0.1 Bloom Filters

- *Algorithm:* Bloom Filter (algo. 1)
- *Input:* Element x to insert or query, Bloom filter B with bit array of size m, and k hash functions $h_1, h_2, \ldots, h_k$.
- *Complexity:* Insertion: $\mathcal{O}(k)$, Query: $\mathcal{O}(k)$
- *Data structure compatibility:* Can be implemented with bit arrays and hash tables. N/A if unrelated.
- *Common applications:*
  - Network routing protocols for efficient membership testing.
  - Caching and database applications to check membership of data without storing the data itself.
  - Spell checking and web page URL checking to minimize storage.

Problem. Bloom Filter

The Bloom Filter is a probabilistic data structure that provides an efficient way to test whether an element is a member of a set, allowing for false positives but no false negatives.

Description

The Bloom Filter uses multiple hash functions to map elements to a fixed-size bit array, allowing it to efficiently determine membership with reduced space requirements [1].

Input:

- A set of elements to insert into the Bloom filter. - An element to query for membership. - The size of the bit array (m) and the number of hash functions (k).

Complexity:

- Insertion operation: $\mathcal{O}(k)$, where k is the number of hash functions. - Query operation: $\mathcal{O}(k)$.

Applications:

1. Network Routing: Bloom filters are used in routing protocols to quickly determine if a node is part of a network without storing full addresses.

2. Database Caching: They are utilized in databases to check for the existence of records before performing expensive disk lookups.

3. Spell Checking: Used to minimize the number of words stored in a dictionary while still allowing for fast membership tests.

These applications are related in that they all benefit from the space-efficient nature of Bloom filters, making them suitable for scenarios with large datasets where memory constraints are a concern.

Algorithm 1: Bloom Filter Operations

Input : Element x, Bloom filter B with bit array of size m, k hash functions $h_1, h_2, \ldots, h_k$
Output: Updated Bloom filter B after insertion, or query result

```
Function Insert(Element x):
    for i ← 1 to k do
        Compute hash h_i(x);
        Set B[h_i(x)] ← 1;
    end for
    return Bloom filter B;
end
Function Query(Element y):
    for i ← 1 to k do
        if B[h_i(y)] = 0 then
            return Element y is definitely not in the set;
        end if
    end for
    return Element y might be in the set (false positive possible);
end
```

References.

[1] Li Luo, Dong Guo, Richard T.B. Ma, Ori Rottenstreich, and Xiaojun Luo. "Optimizing Bloom Filter: Challenges, Solutions, and Comparisons". In: *IEEE Communications Surveys & Tutorials* 21.2 (2019), pp. 1912–1949. DOI: 10.1109/COMST.2018.2889329 (cit. on p. 1).

0.2 Fibonacci heaps

- *Algorithm:* Fibonacci heaps (algo. 2, algo. 3,algo. 4,algo. 5)
- *Input:* A set of elements to be managed, each with an associated key(positive integers).
- *Complexity:* insertion: $\mathcal{O}(1)$, extractMin: $\mathcal{O}(\log n)$, findMin: $\mathcal{O}(1)$, decreaseKey: $\mathcal{O}(1)$
- *Data structure compatibility:* Priority queues
- *Common applications:* Used in fast algorithms for problems like minimum spanning tree algorithms, and graph algorithms that require efficient priority queue operations.

Problem. Fibonacci heaps

A Fibonacci heap is a data structure that provides a collection of trees that are used to realize a priority queue. It allows for efficient merging of heaps and supports operations such as insertion, deletion, decrease key, and finding the minimum element.

Description

A Fibonacci heap consists of a group of rooted trees that satisfies the min-heap property. So, for the input, each element is represented by a node within a tree, who has a key attribute. The time complexities for various operations in a Fibonacci heap are as follows [1]:

$$\begin{aligned} \text{Insert} &: O(1) \\ \text{FindMin} &: O(1) \\ n\text{ExtractMin} &: O(\log n) \\ \text{DecreaseKey} &: O(1) \\ \text{Delete} &: O(\log n) \\ \text{Union} &: O(1) \end{aligned}$$

- **Insert**: Inserting a node contains adding a node to the root list. So this takes constant time, $O(1)$.
- **FindMin**: We always maintain a pointer to the minimum node, so just constant time is enough, $O(1)$.
- **ExtractMin**: To remove the minimum element, we must reorganize the heap, which requires merging trees and this needs $O(\log n)$ time.
- **DecreaseKey**: Decreasing the key of a node can be done by adjusting pointers and promoting the node to the root list. So this can be done in constant time, $O(1)$.
- **Delete**: To delete a node, we first decrease its key to negative infinity and then perform an extract-min operation, resulting in $O(\log n)$.
- **Union**: Unioning two heaps involves combining their root lists, which can be done in constant time, $O(1)$.

According to [2], Fibonacci heaps can enhance the performance of several graph algorithms through efficient priority queue operations. Below are the main applications fields and the improverments of the time complexities:

- **Single-Source Shortest Paths:** Fibonacci heaps improve Dijkstra's algorithm, reducing the time complexity to:

$$O(E + V \log V)$$

- **All-Pairs Shortest Paths:** Used in Dijkstra's algorithm, the updated time complexity is:

$$O(VE + V^2 \log V)$$

- **Weighted Bipartite Matching:** The Hungarian algorithm can by optimized with Fibonacci heaps, updating the time complexity to:

$$O(VE + V^2 \log V)$$

- **Minimum Spanning Tree:** Prim's algorithm benefits from Fibonacci heaps, resulting in:

$$O(E + V \log V)$$

Pseudo-code template comes from ECE477 FA 2024 slides [3]

Algorithm 2: Fibonacci heaps

Input : A set of elements with associated keys(positive integers)
Output: The minimum element and the updated state of the heap

```
Function Insert(H, x):
    create a new node x with key and value;
    set its degree of x to 0;
    set its children pointer of x to NULL;
    add x to the root list of H;
    update the minimum pointer;
end
Function FindMin(H):
    return the node pointed to by the minimum pointer in heap H
end
```

Algorithm 3: Fibonacci heaps [1]

Input : A set of elements with associated keys(positive integers)
Output: The minimum element and the updated state of the heap

```
Function Delete(H, x):
    set the key of node x to negative infinity;
    call ExtractMin to remove x from the heap;
end
Function Union(H_1, H_2):
    if H_2 is empty then
        return H_1
    end if
    if H_1 is empty then
        return H_2
    end if
    H ← H_1;
    H.min ← min(H_1.min, H_2.min);
    H_1.last.next ← H_2.first;
    H_2.first.prev ← H_1.last;
    H_1.size ← H_1.size + H_2.size;
    return H
end
return
```

Algorithm 4: Fibonacci heaps

Input : A set of elements with associated keys(positive integers)
Output: The minimum element and the updated state of the heap

```
Function DecreaseKey(H, x, k):
    if k < x.parentNode.value then
        set x.key to k;
        if x in the root list then
            update the minimum pointer;
        else
            cut x from its parent node;
            add x to the root list of H;
            update the degree of its parent node;
            set parent node as current node;
            while current node is marked as true do
                cut current node from its parent node;
                add current node to the root list of H;
                set current node's parent as current node;
            end while
            if current node is not in the root list then
                mark it as true;
            end if
        end if
    else
        set x.key to k;
    end if
    return;
end
```

Algorithm 5: Fibonacci heaps

Input : A set of elements with associated keys(positive integers)
Output: The minimum element and the updated state of the heap

```
Function ExtractMin(H):
    identify the minimum node minNode in H;
    remove minNode from the root list;
    if minNode has children then
        add them to the root list of H;
    end if
    Initialize an array to track trees of different degrees;
    for each root in the root list do
        if there exists another tree of the same degree then
            link the two trees together;
            update the degree of the resultant tree;
        end if
    end for
    update the minimum pointer;
    return minNode
end
```

References.

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. Cambridge, MA: The MIT Press, 2009 (cit. on pp. 3, 4).

[2] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *J. ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874. URL: https://doi.org/10.1145/28869.28874 (cit. on p. 3).

[3] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on p. 4).

0.3 Kd-Trees

- *Algorithm:*
  - k-d Tree Construction (algo. 6)
  - Nearest Neighbor Search (algo. 7)
- *Complexity:*
  - Time Complexity:
    - Insertion: O(log n) on average, O(n) in the worst case
    - Search (Nearest Neighbor): O(log n) on average, O(n) in the worst case
    - Deletion: O(log n) on average, O(n) in the worst case
  - Space Complexity: O(n), where n is the number of points stored in the tree
- *Data structure compatibility:* k-d trees can be compatible with the following data structures:
  - Arrays: Used to store the initial set of points
  - Linked Lists: Used for dynamic storage and traversal of points
  - Self-balancing Binary Search Trees (e.g., AVL Trees, Red-Black Trees): Used for maintaining tree balance
  - Other Spatial Partitioning Structures (e.g., R-trees, Ball Trees): Can be combined with k-d trees to optimize specific queries
- *Common applications:*
  - Computational Geometry: Used for range queries, nearest neighbor searches, etc.
  - Machine Learning: Used in K-Nearest Neighbors (KNN) algorithms
  - Computer Graphics: Used for collision detection, ray tracing, etc.
  - Geographic Information Systems (GIS): Used for indexing and retrieving spatial data
  - Robotics: Used for path planning and environmental perception

Problem. Kd-Trees

Given a set of points in a k-dimensional space, construct a k-d tree to facilitate efficient spatial searches. The tree should use median splits along alternating dimensions at each level to ensure balance.

Description

The k-d tree is a binary search tree specifically designed for organizing points in k-dimensional space.

Input: A set of n points, where each point has k coordinates.

Complexity: - Construction time is $O(n \log n)$ on average. - Space complexity is $O(n)$.

Applications: 1. **Nearest Neighbor Search:** Quickly find the closest point to a given target point. This is often used in machine learning and pattern recognition tasks. 2. **Range Searching:** Efficiently retrieve all points within a specific range in multi-dimensional space, applicable in geographic information systems (GIS).

These applications relate to one another in that they often share the same underlying data structure (the k-d tree) to enhance performance in multi-dimensional searches.

Algorithm 6: k-d Tree Construction

Input : A set of points P and depth d
Output: A k-d tree T

```
1  BuildKDTree(P, d) if P is empty then
2  |  return null
3  end if
4  axis ← d mod k;
5  Sort points by the axis;
6  median ← median point of P;
7  node ← new node with point median;
8  node.left ← BuildKDTree(points left of median, d + 1);
9  node.right ← BuildKDTree(points right of median, d + 1);
10 return node;
```

Algorithm 7: Nearest Neighbor Search

Input : Root of k-d tree T, target point P, depth d, best guess *best*
Output: Nearest neighbor point

```
1  NearestNeighbor(T, P, d, best) if T = null then
2  |  return best
3  end if
4  axis ← d mod k;
5  if T.point is closer to P than best then
6  |  best ← T.point;
7  end if
8  if P[axis] < T.point[axis] then
9  |  best ← NearestNeighbor(T.left, P, d + 1, best);
10 |  if distance from P to splitting plane < distance to best then
11 |  |  best ← NearestNeighbor(T.right, P, d + 1, best);
12 |  end if
13 end if
14 else
15 |  best ← NearestNeighbor(T.right, P, d + 1, best);
16 |  if distance from P to splitting plane < distance to best then
17 |  |  best ← NearestNeighbor(T.left, P, d + 1, best);
18 |  end if
19 end if
20 return best;
```

0.4 Subtree isomorphism

- *Algorithm:* Subtree isomorphism (algo. 8)
- *Input:* A host tree T_h and a pattern tree T_p.
- *Complexity:* $O(|T_h| \cdot |T_p|)$ for ordered trees; NP-complete for unordered trees.
- *Data structure compatibility:* Tree representations, e.g., adjacency lists or parent-child node relationships.
- *Common applications:* Computational biology, structured text database querying, and compiler optimization.

Problem. Subtree isomorphism

Given a host tree T_h and a pattern tree T_p, determine whether T_p is isomorphic to a subtree of T_h. That is, find a subtree of T_h whose structure and node relationships are identical to T_p.

Description

Subtree Isomorphism is a classic problem in computer science, where the goal is to determine whether a smaller pattern tree T_p is isomorphic to a subtree of a larger host tree T_h. It has diverse applications in fields such as bioinformatics, database querying, and hierarchical pattern matching [2].

Variants of the Problem

The problem has several variants:

- **Rooted vs. Unrooted Trees**: For rooted trees, the isomorphism must map the roots of T_p and T_h. Unrooted trees remove this constraint, adding complexity.
- **Ordered vs. Unordered Trees**: Ordered trees require that child nodes maintain a specific order, whereas unordered trees do not.
- **Weighted Trees**: Subtree isomorphism can be extended with a weight function to maximize the similarity score between mapped vertices and edges [1].

Complexity and Algorithms

The computational complexity depends on the tree properties:

- For rooted and ordered trees, algorithms based on dynamic programming achieve a complexity of $O(|T_h| \cdot |T_p|)$ [2].
- For unordered trees, the general problem is NP-complete. Recent advancements leverage bipartite graph matching to achieve near-optimal algorithms for specific cases [1].
- For trees of bounded degree Δ, the problem can be solved in $O(|T_h||T_p|\Delta)$, exploiting the structure of matching instances [2].

Matching Process

The core of solving the Subtree Isomorphism problem lies in efficiently matching the structure of the pattern tree T_p to potential subtrees in the host tree T_h. The process often involves breaking down the trees into smaller subtrees and recursively comparing them, leveraging dynamic programming and graph-matching techniques.

For **rooted trees**, the problem can be solved by recursively computing the isomorphism for every pair of subtrees rooted at corresponding nodes in T_p and T_h. Each pair is evaluated based on:

1. **Node Mapping**: Verifying that the roots of the two subtrees can be mapped.

2. **Children Matching**: Using bipartite graph matching to pair the children of the roots optimally. Each edge in the bipartite graph represents the cost or weight of matching a child from T_p with a child from T_h.

This approach ensures that the subtree isomorphism for larger subtrees builds upon the results of smaller ones, achieving $O(|T_h| \cdot |T_p|)$ complexity for rooted and ordered trees [2].

For each pair of nodes being compared, their children are treated as vertices of a **bipartite graph**. Edges between these vertices are weighted based on the similarity or cost of matching their respective subtrees. The Hungarian algorithm or other bipartite matching techniques are then applied to find the maximum weight matching. This step ensures optimal pairing of children, a critical part of achieving efficient subtree isomorphism.

Key optimizations include:

- Precomputing subtree sizes and weights to reduce redundant calculations.

- Using adjacency lists for sparse tree structures to speed up graph construction and traversal [1].

For **unrooted trees**, the challenge is compounded by the need to determine the optimal roots. This requires trying all possible root combinations, increasing the computational overhead to $O(|T_h|^2 \cdot |T_p|)$ in naive implementations. Recent advancements reduce this by leveraging properties such as:

- **Tree Decomposition**: Breaking unrooted trees into simpler rooted subtrees.

- **Shared Subproblems**: Avoid redundant calculations by maintaining a shared table of previously solved subtree isomorphisms [1, 2].

When dealing with **weighted trees**, the matching process incorporates a weight function $w(v, u)$ for each vertex pair (v, u) and $w(e, f)$ for each edge pair (e, f). The goal is to maximize the total weight of the isomorphism. Matching instances in this context are solved using advanced techniques like:

- Scaling algorithms for weighted bipartite graphs.

- Primal-dual approaches that efficiently handle large weights [2].

The combination of bipartite matching, dynamic programming, and graph-theoretic optimizations ensures that the matching process is both efficient and scalable, making it suitable for large real-world trees in applications like molecular pattern matching and hierarchical data analysis [2, 1].

Applications

The problem finds applications in:

- **Bioinformatics**: Comparing phylogenetic trees and molecular structures.
- **Database Systems**: Querying hierarchical data structures such as XML or JSON.
- **Pattern Recognition**: Identifying template matches in visual data [1].

Recent Advances

Recent research has explored output-sensitive algorithms that enumerate all maximum common subtree isomorphisms efficiently. For example, combining polynomial-delay enumeration with advanced bipartite matching algorithms reduces complexity to $O(n^3 + Tn\Delta)$, where T is the number of solutions [2]. Additionally, lower bounds based on the assignment problem highlight the limits of further optimization for general trees [1].

Details of the algorithm are further elaborated in Algorithm 8 [3].

Algorithm 8: Subtree Isomorphism

Input : Host tree T_h, pattern tree T_p, weight function w (optional)
Output: True if T_p is isomorphic to a subtree of T_h, False otherwise

Function `MaxWeightMatching(`*childrenA*, *childrenB*, *weights*`)`**:**
  Construct a bipartite graph with *childrenA* and *childrenB* as vertex sets;
  Add edges between all pairs (u, v) with weights based on *weights*;
  return the maximum weight matching;
end
Function `RecursiveMatch(`*subtreeA*, *subtreeB*`)`**:**
  if *subtreeA and subtreeB are empty* **then**
    return True;
  end if
  if *only one of subtreeA or subtreeB is empty* **then**
    return False;
  end if
  if *root of subtreeA* $\neq$ *root of subtreeB* **then**
    return False;
  end if
  childrenA $\leftarrow$ children of root in *subtreeA*;
  childrenB $\leftarrow$ children of root in *subtreeB*;
  weights $\leftarrow$ similarity scores between child pairs;
  matching $\leftarrow$ `MaxWeightMatching(`*childrenA*, *childrenB*, *weights*`)`;
  for *each matched pair* (u, v) *in matching* **do**
    if *not* `RecursiveMatch(`*subtreerooted at* u, *subtreerooted at* v`)` **then**
      return False;
    end if
  end for
  return True;
end
Function `SubtreeIsomorphism(`T_h, T_p, w`)`**:**
  for *each node* v *in* T_h **do**
    if `RecursiveMatch(`*subtreerooted at* v, T_p`)` **then**
      return True;
    end if
  end for
  return False;
end

References.

[1] Amir Abboud, Arturs Backurs, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Or Zamir. "Subtree Isomorphism Revisited". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2016, pp. 1256–1271. DOI: 10.1137/1.9781611974331.ch90 (cit. on pp. 8, 9).

[2] Andre Droschinsky, Nils M. Kriege, and Petra Mutzel. "Faster Algorithms for the Maximum Common Subtree Isomorphism Problem". In: *arXiv preprint arXiv:1602.07210* (2016). https://doi.org/10.48550/arXiv.1602.07210 (cit. on pp. 8, 9).

[3] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on p. 9).

0.5 All-pairs shortest path

- *Algorithm:* Floyd-Warshall algorithm (algo. 9)
- *Input:* A weighted graph with vertices and edges, represented by an adjacency matrix where the weight of an edge is the distance between the vertices.

- *Complexity:* $\mathcal{O}(n^3)$, where n is the number of vertices in the graph.
- *Data structure compatibility:* It Can be implemented using an adjacency matrix or adjacency list
- *Common applications:* Finding the shortest paths between all pairs of nodes in a graph is widely used in routing algorithms, traffic network analysis.

Problem. All-pairs shortest path

The goal of the problem is that given a weighted graph, we need to find the shortest paths between all pairs of vertices. If no edge exists between two vertices, the weight is set to infinity. The problem is solved by computing the shortest paths between each pair of vertices using dynamic programming.

Description

Problem Summary

The All-Pairs Shortest Path , also called APSP, is a problem that aims to find out the shortest paths between every pair of nodes in a graph. The problem can be solved by various algorithms, which is shown in the graph below:

Algorithm	**Time Complexity**	**Space Complexity**
Floyd-Warshall	$O(n^3)$	$O(n^2)$
Johnson's Algorithm	$O(n^2 \log n + n^3)$	$O(n^2)$
Dijkstra (Re-run for each vertex)	$O(n^2)$ or $O((n+e)\log n)$	$O(n^2)$
Bellman-Ford (Re-run for each vertex)	$O(n \cdot e)$	$O(n^2)$
Matrix Multiplication	$O(n^3 \log n)$	$O(n^2)$
Dantzig's Algorithm	$O(n^3)$	$O(n^2)$

Table 1: All-Pairs Shortest Path Algorithms Comparison[1]

Algorithm	**Graph Types**
Floyd-Warshall	Directed/undirected, positive/negative weights
Johnson's Algorithm	Directed, positive/negative weights
Dijkstra (for each vertex)	Directed/undirected, no negative weights
Bellman-Ford (for each vertex)	Directed, handles negative weights
Matrix Multiplication	Directed/undirected, integer weights
Dantzig's Algorithm	Directed/undirected

Table 2: All-Pairs Shortest Path Algorithms Comparison[2]

Algorithm	**Remarks**
Floyd-Warshall	Simple to implement, but can not deal with negative cycles
Johnson's Algorithm	Faster for sparse graphs, but can not deal with negative cycles
Dijkstra (for each vertex)	Suitable for sparse graphs, but can not solve negative edges
Bellman-Ford (for each vertex)	Slower than Dijkstra, but works with negative edges wihtout cycles
Matrix Multiplication	Suitable for dense graphs, based on matrix multiplication
Dantzig's Algorithm	Another variant, related to dynamic programming

Table 3: All-Pairs Shortest Path Algorithms Comparison[3]

Floyd-Warshall algorithm

Since the Floyd-Warshall algorithm is a classic algorithm for solving the all-pairs shortest path problem. We mainly focus on this algorithm in this project.

Algorithm Summary

The Floyd-Warshall algorithm is based on **dynamic programming**. Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges with associated weights, the algorithm incrementally takes each vertex as an intermediary, updating the shortest paths between pairs of vertices by determining whether passing through an intermediate vertex results in a shorter path length.

This ensures that the shortest path between any two vertices is computed, even though the shortest path need to go through other intermediate vertices.

Basic Steps

- Creating **D**, an initial adjacency matrix of the graph, to represent the initial path lengths between each pair of vertices(infinity if no direct edge exists).
- The algorithm considers each vertex as an intermediary and checks whether treating this vertex as a "middle" node leads to a shorter path than the current shortest one between two other vertices.
- In each iteration, after dealing with k intermediary vertices, the algorithm always tries to update the shortest path between i and j by considering the k+1-th vertex as an intermediary.

Core idea

For dynamic programming, recurrence relation is a vital part of the algorithm. In Floyd-Warshall algorithm, we are always trying to find out whether take k as intermediate point will shorten the shortest path. Just as the following figure shows, we can set the problem into two parts based on k, find than the shortest path for i to k, and k to j, then add them together.

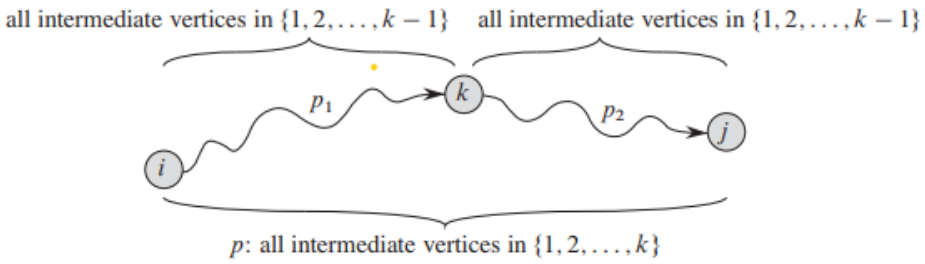

Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

Figure 1: Dynamic Programming for Floyd-Warshall algorithm [1]

So, the recurrence relation can be concluded as [1]:

$$D[i][j] = min(D[i][j], D[i][k] + D[k][j])$$

Where:

- $D[i][j]$ represents the shortest path from vertex i to vertex j.
- k is the current intermediary vertex.

Complexity Type	Best Case	Average Case	Worst Case
Time Complexity	$O(V^3)$	$O(V^3)$	$O(V^3)$
Space Complexity	$O(V^2)$	$O(V^2)$	$O(V^2)$

Table 4: Time and Space Complexity

Complexity

The time complexity of the Floyd-Warshall algorithm is $\mathcal{O}(n^3)$, where n is the number of vertices. This is because the algorithm consists of three nested loops that iterate over all pairs of vertices, with each loop taking linear time in relation to the number of vertices [2].

For space complexity, we need to create a 2-D matrix that can be used to record the shortest distances among each pair of nodes. So, the size of the matrix is V x V, where V is the number of vertices [2].

Pseudo-code template comes from ECE477 FA 2024 slides [3]

Algorithm 9: Floyd-Warshall algorithm

Input : A weighted graph with vertices V and edges E represented by an adjacency matrix D
Output: The shortest path matrix D where $D[i][j]$ is the shortest path distance between vertices i and j

```
Function FloydWarshall(D):
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i][j] ← min(D[i][j], D[i][k] + D[k][j])
            end for
        end for
    end for
    return D
end
```

References.

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. Cambridge, MA: The MIT Press, 2009 (cit. on p. 12).

[2] GeeksforGeeks. *Time and Space Complexity of Floyd-Warshall Algorithm*. Accessed: 2024-11-19. 2024. URL: https://www.geeksforgeeks.org/time-and-space-complexity-of-floyd-warshall-algorithm/ (cit. on p. 13).

[3] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on p. 13).

0.6 Clique problem

- *Algorithm:* Born-Kerbosch algorithm (algo. 10)
- *Input:* Graph $G = (V, E)$, where V is the set of vertices and E is the set of edges.
- *Complexity:* $\mathcal{O}(3^{\frac{n}{3}})$ in the worst case, where n is the number of vertices
- *Data structure compatibility:* Adjacency matrix/list
- *Common applications:* Social network analysis, bioinformatics (protein interaction networks), network security (clique-based community detection)

Problem. Clique problem

The clique problem includes detecting cliques in a given graph. A clique is defined as a subset of vertices, where each pair of vertices in it is connected by an edge. The clique task is often, in a given graph, to find either the largest clique or all cliques of a specific size.

Description

A clique in an undirected graph is a subset of vertices such that every two distinct vertices are adjacent.

Given the graph $G = (V, E)$ where:

- $V = \{A, B, C, D, E\}$
- $E = \{(A, B), (A, C), (B, C), (C, D), (D, E)\}$

A clique of size 3 in this graph would be $\{A, B, C\}$, as every pair of vertices within this subset is connected by an edge.

The problem can be divided into various forms, mainly including identifying cliques of a specified size or to find the largest clique in a given graph.

Algorithm	**Description**
Bron-Kerbosch [2]	Using backtracking to find all maximal cliques in undirected graphs.
Greedy Algorithm [1]	A heuristic that tries to build a clique by adding vertices iteratively.
Simulated Annealing [5]	A probabilistic algorithm that explores different subsets of vertices.
Integer Linear Programming (ILP) [7]	Taken it as an optimization problem using integer programming.
Coloring-based Algorithm [3]	Using graph coloring techniques to find cliques.

Table 5: Summary of Algorithms for the Clique Problem[1]

Algorithm	**Time Complexity**
Bron-Kerbosch [2]	$\mathcal{O}(3^{n/3})$
Greedy Algorithm [1]	$\mathcal{O}(n^2)$
Simulated Annealing [5]	Varies with graph
Integer Linear Programming (ILP) [7]	
Coloring-based Algorithm [3]	Varies with the graph

Table 6: Summary of Algorithms for the Clique Problem[2]

Algorithm	**Notes**
Bron-Kerbosch [2]	Efficient for small graphs, but in the worst case it will be exponential.
Greedy Algorithm [1]	Faster than exact algorithms but can not guarantee the largest clique.
Simulated Annealing [5]	Just a heuristic, it can not be guaranteed to find the largest one.
Integer Linear Programming (ILP) [7]	May not scale well for very large graphs.
Coloring-based Algorithm [3]	Generally faster for sparse graphs, but can be difficult to implement.

Table 7: Summary of Algorithms for the Clique Problem[3]

Difficulty of Clique Problem

The problem is NP-complete, which means solving it exactly for large graphs is difficult, the proof is as follow:

The Clique Problem is NP-hard

The Clique Problem is NP-hard, which means no known algorithm that can solve it efficiently in all cases (unless P = NP). The proof with 3-SAT Problem [6] is as follows:

The **3-SAT Problem** is the following decision problem: given a Boolean formula ϕ in conjunctive normal form (CNF) where each clause contains exactly 3 literals, determine if there exists a truth assignment that satisfies the formula.

Reduction from 3-SAT to Clique

Given: A 3-SAT formula ϕ with m clauses and n variables. We need to detemine whether there exists a satisfying assignment for ϕ.

Construct a graph $G = (V, E)$ as follows:

- **Vertices:** For each clause C_j in ϕ, create 3 vertices, corresponding to the 3 literals in that clause.
- **Edges:** Add an edge between two vertices if and only if there is an edge between x_i and $\neg x_i$, but no edge between x_i and x_i, or between $\neg x_i$ and $\neg x_i$.
- **Clique Size k:** Set $k = m$, where m is the number of clauses in the 3-SAT formula.

Verification of the Reduction

We now need to prove that the 3-SAT formula ϕ is satisfiable if and only if the constructed graph G has a clique of size k.

- **If ϕ is satisfiable:** If ϕ is satisfiable, that means each clause contains at least one true literal. so for each clause C_j, we can select the true literal in the satisfying assignment. This selection gives us one vertex from each clause. Since no two selected literals contradict each other , the corresponding vertices in the graph will form a complete subgraph, i.e., a clique of size m. Thus, the graph contains a clique of size m.
- **If there is a clique of size k in G:** If there exists a clique of size m in the graph, then we can select one vertex from each of the m clauses. These vertices correspond to literals that do not contradict each other This selection of literals satisfies the formula ϕ. Hence, ϕ is satisfiable.

The Clique Problem is in NP

We know that a problem is in NP if given a proposed solution, we can verify whether it is correct in polynomial time.

For the Clique Problem, a proposed solution is a subset of k vertices in the graph. To verify if these k vertices form a clique, we simply need to check if every pair of vertices in the subset is connected by an edge. This can be done by checking all pairs of vertices in the subset, which requires $O(k^2)$ time. Since $k \leq |V|$, this verification can be done in polynomial time. Therefore, the Clique Problem is in NP.

The Clique Problem is in NP and is NP-hard. So, the Clique Problem is NP-complete.

Born-Kerbosch Algorithm

Here, I focus on Bron-Kerbosch Algorithm to solve all-cliques problem problems and Cliquer for the maximum-clique problem.

The **Born-Kerbosch Algorithm** is an algorithm that is based on **recursive backtracking** and can be described as follows [2] [**geeksforgeeks'bronkerbosch**]:

1. define three sets:
    - **R**: contains the vertices that have already been included in the current maximal clique.
    - **P**: contains the vertices that still have the chance to be put into the clique (i.e., the vertices that are connected to all vertices in R).
    - **X**: contains the vertices that have already been included in some maximal clique (preventing duplicates during iterations).

2. For each vertex v in the set P, add v to the current set R. We can guarantee that the graph still remains a clique becasue every vertex in R is connected to every other vertex in R. After adding v, update P by putting only the vertices that are connected to both v and every vertex in R.

3. Then do backtracking, removing v from P and add it to the set X.

4. Iterate until a set R forms a maximal clique(algorithm ends) when both P and X are empty.

Pseudo-code template comes from ECE477 FA 2024 slides [6]

Algorithm 10: Bron-Kerbosch Algorithm

Input : A graph with vertices V and edges E
Output: A set of all maximal cliques in the graph

```
1  Function BronKerbosch(R, P, X):
2      if P = ∅ and X = ∅ then
3          return R ;
4      end if
5      foreach v ∈ P do
6          P' ← P ∩ N(v) ;
7          X' ← X ∩ N(v) ;
8          BronKerbosch(R ∪ {v}, P', X') ;
9          P ← P \ {v} ;
10         X ← X ∪ {v} ;
11     end foreach
12 end
```

References.

[1] K. L. Berman, M. K. Gavril, and R. K. Guy. "Heuristic algorithms for finding the largest clique in a graph". In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. 1981, pp. 80–85 (cit. on p. 14).

[2] C. Bron and J. Kerbosch. "Finding all cliques in an undirected graph". In: *Communications of the ACM* 16.9 (1973), pp. 575–577. DOI: 10.1145/362248.362272 (cit. on pp. 14, 15).

[3] M. L. Fredman and D. E. Willard. "Trans-dichotomous algorithms for minimum spanning trees and shortest paths". In: *SIAM Journal on Computing* 19.5 (1990), pp. 952–971. DOI: 10.1137/S0097539791364416 (cit. on p. 14).

[5] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671 (cit. on p. 14).

[6] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on pp. 14, 16).

[7] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003. ISBN: 978-3540003791 (cit. on p. 14).

0.7 Color coding

- *Algorithm:* Color coding (algo. 11)
- *Input:* A graph $G = (V, E)$, target subgraph H with k vertices.
- *Complexity:* $O(e^k \cdot n^c)$, where c depends on the specific graph traversal technique.
- *Data structure compatibility:* Adjacency matrix, adjacency list, and hash tables (for efficient vertex coloring and mapping).
- *Common applications:* Path and cycle detection, motif discovery in biological networks, graph mining.

Problem. Color coding

Given a graph $G = (V, E)$ and a target subgraph H with k vertices, determine if G contains H as a subgraph. Specifically, identify colorful embeddings of H, where the vertices of H are mapped to vertices of G with distinct colors.

Description

Color coding is a randomized algorithmic technique designed to efficiently find small substructures, such as paths, cycles, or tree-like motifs, within large graphs. Originally introduced by Alon, Yuster, and Zwick in 1995, the algorithm has become a foundational tool for solving subgraph isomorphism problems, particularly when the target subgraph has a fixed size k and bounded treewidth [1].

Key Idea

The color coding algorithm is a clever and efficient technique designed to simplify the detection of small substructures in large graphs by converting the problem into a "colorful" subgraph detection task. It combines randomness and combinatorics to reduce the computational complexity of identifying specific patterns like paths, cycles, or motifs in graphs.

Random Coloring: The algorithm begins by assigning k colors randomly to the vertices of the graph G. A subgraph H is considered **colorful** if all its vertices are assigned distinct colors. This transformation allows the algorithm to focus on a smaller, constrained problem of finding colorful embeddings of H within G [1].

Colorful Subgraph Detection: The key insight is that detecting colorful subgraphs is computationally simpler than searching for arbitrary subgraphs. By leveraging dynamic programming or matrix multiplication techniques, colorful copies of the target subgraph H can be efficiently identified in G. For instance, for detecting a colorful path of length k, the algorithm recursively builds paths by extending shorter colorful paths with new vertices [4].

Repetition for Success: Since a single random coloring might not produce a colorful embedding of H, the algorithm is repeated $O(e^k)$ times to ensure a high probability of success. The probability of failure decreases exponentially with the number of iterations, making the randomized approach both reliable and efficient in practice [2].

Derandomization: To eliminate the randomness, a deterministic version uses k-perfect hash families, which systematically explore all possible colorings. This guarantees that every k-vertex subset of G will be colorful for at least one coloring in the hash family [1].

Advantages of Color Coding

1. **Efficient Search**: Reduces the search space by focusing on colorful subgraphs, making it computationally tractable for small k.

2. **Modularity**: Can be combined with other graph algorithms, such as dynamic programming or fast matrix multiplication, for added efficiency.

3. **Generality**: Applicable to various subgraph detection problems, including paths, cycles, and tree-like motifs [4].

Complexity

The complexity of the color coding algorithm is influenced by the size of the target subgraph k, the structure of the graph G, and the method of implementation (randomized or deterministic).

Randomized Complexity

In the randomized version:

The algorithm runs in $O(e^k \cdot n^c)$, where: - e^k represents the number of iterations required to ensure a high probability of success (each iteration has a $\sim e^{-k}$ chance of missing a colorful subgraph).

- n^c accounts for the cost of searching for colorful subgraphs, typically achieved using dynamic programming or recursive methods, with c depending on the graph's structure and the traversal technique [1, 2].

Deterministic Complexity

In the derandomized version:

- The algorithm constructs a k-perfect hash family to replace random coloring. The size of the family is $O(e^k \cdot \text{polylog}(n))$, which increases preprocessing time but guarantees coverage of all possible colorings [1].

- The total runtime for the deterministic version becomes $O(e^k \cdot n^c \cdot \log(n))$, making it slightly more expensive but eliminating the probability of failure.

Special Cases

1. **Subgraphs with Bounded Treewidth**: - For fixed treewidth t, the algorithm can exploit tree decompositions, reducing the complexity to $O(2^t \cdot e^k \cdot n)$, which is efficient for small t [2].

2. **Path Detection**: - Detecting paths of length k benefits from matrix multiplication optimizations, achieving $O(2^{O(k)} \cdot n^\omega)$, where ω is the matrix multiplication exponent (currently $\omega < 2.37$) [4].

Applications

Color coding has a wide range of applications in graph theory and beyond:

- **Bioinformatics**: Identifying motifs in protein-protein interaction networks, where small, tree-like patterns are biologically significant.

- **Graph Mining**: Detecting specific paths, cycles, or substructures in social or communication networks.

- **Theoretical Computer Science**: Solving parameterized problems, such as finding paths or cycles of fixed length, that are otherwise computationally expensive [1, 2].

Recent Advances

Modern implementations extend color coding to efficiently enumerate subgraphs, incorporating dynamic programming techniques to reduce time and memory requirements. For example, algorithms now handle larger treewidth values by exploiting the properties of tree decompositions and incorporating matrix multiplication for colorful path detection [4, 2]. These improvements have made color coding a practical tool for large-scale graph analysis and pattern detection.

Details of the algorithm are further elaborated in Algorithm 11 [3].

Algorithm 11: Color Coding

Input : Graph $G = (V, E)$, target subgraph H of size k
Output: True if H is a subgraph of G, False otherwise

```
Function IsColorful(H, coloring):
    foreach vertex v in H do
        if color of v in coloring is not unique then
            return False;
        end if
    end foreach
    return True;
end
Function FindColorfulSubgraph(G, H, coloring):
    Initialize DP table dp of size |V(G)| × k;
    foreach vertex v in G do
        if vertex v matches the corresponding vertex in H under coloring then
            dp[v][1] ← 1;
        end if
    end foreach
    for i ← 2 to k do
        foreach subset S of G of size i do
            foreach vertex v in S do
                if S \ {v} forms a valid subgraph under coloring then
                    dp[S][i] ← ∑_{u∈S\{v}} dp[S \ {v}][i − 1];
                end if
            end foreach
        end foreach
    end for
    foreach subset S of G of size k do
        if S forms a colorful subgraph of H then
            return True;
        end if
    end foreach
    return False;
end
Function ColorCoding(G, H, k):
    for i ← 1 to O(e^k) do
        coloring ← Assign random colors to vertices of G from 1 to k;
        if FindColorfulSubgraph(G, H, coloring) then
            return True;
        end if
    end for
    return False;
end
Function GeneratePerfectHashFamily(n, k):
    Initialize an empty hash family;
    Use combinatorial techniques to generate a family of hash functions;
    return hash family;
end
Function ColorCodingDeterministic(G, H, k):
    hashFamily ← GeneratePerfectHashFamily(|V(G)|, k);
    foreach coloring ∈ hashFamily do
        if FindColorfulSubgraph(G, H, coloring) then
            return True;
        end if
    end foreach
    return False;
end
```

References.

[1] Noga Alon, Raphael Yuster, and Uri Zwick. "Color-coding". In: *Journal of the ACM (JACM)* 42.4 (1995), pp. 844–856. DOI: 10.1145/210332.210337 (cit. on pp. 18, 19).

[2] Tomáš Valla Josef Malík Ondřej Suchý. "Efficient Implementation of Color Coding Algorithm for Subgraph Isomorphism Problem". In: *arXiv preprint arXiv:1908.11248* (2019). Available at https://arxiv.org/abs/1908.11248 (cit. on pp. 18, 19).

[3] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on p. 19).

[4] Wikipedia contributors. "Color coding (algorithm)". In: *Wikipedia* (2024). Available at https://en.wikipedia.org/wiki/Color-coding (cit. on pp. 18, 19).

0.8 Dulmage-Mendelsohn decomposition

- *Algorithm:* Dulmage-Mendelsohn decomposition (algo. 12)
- *Input:* A bipartite graph $G = (X, Y, E)$, where X and Y are the two sets of vertices and E is the set of edges.
- *Complexity:* Time complexity: $\mathcal{O}(n^2)$, where n is the number of vertices in the graph.
- *Data structure compatibility:* Can be implemented using adjacency matrices or adjacency lists for the bipartite graph.
- *Common applications:*
  - Maximum matching problem in bipartite graphs.
  - Community detection in social network analysis.
  - Resource allocation problems, such as task scheduling and job assignments.

Problem. Dulmage-Mendelsohn Decomposition

The Dulmage-Mendelsohn Decomposition is a structural analysis method for bipartite graphs or sparse matrices. It partitions the graph or matrix into distinct subsets that reveal unmatched elements, matched components, and over-constrained structures, providing insights into system solvability and numerical properties.

Description

The Dulmage-Mendelsohn Decomposition partitions a bipartite graph or sparse matrix into distinct subsets, which include fully matched components, unmatched elements, and over-constrained structures. This decomposition provides valuable insights into the structure of the matrix, aiding in solving optimization problems, network flow analysis, and improving the efficiency of algorithms for matching and assignment problems [1].

Input:

- A bipartite graph $G = (X, Y, E)$, where X and Y are the two vertex sets, and E is the set of edges between them. - The objective is to decompose G into various components, including matched, unmatched, and over-constrained parts.

Complexity:

- Decomposition operation: $\mathcal{O}(n^2)$, where n is the number of vertices in the graph. - The complexity depends on the specific structure of the graph and the decomposition algorithm used.

Applications:

1. **Maximum Matching**: Used in solving the maximum matching problem for bipartite graphs, such as in job assignments and resource allocation.

2. **Network Flow Analysis**: Helps in analyzing flow problems in bipartite networks, such as transportation or communication networks.

3. **Resource Allocation**: Applied in problems like task scheduling or project assignment, where matching elements from two sets is required.

These applications all benefit from the ability of the Dulmage-Mendelsohn Decomposition to break down complex bipartite graphs into manageable subsets, improving the efficiency and clarity of optimization and matching algorithms.

Algorithm 12: Dulmage-Mendelsohn Decomposition

Input : Bipartite graph $G = (U, V, E)$ or matrix A
Output: Partition of rows and columns into subsets: unmatched, matched, and over-determined

Function `MaxMatching(`*Graph* G`):`
    Find a maximum matching M of G;
    Mark all matched nodes in U and V;
    return *Matching* M*, matched nodes*;
end

Function `Reachable(`*Graph* G*, Matching* M`):`
    Identify free (unmatched) nodes in U and V;
    Perform alternating path search starting from free nodes to determine reachable nodes;
    return *Set of reachable and non-reachable nodes*;
end

Function *`DulmageMendelsohnDecomposition`Graph* G**:**
    $M \leftarrow$ `MaxMatching(`G`)`;
    $(R_U, R_V) \leftarrow$ `Reachable(`G, M`)`;
    Partition nodes in U and V into subsets:
    - Unmatched: Nodes in U or V not part of any matching.
    - Matched: Nodes in U and V fully matched by M.
    - Over-determined: Nodes reachable via alternating paths but not fully matched.

    Reorder rows and columns of A based on the partition;
    return *Partitioned graph or reordered matrix* A;
end

References.

[1] Li Luo, Dong Guo, Richard T.B. Ma, Ori Rottenstreich, and Xiaojun Luo. "Optimizing Bloom Filter: Challenges, Solutions, and Comparisons". In: *IEEE Communications Surveys & Tutorials* 21.2 (2019), pp. 1912–1949. DOI: 10.1109/COMST.2018.2889329 (cit. on p. 21).

0.9 Graduation Problem

- *Algorithm:* Graduation Prediction Model (algo. 13)
- *Input:* A student with academic records, including grades, completed courses, and external factors such as financial or health status.
- *Complexity:* Enrollment: $\mathcal{O}(1)$, Check eligibility: $\mathcal{O}(m)$ (where m is the number of courses), Graduation prediction: $\mathcal{O}(m)$
- *Data structure compatibility:* Student records, course catalogs, and eligibility criteria.
- *Common applications:* Used in academic institutions for evaluating students' progress towards graduation, determining which students are at risk of not graduating, and predicting graduation timelines.

Problem. Graduation Problem

The Graduation Problem involves determining whether a student meets all academic and external criteria required to graduate. It takes into account factors such as completed coursework, grades, extracurricular activities, and potentially external factors like health or financial issues. The challenge is to predict, based on current academic standing, whether a student will be able to graduate on time or identify any risks that could delay graduation.

Description

The Graduation Problem involves evaluating whether a student meets the academic and external criteria necessary to graduate. The problem can be modeled as a bipartite graph or a system where one set represents the academic requirements (such as completed courses or grades) and the other set represents the students. This decomposition reveals students who are fully meeting the requirements, those who are missing some requirements, and those at risk due to external factors, such as health or financial issues. The method allows for a structured approach to assess a student's graduation status and predict their ability to graduate on time.

Input:

- A set of students and their academic records (courses completed, grades achieved). - External factors such as health or financial support that could influence graduation eligibility. - A process to assess whether the student meets graduation requirements, such as coursework completion or meeting minimum grade thresholds.

Complexity:

- Assessment operation: Varies based on the number of courses, grades, and external factors considered, with a typical complexity of $\mathcal{O}(n)$ where n is the number of students or courses. - Graduation prediction operation: Depends on the number of criteria and the method used, often involving classification or decision-tree-based analysis.

Applications:

1. Academic Advising: The Graduation Problem helps academic advisors assess students' graduation readiness, providing guidance on whether they need to complete additional coursework or fulfill other requirements.

2. Early Warning Systems: It is used in systems that track student progress and identify those at risk of not graduating on time, allowing interventions before it's too late.

3. Curriculum Optimization: By identifying students who face challenges in meeting graduation requirements, universities can optimize their curricula and provide targeted support.

These applications are connected in that they all utilize the decomposition method to reveal students' academic and external situation, making it easier to predict graduation outcomes and optimize support systems for at-risk

students.

Algorithm 13: Graduation Prediction Model

Input : Student S with grades, course completion, external factors (e.g., financial support, health)
Output: Prediction of graduation (yes/no)

Function `EnrollCourse`(*Student* S*, Course* C)**:**
    If student S has passed prerequisites for C, enroll in course C;
    return *Updated student record with new course enrollment*;
end

Function `CompleteRequirements`(*Student* S)**:**
    For each required course C for graduation: **if** *Student* S *has not completed* C **then**
        return *Student has unmet requirements*
    end if
    return *All graduation requirements met*;
end

Function `CheckEligibility`(*Student* S)**:**
    If the student has completed all required courses and has satisfactory grades: **return** *Student is eligible for graduation*;
    Otherwise: **return** *Student is not eligible for graduation*;
end

Function `PredictGraduation`(*Student* S)**:**
    if `CompleteRequirements`(S) *is successful and* `CheckEligibility`(S) *is true* **then**
        return *Yes, Student* S *will graduate*;
    end if
    return *No, Student* S *will not graduate*;
end

0.10 Matching

- *Algorithm:* Matching (algo. 14 15 16)
- *Input:* A graph $G = (V, E)$, optionally with weights on edges, and the specific type of matching problem (e.g., maximum matching, maximum weight matching, or stable matching).
- *Complexity:*
  - $O(m\sqrt{n})$ for bipartite maximum matching using the Hopcroft-Karp algorithm, where n is the number of vertices and m is the number of edges [4].
  - $O(n^3)$ for general graph maximum matching using Edmonds' Blossom algorithm [1].
  - $O(n^3)$ for maximum weight matching in bipartite graphs using the Hungarian algorithm [3].
- *Data structure compatibility:* Adjacency list or adjacency matrix for graph representation; priority queues for weighted matching.
- *Common applications:* Resource allocation, job assignment, stable marriage, network design, bioinformatics, and scheduling.

Problem. Matching

Given a graph $G = (V, E)$, the goal is to find a subset of edges $M \subseteq E$ such that no two edges in M share a vertex. Additional objectives may include maximizing the size of M, maximizing the total weight of edges in M, or ensuring stability based on preferences.

Description

The matching problem is a cornerstone of graph theory and combinatorial optimization. A matching M in a graph $G = (V, E)$ is defined as a subset of edges where no two edges share a common vertex. The problem has several variants, each addressing different objectives and constraints:

- **Maximum Matching**: Finds the largest matching in terms of the number of edges [4].

- **Maximum Weight Matching**: Optimizes the total weight of the matching in weighted graphs [3].

- **Perfect Matching**: Ensures all vertices in the graph are matched without overlap.

- **Stable Matching**: Satisfies stability conditions based on vertex preferences, commonly used in applications like the stable marriage problem [1].

Key Algorithms

1. **Greedy Matching**:

- This simple algorithm iteratively selects edges that do not conflict with existing edges in the matching, ensuring that no two edges share a vertex. While it is fast and straightforward, it does not guarantee an optimal solution for maximum or weighted matching.

- **Strengths**: Useful for quick approximations or large-scale problems where exact solutions are computationally prohibitive.

- **Weaknesses**: Suboptimal for maximum matching as it does not account for augmenting paths.

- **Time Complexity**: $O(m)$, where m is the number of edges.

2. **Hopcroft-Karp Algorithm**:

- This algorithm computes a maximum cardinality matching in bipartite graphs by repeatedly finding and augmenting along shortest augmenting paths.

- **Approach**: Uses BFS to find layers of augmenting paths and DFS to augment along these paths efficiently.

- **Time Complexity**: $O(m\sqrt{n})$, where n is the number of vertices and m is the number of edges.

3. **Edmonds' Blossom Algorithm**:

- Extends matching to non-bipartite graphs by identifying and collapsing odd-length cycles (blossoms), enabling the detection of augmenting paths.

- **Applications**: Used in scenarios requiring maximum matching in general graphs, such as network routing and resource allocation [3].

- **Innovations**: Introduced the concept of blossom contraction, which significantly advanced the theory of matchings.

- **Time Complexity**: $O(n^3)$.

4. **Hungarian Algorithm**:

- Solves the assignment problem by finding maximum weight matchings in bipartite graphs using a combination of augmenting paths and dual variables.

- **Approach**: Uses a primal-dual method to ensure feasibility and optimality of the solution at each step.

- **Applications**: Widely used in job assignment, scheduling, and transportation networks [1].

- **Time Complexity**: $O(n^3)$.

5. **Gale-Shapley Algorithm**:

- Computes stable matchings in bipartite graphs based on preference lists for two sets of vertices (e.g., students and schools, or workers and jobs).

- **Approach**: Uses a deferred acceptance mechanism to iteratively refine proposals and rejections until a stable configuration is reached.

- **Applications**: Central to problems in market design, such as the stable marriage problem, college admissions, and organ matching [1, 3].

- **Time Complexity**: $O(n^2)$, where n is the total number of participants.

Applications

Matching problems are integral to a variety of domains:

- **Resource Allocation**: Assigning workers to jobs or resources to tasks.

- **Network Design**: Constructing efficient communication or transport networks.

- **Scheduling**: Matching tasks to time slots or resources.

- **Bioinformatics**: Identifying molecular patterns or motifs.

- **Market Design**: Solving the stable marriage problem and other allocation challenges.

The matching problem's versatility stems from its ability to model real-world scenarios, making it a foundational tool in algorithmic design [1, 4, 3].

Details of the algorithm are further elaborated in Algorithm 14 15 16 [2].

Algorithm 14: Hopcroft-Karp Algorithm

Input : A bipartite graph $G = (U, V, E)$
Output: Maximum matching M

```
Function BFS(G, pairU, pairV, dist):
    queue ← empty queue;
    foreach u ∈ U do
        if pairU[u] = NIL then
            dist[u] ← 0;
            queue.enqueue(u);
        end if
        else
            dist[u] ← ∞;
        end if
    end foreach
    dist[NIL] ← ∞;
    while queue is not empty do
        u ← queue.dequeue();
        if dist[u] < dist[NIL] then
            foreach v ∈ neighbors[u] do
                if dist[pairV[v]] = ∞ then
                    dist[pairV[v]] ← dist[u] + 1;
                    queue.enqueue(pairV[v]);
                end if
            end foreach
        end if
    end while
    return dist[NIL] ≠ ∞;
end
Function DFS(u, pairU, pairV, dist):
    if u ≠ NIL then
        foreach v ∈ neighbors[u] do
            if dist[pairV[v]] = dist[u] + 1 then
                if DFS(pairV[v], pairU, pairV, dist) then
                    pairV[v] ← u;
                    pairU[u] ← v;
                    return True;
                end if
            end if
        end foreach
        dist[u] ← ∞;
        return False;
    end if
    return True;
end
Function HopcroftKarp(G):
    pairU[u] ← NIL for all u ∈ U;
    pairV[v] ← NIL for all v ∈ V;
    dist ← empty map;
    matching ← 0;
    while BFS(G, pairU, pairV, dist) do
        foreach u ∈ U do
            if pairU[u] = NIL then
                if DFS(u, pairU, pairV, dist) then
                    matching ← matching + 1;
                end if
            end if
        end foreach
    end while
    return matching;
end
```

Algorithm 15: Hungarian Algorithm

Input : Weighted bipartite graph $G = (U, V, E, w)$, where w is the weight function.
Output: Maximum weight matching M

```
Function AugmentMatching(path, M):
    foreach edge ∈ path do
        if edge ∈ M then
            M.remove(edge);
        end if
        else
            M.add(edge);
        end if
    end foreach
    return M;
end
Function HungarianAlgorithm(G):
    M ← ∅;
    labels ← initial labels for vertices based on edge weights;
    while there exists an augmenting path do
        Find an augmenting path using dual variables;
        M ← AugmentMatching(path, M);
    end while
    return M;
end
```

Algorithm 16: Gale-Shapley Algorithm

Input : Two sets U and V with preference lists for all members.
Output: Stable matching M

```
Function GaleShapley(U, V):
    M ← empty matching;
    freeMen ← U;
    while freeMen is not empty do
        u ← any man in freeMen;
        v ← first woman on u's preference list;
        if v is free then
            M.add(u, v);
            freeMen.remove(u);
        end if
        else if v prefers u over her current match u' then
            M.remove(u', v);
            M.add(u, v);
            freeMen.add(u');
            freeMen.remove(u);
        end if
        else
            u removes v from his preference list;
        end if
    end while
    return M;
end
```

References.

[1] L. Lovasz and M. D. Plummer. *Matching Theory.* Referenced in course notes from Stanford MS&E 319. 1986 (cit. on pp. 26, 27).

[2] Manuel. *ECE477 – Introduction to Algorithms (lecture slides).* 2023 (cit. on p. 27).

[3] Unknown. "Matching problems — Combinatorial Optimization Class Notes". In: *Fiveable Library* (2024). https://library.fiveable.me/combinatorial-optimization/unit-2/matching-problems/study-guide/zvoer62r3O1Xhk86 (cit. on pp. 26, 27).

[4] Wikipedia contributors. *Matching (graph theory)*. https://en.wikipedia.org/wiki/Matching_(graph_theory). 2024 (cit. on pp. 26, 27).

0.11 A* Search

- *Algorithm:* A* Search (algo. 17)
- *Input:* A graph with nodes, edges, associated costs, a start node, a goal node, and a heuristic function estimating the cost from a node to the goal.
- *Complexity:* In the worst case, $\mathcal{O}(b^d)$, where b is the branching factor and d is the depth of the solution.
- *Data structure compatibility:* Priority queue (or min-heap) for frontier; hash set for explored nodes.
- *Common applications:* Pathfinding in games, robotics, network routing, AI planning.

Problem. A* Search

Find the shortest path from a given start node to a goal node in a graph, where each edge has a cost, and the algorithm uses a heuristic function to estimate the remaining distance to the goal, thus making A* search an informed search.

Description

A* Search is a widely-used informed search algorithm, commonly categorized under best-first search techniques. It is designed to efficiently find the shortest path in a weighted graph, combining the advantages of both Dijkstra's algorithm and Greedy Best-First Search. The algorithm begins at a specified start node and seeks to reach a given goal node while minimizing the total cumulative cost, which can represent factors like distance, time, or resource consumption. A* is particularly powerful because it integrates a heuristic that provides problem-specific information to guide the search process [10].

The main idea behind A* is to maintain a tree of paths originating from the start node, where at each iteration, the most promising path is extended based on a combination of the cost incurred so far and an estimate of the remaining cost to the goal. At each step, A* selects the path to extend based on the minimization of the following function [2]:

$$f(n) = g(n) + h(n)$$

where:

- n is the current node,
- $g(n)$ is the cost incurred to reach n from the start node,
- $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal node.

A* is an optimal search algorithm provided the heuristic function $h(n)$ is **admissible**, meaning that it never overestimates the actual cost to the goal [8]. This ensures that the algorithm will return the least-cost path.

Priority Queue and Search Process

A* uses a priority queue, often called the open set or frontier, to explore nodes in the order of their estimated total cost $f(n)$. At each iteration, the node with the lowest $f(n)$ value is expanded first, and its neighboring nodes are evaluated. The algorithm terminates when the goal node is removed from the queue, at which point the $f(n)$-value of the goal represents the cost of the shortest path found [4].

To trace the actual path, A* maintains a record of each node's predecessor. When the goal is reached, the sequence of nodes from the start to the goal can be reconstructed by following these predecessor pointers back to the start node [5].

Heuristics and Admissibility

The performance of A* heavily depends on the choice of the heuristic function $h(n)$, which should be problem-specific. A* guarantees optimality when the heuristic is admissible. For example:

- In pathfinding on geographical maps, $h(n)$ could represent the Euclidean (straight-line) distance from the current node n to the goal, which acts as a lower bound for the true path cost [10].
- In grid-based systems, commonly used in video game maps or robotics, heuristic functions like the **Taxicab distance** (Manhattan distance) or the **Chebyshev distance** are used depending on whether movement is restricted to 4 or 8 directions [1].

Consistency (Monotonicity) of Heuristics

A* is also guaranteed to find an optimal solution if the heuristic is **consistent (or monotone)**. A heuristic is consistent if, for any two adjacent nodes x and y, the following inequality holds:

$$h(x) \leq d(x, y) + h(y)$$

where $d(x, y)$ is the true cost to move from node x to node y [3].

The consistency property is crucial in making A* efficient. When a heuristic is consistent, the algorithm is guaranteed to expand each node at most once. This reduces the computational overhead that would otherwise occur if nodes were reprocessed, as seen in search algorithms that lack this property. In contrast, when a heuristic is only admissible (but not consistent), A* may revisit nodes, leading to additional overhead in maintaining and updating paths, thus potentially slowing down the search process. In this case, A* behaves like Dijkstra's algorithm but with the additional benefit of guiding the search using the heuristic, thus speeding up the process [4].

Applications

A* is widely used in many domains, including robotics, gaming, and geographic information systems.

- In robotics, it enables pathfinding and navigation for autonomous systems, where efficient real-time decisions are critical [6].
- In video games, A* is often the go-to algorithm for non-player character (NPC) movement in complex environments.
- Similarly, in GIS (Geographic Information Systems), A* is used for route optimization by considering distance, traffic, and terrain [9].

The flexibility of A* lies in its ability to adjust to various problems by modifying the heuristic function. A well-suited heuristic can greatly reduce the search space, improving the algorithm's efficiency without sacrificing

optimality [10].

The relation between the heuristic and the actual problem space significantly affects the performance of A*. For instance, in pathfinding, the heuristic can drastically reduce the number of nodes explored by providing accurate distance estimates.

Details of the algorithm are further elaborated in Algorithm 17 [7].

Algorithm 17: A* Search

Input : Graph G=(V, E) with edge cost, start node s, goal node g, heuristic function h(n)
Output: The path from start node to goal node with the smallest cost

```
Function cost(current, neighbor):
    return the cost between current and neighbor ;
end
Function Reconstruct(cameFrom, current):
    totalPath ← {current} ;
    while current in cameFrom do
        current ← cameFrom[current] ;
        totalPath.prepend(current) ;
    end while
    return totalPath ;
end
Function A*Search(G, s, g, h):
    frontier ← {s} ;
    exploredSet ← {} ;
    gScore[s] ← 0 ;
    gScore[v] ← ∞ for all v ≠ s ;
    fScore[s] ← h(s) ;
    fScore[v] ← ∞ for all v ≠ s ;
    cameFrom ← an empty map that indicates the parent of the node;
    while frontier is not empty do
        current ← node in frontier with lowest fScore[current] ;
        if current = g then
            return ReconstructPath(cameFrom, current) ;
        end if
        frontier.remove(current) ;
        exploredSet.add(current) ;
        for each neighbor of current do
            if neighbor in exploredSet then
                continue ;
            end if
            newgScore ← gScore[current] + cost(current, neighbor) ;
            if neighbor not in frontier then
                frontier.add(neighbor) ;
            end if
            if newgScore >= gScore[neighbor] then
                continue ;
            end if
            cameFrom[neighbor] ← current ;
            gScore[neighbor] ← newgScore ;
            fScore[neighbor] ← gScore[neighbor] + h(neighbor) ;
        end for
    end while
    return failure ;
end
```

References.

[1] *A* Search Algorithm*. https://en.wikipedia.org/wiki/A*_search_algorithm. Accessed: 2024-10-17 (cit. on p. 31).

[2] *Course Materials from VE492 - Introduction to Algorithms*. Available online (cit. on p. 30).

[3] Rina Dechter and Judea Pearl. "Generalized Best-First Search Strategies and the Optimality of A*". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 505–536 (cit. on p. 31).

[4] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107 (cit. on p. 31).

[5] Sven Koenig, Maxim Likhachev, and David Furcy. "Lifelong Planning A*". In: *Artificial Intelligence* 155 (2004), pp. 93–146 (cit. on p. 31).

[6] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006 (cit. on p. 31).

[7] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on p. 32).

[8] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984 (cit. on p. 30).

[9] Thomas K Peucker and Nicholas R Chrisman. "Cartographic Data Structures". In: *American Cartographer* 5.1 (1978), pp. 55–69 (cit. on p. 31).

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall, 2010 (cit. on pp. 30–32).

0.12 Edge Detection

- *Algorithm:* Canny Edge Detector (algo. 18)
- *Input:* Grayscale image
- *Complexity:* $\mathcal{O}(n \log n)$, where n is the number of pixels
- *Data structure compatibility:* 2D arrays (images)
- *Common applications:* Computer vision, image processing, object detection, and medical imaging.

Problem. Edge Detection

Edge detection is a commonly used technique for image processing, mainly used for identifying and locating sharp discontinuities. Often, its output is a binary image that highlights the edges found in the input image.

Description

Edge detection is crucial in various fields, including image processing as well as computer vision. The primary goal is to identify significant transitions in pixel intensity, which indicate object edges or variations in the orientation of surface.

The input is typically a grayscale image represented as a 2D array of pixel intensities.

Common applications include:

Medical Imaging: Identifying structures in X-rays or MRIs.

Autonomous Vehicles: Detecting road edges and obstacles.

Facial Recognition: Locating facial features for identification.

Here, I choose canny edge detector to finish the algorithm, so we will only focus on that in the rest part. The steps for the algorithm is as follows [1]

Table 8: Common Edge Detection Algorithms [3]

Algorithm	Description
Canny	Multi-stage edge detector that uses Gaussian smoothing and hysteresis thresholding.
Sobel	Uses convolution with Sobel operators to find gradients.
Prewitt	Similar to Sobel, but with different convolution kernels.
Laplacian	Detects edges by finding areas of rapid intensity change.

- **Step 1: Conduct Gaussian filter** The filter functions by applying convolution with a Gaussian kernel to the image, smoothing the image and diminishing high-frequency noise that could lead to false edges For the 2D Gaussian function, which includes both x and y, we have :

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

According to the function, we have Gaussian kernel as a matrix:

$$G = \begin{bmatrix} g(0,0) & g(0,1) & g(0,2) \\ g(1,0) & g(1,1) & g(1,2) \\ g(2,0) & g(2,1) & g(2,2) \end{bmatrix}$$

- **Step 2: Compute gradients using Sobel operators** The Sobel operator is utilized to determine the gradient of the image along the x-direction (G_x) and the y-direction (G_y). The horizontal Sobel kernel (G_x) is defined as:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The vertical Sobel kernel (G_y) is defined as:

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The overall gradient magnitude G is calculated using the formula for the Euclidean norm, which provides the strength of the edges in the image.

$$\sqrt{G\_x^2 + G\_y^2}$$

The gradient direction θ is calculated by the 'atan2' function, which gives the angle of the gradient vector.

$$\theta = \text{atan2}(G_y, G_x)$$

- **Step 3: Non-maximum suppression** The process is implemented to reduce edge thickness by checking each pixel's gradient strength and only keeping the local maximum in the direction of the gradient. This effectively eliminates pixels that are not part of an edge.

- **Step 4: Hysteresis thresholding** This step uses two thresholds to decide which edges are strong, weak, or non-edges. We implement two thresholds: one high and one low. Pixels whose gradient strength surpasses the high threshold are considered strong edges, while those beneath the low threshold should be eliminated. Pixels within the interval of the two thresholds are identified as weak edges. They should be kept in the final edge map when linked to strong edges.

Pseudo-code template comes from ECE477 FA 2024 slides [2]

Algorithm 18: Canny Edge Detector

Function `Canny(`*image*`)`**:**
    smoothed $\leftarrow$ GaussianFilter(*image*) ;
    $G_x, G_y \leftarrow$ Sobel(*smoothed*) ;
    $G \leftarrow \sqrt{G_x^2 + G_y^2}$;
    $\theta \leftarrow \text{atan2}(G_y, G_x)$;
    nonMaxSuppressed $\leftarrow$ NonMaxSuppression(G, θ) ;
    edges $\leftarrow$ HysteresisThreshold(*nonMaxSuppressed*) ;
    return *edges* ;
end

References.

[1] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851 (cit. on p. 33).

[2] Manuel. *ECE477 – Introduction to Algorithms (lecture slides)*. 2023 (cit. on p. 34).

[3] "Recent advances on image edge detection: A comprehensive review". In: *Neurocomputing* 503 (2022), pp. 259–271. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2022.06.083. URL: https://www.sciencedirect.com/science/article/pii/S0925231222008141 (cit. on p. 34).

0.13 JPEG (Encoding and Decoding)

- *Algorithm:* JPEG (Encoding and Decoding) (algo. 19 20 21 22)
- *Input:* Raw image data(Encoding) or JPEG-compressed image data(Decoding)
- *Complexity:* $\mathcal{O}(n \log n)$
- *Data structure compatibility:* Image matrices, DCT coefficients, Huffman coding trees.
- *Common applications:* Image compression in digital photography, web images, and other multimedia applications.

Problem. JPEG (Encoding and Decoding)

JPEG is a lossy image compression technique aimed at reducing file size by transforming the image data into frequency components using the Discrete Cosine Transform (DCT) and quantizing them. The problem involves encoding an image into a compressed format that can later be decoded back into an approximation of the original.

Description

JPEG (Joint Photographic Experts Group) compression is a widely adopted lossy compression technique for digital images, primarily in the domain of photography and web applications. The primary objective of JPEG compression is to reduce file size while maintaining a visually acceptable image quality. This reduction is crucial for efficient storage and transmission of images over the web and in resource-constrained devices.

The algorithm operates by dividing an image into non-overlapping blocks of 8x8 pixels. Each block undergoes a series of transformations to compress the data. The general process of JPEG Encoding and Decoding is shown in the following Figure 2[2].

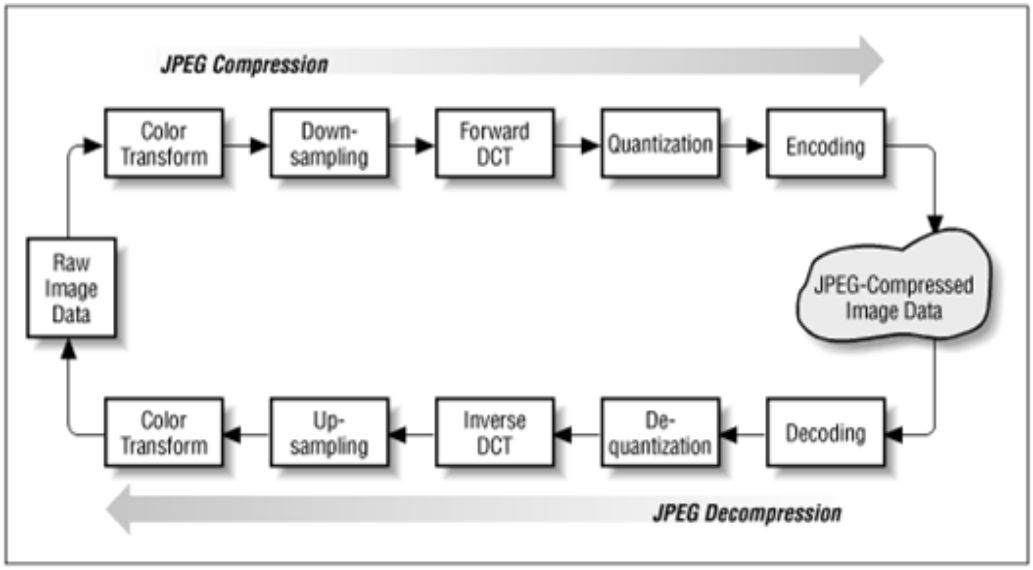

Figure 2: JPEG Process

JPEG Encoding:

Color Transform:

The input raw image data (typically in RGB format) is first converted into a different color space, most commonly YC_bC_r. This separates the image into one luminance channel (Y) and two chrominance channels (C_b and C_r). The Y channel contains brightness information, while C_b and C_r contain color information. This step is important because the human eye is more sensitive to brightness than to color, allowing for more aggressive compression in chrominance channels [6]. The algorithm is shown in the following 19.

Algorithm 19: Color Transform

Input : Image in RGB color space, where each pixel has R , G, and B components.
Output: Image in YC_bC_r color space, where each pixel has Y, C_b, and C_r components.

Function `RGBtoYCbCr`(R, G, B)**:**
  foreach *pixel in the image* **do**
    $$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B;$$
    $$C_b = 128 - 0.168736 \cdot R - 0.331264 \cdot G + 0.5 \cdot B;$$
    $$C_r = 128 + 0.5 \cdot R - 0.418688 \cdot G - 0.081312 \cdot B;$$
  end foreach
  return Transformed image with Y, C_b, and C_r values ;
end

Forward Discrete Cosine Transform (DCT):

The next step is to apply the DCT to 8x8 blocks of pixels for each channel. DCT is highly efficient in capturing image redundancy and makes subsequent compression easier [1]. The DCT converts the spatial information (pixel intensities) into frequency components. In simpler terms, the DCT represents each block of the image as a sum of cosine waves of varying frequencies and amplitudes. Most of the important visual information is contained in the low-frequency components. The algorithm is shown in the following 20.

Algorithm 20: Forward Discrete Cosine Transform (DCT)

Input : An 8x8 block of image data in the spatial domain, denoted by $f(x, y)$ where $x, y = 0, 1, ..., 7$.
Output: An 8x8 block of DCT coefficients in the frequency domain, denoted by $F(u, v)$ where $u, v = 0, 1, ..., 7$.

Function `DCT`$(f(x, y))$**:**
  for $u = 0$ **to** 7 **do**
    for $v = 0$ **to** 7 **do**
      $$F(u, v) = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x, y)\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$
      $$\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & u = 0 \\ 1, & u \neq 0 \end{cases} \qquad \alpha(v) = \begin{cases} \frac{1}{\sqrt{2}}, & v = 0 \\ 1, & v \neq 0 \end{cases}$$
    end for
  end for
  return 8x8 block of DCT coefficients $F(u, v)$;
end

Quantization:

After DCT, the frequency coefficients are quantized. A quantization table controls the degree of compression, and adjusting this table can balance the trade-off between file size and image quality [4]. This step reduces

the precision of less important high-frequency components (i.e., those that represent fine details in the image), which the human eye is less likely to notice. Quantization is the primary source of lossy compression in JPEG. Each coefficient is divided by a value from a quantization table and rounded to reduce the amount of data.

Encoding:

The quantized values are then encoded using entropy encoding techniques like Huffman coding or arithmetic coding. This step further reduces the file size by using shorter codes for more frequently occurring values and longer codes for less frequent values. The algorithm is shown in the following 21.

Algorithm 21: Huffman Coding

Input : An array of original symbols with their frequencies
Output: Encoded array and Huffman encoding dictionary

```
1  Function GetCode (root, code):
2      if root.byte ≠ null then
3          code ← code + root.num;
4          dict[byte] ← code;
5          code.clear();
6          return dict;
7      end if
8      else
9          code ← code + root.num;
10         dict ← getCode(root.left, code);
11         dict ← getCode(root.right, code);
12     end if
13     return dict;
14 end
15 Function HuffmanCoding():
16     Scan the array, get the frequency of each byte;
17     Store the result in a new list arrFreq, where each entry has fields: freq, num, and byte;
18     while arrFreq.num() ¿ 1 do
19         Sort arrFreq by frequency (highest to lowest);
20         Create a new node n0;
21         Pop the two nodes with the smallest frequencies from arrFreq: n1, n2;
22         n0.freq ← n1.freq + n2.freq;
23         n0.byte ← null;
24         n0.left ← n1, n0.right ← n2;
25         n1.num ← 0, n2.num ← 1;
26         Insert n0 into arrFreq;
27     end while
28     root ← arrFreq[0];
29     code ← emptyString();
30     dict ← getCode(root, code);
31     Scan the array and replace each character with dict[char], generating the encodedArr;
32     return encodedArr and dict;
33 end
```

The result of this step is the compressed image data in JPEG format. This step further reduces the file size by eliminating redundancies in the quantized data, achieving high compression ratios [5].

JPEG Decoding:

Decoding:

JPEG decoding begins by reversing the entropy encoding process, which involves undoing the Huffman or arithmetic encoding applied during compression. This step retrieves the compressed data stream and reconstructs the quantized Discrete Cosine Transform (DCT) coefficients for each 8x8 pixel block of the image. During this

process, the bitstream is parsed, and the original symbol frequencies used in encoding are leveraged to recover the compressed image data. The entropy decoding effectively restores the coefficients to their compressed but quantized form, which can then be used for further reconstruction steps [6]. The algorithm is shown in the following 22.

Algorithm 22: Huffman Decoding

Input : Encoded array and Huffman encoding dictionary
Output: Decoded array (original symbols)

```
1  Function Decode (encodedArr, root):
2      currentNode ← root;
3      decodedArr ← empty list;
4      foreach bit in encodedArr do
5          if bit = 0 then
6              currentNode ← currentNode.left;
7          end if
8          else
9              currentNode ← currentNode.right;
10         end if
11         if currentNode.byte ≠ null then
12             Append currentNode.byte to decodedArr;
13             currentNode ← root;
14         end if
15     end foreach
16     return decodedArr;
17 end
18 Function HuffmanDecoding():
19     encodedArr ← the encoded message;
20     dict ← Huffman encoding dictionary;
21     Build the Huffman tree based on dict and assign root;
22     decodedArr ← Decode (encodedArr, root);
23     return decodedArr;
24 end
```

Dequantization:

Once the quantized coefficients are retrieved, the next step is dequantization. In this process, each coefficient is multiplied by the corresponding value in the quantization table used during compression. This operation attempts to restore the DCT coefficients closer to their original values, although some precision is lost due to rounding during the quantization stage. The use of a carefully designed quantization table is crucial, as it allows for high compression ratios while maintaining perceptual quality in the image. However, dequantization cannot reverse the lossy nature of JPEG compression, and some fine details may remain permanently discarded [4].

Inverse DCT (IDCT):

After dequantization, the frequency domain representation of the image (i.e., the DCT coefficients) is converted back into the spatial domain using the Inverse Discrete Cosine Transform (IDCT). Each 8x8 block of DCT coefficients is transformed back into an 8x8 block of pixel intensities. The result is a rough approximation of the original pixel values, with some degradation due to the loss of higher frequency components, which represent fine details. The IDCT step effectively reconstructs the image block by block, although highly compressed images may exhibit visible artifacts such as blocking or blurring [1].

Color Transform:

The final step in the JPEG decoding process is the color space transformation. During compression, images are typically converted from the RGB color space to the YC_bC_r color space, which separates luminance (Y) from chrominance (C_b and C_r) information. This color space is used because the human eye is more sensitive

to changes in brightness than color, allowing greater compression of chrominance data. After the IDCT step, the image is converted back from the YC_bC_r color space to the original RGB format (or another color format depending on the source image). This transformation ensures that the decompressed image can be displayed correctly on devices that use the RGB color model, such as monitors and cameras [3]. At this stage, the image is fully reconstructed and ready for display, although some loss of quality remains, primarily due to quantization and lossy compression techniques.

References.

[1] Nasir Ahmed, T Natarajan, and KR Rao. "Discrete cosine transform". In: *IEEE Transactions on Computers* 100.1 (1974), pp. 90–93 (cit. on pp. 38, 40).

[2] Liu Yang. *JPEG Encoder and Decoder*. https://blog.csdn.net/l_yangliu/article/details/7228984. [Online; accessed 13-October-2024]. 2024 (cit. on p. 37).

[3] Sanjit K Mitra and Giovanni L Sicuranza. *Nonlinear Image Processing*. Elsevier, 2000 (cit. on p. 41).

[4] William B. Pennebaker and Joan L. Mitchell. *JPEG: Still Image Data Compression Standard*. Springer Science & Business Media, 1992 (cit. on pp. 38, 40).

[5] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2017 (cit. on p. 39).

[6] Gregory K. Wallace. "The JPEG still picture compression standard". In: *Communications of the ACM* 34.4 (1991), pp. 30–44 (cit. on pp. 38, 40).