Graph Algorithms and Parallel Computing
Study-Ready Notes

Compiled by Andrew Photinakis

October 2nd, 2025

Contents

1 Graph Representations

1.1 Undirected Graph Representation

- **Adjacency Matrix**: Square matrix where entry (i,j) = 1 if vertices i and j are connected, 0 otherwise
- **Adjacency List**: For each vertex, list of adjacent vertices

Example G1 (Undirected Graph)
Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 1: Undirected graph with vertices A,B,C,D,E

1.2 Directed Graph Representation

Example G2 (Directed Graph)
Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 2: Directed graph with vertices P,Q,R,S,T

[Summary] Graph representations include adjacency matrices (good for dense graphs) and adjacency lists (good for sparse graphs). Directed graphs have asymmetric matrices.

2 Graph Traversal Algorithms

2.1 Depth-First Search (DFS)

- Explores as far as possible along each branch before backtracking

- Uses stack (implicit or explicit) for traversal
- Applications: Cycle detection, topological sorting, maze solving

Example DFS Tree:

- A → D, E, B
- B → A, E, C
- C → D, E, B
- D → A, C
- E → A, B, C

2.2 Breadth-First Search (BFS)

- Explores all neighbors at current depth before moving deeper
- Uses queue for traversal
- Applications: Shortest path in unweighted graphs, social networks

Example BFS Tree:

- A → B, E, D
- B → E, C
- E → C
- D → C

[Summary] DFS goes deep first using stack, BFS goes wide first using queue. DFS finds paths, BFS finds shortest paths in unweighted graphs.

3 Minimum-Cost Spanning Trees (MCST)

3.1 Definition and Applications

- **Spanning Tree**: Connected subgraph containing all vertices with no cycles
- **Minimum-Cost Spanning Tree**: Spanning tree with minimum total edge weight
- **Applications**: Network design, circuit wiring, clustering

Network Example:

- Computer network with bidirectional links
- Each link has positive cost (message sending cost)
- Broadcast message from arbitrary computer
- Goal: Minimize total broadcast cost

3.2 Prim's Algorithm

```
def prim_mst(graph, start_node):
    mst = set()
    visited = {start_node}
    edges = [
        (cost, start_node, to)
        for to, cost in graph[start_node].items()
    ]
    heapify(edges)

    while edges and len(visited) < len(graph):
        cost, frm, to = heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.add((frm, to, cost))
            for to_next, cost2 in graph[to].items():
                if to_next not in visited:
                    heappush(edges, (cost2, to, to_next))
    return mst
```

Algorithm Steps:

1. Start with any node as root
2. Grow tree greedily by adding cheapest edge connecting tree to outside vertex
3. Repeat until all vertices are included

Complexity: $O(E \log V)$ with binary heap

[Summary] MCST finds minimum weight tree spanning all vertices. Prim's algorithm grows tree greedily from start node.

4 Shortest Path Algorithms

4.1 Single-Source Shortest Paths (Dijkstra's Algorithm)

- Finds shortest paths from source vertex to all other vertices
- Works for weighted graphs with non-negative weights
- Based on greedy principle

Algorithm:

1. Initialize $d[v] = 0$ for source, ∞ for others
2. For each vertex, compute: $d[x] = \min\{d[x], d[v] + w(v, x)\}$

3. Always pick vertex with minimum distance

Mathematical Formulation:

$$d[x] = \min\{d[x], d[v] + w(v,x)\}, \text{ where } v, x \in V$$

4.2 All-Pairs Shortest Paths

Recursive Solution:

$$dist(i,j) = \begin{cases} w(i,j) & \text{if } k = 0 \\ \min\{dist(i,j), [dist(i,k) + dist(k,j)]\} & \text{if } k \geq 1 \end{cases}$$

Matrix Operations Approach:

- Replace 'multiply' by 'ADD'
- Replace 'add' by 'MINIMUM'
- Ignore infinity entries

[Summary] Dijkstra finds single-source shortest paths, all-pairs uses dynamic programming. Both use greedy/minimization principles.

5 Transitive Closure

5.1 Definition and Applications

- **Transitive Closure**: Directed graph where edge (i,j) exists if there's a directed path from i to j in original graph
- **Security Application**: Identify all users with permission (direct or indirect) to access accounts
- Many applications in database systems, compiler optimization

5.2 Warshall's Algorithm

```
procedure WARSHALL(G=[V,E])
    Input: n × n matrix A representing adjacency
    Output: transitive closure matrix T

    for i ← 1 to n do
        for j ← 1 to n do
            t[i,j] ← a(i,j)

    for k ← 1 to n do
        for i ← 1 to n do
```

```
            for j ← 1 to n do
                if NOT t[i,j] then
                    t[i,j] ← t[i,k] AND t[k,j]
    return T
```

Complexity: $\Theta(n^3)$
Improvement: Algorithm can be optimized for better performance
[Summary] Transitive closure identifies all reachable pairs in a graph. Warshall's algorithm computes it in cubic time using dynamic programming.

6 Matrix Operations on Graphs

6.1 Connectivity and Path Counting

Paths of Length 2:

- Replace 'multiply' by 'AND' and 'add' by 'OR' for existence
- Keep 'add' and replace 'multiply' by 'AND' for counting

Example:

$$C^2 = C \times C = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

6.2 All-Pairs Shortest Paths via Matrix Operations

Operations:

- Replace 'multiply' by 'ADD'
- Replace 'add' by 'MINIMUM'
- Handle infinity entries appropriately

[Summary] Matrix operations can compute connectivity and shortest paths by redefining multiplication and addition operations.

7 Optimal Binary Search Trees (OBST)

7.1 Problem Definition

- Given keys with access probabilities
- Find BST arrangement that minimizes expected access cost
- Cost = $\sum$ (probability $\times$ depth) for all keys

Example: Keys A,B,C,D with probabilities (0.1, 0.2, 0.4, 0.3)

7.2 Recursive Structure

$$c[i,j] = c[i,k-1] + c[k+1,j] + \sum_{s=i}^{j} p_s$$

Where:

- $c[i,j]$ = cost of optimal BST for keys i through j
- k = root of subtree
- $\sum p_s$ = sum of probabilities in current subtree

7.3 Bottom-Up Computation

Base Cases:

- $c[i,j] = 0$ if $i = 0$ or $i \geq j$
- $c[i,j] = p_i$ if $i = j$

Example Computation with P=0.2, Q=0.4, R=0.1, S=0.3:

- C[1,1] = 0.2, C[2,2] = 0.4, C[3,3] = 0.1, C[4,4] = 0.3
- C[1,2] = 0.8 (Q root), C[2,3] = 0.6 (Q root), C[3,4] = 0.5 (S root)

7.4 Parallel OBST Computation

- Compute diagonals in parallel
- C(1,2), C(2,3), C(3,4) on n-1 processors
- C(1,3), C(2,4) on n-2 processors
- Load balancing needed for initial unbalanced assignments

[Summary] OBST minimizes expected search cost using dynamic programming. Parallel computation processes matrix diagonals concurrently.

8 Subgraph Matching

8.1 Problem Definition

- Given data graph G and query graph Q
- Find all subgraphs of G isomorphic to Q
- Applications: Social networks, web graphs, relational databases

Formal Definition:

- G(V,E), Q(Vq, Eq)
- Find subgraph g(Vg, Eg) where Vq $\rightarrow$ Vg and Eq $\rightarrow$ Eg

8.2 Query Decomposition

- Decompose complex query into simpler components (twigs)
- Each processor searches for specific twig in distributed graph
- Handle large graphs: $|E| - O(10^9)$ and $|V| - O(10^8)$

Parallelization Strategy:

- Distribute G across computers
- Each computer searches for assigned twig pattern
- Combine results from all processors

[Summary] Subgraph matching finds pattern occurrences in large graphs. Parallel approach decomposes query and distributes search.

9 Process Assignment and Scheduling

9.1 Basic Concepts

- **Assignment**: Processes to processing elements (WHERE)
- **Scheduling**: When to execute each task (WHEN)
- **Programming Models**: SPMD, MPMD, Shared Memory, Message Passing

9.2 Critical Factors

Granularity:

- Coarse vs Fine grain
- Ratio of computation to communication
- Higher ratio $\rightarrow$ better speedup and efficiency

Overheads:

- Coordination costs
- Synchronization
- Data communication

Scalability:

- Proportionate speedup with more processors
- Affected by memory-CPU bandwidth, network, algorithm characteristics

9.3 System Characteristics

Processor Types:

- **Homogeneous**: Identical processors, uniform costs
- **Heterogeneous**: Varying capabilities, speeds, resources

Network Types:

- Homogeneous/heterogeneous communication bandwidth
- Mobile systems with disconnections

Total Cost Calculation:

$$\text{Total Cost} = \text{computing costs} + \text{communication costs}$$

[Summary] Process assignment and scheduling consider granularity, overheads, scalability. Systems can be homogeneous or heterogeneous.

10 Decomposition Strategies

10.1 Domain Decomposition

- Divide data into discrete chunks
- Each process works on portion of data
- Examples: Matrix operations, image processing
- Maintain high computation/communication ratio (R/C)

10.2 Functional Decomposition

- Each processor performs different function
- Examples: Signal processing pipelines
- Match system (R,C) to application (r,c) characteristics

[Summary] Domain decomposition divides data, functional decomposition divides tasks. Both aim to optimize computation/communication ratio.

11 Load Balancing

11.1 Static Load Balancing

- Fixed policy based on a priori knowledge
- Does not adjust to system state changes

Advantages:

- Simple, low cost
- Easy session state management

Disadvantages:

- Cannot adapt to dynamic changes
- May lead to poor resource utilization

11.2 Dynamic Load Balancing

- Adjusts based on current system state
- Handles uncertainty in execution times, resource availability

Types:

- **Sender-initiated**: Overloaded nodes send work
- **Receiver-initiated**: Underloaded nodes request work
- **Centralized vs Decentralized**

Challenges:

- State monitoring overhead
- Network topology considerations
- Process migration costs

11.3 Process Migration

Migration Process:

1. Migration request initiated
2. Process suspended on source host
3. Process state transferred to destination

4. Process resumes execution on new host

Migration Scenarios:

- Host A has no more work → finds work elsewhere
- Host cannot proceed due to resource constraints

[Summary] Static balancing is simple but inflexible. Dynamic balancing adapts but has overheads. Process migration enables load redistribution.

[Mnemonic]

- **DFS**: Deep First Search (Stack)
- **BFS**: Broad First Search (Queue)
- **MCST**: Minimum Cost Spanning Tree
- **OBST**: Optimal Binary Search Tree

[Concept Map]

- Graph Algorithms → Representations → Traversal → Shortest Paths → Spanning Trees
- Scheduling → Assignment → Load Balancing → Static/Dynamic → Migration
- Parallel Computing → Decomposition → Domain/Functional → Granularity → Scalability