# 1 Lecture 3, November 4, 1999

We continue our study of the use of dynamic programming for fixed parameter versions of NP-hard optimization problems on graphs.

## 1.1 Longest path

Consider the following problem:

**$k$-path:** Does the graph contain a simple path of length $k$ (not repeating any vertex)?

A simple path of length more than $k$ includes a simple path of length $k$, so the $k$-path problem asks to find the long simple paths in a graph. It is well known that shortest paths between pairs of vertices can be found in polynomial time. However, finding longest paths is NP-hard, via a reduction from Hamiltonian cycle.

Observe that Hamiltonian cycle can be solved in time $(n-1)!n^c$ for some constant $c$, by enumerating all permutations on $n$ vertices starting at $v_1$, and then checking for each permutation whether every two cyclically adjacent vertices in the permutation are also adjacent in $G$.

Dynamic programming can be used to give a more efficient solution. We prepare a table $T$ with $n-1$ columns and $(n-1)2^{n-2}$ rows. Column $j$ corresponds to simple paths with $j$ edges. The rows are partitioned to $n-1$ blocks, numbered from 2 to $n$. Within a block $i$, row $l$ corresponds to a subset $S_l \subset V \setminus \{v_1, v_i\}$. Entry $(i,l,j)$ in $T$ is 1 if there is a simple path of $j$ edges starting at $v_1$, visiting all vertices in $S_l$, and ending at $v_i$. It is 0 otherwise. Using dynamic programming, we fill the table column by column. To determine the value of entry $(i,l,j+1)$, we consider all neighbors $v_h$ of $v_i$ in $S_l$, and set entry $(i,l,j+1)$ to 1 if for at least one such $v_h$, we had the entry $(h,l',j)$ equal to 1, where $S_{l'} = S_l \setminus \{v_h\}$. The graph has a Hamiltonian path if some entry in column $n-1$ is equal to 1, and a Hamiltonian cycle if it has a Hamiltonian path whose endpoints are adjacent in $G$.

When $k$ is small, $k$-path can be solved in time roughly $n^k$ by guessing the $k$ vertices on the path in order. The approach for Hamiltonicity can be used to cut down the running time to roughly $\binom{n}{k}2^k$ (saving a factor of roughly $(ck)^k$ for some $c>0$) by only guessing the subset of vertices visited by the path, and then using dynamic programming to find a Hamiltonian path in the subgraph induced by these vertices. However, even this running time is polynomial only when $k$ is a constant.

We now present two simple randomized algorithms that can solve $k$-path in polynomial time for values of $k$ that grow moderately with $n$.

To motivate the first of these algorithms, assume that the vertices of $G$ are numbered from 1 to $n$, and that we are interested only in *increasing paths*, in which the sequence of vertices along the path is monotonically increasing. It is not hard to see that dynamic programming can be used to find the longest increasing path in polynomial time. (For each $i$, maintain $l(i)$, the length of the longest increasing path ending at $i$. Then $l(i) = \max_{j<i}[l(j)+1]$ where the maximum is taken only over vertices $j$ that are neighbors of $i$.)

Our first randomized algorithm is as follows:

1. Randomly permute the names of the vertices.

2. Find the longest increasing path with respect to the permuted order.

Assume that $G$ has a simple path of length $k$. Then after step 1, there is probability $1/k!$ that this path forms an increasing sequence. Then step 2 will find this path. Hence the success probability of this algorithm is at least $1/k!$ (it can be larger, if there are several long paths), and it runs in polynomial time.

To amplify the success probability to $1 - e^c$ for an arbitrarily large $c > 0$, repeat the algorithm $c \cdot k!$ times, with independent coin tosses. The running time remains polynomial as long as $k = O(\log n / \log \log n)$.

The above algorithm has one sided error – it may fail to find a $k$-path even if the graph contains one, due to the need to stop the algorithm after a polynomial number of iterations.

The second randomized algorithms uses the method of *color coding*.

1. Randomly and idependently color each vertex with one of $k$ colors.
2. Use dynamic programming to find a length $k$ path that does not repeat the same color twice. This can be done in time proportional to $n2^k$, because all we need to remember in order to build such a path is the last vertex visited, and the subset of colors already visited. The rest of the the details are similar to the dynamic programming algorithm for Hamiltonian path.

Assume that $G$ has a simple path of length $k$. Then after step 1, there is probability $k!/k^k \simeq 1/e^k$ that all vertices of the path receive distinct colors. Then step 2 will find this path. Hence the success probability of this algorithm is at least $1/e^k$, and it runs in polynomial time.

To amplify the success probability to $1 - e^c$ for an arbitrarily large $c > 0$, repeat the algorithm $c \cdot e^k$ times, with independent coin tosses. The running time remains polynomial as long as $k = O(\log n)$.

Also this algorithm has one sided error. However, it can be derandomized, giving a somewhat more complicated deterministic polynomial time algorithm that always finds a path of length $k$ if it exists, when $k = O(\log n)$. The details of the derandomization are beyond the scope of this course.

### 1.2 Bandwidth

A square matrix is said to have bandwidth $k$ if all its nonzero entries are concentrated in a band of width $k$ around its diagonal. It is relatively easy to store and manipulate matrices with small bandwidth. For sparse matrices that have large bandwidth, one often seeks a permutation on its rows and a permutation on its columns that brings the matrix into a form with low bandwidth. When the matrix is symmetric, one often wants the same permutation to be performed on rows and on columns, preserving the symmetry.

Minimizing the bandwidth of a square symmetric matrix can be represented in graph theoretical terms. The matrics can be viewed as the adjacency matrix of an undirected graph, where a nonzero entry $A_{i,j}$ signifies the existence of an edge $(v_i, v_j)$. A permutation of row/column names corresponds to a permutation on the names of vertices. This leads to the following problem:

$k$**-bandwidth:** Does the graph have a linear arrangement (numbering the vertices from 1 to $n$) in which the length of the longest edge (measured in terms of difference in numbering of its endpoints) is at most $k$?

For fixed $k$, this problem can be solved in polynomial time using dynamic programming. The dynamic program for this problem requires some sofistication.

We wish to find a linear arrangement of bandwidth at most $k$. We will build it by a extending a partial linear arrangement (that includes the first $j$ vertices in the linear arrangement) by one more vertex, in all possible consistent ways. Namely, we need to add one more vertex, such that no edge so far has length more than $k$, and moreover, such that this vertex did not previously appear in the linear arrangement. This last condition is the tricky one – how do we know which vertices appeared in the prefix?

To solve the above problem, we make the following two observations:

1. A graph has bandwidth at most $k$ iff each of its connected components has bandwidth at most $k$. Hence we may w.l.o.g. assume that the graph is connected. This is a simple observation, but essential to the success of our algorithm.

2. The bandwidth of a graph is always at least $\Delta/2$, where $\Delta$ is the maximum degree of the graph. Hence $\Delta \leq 2k$. This observation helps us reduce the running time of the algorithm, though is not as essential as the first one.

From a prefix of a linear arrangement of length $j$, we shall record only:

1. The last $k$ vertices, which we call the "buffer" vertices.

2. The set of "dangling" edges – those who have one end point in a buffer vertex and the other in a yet unvisited vertex. In the dynamic programming algorithm, we only maintain those partial linear arrangements in which no edge skips over the buffer vertices.

The point is that this representation implicitly includes all vertices in the prefix – those are the vertices which after the removal of the dangling edges still have a path to some buffer vertex (due to observation 1 above). The amount of information we need to record is $n^k$ possibilities for the buffer vertices, and at most $2^{2k^2}$ possibilities for the dangling edges (by observation 2 above), giving a polynomial running time when $k$ is fixed.

(Homework in Handout 2)

**References:**


- N. Alon, R. Yuster, U. Zwick. "Color Coding". *Journal of the ACM* 42, 844–856, 1995.

- J. Saxe. "Dynamic programming algorithms for recognizing small bandwidth graphs in polynomial time". *SIAM Journal on Algebraic Methods* 1 (1980), 363–369. 1980.