figure 4: Entry and exit portals on square

Assumption 1 *There is a tour of cost at most* $(1+\epsilon')\mathrm{OPT}$ *such that the tour enter or exit a square from the portals. Proof will be presented on next lecture.*