Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme

Programming, Data Structures and Algorithms using Python

Summary of weeks 4 to 6

Content

- Graphs
- Graph representation
- Breadth First Search(BFS)
- Depth First Search(DFS)
- Applications of BFS & DFS
- Directed Acyclic Graphs(DAGs)
- Shortest Paths in Weighted Graphs
- Dijkstra’s algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall algorithm
- Trees
- Spanning trees
- Prim’s algorithm
- Kruskal’s algorithm
- Efficient data structures

Graphs

- A graph represents relationships between entities
  - Entities are vertices/nodes
  - Relationships are edges
- A graph can be directed or undirected
  - A is a parent of B – directed
  - A is a friend of B – undirected
- Paths are sequence of connected edges
- Reachability: is there a path from node u to node v?

Graph representation

- G = (V, E)
  - $|V| = n$
  - $|E| = m$
  - If G is connected, m can vary from n – 1 to n(n – 1) / 2
- Adjacency matrix
  - n×n matrix, AMat[i, j] = 1 iff (i, j) ∈ E
- Adjacency list
  - Dictionary of lists
  - For each vertex i, AList[i] is the list of neighbors of i

Breadth First Search(BFS)

- Breadth first search is a systematic strategy to explore a graph, level by level
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m+n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges

Depth First Search(DFS)

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbors
- Complexity is $O(n^2)$ using adjacency matrix, $O(m+n)$ using adjacency list
- Useful features can be found by recording the order in which DFS visits vertices

Applications of BFS & DFS

- Paths discovered by BFS are shortest paths in terms of number of edges
- BFS and DFS can be used to identify connected components in an undirected graph
- BFS and DFS identify an underlying tree
- Use of DFS numbering
  - Strongly connected components
  - Articulation points(cut vertices) and bridges(cut edges)

Directed Acyclic Graphs(DAGs)

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sorting
  - It gives a feasible schedule that represents dependencies
  - Complexity is $O(n^2)$ using adjacency matrix, $O(m+n)$ using adjacency list
- Longest paths
  - Directed acyclic graphs are a natural way to represent dependencies
  - Complexity is $O(m+n)$

Shortest Paths in Weighted Graphs

- Single source shortest paths (Dijkstra's algorithm)
  - Find shortest paths from a fixed vertex to every other vertex
- All pairs shortest paths (Floyd-Warshall algorithm)
  - Find shortest paths between every pair of vertices i and j
- Negative edge weights and Negative cycles
  - If a graph has a negative cycle, shortest paths are not defined
  - Without negative cycles, we can compute shortest paths even if some weights are negative (Bellman-Ford Algorithm)

Dijkstra's algorithm

- Dijkstra's algorithm computes single source shortest paths
- Uses a greedy strategy to identify vertices to visit
  - Next vertex to visit is based on shortest distance computed so far
  - Correctness requires edge weights to be non-negative
- Complexity is $O(n^2)$
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
- Complexity can be improved to $O((m+n)\log n)$ by using efficient min-heap data structure

Bellman-Ford Algorithm

- Bellman-Ford algorithm computes single source shortest paths with negative weights
- Dijkstra's algorithm assumes non-negative edge weights
  - Final distance is frozen each time a vertex "burns"
  - Should not encounter a shorter route discovered later
  - Without negative cycles, every shortest route is a path
  - Every prefix of a shortest path is also a shortest path
- Iteratively find shortest paths of length 1, 2, …, n – 1
- Update distance to each vertex with every iteration
- Complexity is $O(n^3)$ using adjacency matrix, $O(mn)$ using adjacency list

Floyd-Warshall algorithm

- Floyd-Warshall algorithm computes all pairs shortest paths
- Complexity using simple nested loop implementation is $O(n^3)$

Trees

- A tree is a connected acyclic graph
- A tree with n vertices has exactly $n - 1$ edges
- Adding an edge to a tree creates a cycle
- Deleting an edge from a tree splits the tree
- In a tree, every pair of vertices is connected by a unique path

Spanning trees

- Retain a minimal set of edges so that graph remains connected
- A graph can have multiple spanning trees
- Minimum Cost Spanning Tree (MCST) – among the different spanning trees, choose one with minimum cost
- A graph can have multiple MCSTs, but the cost will always be unique
- Building a MCST
  - Prim’s algorithm
  - Kruskal’s algorithm

Prim's algorithm

- Start with a smallest weight edge overall
- Incrementally grow the MCST from any vertex
- Extend the current tree by adding the smallest edge from some vertex in the tree to a vertex not yet in the tree
- Implementation is similar to Dijkstra's algorithm
- Complexity is $O(n^2)$
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
- Complexity can be improved to $O((m+n)\log n)$ by using efficient min-heap data structure

Kruskal's algorithm

- Start with n components, each an isolated vertex
- Process edges in ascending order of cost
- Include edge if it does not create a cycle
  - Challenge is to keep track of connected components
  - Maintain a dictionary to record component of each vertex
  - Initially each vertex is an isolated component
  - When we add an edge (u, v), merge the components of u and v
- Complexity is $O(n^2)$ due to naive handling of components, can be improved to $O(m \log n)$ by using efficient union-find data structure

Efficient data structures

1. Union-Find
   - Across m operations, amortized complexity of each Union() operation is $\log m$
   - With clever updates to the tree, Find() has amortized complexity very close to $O(1)$
2. Priority queues
   - insert() operation is $O(\sqrt{n})$
   - delete_max() operation is $O(\sqrt{n})$
   - Processing n items is $O(n\sqrt{n})$

Efficient data structures

3. Heaps

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children (max-heap)
- No “holes” allowed and cannot leave a level incomplete
- insert() operation is $O(\log n)$
- delete_max() / delete_min() operation is $O(\log n)$
- heapify() builds a heap in $O(n)$
- Heaps can also be used to sort a list in place in $O(n \log n)$

Efficient data structures

4. Search trees
   - For each node with value v, all values in the left subtree are < v
   - For each node with value v, all values in the right subtree are > v
   - No duplicate values
   - Each node has three fields, value, left, right
   - Traversals: In-order, Pre-order, Post-order
   - Worst case: An unbalanced tree with n nodes may have height $O(n)$
   - find(), insert() and delete() all walk down a single path

Efficient data structures

5. Balanced search trees
   - Left and right subtrees should be “equal”
   - Two possible measures: size and height
   - Height balanced trees: height of left child and height of right child differ by at most 1 (AVL trees)
   - Using rotations, we can maintain height balance
   - AVL trees with n nodes will have height $O(\log n)$
   - find(), insert() and delete() all walk down a single path, take only $O(\log n)$