Representing and Exploring Graphs
(Depth First Search and Breadth First Search)

Benjamin Sach

\[ G \text{ is a Graph,} \]

\[ V \text{ is the set of vertices} \]
\[ |V| \text{ is the number of vertices} \]

\[ E \text{ is the set of edges} \]
\[ |E| \text{ is the number of edges} \]

\[ v \in V \text{ is a vertex} \]
\[ (u, v) \in E \text{ is an edge from } u \text{ to } v \]

This lecture focuses on exploring \textbf{undirected} and \textbf{unweighted} graphs
\textit{though everything discussed will work for directed, unweighted graphs too}.

Graph representations

\[ E \text{ is the set of edges} \]
\[ |E| \text{ is the number of edges} \]

Adjacency Matrix

\[ \begin{array}{cccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

We will in general assume that the vertices are numbered \(1, 2, 3 \ldots |V|\).
**Graph representations**

- **Adjacency Matrix**
  - We will in general assume that the vertices are numbered \(1, 2, 3 \ldots |V|\)
  - Both representations are symmetric because the graphs are undirected

- **Adjacency List**
  - We will in general assume that the vertices are numbered \(1, 2, 3 \ldots |V|\)
  - Adjacency List (using linked lists)

**Basic operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>(\Theta(</td>
<td>V</td>
<td>^2))</td>
</tr>
<tr>
<td>Is there an edge from vertex (u) to (v)?</td>
<td>(O(1)) time</td>
<td>(O(\text{deg}(u))) time</td>
<td>(O(1)) time</td>
</tr>
<tr>
<td>List all the edges leaving (u \in V)</td>
<td>(O(V)) time</td>
<td>(O(\text{deg}(u))) time</td>
<td>(O(\text{deg}(u))) time</td>
</tr>
</tbody>
</table>

\(\text{deg}(u)\) - the degree, is the number of edges leaving vertex \(u\)

**Example**

- Adjacency Matrix
  - \(V\) is the set of vertices
  - \(|V|\) is the number of vertices
  - \(E\) is the set of edges
  - \(|E|\) is the number of edges

- Adjacency List
  - Adjacency List (using linked lists)
  - Adjacency List (using hash tables)
Basic operations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List (using linked lists)</th>
<th>Adjacency List (using hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>Is there an edge from vertex $u$ to $v$?</td>
<td>$O(1)$ time</td>
<td>$O(\deg(u))$ time</td>
<td>$O(1)$ time</td>
</tr>
<tr>
<td>List all the edges leaving $u \in V$</td>
<td>$O(V)$ time</td>
<td>$O(\deg(u))$ time</td>
<td>$O(\deg(u))$ time</td>
</tr>
</tbody>
</table>

Basic hash tables give expected time complexities (constant time ‘on average’)
- no worst case guarantees against collisions

If you’re interested in hashing with provable guarantees, come to COMS31900 (Advanced Algorithms)

Fortunately, few algorithms need to ask “is there an edge?”
Normally, “tell me all the edges” is fine

why don’t we always use these?

Intersection Graphs

each circle is a vertex

the intersections of these circles correspond to the graph below

For this course we’ll stick to Adjacency Matrices and Lists (using linked lists)
(for today just Adjacency Lists)

Another reason to do so is that you may not have control over how your graph is stored.

it may not even be stored as a graph...

Fortunately in many cases, you can pretend your graph is stored as one of the above.
Intersection Graphs

each intersection is an edge

the intersections of these circles correspond to the graph below

we can pretend we have an Adjacency Matrix without building one

if we only store the circles,

the intersections of these circles correspond to the graph below

Other intersection graphs...

both of these collections also correspond to the example graph

there is an edge if two intervals intersect

we can pretend we have an Adjacency Matrix without building one

these are 1D intervals on a line spread out for visual clarity

Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

a node is a state that the game can be in

there is an edge if you can get from one state to another in a single move

Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

a node is a state that the game can be in

storing this uses much less space than storing this graph

we can pretend we have an Adjacency List without building one

there is an edge if you can get from one state to another in a single move
Exploring a graph

**Goal:** visit every vertex in a connected graph efficiently

We are going to use an adjacency list so the graph is really stored like this:

The **bag** (secretly a data structure)

- We can put a vertex in the bag
- We can take a vertex from the bag
- We don't care how it works (for now)

(the bag doesn't forget or change things)

**TRAVERSE**

- put $s$ into the bag
- while the bag is not empty
- take $u$ from the bag
- if $u$ unmarked
- mark $u$
- for every edge $(u,v)$
- put $v$ into the bag

**Questions:**

1. Does **TRAVERSE** always stop?
2. Does it always visit every vertex?
3. How fast is it? (in the worst case)

Some time later...

- we may put a vertex in the bag many times but each edge is only processed twice (once in each direction)
- number of 'puts' $\leq 2|E|$
Exploring a graph

TRAVERE(s)
put s into the bag
while the bag is not empty
take u from the bag
if u unmarked
mark u
for every edge (u, v)
put v into the bag

Questions:
1. Does TRAVERSE always stop? Yes
2. Does it always visit every vertex? Yes
3. How fast is it? (in the worst case) $O(|E|)$

Time complexity:
- $O(1)$ time every time we take from the bag
- $O(1)$ time every time we put into the bag
- What is the total number of bag operations?
we do at most $2|E|$ put operations
so we do at most $2|E|$ take operations
The overall time complexity is $O(|E|)$

What if we had used an adjacency matrix?

Finding all edges leaving $u$ would have taken $O(|V|)$ time
overall, the time complexity would have been $O(|V|^2)$ instead of $O(|E|)$.
How should we implement the bag?

Queue

Operations take $O(1)$ time

 Traverse visits every vertex in a connected graph in $O(|E|)$ time

but in different orders with different types of bag

(and it works for directed graphs too)

with a Stack the algorithm is called

Depth First Search

Applications
- Shortest paths in unweighted graphs
- Max-Flow
- Testing whether a graph is bipartite

with a Stack the algorithm is called

Depth First Search

Applications
- Finding (strongly) connected components
- Topologically sorting a Directed Acyclic Graph
- Testing for planarity (and it works for directed graphs too)

Goal: find the distance from the source $s$ to every other vertex

Breadth-First Search (BFS) is Traverse using a queue as the bag

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction

Base Case: BFS trivially marks all vertices at distance 0 before marking any vertex at distance > 0.

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i - 1)$ before marking any vertex at distance > $(i - 1)$.

When BFS marks a vertex at distance $(i - 1)$, it puts all its neighbours in the queue.

Immediately after marking all vertices at distance $(i - 1)$:
- every vertex at distance $i$ is in the queue.
- no vertex at distance $> i$ has been put in the queue.
  - because we haven’t marked any of its neighbours
    - (they are all at distance $> (i - 1)$)

Therefore, all vertices at distance $i$ will be marked before any at distance $> i$.
Shortest paths in unweighted graphs

**Goal:** find the distance from the source to every other vertex

**Breadth-First Search (BFS) is TRAVERSE using a queue as the bag**

**Lemma:** In BFS, the vertices are marked in distance order (smallest first)

With a little bit of bookkeeping, we can use this to find shortest paths in unweighted graphs in $O(|V| + |E|)$ time.

This also works for directed, unweighted graphs.

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE to track the distances from $s$

**BFS(s)**

for all $v$, set $\text{dist}(v) = \infty$ (NEW!), set $\text{dist}(s) = 0$ (NEW!), put $s$ into the queue while the queue is not empty take $u$ from the queue if $u$ unmarked mark $u$ for every edge $(u,v)$ put $v$ into the queue if $\text{dist}(v) = \infty$ (NEW!), $\text{dist}(v) = \text{dist}(u) + 1$ (NEW!)

$\text{dist}(v)$ gives the distance between $s$ and $v$

**Correctness sketch:** (using the Lemma)

For each $v$, $\text{dist}(v)$ only changes once, when it is first discovered and inserted into the queue.

Let $u$ be the vertex which first discovered $v$

Assume that there is a path of length less than $\text{dist}(u) + 1$

Let $x$ be the previous vertex on this path

$x$ has distance $< \text{dist}(u)$ so should have been marked before $u$ and discovered $v$ first

**Conclusion**

**TRAVERSE visits every vertex in a connected graph in $O(|E|)$ time**

with a Stack the algorithm is called **Depth First Search (DFS)**

**Applications**

Max-Flow
Testing whether a graph is bipartite
Finding (strongly) connected components
Topologically sorting a Directed Acyclic Graph
Testing for planarity

**Shortest paths in unweighted graphs**

take $O(|V| + |E|)$ time using BFS (works for directed graphs too)

**Applications**

Finding (strongly) connected components
Topologically sorting a Directed Acyclic Graph
Testing for planarity

**Question**

What does TRAVERSE do on an unconnected graph?