Avancerade algoritmer DD2440

Homework C

Jim Holmström - 890503-7571

November 24, 2011

Exercise 1. Multiply the following two polynomials using FFT.

$$p(x) = x^3 + 3x^2 + x - 1$$
$$q(x) = 2x^3 - x^2 + 3$$

If you use a recursive version of FFT, you only need to show the top-most call to FFT (and FFT inverse), what recursive calls are made from the top-most level, what those calls return and how the final result is computed.

Solution. Algorithm:

Define $r = pq$
$p = \sum_{i=0}^{m} p_i x^i$
$q = \sum_{i=0}^{m} q_i x^i$
$r = pq = \sum_{i=0}^{m} \sum_{j=0}^{m} p_i q_j x^{i+j} = \{\text{Cauchy product}\} = \sum_{k=0}^{2m} \left(\sum_{i=0}^{k} p_i q_{k-i} \right) x^k = \sum_{i=0}^{2m} r_k x^k$
Where c_k is the convolution between p_i and q_j
Now using the ordinary shortcut in the fourier domain to calculate the convolution using the convolution theorem.

* $\hat{p} = \mathcal{F}\{p_i\}$ and $\hat{q} = \mathcal{F}\{q_j\}$ using fft.
* $\hat{r} = \hat{p} \cdot \hat{q}$, using termvise multiplication.
* $\mathcal{F}^{-1}\hat{c}$

Represent the polynomails p, q as vectors of their coefficients where the constant coefficient is the first element $p = (-1, 1, 3, 1, 0, 0, 0, 0)$ and $q = (3, 0, -1, 2, 0, 0, 0, 0)$ (note the zero padding)

The MATLAB code would have been: `ifft(fft([-1,1,3,1,0,0,0,0]).*fft([3 0 -1 2 0 0 0 0]))` wich yields `[-3 3 10 0 -1 5 2 0]`

Now showing the first steps by hand using the Cooley-Tukey FFT algorithm:
Using the 8^{th} root of unity as base $e^{2\pi i \frac{k}{8}}$ The principle of Cooley-Tukey:

$\hat{x}_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{n}{N} k} = \{\text{Break it up in even and odd part}\} = \sum_{m=0}^{N/2-1} x_{2m} e^{-2\pi i \frac{2m}{N} k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-2\pi i \frac{2m+1}{N} k} = \sum_{m=0}^{N/2-1} x_{2m} e^{-2\pi i \frac{2m}{N} k} + e^{-2\pi i \frac{1}{N} k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-2\pi i \frac{2m}{N} k}$ Where to two sums are halfsized fft's and running the same argument again until we hit size 1 wich trivially has itself as fft.

In our case for p we have $\hat{p}_k = [\text{fft}([-1, 3, 0, 0])_{2k} + (\sqrt{2} - \sqrt{2}i)^k \cdot \text{fft}([1, 1, 0, 0])_{2k}]$ $\hat{q}_k = [\text{fft}([3, -1, 0, 0])_{2k}, (\sqrt{2} - \sqrt{2}i)^k \cdot \text{fft}([0, 2, 0, 0])_{2k}]$

Pointwise multiplication on $\hat{p}$ and $\hat{q}$ yields $\hat{r}$ ifft are almost the same thing as fft except for that the root of unity goes in the other direction that is the minus sign is replaced with a plus.

ifft the same way as above both with interchanged sign yields

$r = [-3, 3, 10, 0, -1, 5, 2, 0]$

Exercise 2. Let $A = \{a_{i,j} : a_{i,j} = a_{i-1,j-1} \bigwedge a_{i,1} = a_{i-1,n}\}$ for $i, j \in [2, n]$. Give an $O(nlog(n))$-time algorithm for computing Ax where x is a vector of length n

Solution. Note that all indices in the following equations are beginning at 0 and taken modulo n The matrix A is circulant matrix.

Lemma 0.1. $A \in Circulant \Rightarrow A_{i,j} = A_{i+k,j+k} \forall k \in \mathbb{Z}$

Proof. Repeating the property $A_{i,j} = A_{i-1,j-1}$ together with $a_{i,0} = a_{i-1,n-1}$[1] $k \in \mathbb{N}$ times yields $A_{i,j} = A_{i-k,j-k}$ then exchanging $(i-k, j-k) \leftrightarrow (i, j) \Rightarrow A_{i,j} = A_{i+k,j+k}$ showing that it works for both directions and thus one can instead say $k \in \mathbb{Z}$ □

Lemma 0.2. $A \in$Circulant $\Rightarrow Ax = a * x$ [2] where a is the first colomn of A

[1]To make it work modulo n
[2]Circular convolution

Proof. $a*x = \{\text{Since convolution is commutative}\} = (x*a)_k \overset{def}{=} \sum_{i=0}^{n-1} x_i a_{k-i} = \sum x_i A_{k-i,0} =$ $\{\text{Using lemma}\} = \sum x_i A_{k,i} = \sum A_{k,i} x_i \overset{def}{=} (Ax)_k = Ax$ □

We also are going to use the convolution theorem: $\mathcal{F}\{c * x\} = \mathcal{F}\{c\} \cdot \mathcal{F}\{x\}$ [3]

The algorithm:

* Pickout a from A wich is $O(n)$
* Calculate $\hat{a} = \mathcal{F}\{a\}$ and $\hat{x} = \mathcal{F}\{x\}$ with fft where both a and x is of length n, fft is known to have implementations running at $O(nlogn)$
* $\widehat{Ax} = \hat{a} \cdot \hat{b}$ still having the same length, pointwise multiplication runs at $O(n)$
* $Ax = \mathcal{F}^{-1}\{\widehat{Ax}\}$ with ifft, similary runs in $O(nlogn)$

The algorithm gives us the vector Ax in $O(nlogn)$ operations. □

Exercise 3. Show how to formulate the Vertex Cover problem as an integer linear program (with a polynomial number of constraints).

Solution. Vertex Cover problem: Find the minimal vertex cover C in a graph G with vertices V and edges E

Define $\bar{x} \in \{0,1\}^n$ as

$$x_i = \begin{cases} 1 & \text{if vertex-}i \in C \\ 0 & \text{else} \end{cases}$$

and $\tilde{E}$ as

$$e_{ij} = \begin{cases} 1 & \text{if there is an edge between vertex-}i\text{ and vertex-}j \\ 0 & \text{else} \end{cases}$$

$\tilde{E}^T = \tilde{E}$ (undirected graph) $\bar{x} \geq \bar{0}$

With the constraint "An edge of the graph has at least on vertex in the vertex-cover set connected to it" $e_ij = 1 => x_i = 1$ `OR` $x_j = 1$ or more mathematical notation $e_{ij} = 1 \Rightarrow x_i + x_j \geq 1$ to get a lower amount of constraints.

Define the vector $c = (1, 1, .., 1)$ with length n, representing that each vertice gets the same weight.
We want to minimize the number of vertices in the cover that is $\sum x_i = |\hat{x}|_1 = c^T x$

To sum up we have:
$min\{c^T x\}$

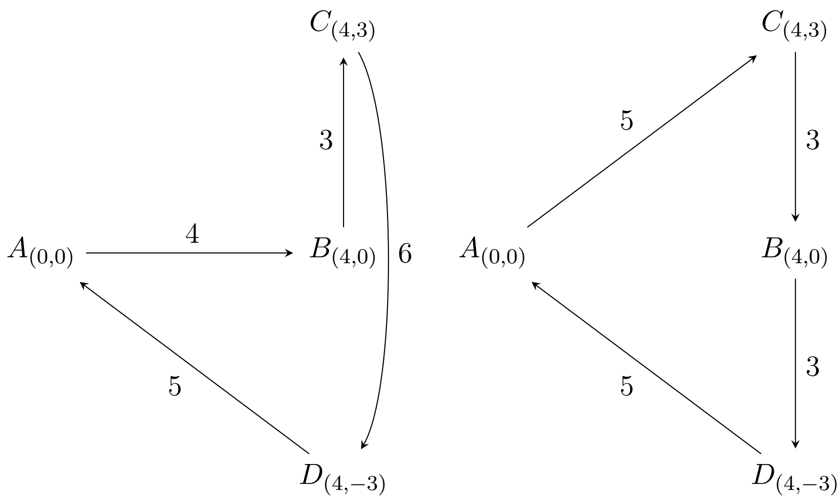
[3]Where $\cdot$ is pointwise multiplication, that is treating $\mathcal{F}\{x\}$ and $\mathcal{F}\{c\}$ as discrete functions.

with constraints $\{e_{ij} = 1 \Rightarrow x_i + x_j \geq 1\}$
$x \in \{0, 1\}^n$

Both the cost-function and constraints are linear and the domain is integer wich gives us a integer linear program.
Since $\tilde{E}$ is of size $O(n^2)$ the number of constraints is $O(n^2)$ thus polynomial number of linear constraints.

Exercise 4. Give an example of Euclidian TSP where the nearest neighbor heuristic fails to find the optimal solution.

Solution. The nearest neighbor is a greedy algorithm and will sometimes choose to walk to a city that would have been better to walk to in a later stage, giving a shorter total walk.
This simple example that gets the essence of the problem of the greedy approach of the algorithm, here starting at city A [4]:



(NN) $(Better)$

(NN): $\Sigma = 18$
(Better): $\Sigma = 16$

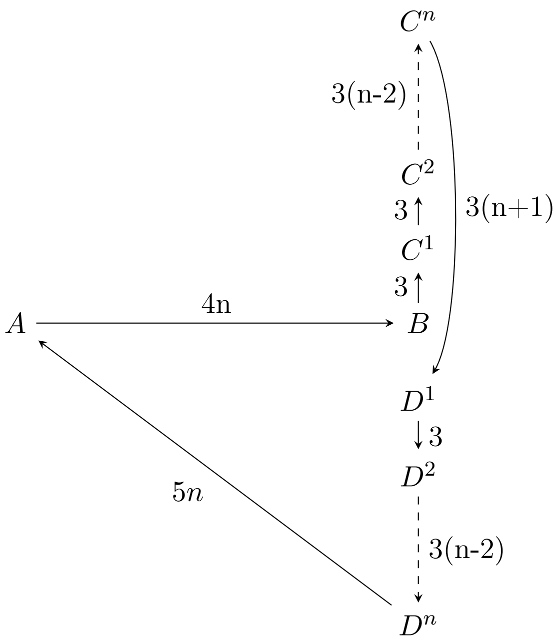
(NN) solution $>$ (Better) solution
Showing that this example of Euclidian TSP when starting at city A has a non optimal solution since $\exists$ a better solution. $\square$

Exercise 5. Generalize the result for nearest neighbor (NN) as follows. Find a constant $K > 1$ and an increasing function $f(n)$ such that, for each n there is a Euclidian TSP instance with $f(n)$ cities for which NN, starting in city 1 will

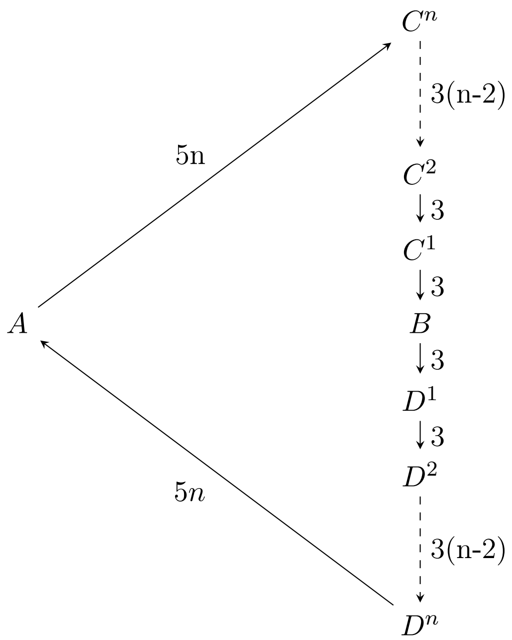
[4] Doesn't matter if we went to D instead of C from B by symmetri-reason, just relabel $C \leftrightarrow D$

yield a solution that is at least K times larger than the optimum solution. You get 2 bonus points if K is such that Christofides' algorithm is guaranteed to do better on these instances.

Solution. Choosing A as city "1" and constructing the TSP instances as below with $f(n) = \underline{2n+2}$ cities. We assume that $n > 0$.



(NN)



$(Better)$

(NN): $\Sigma = 4n + 3n + 3(n+1) + 3(n-1) + 5n = 18n$
(Better): $\Sigma = 5n + 3n + 3n + 5n = 16n$

(NN) $= K\cdot$ (Better) $\leq$ K $\cdot$ (OPT) $K = \frac{(NN)}{(Better)} = \frac{18n}{16n} = \underline{\frac{9}{8}} > 1$

B is always the closest city from A [5] The walk will always be $K = \frac{9}{8}$ worse then a better solution and therefore it is at least $\frac{9}{8}$ times worse then the optimal solution.

[5] Doesn't matter if we went to D^1 instead of C^1 from B by symmetri-reason just relabel $C^i \leftrightarrow D^i$