Lecture 13: Graphs I: Breadth First Search

Lecture Overview

- Applications of Graph Search
- Graph Representations
- Breadth-First Search

Recall:

Graph \( G = (V, E) \)

- \( V \) = set of vertices (arbitrary labels)
- \( E \) = set of edges i.e. vertex pairs \((v, w)\)
  - ordered pair \( \Rightarrow \) directed edge of graph
  - unordered pair \( \Rightarrow \) undirected

![Example to illustrate graph terminology](image)

Figure 1: Example to illustrate graph terminology
Graph Search

“Explore a graph”, e.g.:

- find a path from start vertex $s$ to a desired vertex
- visit all vertices or edges of graph, or only those reachable from $s$

Applications:

There are many.

- web crawling (how Google finds pages)
- social networking (Facebook friend finder)
- network broadcast routing
- garbage collection
- model checking (finite state machine)
- solving puzzles and games

Pocket Cube:

Consider a $2 \times 2 \times 2$ Rubik’s cube

![Rubik's cube](image)

Configuration Graph:

- vertex for each possible state
- edge for each basic move (e.g., 90 degree turn) from one state to another
- undirected: moves are reversible

Diameter starting from the solved state = “God’s Number”
11 for $2 \times 2 \times 2$, 20 for $3 \times 3 \times 3$, $\Theta(n^2/\log n)$ for $n \times n \times n$ [Demaine, Demaine, Eisenstat Lubiw Winslow 2011]
# vertices = $8! \cdot 3^8 = 264,539,520$ where $8!$ comes from having 8 cubelets in arbitrary positions and $3^8$ comes as each cubelet has 3 possible twists.

This can be divided by 24 if we remove cube symmetries and further divided by 3 to account for actually reachable configurations (there are 3 connected components).

**Graph Representations:** (data structures)

**Adjacency lists:**

Array $\text{Adj}$ of $|V|$ linked lists

- for each vertex $u \in V, \text{Adj}[u]$ stores $u$’s neighbors, i.e., $\{v \in V \mid (u, v) \in E\}$. $(u, v)$ are just outgoing edges if directed. (See Fig. 2 for an example.)
- in Python: $\text{Adj} = \text{dictionary of list/set values}; \text{vertex} = \text{any hashable object (e.g., int, tuple)}$
- advantage: multiple graphs on same vertices

**Implicit Graphs:**

$\text{Adj}(u)$ is a function — compute local structure on the fly (e.g., Rubik’s Cube). This requires “Zero” Space.
Breadth-First Search

Explore graph level by level from $s$

- level 0 = \{s\}
- level $i$ = vertices reachable by path of $i$ edges but not fewer

Figure 2: Adjacency List Representation: $\text{Space } \Theta(V + E)$

Figure 3: Illustrating Breadth-First Search
• build level $i > 0$ from level $i - 1$ by trying all outgoing edges, but ignoring vertices from previous levels

Breadth-First-Search Algorithm

BFS $(V, Adj, s)$:

// Initialize
// level is an array (or a hash table)
level[$s$] = 0
for every $u \neq s$, level[$u$] = $\infty$
// parent is an array (or a hash table)
for every $u \in V$, parent[$u$] = null
// frontier is a queue
frontier = [$s$]
while frontier $\neq$ empty:
    remove node $u$ from the front of the frontier list
    for $v$ in $Adj[u]$:
        if level[$v$] = $\infty$:
        # not yet seen
            level[$v$] = level[$u$] + 1
            parent[$v$] = $u$
            frontier.append($v$)  # Add to the end of the list
        else:
        # already seen
            do nothing

Example

Figure 4: Breadth-First Search Frontier
Analysis:

- vertex $V$ enters frontier only once (because $\text{level}[v]$ then set)
  - base case: $v = s$
- $\implies$ Adj[$v$] looped through only once
  \[
  \text{time} = \sum_{v \in V} |\text{Adj}[V]| = \begin{cases} 
  |E| & \text{for directed graphs} \\
  2|E| & \text{for undirected graphs}
  \end{cases}
  \]
- $\implies O(E)$ time

- $O(V + E)$ ("LINEAR TIME") to also list vertices unreachable from $v$ (those still not assigned level)

Shortest Paths:

cf. L15-18

- for every vertex $v$, fewest edges to get from $s$ to $v$ is
  \[
  \begin{cases} 
  \text{level}[v] & \text{if } v \text{ assigned level} \\
  \infty & \text{else (no path)}
  \end{cases}
  \]
- parent pointers form shortest-path tree = union of such a shortest path for each $v$
  \[
  \implies \text{to find shortest path, take } v, \text{parent}[v], \text{parent}[\text{parent}[v]], \text{etc., until } s \text{ (or None)}
  \]