COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Overview

- The decrease-and-conquer design strategies
- Decrease by a constant
  a. Insertion sort
  b. Depth-first search (DFS)
  c. Breadth-first search (BFS)
- Decrease by a constant factor
  a. Binary search
  b. Exponentiation by Squaring
- Decrease by variable size
  a. Euclid's Algorithm (GCD)
  b. Selection problem.

Decrease-and-conquer

- Reduce problem instance to smaller instance of the same problem
- Solve smaller instance
- Extend solution of smaller instance to obtain solution to original instance
- Can be implemented either top-down (recursive) or bottom-up (iterative)

Exponentiation problem: compute a^n

Different approaches to solve this problem:

1. Brute force: $a^n = a * a * \ldots * a$.

   $power \leftarrow 1$
   for $i \leftarrow 1..n$
   $\quad power \leftarrow power * a$
   return $power$

2. Divide and conquer:

   if $n = 1$, return a;
   else return $a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$

Computing a^n (2)

3. Decrease by one:
   if$n = 1$, return a;
   else return $a^{n-1} * a$

4. Decrease by constant factor (exponentiation by squaring):
   if$n = 1$, return a;
   else if n is even, return $(a^{n/2})^2$;
   else if n is odd, return $(a^{(n-1)/2})^2 * a$.

Question: what is the time complexity of each algorithm?

Insertion sort

- Comparison-based sorting (i.e., sort by swapping elements)
- Thinking recursively:
  To sort array $A[0..n-1]$, sort $A[0..n-2]$ recursively, and then insert $A[n-1]$ in its proper place among the sorted $A[0..n-2]$
- Usually implemented bottom up (nonrecursively)
- At the start of ith iteration, the first i elements are already sorted. We will insert the $(i+1)$-th element in its proper place in the sorted array.
- Example: 6, 4, 1, 8, 5

Pseudocode of insertion sort

ALGORITHM *InsertionSort*($A[0..n-1]$)

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Complexity of insertion sort

- Time efficiency:

$$C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$$
$$C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$$
$$C_{best}(n) = n - 1 \in \Theta(n)$$

- Best case: a sorted array; excellent performance on almost sorted arrays
- Overall, best elementary sorting algorithm (say, $n = 10$)
- Can be combined with quicksort to decrease the total running time of quicksort by about 10%
- Stable: reserve the relative order of elements with equal keys
- Space efficiency: in-place sorting

Definitions

- A **graph** $G =< V, E >$ is defined by a pair of two sets:
  a. a finite set of **Vertices** V
  b. a set of **Edges** E of pairs of the vertices in V.
- Graphs can be directed (**digraph**) and undirected:
  a. In an undirected graph, $(u, v) = (v, u)$
  b. In a directed graph, (u, v) implies that the edge goes from u to v.
- A **weighted graph** is a graph with numbers (**weights** or **costs**) assigned to its edges.

Graph representations

- **vertices**: stored as an array or list
- **edges**: stored as an *adjacency* matrix or *adjacency* lists.
- An adjacency matrix A of a graph with n vertices is an $n \times n$ matrix.
- $A[i,j] = 1$ if there is an edge from ith vertex to jth vertex.
- For an undirected graph, A is always symmetric.
- **weight** or **cost matrix**:
  $A[i,j] =$ weight of edge from ith vertex to jth vertex if edge exists
  $A[i,j] = \infty$ if there is no such edge.
- Adjacency lists of a graph is a collection of linked lists (one for each vertex) that contain all the vertices adjacent to the list's vertex

Path and Length

- A **path** from vertex u to v of a graph G is a sequence of adjacent vertices that starts with u and ends with v.
- A path is **simple** if all vertices of a path are distinct.
- The **length** of a path is the total number of edges in the path.

Connectivity and Acyclicity

- A graph is **connected** if for every pair of its vertices u and v there is a path from u to v
- A subgraph of a given graph $G =< V, E >$ is a graph $G' =< V', E' >$ such that $V' \subseteq V$ and $E' \subseteq E$
- A **connected component** is a maximal connected subgraph of a given graph.
  A connected component is not expandable via an inclusion of an extra vetex in the given graph.
- A **cycle** or **circuit** is a path of length > 0 that starts and ends at the same vertex and does not traverse the same edge more than once.
- A graph with no cycles is **acyclic**.

Depth-first search (DFS)

- Visit a graph's vertices by going as far as it can, backtrack if no adjacent unvisited vertex is available (i.e., dead end).
- Use a **stack**: Last In First Out (**LIFO**)
  - a vertex is pushed onto the stack when it's reached for the first time
  - a vertex is popped off the stack when it becomes a dead end.

Pseudocode of $DFS(G)$

ALGORITHM $DFS(G)$

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        $dfs(v)$

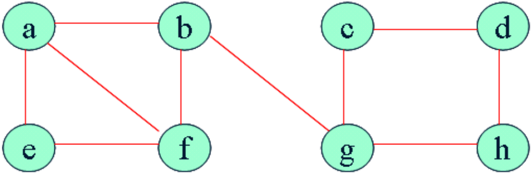
$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable $count$
$count \leftarrow count + 1$; mark v with $count$
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        $dfs(w)$

DFS Forest

Construct a depth-first search forest for an undirected graph:

- **tree edges**: reach previously unvisited vertices.
  when a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached.
- **back edges**: connect to previously visited vertices (ancestors) other than their parents.

DFS traversal



- DFS traversal stack:
- DFS forest:

Complexity of DFS

- DFS can be implemented with graphs represented as:
  a. adjacency matrices: $\Theta(|V|^2)$
  a. adjacency lists: $\Theta(|V| + |E|)$
- Two distinct ordering of vertices:
  a. an order in which vertices are first encountered (pushed onto stack)
  b. an order in which vertices become dead-ends (popped off stack)

DFS applications

- Connectivity
  1. Start a DFS traversal at an arbitrary vertex
  2. After the agrorithm halts, check whether all the graph's vertices have been visited
  3. If yes, the graph is connected; otherwise, it is not connected.
- Acyclicity
  1. Construct a DFS forest
  2. If there is a back edge from some vertex u to another vertex v, the graph has a cycle; otherwise, it is acyclic.
- Connected components

  Exercises 5.2 Problem 7: Explain how one can identify connected components of a graph by using a DFS?

Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a **queue**: First In First Out (**FIFO**)
- Similar to level-by-level tree traversal

Pseudocode of $BFS(G)$

ALGORITHM $BFS(G)$

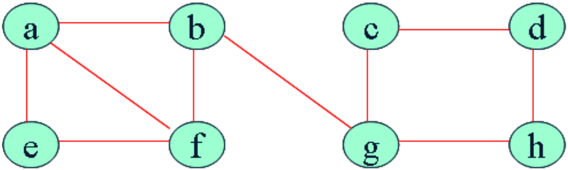
```
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked with consecutive integers
    //in the order they have been visited by the BFS traversal
    mark each vertex in V with 0 as a mark of being "unvisited"
    count ← 0
    for each vertex v in V do
        if v is marked with 0
          bfs(v)

    bfs(v)
    //visits all the unvisited vertices connected to vertex v by a path
    //and assigns them the numbers in the order they are visited
    //via global variable count
    count ← count + 1;   mark v with count and initialize a queue with v
    while the queue is not empty do
        for each vertex w in V adjacent to the front vertex do
            if w is marked with 0
                count ← count + 1;   mark w with count
                add w to the queue
        remove the front vertex from the queue
```

BFS Forest

Construct a breadth-first search forest for an undirected graph:

- **tree edges**: reach previously unvisited vertices, same as DFS. when a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached.
- **cross edges**: connect to previously visited vertices other than their parents.
  Note: unlike the back edges in DFS tree, the cross edges connect vertices either on the same or adjacent levels of a BFS tree.

BFS traversal



- BFS traversal queue:
- BFS forest:

Complexity of BFS

- BFS has the same time efficiency as DFS
- Can be implemented with graphs represented as:
  a. adjacency matrices: $\Theta(|V|^2)$
  b. adjacency lists: $\Theta(|V| + |E|)$
- Single ordering of vertices:
  Same order of vertices added/deleted from queue.

BFS applications

- Connectivity
- Acyclicity
- Connected components
- Finding a minimum-edge path between two given vertices.

Greatest common divisor (gcd)

- Euclid's algorithm is based on repeated application of equality

$$gcd(m, n) = gcd(n, \quad m \mod n)$$

- Example: $gcd(80, 44) = gcd(44, 36) = gcd(36, 8)$
  $= gcd(8, 4) = gcd(4, 0) = 4$
- The input size is measured by the second number
- Decrease by variable size at each iteration, but at least decrease by half after two consecutive iterations.
- Efficiency: $T(n) \in O(\log n)$

- See Section 1.1 for more descriptions of the gcd algorithms.

Find the k-th smallest element

- $k = 1$: minimum
- $k = n$: maximum
- $k = \lceil n/2 \rceil$: median
- In general, find the kth smallest element, where $1 \leq k \leq n$
- Approaches:
  a. Sorting-based algorithm:
     Sort and return the k-th element
     Time efficiency (if sorted by mergesort): $\Theta(n \log n)$
  b. Partition-based algorithm:
     using the partition process similar to Quicksort.

Partition-based selection

Find the kth smallest item in A[1..n]

- Let s be a split position obtained by a partition
- If $s = k$, the problem is solved;
- if $s > k$, search for the kth smallest element in the left part;
- if $s < k$, search for the $(k - s)$th smallest element in the right part.
- Example: tracing the median selection process:

$$4, 1, 10, 9, 7, 12, 8, 2, 15$$

Complexity

- Average case (average split in the middle):

$$C(n) = C(n/2) + (n+1), \quad C(n) \in \Theta(n).$$

- Worst case (degenerate split): $C(n) \in \Theta(n^2)$
- A more sophisticated choice of the pivot leads to a complicated algorithm with $\Theta(n)$ worst-case efficiency.

Exercises

- Section 5.1: #3, 4, 5, 7
- Section 5.2: #1, 4, 6, 7
- Section 5.6: #2, 3

Reminder:

Midterm Examination
Friday July 24 2009