Parma University, Math. Department

Algorithms and Data Structures
All pairs shortest path

Marco Pellegrini

Spring 2002, Lecture 19

Summary

- All pairs shortest path problems,
  Floyd-Warshall algorithm
  Transitive closure of a graph.
- Material CLR 26.2

All-pair shortest path problem.

- Given a directed graph G=(V,E) and a weight function w:E $\rightarrow$R, compute for each pair of nodes the length of the shortest path connecting them in G.
- We saw Dijksrta's algorithm solving SSSP in time $O((|V|^2 + |E|))$ with a standard priority queue Q.
- Using a "smart heap" Dijkstra's algorithm runs in time $O((|V| + |E|)\log |V|)$.
- Using Fibonacci heaps Dijkstra runs in time $O(|E| + |V|\log |V|)$. (An advanced data structure)
- For the APSP problem we can just repeat SSSP code |V| times getting: $O(|V|^3 + |V||E|)$, $O((|V|^2 + |E||V|) \log |V|)$ and $O(|V||E| + |V|^2 \log |V|)$.
- Exercise: which implementation should you choose when $|E| = c|V|$, and when $|E|=c|V|^2$?

Negative weights and positive weights

- So, if only positive weights are involved we can solve APSP for *dense graphs*, $|E| = c|V|^2$ in time $O(|V^3|)$ by iterating Dijksta+array DS.
- We shall see now Floyd-Warshall's algorithm.
- FW runs in time $O(|V|^3)$.
- FW allows negative weights (but not negative cycles)
- FW uses the matrix representation of graphs, which is usually preferred for dense graphs.
- FW is quite simpler than Dijkstra.

Input and Output.

- The weights are encoded in a weight matrix w[1..n][1.. n] where w(i,j)=0 if i=j,
  w(i,j)= infinity if (i,j) is not in E,
  w(i,j) = weight of edge (i,j) otherwise.
- The Output will be a Matrix d[1..n][1..n] where at the end d(i,j) is the shortest distance between node i and node j in G. Or infinity if no such path exists.
- The shortest paths are encoded in a matrix p[1..n][1..n] where p(i,j) = Nil if i=j or there is no path between i and j. Otherwise p(i,j) = x where x is the index of the node before j in a shortest path between i and j.

Floyd-Warshall, the main idea.

- The main idea is to compute first shortest paths allowed to touch certain nodes, and gradually relax the condition so that at the end we can have any node in a shortest path, so we have the true S.P.
- d(i,i) = 0 by default.
- So take nodes i and j, with i$\neq$j. We allow only shortest paths that have no intermediate nodes between i and j. We have $d_0(i,j) = w(i,j)$.
- Now we assume inductively that we know all shortest paths that pass through nodes {1,2,..,k-1} and we allow now node k in the path.
- If k is not in the path (i....j) no need to update, if k appears, then the subpaths (i...k) and (k...j) do not contain k in their interior and we can compute the length of the new path.

main idea + code

- $d_k(i,j) = \min(d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j))$
- It works also when k is equal to i or j.
- We assume for the moment to have matrix D[0],..,D[n]
-
```
Floyd-Warshall(W)
  D[0] := W
  FOR k := 1 TO n DO
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        D[k](i,j) := min( D[k-1](i,j), D[k-1](i,k) + D[k-1](k,j))
      END FOR
    END FOR
  END FOR
RETURN D[n]
```


Observations.

- The code is obviously $O(|V|^3)$.
- The proof of correctness is no more than a restatement of the main idea.
- The code is rather simple and uses no auxiliary data structures.
- We used the assumption that there are no negative cycles implicitly when we assumed that the new node k can appear only once in a path from i to j.
- If there is a negative cycle containing k, then we get smaller and smaller weight by cycling in it, so we would pass through k many times.

The predecessor matrix.

- If we do no allow any intermediate node in a path then the predecessor matrix has this form;
- $p_0(i,j)$ = NIL if i=j or w(i,j)=infinity
- $p_0(i,j) = i$ if $i \neq j$ and p(i,j) < infinity.
- Now when we go from k-1 to k, if the shortest path length d(i,j) is not updated then also $p_k(i,j) = p_{k-1}(i,j)$.
- If the shortest path d(i,j) is updated then the new path will pass through k so: $p_k(i,j) = p_{k-1}(k,j)$.
- Exercise: incorporate in the code of Floyd-Warshall the lines that compute P, assume that we have an array of matrices d[0],..d[n].

Storage.

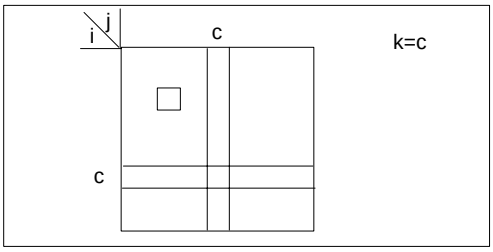
- So far we have used in FW algorithm an array of n n-by-n matrices, which make the storage $O(n^3)$.
- Can we do better, say $O(n^2)$? Look at this:

```
Floyd-Warshall2(w)
 d := w
 FOR k := 1 TO n DO
       FOR i := 1 TO n DO
         FOR j:= 1 TO n DO
            d[i,j] := min (d[i,j], d[i,k] + d[k,j]
       END FOR
     END FOR
   END FOR
  RETURN D.
```

- This code uses only 1 matrix, so storage is $O(n^2)$, but what does it compute?

Correctness of Floyd-Warshall2

- Before we would read values in matrix D[k-1] and write in matrix D[k].
- Now we read and write in the same matrix D.



Correctness of FW2 (cntd)

- Assume that FW2 simulates FW for k=1,..,c-1 and let us start two inner loops when k=c.
- The two inner loops visit all entries in D and for entry (i,j) they read the value in (i,j), read values in (c,j) and (i,c) and then write in (i,j).
- Any (i,j) is read and written only once while k=c, except the c-column and the c-row that are read many times.
- Things can go wrong only if FW2 writes something different from FW in the c-column or the c-row. But, since there are no negative cycles $D[c,c] = D_{c-1}[c,c]=0$, so from the formula, neither FW nor FW2 change any of the values in the c-column and the c-row. (end of proof).

Transitive closure

- In the transitive closure problem we are interested only in computing whether there is a path between node i and j in G.
- We can use the same scheme by induction on the index of the maximum intermediate node on a path between i and j. We use an array of Boolean matrices. $T_k[i,j]$ is 1 if there is a path from i to j passing through {1,...,k}, 0 if such path does not exists.
-

```
Transitive-closure((V,E))
 T0 = E
 FOR i := 1 TO n DO  T0[i,i] = 1 END FOR;
 FOR k := 1 TO n DO
   FOR i := 1 TO n DO
      FOR j:= 1 TO n DO
        Tk[i,j] := Tk-1[i,j] OR ( Tk-1[i,k] AND Tk-1[k,j])
END FOR, END FOR, END FOR
```


Homework

- HW5.2 CLR 26.2-5 Use the output of FW to decide whether there are cycles of negative weight in G.
- HW5.3 Modify FW2 so to detect negative cycles in G.

Conclusions

- We have seen a method to compute all pairs shortest paths in a weighted undirected graph with possibly negative weights.
- The method can be used to detect negative cycles.
- The method is efficient for dense graphs.
- Next lecture: SSSP and APSP for sparse graphs with negative weights (CLR 25.3 and 26.3)