Reachability as Transitive Closure

Algorithm: Design & Analysis

[15]
In the last class…

- Optimization Problem
- MST Problem
  - Prim’s Algorithm
  - Kruskal’s Algorithm
- Single-Source Shortest Path Problem
  - Dijkstra’s Algorithm
- Greedy Strategy
Reachability as Transitive Closure

- Shortest Path and Transitive Closure
- Washall’s Algorithm for Transitive Closure
- All-Pair Shortest Paths
- Matrix for Transitive Closure
- Multiplying Bit Matrices - Kronrod’s Algorithm
Fundamental Questions

- For all pair of vertices in a graph, say, $u$, $v$:
  - Is there a path from $u$ to $v$?
  - What is the shortest path from $u$ to $v$?

- Reachability as a (reflexive) transitive closure of the adjacency relation, which can be represented as a bit matrix.
Transitive Closure by Shortcuts

- The idea: if there are edges $s_i s_k, s_k s_j$, then an edge $s_i s_j$, the “shortcut” is inserted.

Note: the triple (1,5,3) is considered more than once.
Trans. Closure by Shortcuts: algorithm

- Input: $A$, an $n \times n$ boolean matrix that represents a binary relation
- Output: $R$, the boolean matrix for the transitive closure of $A$
- Procedure
  - void simpleTransitiveClosure(boolean[ ][ ] A, int n, boolean[ ][ ] R)
  - int i,j,k;
  - Copy $A$ to $R$;
  - Set all main diagonal entries, $r_{ii}$, to true;
  - while (any entry of $R$ changed during one complete pass)
    - for (i=1; i≤ n; i++)
      - for (j=1; j≤ n; j++)
          - for (k=1; k≤ n; k++)
            - $r'_{ij}=r_{ij} \lor (r_{ik} \land r_{kj})$
  
  The order of (i,j,k) matters
Shortcuts in different order

- Duplicated checking may be deleted by changing the order of the vertices.

Pass one
Check the vertices in decreasing order.

No edge is added in Pass two. End.
Change the order: Washall’s Algorithm

- `void simpleTransitiveClosure(boolean[][] A, int n, boolean[][] R)`
- `int i,j,k;`  
- Copy $A$ to $R$;
- Set all main diagonal entries, $r_{ii}$, to `true`;
- `while (any entry of R changed during one complete pass)`
  ```
  for (k=1; k≤n; k++)
      for (i=1; i≤n; i++)
          for (j=1; j≤n; j++)
              r_{ij}=r_{ij} ∨ (r_{ik} ∧ r_{kj})
  ```
  Note: “false to true” can not be reversed
The highest intermediate vertex in the intervals $(s_i s_k), (s_k s_j)$ are both less than $s_k$.

A specific order is assumed for all vertices.

Vertical position of vertices reflect their vertex numbers.
Correctness of Washall’s Algorithm

- **Notation:**
  - The value of $r_{ij}$ changes during the execution of the body of the “for $k$...” loop
    - After initializations: $r_{ij}^{(0)}$
    - After the $k$th time of execution: $r_{ij}^{(k)}$
Correctness of Washall’s Algorithm

If there is a simple path from \( s_i \) to \( s_j (i \neq j) \) for which the highest-numbered intermediate vertex is \( s_k \), then \( r_{ij}^{(k)} = \text{true} \).

Proof by induction:

- Base case: \( r_{ij}^{(0)} = \text{true} \) if and only if \( s_is_j \in E \)
- Hypothesis: the conclusion holds for \( h < k (k \geq 0) \)
- Induction: the simple \( s_is_j \)-path can be looked as \( s_is_k \)-path+\( s_k s_j \)-path, with the indices \( h_1, h_2 \) of the highest-numbered intermediate vertices of both segment strictly (simple path) less than \( k \). So, \( r_{ik}^{(h_1)} = \text{true} \), \( r_{kj}^{(h_2)} = \text{true} \), then \( r_{ik}^{(k-1)} = \text{true} \), \( r_{kj}^{(k-1)} = \text{true} \) (Remember, false to true can not be reversed). So, \( r_{ij}^{(k)} = \text{true} \)
Correctness of Washall’s Algorithm

- If there is no path from $s_i$ to $s_j$, then $r_{ij} = false$.

Proof

- If $r_{ij} = true$, then only two cases:
  - $r_{ij}$ is set by initialization, then $s_i s_j \in E$
  - Otherwise, $r_{ij}$ is set during the $k$th execution of (for $k=1,2,...$) when $r_{ik}^{(k-1)} = true$, $r_{kj}^{(k-1)} = true$, which, recursively, leading to the conclusion of the existence of a $s_i s_j$-path. (Note: If a $s_i s_j$-path exists, there exists a simple $s_i s_j$-path)
All-pairs Shortest Path

- Non-negative weighted graph
- **Shortest path property**: If a shortest path from x to z consisting of path P from x to y followed by path Q from y to z. Then P is a shortest xz-path, and Q, a shortest zy-path.
- The regular matrix representing a graph can easily be transformed into a (minimum) distance matrix $D$ (just replacing 1 by edge weight, 0 by infinity, and setting main diagonal elements as 0)
Computing the Distance Matrix

- Basic formula:
  - \( D^{(0)}[i][j] = w_{ij} \)
  - \( D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]) \)

- Basic property:
  - \( D^{(k)}[i][j] \leq d_{ij}^{(k)} \)
    where \( d_{ij}^{(k)} \) is the weight of a shortest path from \( v_i \) to \( v_j \) with highest numbered intermediate vertex \( v_k \).
All-Pairs Shortest Paths

- Floyd algorithm
  - Only slight changes on Washall’s algorithm.

```c
Void allPairsShortestPaths(float [][] W, int n, float [][] D)
int i, j, k;
Copy W into D;
for (k=1; k<=n; k++)
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      D[i][j] = min (D[i][j], D[i][k]+D[k][j];
```

- Routing table tracking the path
Matrix Representation

- Define family of matrix $A^{(p)}$:
  - $a_{ij}^{(p)}=true$ if and only if there is a path of length $p$ from $s_i$ to $s_j$.
- $A^{(0)}$ is specified as identity matrix. $A^{(1)}$ is exactly the adjacency matrix.
- Note that $a_{ij}^{(2)}=true$ if and only if exists some $s_k$, such that both $a_{ik}^{(1)}$ and $a_{kj}^{(1)}$ are true. So, $a_{ij}^{(2)}=\bigvee_{k=1,2,\ldots,n}(a_{ik}^{(1)} \land a_{kj}^{(1)})$, which is an entry in the Boolean matrix product.
Boolean Matrix Operations: Recalled

- Boolean matrix product $C = AB$ as:
  $\forall i, j (c_{ij}) = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj})$

- Boolean matrix sum $D = A + B$ as:
  $\forall i, j (d_{ij}) = a_{ik} \lor b_{kj}$

- $R$, the transitive closure matrix of $A$, is the sum of all $A^p$, $p$ is a non-negative integer.

- For a digraph with $n$ vertices, the length of the longest simple path is no larger than $n-1$. 
A bit string of length $n$ is a sequence of $n$ bits occupying contiguous storage (word boundary) (usually, $n$ is larger than the word length of a computer).

If $A$ is a bit matrix of $n \times n$, then $A[i]$ denotes the $i$th row of $A$ which is a bit string of length $n$. $a_{ij}$ is the $j$th bit of $A[i]$.

The procedure $\text{bitwiseOR}(a,b,n)$ compute $a \lor b$ bitwise for $n$ bits, leaving the result in $a$. 
Straightforward Multiplication of Bit Matrix

- Computing $C = AB$
  - <Initialize $C$ to the zero matrix>
  - for (i=1; i≤n, i++)
  - for (k=1; k≤n, k++)
  - if ($a_{ik} == \text{true}$) bitwiseOR($C[i], B[k], n$)

In the case of $a_{ik}$ is $\text{true}$, $c_{ij} = a_{ik}b_{kj}$ is true iff. $b_{kj}$ is true. As a result:

$C[i] = \bigcup_{k \in A[i]} B[k], (A[i] = \{k | ik == \text{true}\})$

Union for $B[k]$ is repeated each time when the $k$th bit is true in a different row of $A$ is encountered.

Thought as a union of sets (row union), $n^2$ unions are done at most.
Reducing the Duplicates by Grouping

- Multiplication of $A$, $B$, two $12 \times 12$ matrices

\[
\begin{pmatrix}
A_1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
A_2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
A_3 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
A_4 & 0 & 1 & 0 & 1 & & & & & & & & \\
A_5 & & & & & & & & & & & & \\
A_6 & & & & & & & & & & & & \\
A_7 & & & & & & & & & & & & \\
A_8 & & & & & & & & & & & & \\
A_9 & & & & & & & & & & & & \\
A_{10} & & & & & & & & & & & & \\
A_{11} & & & & & & & & & & & & \\
A_{12} & & & & & & & & & & & & \\
\end{pmatrix}
\]

- 12 rows of $B$ are divided evenly into 3 groups, with rows 1-4 in group 1, etc.
- With each group, all possible unions of different rows are pre-computed. (This can be done with 11 unions if suitable order is assumed.)
- When the first row of $AB$ is computed, $(B[1] \cup B[3] \cup B[4])$ is used in stead of 3 different unions, and this combination is used in computing the 3rd and 7th rows as well.
The Segmentation for Matrix A

The $n \times n$ array

$$\begin{array}{cccccccc}
1 & 2 & \ldots & t & t+1 & \ldots & 2t & \ldots & (j-1)t+1 & \ldots & jt & \ldots \\
\end{array}$$

$A_i$

- bitSeg($A[i], 1, t$)
- bitSeg($A[i], 2, t$)
- bitSeg($A[i], j, t$)

Bit string of length $t$, looked as a $t$-bit integer
### An Example

A table and a diagram illustrating the process of bit segmentation for different groups.

#### Group 1 (1…t)

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
<th>A11</th>
<th>A12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Group 2 (t+1…2t)

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
<th>B8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Group 3 (2t+1…3t)

<table>
<thead>
<tr>
<th>B9</th>
<th>B10</th>
<th>B11</th>
<th>B12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**bitSeg(A[7], 1, t)**

\[= 1011_2 = 11\]

---

Where to store?
Storage of the Row Combinations

- Using one large 2-dimensional array
- Goals
  - keep all unions generated
  - provide indexing for using
- Coding within a group
  - One-to-one correspondence between a bit string of length \( t \) and one union for a subset of a set of \( t \) elements
- Establishing indexing for union required
  - When constructing a row of \( AB \), a segment can be notated as a integer. Use it as index.
Storage the Unions

allUnion

one row for one group
column indexed by \text{bitSeg}(A[i,j,t])

\[
\begin{bmatrix}
\phi & 4 & 3 & 3,4 & 2 & 2,4 & 2,3 & 2,3,4 & 1 & 1,4 & 1,3 & 1,3,4 & 1,2 & 1,2,4 & 1,2,3 & 1,2,3,4 \\
\phi & 8 & 7 & 7,8 & 6 & 6,8 & & & & & & & & & \\
\phi & 12 & 11 & 11,12 & 10 & 10,12 & & & & & & & & & \\
\end{bmatrix}
\]

\(i,j,k\) stands for \(B_i \cup B_j \cup B_k\)
Array for Row Combinations

The array: \textit{allUnions}

Containing all possible row combinations, totaling $2^t$, within $j$th group of $B$

Indexed by segment coding for Matrix A

Containing in each row the union, of which the code is exactly $i$, looked as a $t$-bit binary number.
Cost as Function of Group Size

Cost for the pre-computation

- There are $2^t$ different combination of rows in one group, including an empty and $t$ singleton. Note, in a suitable order, each combination can be made using only one union. So, the total number of union is $g[2^t-(t+1)]$, where $g=n/t$ is the number of group.

Cost for the generation of the product

- In computing one of $n$ rows of $AB$, at most one combination from each group is used. So, the total number of union is $ng$
Selecting Best Group Size

- The total number of union done is:
  \[ g[2^t-(t+1)]+n(g-1) \approx (n2^t)/t+n^2/t \quad (\text{Note: } g=n/t) \]

- Trying to minimize the number of union
  - Assuming that the first term is of higher order:
    - Then \( t \geq \log n \), and the least value is reached when \( t=\log n \).
  - Assuming that the second term is of higher order:
    - Then \( t \leq \log n \), and the least value is reached when \( t=\log n \).

- So, when \( t \approx \log n \), the number of union is roughly \( 2n^2/\log n \), which is of lower order than \( n^2 \). We use \( t=\lfloor \log n \rfloor \)

For simplicity, exact power for \( n \) is assumed.
Sketch for the Procedure

- $t = \lfloor \log n \rfloor$; $g = \lceil n/t \rceil$;
- <Compute and store in allUnions unions of all combinations of rows of $B$>
- for $i=1$; $i \leq n$; $i++$  
  - <Initialize $C[i]$ to 0>
- for $j=1$; $j \leq g$; $j++$
  - $C[i] = C[i] \cup \text{allUnions}[j][\text{bitSeg}(A[i], j, t)]$
Kronrod Algorithm

- **Input**: A, B and n, where A and B are \(n \times n\) bit matrices.
- **Output**: C, the Boolean matrix product.

**Procedure**

- The processing order has been changed, from “row by row” to “group by group”, resulting in the reduction of storage space for unions.
Complexity of Kronrod Algorithm

- For computing all unions within a group, $2^t-1$ union operations are done.
- One union is bitwiseOR’ed to $n$ row of $C$
- So, altogether, $(n/t)(2^t-1+n)$ row unions are done.
- The cost of row union is $\lceil n/w \rceil$ bitwise or operations, where $w$ is word size of bitwise or instruction dependent constant.
Home Assignments

- pp.446-
  - 9.10
  - 9.12
  - 9.16
  - 9.17