CS 787: Advanced Algorithms

DP-Based Approximations

Instructor: Dieter van Melkebeek

We discuss approximation algorithms based on Dynamic Programming techniques. Through Makespan and Knapsack, we see common ideas such as the dual view of parameters and bucketing and rounding values to a smaller distinct number of values, both of which can help facilitate the use of DP to approximate a solution. The Traveling Salesman Problem is also explored; the Euclidean version of TSP has a DP-based approximation algorithm.

1 Makespan

Definition 1. (Makespan Problem) *Given n jobs with duration $t_1, t_2, ..., t_n \in \mathbb{N}$ and m identical machines, the goal is to schedule all n jobs without preemption such that the latest finish time is minimized.*

The problem allows a simple greedy approximation algorithm: Take the jobs one by one and schedule the next job on the machine wth the earliest finish time so far. We leave it as an exercise to show that this simple greedy algorithm yields a factor-2 approximation. By ordering the jobs from largest to smallest, one can show that the approximation factor improves to 4/3. We won't give the argument, but immediately switch to a DP approach that allows us to realize every factor $1+\epsilon$ in polynomial time for every positive constant ϵ, i.e., a polynomial-time approximation scheme (PTAS).. The degree of the polynomial will grow with $1/\epsilon$. We point out that a fully polynomial-time approximation scheme (FPTAS) does not exist unless $\mathsf{P} = \mathsf{NP}$. This follows because the standard reduction from 3-SAT to Makespan only needs values of t_i that are bounded by a polynomial in n.

The basic idea for the PTAS is to relate Makespan to a bin packing problem and then perform a binary search for an approximately optimal solution by using a polynomial-time algorithm that solves the bin packing problem using DP.

We can look at this problem from another perspective, making finishing time an input, and number of machines needed an output. $\#BINS(I, t) =$ the smallest m for which I (a set of jobs) can be scheduled on m machines and finish in time no later than t. The optimal solution of the original problem can then be expressed as $OPT(I, m) = \min\{t : \#BIN(I, t) \leq m\}$.

1.1 Factor $(1 + \epsilon)$ approximation algorithm

In the following, we construct an approximation algorithm with factor $(1 + \epsilon)$, for any fixed $\epsilon > 0$.

The **Key Ingredient** is a strongly polynomial time algorithm $A(I, t, m)$ with the following behavior ($\forall \epsilon > 0$).

1. $OPT(I, m) \leq t \implies A(I, t, m)$ reports success.

2. If $A(I, t, m)$ reports success, then it also outputs a valid solution with finish time $\leq (1 + \epsilon)t$.

Leaving $A(I, t, m)$ as a black box for now, we can perform a binary search for the "smallest" t such that $A(I, t, m)$ reports success. Start with a lower bound of

$$l = \max \left(\max(t_i), \frac{1}{m} \sum_{i=1}^{n} t_i \right)$$

In other words, we know that OPT will be at least the greater of the time it takes for the longest job to run, or the optimal time it would take to run all jobs on m machines with (100% efficient) preemption and parallelization, i.e. no machine is ever left idle while a job remains. An upper bound is

$$u = \max(t_i) + \frac{1}{m} \sum_{i=1}^{n} t_i$$

To see this, consider an optimal schedule, and further consider any machine which finishes at time OPT in this schedule. The last job to finish on this machine is j'.

- If j' is the only job scheduled on this machine, then its duration must be at least the duration of any of the other jobs. i.e., it has duration $t_{j'} = \max(t_i)$, and $OPT = \max(t_i)$ in this case.
- If j' is not the only job on this machine, then let its start time be t_0. Every machine must be fully scheduled at least to time t_0, because if that weren't the case, j' could be moved to one of the machines that finished before t_0 and the original schedule would not be optimal, which is a contradiction.

  (Note, if there are several machines that finish at time OPT, then moving j' as above would not reduce the global finish time. However, if this process is repeated (using a new j' each time) eventually the above condition will be reached.)

  If every machine is fully scheduled to at least time t_0, then $\frac{1}{m} \sum t_i \geq t_0$. Then we have:

$$\begin{aligned} OPT &= t_{j'} + t_0 \\ &\leq \max(t_i) + t_0 \\ &\leq \max(t_i) + \frac{1}{m} \sum_{i=1}^{n} t_i \end{aligned}$$

In either case, the upper bound holds. Also note that $u \leq 2l$. This will be useful later when analyzing the complexity of the scheme. Now, starting with interval $[l, u]$ which contains OPT, perform a binary search for the smallest t such that $A(I, t, m)$ reports success. At each iteration we will have a new interval which contains OPT whose size is half that of the interval on the last iteration.

Eventually, the interval size will become less than (ϵl) (because l is fixed and the size is strictly decreasing). At this point, we have interval $[l', u']$ and,

$$\begin{aligned} u' - l' &\leq \epsilon l \\ u' &\leq l' + \epsilon l \\ &\leq l' + \epsilon l' \\ &\leq (1 + \epsilon) OPT \end{aligned}$$

Because $u' \geq OPT$ at every iteration, we know that $A(I, u', m)$ will output success and output a valid schedule. Further, because $u' \leq (1+\epsilon)OPT$, we know that this schedule will have stopping time $\leq (1+\epsilon)^2 OPT$, which is good enough because we can make ϵ arbitrarily small.

The number of steps in the binary search is $\log_2\left(\frac{\text{size of the initial interval}}{\text{size of the stopping interval}}\right)$. Our initial interval had size at most l ($u \leq 2l$, from earlier). The stopping interval had size at most ϵl.

$$\begin{aligned}
\#Steps &= \log_2\left(\frac{u-l}{u'-l'}\right) \\
&\leq \log_2\left(\frac{l}{u'-l'}\right) \\
&\leq \log_2\left(\frac{l}{\epsilon l}\right) + 1 \\
&= \log_2\left(\frac{1}{\epsilon}\right) + 1
\end{aligned}$$

Thus the number of steps does not depend upon the bit-length of the input, only ϵ. As long as there actually is a polynomial time algorithm $A(I, t, m)$ as described, then we have shown a strongly polynomial algorithm to approximate the optimal solution of Makespan to within any arbitrarily small but fixed factor.

1.2 Construction of algorithm A

The following key lemma will be used throughout the construction:

Lemma 1. *If the number of distinct t_i in instance I is $\leq k$, there exists an algorithm B that computes $\#BINS(I, t)$ exactly in time $O(n^{2k})$ and outputs a corresponding schedule.*

Proof. We will construct an algorithm based on dynamic programming. The idea is to first reduce the problem by considering possible schedules for a single bin (machine). Note that since we have a constant number of possible job lengths, we can represent the subproblem of jobs to be scheduled on a single machine as $(n_1, n_2, .., n_k)$ where $n_i =$ number of jobs of duration t_i, for $1 \leq i \leq k$. The number of these subproblems is upper bounded by the number of ways to set each n_i to a number between 0 and n, the total number of jobs. Setting k values each between 0 and n can be done in $(n+1)^k$ ways, so the $\#Subproblems \leq (n+1)^k$.

By brute force, we can find the set S of all subproblems which can finish within time t on a single machine. Again the size of S and the time it takes to construct is $\leq (n+1)^k$.

$$\#BINS(I, t) = \begin{cases} 1 & \text{if } I \in S \\ 1 + \min_{I' \in S}(\#BINS(I \setminus I', t)) & \text{otherwise} \end{cases}$$

Using memoization, $O(n^k)$ values need to be calculated (t is fixed, $\leq (n+1)^k$ subproblems). Also, at most $O(n^k)$ other subproblems need to be considered for each one to compute the minimum term. Thus, the total runtime is $O(n^k n^k) = O(n^{2k})$. □

In order to make full use of Lemma 1, we need to drive down the number of distinct job lengths to a constant k. The first observation to make is that if we truncate the job lengths, we stand to significantly reduce the number of distinct lengths under consideration. Consider intervals

$\left[\frac{t}{(1+\epsilon)^i}, \frac{t}{(1+\epsilon)^{i-1}}\right)$, where t is the target bin size, and $i \in \mathbb{Z}^+$. All jobs in this interval will be truncated to its left endpoint, $\frac{t}{(1+\epsilon)^i}$, as illustrated in Figure 1. Notice that for any valid schedule for these truncated jobs, the corresponding schedule with true job lengths is no more than a factor of $(1+\epsilon)$ longer. Also, we cannot have jobs with length greater than t, since that would immediately imply that the given bin packing problem is unsolvable (this situation never occurs, due to our selection of the lower bound for the binary search in Section 1.1).

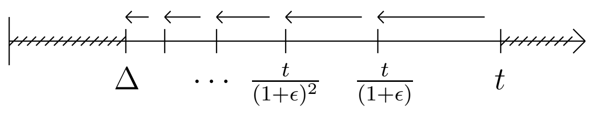


Figure 1: Job truncation for Makespan

From the above, we have a potential algorithm sketch:

- Transform the original instance $I \rightsquigarrow \widetilde{I}$ by truncating job lengths as described above
- Solve $\widetilde{I}$ (report failure if schedule is not found using $\leq m$ machines)
- Apply this schedule to the original jobs, I

Claim 1. *This algorithm satisfies the required properties of algorithm A.*

Proof. Listed are the properties and how they are satisfied.

1. $OPT(I,m) \leq t \implies A(I,t,m)$ reports success.

   If I can be scheduled in some way on m machines, then certainly $\widetilde{I}$, which has the same jobs with possibly shorter durations can as well, and a brute force strategy would be sure to find such a schedule.

2. If $A(I,t,m)$ reports success, then it also outputs a valid solution with finish time $\leq (1+\epsilon)t$.

   Let B_j be the set of jobs on machine j in the returned schedule, and $T(I,j)$ be the total processing time of machine j on schedule I.

$$\begin{aligned}
T(I,j) &= \sum_{i \in B_j} t_i \\
&\leq \sum_{i \in B_j} (1+\epsilon)\tilde{t_i} \\
&= (1+\epsilon) \sum_{i \in B_j} \tilde{t_i} \\
&= (1+\epsilon) T(\widetilde{I}, j) \\
&\leq (1+\epsilon)t
\end{aligned}$$

   Thus, $FinishTime(I) = \max_{j \in [k]} T(I,j) \leq (1+\epsilon)t$ as required.

□

So far, so good. But unfortunately, this is not a strongly poly-time algorithm. The number of truncated items, k, is still not constant. This is evident since for any given instance, we can always construct another instance that uses more of the infinitely many rounding points, by adding a job which would be truncated to a different running time than any current job.

Thus, we must modify our approach to drive down the number of truncated items to a constant. We will follow the intuition that smaller jobs are more conducive to being distributed evenly across machines, and begin by scheduling the largest jobs first.

For the time being, we forget about all jobs with size $\leq \Delta$, where Δ will be appropriately chosen. Then the number of distinct $\tilde{t_i}$ is $k \simeq \log_{1+\epsilon}(\frac{t}{\Delta})$. Since we want k to be constant, take $\Delta \geq \delta t$ for some constant δ. Now we'll argue that by setting δ appropriately we will have a strongly poly-time algorithm.

First, we truncate the t_i in the original instance I as discussed earlier, but we only consider the jobs whose (original) time falls above Δ. Call this instance $\widetilde{I}$. After we find an exact solution to the instance $\widetilde{I}$, we still need to deal with the smaller jobs. We distribute these greedily, but never place a job to make a bin fuller than $(1+\epsilon)t$ in I. We open new bins as needed.

```
A($I$, $t$, $m$)
(1)    Transform $I \rightsquigarrow \dot{I}$ by dropping $\{t_i \in I : t_i \leq \Delta\}$
(2)    Transform $\dot{I} \rightsquigarrow \widetilde{I}$ by rounding remaining $t_i$ to $\frac{t}{(1+\epsilon)^j}, 0 \leq j \leq k$
(3)    Schedule $\widetilde{S} = B(\widetilde{I}, t)$
(4)    **if** $\#BINS(\widetilde{S}) > m$ **then return** failure
(5)    Schedule $S$ = schedule for $I$ corresponding to $\widetilde{S}$ (minus small jobs)
(6)    Greedily add small jobs to $S$, keeping bin size $\leq (1+\epsilon)t$, open new bins if
       needed
(7)    **if** $\#BINS(S) > m$ **then return** failure
(8)                        **else return** success + S
```

Theorem 1. *Given an algorithm B that satisfies Lemma 1, and setting $\delta \leq \epsilon$,*

1. *$OPT(I, m) \leq t \implies A(I, t)$ reports success.*
2. *$A(I, t)$ reports success $\implies A(I, t)$ outputs a valid schedule with finish time $\leq (1+\epsilon)t$.*
3. *A runs in strongly polynomial time.*

Proof. To see that (2) is satisfied by A, note that line 8 must be executed in order for A to report success. Reaching that line requires a schedule using number of bins $\leq m$, with each filled to a size $\leq (1+\epsilon)t$.

We prove (1) by showing its contrapositive. Consider the case where failure is returned at line 4. If algorithm B, which uses exact DP over a restricted number of subproblems, cannot find a schedule for only the largest jobs in time t, then there is no way that $OPT \leq t$. Now consider the case where failure is returned at line 7. It follows that at some point we opened up the $(m+1)^{\text{st}}$ bin because we tried to schedule a small job that did not fit in any of the existing bins of size up to $(1+\epsilon)t$. Because $\delta \leq \epsilon$, we know that the m bins were filled to a level greater than $(1+\epsilon)t - \delta t \geq t$.

Of course, if m bins are full to at least t while more jobs still need to be scheduled, we certainly know that $OPT > t$.

Thus in the final algorithm we will choose $\delta = \epsilon$, and this will cause $k = \log_{1+\epsilon}(\epsilon^{-1})$ which is constant as required.

It is also clear to see that (3) is satisfied, because B runs in polynomial time once k is constant, and we have successfully restricted k to a constant with our transformations of I. □

1.3 Summary of Techniques

We employed several different ideas for the above DP-based approximation algorithm to take shape.

Dual view of parameters. The original problem gave the number of machines as input and time as an objective. $\#BINS$ took time as input and minimized the number of machines.

Approximate binary search. Searching to within some "close enough" range of our target.

Use DP when k small. When the number of possible values for a parameter (job durations in this case) is small, we can use Dynamic Programming to efficiently solve the reduced space of subproblems.

Bucketing & Rounding. Helps to reduce the possible parameter values.

Handle most important first. In this case, the larger jobs were most important. In general it may be harder to tell what is important.

2 Knapsack

As mentioned, we cannot hope to get a FPTAS for the Makespan problem unless $P = NP$. Knapsack, on the other hand, has a FPTAS. We begin with the definition of the Knapsack problem.

Definition 2 (Knapsack). *Given n items with integer weights $w_1, w_2, \ldots, w_n \geq 0$, integer values $v_1, v_2, \ldots, v_n \geq 0$, and a knapsack with capacity W, find subset $S \subseteq [n]$ such that $\sum_{i\in S} w_i \leq W$ and $\sum_{i\in S} v_i$ is maximized.*

We also define the notion of pseudo-polynomial time, which requires that running time be polynomial in the values of the inputs as opposed to the size of the problem instance.

Definition 3 (Pseudo-Polynomial Time). *Polynomial time, where integers in the input string are given in unary representation, rather than in binary.*

2.1 Exact DP Approach

The first step in creating our approximation algorithm for Knapsack is to create a pseudo-polynomial time algorithm to solve it. To achieve this step, we will apply dynamic programming. We will build a table A such that $A(i, v)$ is the minimum knapsack capacity for which $\max_{S\in P([i])}(\sum_{s\in S} v_s) = v$. A full table would consist of $n \times (\sum v_i + 1)$ entries. The exact recurrence is as follows:

$$A(i, 0) = 0, (i = 1, .., n)$$
$$A(i, v) = \min\left(A(i-1, v - v_i) + w_i, A(i-1, v)\right)$$

It can be filled in order $A(1,1)$, $A(2,1)$, ..., $A(n,1)$; $A(1,2)$, $A(2,2)$, ..., $A(n,2)$; ... until the largest v is found such that $A(n,v) \leq W$. This v is the optimal value of the objective function, call it V.

Each evaluation of A takes $O(1)$ time, and $O(nV)$ of these are performed. So the total running time of this exact DP algorithm is $O(nV)$, making it a pseudo-polynomial time algorithm. Similarly, one can develop an exact algorithm that runs in time $O(nW)$.

2.2 Approximate approach

The idea is similar to the one used in designing approximation scheme for the Makespan problem. We first bucket the values by dividing them by a suitably large factor and rounding down. We solve the instance exactly and argue that the solution we obtained cannot differ much from the optimal solution.

Let I be the original instance. Transform $I \rightsquigarrow \widetilde{I}$ such that $\tilde{v_i} = \lfloor \frac{v_i}{f} \rfloor$. Note that this also gives

$$\frac{v_i}{f} - 1 < \tilde{v_i} \leq \frac{v_i}{f}$$

We can solve instance $\widetilde{I}$ to get solution $\widetilde{S}$ using our exact algorithm above, and use that solution as an approximate solution to I. The following analysis will reveal the factor f to scale by in order to make this a FPTAS.

$$\begin{aligned} v(\widetilde{S}) &= \sum_{i \in \widetilde{S}} v_i \\ &\geq f \sum_{i \in \widetilde{S}} \tilde{v_i} \\ &\geq f \sum_{i \in S} \tilde{v_i} \\ &\geq f \sum_{i \in S} \left(\frac{v_i}{f} - 1 \right) \\ &= OPT - f|S| \\ &\geq OPT - fn \end{aligned}$$

We would like a $1 - \epsilon$ approximation algorithm, or $v(\widetilde{S}) \geq (1-\epsilon)OPT$. This holds provided that

$$\begin{aligned} OPT - fn &\geq (1-\epsilon)OPT \\ f &\leq \frac{\epsilon OPT}{n} \end{aligned}$$

The optimal solution has a lower bound of the maximum item value, $\max(v_i)$, as long as we assume that all input items satisfy $w_i > W$, which is reasonable because we could throw out any items not satisfying and get the same answer to Knapsack. So setting $f = \frac{\epsilon \max(v_i)}{n}$ satisfies the above, making this a $(1-\epsilon)$ approximation algorithm. The runtime is $O(n\widetilde{V}) = O(n\frac{\max(v_i)}{f}) = O(\frac{n^3}{\epsilon})$, which makes this a FPTAS.

3 Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) asks the following: Given cities and the distances between each pair of cities, how can one visit each city exactly once and return to their home city in the shortest possible route.

Definition 4 (TSP)**.** *Given a complete graph $G = (V, E)$ and edge weights $w : E \to [0, \infty)$, find a Hamiltonian cycle of minimum weight or report "no HC".*

Theorem 2. *TSP is NP-hard to approximate to within any polytime computable factor $f(n)$ ($n = |V|$).*

Proof. Consider the following polynomial time reduction from the Hamiltonian Cycle (HC) problem to TSP. Given an arbitrary graph $G(V, E)$ for the HC problem, construct an instance K_n, w of TSP by making G a complete graph K_n for TSP and defining w s.t. $w(e) = 1$ if $e \in E(G)$ or $n \cdot f(n)$ otherwise. Note that as $f(n)$ is polytime computable, this is indeed a polytime reduction.

We know that there is a HC in any complete graph, so an OPT value for the TSP instance will always exist. Specifically, if there is a HC in G, then $OPT \leq n$, and if there is not, then $OPT \geq n \cdot f(n)$. Thus if we had a factor within $f(n)$ approximation algorithm for TSP, we could reduce any HC instance to a TSP instance and use the approximation algorithm to distinguish between "yes" and "no" cases of HC. □

3.1 Metric TSP

Definition 5 (Metric-TSP)**.** *We define Metric-TSP to be a special case of the TSP problem where the associated weight function w is a metric on V. w is a metric on the set V iff $(\forall u, v, x \in V)$,*

$$\begin{aligned} w(u,u) &= 0 \\ w(u,v) &= w(v,u) \\ w(u,x) &\leq w(u,v) + w(v,x) \end{aligned}$$

Now we will give polynomial time factor 2 and factor 3/2 approximation algorithms for the Metric-TSP problem. This is in strong contrast with the general TSP problem which is NP-hard to approximate within any reasonable factor (however factors as bad as 2^n for the general TSP can be computed in polynomial time). Recently, for the special case of graph metrics, i.e., when the metric coincides with the shortest path distance in an unweighted graph, the approximation factor of 3/2 has been improved to $\frac{14(\sqrt{2}-1)}{12\sqrt{2}-13} \approx 1.461$ using an LP-based approach.

Lemma 2. *Given an instance of TSP, (G, w), $MST(G, w) \leq OPT$ where MST is the sum of the weights of a minimum spanning tree in G.*

Proof. Consider the Hamiltonian cycle on G with minimum total cost OPT. Removing an arbitrarily chosen edge from the tour results in a tree with cost at most OPT. This tree also has to be of cost at least $MST(G, w)$. □

Theorem 3. *Metric-TSP can be approximated within a factor of 2 in polynomial time.*

Proof. Let (G, w) be an instance of the Metric-TSP problem. Find a minimum spanning tree on the instance.

Now consider an Euler(ish) tour (visits every edge at least once, instead of only once, and ends in the origin vertex). Building such a tour takes polynomial time. Because it is built on a tree, the tour will visit every edge twice. Therefore the cost of the tour is exactly twice the cost of the MST and hence $\leq 2OPT$.

Now convert the tour into a Hamiltonian cycle by short-cutting. To do this, you simply visit the vertices in the same order as in the tour, but travel directly between vertices, so you only have n edges in your cycle, and none of them are revisited. An example of both an original tour and the short-cutted HC is shown in Figure 2. Because w is a metric, the triangle inequality holds. Thus traveling between two vertices directly in the HC costs no more than traveling between the two vertices with possible intermediate vertices in the original tour. This means that the cost of the HC given by this algorithm is $\leq 2OPT$. In other words, this is a factor 2 approximation algorithm for TSP.

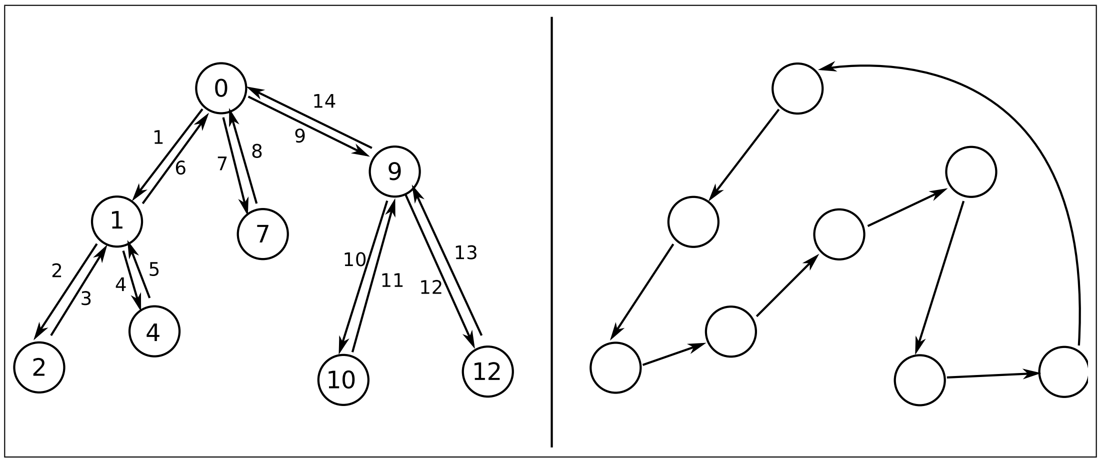


Figure 2: Euler(ish) Tour and Short-cutting

□

Theorem 4. *Metric-TSP can be approximated within a factor of* $3/2$ *in polynomial time.*

Proof. The algorithm is a modified version of the above algorithm. We start by finding the MST for the given instance and improve the method of short-cutting to generate a Hamiltonian tour.

Consider the MST obtained above. Since the sum of degrees of all the vertices is exactly twice the number of edges in the tree we must have even number of vertices of odd degree. If we match these vertices, pairing them and connecting the members of each pair by an edge, the resulting graph will have only even degree vertices. Therefore there must exist a proper Euler tour in this graph which visits every edge exactly once. The cost of the resulting graph is $MST(G, w) +$ (the weight of added edges for the matching).

Note that a minimal cost matching will have cost at most half the cost of the optimal Hamiltonian cycle. This follows from the fact that the optimal HC for just the vertices to be matched has cost less than the optimal HC for the entire graph. Further an HC on these vertices can be viewed as the union of two disjoint matchings: M_1, being every other edge starting from the first edge in the cycle, and M_2, all other edges in the cycle. It follows that at least one of M_1 and M_2 has cost at most half the cost of the cycle itself, which is again at most half the cost of the optimal cycle on the entire graph.

There is a polynomial time algorithm for finding minimal matching in a graph. Once we have this we can see that the we can obtain a graph which has an Euler tour of cost at most $3/2$ times the cost of the optimal Hamiltonian path in the given complete graph. We will short-cut this Euler tour as we did in the previous algorithm to reach our approximately optimal Hamiltonian cycle without increasing the cost. □

3.2 Euclidean TSP

Next we consider the Euclidean restriction of TSP. As the name suggests, all the vertices are points in an n-dimensional space, in general. For simplicity, we only consider the case where the points are on a plane. We give a PTAS for TSP in this case. The algorithm can be extended to the general case.

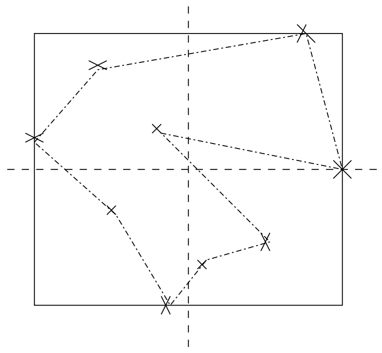


Figure 3: Division into subproblems.

First we present an intuitive idea of the algorithm. We will use divide and conquer strategy. To divide into subproblems, we consider a bounding box and divide it into four rectangles. We would like to solve the problem for each rectangle separately and then combine the solutions. Our base case is when we have only one point inside a rectangle. But we have to be a slightly careful in defining our subproblems because the subproblem is not the same as our original problem. We do not find the optimal tour for the rectangles separately but instead we should find a tour that can be connected to tours in other parts. In other words, we need to take into account the fact that a Hamiltonian tour can "enter" and "leave" a rectangle at various points. Now we can have many subproblems for each rectangle we consider. We find optimal solutions for all these subproblems and then combine them in an optimal way.

So we know what our subproblems look like but still there are some more issues that we need to consider.

- Some points may be very close to each other and we may need need to go very deep into the recursion to have these points in different rectangles. To bound the recursion depth, we will merge points which are very close into one point. That will be good enough for an approximation algorithm.

- The number of subproblems should be small. The number of subproblems depends upon two factors - the number of portals per side where the tour can enter the rectangle and the number of crossings per side. We can bound the number of portals per side by choosing some points on each side, suitably placed, where the tour can enter.

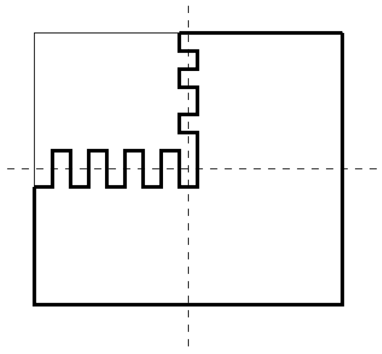


Figure 4: Boundary Crossings

  Bounding the number of crossings is more difficult. We can have instances of TSP, as shown in Figure 4, in which any reasonable approximation must cross the boundaries many times. We handle this by changing our boundaries. If required, we move a boundary line sideways so as to have less number of crossings.

This is the high-level intuition. It takes a fair amount of effort to distill an efficient algorithm from it, which we do in the next section.

4 Euclidean TSP – the details

We provide the details of the approximation algorithm in three parts:

1. *Perturbing the input.* We modify the input to make sure each point lies in the center of a cell within a course grid. This will limit the depth of our recursion tree. This also will merge points that are close enough together to lie in the same grid cell.

2. *Building the quadtree.* We build a randomly shifted quadtree such that there exists a nearly optimal tour which crosses each node of the quadtree only a few times and only on a few specified *portals.* This will limit the number of subproblems we will need to deal with and thus enable us to employ dynamic programming.

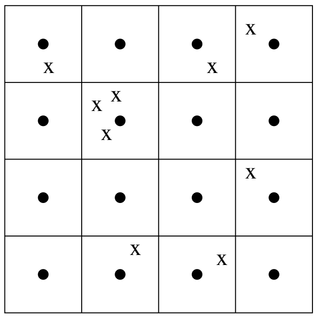
3. *Dynamic programming.* We find the optimal tour under our restrictions using dynamic programming. This gives us a nearly optimal solution for the original problem that, we claim, will be close enough to satisfy the fixed value for ϵ. If we set the algorithm's parameters appropriately, it will run in polynomial time.

4.1 Perturbing the Input

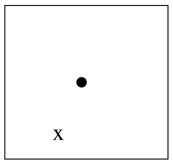
We define our bounding box as the smallest possible square that contains all the points in the problem. A trivial lower bound for the optimal tour is $2L$, where L is the length of the sides of the bounding box.

$$OPT \geq 2L$$

Our input may contain points that are arbitrarily close together. This means that in dividing our problem instances into smaller subproblems, we may need to recurse deeply before we separate some points from seach other. We solve this by merging the problematic points into a single point. Our method for doing this is to place a grid over the bounding box with spacing $s = \frac{L}{2^k}$, where k is some integer constant. We then place every point contained in a given grid cell at the center of that cell. The figure below shows this arrangement. The perturbed problem will have only seven points.



What is our loss in accuracy as a result of this perturbation? Consider the optimal path that goes through the actual point, labeled "x", in the following figure.



The maximum error in our solution to the perturbed instance is twice the maximum distance from a point to the center of its cell, for each point. Thus, we have

$$\widetilde{OPT} \leq OPT + n\sqrt{2}s = OPT + n\sqrt{2}\frac{L}{2^k} \leq (1 + \frac{\sqrt{2}}{2}\frac{n}{2^k})OPT = (1 + \Theta(\epsilon))OPT$$

The last equality above holds provided $k = \log(\frac{n}{\epsilon} + \Theta(\epsilon))$.

So, given a solution to the perturbed problem with cost $\tilde{C}$, we can easily construct a solution to the original problem with cost

$$C \leq \tilde{C} + n\sqrt{2}\frac{L}{2^k} \leq \tilde{C} + \Theta(\epsilon)OPT \leq (1 + \Theta(\epsilon))OPT$$

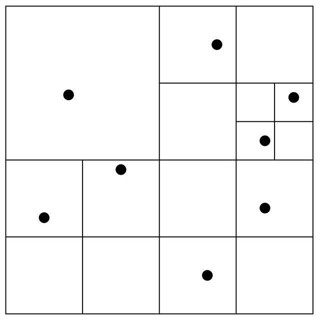
Based on the above arguments, we will set $k = \log(\frac{n}{\epsilon} + \Theta(\epsilon))$.

4.2 Building the Quadtree

This section discusses the method in which we build the recursion tree for our approximation algorithm. In the last section, we saw how the size of the recursion tree was limited by perturbing the problem instance. The specific goal at this stage is to make further restrictions to ensure that we get a small enough number of subproblems such that dynamic programming can be employed. We first look at the standard quadtree.

Definition 6 (Standard quadtree)**.** *A standard quadtree is the recursion tree obtained from the process in which a square is recursively partitioned into four equal subsquares until each square contains at most one point.*

Example: The following figure shows the result of building a standard quadtree on an instance of the Euclidian TSP problem.



$\boxtimes$

If we apply this procedure to the bounding box defined for our Euclidian TSP problem, we have the following two properties:

1. recursion depth $\leq k$
2. number of nodes is $O(kn) = O(n \log(n/\epsilon))$

The first property follows from the way we perturbed the problem instance in the previous section. We chose to align all points to a grid of size 2^k by 2^k and thus it takes k steps of halving the size of the subproblem in each dimension to guarantee there is at most one point in the cell.

The second property can be proven by attributing each node in the recursion tree to some point, and then showing than each point has $O(k)$ nodes associated with it. Attribute each node with one or more points inside it to one of its contained points. Each point can have at most k nodes assigned to it in this way, since this is the maximum depth of recursion. There also may exist nodes with no points, but the immediate parent of such a node must contain one or more points (otherwise we wouldn't have recursed); we can attribute this empty node to one of these points contained in the immediate parent. Each point can be attributed at most four additional nodes in this way, and thus the number of nodes attributed to each point is $O(k)$.

The subproblems we encounter in our algorithm will be characterized by:

- a node in the quadtree
- the tour's entry points and exit points along the boundary of the node (The order in which these points are used is also specified.)

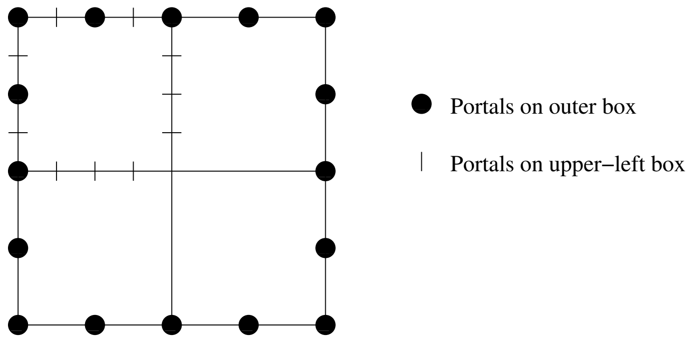
A solution to one of these subproblems will be a union of paths that realizes the given entry and exit points in the correct order and visits all points in the node at least once. (Recall that the use of shortcuts relaxes the "exactly once" requirement.)

We'd like to apply dynamic programming, so we must have a sufficiently limited number of possible subproblems. We already showed that the number of nodes in the quadtree is manageable.

However, the specification of the entry and exit points must be further restricted. We first restrict the entry and exit points to a small number of *portals* that exist along the border of the cell. We will also restrict the number of entry/exit pairs that can be specified for each cell. We cover these restrictions in more detail in the following sections.

4.2.1 Portals

Portals restrict the location of the entry and exit points that may be specified in a subproblem of our Euclidian TSP algorithm. We define $p+1$ equally spaced portals along each edge of a quadtree node's cell. This splits the sides of the cell into p segments of equal length. The following figure shows some portals defined at two different levels in the quadtree, where $p = 4$.



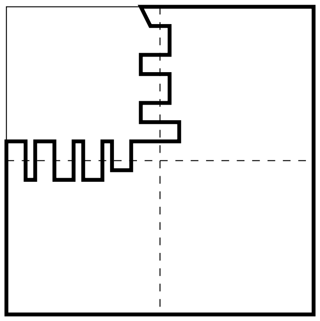
Note that since p is a power of two, portals at a lower level in the quadtree (the outer box in the figure) are also portals at high levels.

The choice of p presents a tradeoff between accuracy and length of description. We will later come back to how to choose an appropriate p.

4.3 Restricting the number of entry and exit points

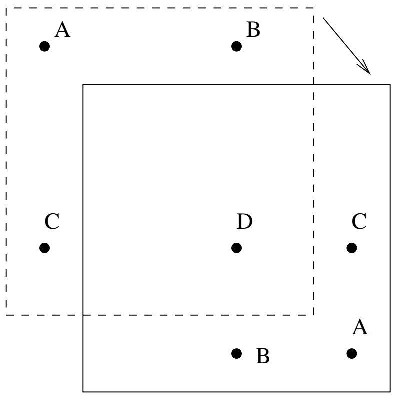
The other restiction we place on our subproblems is the number of entry and exit points that can be specified. We'll limit subproblems such that the maximum number of entry/exit pairs that can be given is f. We now have two parameters p and f that determine how we will restrict entry/exit point locations and quantity per subproblem. We would like to choose values for these parameters such that the subproblems are few enough to apply dynamic programming and yet there exists an optimal tour for the restricted problem that is close enough to OPT. This section discusses how we can use a random procedure to ensure with high probability that limiting the number of entry/exit point pairs will still keep our solution close enough to OPT.

Consider the following pathological case for restricting the number of entry/exit point pairs. The thick line shows the actual optimal path.



Applying our restriction which limits the number of entry and exit points may add a significant amount to the path returned by the algorithm for a case like this. We will show that adding randomness to the way that we build our quadtree will alleviate this situation.

The method we will use is to pick a point v on the grid we defined earlier uniformly at random and build the quadtree using the selected point as its center. The following figure shows how we are in effect shifting the bounding box by v. We will "wrap around" points that fall outside of the shifted grid. The following figure illustrates this. Note the manner in which the points A, B, and C have wrapped around.



We now consider the following for use in the next lecture. Consider a fixed line l in our grid. We can assign each line in the grid a level based on the level in the recursion tree in which it is used to split a cell into smaller subproblems. For example, the horizontal line halfway down the grid is the only horizontal level 1 line. The lines one quarter and three quarters of the way down the grid are level two lines, etc.

Now, considering our fixed line l, what is the probability it is at some level i? Note that there are 2^{i-1} lines at level i and there are a total of 2^k lines, so:

$$Pr[\text{line } l \text{ is at level } i] = \frac{2^{i-1}}{2^k}, \quad i \geq 1$$

The optimal solution to our restricted problem $\tilde{OPT_v}$ will be the length of an optimal tour that crosses the nodes of the quadtree shifted by v at most f times per cell, and only at the portals (of which there are $p+1$ per side of a cell at any level in the quadtree). We will be able to find $\tilde{OPT_v}$ exactly using dynamic programming. What we want is that this is close to $\tilde{OPT}$ (and OPT) for most v. The following lemma, which we will prove in the next lecture, is the key to this property.

Lemma 3 (Structure Lemma)**.**

$$E_v[\widetilde{OPT_v} - \widetilde{OPT}] \leq c\left(\frac{1}{f-7} + \frac{k}{p}\right)L$$

for some constant $c > 0$.

Now, if we choose

$$f = \Theta(\frac{1}{\epsilon}) \qquad p = \Theta(\frac{k}{\epsilon}) = \Theta(\frac{1}{\epsilon}\log(\frac{n}{\epsilon}))$$

then we have with high probability

$$\widetilde{OPT_v} \leq (1 + \Theta(\epsilon))OPT$$

This follows from the fact that the term $(\frac{1}{f-7} + \frac{k}{p})$ from the structure lemma is $\Theta(\epsilon)$ and then by applying Markov's inequality.

4.3.1 Dynamic Programming

We can use dynamic programming to exactly solve our perturbed and restricted version of the original Euclidian TSP instance. Think of our algorithm as a recursive process over the nodes in the quadtree. At each node we will

1. Solve the subproblems.
2. Consider all valid combinations of the subproblem solutions and pick the one of minimum distance.

The number of subproblems is

$$\leq (4p+1)^f = O((\frac{1}{\epsilon}\log(\frac{n}{\epsilon})^{O(\frac{1}{\epsilon})})$$

The number of nodes in the quadtree is

$$O(n\log(\frac{n}{\epsilon}))$$

And the amount of local work per subproblem is

$$O((\text{number of subproblems per node})^4)$$

We thus realize a running time of $O(\frac{n}{\epsilon}\log(\frac{n}{\epsilon})^{O(\frac{1}{\epsilon})})$. It is possible to give the dynamic programming algorithm more hints and improve the running time to $O\left(\left(\frac{1}{\epsilon}\right)^{O(1)} n\log n + \left(\frac{1}{\epsilon}\right)^{O(\frac{1}{\epsilon})}\right)$, but that will not be discussed here.

Note that due to the random shift we used in building the quadtree, we have developed a randomized algorithm. The algorithm can be trivially derandomized by trying all possible choices for v, of which there are 2^{2k}. This adds a factor $\Theta((\frac{n}{\epsilon})^2)$ to the running time, which is undesirable for large instances.

4.4 Proof of the Structure Lemma

The Structure Lemma entails that we can limit the the number of subproblems for the dynamic programming algorithm without increasing the cost of solution of our PTAS to more than $(1 + \Theta(\epsilon))OPT$. We limit the number of subproblems in two ways.

Phase (A) First we limit the number of times the produced tour may pass from one node of the quad tree to an adjacent node to f.

Phase (B) Secondly we limit the positions in which the tour may pass from node to an adjacent node in the quad tree to $(p + 1)$ evenly spaced portals.

Let $\widetilde{\Pi}$ be an optimal tour of the TSP created in phase 1 of the PTAS by perturbing the points so that they all lie on centers of cells on a grid with spacing $\frac{L}{2^k}$. $\widetilde{\Pi}$ has cost $\widetilde{OPT}$.

4.4.1 Phase (A)

We show how we will adjust $\widetilde{\Pi}$ so that the resulting tour never crosses from one node of the quad tree to a neighbour more than f times. We will increase the length of the tour by no more than $c \cdot \frac{1}{f-7} \cdot \widetilde{OPT}$. To prove this we will use the following lemma:

Lemma 4 (Patching Lemma)**.** *Given a segment S of a grid line and a tour Π such that Π crosses S at least 3 times, we can locally change Π such that ≤ 2 of these crossings remain and no new crossings are introduced. The additional cost is $\leq 6|S|$.*

Proof. Let the number of crossings be t.

1. Break up an even number of crossings, leaving 1 or 2 of the t crossings. At the crossings that are not broken up, still introduce breaking points, but leave the crossing in place.
2. For each side of the grid line separately, match up successive breaking points by adding an additional edge between them. Do not add an edge to the breaking points where the crossing is not broken up. Now all breaking points have an even degree. The additional cost to the tour of adding these edges is $\leq 2|S|$.
3. To make the tour connected again, make two tours, one on either side of the edge line, through all breaking points on that side. The cost of adding this tour to the original tour is $\leq 4|S|$.

After this procedure, the breaking points form an Euler graph, so we can visit all the breaking points while only crossing S twice at an additional cost $\leq 6|S|$. And we have reduced the number of crossings of S to 2. □

Now consider a grid line l. We can reduce the number of crossings of l to less than f by applying the following algorithm:

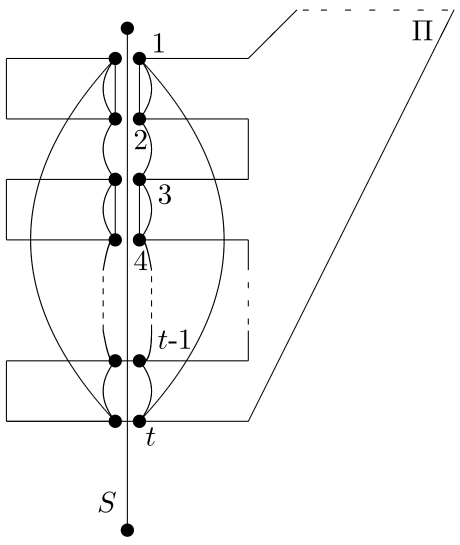


Figure 5: Illustration of the Patching Lemma. Note that all break points lie on segment S but have been drawn apart for clarity. The straight vertical edges connecting break points are the edges introduced in step 2. The curved edges connecting the break points on either side of S are the Euler tours produced in step 3.

```
Input: A grid line $l$, a tour $\Pi$.
Output: An adjusted version of $\Pi$ in which $l$ is crossed at most 4 times.
Fix($l$)
for $j = k$ down to $l$
    for all level $j$ segments $S$ of $l$
        if there are $\geq f$ crossings in $S$
            patch($S$)
```

Note that patching a segment S normally produces just 2 crossings on that segment but, if S lies on both sides of the bounding box as a consequence of wrapping around, we may have to apply patch to S on both sides of the bounding box.

Analysis. We call $P_{l,j}$ the number of times we patch a level j segment of line l. Note that if we are considering a vertical line l, $P_{l,j}$ only depends on the vertical coordinate (y) of our shift. We thus fix a y and calculate the expected cost of fixing a line l over all horizontal shifts x:

$$\begin{aligned}
E_x[\text{cost of fixing } l] &\leq \sum_{i=0}^{k} \Pr[\text{l is at level i}] \cdot E_x\left[\sum_{j=i}^{k} P_{l,j}\frac{6L}{2^k}|\text{l is at level i}\right] \\
&\leq \sum_{i=0}^{k}\left(\frac{2^i}{2^k}\cdot\sum_{j=i}^{k}\left(P_{l,j}\frac{6L}{2^j}\right)\right) \\
&\leq \sum_{j=0}^{k}\frac{6LP_{l,j}}{2^j 2^k}\sum_{i=0}^{j}2^i \\
&= \sum_{j=0}^{k}\frac{12LP_{l,j}}{2^k} = \frac{12L}{2^k}\sum_{j=0}^{k}P_{l,j}.
\end{aligned}$$

The last sum in our expression means the number of times we patch line l in total. Before patching there can be at most $f+1$ crossings and we only keep 2 of those after patching, so we delete $f-3$ crossings. The number of times we patch the line l therefore becomes upper bounded by $\frac{C_l}{f-3}$ with C_l the total number of crossings of $\widetilde{\Pi}$ with line l.

The expected cost for all vertical lines is then upper bounded by:

$$\sum_{l \in \text{Vertical lines}} \frac{12L}{2^k}\frac{C_l}{f-3}.$$

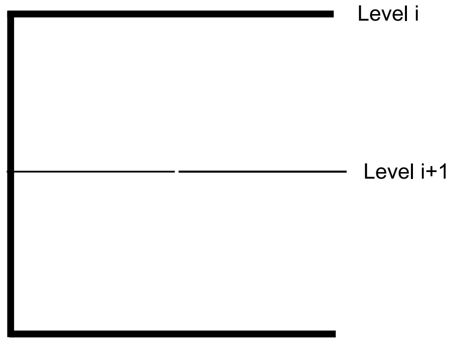
We apply the same argument for the horizontal lines and get an expected cost:

$$\frac{12}{f-3}\frac{L}{2^k}\sum_{l \in \text{All lines}} C_l.$$

The factor $\frac{L}{2^k}\sum_l C_l$ is equal to the length of $\widetilde{\Pi}$ in 1-norm. Recall that 1-norm is the sum of the absolute values of the differences in each coordinate of two points; the term above is thus equal to the number of cells between the two points. Because the Euclidean distance between two cells is $\leq \sqrt{2}$ the 1-norm distance between them, the whole equation is upper bounded by:

$$\frac{12\sqrt{(2)}}{f-3}\widetilde{OPT}.$$

Snag 1. When patching the horizontal lines, it might introduce crossings with the vertical lines. Luckily, it turns out these crossings are only at the corners and do not introduce additional cost.



Suppose we have a vertical line segment S at level $i+1$ [figure above: thin line] and we patch a horizontal line of a higher level [figure above: thick line]. This will remove crossings by adding edges parallel to A and B. Therefore, patching can only introduce crossings at the corners of segment S. We can solve this problem by again patching the vertical segment of width 0 thereby removing the crossings. At most 4 more crossings will be introduced after we apply the patching lemma. If we run the fixing subroutine with $f-4$ instead of f, we can take into account the 4 extra crossings. The cost factor derived above then becomes $\frac{12\sqrt{2}}{f-7}\widetilde{OPT}$.

Snag 2. When we were analyzing the cost of our fixing subroutine for the horizontal lines we assumed (as we did when analysing the vertical line fixing) that we could still relate to the $\widetilde{\Pi}$ tour. Because the vertical line fixing changed the tour, we have to argue that this does not introduce cost not accounted for. Recall that the changes made by fixing the crossings at the vertical lines were only made at the corners. Because there are at most 2 corners per segment, each time time our fixing subroutine is applied, we elimate at least $f-3-4=f-7$ of the crossings in $\widetilde{\Pi}$. Our cost analysis therefore remains valid if our denominator is $f-7$. (Notice the difference between this snag and the previous: because our previous snag already introduced the $f-7$ factor in the analysis, the validity of our argument follows automatically.)

Therefore, the expected cost introduced by phase (A) is upper bounded by:

$$\frac{12\sqrt{2}}{f-7}\widetilde{OPT}$$

4.4.2 Phase (B)

In phase B we move the crossings which we have not removed in phase A, to the nearest portal. Since the crossing which are introduced in phase A during the fixing rounds are at the corners and the corners are portals themselves, we do not have to consider them. The extra cost for moving the crossings is therefore upper-bounded by the cost for moving the crossing of $\widetilde{\Pi}$.

Now we fix a line l. We then calculate the cost for moving the crossings in $\widetilde{\Pi}$ to the portals of l:

$$E[\text{cost for moving crossings in } \widetilde{\Pi} \text{ on } l \text{ to the portals}] \leq \sum_{i=0}^{k}\left(\Pr[l \text{ is at level } i]\sum_{j=i}^{k}\left(\frac{L}{2^j p}C_l\right)\right)$$

Recall that the probability that line l is at level i is $\frac{2^i}{2^k}$. The term $\frac{L}{2^j p}$ is an upper bound on the distance our tour will need to be extended to go from the nearest portal to the crossing and back. Simplifying the expression we get:

$$E[\text{cost for moving crossings in } \widetilde{\Pi} \text{ on l to the portals}] \leq \sum_{i=0}^{k}\left(\frac{2^i}{2^k}\frac{2}{2^i}\frac{L}{p}C_l\right) = \frac{(k+1)L}{p2^k}C_l$$

In order to get the expected total cost, we still need to sum over all possible lines. Using the bound derived in phase A we get for the total cost of moving the crossing to the portals:

$$\frac{k+1}{p}\sum_{l}\left(\frac{L}{2^k}C_l\right) \leq \frac{\sqrt{2}(k+1)}{p}\widetilde{OPT}.$$

Summing the bounds on the total cost for phase (A) and (B) we obtain the bound to be proven (for a good choice of c). This concludes the proof of the Structure Lemma.