Graph Search

Lecture 15
Thursday, October 19, 2017

Why Graphs?

- Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links), and many problems that don’t even look like graph problems.
- Fundamental objects in Computer Science, Optimization, Combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep mathematics

Graph

Definition
An undirected (simple) graph $G = (V, E)$ is a 2-tuple:
- $V$ is a set of vertices (also referred to as nodes/points)
- $E$ is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.

Example
In figure, $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$. 
Example: Modeling Problems as Search

State Space Search
Many search problems can be modeled as search on a graph. The trick is figuring out what the vertices and edges are.

Missionaries and Cannibals
- Three missionaries, three cannibals, one boat, one river
- Boat carries two people, must have at least one person
- Must all get across
- At no time can cannibals outnumber missionaries

How is this a graph search problem? What are the vertices? What are the edges?

Notation and Convention

Notation
An edge in an undirected graphs is an unordered pair of nodes and hence it is a set. Conventionally we use \((u, v)\) for \(\{u, v\}\) when it is clear from the context that the graph is undirected.

- \(u\) and \(v\) are the end points of an edge \(\{u, v\}\)
- Multi-graphs allow
  - loops which are edges with the same node appearing as both end points
  - multi-edges: different edges between same pairs of nodes
- In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.

Example: Missionaries and Cannibals Graph

Graph Representation I

Adjacency Matrix
Represent \(G = (V, E)\) with \(n\) vertices and \(m\) edges using a \(n \times n\) adjacency matrix \(A\) where

- \(A[i, j] = A[j, i] = 1\) if \(\{i, j\} \in E\) and \(A[i, j] = A[j, i] = 0\) if \(\{i, j\} \not\in E\).
- Advantage: can check if \(\{i, j\} \in E\) in \(O(1)\) time
- Disadvantage: needs \(\Omega(n^2)\) space even when \(m \ll n^2\)
Graph Representation II

Adjacency Lists

Represent \( G = (V, E) \) with \( n \) vertices and \( m \) edges using adjacency lists:

- For each \( u \in V \), \( \text{Adj}(u) = \{ v \mid \{ u, v \} \in E \} \), that is neighbors of \( u \). Sometimes \( \text{Adj}(u) \) is the list of edges incident to \( u \).
- Advantage: space is \( O(m + n) \)
- Disadvantage: cannot “easily” determine in \( O(1) \) time whether \( \{i, j\} \in E \)
  - By sorting each list, one can achieve \( O(\log n) \) time
  - By hashing “appropriately”, one can achieve \( O(1) \) time

Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

A Concrete Representation

- Assume vertices are numbered arbitrarily as \( \{1, 2, \ldots, n\} \).
- Edges are numbered arbitrarily as \( \{1, 2, \ldots, m\} \).
- Edges stored in an array/list of size \( m \). \( E[j] \) is \( j \)th edge with info on end points which are integers in range 1 to \( n \).
- Array \( \text{Adj} \) of size \( n \) for adjacency lists. \( \text{Adj}[i] \) points to adjacency list of vertex \( i \). \( \text{Adj}[i] \) is a list of edge indices in range 1 to \( m \).

A Concrete Representation: Advantages

- Edges are explicitly represented/numbered. Scanning/processing all edges easy to do.
- Representation easily supports multigraphs including self-loops.
- Explicit numbering of vertices and edges allows use of arrays: \( O(1) \)-time operations are easy to understand.
- Can also implement via pointer based lists for certain dynamic graph settings.
Part II
Connectivity

Connectivity

Given a graph \( G = (V, E) \):

1. **path**: sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( \{v_i, v_{i+1}\} \in E \) for \( 1 \leq i \leq k - 1 \). The length of the path is \( k - 1 \) (the number of edges in the path) and the path is from \( v_1 \) to \( v_k \). **Note**: a single vertex \( u \) is a path of length 0.

2. **cycle**: sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( \{v_i, v_{i+1}\} \in E \) for \( 1 \leq i \leq k - 1 \) and \( \{v_1, v_k\} \in E \). Single vertex not a cycle according to this definition. **Caveat**: Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term tour.

3. **A vertex** \( u \) is connected to \( v \) if there is a path from \( u \) to \( v \).

4. The **connected component** of \( u \), \( \text{con}(u) \), is the set of all vertices connected to \( u \). Is \( u \in \text{con}(u) \)?

Connectivity contd

Define a relation \( C \) on \( V \times V \) as \( uCv \) if \( u \) is connected to \( v \)

- In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.

- Graph is **connected** if only one connected component.

Connectivity Problems

**Algorithmic Problems**

- Given graph \( G \) and nodes \( u \) and \( v \), is \( u \) connected to \( v \)?
- Given \( G \) and node \( u \), find all nodes that are connected to \( u \).
- Find all connected components of \( G \).

Can be accomplished in \( O(m + n) \) time using **BFS** or **DFS**. **BFS** and **DFS** are refinements of a basic search procedure which is good to understand on its own.
Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

**Explore**$(G, u)$:

- $Visited[1..n] \leftarrow$ FALSE
- // ToExplore, S: Lists
- Add $u$ to ToExplore and to $S$
- $Visited[u] \leftarrow$ TRUE
- while (ToExplore is non-empty) do
- Remove node $x$ from ToExplore
- for each edge $xy$ in Adj$(x)$ do
- if ($Visited[y]$ = FALSE)
- $Visited[y] \leftarrow$ TRUE
- Add $y$ to ToExplore
- Add $y$ to $S$
- Output $S$

Example

![Graph Example]

Properties of Basic Search

**Proposition**

$Explore(G, u)$ terminates with $S = con(u)$.

**Proof Sketch.**

- Once $Visited[i]$ is set to TRUE it never changes. Hence a node is added only once to ToExplore. Thus algorithm terminates in at most $n$ iterations of while loop.
- By induction on iterations, can show $v \in S \Rightarrow v \in con(u)$
- Since each node $v \in S$ was in ToExplore and was explored, no edges in $G$ leave $S$. Hence no node in $V - S$ is in con$(u)$.
- Thus $S = con(u)$ at termination.

Properties of Basic Search

**Proposition**

$Explore(G, u)$ terminates in $O(m + n)$ time.

Proof: easy exercise

BFS and DFS are special case of BasicSearch.

- Breadth First Search (BFS): use queue data structure to implementing the list ToExplore
- Depth First Search (DFS): use stack data structure to implement the list ToExplore
Search Tree

One can create a natural search tree \( T \) rooted at \( u \) during search.

\[
\text{Explore}(G, u):
\]
\[
\begin{align*}
\text{array } & \text{Visited}[1..n] \\
\text{Initialize: } & \text{Visited}[i] \leftarrow \text{FALSE} \text{ for } i = 1, \ldots, n \\
\text{List: } & \text{ToExplore}, S \\
\text{Add } u \text{ to } & \text{ToExplore and to } S, \text{Visited}[u] \leftarrow \text{TRUE} \\
\text{Make tree } & T \text{ with root as } u \\
\text{while } (\text{ToExplore is non-empty}) \text{ do} & \\
\text{Remove node } x \text{ from } & \text{ToExplore} \\
& \text{for each edge } (x, y) \text{ in } \text{Adj}(x) \text{ do} \\
& \text{if } (\text{Visited}[y] = \text{FALSE}) \\
& \quad \text{Visit } y \leftarrow \text{TRUE} \\
& \quad \text{Add } y \text{ to } \text{ToExplore} \\
& \quad \text{Add } y \text{ to } S \\
& \quad \text{Add } y \text{ to } T \text{ with } x \text{ as its parent} \\
\text{Output } & S
\end{align*}
\]

\( T \) is a spanning tree of \( \text{con}(u) \) rooted at \( u \)

Finding all connected components

**Exercise:** Modify Basic Search to find all connected components of a given graph \( G \) in \( O(m + n) \) time.

Part III

Directed Graphs and Decomposition

**Directed Graphs**

**Definition**

A directed graph \( G = (V, E) \) consists of

- a set of vertices/nodes \( V \)
- a set of edges/arcs \( E \subseteq V \times V \)

An edge is an ordered pair of vertices. \((u, v)\) different from \((v, u)\).
Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

- Road networks with one-way streets.
- Web-link graph: vertices are web-pages and there is an edge from page \( p \) to page \( p' \) if \( p \) has a link to \( p' \). Web graphs used by Google with PageRank algorithm to rank pages.
- Dependency graphs in variety of applications: link from \( x \) to \( y \) if \( y \) depends on \( x \). Make files for compiling programs.
- Program Analysis: functions/procedures are vertices and there is an edge from \( x \) to \( y \) if \( x \) calls \( y \).

Directed Graph Representation

Graph \( G = (V, E) \) with \( n \) vertices and \( m \) edges:

- **Adjacency Matrix**: \( n \times n \) asymmetric matrix \( A \). \( A[u, v] = 1 \) if \( (u, v) \in E \) and \( A[u, v] = 0 \) if \( (u, v) \not\in E \). \( A[u, v] \) is not same as \( A[v, u] \).
- **Adjacency Lists**: for each node \( u \), \( Out(u) \) (also referred to as \( Adj(u) \)) and \( In(u) \) store out-going edges and in-coming edges from \( u \).

Default representation is adjacency lists.

A Concrete Representation for Directed Graphs

Concrete representation discussed previously for undirected graphs easily extends to directed graphs.

- Array of edges \( E \)
- Array of adjacency lists

Directed Connectivity

Given a graph \( G = (V, E) \):

- A **(directed) path** is a sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i \leq k - 1 \). The length of the path is \( k - 1 \) and the path is from \( v_1 \) to \( v_k \). By convention, a single node \( u \) is a path of length 0.
- A **cycle** is a sequence of distinct vertices \( v_1, v_2, \ldots, v_k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i \leq k - 1 \) and \((v_k, v_1) \in E \). By convention, a single node \( u \) is not a cycle.
- A vertex \( u \) can reach \( v \) if there is a path from \( u \) to \( v \). Alternatively \( v \) can be reached from \( u \).
- Let \( rch(u) \) be the set of all vertices reachable from \( u \).
**Connectivity contd**

**Asymmetricity:** $D$ can reach $B$ but $B$ cannot reach $D$

**Questions:**
- Is there a notion of connected components?
- How do we understand connectivity in directed graphs?

**Connectivity and Strong Connected Components**

**Definition**

Given a directed graph $G$, $u$ is **strongly connected** to $v$ if $u$ can reach $v$ and $v$ can reach $u$. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation $C$ where $uCv$ if $u$ is (strongly) connected to $v$.

**Proposition**

$C$ is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of $C$: **strong connected components** of $G$.

They partition the vertices of $G$.

$\text{SCC}(u)$: strongly connected component containing $u$.

**Directed Graph Connectivity Problems**

- Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
- Given $G$ and $u$, compute $\text{rch}(u)$.
- Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$.
- Find the strongly connected component containing node $u$, that is $\text{SCC}(u)$.
- Is $G$ strongly connected (a single strong component)?
- Compute all strongly connected components of $G$. 
## Basic Graph Search in Directed Graphs

Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$. Let

\[ \text{Explore}(G, u): \]

array $\text{Visited}[1..n]$

Initialize: Set $\text{Visited}[i] \leftarrow \text{FALSE}$ for $1 \leq i \leq n$

List: $\text{ToExplore}, S$

Add $u$ to $\text{ToExplore}$ and to $S$, $\text{Visited}[u] \leftarrow \text{TRUE}$

Make tree $T$ with root as $u$

while ($\text{ToExplore}$ is non-empty) do

Remove node $x$ from $\text{ToExplore}$

for each edge $(x, y)$ in $\text{Adj}(x)$ do

if ($\text{Visited}[y] = \text{FALSE}$)

$\text{Visited}[y] \leftarrow \text{TRUE}$

Add $y$ to $\text{ToExplore}$

Add $y$ to $S$

Add $y$ to $T$ with edge $(x, y)$

Output $S$

### Properties of Basic Search

**Proposition**

\[ \text{Explore}(G, u) \text{ terminates with } S = \text{rch}(u). \]

**Proof Sketch.**

- Once $\text{Visited}[i]$ is set to $\text{TRUE}$ it never changes. Hence a node is added only once to $\text{ToExplore}$. Thus algorithm terminates in at most $n$ iterations of while loop.
- By induction on iterations, can show $v \in S \Rightarrow v \in \text{rch}(u)$
- Since each node $v \in S$ was in $\text{ToExplore}$ and was explored, no edges in $G$ leave $S$. Hence no node in $V - S$ is in $\text{rch}(u)$.
- **Caveat:** In directed graphs edges can enter $S$.

Thus $S = \text{rch}(u)$ at termination.

**Example**

![Directed Graph Example](example_graph.png)

**Properties of Basic Search**

**Proposition**

\[ \text{Explore}(G, u) \text{ terminates in } O(m + n) \text{ time}. \]

**Proposition**

$T$ is a search tree rooted at $u$ containing $S$ with edges directed away from root to leaves.

**Proof:** easy exercises

- **BFS** and **DFS** are special case of Basic Search.
  - **Breadth First Search (BFS):** use queue data structure to implementing the list $\text{ToExplore}$
  - **Depth First Search (DFS):** use stack data structure to implement the list $\text{ToExplore}$
Exercise

Prove the following:

Proposition

Let $S = \text{rch}(u)$. There is no edge $(x, y) \in E$ where $x \in S$ and $y \not\in S$.

Describe an example where $\text{rch}(u) \neq V$ and there are edges from $V \setminus \text{rch}(u)$ to $\text{rch}(u)$.

Algorithms via Basic Search - I

• Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
• Given $G$ and $u$, compute $\text{rch}(u)$.

Use $\text{Explore}(G, u)$ to compute $\text{rch}(u)$ in $O(n + m)$ time.

Directed Graph Connectivity Problems

• Given $G$ and nodes $u$ and $v$, can $u$ reach $v$?
• Given $G$ and $u$, compute $\text{rch}(u)$.
• Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$.
• Find the strongly connected component containing node $u$, that is $\text{SCC}(u)$.
• Is $G$ strongly connected (a single strong component)?
• Compute all strongly connected components of $G$.

First five problems can be solved in $O(n + m)$ time by via Basic Search (or BFS/DFS). The last one can also be done in linear time but requires a rather clever DFS based algorithm.

Algorithms via Basic Search - II

• Given $G$ and $u$, compute all $v$ that can reach $u$, that is all $v$ such that $u \in \text{rch}(v)$. Naive: $O(n(n + m))$

Definition (Reverse graph.)

Given $G = (V, E)$, $G^{\text{rev}}$ is the graph with edge directions reversed $G^{\text{rev}} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\text{rch}(u)$ in $G^{\text{rev}}$!

• Correctness: exercise
• Running time: $O(n + m)$ to obtain $G^{\text{rev}}$ from $G$ and $O(n + m)$ time to compute $\text{rch}(u)$ via Basic Search. If both $\text{Out}(v)$ and $\text{In}(v)$ are available at each $v$ then no need to explicitly compute $G^{\text{rev}}$. Can do $\text{Explore}(G, u)$ in $G^{\text{rev}}$ implicitly.
Algorithms via Basic Search - III

\[ \text{SCC}(G, u) = \{ v \mid u \text{ is strongly connected to } v \} \]

- Find the strongly connected component containing node \( u \).
  - That is, compute \( \text{SCC}(G, u) \).

\[ \text{SCC}(G, u) = rch(G, u) \cap rch(G^{rev}, u) \]

Hence, \( \text{SCC}(G, u) \) can be computed with \( \text{Explore}(G, u) \) and \( \text{Explore}(G^{rev}, u) \). Total \( O(n + m) \) time.

Why can \( rch(G, u) \cap rch(G^{rev}, u) \) be done in \( O(n) \) time?

II: Graph \( G \) a vertex \( F \)

- and its reachable set \( (G, F) \)

III: Graph \( G \) a vertex \( F \)

- and the set of vertices that can reach it in \( G \): \( (G^{rev}, F) \)

Reachable set of vertices from \( F \)

Set of vertices that can reach \( F \), computed via DFS in the reverse graph \( G^{rev} \).
**IV: Graph $G$ a vertex $F$ and...**

its strong connected component in $G$: $(G, F)$

- Graph $G$
- $rch(G, F)$
- $rch(G^{rev}, F)$

$$SCC(G, F) = rch(G, F) \cap rch(G^{rev}, F)$$

---

**Algorithms via Basic Search - IV**

- Is $G$ strongly connected?
- Pick arbitrary vertex $u$. Check if $SCC(G, u) = V$.

---

**Algorithms via Basic Search - V**

- Find all strongly connected components of $G$.

While $G$ is not empty do
  Pick arbitrary node $u$
  find $S = SCC(G, u)$
  Remove $S$ from $G$

**Question:** Why doesn’t removing one strong connected components affect the other strong connected components?

Running time: $O(n(n + m))$.

**Question:** Can we do it in $O(n + m)$ time?

---

**Modeling Problems as Search**

The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

In a single turn, you may move either token right, down, or left by a distance determined by the **other** token. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.

- A five-move solution for a $4 \times 4$ Vidrach Itky Leda puzzle.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle above, your algorithm would return the number 5.
Undirected vs Directed Connectivity

Consider following problem.

- Given undirected graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes). $R$ and $B$ non-empty.
- Describe linear-time algorithm to decide whether every red node can reach every blue node.

How does the problem differ in directed graphs?

- Given directed graph $G = (V, E)$.
- Two subsets of nodes $R \subset V$ (red nodes) and $B \subset V$ (blue nodes).
- Describe linear-time algorithm to decide whether every red node can be reached by some blue node.