Priority Queues

What is a priority queue?
A priority queue is an abstract data type which consists of a set of elements. Each element of the set has an associated priority or key. Priority is the value of the element or value of some component of an element.

Example:
S : {{Brown, 20}, (Gray, 22), (Green, 21)} priority based on name
{(Brown, 20), (Green, 21), (Gray, 22)} priority based on age

Each element could be a record and the priority could be based on one of the fields of the record.
Example

A Student's record:

Attributes : Name    Age    Sex    Student No.    Marks
Values :       John Brown  21    M     94XYZ23      75

Priority can be based on name, age, student number, or marks

Operations performed on priority queues,
  -inserting an element into the set
  -finding and deleting from the set an element of highest priority

Priority Queues

Priority queues are implemented on partially ordered trees (POTs)
  • POTs are labeled binary trees
  • the labels of the nodes are elements with a priority
  • the element stored at a node has at least as large a priority as the elements stored at the children of that node
  • the element with the highest priority is at the root of the tree
HEAPS

The heap is a data structure for implementing POT's
Each node of the heap tree corresponds to an
element of the array that stores the value in the
node
The tree is filled on all levels except possibly the
lowest, which are filled from left to right up to a
point.
An array A that represents a heap is an object with two
attributes

- length[A], the number of elements in the array and
- heap-size[A], the number of elements in the heap stored
  within the array A

heap_size[A] ≤ length[A]
HEAPS (Contd)


<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>24</td>
<td>21</td>
<td>19</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

Given node with index i,

PARENT(i) is the index of parent of i; PARENT(i) = \lceil i/2 \rceil

LEFT_CHILD(i) is the index of left child of i;
LEFT_CHILD(i) = 2 \times i;

RIGHT_CHILD(i) is the index of right child of i; and
RIGHT_CHILD(i) = 2 \times i + 1

Heap Property

THE HEAP PROPERTY
A[PARENT(i)] \geq A[i]

The heap is based on a binary tree
The height of the heap (as a binary tree) is the number of edges on the longest simple downward path from the root to a leaf.

The height of a heap with n nodes is O (log n).

All basic operations on heaps run in O (log n) time.
\[ n = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^h = 2^{h+1} - 1 \]

**Heap Algorithms**

- **HEAPIFY**
- **BUILD_HEAP**
- **HEAPSORT**
- **HEAP_EXTRACT_MAX**
- **HEAP_INSERT**
HEAPIFY

The HEAPIFY algorithm checks the heap elements for violation of the heap property and restores heap property.

Procedure HEAPIFY (A, i)

Input: An array A and index i to the array. i = 1 if we want to heapify the whole tree. Subtrees rooted at LEFT_CHILD(i) and RIGHT_CHILD(i) are heaps

Output: The elements of array A forming subtree rooted at i satisfy the heap property.

1. \( l \leftarrow \text{LEFT}_\text{CHILD}(i) \);
2. \( r \leftarrow \text{RIGHT}_\text{CHILD}(i) \);
3. if \( l \leq \text{heap size}[A] \) and \( A[l] > A[i] \)
   then largest \( \leftarrow l \);
4. else largest \( \leftarrow i \);
5. if \( r \leq \text{heap size}[A] \) and \( A[r] > A[\text{largest}] \)
   then largest \( \leftarrow r \);
6. if \( \text{largest} \neq i \)
   then exchange \( A[i] \leftrightarrow A[\text{largest}] \);
7. \( \text{HEAPIFY}(A, \text{largest}) \)
Running time of HEAPIFY

Total running time = steps 1 … 9 + recursive call

\[ T(n) = \Theta(1) + T(n/2) \]

Solving the recurrence, we get \( T(n) = O(\log n) \)

BUILD_HEAP

Procedure BUILD_HEAP (A)

Input : An array A of size \( n = \text{length}[A] \), heap_size[A]

Output : A heap of size n

1. \( \text{heap_size}[A] \leftarrow \text{length}[A] \)
2. for \( i \leftarrow \lceil \text{length}[A]/2 \rceil \) downto 1
3. \( \text{HEAPIFY}(A,i) \)
Running time of Build_heap

1. Each call to HEAPIFY takes $O(\log n)$ time
2. There are $O(n)$ such calls
3. Therefore the running time is at most $O(n \log n)$

However the complexity of BUILD_HEAP is $O(n)$

Proof:

In an $n$ element heap there are at most $\lceil n/2^h+1 \rceil$ nodes of height $h$

The time required to heapify a subtree whose root is at a height $h$ is $O(h)$

(this was proved in the analysis for HEAPIFY)

So the total time taken for BUILD_HEAP is given by,

$$
\leq \left\lceil \log \frac{n}{2} \right\rceil \cdot h
$$

$$
\leq \frac{n}{2} \cdot \sum_{h=0}^{\log n} \frac{h}{2^h}
$$

We know that

$$
\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 = O(n)
$$

Thus the running time of BUILD_HEAP is given by, $O(n)$
The HEAPSORT Algorithm

Procedure HEAPSORT(A)
Input : Array A[1…n], n = length[A]
Output : Sorted array A[1…n]
1. BUILD_HEAP[A]
2. for i ← length[A] down to 2
4. heap_size[A] ← heap_size[A]-1;
5. HEAPIFY(A,1)

Example : To be given in the lecture

HEAPSORT

Running Time:
Step 1 BUILD_HEAP takes O(n) time,
Steps 2 to 5 : there are (n-1) calls to HEAPIFY
which takes O(log n) time
Therefore running time takes O(n log n)
HEAP_EXTRACT_MAX

Procedure HEAP_EXTRACT_MAX(A[1...n])
Input : heap(A)
Output : The maximum element or root, heap (A[1...n-1])
1. if heap_size[A] ≥ 1
2. max ← A[1];
4. heap_size[A] ← heap_size[A]-1;
5. HEAPIFY(A,1)
6. return max

Running Time : O (log n) time

HEAP_INSERT

Procedure HEAP_INSERT(A, key)
Input : heap(A[1...n]), key - the new element
Output : heap(A[1...n+1]) with k in the heap
1. heap_size[A] ← heap_size[A]+1;
2. i ← heap_size[A];
3. while i > 1 and A[PARENT(i)] < key
5. i ← PARENT(i);
6. A[i] ← key

Running Time : O (log n) time
Questions:

- What is a heap?
- What are the running times for heap insertion and deletion operations?
- Did you understand HEAPIFY AND and HEAPSORT algorithms?
- Can you write a heapsort algorithm for arranging an array of numbers in descending order?

Quicksort and Mergesort

This week
- Quicksort algorithm
- Quicksort performance
- Quicksort analysis
- Mergesort algorithm
- Mergesort analysis

Further Reading
Chapters 1 and 8 from Textbook
Quicksort

- The worst case running time of Quicksort algorithm is $O(n^2)$
- However, its expected running time is $O(n \log n)$
- Three-step divide-and-conquer process for sorting a subarray $A[l..r]$

**Divide**: partition the array $A[l..r]$ into two nonempty subarrays $A[l..q]$ and $A[q+1, r]$ such that each element of $A[l..q]$ is less than or equal to each element of $A[q+1,..,r]$

**Conquer**: sort the two subarrays $A[l..q]$ and $A[q+1..r]$ by recursive calls to Quicksort

**Combine**: the subarrays are already sorted in place. No work is needed to combine them

---

Example

```
13 02 18 26 76 87 98 11 93 77 65 43 38 09 65 06
13 02 06 26 76 87 98 11 93 77 65 43 38 09 65 18
13 02 06 09 76 87 98 11 93 77 65 43 38 26 65 18
13 02 06 09 11 87 98 76 93 77 65 43 38 26 65 18
11 02 06 09 13 87 98 76 93 77 65 43 38 26 65 18
11 02 06 09 13 87 98 76 93 77 65 43 38 26 65 18
09 02 06 11 13 87 98 76 93 77 65 43 38 26 65 18
09 02 06 11 13 87 98 76 93 77 65 43 38 26 65 18
06 02 09 11 13 87 18 76 65 77 65 43 38 26 93 98
06 02 09 11 13 87 18 76 65 77 65 43 38 26 93 98
02 06 09 11 13 26 18 76 65 77 65 43 38 87 93 98
02 06 09 11 13 26 18 76 65 77 65 43 38 87 93 98
02 06 09 11 13 18 26 76 65 38 65 43 77 87 93 98
02 06 09 11 13 18 26 43 65 38 65 76 77 87 93 98
02 06 09 11 13 18 26 43 38 65 65 76 77 87 93 98
02 06 09 11 13 18 26 38 43 65 65 76 77 87 93 98
02 06 09 11 13 18 26 38 43 65 65 76 77 87 93 98
```

8/18/2005 M KUMAR CSE5311 27
Procedure Quicksort\((A, l, r)\)

**Input**: Unsorted Array \((A, l, r)\)

**Output**: Sorted subarray \(A[0..r]\)

To sort the entire array \(A\), \(l = 1\) and \(r = \text{length}[A]\)

\[
\text{if } l < r \\
\quad \text{then } q \leftarrow \text{PARTITION}(A, l, r) \\
\quad \text{QUICKSORT}(A, l, q-1) \\
\quad \text{QUICKSORT}(A, q+1, r)
\]

Procedure PARTITION\((A, l, r)\)

**Input**: Array \(A[l..r]\)

**Output**: \(A\) and \(q\) such that \(A[i] \leq A[q]\) for all \(i \leq q\) and \(A[j] > A[q]\) for all \(j > q\).

\[
x \leftarrow A[l]; i \leftarrow l; j \leftarrow r; \\
\text{while } i < j \text{ do} \\
\quad \text{while } A[i] \leq x \text{ and } i \leq r \text{ do } i \leftarrow i + 1; \\
\quad \text{while } A[j] > x \text{ and } j \geq l \text{ do } j \leftarrow j - 1; \\
\quad \text{if } i < j \text{ then} \\
\quad \quad \text{exchange } A[i] \leftrightarrow A[j]; \\
q \leftarrow j; \\
\quad \text{exchange } A[l] \leftrightarrow A[q];
\]
Running Time of Quicksort

\[ T(n) = n-1 + T(i-1) + T(n-i) \]

It takes \( n-1 \) comparisons for the partition
Then we sort smaller sequences of size \( i-1 \) and \( n-i \)
Each element has the same probability of being selected as the pivot,
The average running time is given by,

\[
T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} T(i-1) + \frac{1}{n} \sum_{i=1}^{n} T(n-i) \\
T(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n} T(i) = O(n \log n)
\]

Detailed Analysis in the Class

Mergesort

Like Quicksort, Mergesort algorithm also is based on the divide-and-conquer principle.

Divide: This step computes the middle of the array, takes constant time, \( \Theta(1) \)

Conquer : Two subproblems, each of size \( n/2 \) are recursively solved.
Each subproblem contributes \( 2T(n/2) \) to the running time.

Combine: Two sorted sequences are merged, this takes \( \Theta(n) \) time
Example

<table>
<thead>
<tr>
<th>13</th>
<th>02</th>
<th>18</th>
<th>26</th>
<th>76</th>
<th>87</th>
<th>98</th>
<th>11</th>
<th>93</th>
<th>77</th>
<th>65</th>
<th>43</th>
<th>38</th>
<th>09</th>
<th>65</th>
<th>06</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>13</td>
<td>18</td>
<td>26</td>
<td>76</td>
<td>87</td>
<td>98</td>
<td>11</td>
<td>93</td>
<td>77</td>
<td>65</td>
<td>43</td>
<td>38</td>
<td>09</td>
<td>65</td>
<td>06</td>
</tr>
<tr>
<td>02</td>
<td>13</td>
<td>18</td>
<td>26</td>
<td>76</td>
<td>87</td>
<td>98</td>
<td>11</td>
<td>93</td>
<td>77</td>
<td>65</td>
<td>43</td>
<td>38</td>
<td>09</td>
<td>65</td>
<td>06</td>
</tr>
<tr>
<td>02</td>
<td>13</td>
<td>18</td>
<td>26</td>
<td>76</td>
<td>87</td>
<td>98</td>
<td>11</td>
<td>93</td>
<td>77</td>
<td>65</td>
<td>43</td>
<td>38</td>
<td>09</td>
<td>65</td>
<td>06</td>
</tr>
</tbody>
</table>

Procedure MERGESORT(A, l, r)
Input: A an array in the range 1 to n.
Output: Sorted array A.

if l < r
then q ← \lfloor (l+r)/2 \rfloor;
MERGESORT(A, l, q)
MERGESORT(A, q+1, r)
MERGE (A, l, q, r)
Procedure MERGE(A, l, q, r)
Inputs: two sorted subarrays A(l, q) and A(q+1, r)
Output: Merged and sorted array A(l, r)

i ← l;
j ← q+1;
k ← 0;
while (i ≤ q) and (j ≤ r) do
    k ← k+1;
    if A[i] ≤ A[j] then
        TEMP[k] ← A[i];
        i ← i +1;
    else
        TEMP[k] ← A[j];
        j ← j +1;
if j > r then
    for t ← 0 to q – i do
        A[r-t] ← A[q-t];
for t ← 0 to k-1 do
    A[l+t] ← TEMP[t];

Runtime Complexity of Mergesort

```
\[
\begin{array}{c}
n\\ \downarrow\\ \frac{n}{2}\\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
Runtime Complexity of Mergesort

\[ T(n) = 2 \cdot T(n/2) + \Theta(n) \]
\[ T(n/2) = 2 \cdot T(n/4) + n/2 \]
\[ T(n/4) = 2 \cdot T(n/8) + n/4 \]

\[ T(n) = 2(2 \cdot T(n/4) + n/2) + n = 4 \cdot T(n/4) + 2 \cdot n \]
\[ T(n) = 2^3 \cdot T(n/2^3) + 3n \]

\[ \ldots \]

\[ T(n) = 2^k \cdot T(n/2^k) + k \cdot n \text{ if } n = 2^k \text{ then } , k = \log n \]

Therefore, \[ T(n) = 2^k \cdot T(1) + n \cdot \log n \]
\[ = n \cdot \Theta(1) + n \cdot \log n \]
\[ T(n) = O(n \cdot \log n) \]

Questions

- What is a pivot element?
- Do you understand divide-and-conquer?
- What is running time if pivot element is at the center of the array?
- How is Mergesort different from Quicksort?
- Trace the algorithm with the help of an example?
- What is the use of TEMP array?
- Do you know why we execute steps 13 and 14 in the MERGE algorithm?
- What happens if i > q after the while loop (4-11)?
• Find applications for selection sort, heapsort, quicksort, and mergesort sorting algorithms?
• Which of these problems are suitable for different types of sorting operations?
• Identify applications for which each sorting algorithm works best

• Insertion Sort
• Counting Sort
• Find the Maximum
• Find the Minimum
The Odd-Even Mergesort problem

Procedure PARALLEL ODD-EVEN MERGE SORT

1. for i ← 1 to ⌊N/2⌋; N = number of data elements
2. for all PEj such that j is odd and 1 ≤ j ≤ N - 1
3. if [PEj].a > [PEj+1].a
4. swap([PEj].a,[PEj+1].a)
5. for all PEj such that j is even and 1 ≤ j ≤ N - 1
6. if [PEj].a > [PEj+1].a
7. swap([PEj].a,[PEj+1].a)

1 2 3

9 18 27 36 45 54 63 72
64 56 48 40 32 24 16 8

Bitonic Mergesort

A bitonic sequence is a sequence of numbers,

a0,a1, . . . , an-1 such that for 0 ≤ i ≤ n-1
a0,a1, . . . , ai is in ascending order and
ai+1,ai+2, . . . , an-1 is in descending order

let, n = 2^k, and i = n/2 -1.

a0≤a1≤a2, . . . , an/2-2≤an/2-1 . . . and

an/2≥an/2+1≥an/2+2≥an/2+3 . . . an-2 ≥ an-1

Consider, min (a0,a_n/2), min (a1,a_n/2+1), . . . , min(a_n/2-1,a_n-1)
max(a0,a_n/2), max(a1,a_n/2+1), . . . , max(a_n/2-1,a_n-1)

Both are bitonic sequences!!
Sorting Networks

\[
\begin{align*}
\text{min}(a,b) & \quad \text{max}(a,b) \\
\text{max}(a,b) & \quad \text{min}(a,b)
\end{align*}
\]

\text{compare\_exchange\_min}(j); \quad \text{compare\_exchange\_max}(j);

Compare \(a_0\) with \(a_{n/2}\)
\[
\begin{align*}
a_1 & \text{ with } a_{n/2+1} \\
\vdots \\
a_{n/2-1} & \text{ with } a_{n-1}
\end{align*}
\]

Compare data elements \(i\) and \(2^{k-1}+i\) and put the min in position \(2i\) and the max in position \(2i+1\)

Shuffle-Exchange Network

\[
\begin{align*}
45 & \quad + \quad + \quad + \quad 31 \\
67 & \quad + \quad + \quad + \\
78 & \quad + \quad + \quad + \\
89 & \quad + \quad + \quad + \\
87 & \quad + \quad + \quad + \\
75 & \quad + \quad + \quad + \\
63 & \quad + \quad + \quad + \\
31 & \quad + \quad + \quad +
\end{align*}
\]

\(n/2\) switches per stage
\log n stages
Parallel Bitonic Sort Algorithm

1. Compare \([PE(i)].a\) with \([PE(i+2^{k-1})].a\)
   
   \[PE[2i] \leftarrow \text{MIN} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]
   
   \[PE[2i+1] \leftarrow \text{MAX} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]

2. Compare \([PE(i)].a\) with \([PE(i+2^{k-1})].a\)
   
   \[PE[2i] \leftarrow \text{MIN} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]
   
   \[PE[2i+1] \leftarrow \text{MAX} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]

... 

k. Compare \([PE(i)].a\) with \([PE(i+2^{k-1})].a\) and
   
   \[PE[2i] \leftarrow \text{MIN} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]
   
   \[PE[2i+1] \leftarrow \text{MAX} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]

for \(j = 1\) to \(k\)

   Compare \([PE(i)].a\) with \([PE(i+2^{k-1})].a\)
   
   \[PE[2i] \leftarrow \text{MIN} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]
   
   \[PE[2i+1] \leftarrow \text{MAX} \{ [PE(i)].a, [PE(i+2^{k-1})].a \}\]
Given Unsorted list:

98 24 42 83 68 04 93 28

98 24 42 and 83 in ascending order and
68 04 93 and 28 in descending order

98 24 42 83
98 24 42 83
24 98 83 42
24 42 83 98
24 42 83 98

98 24 42 83 68 04 93 28
Graph Algorithms and Maximum Flow Networks

This week
- Graph terminology
- Stacks and Queues
- Breadth-first-search
- Depth-first-search
- Connected Components
- Analysis of BFS and DFS Algorithms

Further Reading
Chapter 22 .. 26 from Textbook

Graph Preliminaries

Examples of modeling by Graphs

Module 1 → Module 2
Module 3 → Module 4
Module 5
Module 6 → Module 7
The town of Konigsberg (now Kaliningrad) lay on the banks and on two islands of the Pregel river. The city was connected by 7 bridges. The puzzle (as encountered by Leonhard Euler in 1736) : Whether it was possible to start walking from anywhere in town and return to the starting point by crossing all bridges exactly once.

**Graph Terminologies**

- A Graph consists of a set 'V' of vertices (or nodes) and a set 'E' of edges (or links).
- A graph can be directed or undirected.
- Edges in a directed graph are ordered pairs.
  - The order between the two vertices is important.
    - Example: (S,P) is an ordered pair because the edge starts at S and terminates at P.
    - The edge is unidirectional
    - Edges of an undirected graph form unordered pairs.
- A multigraph is a graph with possibly several edges between the same pair of vertices.
- Graphs that are not multigraphs are called simple graphs.
Graph Terminologies

The degree \( d(v) \) of a vertex \( v \) is the number of edges incident to \( v \).

\[
d(A) = \text{three}, \quad d(D) = \text{two}
\]

In directed graphs, indegree is the number of incoming edges at the vertex and outdegree is the number of outgoing edges from the vertex.

The indegree of \( P \) is 2, its outdegree is 1.
The indegree of \( Q \) is 1, its outdegree is 1.
Paths and Cycles

A path from vertex $v_1$ to $v_k$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$.

Path from D to E : (D, A, B, E)
Edges in the path : (D, A), (A, B), (B, E)

A path is simple if each vertex in it appears only once.

DABE is a simple path.
ABCDABE is not a simple path.

Vertex $u$ is said to be reachable from $v$ if there is a path from $v$ to $u$.

A circuit is a path whose first and last vertices are the same.

DAEBCEAD, ABEA, DABECD, SPQRS, STRS are circuits

A simple circuit is a cycle if except for the first (and last) vertex, no other vertex appears more than once.

ABEA, DABECD, SPQRS, and STRS are cycles.

A Hamiltonian cycle of a graph $G$ is a cycle that contains all the vertices of $G$.

DABECD is a Hamiltonian cycle of $G_1$
PQRSTP is a Hamiltonian of $G_2$. 
A subgraph of a graph $G = (V,E)$ is a graph $H(U,F)$ such that $U \subseteq V$ and $F \subseteq E$.

$H_1 \{[U_1:A,E,C,D], F_1:[(A,E),(E,C),(C,D),(D,A)]\}$ is a subgraph of $G_1$

$H_2 \{[U_2:S,P,T], F_2:[(S,P),(S,T),(T,P)]\}$ is a subgraph of $G_2$.

Spanning tree of $G_1$

A spanning tree of a graph $G$ is a subgraph of $G$ that is a tree and contains all the vertices of $G$.

Spanning tree of $G_2$
Connectivity

A graph is said to be connected if there is a path from any vertex to any other vertex in the graph.
G1 and G2 are both connected graphs
A forest is a graph that does not contain a cycle.
A tree is a connected forest.
A spanning forest of an undirected graph G is a subgraph of G that is a forest and contains all the vertices of G.
If a graph G(V,E) is not connected, then it can be partitioned in a unique way into a set of connected subgraphs called connected components.
A connected component of G is a connected subgraph of G such that no other connected subgraph of G contains it.

Forest

G(A,B,C,D,E,P,Q,R,S,T) is a forest
G(A,B,C,D,E) is a tree
(A,B,C,D,E) and (P,Q,R,S,T) are connected components
Graph Representations

G1: undirected graph

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency list

A B D E
B A C E
C B D E
D A C
E A B C

G2: Directed Graph

Adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency list

P Q /
Q R /
R S /
S P T
T P R
Depth-first search

Procedure DFS Tree G(V,E)
Input: G = (V,E); S is a stack - initially empty;
'x' refers to the top of stack;
initially mark all vertices 'new';
L[x] refers to the adjacency list of x.
T ← {0};
Output : The DFS tree T;

1. v ← old; v ∈ V
2. push (S,v);
3. while S is nonempty do
4. while there exists a vertex w in L[x] and marked new do
5. T ← T ∪ (x,w);
6. w ← old;
7. push w onto S
8. pop S

DFS

8/18/2005 M KUMAR CSE5311 63
DFS

Initially, T = {0}; S {0}, A,B,C,D,E (all new)
Starts at A :  A, S :  {A}, L[A] = {B,D,E}
Pick B from L[A]; T = {(A,B)} and B (it's marked old)
S = {A,B}, L[B] = {A,C,E}
Pick C from L[B]; T = {(A,B), (B,C)} and C
S = {A,B,C}; L[C] = {B,D,E}
Pick D from L[C] ; T = {(A,B), (B,C), (C,D)} and D
S = {A,B,C,D} ; L[D] ={A,C}; no new vertices;
S = {A,B,C}; L[C] = {B,D,E}
Pick E from L[C]; T = (A,B), (B,C), (C,D),(C,E) and E
S = {A,B,C,E} ; L[E] = {A,B,C}
S = {A,B,C}; L[C] = {B,D,E} 
S ={A,B} ; L[B]= { A,C,E }
S ={A} ; L[A] = { B,C,E}
S = {0}
Breadth-first search

Procedure BFS_Tree G(V,E)
Input: G = (V,E); Q is a queue - initially empty;
x ← Q : remove the front item of queue and denote it by x;
in initially mark all vertices 'new';
L[x] refers to the adjacency list of x.
T ← {0}
Output: The BFS tree T;
1. v ← old; v ∈ V
2. insert (Q,v);
3. while Q is nonempty do
4. x ← Q
5. for each vertex w in L[x] and marked 'new'
6. T ← T ∪ {x,w} ;
7. w ← old;
8. insert (Q,w);

8/18/2005 M KUMAR CSE5311 67

BFS

8/18/2005 M KUMAR CSE5311 68
BFS

Initially, T = {} ; Q {0}, A,B,C,D,E (all new)
Starts at A : A, Q : {A}, L[A] = {B,D,E}
   Pick B from L[A]; T = {{A,B}} and B (it's marked old)
   Q = {B}, L[A] = {B,D,E}
   Pick D from L[A]; T = {{A,B}, (A,D)} and D
   Q = {B,D,E}; L[A] = {B,D,E}; no new vertices;
   Dequeue, Q = {D,E} L[B] = {A,C,E};
   Pick C from L[B]; T ={(A,B), (A,D), (A,E),(B,C)} and C
   Q = {E, C} ; L[D] = {A,C}
   Q = {C} ; L[E] = {A,B,C}
   Q = { 0) ; L[C] = (B,C,E)
   Q = {0};
Connected Components of a Graph

The connected component of a graph \( G = (V,E) \) is a maximal set of vertices \( U \subseteq V \) such that for every pair of vertices \( u \) and \( v \) in \( U \), we have both \( u \) and \( v \) reachable from each other. In the following we give an algorithm for finding the connected components of an undirected graph.

Procedure Connected_Components \( G(V,E) \)
Input : \( G (V,E) \)
Output : Number of Connected Components and \( G_1, G_2 \) etc, the connected components
1. \( V' \leftarrow V \);
2. \( c \leftarrow 0 \);
3. while \( V' \neq 0 \) do
   4. choose \( u \in V' \);
   5. \( T \leftarrow \) all nodes reachable from \( u \) (by DFS_Tree)
   6. \( V' \leftarrow V' - T \);
   7. \( c \leftarrow c+1 \);
   8. \( G_c \leftarrow T \);
   9. \( T \leftarrow 0 \);

Suppose the DFS tree starts at A, we traverse from \( A \rightarrow B \rightarrow C \rightarrow D \) and do not explore the vertices F, G, and H at all! The DFS_tree algorithm does not work with graphs having two or more connected parts.

We have to modify the DFS_Tree algorithm to find a DFS forest of the given graph.
DFS Forest

Procedure DFSForest \_G(V,E)

Input: \( G = (V,E) \); S is a stack - initially empty;
\( 'x' \) refers to the top of stack; initially mark all vertices 'new';
\( L[x] \) refers to the adjacency list of \( x \).
\( F \leftarrow \{0\}; \) The DFS Forest

Output: The DFS tree \( F \);
1. For each vertex \( v \in V \) do
2. if \( v \) is new
3. \( v \leftarrow \text{old}; \)
4. push \((S,v)\);
5. while \( S \) is nonempty do
6. while there exists a vertex \( w \) in \( L[x] \) and marked 'new' do
7. \( F \leftarrow F \cup (x,w) \);
8. \( w \leftarrow \text{old}; \)
9. push \( w \) onto \( S \)
10. pop \( S \)
- Do you know the difference between a simple graph and a multiple graph?
- What is an adjacency matrix?
- What is a Hamiltonian path? What is an Euler path?
- Given a graph, can you find the Hamiltonian and Eulerian paths?
- Given a graph, can you perform DFS and BFS traversals?
- What is the difference between a cycle and a path?
- What are the complexities of basic operations on stacks and queues? Give proof.

---

**Minimum-Cost Spanning Trees**

Consider a network of computers connected through bidirectional links. Each link is associated with a positive cost: the cost of sending a message on each link.

This network can be represented by an undirected graph with positive costs on each edge.

In bidirectional networks we can assume that the cost of sending a message on link does not depend on the direction.

Suppose we want to broadcast a message to all the computers from an arbitrary computer.

The cost of the broadcast is the sum of the costs of links used to forward the message.
Minimum-Cost Spanning Trees

• Find a fixed connected subgraph, containing all the vertices such that the sum of the costs of the edges in the subgraph is minimum. This subgraph is a tree as it does not contain any cycles.
• Such a tree is called the spanning tree since it spans the entire graph G.
• A given graph may have more than one spanning tree
• The minimum-cost spanning tree (MCST) is one whose edge weights add up to the least among all the spanning trees
MCST

- **The Problem**: Given an undirected connected weighted graph \( G = (V, E) \), find a spanning tree \( T \) of \( G \) of minimum cost.

- **Greedy Algorithm for finding the Minimum Spanning Tree of a Graph \( G = (V, E) \)**

  The algorithm is also called **Kruskal's algorithm**.

  - At each step of the algorithm, one of several possible choices must be made,
  - The greedy strategy: make the choice that is the best at the moment

**Kruskal's Algorithm**

- Procedure \( \text{MCST}_G(V, E) \)
- (Kruskal's Algorithm)
- **Input**: An undirected graph \( G(V, E) \) with a cost function \( c \) on the edges
- **Output**: \( T \) the minimum cost spanning tree for \( G \)
- \( T \leftarrow 0; \)
- \( V_S \leftarrow 0; \)
- for each vertex \( v \in V \) do
  - \( V_S = V_S \cup \{v\}; \)
  - sort the edges of \( E \) in nondecreasing order of weight
- while \( |V_S| > 1 \) do
  - choose \( (v, w) \) an edge \( E \) of lowest cost;
  - delete \( (v, w) \) from \( E; \)
  - if \( v \) and \( w \) are in different sets \( W_1 \) and \( W_2 \) in \( V_S \) do
    - \( W_1 = W_1 \cup W_2; \)
    - \( V_S = V_S - W_2; \)
    - \( T \leftarrow T \cup (v, w); \)
- return \( T \)
MCST

- The algorithm maintains a collection VS of disjoint sets of vertices
- Each set W in VS represents a connected set of vertices forming a spanning tree
- Initially, each vertex is in a set by itself in VS
- Edges are chosen from E in order of increasing cost, we consider each edge \((v, w)\) in turn; \(v, w \in V\).
- If \(v\) and \(w\) are already in the same set (say \(W\)) of VS, we discard the edge
- If \(v\) and \(w\) are in distinct sets \(W_1\) and \(W_2\) (meaning \(v\) and/or \(w\) not in \(T\)) we merge \(W_1\) with \(W_2\) and add \((v, w)\) to \(T\).

Consider the example graph shown earlier,
The edges in nondecreasing order
\[
[(A,D),1], [(C,D),1], [(C,F),2], [(E,F),2], [(A,F),3], [(A,B),3],
[(B,E),4], [(D,E),5], [(B,C),6]
\]
EdgeActionSets in VS
Spanning Tree, \(T = \{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\) 0
\((A,D)\) merge
\(\{A,D\}, \{B\}, \{C\}, \{E\}, \{F\}\) \((A,D)\) merge
\(\{A,C,D\}, \{B\}, \{E\}, \{F\}\) \((A,D), (C,D)\) (C,F) merge
\(\{A,C,D,F\}, \{B\}, \{E\}\) \((A,D), (C,D), (C,F)\) (E,F) merge
\(\{A,C,D,E,F\}, \{B\}\) \((A,D), (C,D), (C,F), (E,F)\) (A,F) reject
\(\{A,C,D,E,F\}, \{B\}\) \((A,D), (C,D), (C,F), (E,F)\) (A,B) merge
\(\{A,B,C,D,E,F\}\) \((A,D), (A,B), (C,D), (C,F), (E,F)\) (B,E) reject
\(D,E\) reject
\(B,C\) reject
Complexity

- Steps 1 thru 4 take time $O(V)$
- Step 5 sorts the edges in nondecreasing order in $O(E \log E)$ time
- Steps 6 through 13 take $O(E)$ time
- The total time for the algorithm is therefore given by $O(E \log E)$
- The edges can be maintained in a heap data structure with the property,
  - $E[\text{PARENT}(i)] \leq E[i]$
  - remember, this property is the opposite of the one used in the heapsort algorithm earlier during Week 2. This property can be used to sort data elements in nonincreasing order.
- Construct a heap of the edge weights, the edge with lowest cost is at the root
- During each step of edge removal, delete the root (minimum element) from the heap and rearrange the heap.
- The use of heap data structure reduces the time taken because at every step we are only picking up the minimum or root element rather than sorting the edge weights.

Single Source Shortest Paths

All Pairs Shortest Path Problem
A motorist wishes to find the shortest possible route from from Perth to Brisbane. Given the map of Australia on which the distance between each pair of cities is marked, how can we determine the shortest route?

In a shortest-paths problem, we are given a weighted, directed graph \( G = (V,E) \), with weights assigned to each edge in the graph. The weight of the path \( p = (v_0, v_1, v_2, \ldots, v_k) \) is the sum of the weights of its constituent edges:

- \( v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k-1 \rightarrow v_k \)

The shortest-path from \( u \) to \( v \) is given by

- \( d(u,v) = \min \{ \text{weight} (p) : \text{if there are one or more paths from } u \text{ to } v \} \)
- \( = \infty \) otherwise
The single-source shortest paths problem

Given $G(V,E)$, find the shortest path from a given vertex $u \in V$ to every vertex $v \in V$ ($u \neq v$).

For each vertex $v \in V$ in the weighted directed graph, $d[v]$ represents the distance from $u$ to $v$.

Initially, $d[v] = 0$ when $u = v$.
$d[v] = \infty$ if $(u,v)$ is not an edge
$d[v] = \text{weight of edge } (u,v)$ if $(u,v)$ exists.

Dijkstra's Algorithm: At every step of the algorithm, we compute,
\[ d[y] = \min \{d[y], d[x] + w(x,y)\}, \text{ where } x,y \in V. \]

Dijkstra's algorithm is based on the greedy principle because at every step we pick the path of least weight.

- Dijkstra's Algorithm: At every step of the algorithm, we compute,
\[ d[y] = \min \{d[y], d[x] + w(x,y)\}, \text{ where } x,y \in V. \]
- Dijkstra's algorithm is based on the greedy principle because at every step we pick the path of least weight.
Example:

```
<table>
<thead>
<tr>
<th>Step #</th>
<th>Vertex to be marked</th>
<th>u</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>Unmarked vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>∞</td>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>a,b,c,d,e,f,g,h</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>∞</td>
<td>=</td>
<td>∞</td>
<td>∞</td>
<td>b,c,d,e,f,g,h</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>∞</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>b,d,e,f,g,h</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>d,e,f,g,h</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>∞</td>
<td>e,f,g,h</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>f,g,h</td>
</tr>
<tr>
<td>6</td>
<td>h</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>g,h</td>
</tr>
<tr>
<td>7</td>
<td>g</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>--</td>
</tr>
</tbody>
</table>
```
Dijkstra's Single-source shortest path

- Procedure **Dijkstra's Single-source shortest path** \(G(V,E,u)\)
- Input: \(G = (V,E)\), the weighted directed graph and \(v\) the source vertex
- Output: for each vertex, \(v\), \(d[v]\) is the length of the shortest path from \(u\) to \(v\).
  - mark vertex \(u\);
  - \(d[u] \leftarrow 0\);
  - for each unmarked vertex \(v \in V\) do
    - if edge \((u,v)\) exists \(d [v] \leftarrow \text{weight}(u,v)\);
    - else \(d [v] \leftarrow \infty\);
  - while there exists an unmarked vertex do
    - let \(v\) be an unmarked vertex such that \(d[v]\) is minimal;
    - mark vertex \(v\);
    - for all edges \((v,x)\) such that \(x\) is unmarked do
      - if \(d[x] > d[v] + \text{weight}[v,x]\) then
        - \(d[x] \leftarrow d[v] + \text{weight}[v,x]\)

- Complexity of Dijkstra's algorithm:
  - Steps 1 and 2 take \(\Theta(1)\) time
  - Steps 3 to 5 take \(O(|V|)\) time
  - The vertices are arranged in a heap in order of their paths from \(u\)
  - Updating the length of a path takes \(O(\log V)\) time.
  - There are \(|V|\) iterations, and at most \(|E|\) updates
  - Therefore the algorithm takes \(O((|E| + |V|) \log |V|)\) time.
All-Pairs Shortest Path Problem

Consider a shortest path $p$ from vertex $i$ to vertex $j$

If $i = j$ then there is no path from $i$ to $j$.

If $i \neq j$, then we decompose the path $p$ into two parts,
$\text{dist}(i,k)$ and $\text{dist}(k,j)$

$$\text{dist}(i,j) = \text{dist}(i,k) + \text{dist}(k,j)$$

Recursive solution

$$\text{dist}(i,j) = \begin{cases} w(i,j) & \text{if } k = 0 \\ \min \{ \text{dist}(i,j), [\text{dist}(i,k) + \text{dist}(k,j)] \} & \text{if } k \geq 1 \end{cases}$$

Floyd's Algorithm for Shortest Paths

- Procedure $\text{FLOYDs}_G=[V,E]$
- Input: $n \times n$ matrix $W$ representing the edge weights of an $n$-vertex directed graph.
  That is $W = w(i,j)$ where, (Negative weights are allowed)
- Output: shortest path matrix, $\text{dist}(i,j)$ is the shortest path between vertices $i$ and $j$.

- for $v \leftarrow 1$ to $n$ do
  - for $w \leftarrow 1$ to $n$ do
    - $\text{dist}[v,w] \leftarrow \text{arc}[v,w]$;
  - for $u \leftarrow 1$ to $n$ do
    - for $v \leftarrow 1$ to $n$ do
      - for $w \leftarrow 1$ to $n$ do
        - if $\text{dist}[v,u] + \text{dist}[u,w] < \text{dist}[v,w]$ then
          - $\text{dist}[v,w] \leftarrow \text{dist}[v,u] + \text{dist}[u,w]$

Complexity: $\Theta(n^3)$
Distances after using A as the pivot

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Distances after using B as the pivot

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
Transitive Closure

- Given a directed graph $G=(V,E)$, the transitive closure $C=(V,F)$ of $G$ is a directed graph such that there is an edge $(v,w)$ in $C$ if and only if there is a directed path from $v$ to $w$ in $G$.
- Security Problem: the vertices correspond to the users and the edges correspond to permissions. The transitive closure identifies for each user all other users with permission (either directly or indirectly) to use his or her account. There are many more applications of transitive closure.
- The recursive definition for transitive closure is
  \[ t(i,j) = \begin{cases} 
  0 & \text{if } ij \notin E \text{ and } (i,j) \notin E \\
  1 & \text{if } ij \text{ and } (i,j) \in E 
  \end{cases} \]
Warshall's Algorithm for Transitive Closure

- Procedure **WARSHALL's(G=\{V,E\})**
- **Input:** \(n\times n\) matrix \(A\) representing the edge weights of an \(n\)-vertex directed graph. That is \(a = a(i,j)\) where,
- **Output:** transitive closure matrix, \(t(i,j) = 1\) if there is a path from \(i\) to \(j\), \(0\) otherwise
- **for** \(v \leftarrow 1\) to \(n\) **do**
  - **for** \(w \leftarrow 1\) to \(n\) **do**
    - \(t[v,w] \leftarrow a(v,w)\)
  - **for** \(u \leftarrow 1\) to \(n\) **do**
    - **for** \(v \leftarrow 1\) to \(n\) **do**
      - **for** \(w \leftarrow 1\) to \(n\) **do**
        - **if NOT** \(t[v,w]\) **then**
          - \(t[v,w] \leftarrow t[v,u] \land t[u,w]\)
      - **return** \(T\)

- Hamiltonian Cycle
- Eulerian Path
- Biconnected Components
- Bipartite Graph Matching