UNIVERSITY OF OSLO

INF2220 - Summary

Ivar Haugaløkken Stangeby

December 10, 2014

CONTENTS

1 GRAPHS

1.1 Definitions

Definition (Connected graph)**.** An undirected graph is called **connected** if there is a path from any vertex to any other vertex in the graph. If the graph is directed, we call it **strongly connected**.

Definition (Biconnected graph)**.** A **biconnected** graph is a connected graph with the property that if any vertex were to be removed, the graph will remain connected.

Definition (Minimum spanning tree)**.** Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. We can also assign a weight to each edge and use it to assign a weight to a spanning tree. A **minimum spanning tree** (MST) is a spanning tree with weight less than or equal to the weight of every other spanning tree.

Definition (Acyclic graph)**.** An **acyclic graph** is a directed graph that contains no cycles

Definition (Topological sort)**.** A **topological sort** is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears *after* v_i in the ordering.

Definition (Indegree)**.** The **indegree** of a vertex v is defined as the number of edges (u, v).

1.2 Representation of graphs

ADJACENCY MATRIX There are several ways to represent a graph. The most simple way is called an **adjacency matrix**. The adjacency matrix is a two-dimensional array where for each edge (u, v) we set A$[u][v]$ to **true**. This representations is good for dense graphs. This means that there are a lot of edges in relation to the number of vertices.The space requirement is $\Theta\left(|V|^2\right)$

ADJACENCY LIST If the graph on the other hand is not dense, an **adjacency list** representation is a better option. For each vertex, keep a list of the adjacent vertices. The space requirement is then $O(|E| + |V|)$.

1.3 Graph algorithms

1.3.1 Topological Sort

See Pseudocode It is clear that any cyclic graphs can not be topologically sorted. The ordering is not necessarily unique, thus you might get different results. The usual algorithms for topological sorting have a running time $O(|V| + |E|)$.

1. Find a vertex with no incoming edges (indegree zero)
2. Print this vertex, and remove it, and its edges, from the graph.
3. Repeat from step 1 until all vertices has been checked.

1.3.2 Dijkstra's algorithm

See Pseudocode Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs. For a given source vertex in the graph, the algorithm finds the path with the lowest cost between that vertex, and any other vertex. In essence, the algorithm picks the unvisted vertex with the lowest distance, calculates the distance through it to each unvisited neighbor and updates the neighbor's distance if it is smaller. Mark the vertex as visited when all the neighbors have been checked.
It has a worst case performance of $O\left(|E| + |V| \log |V|\right)$

1. Set starting point distance to 0, and ∞ for the rest.
2. Mark all nodes as unvisited, set initial node to current-node. Fill a list with all the unvisited nodes.
3. For current node - consider all of its unvisited neighbors and calculate their tentative-distance = currentnodedistance + cost of edge traversal.
4. When done considering all the neighbors of the current, mark current as visited and remove it from unvisited-list.
5. If the destination node has been marked visited, or if the smallest tentative distance among the nodes in unvisited set is ∞, then stop. The algorithm has finished.
6. Select the unvisited node with the smallest tentative distance, and set it as new current-node and repeat from step 3.

1.3.3 Prim's algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph.
Its complexity relies entirely on the data structure used. Using an adjacency matrix the algorithm has a worst case performance of $O\left(|V|^2\right)$.

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph
2. Find the minimum weight edge from a vertex in the tree to a vertex not in the tree and transfer it to the tree.
3. Repeat step 2 - until all vertices are in the tree.

1.3.4 Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, it finds a minimum spanning forest.

1. Create a forest F, where each vertex in the graph is a separate tree.

2. Create a set S, containing all the edges in the graph.
3. while S is nonempty and F is not yet spanning.
   a) remove an edge with minimum weight from S.
   b) if that edge connects two different trees, then add it to the forest, combining the two trees into a single tree.

2 Sorting

2.1 Algorithms

2.1.1 Insertion Sort

Insertion sort iterates through an array from left to right, swapping x leftwards until all elements to the left are smaller than x. The complexity of insertion sort is $O(N^2)$.

2.1.2 Mergesort

Mergesort splits an array into two partitions, and then each partition in two until the size of the partitions is one (or < 50 if using the insertion sort optimalization scheme), and then recursively merges adjacent partitions. The complexity of Mergesort is $O(N\log(N))$.

1. Let the array be a partition
2. Divide each partition into two partitions
3. Repeat previous step until all partitions are of size one (Think of this as a binary tree)
4. Merge any leaf nodes with the same parent
5. Repeat previous step until array is sorted

2.1.3 Quicksort

burgrs nd chipz Quicksort is a very fast sorting algorithm, based on very simple principles. The quicksort algorithm has an average time complexity of $O(N\log(N))$, and a worst case of $O(N^2)$. It's is very fast at sorting primitive data types in both Java and C.

3 Hashing

Hashing is a technique used for performing insertions, deletions and searches in constant average time. The general idea of hashing is to have an array of some fixed size to store the items to, and in one way or another generate a hash-key for each object. We need to find a hash function that distributes these keys evenly over the cells in the hash-table.

3.1 Hash Strategy

There are a lot of different strategies when it comes to hashing. We'll cover the following:

1. Separate Chaining
2. Linear Probing
3. Quadratic Probing
4. Double Hashing
5. Rehashing

3.1.1 Separate Chaining

Separate chanining involves keeeping a list of all elements that hash to the same value. We normally use Javas standard library list implementations. Separate Chaining is special in that that the implementation handles collision by ignoring it.
To perform an insert we check the appropriate list to see whether the element is already inserted. It is always appended to the back of the list, unless the element is not found.

3.1.2 Linear Probing

In linear probing f is a linear function of i.

```
void toposort() throws CycleFoundException{
  Queue<Vertex> q = new Queue<Vertex>();
  int counter;

  for each vertex v
  if (v.indegree = = 0)
  q.enqueue(v);

  while (!q.isempty()) {
    Vertex v = q.dequeue()
    v.topNum = ++counter;

    for each Vertex w adjacent to v
    if ( --w.indegree = = 0)
    q.enqueue(w);
  }
  if (counter != NUM_VERTICES )
  throw new CycleFoundException();
}
```

Figure 1.1: Pseudocode to perform topological sort

```
void dijkstra(Vertex s){
  for each Vertex v{
    v.dist = infinity;
    v.known = false;
  }

  s.dist = 0;

  while(there is an unknown distance vertex){
    Vertex v = smallest unknown distance vertex;
    v.known = true;

    for each Vertex w adjacent to v{
      if (!w.known){
        DistType cvw = cost of edge from v to w;

        if (v.dist + cvw < w.dist){
          decrease(w.dist to v.dist + cww);
          w.path = v;
        }
      }
    }
  }
}
```

Figure 1.2: Pseudocode for Dijkstra's algorithm

```
int i;

for (int j = 1; j<array.length; j++){
  Anytype tmp = array[j];

  for(i=j; i > 0 && tmp.compareTo(array[j-1]) < 0; j--){
    array[i] = array[i-1];
  }//for
  array[i] = tmp;
}//for
```

Figure 2.1: Insertion sort pseudocode

```
//Method that makes the recursive calls:
void mergeSort(anytype[] array, anytype[] tmparray,
               int left, int right){
  if(left < right){
    center = (left + right)/2;
    mergeSort(array, tmparray, left, center);
    mergeSort(array, tmparray, center+1, left);
    merge(array, tmparray, left ,center+1,right);
  }//if
}//mergesort

//Start:
void mergeSort(anytype[] array){
  anytype[] tmparray = new anytype[array.length];
  mergeSort(array,tmparray,0,array.length-1);
}//mergeSort

//Algorithm:
void merge(anytype[] array, anytype[] tmparray,
           int leftpos, int rightpos, int rightend){

  int leftend = rightpos -1;
  int tmppos = leftpos;
  int numofelements = rightend - leftpos + 1;

  while(leftpos <= leftend && rightpos <= rightend){
    if(array[leftpos].compareTo(array[rightpos]) <= 0){
      tmparray[tmppos++] = array[leftpos++];
    } else {
      tmparray[tmppos++] = array[rightpos++];
    }//else
  }//while

  while(leftpos <= leftend){
    tmparray[tmppos++] = array[leftpos++];
  }//while

  while(rightpos <= leftend){
    tmparray[tmppos++] = array[rightpos++];
  }//while

  for(int i = 0; i<numofelements; i++, rightend--){
    array[rightend] = tmparray[rightend];
  }//for

}//merge
```

Figure 2.2: Mergesort pseudocode

```
public static <AnyType extends Comparable<? super AnyType>>
              void quicksort( AnyType [] a){
  quicksort(a, 0, a.length-1);
}

private static <AnyType extends Comparable<? super AnyType>>
                anyType median3(AnyType[] a, int left, int right){
  int center = (left, right) / 2;
  if ( a[center].compareTo(a[left]) < 0){
    swapReferences(a, left, center);
  }
  if ( a[right].compareTo(a[left]) < 0){
    swapReferences(a, left, right);
  }
  if (a[right].compareTo(a[center]) < 0){
    swapReferences(a, center, right);
  }

  swapReferences(a, center, right - 1);
  return a[right-1];
}

private static <AnyType extends Comparable<? super AnyType>>
                void quicksort(AnyType [] a, int left, int right){
  if(left+CUTOFF <= right) {
    AnyType pivot = median3(a, left, right);

    int i = left, j = right-1;
    while(true){
      //This looks disturbing, but do NOT touch
      while(a[++i].compareTo(pivot)< 0){}
      while(a[--j].compareTo(pivot)> 0){}
      if(i < j){
        swapReferences(a, i, j);
      } else {
        break;
      }
    }

    swapRefrences(a, i, right-1);
    quicksort(a, left, i-1);
    quicksort(a, i + 1, right);

  } else {
    insertionSort(a, left, right);
}
```

Figure 2.3: Quicksort pseudocode