Algorithms and Data Structures II. Lecture Notes:

Dynamic programming

Ásványi Tibor – asvanyi@inf.elte.hu

December 15, 2023

Contents

References

[1] Ásványi, T, Algorithms and Data Structures I. Lecture Notes
http://aszt.inf.elte.hu/∼asvanyi/ds/AlgDs1/AlgDs1LectureNotes.pdf

[2] Ásványi, T, Algorithms and Data Structures II. Lecture Notes: Elementary graph algorithms
http://aszt.inf.elte.hu/∼asvanyi/ds/AlgDs1/AlgDS2graphs1.pdf

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to Algorithms (Third Edition), *The MIT Press*, 2009.

[4] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[5] Narashima Karumanchi,
Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[6] Neapolitan, Richard E., Foundations of algorithms (Fifth edition), *Jones & Bartlett Learning*, 2015. ISBN 978-1-284-04919-0 (pbk.)

[7] Shaffer, Clifford A.,
A Practical Introduction to Data Structures and Algorithm Analysis, Edition 3.1 (C++ Version), 2011
(See http://aszt.inf.elte.hu/∼asvanyi/ds/C++3e20110103.pdf)

[8] Tarjan, Robert Endre, Data Structures and Network Algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics*, 1987.

[9] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++ (Fourth Edition),
Pearson, 2014.

[10] Ásványi Tibor, Detecting negative cycles with Tarjan's breadth-first scanning algorithm (2016)
http://ceur-ws.org/Vol-2046/asvanyi.pdf

1 Introduction to Dynamic Programming

Dynamic programming is similar to the *divide-and-conquer* method. It also has some trivial base cases or cases which can be solved directly. It also divides more complex or larger problems into smaller subproblems, solves these subproblems and combines their solutions to solve the original problem. The main difference is that the *divide-and-conquer* method solves the subproblems independently, like in *merge sort*. Thus, it is effective when the subproblems and sub-subproblems are typically independent. It becomes inefficient when a larger problem has common subproblems and sub-subproblems recursively. In *dynamic programming*, we solve each subproblem only once and remember its solution whenever we meet that subproblem again.

For example, consider the Fibonacci function.
$F : \mathbb{N} \to \mathbb{N}$
$F_n = F_{n-1} + F_{n-2} \quad \text{if } n > 1$
$F_1 = 1 \quad \text{and} \quad F_0 = 0.$

Provided that $n > 1$, we can divide computing F_n into computing F_{n-1} and F_{n-2} and conquer by adding the results of the two recursive calls. As a result, for example,
$F_9 = F_8 + F_7 = (F_7 + F_6) + F_7 = ([F_6 + F_5] + F_6) + (F_6 + F_5) =$
$([\{F_5 + F_4\} + F_5] + \{F_5 + F_4\}) + (\{F_5 + F_4\} + F_5) = \dots$

Thus, F_9 and F_8 are computed only once, F_7 is calculated twice, F_6 is computed three times, F_5 is computed five times, and so on. In general, F_{n-k} is calculated F_{k+1} times, and it follows that computing F_n needs an exponential time of n.

On the contrary, F_n can be computed in linear time if we start with computing with F_2, F_3, F_4 in this order, and so on, always remembering the last two partial results. This is a trivial case of dynamic programming. In the subsequent two algorithms, we use it more complexly, and the partial results will be stored in matrices.

2 Graphs and their Representations

See the representations of the unweighted and weighted graphs in AlgDs1graphs.pdf and in AlgDs2graphs2w.pdf. We use the adjacency matrix representations here.

3 All-Pairs Paths

This chapter considers how to compute a finite binary relation's *transitive closure.* Next, we introduce algorithm *Floyd-Warshall* that searches for shortest paths between each pair of vertices of a weighted graph. Both algorithms

- are based on the adjacency representation of graphs,
- have computation complexity $\Theta(n^3)$,
- are classical examples of *dynamic programming* [3].

Floyd's *Floyd-Warshall algorithm* solves a more general problem than the *Transitive Closure algorithm* of Roy and Warshall.

3.1 Transitive closure of a graph (TC)

In this subsection, for each pair of vertices (u, v) of a network/graph, we want to determine whether there is any path from u to v. We are interested neither in the path nor in its length. For this purpose, we introduce the following notion.

Definition 3.1 *Given graph $G = (V, E)$, its transitive closure is relation $T \subseteq V \times V$ where*
$(u, v) \in T \iff$ *there is some path from vertex u to vertex v in graph G.*

Notation 3.2 $\mathbb{B} = \{0; 1\}$

Provided that the vertices of a graph can be identified by indices $1..n$, we can represent the graph with adjacency matrix $A/1 : \mathbb{B}[n, n]$. And we can represent its transitive closure with matrix $T/1 : \mathbb{B}[n, n]$ where
$T[i, j] \iff$ there is some path from vertex i to vertex j in the graph represented with matrix A.

To compute matrix T, let us define the matrix sequence $\langle T^{(0)}, T^{(1)}, T^{(n)} \rangle$ where $T^{(n)} = T$.

Notation 3.3 $i \overset{k}{\leadsto} j$ *is a path from vertex i to vertex j where the indices of the vertices between i and j are $\leq k$ ($i > k$ and $j > k$ is possible, where $i, j \in 1..n$ and $k \in 0..n$).*

Definition 3.4 $T_{ij}^{(k)} \iff \exists\, i \overset{k}{\leadsto} j \quad (k \in 0..n \;\wedge\; i, j \in 1..n)$

Property 3.5 *(Recursive relation among the T matrices.)*
$T_{ij}^{(0)} = A[i,j] \vee (i = j) \qquad (i,j \in 1..n)$
$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)} \qquad (k \in 1..n \ \wedge \ i,j \in 1..n)$

Consequence 3.6
$T_{ik}^{(k)} = T_{ik}^{(k-1)} \quad \wedge \quad T_{kj}^{(k)} = T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i,j \in 1..n)$

This means that column k and row k of matrix $T^{(k)}$ are the same as the appropriate column and row of matrix $T^{(k-1)}$. $\quad (k \in 1..n)$

Let us notice that a single matrix T is enough for the whole computation, because $T_{ij}^{(k)}$ depends only on $T_{ij}^{(k-1)}$, $T_{ik}^{(k-1)}$ and $T_{kj}^{(k-1)}$, where $T_{ik}^{(k)} = T_{ik}^{(k-1)} \wedge T_{kj}^{(k)} = T_{kj}^{(k-1)}$

```
TransitiveClosure($A/1, T/1 : \mathbb{B}[n,n]$)
  $i := 1$ to $n$
    $j := 1$ to $n$
      $T[i,j] := A[i,j]$
    $T[i,i] := 1$
  $k := 1$ to $n$
    $i := 1$ to $n$
      $j := 1$ to $n$
        $T[i,j] := T[i,j] \vee (T[i,k] \wedge T[k,j])$
```

Warshall's transitive closure algorithm:
$T(n) = (n + n^2) + (n + n^2 + n^3)$, thus $T(n) \in \Theta(n^3)$.

3.1.1 Computing transitive closure with breadth-first search

When BFS has been completed with $s = i$, the black vertices are those available from vertex i, and the white vertices are those unavailable from vertex i.

As a result, we can simplify BFS. We do not count d and π values, just a boolean label $nonwhite(j)$ for each vertex j. Then $T[i,j] \iff nonwhite(j)$ where $j \in 1..n$. And this is row i of matrix T where $i \in 1..n$. Thus $MT(n,m) \in \Theta(n+m)$ for a single row of T.

Thus, the whole matrix is computed with $\Theta(n*(n+m))$ maximal running time. This is $\Theta(n^2)$ on sparse graphs (where $m \in O(n)$), asymptotically better than Warshall's TC algorithm. But this is $\Theta(n^3)$ on dense graphs (where $m \in \Theta(n^2)$), meaning a longer running time than that of Warshall's TC algorithm because of higher constants hidden in the *Theta* notation.

3.2 All-Pairs Shortest Paths: the Floyd-Warshall algorithm (FW)

Notation 3.7 $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$

Note 3.8 *In the FW algorithm, the undirected graphs will be modelled with digraphs where for each edge (u, v) of the graph, (v, u) is also the edge of the graph, and $w(u, v) = w(v, u)$.*

This model simplifies the forthcoming discussions. In the rest of these lecture notes, graph means digraph by default.

Given a weighted graph with adjacency matrix $A/1 : \mathbb{R}_\infty[n, n]$. FW computes matrix $D/1 : \mathbb{R}_\infty[n, n]$ where $D[i, j]$ is the length of an optimal path from vertex i to vertex j; or $D[i, j] = \infty$, if there is no path from i to j. FW also computes matrix $\pi/1 : \mathbb{N}[n, n]$ where $\pi[i, j]$ is the parent node of vertex j on an optimal path from i to j if $i \neq j$ and there is some path from i to j. Otherwise $\pi[i, j] = 0$.

Precondition: There is no negative cycle in the graph. (The algorithm checks this condition.)
Task: FW constructs the following sequence of matrix pairs:
$\langle (D^{(0)}, \pi^{(0)}), (D^{(1)}, \pi^{(1)}), \ldots, (D^{(n)}, \pi^{(n)}) \rangle$ where $D^{(0)} = A$, $\pi^{(0)}$ and matrix pairs $(D^{(k)}, \pi^{(k)})$ $[k \in 1..n]$ can be constructed according to their properties which we are going to present, $D = D^{(n)} \wedge \pi = \pi^{(n)}$.

Notation 3.9 $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$ *($k \in 1..n$) is a shortest path from vertex i to vertex j with two constraints:*

- *On this path, the indices of the vertices between vertex i and vertex j are $\leq k$.*
- *This path contains no cycle.*

Note 3.10

- *If $i = j$ then $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = \langle i \rangle$.*
- *If $i \neq j$ then $\exists\, i \underset{opt}{\overset{0}{\rightsquigarrow}} j = \langle i, j \rangle \iff (i, j)$ is an edge of the graph.*
- *If $i \neq j \wedge k \in 1..n$, there are two possibilities about path $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$:*
  $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} j \quad \vee \quad i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} k \underset{opt}{\overset{k-1}{\rightsquigarrow}} j.$

Definition 3.11 $D_{ij}^{(k)} = \begin{cases} w(i \underset{opt}{\overset{k}{\leadsto}} j) & \textit{if } i \overset{k}{\leadsto} j \textit{ exists} \\ \infty & \textit{if } i \overset{k}{\leadsto} j \textit{ does not exist} \end{cases}$

Definition 3.12
$\pi_{ij}^{(k)} = \begin{cases} \textit{the parent of vertex } j \textit{ on a path } \; i \underset{opt}{\overset{k}{\leadsto}} j\,, \quad \textit{if } i \neq j \wedge i \overset{k}{\leadsto} j \textit{ exists} \\ 0 \qquad \textit{if } i = j \vee i \overset{k}{\leadsto} j \textit{ does not exist} \end{cases}$

Property 3.13 *Matrix $D^{(0)}$ is equal to adjacency matrix A of the graph, and matrix $D^{(n)}$ is equal to matrix D to be computed.*

Property 3.14
$\pi_{ij}^{(0)} = \begin{cases} i & \textit{if } i \neq j \wedge (i,j) \textit{ is an edge of the graph} \\ 0 & \textit{if } i = j \vee (i,j) \textit{ is not edge of the graph} \end{cases}$
And matrix $\pi^{(n)}$ is equal to matrix π to be computed.

Property 3.15 *based on note 3.10, provided that $k \in 1..n$:*
If $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
then $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$
else $D_{ij}^{(k)} = D_{ij}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$

Consequence 3.16 *Provided that $k \in 1..n$:*
$D_{ik}^{(k)} = D_{ik}^{(k-1)} \wedge \pi_{ik}^{(k)} = \pi_{ik}^{(k-1)} \quad \wedge \quad D_{kj}^{(k)} = D_{kj}^{(k-1)} \wedge \pi_{kj}^{(k)} = \pi_{kj}^{(k-1)}$

Note 3.17 *about the correctness of function* FloydWarshall(A, D, π) *on Figure 1. (We suppose here that the precondition is satisfied, i.e. there is no negative loop in the graph.)* We prove that $D = D^{(n)} \wedge \pi = \pi^{(n)}$ at the end of the function, where D and π are the matrices of the function, while $D^{(n)}$ and $\pi^{(n)}$ are the final theoretical matrices defined in 3.11 and 3.12.

Let us consider matrices D, π of the function and theoretical matrices $D^{(k)}$, $\pi^{(k)}$ where $k \in 1..n$.

The first double for-loop above initialises the matrices of the function as $D = D^{(0)} \wedge \pi = \pi^{(0)}$.

*A single iteration of the main loop ($k := 1$ **to** n) computes the matrix pair $(D^{(k)}, \pi^{(k)})$ from $(D^{(k-1)}, \pi^{(k-1)})$. And this computation is done in the matrix pair (D, π) of the function in the following way.*

Let us suppose that we are at the beginning of iteration k of the main loop where $D = D^{(k-1)} \wedge \pi = \pi^{(k-1)}$. (We have seen that it is valid for $k = 1$.)

```
FloydWarshall($A/1, D/1 : \mathbb{R}_\infty[n,n]$ ; $\pi/1 : \mathbb{N}[n,n]$) : $\mathbb{N}$
  $i := 1$ to $n$
    $j := 1$ to $n$
      $D[i,j] := A[i,j]$
      if $i \neq j \wedge A[i,j] < \infty$
        $\pi[i,j] := i$
      else
        $\pi[i,j] := 0$
  $k := 1$ to $n$
    $i := 1$ to $n$
      $j := 1$ to $n$
        if $D[i,j] > D[i,k] + D[k,j]$
          $D[i,j] := D[i,k] + D[k,j]$
          $\pi[i,j] := \pi[k,j]$
          if $i = j \wedge D[i,i] < 0$
            return $i$ // $i$ is in negative cycle
          else
            SKIP
        else
          SKIP
  return 0 // shortest-path trees computed
```

Figure 1: Algorithm FW – $MT(n) \in \Theta(n^3) \ \wedge \ mT(n) \in \Theta(n^2)$

Now let us suppose that we arrive at condition $D[i,j] > D[i,k] + D[k,j]$ *where* $i, j \in 1..n$, *and for the earlier iterations* $i = i', j = j'$ *of the inner, double loop* $D[i',j'] = D^{(k)}_{i'j'} \wedge \pi[i',j'] = \pi^{(k)}_{i'j'}$ *has been ensured.*

Clearly $D[i,j] = D^{(k-1)}_{ij} \wedge \pi[i,j] = \pi^{(k-1)}_{ij}$, *because* $D[i,j]$ and $\pi[i,j]$ *has not been updated yet. And* $D[i,k] = D^{(k-1)}_{ik} = D^{(k)}_{ik} \wedge D[k,j] = D^{(k-1)}_{kj} = D^{(k)}_{kj} \wedge \pi[k,j] = \pi^{(k-1)}_{kj} = \pi^{(k)}_{kj}$ *according to Consequence 3.16. Performing this conditional statement* $D[i,j] = D^{(k)}_{ij} \wedge \pi[i,j] = \pi^{(k)}_{ij}$ *becomes true according to Property 3.15. (Clearly,* $D[i,i] < 0$ *becomes never true if there is no negative cycle in the graph.)*

Thus performing all the iterations of the inner double loop $D[i,j] = D^{(k)}_{ij} \wedge \pi[i,j] = \pi^{(k)}_{ij}$ *becomes true for all* $i, j \in 1..n$. *This means that* $D = D^{(k)} \wedge \pi = \pi^{(k)}$ *at the end of the actual iteration of the main loop. Therefore* $D = D^{(k-1)} \wedge \pi = \pi^{(k-1)}$ *will be valid at the beginning of the next iteration for* k.

Consequently (by mathematical induction), $D = D^{(n)} \wedge \pi = \pi^{(n)}$ *will be valid after the last iteration of the main loop, where* $k = n$.

Note 3.18 *We have found a negative cycle if a negative value appears in the main diagonal of matrix D of the following program. (Let the reader argue about this statement.)*

Note 3.19 *As a result of the algorithm Transitive Closure,*
$T[i, j] \iff D[i, j] < \infty$ *when algorithm FW has been finished, and FW did not find negative cycle* $(i, j \in 1..n)$.

Practically speaking, algorithm Transitive Closure solves its task more efficiently than algorithm FW because of the simpler bodies of the loops.

3.2.1 Solving the All-Pairs Shortest Paths problem with the Single-Source Shortest Paths algorithms

Let us notice that the solution of the Floyd-Warshall (FW) algorithm, namely the ith rows of matrices D and π computed with it, supply a solution of the Single-Source Shortest Paths problem for $s = i$, provided that there is no negative loop in the input graph.

Similarly, by solving the Single-Source Shortest Paths problem for each $s \in 1..n$, we also receive a solution for the All-Pairs Shortest Paths problem.

Below, we consider some cases.

Provided that our graph is a DAG, we can call the DAG Shortest Paths algorithm for each $s \in 1..n$, and a solution of the All-Pairs Shortest Paths problem can be computed with worst-case time complexity $MT(n, m) \in \Theta(n * (n + m))$.

For sparse graphs, where $m \in O(n)$, $\Theta(n * (n + m)) = \Theta(n^2)$, thus $MT(n, m) \in \Theta(n^2)$. This means that on sparse graphs, performing n times the DAG Shortest Paths algorithm is faster by order of magnitude than performing the FW algorithm, which runs in $MT(n), mT(n) \in \Theta(n^3)$ on DAGs.

For dense graphs, where $m \in \Theta(n^2)$, $\Theta(n * (n + m)) = \Theta(n^3)$, thus $MT(n, m) \in \Theta(n^3)$ which may cause slower run than that of FW, because of the higher constants hidden in the Θ-notation.

We receive similar results when applying Breadth-First Search versus FW to unweighted graphs.

Provided that our graph has no negative edge, we can call the Dijkstra algorithm for each $s \in 1..n$, and a solution of the All-Pairs Shortest Paths problem can be computed with worst-case time complexity $MT(n, m) \in O(n * (n + m) * \log n)$.

For sparse graphs, where $m \in O(n)$, $O(n * (n + m) * \log n) = O(n^2 * \log n)$, thus $MT(n, m) \in O(n^2 * \log n)$. This means that on sparse graphs, performing n times the Dijkstra algorithm is also asymptotically faster than performing the FW algorithm, which runs in $MT(n), mT(n) \in \Theta(n^3)$ on graphs with non-negative edges.

For dense graphs, where $m \in \Theta(n^2)$, $O(n*(n+m)*\log n) = O(n^3*\log n)$, thus $MT(n,m) \in O(n^3*\log n)$ which may cause slower run than that of FW, because $n^3 * \log n$ is asymptotically greater than n^3.

If we know only that the input graph contains no negative cycle, performing QBF for each $s \in 1..n$, $MT(n,m) \in O(n * n * m) = O(n^2 * m)$. Provided that our graph is **not** extremely sparse, i.e. we can suppose that $m \in \Omega(n)$, the upper estimate of the asymptotic running time of $n*$QBF is never better than the asymptotic running time of the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on graphs with no negative cycle.

Notice that we have compared the upper estimate of the asymptotic running time of $n*$QBF to the asymptotic running time of the FW algorithm. Thus, the previous result does not mean that $n*$QBF will be actually slower than FW, provided that they run on the same large input graph.