

chapter 7 solutions algorithm design kleinberg tardos

Published: September 27, 2026 | Index ID: #-

Chapter 7 Solutions Algorithm Design Kleinberg Tardos: A Deep Dive into Network Flow and Matching

When you pick up “Algorithm Design” by Jon Kleinberg and Éva Tardos, you’re stepping into a world of elegant solutions that turn complex problems into tractable puzzles. Chapter 7, in particular, is a cornerstone for anyone studying the interplay between flow networks and matching theory. This article walks through the key solutions presented in Chapter 7, offering clear explanations, pseudocode, complexity analysis, and real world applications. Whether you’re a student preparing for exams or a practitioner tackling network optimization, the insights here will deepen your understanding of the algorithms that power modern infrastructure.

Understanding Chapter 7: Network Flow and Matching

Chapter 7 is split into two main themes: **network flow** and **bipartite matching**. These topics are not only mathematically rich but also highly applicable to scheduling, transportation, and resource allocation. The chapter begins by framing the *maximum flow problem*—finding the greatest amount of “stuff” that can be pushed from a source to a sink in a directed graph with capacity limits. It then moves to the *minimum cut problem*, establishing the famous Max Flow Min Cut theorem that links the two concepts. Finally, the chapter tackles bipartite matching, where we pair elements of two disjoint sets optimally, culminating in the celebrated Hungarian algorithm for weighted matching.

Key Concepts Covered

- Residual graphs and augmenting paths
- Capacity scaling and breadth first search (BFS) for shortest augmenting paths
- Blocking flows and level graphs
- Duality in bipartite matching and the assignment problem
- Complexity bounds for classic algorithms

These concepts form the foundation upon which the chapter’s solutions are built. Understanding them is essential before diving into the algorithmic details.

Classic Algorithms and Their Solutions

Chapter 7 presents several algorithms, each with its own strengths and trade offs. Below we break down the most important ones, providing intuitive explanations, pseudocode, and performance insights.

Ford–Fulkerson Method

The Ford–Fulkerson algorithm is the original approach for computing maximum flow. It iteratively searches for any augmenting path in the residual graph and pushes as much flow as possible along that path. While conceptually simple, its performance depends heavily on the choice of augmenting paths. In the worst case, the algorithm can take exponential time if the capacities are irrational or if it chooses poor paths.

```
function FordFulkerson(G, s, t):
```

```
    flow = 0
```

```

while (exists augmenting path P from s to t in residual graph):
    bottleneck = min{capacity(e) | e in P}
    for each edge e in P:
        increase flow on e by bottleneck
        decrease flow on reverse edge by bottleneck
    flow += bottleneck
return flow

```

Complexity: $O(F * E)$, where F is the maximum flow value and E is the number of edges. This bound is loose; in practice, the algorithm can be faster or slower depending on path selection.

Edmonds–Karp Algorithm

To address the unpredictability of Ford–Fulkerson, Edmonds and Karp introduced a specific strategy: always choose the shortest augmenting path (by number of edges) using BFS. This guarantees that the number of augmenting steps is bounded by $O(VE)$, leading to a polynomial time algorithm.

```

function EdmondsKarp(G, s, t):
    flow = 0
    while (P = BFS(G, s, t)):
        bottleneck = min{capacity(e) | e in P}
        for each edge e in P:
            increase flow on e by bottleneck
            decrease flow on reverse edge by bottleneck
        flow += bottleneck
    return flow

```

Complexity: $O(VE^2)$. The algorithm's use of BFS ensures that each augmenting path is as short as possible, limiting the total number of iterations.

Dinic's Algorithm

Dinic's algorithm improves upon Edmonds–Karp by building a level graph and finding a blocking flow in each phase. A blocking flow saturates all edges leaving a layer, ensuring that the level graph's height decreases with each phase. The result is an algorithm that runs in $O(V^2 E)$ time for general graphs and $O(V^{1/2} E)$ for unit capacity graphs.

```

function Dinic(G, s, t):
    flow = 0
    while (buildLevelGraph(G, s, t)):
        while (P = DFSBlockingFlow(G, s, t)):
            bottleneck = min{capacity(e) | e in P}
            for each edge e in P:
                increase flow on e by bottleneck
                decrease flow on reverse edge by bottleneck
            flow += bottleneck
    return flow

```

return flow

Key ideas: level graph, blocking flow, DFS for augmenting paths within the level graph.

Push–Relabel (Preflow Push) Algorithm

The push–relabel approach takes a different tack. Instead of searching for augmenting paths, it maintains a preflow and local “push” operations that move excess flow forward, while “relabel” operations adjust node heights to allow further pushes. The algorithm guarantees termination after at most $O(V^3)$ operations, and the improved highest label variant achieves $O(V^2 + E)$ performance.

```
function PushRelabel(G, s, t):
    initialize preflow: set flow on edges from s to capacity, others zero
    for each vertex v in V \ {s, t}:
        excess[v] = flow into v
    while (exists vertex v with excess[v] > 0):
        if (can push from v to neighbor u):
            push(v, u)
        else:
            relabel(v)
    return sum of flows out of s
```

Push–relabel is especially efficient for dense graphs and large capacities.

Applications of Network Flow

Network flow algorithms extend beyond theoretical interest; they solve practical problems in diverse domains:

- Transportation: Optimizing shipping routes, airline scheduling, and traffic routing.
- Telecommunications: Maximizing bandwidth usage, designing resilient networks.
- Project Management: Resource allocation, task scheduling with constraints.
- Image Segmentation: Treating pixels as nodes, using min cut to separate foreground and background.
- Social Networks: Influence maximization, community detection.

Each scenario can be modeled as a flow network, where capacities represent limits and flows represent assignments or data packets.

Matching Problems in Bipartite Graphs

While flow networks handle general graphs, bipartite matching focuses on pairing vertices from two disjoint sets. The chapter introduces the *assignment problem*, where each pair has a weight, and the goal is to maximize total weight. The Hungarian algorithm solves this in $O(V^3)$ time. For unweighted bipartite graphs, the Hopcroft–Karp algorithm finds a maximum matching in $O(\sqrt{V} E)$ time.

Hungarian Algorithm

The Hungarian algorithm is a combinatorial optimization technique that finds the minimum cost perfect matching in a weighted bipartite graph. It operates by maintaining a feasible labeling of vertices and iteratively improving the

matching until all edges in the equality graph are matched.

```
function Hungarian(matrix):  
    initialize labeling:  $u[i] = \max_j \text{matrix}[i][j]$ ,  $v[j] = 0$   
    for each i:  
        find augmenting path using slack variables  
        update labels  $u$  and  $v$  to maintain feasibility  
        augment matching along found path  
    return matching, total cost
```

Complexity: $O(n^3)$ for an $n \times n$ matrix. The algorithm is robust and widely used in operations research.

Hopcroft–Karp Algorithm

For unweighted bipartite graphs, Hopcroft–Karp finds a maximum cardinality matching efficiently. It performs BFS to build layers and then uses DFS to find multiple augmenting paths in each phase, reducing the number of phases required.

```
function HopcroftKarp(G, U, V):  
    matching = {}  
    while (BFS layering from unmatched U vertices):  
        for each unmatched  $u$  in  $U$ :  
            DFS( $u$ ) to find augmenting paths  
    return matching
```

Complexity: $O(\sqrt{V} E)$. This efficiency makes it suitable for large sparse graphs.

Advanced Topics and Variants

Chapter 7 also touches on several advanced topics that extend the basic algorithms:

- **Capacity Scaling:** A refinement of Ford–Fulkerson that processes edges in decreasing capacity ranges, improving performance to $O(E^2 \log U)$, where U is the maximum capacity.
- **Minimum cost Flow:** Adding cost to each edge and finding the cheapest way to send a given amount of flow, solvable by successive shortest augmenting paths or cycle cancelling methods.
- **Maximum weight Matching in General Graphs:** The Blossom algorithm (Edmonds' algorithm) generalizes matching to non bipartite graphs, handling odd length cycles.
- **Dynamic Flow:** Adjusting flows efficiently when the network changes over time, using incremental algorithms.

These variants demonstrate the flexibility of flow and matching frameworks, allowing them to adapt to constraints like cost, time, or changing network topology.

Problem Solving Tips for Chapter 7

When tackling exercises from Chapter 7, consider the following strategies:

1. **Model the Problem Clearly:** Translate real world constraints into graph terms—vertices, edges, capacities, and costs.

2. Choose the Right Algorithm: If capacities are large and the graph is dense, push–relabel may win; for sparse graphs with unit capacities, Dinic’s algorithm is often fastest.

Baca Juga (Artikel Terkait)

- [chapter 2 economic systems answers](#)

URL: <http://www.bbgglobalsolutions.com/chapter-2-economic-systems-answers.pdf>

- [chapter 2 economic optimization questions answers](#)

URL: <http://www.bbgglobalsolutions.com/chapter-2-economic-optimization-questions-answers.pdf>

- [chapter 2 differentiation test form b](#)

URL: <http://www.bbgglobalsolutions.com/chapter-2-differentiation-test-form-b.pdf>

- [chapter 2 creating a process flowchart](#)

URL: <http://www.bbgglobalsolutions.com/chapter-2-creating-a-process-flowchart.pdf>

Tags: [#chapter](#) [#solutions](#) [#algorithm](#) [#design](#) [#kleinberg](#) [#tardos](#)

