6.006- Introduction to Algorithms

Lecture 12

Prof. Constantinos Daskalakis

CLRS 22.2-22.3
Graphs

- $G = (V, E)$
- $V$ a set of vertices
  - Usually number denoted by $n$
- $E \subseteq V \times V$ a set of edges (pairs of vertices)
  - Usually number denoted by $m$
  - Note $m \leq n(n-1) = O(n^2)$
- Flavors:
  - Pay attention to order of vertices in edge: *directed* graph
  - Ignore order: *undirected* graph
    - Then only $n(n-1)/2$ possible edges
Examples

• **Undirected**
  - $V = \{a, b, c, d\}$
  - $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{c, d\}\}$

• **Directed**
  - $V = \{a, b, c\}$
  - $E = \{(a, c), (a, b), (b, c), (c, b)\}$
Pocket Cube

- $2 \times 2 \times 2$ Rubik’s cube
- Configurations are adjacent, if one can be obtained from the other by quarter turns
- **Basic Question:** is solved state reachable with such moves from the starting state?
Representation

• To solve graph problems, must examine graph
• So need to represent in computer
• Four representations with pros/cons
  ▪ *Adjacency lists* (of neighbors of each vertex)
  ▪ *Incidence lists* (of edges from each vertex)
  ▪ *Adjacency matrix* (of which pairs are adjacent)
  ▪ *Implicit representation* (as neighbor function)
Example
Searching Graph

- We want to get from current Rubik state to “solved” state
- How do we explore?
Breadth First Search

- Start with vertex v
- List all its neighbors (distance 1)
- Then all their neighbors (distance 2)
- Etc.
Depth First Search

• Like exploring a maze
• From current vertex, move to another
• Until you get stuck
• Then backtrack till you find a new place to explore
Problem: Cycles

• What happens if unknowingly revisit a vertex?
• BFS: get wrong notion of distance
• DFS: may get in circles
• Solution: mark vertices
  ▪ BFS: if you’ve seen it before, ignore
  ▪ DFS: if you’ve seen it before, back up
Breadth First Search (BFS)
Outline

- Initial vertex $s$
  - Level 0
- For $i=1,...$
  - grow level $i$
    - Find all neighbors of level $i-1$ vertices
    - (except those already seen)
    - i.e. level $i$ contains vertices reachable via a path of $i$ edges and no fewer
Example
Outline

• Initial vertex s
  - Level 0
• For i=1,…
  - grow level i
  - Find all neighbors of level i-1
  - (except those already seen)
  - i.e. level i contains vertices reachable via a path of i edges and no fewer

• Where can the other edges of the graph be?
  - Only between nodes in same or adjacent levels
Algorithm

• BFS(V, Adj, s)

\[ \text{level} = \{s: 0\}; \text{parent} = \{s: \text{None}\}; i = 1 \]
\[ \text{frontier} = [s] \quad \# \text{previous level, } i-1 \]

while \text{frontier}

\[ \text{next} = [] \quad \# \text{next level, } i \]

for u in \text{frontier}

for v in Adj[u]

if v not in \text{level} \quad \# \text{not yet seen}

\[ \text{level}[v] = i \quad \# \text{level of } u+1 \]
\[ \text{parent}[v] = u \]
\[ \text{next}.append(v) \]

\text{frontier} = \text{next}

i += 1
Analysis: Runtime

• Vertex $v$ appears at the **frontier** at most once
  ▪ Since then it has a level
  ▪ And nodes with a level aren’t added again
  ▪ Total time spent adding nodes to **frontier** $O(n)$

• $\text{Adj}[v]$ only scanned once
  ▪ Just when $v$ is in **frontier**
  ▪ Total time $\sum_v |\text{Adj}[v]|$
    • This sum counts each “outgoing” edge
    • So $O(m)$ time spend scanning adjacency lists

• Total: $O(m+n)$ time --- “Linear time”
Analysis: Correctness

i.e. why are all nodes reachable from $s$ explored?

- **Claim:** If there is a path of $L$ edges from $s$ to $v$, then $v$ is added to next when $i=L$ or before

- **Proof:** induction
  - Base case: $s$ is added before setting $i=1$
  - Path of length $L$ from $s$ to $v$
  - $\Rightarrow$ path of length $L-1$ from $s$ to $u$, and edge $(u,v)$
  - By induction, add $u$ when $i=L-1$ or before
  - If $v$ has not already been inserted in next before $i=L$, it gets added when scan $u$ at $i=L$
  - So it happens when $i=L$ or before
Shortest Paths

• From correctness analysis, conclude more:
  ▪ Level[v] is length of shortest s—v path

• Parent pointers form a shortest paths tree
  ▪ Which is union of shortest paths to all vertices

• To find shortest path, follow parent pointers
  ▪ Will end up at s
Depth First Search (DFS)
Outline

• Explore a maze
  ▪ Follow path until you get stuck
  ▪ Backtrack along breadcrumbs till find new exit
  ▪ i.e. recursively explore
Algorithm

• *parent* = \{s: None\}
• call DFS-visit (V, Adj, s)

Routine DFS-visit (V, Adj, u)
  for v in Adj[u]
    if v not in *parent*  # not yet seen
      *parent*[v] = u
      DFS-visit (V, Adj, v)  # recurse!
Demo (from s)

1 (in tree)  2 (in tree)  3 (in tree)  4 (back edge)  5 (forward edge)  6 (in tree)  7 (cross edge)
Runtime Analysis

• Quite similar to BFS
• DFS-visit only called once per vertex v
  ▪ Since next time v is in parent set
• Edge list of v scanned only once (in that call)
• So time in DFS-visit is 1/vertex + 1/edge
• So time is O(n+m)
Correctness?

- Trickier than BFS
- Can use induction on length of shortest path from starting vertex
  - Induction Hypothesis: “each vertex at distance k is visited”
  - Induction Step:
    - Suppose vertex v at distance k
    - Then some u at distance k-1 with edge (u,v)
    - u is visited (by induction hypothesis)
    - Every edge out of u is checked
    - If v wasn’t previously visited, it gets visited from u
Edge Classification

- **Tree edge** used to get to new child
- **Back edge** leads from node to ancestor in tree
- **Forward edge** leads to descendant in tree
- **Cross edge** leads to a different subtree

- To label what edge is of what type, keep global time counter and store interval during which vertex is on recursion stack
Tradeoffs

- Solving Rubik’s cube?
  - BFS gives shortest solution

- Robot exploring a building?
  - Robot can trace out the exploration path
  - Just drops markers behind

- Only difference is “next vertex” choice
  - BFS uses a queue
  - DFS uses a stack (recursion)