Course: Advanced Algorithms by Dr. Zarei

Homework: HW4

Name: Mohammad Mohammadi

Student ID: 402208592

Question 1

Exercise 1.7 Consider the following problem. A shipping company has to decide how to distribute a load consisting of a n containers over its ships. For 1 ≤ i ≤ n, let wi denote the weight of container i. The ships are identical, and each ship can carry containers with a maximum total weight W. (Thus if C(j) denotes the set of containers assigned to ship j, then (sum of wi)≤W for all i in C(j)) The goal is to assign containers to ships in such a way that a minimum number of ships is used. Give a 2-approximation algorithm for this problem, and prove the approximation ratio of your algorithm. Hint: There is a very simple greedy algorithm that gives a 2-approximation.

Answwer 1

We can use a simple greedy approach known as the "Next Fit" algorithm. Here's how it works:

1. Sort the containers in decreasing order of their weights (from heaviest to lightest).
2. Start with the first ship and begin loading containers sequentially from the sorted list.
3. If the current container does not fit in the current ship (i.e., adding it would exceed the weight limit W), close this ship and start loading a new ship.
4. Repeat this process until all containers are assigned to ships.

This algorithm is a 2-approximation because, in the worst case, it uses no more than double the number of ships than the optimal solution. To understand why, consider the heaviest container. No two of these can be in the same optimal ship because their combined weight would exceed W. Thus, each ship in the optimal solution must have a container as heavy as the one that caused the Next Fit algorithm to start a new ship. Therefore, the number of ships used by Next Fit is at most twice the number of ships in the optimal solution.

Question 2

Exercise 2.1 Let G = (V,E) be an (undirected) graph. A verter cover of G is a subset C of V such that for every edge (u,v) € E we have u € C or v € C (or both). An independent set of G is a subset I of V such that no two vertices in I are connected by an edge in E.

1. Prove the following statement: C is a vertex cover of G if and only if V \ C is an independent set of G.
2. It follows from (1) that if we can compute a vertex cover for G, we can also compute an independent set for G. Now suppose we want to compute a maximum-size independent set on G,

and suppose we have a 2-approximation algorithm ApproxMinVertexCover for finding a minimum-size vertex cover. Consider the following algorithm for computing a

---

```
ApproxMaxIndependentSet(G)

1:  C <- ApproxMinVertexCover(G)        > G=(V,E) is an undirected graph

2: return V\\C
```

Prove or disprove: ApproxMaxIndependentSet is a 2-approximation algorithm.

Answer 2

Part 1.

Proving the Statement "C is a vertex cover of G if and only if V \ C is an independent set of G"

- A vertex cover in a graph G is a set c of vertices such that every edge in G has at least one of its endpoints in c. This implies that for every edge (u,v) in G, at least one of u or v is in c.
- Now, consider the set V \ c, which is the complement of the vertex cover. If this set were not an independent set, it would mean there exists an edge in G with both endpoints in V \ c. However, this contradicts the definition of a vertex cover, as both endpoints of this edge would not be in c. Thus, V \ c must be an independent set.
- Conversely, if V \ c is an independent set, then no edge in G has both its endpoints in this set. Hence, at least one endpoint of every edge must be in c, making c a vertex cover.

Part 2.

ApproxMaxIndependentSet as a 2-approximation algorithm

- The integrality gap of the integer linear programming formulation for the minimum vertex cover problem is 2. This suggests that the linear programming relaxation, which is a 2-approximation algorithm, provides a vertex cover that is at most twice the size of the minimum vertex cover.
- Given this, the algorithm ApproxMaxIndependentSet(G), which returns V \ c where C is the vertex cover obtained by ApproxMinVertexCover(G), will yield an independent set. This independent set, being the complement of a vertex cover that is at most twice the size of the minimum, should intuitively be no smaller than half the size of the maximum independent set (since the maximum independent set and minimum vertex cover are complementary in size).
- Therefore, ApproxMaxIndependentSet can be considered a 2-approximation algorithm for the maximum independent set problem.

Question 3

Exercise 3.7 Vertex Cover is NP-complete, so there is no polynomial-time algorithm that solves Vertex Cover optimally unless P=NP. Prove that this implies that there is no FPTAS for Vertex Cover unless P=NP. (You are not allowed to use the fact mentioned in the Course Notes that Vertex Cover cannot be approximated to within a factor 1.3606 unless P=NP; your proof that an FPTAS does not exist should only be based on the fact that Vertex Cover is NP-complete.)

Answer 3

To solve this question, which is proving that there is no Fully Polynomial-Time Approximation Scheme (FPTAS) for Vertex Cover unless P=NP, we need to build upon the fact that Vertex Cover is NP-complete. The proof of NP-completeness for Vertex Cover involves showing that it is in NP and that it is NP-hard, by reducing a known NP-complete problem, such as 3-SAT, to Vertex Cover in polynomial time.

This reduction not only proves that Vertex Cover is NP-hard but also that it is NP-complete since it is already in NP. This implies that, unless P=NP, no polynomial-time algorithm can solve Vertex Cover optimally, and hence, no FPTAS can exist for Vertex Cover.

The proof of why there is no Fully Polynomial-Time Approximation Scheme (FPTAS) for Vertex Cover, given that Vertex Cover is NP-complete, can be constructed by leveraging the properties of NP-completeness and the definition of FPTAS.

1. NP-Completeness of Vertex Cover: Vertex Cover is a well-known NP-complete problem, which implies that, unless P=NP, no polynomial-time algorithm can solve it exactly.

2. Definition of FPTAS: An FPTAS is an approximation scheme for optimization problems that can produce, for any $\epsilon>0$, a solution that is within a factor of $(1+\epsilon)$ of being optimal, in time polynomial in the input size and $1 / \epsilon$.

3. Implications of NP-Completeness: If an NP-complete problem had an FPTAS, it would imply that this problem can be approximated to any degree of accuracy in polynomial time. This, in turn, would suggest that an exact solution could be found in polynomial time by setting ϵ to a sufficiently small value. This is because as ϵ approaches zero, the approximation becomes exact.

4. Contradiction: Since the exact solution to an NP-complete problem cannot be found in polynomial time (unless P=NP), the existence of an FPTAS would contradict the definition of NP-completeness. Therefore, we can conclude that no FPTAS can exist for Vertex Cover unless P=NP.

This proof structure relies solely on the definition of NP-completeness and FPTAS.

Question 4

Prove the following lemma and based on it, propose a 3/2-approximation algorithm for TSP when the underlying graph supports triangle inequality.

Lemma 4.5 Let G = (V, E) be a graph and let V* be a subset of V containing an even number of vertices. Let OPT denote the minimum length of any tour on G, and let M* be a perfect matching on the complete graph G* = (V* , E*), where the lengths of the edges in E* are equal to the lengths of the corresponding edges in E. Then the length of M* is less than or equal to one half of OPT.

Answer 4

To prove this lemma, one would typically argue that since M* is a perfect matching, it pairs up vertices such that the sum of the distances between paired vertices is minimized. Because the tour represented by OPT must at least once visit each vertex in V, *the distance of OPT must be at least twice the distance of M*, since each edge in M* can be seen as a shortcut between visits to its endpoints in OPT. This is under the assumption that the graph satisfies the triangle inequality, which ensures that direct travel between two points is never more costly than a path with additional stops in between.

Based on this lemma, a 3/2-approximation algorithm for the Traveling Salesman Problem (TSP) when the underlying graph supports triangle inequality can be constructed as follows:

1. Compute a minimum spanning tree (MST)T of G.
2. Double every edge in T, which creates an Eulerian graph (since all vertices now have an even degree).
3. Find an Eulerian tour of this graph.
4. Convert the Eulerian tour into a Hamiltonian cycle (a TSP tour) by skipping repeated vertices (shortcutting, which does not increase the total length due to the triangle inequality).
5. Return this Hamiltonian cycle as the solution to the TSP.

The approximation ratio comes from the fact that the cost of the MST is a lower bound for OPT, and the cost of the doubled MST is, at most, twice the MST. When converting the Eulerian tour to a Hamiltonian cycle, the cost does not increase due to the triangle inequality, ensuring that the total cost is at most (3/2 OPT) since the longest path in the MST is at most the length of OPT, and the addition of shortcuts to create the Hamiltonian cycle can at most add half of the OPT (as per the lemma provided).

Question 5

In the MAX-CUT problem, we are given an unweighted undirected graph G = (V, E). We define a cut (S, V - S) as in Chapter 23 and the weight of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that for each vertex v, we randomly and independently place v in S with probability 1/2 and in V - S with probability 1/2. Show that this algorithm is a randomized 2-approximation algorithm

Answer 5

To show that the given algorithm is a randomized 2-approximation algorithm for the MAX-CUT problem, we need to demonstrate that the expected weight of the cut produced by the algorithm is at least half the weight of the maximum cut. Here's how we can argue this:

1. Understanding the Expected Weight of a Cut: Let's consider an edge e = (u, v) in E. The probability that u and v are in different sets (which means the edge e contributes to the cut) is 1/2, because we place each vertex in S or V - S independently with equal probability. Since there are |E| edges, and each contributes to the cut with probability 1/2, the expected number of edges in the cut is |E|/2.

2. Maximum Weight of a Cut: Let W be the weight (or the number of edges) of the maximum cut in graph G.
3. Expected Weight of the Random Cut: The expected weight of the cut produced by the algorithm is |E|/2. Since the maximum cut cannot have more edges than the total number of edges in the graph, we have W ≤ |E|.
4. Comparison with the Maximum Cut: The expected weight of the cut produced by the algorithm is half the total number of edges |E|/2. Since the maximum cut W is at most |E|, the expected weight of the algorithm's cut is at least W/2. Therefore, the expected weight of the cut is at least half the weight of the maximum cut.
5. Concluding the 2-approximation: Because the expected weight of the cut is at least W/2, the algorithm gives a cut whose expected weight is at least half of the maximum cut. This means that on average, the cut found by this randomized algorithm has a weight that is at least half the weight of an optimal cut. Hence, the algorithm is a 2-approximation algorithm, as it guarantees an expected approximation ratio of 2.

It's important to note that while the expected weight of the cut is at least half of the optimal cut, any single run of the algorithm may produce a cut that is better or worse than this expectation. However, over multiple runs, the average performance of the algorithm will adhere to this 2-approximation ratio.

Double-click (or enter) to edit

Question 6

Two rooted trees T1 and T2 are said to be isomorphic if there exists a one-to-one onto mapping f from the vertices of T1 to those of T2 satisfying the following condition: for each internal vertex v of T1 with the children v1, ..., vk, the vertex f(v) has as children exactly the vertices f(v1), ..., f(vk). Observe that no ordering is assumed on the children of any internal vertex. Devise an efficient randomized algorithm for testing the isomorphism of rooted trees and analyze its performance. (Hint: Associate a polynomial Pv with each vertex v in a tree T. The polynomials are defined recursively, the base case being that the leaf vertices all have P = x0. An internal vertex v of height h with the children v1, ..., vk has its polynomial defined to be

(xh - Pv1)(xh - Pv2)...(xh - Pvk).

Note that there is exactly one indeterminate for each level in the tree.)

Answer 6

To solve the isomorphism problem for two rooted trees T1 and T2 using the given hint, we can use an algorithm based on polynomial association. Here is explanation of such an algorithm:

1. Initialization:
    - For each tree T1 and T2, start by assigning a unique polynomial to all leaf nodes. According to the hint, all leaf vertices have a polynomial P = x0.

2. Recursive Polynomial Assignment:
    - Proceed to the internal nodes. For each internal node v, calculate its polynomial based on the polynomials of its children.
    - For a node v of height h with children v1, ..., vk, define its polynomial Pv as (xh - Pv1)(xh - Pv2)...(xh - Pvk).
    - Perform this calculation from the leaves up towards the root, so that when you reach an internal node, all of its children have already been assigned polynomials.
3. Polynomial Comparison:
    - After assigning polynomials to all nodes in both trees, compare the polynomial associated with the root of T1 to the root of T2.
    - If the polynomials match, then the trees are isomorphic. If they do not match, the trees are not isomorphic.
4. Randomization:
    - To make the algorithm efficient and randomized, instead of actually expanding the polynomials, use a random value for x and evaluate the polynomial at this point for each node.
    - Since you are using random values, repeat the evaluation process multiple times to increase the confidence in the result. If the evaluated result matches for the roots across all trials, it is very likely (but not certain) that the trees are isomorphic.
5. Analysis:
    - The algorithm is efficient because it avoids the need to expand the polynomials fully; it merely requires polynomial evaluation, which is much faster.
    - The running time of the algorithm will be O(n), where n is the number of nodes in the trees because it processes each node once.
    - The probability of error can be made arbitrarily small by increasing the number of trials, as the chance that two non-isomorphic trees have the same polynomial evaluation for their roots multiple times is very low.

Remember, this algorithm assumes that a collision (two different trees having the same polynomial evaluation) is rare. In practice, for sufficiently large random x and a sufficiently large number of trials, this is a reasonable assumption. This makes the algorithm probabilistic; while it can quickly determine that two trees are not isomorphic, there is a small chance it might mistakenly identify two non-isomorphic trees as isomorphic if they happen to have the same polynomial evaluation at the chosen random points.

Question 7

Given a randomized algorithm for testing the existence of a perfect matching in a graph, describe how we can use this to construct a maximum matching in a graph G.

Answer 7

To construct a maximum matching in a graph G using a randomized algorithm that tests for the existence of a perfect matching, we can follow a process as outlined below:

1. Initial Test:
    - Run the randomized algorithm to check if there is a perfect matching in the graph G. If a perfect matching is found, it is also the maximum matching, and we are done.
2. Subgraph Creation:
    - If the graph G does not have a perfect matching, start creating subgraphs of G by removing one vertex at a time and all edges connected to it. This process effectively checks for perfect matchings in $G-\{v\}$, where v is the vertex being removed.
3. Iterative Testing:
    - For each subgraph created, run the randomized algorithm to test for the existence of a perfect matching.
4. Vertex Restoration and Matching Extension:
    - If removing a certain vertex v from G results in a subgraph that has a perfect matching, restore v and its incident edges to the subgraph.
    - Attempt to extend the matching by checking if any of the restored edges can be added to the matching without violating the matching conditions (i.e., no two edges share a common vertex).
5. Matching Augmentation:
    - If no perfect matching is found in the subgraph, continue removing vertices one by one and repeating the process. Each time a perfect matching is found in a subgraph, attempt to augment the matching in the original graph G by adding edges that are not yet matched and do not interfere with the existing matched edges.
6. Termination and Output:
    - This process is repeated until no more vertices can be removed, which means that all vertices are either part of the matching or have been tested for augmenting paths that could not be extended.
    - The matching that you have at this point is the maximum matching, as you cannot add more edges to the matching without violating the matching condition.

The rationale behind this approach is based on the principle that a maximum matching is a matching that contains the largest possible number of edges. No matching can have more edges than a maximum matching, and a perfect matching is a special case where all vertices are matched. If a perfect matching does not exist, then by systematically checking subgraphs for perfect matchings and attempting to extend partial matchings, you can incrementally build up a maximum matching.

Question 8

Let G(V,E) be a multigraph. Devise a data structure that processes any arbitrary sequence of edge contractions in G, such that at any given point where the set of edges contracted is F, the graph G/F is

available in the adjacency matrix format. Furthermore, it should possible to efficiently determine for any edge in E/F the corresponding edge in E. Your data structure should require O(n) time per contraction and use a polynomial amount of space. Can you modify this to provide the adjacency list format for G/F using only O(m) space?

Remark: Note that the time bound is independent of the number of edges. For this, the multigraph needs to be represented as a graph with integer edge weights that represent the multiplicities of the edges. You may assume that the number of edges in the multigraph is polynomial in n, although this is not strictly necessary.

Answer 8

Solving this problem requires designing a data structure that can handle edge contractions in a multigraph while maintaining the ability to represent the graph in adjacency matrix and adjacency list formats, and to trace each edge in the contracted graph back to the original graph.

1. Graph Representation:
    - Represent the multigraph using an adjacency matrix with integer weights, where the weight represents the multiplicity of the edge. This makes it possible to represent multiple edges between two vertices.
2. Edge Contraction:
    - When contracting an edge (u,v), you need to merge the vertices u and v into a single vertex. To do this efficiently, use a union-find data structure (disjoint set) which can perform union and find operations in nearly O(1) amortized time.
    - After the contraction, iterate over the rows and columns corresponding to vertices u and v in the adjacency matrix, and update the weights to reflect the contraction. The new weights are the sums of the weights from both u and v's rows and columns since all edges incident to u and v are now incident to the new merged vertex.
3. Maintaining the Adjacency Matrix:
    - The adjacency matrix can be updated in O(n) time after each contraction by merging the rows and columns of the contracted vertices.
    - To maintain a polynomial space complexity, ensure that the adjacency matrix is sparse and possibly use a hash table to store only non-zero entries.
4. Tracing Edges:
    - Maintain a mapping from the edges in the contracted graph to the original edges. This can be achieved with a hash table that maps each edge in the contracted graph (now identified by the merged vertices) to a list of original edges it represents.
5. Adjacency List Format:
    - To convert the adjacency matrix to an adjacency list format in O(m) space, iterate over the adjacency matrix and for each non-zero entry, add an edge to the adjacency list. This requires traversing each row's non-zero entries, which corresponds to the edges of the vertex associated with that row.

- Since each edge contraction might affect the adjacency list of multiple vertices, you would need to update the adjacency lists of these vertices after each contraction. This update should also be done in O(n) time.

6. Optimization Considerations:
    - To ensure the operations are efficient, additional bookkeeping may be required. For instance, keeping track of the size of each set in the union-find structure can help in always attaching the smaller set to the larger one, thus optimizing the union operation.