Graph Algorithms

9.1 – Definitions

* Graph G: a set of **vertices** V and **edges** (or **arcs**) E
* E = (u, v)
* If edge pairs are ordered (order matters) then G is **directed**, otherwise **undirected**
* **Digraph** = directed graph
* Vertex u is **adjacent to** vertex v only if (u, v) belongs to edges
* Edge objects can also contain **cost/weight**
* A **path** is a sequence of vertices where pairs of consecutive vertices belong to edges
* **Length** of a path of N vertices is N – 1
* If the graph contains an edge (v, v) then the path v, v is a **loop**
* A **simple path** is a path where all vertices are distinct except that the first and last vertices can be the same
* A **cycle** in a *directed* graph is a path of length at least 1 where v1 = vN, and is simple if the path is simple
* A cycle in an *undirected* graph must additionally have all distinct edges. For eg: u, v, u is not a cycle.
* A directed graph without cycles is **acyclic** and is called **Directed Acyclic Graph (DAG)**
* An undirected graph is **connected** if there’s a path from every vertex to any vertex
* A directed graph is **strongly connected** if there’s a path from every vertex to any vertex
* A directed graph is **weakly connected** it is strongly connected without considering directions
* 2 ways to represent a graph:
  + **Adjacency matrix**: a matrix of N vertices x N vertices. All the existing edges are filled with true/false if unweighted or the weight/arbitrary constants if unweighted
    - Better if G is dense: space complexity = O(|V|2), otherwise too many empty cells
  + **Adjacency list**: a list of adjacent vertices inside every Vertex object
    - Better if G is sparse: space complexity = O(|E| + |V|), more storage needed if you want to store the weight

9.2 – Topological Sort

* Basic idea: if there is a path from u to v, then u should appear before v in the ordering
* Algorithm:
  + Queue all vertices with no incoming edges (indegree = 0)
  + Print vertex/assign topological number
  + Remove vertex and its edges
  + Repeat till no edges left

9.3 – Shortest Path Algorithms

* Unweighted Shortest-Path Algorithm
  + Find the shortest unweighted path from a given source in a graph using Breadth First Search. Running time: O(|V|2).
  + Basic idea: search for vertices for every distance, starting from 0. For every unknown vertex v of that distance, mark v as known and set v’s unknown adjacent vertices to v.distance + 1, and their prev to v. Pseudocode:

for each Vertex v {

v.distance = INFINITY

v.known = false

}

s.distance = 0

for(int currDist = 0; currDist < NUM\_VERT; currDist++){

for each Vertex v {

if(!v.known && v.distance == currDist){

v.known = true

for each Vertex w adjacent to v{

if(w.distance == INFINITY){

w.dist = currDist + 1

w.prev = v

}

}

}

}

}

* Weighted Shortest-Path Algorithm: Dijkstra’s Algorithm
  + Greedy algorithm. Runs in O(|V|2)
  + Basic idea: Set v to the smallest unknown vertex, and mark as known. For every unknown adjacent vertex w to v, if w.cost > v.cost + cost(w,v) then set w.dist = v.cost + cost(w,v). Set w.prev to v.

for each Vertex v {

v.distance = INFINITY

v.known = false

}

s.distance = 0

while(there exists an unknown vertex {

Vertex v = vertex with smallest distance

v.known = true

for every adjacent edge v to w {

if(!w.known) {

Double wDist = cost of edge

if(v.dist + wDist < w.distance) {

w.distance = v.dist + wDist

w.prev = v

}

}

}

}

* Shortest Path with Negative Edge Costs
  + Dijkstra’s doesn’t work. We thus modify Dijkstra’s a little bit, at expense of running cost: O(|E|\*|V|)
  + Basic idea: Enqueue the source. While queue is not empty, for every neighbor w of dequeue() if w.cost > v.cost + cost(w,v) then update w and enqueue w if it’s not already in the queue.
  + Set v to the smallest unknown vertex, and mark as known. For every unknown adjacent vertex w to v, if w.cost > v.cost + cost(w,v) then set w.dist = v.cost + cost(w,v). Set w.prev to v.

9.5 – Minimum Spanning Tree

* A minimum spanning tree of an undirected connected graph G is a sequence of graph edges that connects all the vertices of G at lowest cost.
* Prim’s Algorithm
  + Similar to Djikstra’s. Runtime is O(|V|2)
    - Select unknown vertex with lowest cost and mark as known
    - For every unknown neighbor w, if cost(v, w) < w.distance then w.distance = cost(v, w) and w.prev = v
* Kruskal’s Algorithm
  + Uses sets. Runtime is O(|E|log|V|)
    - Create priority queue of all the edges
    - While all the edges are not counted,
    - Get the 2 sets of the vertices of the smallest edge
    - If they are NOT the same sets, add the edge to the mst
    - Union the 2 sets

9.7 – NP-completeness

* NP complete problems are those for which
  + No linear time solutions have been found
  + They are not guaranteed to run in polynomial time
  + Some can take exponential time
* They are all the same type of problems. Either all have polynomial time solutions or none of them do.
* **Undecidable problems** are those which are proven to be impossible to be solved by a computer
  + **Halting problem**: can you have a program that checks whether a given program will halt? No, because the program can’t also check itself. It’s thus recursively undecidable.
* **Nondeterministic polynomial-time/ NP** problems can be verified in polynomial time but not solved in polynomial time
* **NP-complete problems** are problems such that any NP problem can be polynomially reduced to it
  + **Travelling salesman problem**

Sorting

* Insertion sort
  + Sorted and unsorted portion strategy. Card hand strategy. Runs in O(n2).
  + Barrier put after first index. For ex:
    - P=0: 4|3 2 1
    - P=1: 3 4 | 2 1
    - P=2: 2 3 4 | 1
    - P=3: 1 2 3 4 |
  + Pseudocode:

for(int i = 1; i < n; i++) {

element = array[i]

j = i

while(array[j - 1] > element && j > 0) {

array[j] = array[j - 1];

j--;

}

array[j] = element;

}

* Selection sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sort | Best case | Worst case | Average case | Works well if |
| Insertion | O(n) | O(n2) | O(n2) | Input is small/mostly sorted |
| Selection | O(n2) | O(n2) | O(n2) | Input is small |
| Heapsort | O(nlg(n)) | O(nlg(n)) | O(nlg(n)) | Input is big. Space is imp. |
| Mergesort | O(nlg(n)) | O(nlg(n)) | O(nlg(n)) |  |

* + Minimum-swap strategy. Runs in O(n2)
  + Find the minimum in the array after the element and swap. For ex:
    - P=0: 4 3 2 1
    - P=1: 1 3 2 4
    - P=2: 1 2 3 4
  + Pseudocode:

for(int i = 0; i < n - 1; i ++) {

int min = i;

for(int j = i + 1; j < n; j ++) {

if(array[j] < array[min]) {

min = j;

}

}

if(i != min) {

swap(i, min);

}

}

* Heapsort
  + First buildHeap, then keep doing deleteMin till heap is over. O(nlg(n)) time.
* Mergesort
  + Recursively divide the list till there is only one element
  + Perform merging algorithm to merge 2 sorted lists

Priority Queues

6.1 – Model

* Priority queue is a data structure with two basic operations: insert and deleteMin

This is similar to a queue (enqueue = insert and deleteMin = dequeue)

6.2 – Simple implementations

* There are many data structures with their pros and cons
  + LinkedLists: Make insertions in the head and traverse list to delete the minimum. This means that insertions are O(1) and deleteMin is O(N).
  + Binary Search Tree: Gives O(lg N) average time for insertions and findMin. But tree could become unbalanced, slowing down the algorithm.

6.3 – Binary Heaps

* Most popular data structure for heaps.
* Binary heaps have 2 properties: structure property and heap order property, they need to be preserved after every operation
* Structure Property: heap must be a complete binary tree
* Heap order property: any parent is greater/smaller than or equal to all his children
* Heap operations:
  + Insert algorithm in a min heap in O(lg(n)) time: **percolating up** 
    - Create a hole in the next available position in the heap/array
    - If num > parent, place num into hole, exit
    - Else, place parent into hole and create hole at parent
  + deleMin algorithm: removing the minimum of the heap: **percolating down**
    - Remove the minimum in O(1), creating a hole
    - Let the last node of the list be N
    - If the hole’s children are smaller than N, put the smaller child in the hole
    - Repeat till N fits in
  + buildHeap algorithm: creating a heap from an array in O(n) time
    - use the existing array to create a heap of random order
    - Starting at element array.size / 2 (the last parent), percolateDown every node till the min

Tree

4.1 – Preliminaries

* Root
* Child
* Parent
* Edge
* Leaves
* Siblings
* Grandparent
* Grandchild
* Path
* Path length
* Depth
* Height
* Ancestor
* Descendant
* Traversals:
  + Preorder: action, explore left child, explore right child
  + Inorder: explore left child, action, explore right child
  + Postorder: explore left child, explore right child, action

4.2 – Binary Trees

* Full/perfect binary tree: every node except leaves have 2 children.
* Complete binary tree: every layer except last must be filled. If last layer incomplete, all leaves should be as left as possible.
* Expression trees
  + Read the infix expression character by character
  + If character, push it to stack as a one-node tree
  + If symbol, pop twice and make the popped 2 children of the symbol
* Inductive proofs
  + A perfect binary tree of height h has 2h + 1 – 1 nodes
    - A perfect binary tree of height h + 1 will have a root and two branches, each a subtree of height h.
    - Assuming that the theorem holds, each of them have 2h – 1 nodes.
    - Thus, the total number of nodes is 2\*(2h – 1) + 1, which is 2h+1 – 1.
  + A perfect binary tree with n nodes has height lg(n + 1) – 1
    - A perfect binary tree of n nodes will have two same branches, each having (n – 1)/2 nodes
    - Assuming the theorem holds, each of them should have a height of lg((n + 1)/2) – 1 = lg(n + 1) – 2.
    - This is true since the heights of a parent and child differ by 1.
  + A perfect binary tree of height h has 2h leaves
    - Two perfect binary trees of height h will have a total of 2h\*2 = 2h + 1 leaves.
    - The height of this tree will be one more than the original tree.
    - Thus a tree of height h will have 2h+1 nodes
  + Over half of all the nodes in a perfect binary tree are leaf nodes.
    - The total number of nodes in a perfect binary tree is 2h + 1 – 1
    - The total number of leaves in a perfect binary tree is 2h
    - Thus the ratio of leaves to nodes is
    - The range of the ratio is thus 0.5 < ratio <= 1.

4.3 – BST ADT

* boolean contains(Node root)
* Node findMin/findMax(Node root)
* Node insert(int x, Node root)
* Node remove(int x, Node root)

4.4 – AVL Trees

* Binary trees in which the depths of any subtree differ by at most 2
* Single rotation
* Double rotation

Lists, Stacks, Queues

3.1 – ADTs

* An ADT is a logical description of an object that has a set of functions. It is a mathematical abstraction. A data structure is the actual implementation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Access | Search | Insertion | Deletion |
| Array | O(1) | O(n) | O(n) | O(n) |
| Single LinkedList | O(n) | O(n) | O(n) | O(n) |
| Double LinkedList | O(n) | O(n) | O(n) | O(n) |
| Stack | O(1) | O(1) | O(1) | O(1) |
| Queue | O(1) | O(1) | O(1) | O(1) |

* Tower of Hanoi
  + Pseudocode: Runtime is 2n - 1

void tower(int from, int to, int via, int disks) {

if(disks > 0){

tower(from, via, to, disks - 1);

print("Move disk " + disk + " from " + from + " to " + to );

tower(via, to, from, disks - 1);

}

}

* Stacks
  + Symbol balancing algorithm
    - Make empty stack
    - If next symbol is closing
      * If stack is empty, report
      * Else, pop the stack. Report if element is not the matching opening symbol
    - Else, push next opening symbol
    - Once file reading ends, report if stack not empty.
  + Postfix expressions: for example 6 5 2 3 + 8 ∗ + 3 + ∗
    - If number, push to stack
    - Else, pop twice and carry out operation and push result
    - Once done, pop last element, which is the result
  + Infix to postfix
    - If character, print
    - Else
      * If close brack, keep popping till open bracket
      * If (, +, \*, keep popping till lower or equal priority/(
      * Push operator
    - Finally, pop till stack is empty
* Queues
  + Array implementation uses front, back, size variables
  + Linked

Hashing

* Good hash function
  + Hash value fully determined by input
  + Hash function uses all the input data
  + Hash function uniformly distributes data
  + Hash function gives very different values for similar input
  + Table size is prime
* Load factor L is m/n, the number of elements over table size.
* Separate chaining
  + Hash table maps keys to linked lists
  + Searching
    - Average: 1 + L/2
    - To optimize searching, maintain L around 1.0
    - If L crosses 1, then rehash
* Probing
  + Upon collision, another free spot is chosen using calculations
  + Load factor should be kept below 0.5, L < 0.5
  + Linear Probing:
    - If the next position attempted should be f, then f(i) = i where i is the number of attempts
  + Quadratic probing
    - Collision function: f(i) = i2
    - If the table size is prime and if at least half the table is empty, then a new element can always be inserted using quadratic probing
  + Double hashing
    - Collision function: f(i) = i\*hash2(x)
    - We must ensure that all the cells can be probed using hash2(x)
    - The table size must be prime
* Rehashing
  + Good to rehash then load factor becomes too high
  + How to rehash:
    - Create a new table of a size that is the first prime at least twice as large as the original table
    - Update hash function if needed
    - Scan the original table and for every element, rehash it to the new table