W4231: Analysis of Algorithms

11/9/99

- Matching
- Dynamic Programming

Bipartite Graph

A bipartite graph is a (typically undirected) graph $G = (V, E)$ where the set of vertices can be partitioned into subsets V_1 and V_2 such that each edge has an endpoint in V_1 and an endpoint in V_2.

Often bipartite graphs represent relationships between different entities: clients/servers, people/projects, printers/files to print, senders/receivers . . .

Often we will be given bipartite graphs in the form $G = (L, R, E)$, where L, R is already a partition of the vertices such that all edges go between L and R.

Matching

A matching in a graph is a set of edges that do not share any endpoint.

In a bipartite graph a matching associates to some elements of L precisely one element of R (and vice versa).

A matching can be seen as an **assignment**.

Optimization problem: given a bipartite graph, find the matching with the largest number of edges.

Algorithm

Similar in spirit to Ford-Fulkerson:

Proceed in phases, start with an empty matching.

At each phase, we either find a way to get a bigger matching, or we get convinced that we have the largest possible matching.

Alternating Paths

Let $G = (L, R, E)$ be a bipartite graph, $M \subseteq E$ be a matching.

A vertex is **covered** by M if it is the endpoint of one of the edges in E.

An **alternating path** is a path of odd length that starts with a non-covered vertex, ends with a non-covered vertex, and alternates between using edges not in M and edges in M.

Use of Alternating Path

If we find an alternating path, then we can increase the size of M by discarding all the edges in of M which are in the path, and taking all the others.

If there is no alternating path, then the matching is optimal.

Proof

If there is no alternating path, then the matching is optimal

Suppose M is not optimal, that is, some other matching M^* has more edges. We prove that M must have an alternating path.

Let $G = (L, R, E')$ be the graph where E' contains the edges that are either in M or in M^* **but not in both** (i.e. $E' = M \oplus M^*$).

Every vertex of G has degree at most two. Furthermore if the degree is two, then one of the two incident edges is coming from M and the other from M^*.

Since the maximum degree is 2, G is made out of paths and cycles. Furthermore the cycles are of even length and contain each one an equal number of edges from M and form M^*.

But since $|M^*| > |M|$, M^* must contribute more edges than M to G, and so there must be some path in G where there are more edges of M^* than of M. This is an augmenting path for M.

An Algorithm

An augmenting path, if it exists, can be found with a variation of BFS in $O(m + n) = O(m)$ time.

The algorithm has at most $\min\{|L|, |R|\} \leq n/2$ phases, so the total time is $O(mn)$.

Using Max-Flow

Direct all edges from L to R. Give them capacity 1.

Add a vertex s and connect it to every element of L using capacity-one edges.

Add a vertex t and connect every vertex of R to t using capacity-one edges.

Now we have a network.

The maximum flow is equal to the maximum matching

An augmenting path can be found in $O(m+n)$ time, and there are no more than $n/2$ phases, so the running time is as before.

In fact it is the **same algorithm**.

Why the Max Flow is equal to the Max Matching

- For a matching with k edges there is a flow of cost k.

  Let $(u_1, v_1), \ldots, (u_k, v_k)$ be the edges in the matching. Then $s \to u_i \to v_i \to t$ are all augmenting paths, and running the Ford-Fulkerson algorithm with this choice of paths we get a flow of cost k.

- For an integer flow of cost k there is a matching of cost k.

  Consider $M = \{(u,v) : f(u,v) > 0\}$. It has to be a matching: if f is integer and $f(u,v) > 0$ then $f(u,v) \geq 1$, and there cannot be a v' such that $f(u,v') > 0$, because there is only capacity 1 entering u, and by flow conservation at most one can get out. Similarly for v.

- If capacities are integers, then the max-flow is integer.

  The Ford-Fulkerson algorithm does produce an integer flow.

Dynamic Programming

Dynamic programming applies to problems that could be solved using divide and conquer.
(Where the task of solving a big problem instance can be reduced to the tasks of solving smaller problem instances.)

Instead of a "top-down" (or bigger-to-smaller) recursive algorithm, with dynamic programming we design "bottom-up" (or smaller-to-bigger) iterative algorithms.

We first solve all the smallest instances, then the slightly larger ones. . . Along the way, we keep in memory (in a look-up table) all the solutions found so far.

Fibonacci Numbers

This is more an "analogy" than an "example."

Recall that Fibonacci numbers are defined as

$F_0 = 0,\ F_1 = 1,\ F_{k+2} = F_k + F_{k+1}$

A "divide and conquer" algorithm would be

```
Fib-Rec(n)
    if (n=0) then return 0
    else if (n=1) then 1
    else return Fib-Rec(n − 1) + Fib-Rec(n − 1)
```


Running Time

The time $T(n)$ that the algorithm Fib-Rec() takes to compute the n-th Fibonacci number is given by the recurrence relation

$$T(n) = T(n-1) + T(n-2) + O(1),\ T(1) = O(1),\ T(0) = O(1)$$

and if we just consider the number $R(n) < T(n)$ of recursive calls generated by the algorithm on input n, we have

$$R(0) = 1,\ R(1) = 1,\ R(n) = R(n-1) + R(n-2)$$

So $T(n) > R(n) > F_n > (1.6)^{n-2}$

The algorithm takes time exponential in n.

Reason for Huge Running Time

Algorithm Fib-Rec() keeps computing over and over the same thing.

The computation of Fib-Rec(11) involves 55 recursive calls to Fib-Rec(2), each one producing a recursive call to Fib-Rec(1) and one to Fib-Rec(0).

Efficient Version

```
Fib-DP(n)
  if (n=0) then return 0
  else if (n=1) then return 1
  else
    create integer vector F[] with n entries
    F[0] := 0; F[1] := 1;
    for i = 2 to n
      F[i] := F[i−1] + F[i−2]
    return F[n]
```

Running time is $O(n)$.

Source of the improvement: now we compute F_2 (and all other intermediate values) only once.

Shortest Paths

Let $G = (V, E)$ be a directed graph, and let $w_{u,v} \geq 0$ be the weight of edge (u, v). Let $V = \{1, \ldots, n\}$.

Idea for recursive solution to find shortest path between s and t, $s, t \in V$: the shortest path between s and t is

- either the shortest of the paths from s to t not passing through n
- or the shortest path from s to n concatenated with the shortest path between n and t.

Recursive Version

```
SP-REC(s, t, V, E, w)
  if |V| = 0 then return w_{s,t}
  else
    k = |V|
    l := SP (s, t, V − {k}, E, w)
    l' := SP (s, k, V − {k}, E, w) + SP(k, t, V − {k}, E, w)
    return min{l, l'}
```

Algorithm SP(s, t, V, E, w) finds the length of the shortest of the paths between s and t, among those that can pass through the vertices of V. (If $(u, v) \notin E$, then assume $w_{u,v} = \infty$ in the implementation.)

Iterative Version — General Idea

Define a $n \times n \times (n + 1)$ matrix M.

Design an algorithm that will fill the matrix so that $M[s, t, k]$ is the length of the shortest of the paths between s and t, among those that can pass through the vertices $\{1, \ldots, k\}$.

Once the matrix is filled, the length of the shortest path from s to t is in $M[s, t, n]$.

So, once the matrix is filled, we have the length of the shortest paths between any two vertices.

Filling the Matrix

We know what is $M[s, t, 0]$: this is just the length of the edge from s and t, if it exists. (Otherwise we set it to ∞ by convention.)

The value of $M[s, t, k]$ can then be computed based on precomputed values of $M[\cdot, \cdot, k-1]$ since

$$M[s,t,k] = \min\{M[s,t,k-1],\ M[s,k,k-1] + M[k,t,k-1]\}$$

Algorithm

```
SP-DP(V, E, w)
  create n × n × (n + 1) matrix M
  for u := 1 to n
    for v := 1 to n
      if (u, v) ∈ E then M[u, v, 0] := w_{u,v}
      else M[u, v, 0] = ∞
  for k := 1 to n
    for u := 1 to n
      for v := 1 to n
        M[u, v, k] := min{ M[u, v, k − 1],
                           M[u, k, k − 1] + M[k, v, k − 1] }
```

Runs in time $O(n^3)$.

Reconstructing the actual paths

Together with M, we also create a matrix P of same dimension, such that $P[u, v, k]$ says which vertex comes before v in the shortest path from u to v among those that can pass through $\{1, \dots, k\}$.

Once we have the final M and P, we can reconstruct the shortest path from u to v backwards. Start from v, and look up $v' = P[u, v, n]$. Then look up $v'' = P[u, v', n]$ and so one. Eventually we get to u, by way of a shortest path.

Algorithm that also computes P

```
SP-DP-2(V, E, w)
  create n × n × (n + 1) matrices M and P
  for u := 1 to n
    for v := 1 to n
      if (u, v) ∈ E then M[u, v, 0] := w_{u,v}; P[u, v, 0] = u
      else M[u, v, 0] = ∞; P[u, v, 0] = u
  for k := 1 to n
    for u := 1 to n
      for v := 1 to n
        l_1 = M[u, v, k − 1]; l_2 = M[u, k, k − 1] + M[k, v, k − 1]
        if l_1 ≤ l_2 then
          M[u, v, k] := M[u, v, k − 1]; P[u, v, k] := P[u, v, k − 1]
        else
          M[u, v, k] := M[u, k, k − 1] + M[k, v, k − 1]; P[u, v, k] = k
```


Partitioning a Set of Integers

Given integers $a_1, \dots, a_n$ we want to find if there is a subset of them that sums to half to the total value.

That is, if there is $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i$.

More general problem (easier to solve recursively): given $a_1, \dots, a_n$ and B, we want to find if there is a subset $S \subseteq \{1, \dots, n\}$ such that

$$\sum_{i \in S} a_i = B$$

Recursive structure

We can reduce the task of solving the problem on an instance with n integers to the task of solving the problem on two instances with $n - 1$ integers.

The instance $a_1, \dots, a_n, B$ has solution iff at least one of the following condition holds:

- there is a solution for $a_1, \dots, a_{n-1}, B$ (there is a solution without a_n);
- there is a solution for $a_1, \dots, a_{n-1}, B - a_n$ (there is a solution with a_n).

Matrix etc.

Through that recursion, we always reduce the problem to instances of the form $a_1, \dots, a_k, B'$, where $1 \le k \le n - 1$ and $0 \le B' \le B$.

We get a dynamic programming algorithm by defining $n \times (B + 1)$ boolean matrix M with the intended meaning that $M[k, B'] = true$ iff the instance $a_1, \dots, a_k, B'$ has a solution.

Then $M[1, B'] = true$ iff $a_1 = B'$

And $M[k, B'] = M[k - 1, B'] \vee M[k - 1, B' - a_k]$.

Algorithm

```
SubsetSum-DP(a_1, ..., a_n, B)
  create n × (B + 1) matrix M
  for b := 1 to B
    M[1, b] := false
  M[1, a_1] := true; M[1, 0] := true
  for k := 2 to n
    for b := 1 to B
      M[k, b] := M[k − 1, b] ∨ M[k − 1, b − a_k]
  return M[n, b]
```

Note: the expression $M[k - 1, B' - a_k]$ is clearly undefined if $B' < a_k$. Whenever it is undefined, we consider it as $false$.

More precise implementation

```
SubsetSum-DP(a_1, ..., a_n, B)
  create n × (B+1) matrix M
  for b := 1 to B
    M[1, b] := false
  M[1, a_1] := true; M[1, 0] := true
  for k := 2 to n
    for b := 1 to a_k − 1
      M[k, b] := M[k − 1, b]
    for b := a_k to B
      M[k, b] := M[k − 1, b] ∨ M[k − 1, b − a_k]
  return M[n, B]
```


Getting the Subset

Suppose we have constructed M, and $M[n, B] = true$ and now we want to find the subset of integers whose sum is B.

```
ConstructSet(a_1, ..., a_n, B, M)
  S := ∅
  b := B
  for k = n down to 1
    if a_k == b then S := S ∪ {a_k}
      return S
    if (a_k > b) ∧ M[k − 1, b − a_k] then
      S := S ∪ {a_k}
      b := b − a_k
    return S
```