Data Structures & Algorithms

CS216 Final Review
Spring 2022

Graphs

- A graph is a set of vertices V with connective edges E. G = (V, E).
- Undirected vs directed graph. Indicates flow direction of edge.
- Adjacency matrices, lists, and hashmaps are possible storage structures
  - Matrix: 2-dimensional array; row and column for each vertex; boolean (or numerical) entries.
  - List: One list for each vertex storing its neighbors. Each node may store additional info, e.g. weight.
  - Hashmap: Possible neighbors used as keys with value perhaps storing edge weight.
- Fundamental types of (sub)graphs
  - Cycle: Non-trivial, non-intersecting path from vertex back to itself.
  - Connected (undirected): Path exists between any two given vertices
  - Tree (undirected): Connected acyclic graph
  - Strongly connected (directed): Path exists between any given start vertex and any given end vertex.
  - DAG: directed acyclic graph

Depth-first search (dfs)

- Depth-first search explores entire graph in depth-first manner, often marking the 'time' that each vertex is entered and left.
- Procedure explore(v)
  - visited(v) = true; previsit(v)
  - for each (v, u) in E
    - if not visited(u), explore(u)
  - postvisit(v)
- Procedure dfs()
  - for each v in V
    - if not visited(v), explore(v)
- Depth-first search can be used to identify the (strongly) connected components.
- Time complexity: O(|V| + |E|). Each vertex visited at least once and each edge traversed at most twice.

Graph decomposition

- Undirected graphs can be decomposed into their connected components.
- Directed graphs can be decomposed into a DAG of their strongly connected components; *i.e.* metagraphs.
- Procedure sinkComp(G):
  - G^R.dfs()
  - v = highest post vertex in G^R
  - G.explore(v)
  - return largest subgraph of G whose vertices are visited
- Procedure metaVertices(G):
  - $G_0 = G$
  - while G_0 non-empty
    - store comp = sinkComp(G_0)
    - G_0.trim(comp)

Breadth-first search (bfs)

- Breadth-first search explores regions of graph reachable from starting vertex s in breadth-first manner, marking each vertices' distance (# of edges) from s.
- Useful for finding shortest paths on unweighted graphs.
- Procedure bfs(s):
  - for all v in V, dist(v) = infinity; dist(s) = 0; Q = empty queue
  - Q.enqueue(s)
  - while Q is non-empty
    - u = Q.dequeue()
    - for all (u, v) in E
      - if dist(v) = infinity
        - Q.enqueue(v); dist(v) = dist(u) + 1
- Time complexity: O(|V| + |E|). Each vertex visited at least once and each edge traversed at most twice.

Dijkstra’s algorithm

- Hybrid search to find shortest paths on weighted graph from some starting vertex.
- Procedure dijkstra(s):
    - For all v in V, {dist(u) = infinity; prev(u) = null}; dist(s) = 0
    - pq = priority queue of all v in V (dist entries as keys)
    - while pq is not empty:
        - u = pq.deleteMin()
        - for all edges (u, v) in E
            - if dist(v) > dist(u) + length(u, v)
                - dist(v) = dist(u) + length(u, v); prev(v) = u
                - pq.decreaseKey(v)
- The idea: Emanate from vertex not yet emanated from which is nearest to start vertex. Update that vertex’s neighbor’s distances if appropriate.
- Time complexity: O((|V| + |E|) log V).

Minimum spanning trees (MST) and Kruskal's algorithm

- MST are connected subgraphs of a connected, undirected, weighted graph whose total weight is minimal. (*e.g.* minimum cost to maintain connectivity of network.)
- Often not unique; can be found via the following greedy algorithm.
- Procedure kruskal():
  - set aside all edges in graph
  - while graph is not connected
    - add edge with lowest weight that does not form a cycle
- Given candidate edge in loop above, how to know if cycle would form?
  - Does not form cycle if and only if edge links two distinct connected components.
  - Developed a model for disjoint sets, elements of which are vertices of connected components.
    - Checking a single edge occurs in $O(\log |V|)$ time.
    - Yields $O(|E| \log |V|)$ time complexity for entire algorithm.

Sorting algorithms

- Demonstrated in class that any sorting algorithm which operates via comparisons is at best O(n log n).
- Developed a number of sorting algorithms with best possible time complexity.
  - Merge sort: Split structure into two substructures of half-length. Repeat.
    - Remerge with O(log n) merge steps → O(n log n)
    - Memory complexity: O(n) needed for merging container. *Stable: original order of like-keys.*
  - Quick sort: Split structure into left- and right-hand side by comparing to random pivot. Repeat.
    - Best and average case: O(log n) pivot steps → O(n log n)
    - Worst case: O(n) pivot steps → $O(n^2)$
    - Average memory complexity: O(log n) on the call stack. *Unstable: not stable.*
  - Heap sort: max-Heapify array structure. Repeatedly removeMax and move to back.
    - Heap insert/removeMax each O(log n) → O(n log n)
    - Memory complexity: O(1) since done in-place. *Unstable: not stable.*

Pointed review & exercise

- **Exercise 1:** What is the time complexity of sinkComp(G)?
- **Exercise 2:** *Algorithms* 3.13 & 3.16
- **Exercise 3:** For an undirected graph, suppose one tried populating the distance/previous structures using dfs instead of bfs. What is the time complexity?
- **Exercise 4:** Suppose one replaces the priority queue in Dijkstra's algorithm with a naive array of (distance, vertex) tuples. What is the time complexity? Reflect.
- **Exercise 5:** *Algorithms* 4.13 & 4.14
- **Exercise 6:** Read about Prim's algorithm in *Algorithms* 5.1.5.
  - Argue that Prim's algorithm returns a MST.
  - Why may Prim's be preferable to Kruskal's algorithm for graph objects with a previous structure?
- **Exercise 7:** Think about the implementation of mergeSort, quickSort, and heapSort on arrays of Comparable in Java. Code if time permits.