Algorithms in a Nutshell

Session 6
Graph Algorithms
1:00 – 1:50
Outline

• Overview
• Themes
  – Adjacency lists vs. adjacency matrix
  – Search strategy (breadth first vs. depth first)
  – Space vs. Time
• DIJKSTRA’S ALGORITHM
  – Implementations for sparse and dense graphs
Graphs

• Common data structure
  – Represents information relationships

Vertices: v1, v2, v3, v4, v5

Edges: (v1,v2), (v1,v3), (v1,v5), (v2,v4), (v4,v5)
Graphs

- Common data structure
  - Represents information relationships

Vertices: $v_1, v_2, v_3, v_4, v_5$

Edges: $(v_1,v_2), (v_1,v_3), (v_1,v_5), (v_2,v_4), (v_4,v_5)$
Graph Representation Options

- **Adjacency matrix**
  - Two dimensional
  - Non-zero represents edge
  - Find edge by matrix[i][j] index
  - Space: $O(V^2)$

- **Adjacency lists**
  - Array of linked lists
  - Find edge requires search
  - Space: $O(V+E)$
Graph Representation Options

- Adjacency matrix
  - Two dimensional
  - Non-zero represents edge
  - Find edge by matrix[i][j] index
  - Space: $O(V^2)$

- Adjacency lists
  - Array of linked lists
  - Find edge requires search
  - Space: $O(V+E)$
Does edge exist between “Boston” and “Providence”?

• Adjacency matrix
  – Use hash table to determine integer \(i\) associated with “Boston”
  – Use hash table to determine integer \(j\) associated with “Providence”
  – Edge exists if \(\text{edge}[i][j] > 0\)

• Adjacency lists
  – Use hash table to determine integer \(i\) associated with “Boston”
  – Search the linked list \(\text{vertices}[i]\) to see if a node exists whose name is “Providence”
  – Edge exists if node found
Normalized Graph Representation

- Assume all vertices are in the range \([0, n]\)
  - Enables efficient edge lookup for adjacency matrix
- Assume all requests are normalized
  - Avoids hash table lookup

```c
bool isEdge (int u, int v)
int edgeWeight (int u, int v)
void addEdge (int u, int v, int w)
```

```c
// Graph
#define VERTICES_MAX 100

struct Graph {
    int numVertices;
    int numEdges;
    int* adjacencyMatrix;
    int* edgeWeights;
    int* vertexDegrees;
    int* vertexLabels;
    int* vertexIndices;
};

Graph* createGraph(int n) {
    ... // initialize graph
    return graph;
}

void addEdge(Graph* graph, int u, int v, int weight) {
    ... // add edge and update adjacency matrix
}

int isEdge(Graph* graph, int u, int v) {
    ... // check if edge exists
    return edgeExists;
}

int edgeWeight(Graph* graph, int u, int v) {
    ... // return edge weight
    return weight;
}
```
Common Graph Problems

• Is there a path from $V_0$ to vertex $V_1$?
• What is shortest path from $V_0$ to vertex $V_1$?
  – In number of edges traversed
  – In accumulating edge weights
• What is the shortest path between any two vertices?
  – In accumulating edge weights
Maze Example

- Problem: Solve a rectangular maze
  - “Is there a path from S to T”
- Mapping a problem to a graph
  - Identify vertices and edges

Vertex represents maze decision point

Edge represents path in maze between decision points
Maze Search

- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options

- To implement
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision
Maze Search

- **Depth-First Strategy**
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options

- **To implement**
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision
Maze Search

- **Depth-First Strategy**
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options

- **To implement**
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision
Maze Search

- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options

- To implement
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision
Maze Search

- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options

- To implement
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision
Depth-first search of a graph

- Backtracking using recursion
  - Each invocation of `dfs_visit` visits a new vertex
  - Only called on White vertices
- Record search progress by coloring vertices after they have been visited
  - White = Unvisited
  - Gray = Visited but haven’t visited all neighbors
  - Black = Visited and have visited all neighbors
- To record search path, use `pred` array to store path
- If graph is disconnected
  - Lines 5-7 completes search

```
def depthFirstSearch(G, s):
    for v in G:
        pred[v] = -1
        color[v] = "White"
    dfs_visit(s)

def dfs_visit(v):
    color[v] = "Gray"
    for each neighbor u of v:
        if color[u] == "White":
            pred[u] = v
            dfs_visit(u)
    color[v] = "Black"
```

As each `dfs_visit` completes, unvisited vertices initially passed over are explored (i.e., 6 was a White neighbor of 2). Completed vertices are colored Black.

If the graph is unconnected then some vertex will be colored White. Continue to explore these unvisited vertices.

`pred[]` information records depth-first forest discovered, shown as arrows.
Code Check

- Code check
  - Debug figure6_10.exe
  - Review code
- Breakpoint in dfs_visit
  - Note stack trace when u=15
Implementation Details

- Keep track of “active search horizon”
  - The recursion stack of dfs_visit invocations
- Backtrack to revisit earlier decision

```cpp
void dfs_visit (Graph const &graph, int u, vector<int> &pred, vector<vertexColor> &color) {
  color[u] = Gray;

  // process all neighbors of u.
  for (VertexList::const_iterator ci = graph.begin(u); ci != graph.end(u); ++ci) {
    int v = ci->first;

    // Explore unvisited vertices immediately and record pred[]. Once
    // recursive call ends, backtrack to adjacent vertices.
    if (color[v] == White) {
      pred[v] = u;
      dfs_visit (graph, v, pred, color);
    }
  }

  color[u] = Black; // our neighbors are complete; now so are we.
}
```
Breadth-First Search Strategy

• Systematic exploration of graph
  – Visit all vertices that are $k$ edges away from initial vertex before visiting vertices $k+1$ edges away

• Only visit unmarked vertices
  – Use same coloring scheme as DEPTH-FIRST SEARCH

• Maintain “active search horizon”
  – Use queue to store to-be-visited vertices
Queue Data structure

- Insert elements to the end
- Remove elements from the front
Maze Search

• Breadth-First Strategy
  – Visit vertices $k$ edges away before visiting those $k+1$ edges away
  – Never visit the same vertex twice – avoids infinite loops

• To implement
  – Use queue to keep track of “active search horizon”
Maze Search

- **Breadth-First Strategy**
  - Visit vertices $k$ edges away before visiting those $k+1$ edges away
  - Never visit the same vertex twice – avoids infinite loops

- **To implement**
  - Use queue to keep track of “active search horizon”
Maze Search

• Breadth-First Strategy
  – Visit vertices \( k \) edges away before visiting those \( k+1 \) edges away
  – Never visit the same vertex twice – avoids infinite loops

• To implement
  – Use queue to keep track of “active search horizon”
Maze Search

• Breadth-First Strategy
  – Visit vertices $k$ edges away before visiting those $k+1$ edges away
  – Never visit the same vertex twice – avoids infinite loops

• To implement
  – Use queue to keep track of “active search horizon”
Maze Search

- **Breadth-First Strategy**
  - Visit vertices $k$ edges away before visiting those $k+1$ edges away
  - Never visit the same vertex twice – avoids infinite loops

- **To implement**
  - Use queue to keep track of “active search horizon”
Maze Search

- **Breadth-First Strategy**
  - Visit vertices \(k\) edges away before visiting those \(k+1\) edges away
  - Never visit the same vertex twice – avoids infinite loops

- **To implement**
  - Use queue to keep track of “active search horizon”
Maze Search

• Breadth-First Strategy
  – Visit vertices $k$ edges away before visiting those $k+1$ edges away
  – Never visit the same vertex twice – avoids infinite loops

• To implement
  – Use queue to keep track of “active search horizon”
Maze Search

• Breadth-First Strategy
  – Visit vertices $k$ edges away before visiting those $k+1$ edges away
  – Never visit the same vertex twice – avoids infinite loops

• To implement
  – Use queue to keep track of “active search horizon”
Breadth-first search of a graph

- **Systematic**
  - Exploration of graph
    - Will find shortest path from $s$ to every node in graph
  - Will leave unreachable vertices unvisited
- **Non-recursive**
Code Check

- Code check
  - Debug figure6_12.exe
  - Review code
- Breakpoint in bfs_visit
  - Stop when $u = 2$
  - Colored vertices as shown
  - $pred[]$ info as shown

<table>
<thead>
<tr>
<th>$w$</th>
<th>$dist[]$</th>
<th>$pred[]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>INF</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>INF</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>INF</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>INF</td>
<td>-1</td>
</tr>
<tr>
<td>12</td>
<td>INF</td>
<td>-1</td>
</tr>
<tr>
<td>13</td>
<td>INF</td>
<td>-1</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>INF</td>
<td>-1</td>
</tr>
</tbody>
</table>
Implementation Details

- Keep track of “active search horizon”
  - Queue holds vertices to be visited
  - Only add “Gray” nodes to Queue

```cpp
void bfs_search (Graph const &graph, int s,       /* in */
                 vector<int> &dist, vector<int> &pred){ /* out */
  // initialize dist and pred. Begin at s
  // and mark as Gray since we haven't yet visited its neighbors.
  const int n = graph.numVertices();
  pred.assign(n, -1);
  dist.assign(n, numeric_limits<int>::max());
  vector<vertexColor> color (n, White);

dist[s] = 0;
color[s] = Gray;

queue<int> q;
q.push(s);
while (!q.empty()) {
  int u = q.front();

  // Explore neighbors of u to expand the search horizon
  for (VertexList::const_iterator ci = graph.begin(u);
           ci != graph.end(u); ++ci) {
    int v = ci->first;
    if (color[v] == White) {
      dist[v] = dist[u] +1;
      pred[v] = u;
      color[v] = Gray;
      q.push(v);
    }
  }
  q.pop();
  color[u] = Black;
}
}```
Space vs. Time

- Depth-First and Breadth-First both iterate over the edges for a vertex
  - Adjacency List via Iterator
  - Adjacency Matrix via double-loop
- Costs change if **sparse** or **dense** graph

```cpp
// Explore neighbors of u to expand
// search horizon
for (VertexList::const_iterator ci = graph.begin(u);
     ci != graph.end(u); ++ci) {
    int v = ci->first;
    ...
}
```

```cpp
// Explore neighbors of u to expand
// search horizon
for (int v = 0; v < n; v++) {
    if (graph.edge[u][v] == 0) { continue; }
    ...
}
```
Searching

• Breadth-First and Depth-First can determine whether path exists between two vertices
  – What if you wanted to consider edge weights?
  – That is, find shortest path between $v_0$ and $v_1$?
    • Breadth-first finds path with smallest number of edges

• Single-Source Shortest Path
  – Edges are now directed and have weights
  – DIJKSTRA’S ALGORITHM (1959)
Searching with purpose

- Breadth-First and Depth-First are blind searches
  - BFS ignores context as it systematically executes
  - DFS selects a direction at random
- Goal: find shortest distance using edge weights
  - How do we avoid generating all possible paths?
- Employ Greedy Strategy
  - Find shortest distance from \( v_0 \) to all vertices
  - Computing for all makes problem easier to solve!
Single-Source Shortest Path

- Goal: find shortest distance from 0 to 3
- Key idea
  - Compute running “shortest distance” from source to all vertices
  - Expand marked region by adding the vertex with smallest distance (marked in yellow below)
Single-Source Shortest Path

- **Goal:** find shortest distance from 0 to 3
- **Key idea**
  - Compute running “shortest distance” from source to all vertices
  - Expand marked region by adding the vertex with smallest distance (marked in yellow below)

```
<table>
<thead>
<tr>
<th>dist</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>∞</td>
<td>8</td>
<td>∞</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>∞</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
```

*How can we efficiently locate the vertex with smallest distance?*

*Use a Priority Queue!*
Priority Queue data structure

• Add element with associated numeric priority
  – Lower priority numbers imply greater priority
• Retrieve element with lowest priority

  *If these are the only operations you need, then you can use an ordinary Binary Heap for efficient implementation. However, we also need:

• Decrease priority of existing element
  – How to avoid $O(q)$ search for element within PQ?
Binary Heap with extra space

- We can use binary heap as PQ here because
  - We know maximum size will be \( n \)
- \textit{decreaseKey} operation can be done in \( O(\log q) \)
  - Store additional space, only \( O(n) \)

```cpp
class BinaryHeap {
public:
  BinaryHeap (int);
  ~BinaryHeap ();

  bool isEmpty() { return (_n == 0); }  
  int smallest();
  void insert (int, int);
  void decreaseKey (int, int);

private:
  int _n;                               // number of elements in binary heap
  ELEMENT_PTR _elements;                // values in the heap
  int * _pos;                           // pos[i] is index into elements for ith value
};
```
DIJKSTRA’S ALGORITHM

- Initialization
  - Construct PQ with $n$ vertices

- Core step
  - Extract vertex $u$ with smallest distance
  - If distance $(s,u) + (u,v) \leq (s,v)$ for a neighboring $v$ of $u$, then reduce $\text{dist}[v]$ and its location in PQ

- How to reproduce actual shortest path?
  - Follow $\text{pred}[]$ reference which is computed by the algorithm

---

Dijkstra’s Algorithm

<table>
<thead>
<tr>
<th>PQ</th>
<th>Priority Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>$O((V+E) \log V)$</td>
<td>same</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weighted Directed Graph</th>
<th>Overwrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Image of graph]</td>
<td>[Image of graph]</td>
</tr>
</tbody>
</table>

singleSourceShortest $(G, s)$

1. \( \text{PQ} = \text{new Priority Queue} \)
2. set $\text{dist}[v]$ to $\infty$ for all $v \in G$
3. set $\text{pred}[v]$ to $-1$ for all $v \in G$
4. $\text{dist}[s] = 0$
5. \( \text{foreach } v \in G \) do
6. \( \text{PQ.insert}(v, \text{dist}[v]); \)

7. while (PQ is not empty) do
8. \( u = \text{getMin}(\text{PQ}) \)
9. \( \text{foreach neighbor } v \text{ of } u \) do
10. \( w = \text{weight of edge } (u,v) \)
11. \( \text{newLen} = \text{dist}[u] + w \)
12. if (newLen < dist[v]) then
13. \( \text{decreaseKey}(\text{PQ}, v, \text{newLen}) \)
14. \( \text{dist}[v] = \text{newLen} \)
15. \( \text{pred}[v] = u \)
16. end

Remove vertex $u$ from PQ with least distance from $s$. If path from $(s,u)$ and $(u,v)$ is shorter than best computed distance $(s,v)$, adjust $\text{dist}[v]$ and PQ.

1st iteration: remove 0 and adjust

2nd iteration: remove 1 and adjust

5th iteration: remove 3 and done

3rd iteration: remove 4 and adjust

4th iteration: remove 2 and adjust

---

Algorithms in a Nutshell (c) 2009, George Heineman
void singleSourceShortest(Graph const &g, int s, /* in */
    vector<int> &dist, vector<int> &pred) { /* out */
    // initialize dist[] and pred[] arrays. Start with vertex s by setting
    // dist[] to 0. Priority Queue PQ contains all v in G.
    const int n = g.numVertices();
    pred.assign(n, -1);
    dist.assign(n, numeric_limits<int>::max());
    dist[s] = 0;
    BinaryHeap pq(n);
    for (int u = 0; u < n; u++) { pq.insert (u, dist[u]); }

    // find vertex in ever shrinking set, V-S, whose dist[] is smallest.
    // Recompute potential new paths to update all shortest paths
    while (!pq.isEmpty()) {
        int u = pq.smallest();

        // For neighbors of u, see if newLen (best path from s->u + weight
        // of edge u->v) is better than best path from s->v. If so, update
        // in dist[v] and re-adjust binary heap accordingly. Compute in
        // long to avoid overflow error.
        for (VertexList::const_iterator ci = g.begin(u); ci != g.end(u); ++ci) {
            int v = ci->first;
            long newLen = dist[u] + ci->second;
            if (newLen < dist[v]) {
               pq.decreaseKey (v, newLen);
               dist[v] = newLen;
               pred[v] = u;
            }
        }
    }
}
Summary

• Rich family of graph algorithms
  – BFS and DFS provide search strategies
  – Greedy Algorithms (PRIM’s Minimum Spanning Tree)
  – Dynamic Programming

• Algorithm designer Robert Tarjan said
  – “with the right data structure most quadratic problems can be solved in $O(n \log n)$” (paraphrased)
Performance Comparison

- Compare the following performance families
  - $O((V+E) \log V)$ \quad DIJKSTRA’S ALGORITHM
  - $O(V^2 + E)$ \quad DIJKSTRA’S ALGORITHM DG

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>$O((V+E) \log V)$</th>
<th>Comparison</th>
<th>$O(V^2+E)$</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse: $E$ is $O(V)$</td>
<td>$O(V \log V)$</td>
<td>Is smaller than $O(V^2)$</td>
<td></td>
<td>4096 Vertices 6000 Edges (0.03%)</td>
</tr>
<tr>
<td>Break-Even: $E$ is $O(V^2/\log V)$</td>
<td>$O(V^2 + V \log V)$ = $O(V^2)$</td>
<td>Is equivalent to $O(V^2+V^2/\log V)$ = $O(V^2)$</td>
<td></td>
<td>4096 Vertices 1,398,101 Edges (8%)</td>
</tr>
<tr>
<td>Dense: $E$ is $O(V^2)$</td>
<td>$O(V^2 \log V)$</td>
<td>Is larger than $O(V^2)$</td>
<td></td>
<td>4096 Vertices 4,193,280 Edges (25%)</td>
</tr>
</tbody>
</table>