Theory of Computing

Lecture 3

MAS 714

Hartmut Klauck

How fast can we sort?

- There are deterministic algorithms that sort in worst case time O(n log n)
- This is best possible for comparison based algorithms
- Do better algorithms exist?
  - Example [Andersson et al. 95]:
    On a unit cost RAM, word length w, one can sort n integers in the range 0…2^w in time O(n loglog n)
    Even in O(n) if $w>\log^2 n$
  - Not comparison based!

A linear time sorting algorithm

- Assume all A[i] are integers between 1 and m
- Sorting algorithm:
  - for i=1 to m
    - c[i]=0
  - for i=1 to n
    - c[A[i]]=c[A[i]]+1
  - for i=1 to m
    - If c[i]> 0 output i, c[i] times
- Clearly this algorithm runs in time O(n+m), linear if m=O(n)
- Algorithm is not comparison based (second loop)

Counting Sort

- The above algorithm has the drawback that it sorts only a list of numbers (keys), with no other data attached
- To properly sort (including additional data) we need to compute where items with key K start in the sorted sequence and move the data there
- Furthermore we want the algorithm to be *stable*
  - Stability: Items with the same key remain in the same order

Counting Sort

- for i=1 to m: C[i]=0 //Initialize
- for i=1 to n: C[A[i]]++ //Count elements
- Pos[1]=1 //Array of positions
- for i = 2 to m: //Compute positions
  Pos[i]=Pos[i-1]+C[i-1]
- for i=1 to n: //Produce Output
  Output[Pos[A[i]] = A[i]
  Pos[A[i]]++

Counting Sort

- The third loop computes the position Pos[i], at which elements with key i start in the sorted array
  - Pos[1]=1
  - Pos[i]=Pos[i-1]+C[i-1]
- The fourth loop copies elements A[i] into the array Output, at the correct positions
  - Data Dat[i] attached to the keys may be copied as well
- The algorithm is stable, because we keep elements with the same key in their original order

Linear time sorting

- ***Radix sort*** sorts n integer numbers of size n^k in time O(kn)
- This is linear time for k=O(1)
- I.e., we can sort polynomial size integers in linear time

Radix Sort

- Main Idea:
  - Represent n numbers in a number system with base n
  - Given that numbers are size n^k the representation has at most k+1 digits
  - Sort by digits from the least significant to the most significant
  - Use a stable sorting algorithm
    - For each step use Counting Sort

Radix Sort

- Rewrite keys x in the format $\sum_{i=0...k} x_i n^i$
- x is then represented by $(x_k,..,x_0)$
- Sort the sequence by digit/position 0, i.e. sort the sequence using the x_0 digits as keys
- Stably sort on position 1
- etc. for all positions k
- Time is O(kn)=O(n) for k=O(1)
- Note: not comparison based, only works for sorting „small“ integer numbers

Radix Sort

- Correctness:
- Let x,y be two numbers in the sequence.
- Let x_i denote the most significant position on which they differ
- Then step i puts x,y in the right order, and later steps never change that order (due to the stability of counting sort)

Further topics about sorting

- Time versus space
- Sorting on parallel machines
- Sorting on word RAMs, faster than n log n
- Deterministic sorting in O(n log n)

Graph Algorithms

- Many beautiful problems and algorithms
- Good setting to study algorithm design techniques

Graphs

- A graph G=(V,E) consists of a set of vertices V and a set E of edges. $E \subseteq V \times V$
  - usually there are n vertices
  - usually there are m edges
- Graphs can be undirected $(i,j) \in E \Rightarrow (j,i) \in E$ or directed (no such condition)
  - Edges of undirected graphs are pairs of vertices
- Edges (i,i) are called selfloops and are often excluded

Graph problems

- Example: Friendship graph
  - Vertices represent people
  - Edges are between friends
- Example: What is the largest size of a set S of vertices such that every pair of vertices in S are connected
  - Clique
- Example: Find a large set of edges so that no vertex is in more than one edge
  - Matching

Graph Rpresentations

- There are two main ways to represent graphs:
  - Adjacency Matrix
  - Adjacency List

Adjacency Matrix

- The ***adjacency matrix*** of a graph G=(V,E) has n rows and columns labeled with vertices
- A[i,j]=1 iff (i,j)$\in$ E
- Works for both undirected and directed graphs
  - undirected graphs may use only the entries above the diagonal

AdjacencyMatrix

- Advantages:
  - easy access to edges
  - can do linear algebra on the matrix
- Disadvantage:
  - not a compact representation of sparse graphs
  - sparse means $m=o(n^2)$ [or even $m=O(n)$]
  - Algorithms take time n^2 at least for many problems

Adjacency List

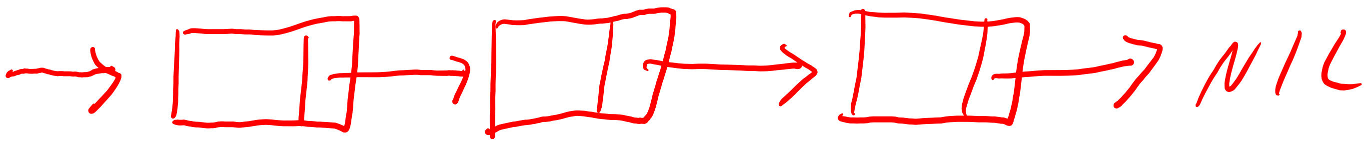
- The ***adjacency list*** of G=(V,E) is an array of length n. Each entry in the array is a list of edges adjacent to $v \in V$
- For directed graphs a list of edges starting in v
- Size of the representation is O(n+m) entries, close to optimal
- It is harder to find a specific edge
- Standard representation for graphs

Linked Lists

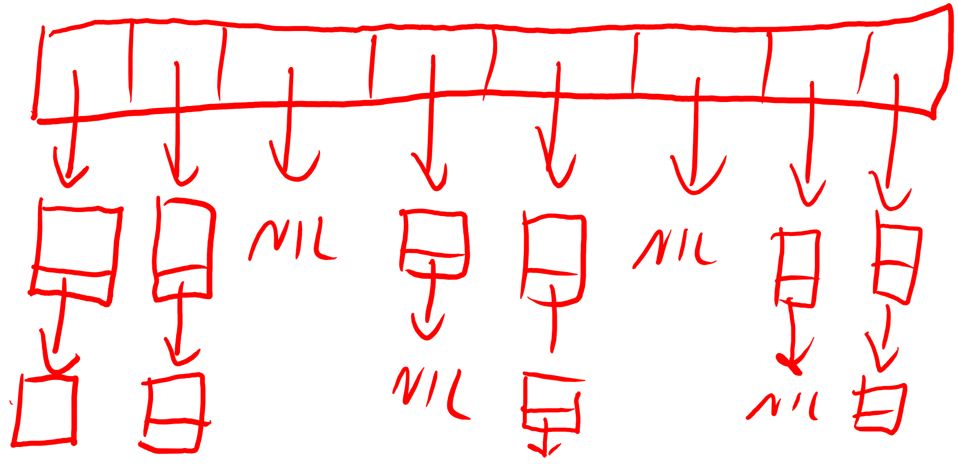
- The list of vertices adjacent to v has variable length for different v
- Use a ***linked list***
- Linked lists are a datastructure to represent sequences
  - A linked list consists of nodes
  - Each node consists of a cell for data and a pointer
  - There is a pointer to the first element
  - Last element points to NIL
  - It is easy to add an element into a linked list, and to sequentially read the list
- Advantage over arrays: length is arbitrary/can be changed
- Disadvantage: no direct access to edges

Linked List/Adjacency List

- Example of a linked list



- Adjacency list



Weighted Graphs

- Graphs often come with weights
  - Weights on vertices
  - Weights on edges
- Example: Directed Graph with weighted edges
  - Represent as a matrix of weights
  - Either 0 or ∞ marks absence of an edge

Example Problem

- **Single Source Shortest Path (SSSP)**
- Give a directed graph G with nonnegative edge weights, a vertex s
  - Inputs(V,E) and W: $E \rightarrow R^+$ and s
- Output: the length of the shortest paths in the graph from s to all other vertices
  - Array of n distances
- Explanation: A path from s to v is a sequence of edges (s,v_1), $(v1,v_2)\ldots(v_t,v)$
- The length of a path is the sum of edge weights on the path

Traversing Graphs

- Example: Finding the exit of a maze

Traversing Graphs

- We are given a graph G=(V,E)
- Starting vertex s
- The goal is to traverse the graph, i.e., to visit each vertex at least once
  - For example to find a marked vertex t or decide if t is reachable from s
- Two variants:
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

Traversing Graphs

- Common to both procedures:
  - Use a datastructure with the following operations:
    - Insert a vertex
    - Remove a vertex
  - Maintain an active vertex (start with s)
  - Maintain an array of vertices already visited
  - Then:
    - Insert all (unvisited) neighbors of the active vertex, mark it as visited
    - Remove a vertex v and make it active

The Datastructure

- We distinguish by the rule that determines the next active vertex

- Alternative 1: queue
  - FIFO (first in first out)
- Alternative 2: stack
  - LIFO (last in first out)

Result

- Alternative 1: FIFO
  - Breadth First Search
  - Neighbors of s will be visited before their neighbors etc.
- Alternative 2: LIFO
  - Depth First Search
  - Insert neighbors, last neighbor becomes active, then insert his neighbors, last neighbor becomes active etc.

Traversing Graphs

- With both methods eventually all reachable vertices are visited
- Different applications:
  - BFS can be used to find shorted paths in unweighted graphs
  - DFS can be used to topologically sort a directed acyclic graph

Datastructures: Queue

- A queue is a linked list together with two operations
  - Insert: Insert an element at the rear of the queue
  - Remove: Remove the front element of the queue
- Implementations is as a linked list
  - We need a pointer to the rear and a pointer to the front

BFS

- Every time we put a vertex v into the queue, we also remember the predecessor of v, i.e., the vertex π(v) as who‘s neighbor v was queued
- And remember d(v), which will be the distance of v from s

BFS

- Procedure:
  - For all v:
    - visit(v)=0, d(v)=∞,π(v)=NIL
  - d(s)=0
  - Enter s into the queue Q
  - While Q is not empty
    - Remove v from Q
    - visit(v)=1, enter all neighbors w of v with visit(w)=0 into Q and set π(w)=v, d(w)=d(v)+1

BFS

- Clearly the running time of BFS is O(m+n)
  - n to go over all vertices
  - m to check all neighbors
  - Each queue operation takes constant time
- BFS runs in linear time

BFS tree

- Consider all edges (π(v), v)
- Claim: These edges form a tree

- This tree is called the BFS tree of G (from s)
  - vertices not reachable from s are not in the tree

BFS tree

- Proof (of Claim):
  - Each visited vertex has 1 predecessor (except s)
  - V_s is the set of visited vertices
  - Graph is directed
  - There are $|V_s-1|$ edges
  - Hence the edges form a tree

Shortest Paths

- BFS can be used to compute shortest paths
  - in unweighted graphs
- Definition:
  - Graph G, vertex s
  - δ_G(s,v) is the minimum number of edges in any path from s to v
    - No path: ∞

Shortest Paths

- Lemma:
  - Let (u,v) be an edge
  - Then: $\delta(s,v) \leq \delta(s,u)+1$

- Proof: v is reachable $\Rightarrow$ u is reachable
  - Shortest path from s to u cannot be longer than shortest path from s to v plus one edge
    - Triangle inequality

Shortest Paths

- Lemma:
  - The values d(v) computed by BFS are the δ(s,v)
- Proof:
  - First, show that d(v)$\geq\delta$(s,v)
  - Induction over the number of steps
    - Surely true in the beginning
    - Suppose true, when we queue a vertex
    - Then also true for the neighbors

Shortest Paths

- Now we show that $d(v) \leq \delta(s,v)$
- **Observation:** For all vertices in Q, d(v) is only different by 1 (and Q has increasing d(v) by position in Q)
- Now assume that $d(v) > \delta(s,v)$ for some v
  - Choose some v with minimum $\delta(s,v) < d(v)$
- v is reachable from s (otherwise $\delta(s,v) = \infty$)
- Consider the predecessor u of v on a shortest path s to v
  - $\delta(s,v) = \delta(s,u) + 1$

Shortest Paths

- $d(v) > \delta(s,v) = \delta(s,u)+1 = d(u)+1$
  - Because v is minimal "violator"
- At some point u is removed from the queue
  - If v is unvisited and not in the queue, then $d(v)=d(u)+1$
  - If v is visited already then by our observation $d(v) \leq d(u)$
  - If v is unvisited, and in the queue, then $d(v) \leq d(u)+1$ (observation)
- Contradiction in any case

Shortest Paths

- Lemma: The BFS tree is a shortest path tree
- Proof:
  - We already saw it is a tree
  - (π(v),v) is always a graph edge
  - d(v) is the depth in the BFS tree
    - Induction: true for s
    - True for level d $\Rightarrow$ when v is added in level d+1 then d(v)=d+1
  - Hence a path from the root s following tree edges is a shortest path (has length d(v))

BFS

- Runs in time O(m+n) on adjacency lists
- Visits every vertex reachable from s
- Can be used to compute shortest paths from s to all other vertices in directed, unweighted graphs