**HW5)**

**1. Exercise 19.3-1 on page 522**

Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a marked root. Argue that it doesn’t matter to the analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.

SOLUTION: A root x is marked if (1) one of his child is removed for any reason (a child being promoted as a new root also counts) or (2) x used to be a marked child of the H.min, and H.min is extracted by FIB-HEAPEXTRACT-MIN. If x were a marked non-root and lost one child, it would have promoted as a new root and unmarked at then, therefore whether x has been a marked non-root is irrelevant to analyzing how it came to be a marked root.

**2. Problem 19-3 on page 529 ---** ***More Fibonacci-heap operations***

We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

***a.*** The operation FIB-HEAP-CHANGE-KEY.H; x; k/ changes the key of node x to the value k. Give an efficient implementation of FIB-HEAP-CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is greater than, less than, or equal to x:*key*.

SOLUTION:

1. k > x.key: It is possible that k is larger than keys of some children of x. Therefore, update the key x.key ← k and then push x down until the min heap property is preserved. The worst actual cost is O(log n), and the potential does not change, therefore the amortized cost is O(log n).
2. k = x.key: Does nothing, therefore the potential does not change, and the amortized cost is equal to the actual cost (of the comparison) which is O(1).
3. k < x.key: Simply call FIB-HEAP-DECREASE-KEY(H, x, k), whose amortized cost is O(1).

***b.*** Give an efficient implementation of FIB-HEAP-PRUNE.H; r/, which deletes q D min.r;H:*n*/ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (*Hint:* You may need to modify the data structure and potential function.)

SOLUTION:

The amortized cost of deleting a given node is O(log n), but that does not mean the answer to this problem is O(q log n) because we are allowed to delete any q nodes we choose. Intuitively, we try deleting q leaves. In the worst case, every deleted leaf will incur cascading cut until the root (in each individual tree), which implies the actual cost of FIB-HEAP-PRUNE(H, r) is c = q log n. Now we use the potential function (as in CLRS page 509) Φ(H) = t(H) + 2m(H) to see what is the potential change. Note that I ignored all ±1’s during the calculation for better understanding, which does not affect the final amortized cost.

t(Hafter) − t(Hbefore) = q log n

since in the worst case q log n new trees are created due to the promotion during the cascading cut, and Hafter has q log n more trees.

m(Hafter) − m(Hbefore) = −q log n ⇒ 2m(Hafter) − 2m(Hbefore) = −2q log n

because in the worst case q log n marked nodes are promoted as new roots, which means there are q log n less marked nodes in Hafter. Then,

cˆ = c + Φ(Hafter) − Φ(Hbefore) = q log n + q log n − 2q log n = O(1)

Therefore, removing leaf nodes has an amortized cost O(1).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**HW6)**

1. You are given jugs of capacities c1, c2, and c3 and have to get a target amount t. Design an algorithm based on breadth-first search to determine how to get t, or to verify its impossibility. Remember, the graph is implicit—you do not have an explicit form of it (in a an adjacency structure, say).

SOLUTION: Let’s first start with the simple case where we have only 3 jugs, and see how to trivially extend it to allow for infinitely many jugs in theory.

For any vertex (i, j, k), we can create exactly 6 outgoing edges because we have 3 options for the jug from where we are pouring the water, and 2 options for the jug to where we are pouring the water (rule of product, 3 × 2 = 6). For the first edge, which determines the next state after we pour the water from the 1st jug to the 2nd one, we can determine its destination vertex as follows:

• If i = 0: this edge is a self loop pointing to the same vertex (i, j, k).

• If i + j > c2: this edge points to the vertex (i − (c2 − j), c2, k) because only the marginal amount of water (c2 − j) will be poured to the 2nd jug.

• Otherwise: the edge points to the vertex (0, i + j, k).

The destination vertices of the other 5 edges can be achieved in the exactly same way with slight change in the parameters.

Then, given any starting state (i.e., vertex), we can lively explore the graph by finding the destination vertices of 6 edges for each vertex. Sometimes, a vertex may have more than one self loop, which is natural and accepted. During the exploration, we maintain the color and the predecessor (π) of each vertex, and if we see an already visited vertex (gray or black), we simply skip it just like the BFS only explores the white unvisited vertices. If we find a vertex containing the number t, we use the predecessors list to find out the trace from the starting state to this state, and this trace shows how to pour waters to get the target amount t. If we did not see any vertex having t but the exploration terminated because no new vertex is found, we report error saying “It is impossible to get t”.

1. Justify the correctness of your algorithm in problem (1). This is just shy of a formal CLRS3-style proof.

SOLUTION: All vertices that are reachable from the starting vertex correspond to all possibly reachable states from the initial state of 3 jugs. Since our exploration process ensembles the BFS, our exploration will reach all nodes that are reachable from the starting vertex. Therefore, our exploration will find out the vertex containing t if it is there, and will report error if it is not in the implicit graph.

1. Analyze the time required by your algorithm in problem (1).

SOLUTION: If c1, c2, c3 and the parameters in the initial vertex are all integers, there are at most (c1+1)(c2+1)(c3+1) possible vertices because only integer differences will be applied. We also know there are 6 edges for each node, so |V | = O(c1c2c3) and |E| = 6|V | = O(c1c2c3). Since the the complexity of our exploration is same as the BFS, the complexity of our exploration is O(|V | + |E|) = O(c1c2c3). If they are fractional numbers, suppose the finest precision among the parameters is 0.0001, then at most there are 1/0.0001 × c1 + 1 possible values for the amount of water at the 1st jug (all values within the interval [0,c1]. Then, there are (10000c1 + 1)(10000c2 + 1)(10000c3 + 1) vertices at most in the implicit graph, and our complexity is still O(c1c2c3) because the precision is a constant. Then, the complexity is still O(c1c2c3) regardless of the precision.

5. This is a version of Problem 22-3 on page 623 of CLRS3. Given an undirected, connected graph G = (V, E), an Eulerian cycle is a cycle that traverses each edge of G exactly once, although it may through a vertex more than once. An Eulerian path is a path that traverses each edge of G exactly once, although it may through a vertex more than once.

(a) Prove that an undirected graph has an Eulerian cycle if and only if all vertices have even degree and has an Eulerian path if at most two vertices have odd degree. What happens if only one vertex has odd degree?

SOLUTION: An Euler cycle is a single cycle that traverses each edge of G exactly once, but it might not be a simple cycle. An Euler cycle can be decomposed into a set of edge-disjoint simple cycles, however. If G has an Euler cycle, therefore, we can look at the simple cycles that, together, form the Euler cycle. In each simple cycle, each vertex in the cycle has one entering edge and one leaving edge. In each simple cycle, therefore, each vertex v has in-degree(v) = out-degree(v), where the degrees are either 1 (if v is on the simple cycle) or 0 (if v is not on the simple cycle). Adding the in- and out-degrees over all edges proves that if G has an Euler cycle, then in-degree(v) = out-degree(v) for all vertices v.

We can prove the converse – that if in-degree(v) = out-degree(v) for all vertices v, then G has an Euler cycle – using a constructive proof. Let us start at a vertex u and, via random traversal of edges, create a cycle. We know that once we take any edge entering a vertex v 6= u, we can find an edge leaving v that we have not yet taken. Eventually, we get back to vertex u, and if there are still edges leaving u that we have not taken, we can continue the cycle. Eventually, we get back to vertex u and there are no untaken edges leaving u. If we have visited every edge in the graph G, we are done. Otherwise, since G is connected, there must be some unvisited edge leaving a vertex, say v, on the cycle. We can traverse a new cycle starting at v, visiting only previously unvisited edges, and we can splice this cycle into the cycle we already know. That is, if the original cycle is hu, · · · , v, w, · · · , ui, and the new cycle is hv, x, · · · , vi, then we can create the cycle hu, · · · , v, x, · · · , v, w, · · · , ui. We continue this process of finding a vertex with an unvisited leaving edge on a visited cycle, visiting a cycle starting and ending at this vertex, and splicing in the newly visited cycle, until we have visited every edge.

Suppose a graph G contains an Euler path P. Then, for every vertex v, P must enter and leave v the same number of times, except when it is either the starting vertex or the final vertex of P. When the starting and final vertices are distinct, there are precisely 2 odd degree vertices. When these two vertices coincide, there is no odd degree vertex.

Conversely, suppose G contains 2 odd degree vertex u and v. Then, we temporarily add a dummy edge (u, v) to G. Now the modified graph contains no odd degree vertex. By the above proof of Euler cycle, this graph contains an Euler cycle that also contains (u, v). Remove (u, v) from the cycle, and now we have an Euler path where u and v serve as initial and final vertices.

(b) Give an O(|V | + |E|) algorithm to find an Eulerian cycle (or path) if one exists and to report that neither exists, if that is the case.

SOLUTION: We can use Hierholzer’s algorithm to find a Eulaer cycle or path of which the idea is the same as the constructive proof. If the graph contains two odd degree vertices, we temporarily add a dummy edge between them and apply Hierholzer’s algorithm to find the Euler cycle, and then remove the dummy edge.

• Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.

• As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

By using a data structure such as a doubly linked list to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each, so the overall algorithm takes linear time.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**HW7)**

1. Modify Dijkstra’s algorithm so that it works with negative edge weights, as long as there are no negative cycles. Vertices that have come off the priority queue may have to be put back on.

SOLUTION: If we find a negative edge to a vertex v that is already out of priority queue (that is vertices for which a shortest path length has already been calculated assuming there were no negative edges connecting to it), then we should calculate new shortest path through the negative edge and update the v.d value and again push this new vertex to the priority queue. Therefore, we need to modify the RELAX to allow visiting a vertex more than once as shown:

Algorithm 1: RELAX-NEGATIVE(u, v, w)

1 if if v.d > u.d + w(u, v) then

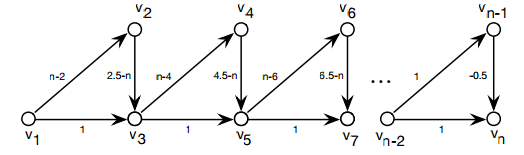
2 v.d = u.d + w(u, v)

3 v.π = u

4 if v 6∈ Q then

5 INSERT(Q, v)

However, the modified Dijkstra’s algorithm can take exponential time in the worst case. Specifically, we can construct a weighted graph of n vertices with negative weights, such that Dijkstras algorithm calls Θ(2n/2 ) RELAX. For example, we can construct the graph with negative weights as follows. Let T (n)



be the number of relaxation on v1, · · · vn. Then we can build a recurrence as

T (n) = 2 + T (n − 2) + 1 + T (n − 2) = 2T (n − 2) + 3 = Θ(2n/2 ),

where the first two relaxations are for (v1, v2) and (v1, v3), T (n − 2) relaxations are for v3, · · · , vn, one relaxation for (v2, v3) and T (n − 2) relaxations are for v3, · · · , vn. Note that v1.d < v3.d · · · < vn−2.d < vn−1.d < vn−3.d < · · · < v2.d during the execution of the algorithm.

2. The Floyd-Warshall all-pairs shortest path algorithm (section 25.2 of CLRS) computes, for each pair of vertices u, v, the shortest path from u to v. However, if the graph has negative cycles, the algorithm fails. Describe a modified version of the algorithm (with the same asymptotic time complexity) that correctly returns shortest-path distances, even if the graph contains negative cycles. That is, if there is a path from u to some negative cycle, and a path from that cycle to v, the algorithm should output −∞ as the length of the shortest path from u to v. For other pairs of vertices the algorithm should correctly find the length of the shortest directed path.

SOLUTION:

Algorithm 2: FLOYD-WARSHALL(W)

1 n = W.rows

2 D0 = W

3 for k = 1 to n do

4 let Dk = d k ij be a new n × n matrix

5 for i = 1 to n do

6 for j = 1 to n do

7 d k ij = min(d k−1 ij , dk−1 ik + d k−1 kj )

8 if i == j and d k ij < 0 then

9 d k ij = −∞

10 return Dn

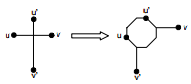
Notice that the Floyd-Warshall algorithm computes the weight of the path from a node to itself. This weight will be updated if and only if there is a negative circle. Otherwise dii = 0 will be the minimum for any node i. Therefore, we just need to modify the Floyd-Warshall algorithm by checking each update of dii . If any update changes dii to be smaller than 0, there exists a negative weighted cycle and we set dii = −∞, and any path using that cycle will result in −∞. Algorithm 2 shows the modified algorithm. Checking if dii < 0 takes constant time (Line 8-10) and the running time will remain to be Θ(n 3 ).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**HW8)**

1. Use the widget on page 4 of Professor Reingold’s graph coloring slides to prove that 3-coloring a planar graph is NP-complete. Remember, to prove NP-completeness you must prove NP-hardness and that the problem is in the class NP.

SOLUTION: It is easy to show that the gadget itself is 3-colorable as Fig. 1. Furthermore, this gadget has a symmetric property, which is the color of the node at the 11:00 clock postion could be always the same as the node at the 5:00 clock postion, and the color of the node at the 1:00 clock position could be always the same as the node at the 7:00 clock position, and so on. Thus we can always replace a crossed edge (u, v) with u embedded at a corner of the gadget and v connected to the opposite corner with an edge, as Fig. 2, which takes time O(E2 ). The resulting graph is planar and the symmetric property of the gadget implies that u and v cannot be assigned the same color. We have shown a reduction of the 3-colorability for an arbitrary graph to the same problem for a planar by using this gadget.

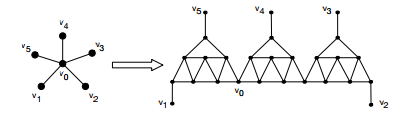


Since we can verify a color assignment by checking if the end points of every edge have different colors in O(E) time, 3-coloring a planar graph belongs to the class NP. Because 3-colorable is NP-complete for general graphs, we can prove that 3-colorable is NP-complete for planar graphs.

2. We want to prove that 3-coloring a graph is NP-complete even all vertices in the graph have degree at most 4 (the construction used in the graph coloring slides has vertices with much higher degrees). To do this we must show how to convert a graph with vertices of unrestricted degrees to a graph in which all vertices have degree at most 4. The idea is to replace a vertex of degree 5 with a widget. Use this idea to prove that 3-coloring a planar graph is NP-complete even all vertices in the graph have degree at most 4.

SOLUTION: In each trestle-like graph, the degree of “outer” vertices is at most 2, and the degree of “inner” vertices is at most 4. Since the two outer vertices have degree 2, two copies of trestle-like graphs can be fused at those vertices without increasing the degree of any vertex beyond 4. This graph will be 3-colorable if and only if the outer vertices all have the same color.

For a vertex v of degree d, the d edges incident to v can be distributed among the d outer vertices of the graph representing v without changing the colorability properties, as Figure 4.



In the solution to Question 1, we have shown how to transform any graph to a planner graph, and we can use trestle-like graph to replace any vertex with degree d > 4. Therefore, an arbitrary graph can transform to a planner graph with degree at most 4 in O(E2 + V ) time. Since we can verify a color assignment by checking if the end points of every edge have different colors in O(E) time, 3-coloring a planar graph belongs to the class NP. Because 3-colorable is NP-complete for general graphs, we can prove that 3-colorable is NP-complete for planar graphs with degree at most 4.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**HW9)**

1. The greedy coloring of a graph G = (V, E) is the coloring obtained by taking the vertices V in sequence, assigning to each the first available color ck. Show that the greedy coloring of a graph does not approximate the optimal coloring to within any constant ratio. (Hint: Consider a bipartite graph of vertices a1, a2, . . . , an and b1, b2, . . . , bn with edges (ai , bj ) for all i 6= j.)

Solution: Consider a a bipartite graph of vertices a1, a2, ... , an and b1, b2, ... , bn with edges (ai , aj ) for all i 6= j. In optimal coloring, we just need to use two colors: color a1, a2, ... , an using one color; color b1, b2, ... , bn with the other. The greedy algorithm, however may, color the vertices with n 2 colors for certain ordering of vertices. Let the vertices be in the following order: a1, b1, a2, b2, ..., an, bn. According to the greedy algorithm, a1 and b1 will be given the same color. a2 and b2 will be given a new color as there is edge (a2, b1). a3 and b3 will be given a new color as where are edge (a3, b1) and (a3, b2) and so on. To color all the vertices, we need n/2 colors. The approximation ratio is n/4 . Thus the greedy algorithm cannot achieve a constant approximation ratio.

2. Show that for any graph G = (V, E) there is an ordering of the vertices such that the greedy algorithm yields an optimal coloring.

Solution: For a graph G = (V, E), it is straight forward that there always exists a optimal coloring (of all coloring, the one that uses minimum number of colors). Suppose the optimal coloring use k colors to color G. Let the vertices of G are grouped by V1, V2, ... Vk. We require that the vertices in the same group associated with the same color by the optimal coloring. The vertices in different group are colored by different colors. Then we order the vertices by V1, V2, ... Vk. The greedy algorithm will always color the vertices in the same group with the same color and vertices in different groups with different colors. Thus the greedy strategy will use k colors which is the minimum number of colors to use.

3. Prove that it is NP-complete to determine that optimal ordering.

We can reduce the graph coloring problem to the optimal ordering. Thus the optimal ordering is NP-Hard. The decision version of the optimal ordering, where we just need to verify if an ordering can result in a k-color coloring, is in NP.

If one is able to find out a solution to the optimal ordering problem, he can use the aforementioned greedy algorithm to color the graph by following the given order. Then, the color given by the greedy algorithm is the optimal color. Because the answer given by the greedy algorithm is also the answer to the coloring problem, this is a constant-time reduction.