CSCE 689: Special Topics in Modern Algorithms for Data Science Fall 2023

Lecture 21 — October 20, 2023

Prof. Samson Zhou *Scribe: Zhitong Chen*

1 The Semi-streaming Model for Graphs

- **Graph Definition**:
  - Input graph: $G = (V, E)$
  - Vertices: V with a total count of n
  - Edges: E with a total count of m
- **Insertion-Only Stream of Edges**:
  - Edges arrive sequentially in a data stream and cannot be removed (insertion-only model).
- **Storage Constraints**:
  - Limited to using $n \cdot \text{polylog}(n)$ space.
  - This space is adequate to do interesting things such as storing a matching or a spanning tree, without storing the entire graph as the maximum number of edges m can be $O(n^2)$.
- **Key Takeaway**:
  - Although it's possible to store a matching, the entire graph can't be stored due to potentially large number of edges.

2 Maximum Matching

- **Greedy Algorithm for Maximum Matching**:
  - Recall that the greedy algorithm provides a 2-approximation to the maximum matching because it is a maximal matching.
  - Utilizes $O(n)$ space.
- **Open Problem**:
  - **Question**: Is it feasible to attain a C-approximation to the maximum (cardinality) matching using $n \cdot \text{polylog}\, n$ space when $C < 2$?
    - **Context**: This problem challenges the existing boundaries of space-efficient algorithms in graph theory, specifically for the problem of finding large matchings.
    - **Significance**: A positive answer to this question would imply that we can achieve better solutions for maximum matching in graphs, even with severe space constraints. This has broad implications for big data and streaming algorithms, where space is at a premium.

    - **Challenge**: The primary difficulty lies in significantly improving the approximation ratio while still operating within a sublinear space regime.

- **Challenge**: In conclusion, the journey towards finding a C-approximation for the maximum matching in a graph, with $C < 2$ and under the stringent space constraint of $n \cdot \text{polylog}(n)$, is a path riddled with challenges and opportunities. This problem encapsulates the quintessential struggle in computer science and graph theory: achieving efficiency and accuracy, all while operating within tight resource bounds. The resolution of this problem has the potential to not only advance our theoretical understanding but also to catalyze improvements in practical applications spanning various domains.

3 Connectivity in Graphs

- Connected graph: There exists a path between i and j for any pair $i, j \subseteq V = [n]$ of vertices
- Goal: Given a graph G, determine whether G is a connected graph
- Intuition:
  - How to find a spanning tree in the offline setting?
  - Minimum spanning tree algorithms (Kruskal, Prim)
    - Kruskal: Greedily add minimum weight edge to spanning forest
    - Prim: Greedily grow minimum spanning tree
    - The goal is to greedily add edges to form a minimum spanning forest to determine the connectivity of the graph.
- Algorithm:
  1. Initialize $F = \emptyset$.
  2. For each edge $e = (u, v)$:
     1. If adding (u, v) to F does not create a cycle, update F: $F \leftarrow F \cup \{(u, v)\}$.
     2. If $|F| = n - 1$, return *GRAPH IS CONNECTED.*
  3. Return *GRAPH IS NOT CONNECTED.*

  - Algorithm can keep at most n edges, so the total space usage is $O(n)$ words of space

4 Bipartiteness

- Bipartite graph: Graph can be partitioned into two disjoint sets L and R so that every edge is between a vertex in L and a vertex in R
- Goal: Given a graph G, determine whether G is a bipartite graph

An application is Circuit Design: In electrical engineering and VLSI design, bipartiteness testing is used to determine if a circuit can be divided into two complementary parts for optimal partitioning.

- What is a necessary and sufficient condition for bipartiteness?
  - A graph is bipartite if and only if it can be colored using two colors (a coloring of a graph is an assignment of colors to vertices such that no two vertices share the same color)
  - A graph is bipartite if and only if it has no odd cycles
- How to perform bipartiteness testing in the central setting?
  - Start at arbitrary vertex, run BFS, and assign alternating levels to different side until there is a contradiction

Bipartiteness in the Streaming Model

- Bipartiteness is a monotone property, i.e., additional edges to a graph that is not bipartite will result in a graph that is not bipartite

Algorithm

1. Initialize $F = \emptyset$.
2. For each edge $e = (u, v)$:
   1. If adding (u, v) to F does not create a cycle, update F: $F \leftarrow F \cup \{(u, v)\}$.
   2. If adding (u, v) to F creates an odd cycle, return *GRAPH IS NOT BIPARTITE*.
3. Return *GRAPH IS BIPARTITE*.

Algorithm Properties

- **Tree Maintenance**: The algorithm maintains a tree structure throughout its execution because it deliberately avoids adding any edges that would create cycles.
- **Space Usage**: Since a tree with n vertices has at most $n - 1$ edges, the algorithm keeps at most n edges in total. Therefore, the total space usage of the algorithm is $O(n)$ words of space.