Streaming Graph Algorithms
Finding the Majority Element

Consider the following problem:

- Given an unsorted list of arbitrary elements, find the element that makes up more than half the elements in the list.

Example

Given:

\[-1, 8, \lambda, 8, 8, a, \ast, 8, \ast, 94, 8, 8, 8, 8, \emptyset\]

...return 8.
Naïve Solution

**MajorityElement**\((A \leftarrow [a_1, a_2, \ldots, a_n])\)

**Input:** A finite list of \(n\) elements, \(A\)

**Output:** The majority element in the list

1. for \(i \leftarrow 1\) to \(n\) do
2.     let count \(\leftarrow 0\)
3.     for \(j \leftarrow 1\) to \(n\) do
4.         if \(a_j = a_i\) then
5.             count \(\leftarrow\) count + 1
6.         if count > \(\lceil n/2 \rceil\) then
7.             return \(a_i\)
8.     return “None”

□ This algorithm runs in \(O(n^2)\) time and \(O(1)\) space.
**Divide and Conquer Solution**

```
MAJORITYELEMENT(A ← [a₁, a₂, ..., aₙ])

Input: A finite list of \( n \) elements, \( A \)
Output: The majority element and the number of times it occurs

1: if \( n = 1 \) then
2: return \((a₁, 1)\)
3: else
4: let \( \text{mid} \leftarrow \lfloor n/2 \rfloor \)
5: let \((a_l, c_l) \leftarrow \text{MAJORITYELEMENT}([a₁, ..., a_{\text{mid}}])\)
6: let \((a_r, c_r) \leftarrow \text{MAJORITYELEMENT}([a_{\text{mid}+1}, ..., aₙ])\)
7: if \( a_l = \text{"None"} \) and \( c_r = \text{"None"} \) then
8: return \("\text{None"}, -1\)
9: else if \( a_l = a_r \) then
10: return \((a_l, c_l + c_r)\)

...
**Divide and Conquer Solution**

\[ \text{MajorityElement}(A \leftarrow [a_1, a_2, \ldots, a_n]) \]

\[ \begin{align*}
11: & \quad \text{else} \\
12: & \quad c_l \leftarrow c_l + \text{COUNT}(a_l, [a_{\text{mid}+1}, \ldots, a_n]) \\
13: & \quad c_r \leftarrow c_r + \text{COUNT}(a_r, [a_1, \ldots, a_{\text{mid}}]) \\
14: & \quad \text{if } a_l \neq \text{"None" and } c_l > \lfloor n/2 \rfloor \text{ then} \\
15: & \quad \quad \text{return } (a_l, c_l) \\
16: & \quad \text{else if } a_r \neq \text{"None" and } c_r > \lfloor n/2 \rfloor \text{ then} \\
17: & \quad \quad \text{return } (a_r, c_r) \\
18: & \quad \text{else} \\
19: & \quad \quad \text{return } (\text{"None"}, -1)
\end{align*} \]

- This algorithm runs in \(O(n \log n)\) time and \(O(n)\) space.
**MajorityElement**($A \leftarrow [a_1, a_2, \ldots, a_n]$)

**Input:** A finite list of $n$ elements, $A$

**Output:** The majority element in the list

1: let $T \leftarrow []$

2: for $i \leftarrow 1$ to $n$ do

3: if $a_i \notin T$ then

4: $T[a_i] \leftarrow 1$

5: else

6: $T[a_i] \leftarrow T[a_i] + 1$

7: if $T[a_i] > \lfloor n/2 \rfloor$ then

8: return $a_i$

9: return “None”

- This algorithm runs in $O(n)$ time and $O(n)$ space.
Boyer-Moore Majority Vote Algorithm

\[ \text{MAJORITY\_VOTE}(A \leftarrow [a_1, a_2, \ldots, a_n]) \]

**Input:** A finite list of \( n \) elements, \( A \)

**Output:** The majority element in the list, assuming one exists

1: \text{let } x \leftarrow a_1
2: \text{let } votes \leftarrow 1
3: \text{for } i \leftarrow 2 \text{ to } n \text{ do}
4: \quad \text{if } votes = 0 \text{ then}
5: \quad \quad x \leftarrow a_i
6: \quad \quad votes \leftarrow 1
7: \quad \text{else if } a_i = x \text{ then}
8: \quad \quad votes \leftarrow count + 1
9: \quad \text{else}
10: \quad \quad votes \leftarrow count - 1
11: \text{return } x
Boyer-Moore Majority Vote Algorithm

Example

Let \( A = [-1, 8, \lambda, 8, 8, a, *, 8, *, 94, 8, 8, 8, 8, \emptyset] \).

<table>
<thead>
<tr>
<th>Element</th>
<th>Candidate</th>
<th>Votes</th>
<th>(continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( \lambda )</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>( \lambda )</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>*</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

This algorithm runs in \( O(n) \) time and \( O(1) \) space.
Boyer-Moore Majority Vote Algorithm

- In a sense, this algorithm is “complete”, but not “sound”.
  - If there is a majority element, this algorithm will find it.
  - If this algorithm finds an element, that element is not necessarily the majority element.

**MajorityElement**

\[
\text{Input:} \ A \leftarrow [a_1, a_2, \ldots, a_n] \\
\text{Output:} \ \text{The majority element in the list}
\]

1: let \( x \leftarrow \text{MajorityVote}(A) \)
2: if \( \text{Count}(x, A) > \lceil n/2 \rceil \) then
3: return \( x \)
4: else
5: return “None”

- This algorithm, too, runs in \( O(n) \) time and \( O(1) \) space.
Boyer-Moore Majority Vote Algorithm

- The Majority Vote algorithm loses context.
- The algorithm doesn’t keep complete occurrence information.
- The algorithm is not capable of backtracking.
- The algorithm discards data it has already processed.
- The algorithm is capable of operating in constant space.

- Karp, Shenker, and Papadimitriou generalized the Majority Vote algorithm to find all elements with frequency $\geq \theta$ in $O(1/\theta)$ space, although it requires two passes.
Streaming Algorithms

- A **streaming algorithm** processes an indefinite stream of elements, \( S = \langle s_1, s_2, \ldots, s_n, \ldots \rangle \).

- Since the goal is to avoid storing the dataset in memory, the algorithm must operate under severe memory constraints.

- Since the algorithm cannot know the future, it must process data as it arrives, in the order it arrives.

- Streaming algorithms are ideal for processing massive datasets or responding to real-time events.
The Seven Bridges of Königsberg

- A graph $G = (V, E)$ consists of a set of vertices, $V$, and a set of edges, $E$. By convention, $|V| = n$ and $|E| = m$.
- Each edge $e = (u, v)$ connects two vertices, $u$ and $v$.
- Graphs are used to represent relationships between entities.
Graphs

Graphs can be used to represent:

- Bridges and landmasses
- Cities and roads
- Computers on the Internet
- Flights between airports
- Diseased populations
- States and transitions
- Friends on social networks
- ... 

Given a graph, we might be interested in:

- Eulerian paths
- Hamiltonian paths
- Spanning trees
- Network flow
- Densest subgraphs
- Shortest paths
- Largest cliques
- ...
Finding A Minimum Spanning Tree

Consider the following problem:

- Given a connected, undirected, weighted graph, find the tree of minimum total weight that connects all vertices.

Example

Given:

```
<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... return \{(u, v, 3), (u, x, 2), (w, x, 2)\}. ```
Kruskal’s Algorithm

\textbf{MinimumSpanningTree}(G \leftarrow (V, E))

\textbf{Input:} A connected, undirected, weighted graph \( G \)
\textbf{Output:} A minimum spanning tree of \( G \)

1. \textbf{let} \( T \) be an empty tree
2. \textbf{while} \( E \neq \emptyset \) do
3. \textbf{let} \( e \leftarrow \text{MinWeightEdge}(E) \)
4. \( E \leftarrow E \setminus \{e\} \)
5. \textbf{if} \( T \cup \{e\} \) does not contain a cycle \textbf{then}
6. \( T \leftarrow T \cup \{e\} \)
7. \textbf{return} \( T \)

- This algorithm runs in \( O(m \log m) \) time and \( O(n + m) \) space.
- Selecting the minimum weight edge requires that the algorithm access the edge set multiple times.
Graph Streaming Algorithms

- Often, a semi-streaming model must be used.
  - An algorithm is allowed $O(n \text{ polylog } n)$ space.
  - The time constraint is similarly relaxed.

McGregor distinguishes three types of graph streams:

- **Insert-Only**
  An unordered stream of edges,
  $S = \langle e_1, e_2, \ldots, e_m \rangle$

- **Graph Sketch**
  A stream of edge additions and removals,
  $S = \langle a_1, a_2, \ldots, a_k \rangle$, where
  $a_i = (e_i, \delta_i \in \{1, -1\})$.

- **Sliding Window**
  An indefinite stream where at time $t$, only the last $w$ edges are considered,
  $S = \langle \ldots, e_{t-w+1}, \ldots, e_{t-1}, e_t, \ldots \rangle$. 
Finding a Minimum Spanning Tree

The Cycle Property

For any cycle $C$ in a graph $G$, if $e$ is the strictly maximum weight edge in $C$, then $e$ is not part of any MST of $G$.

\[
\text{MINIMUM SPANNING TREE}(G \leftarrow \langle e_1, e_2, \ldots, e_m \rangle)
\]

**Input:** A stream of edges defining a connected, undirected graph $G$

**Output:** A minimum spanning tree of $G$

1: let $T$ be an empty tree
2: for $e \in G$ do
3: \hspace{1em} $T \leftarrow T \cup \{e\}$
4: \hspace{1em} if $T$ contains a cycle, $C$ then
5: \hspace{2em} $T \leftarrow T \setminus \{\text{MAX WEIGHT EDGE}(C)\}$
6: return $T$

- This algorithm only requires one pass through the edge set.
## Selected Graph Streaming Algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
<th>Insert-Only</th>
<th>Graph Sketch</th>
<th>Sliding Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Two-coloring</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Min. Spanning Tree</td>
<td>✓</td>
<td>(1 + $\epsilon$)-approx. or $O(\log n)$ passes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. Matching</td>
<td>2-approx.</td>
<td>Multiple passes</td>
<td></td>
<td>(3 + $\epsilon$)-approx.</td>
</tr>
</tbody>
</table>

*McGregor, Feigenbaum et al.*
Future Work

- Most research has been done with undirected graphs, yet many naturally occurring graphs are directed.

- Constructing a graph generally does not produce edges in a truly random order; can we take advantage of that fact?
Questions?
References


References


