Assignment three

CSC425 |Instructor: Valerie King | UVIC | Fall, 2018

Zhaocheng Li (V00832770)

Question 1:

(a). Solution:

We are familiar with Karger's merge algorithm for the mincut problem.
In this question, we use array of vertices with linked list of neighbors for each vertex, to represent the graph.

After each merge, it cost too much time if we want to modify and equalize these two vertices throughout the array and linked list.

Hence we apply a data structure hash table to store the merged vertices(keys)--index after merge(values) correspondence. This is implemented like the dictionary in Python, the hash function should be perfect to avoid collisions.

In this way, we can equalize two merged vertices by giving same value on the hash table for these two vertices each time in only O(1) expected.

For the implementation of merging:

Input: an array of size n
repeat until only two vertices left:
    # pick an edge uniformly randomly (not problem here)
    # contract the edge
    Put two endpoints in hash table and label the same value.
    label other vertices having the same value in hash table with endpoints the same value as well
    Remove the self loop (two vertices with the same value in hash table)in linked list of each vertex

The above algorithm will cost $O(n^2)$, since it go through all vertices in the array and their linked lists

(b). Solution:
We are going to store all the edge in an array of size m, which is the sum of size of linked lists, and generate a number randomly in the range {1,m}.

Question 2:
The way we prove for the competitive ratio of marking algorithm is similar to what we did for competitive ratio of the LRU algorithm.

Recall the algorithm of the marking algorithm. A marking algorithm maintains a bit for marking for each page in the cache. It

- Starts with all pages unmarked;
- For a hit, make the page;
- For a miss, if eviction is needed, then evict an unmarked page.
    - If all pages in the cache are marked, unmark all of them first.

Then we apply a phase partitioning method for a sequence of requests. Divide requests into a sequence of phases such that in each phase, it will invoke k different pages. I.e., the first request in the next phase is different from all k requests from last phase.

Consider the procedure of marking algorithm above, it starts the phase with all pages unmarked, while at the end of the phase, all k pages in this phase are marked. For the first request to object p, it becomes marked. Then p is kept in the cache until the end of the phase. The algorithm costs 1 for p throughout the phase.
Thus for each phase, the cost of the marking algorithm is at most k.
However, for the optimal offline strategy,Since each phase and next following page are totally k+1 distinct pages. By the offline algorithm, which choose the nest request furthest in the future, the cost is only 1 for each phase.

So the competitive ratio denoted as c.r., for the marking algorithm (M) is

$$\begin{aligned} \text{c.r.}(M) &= \frac{M(phase1)+M(phase2)+ \dots +M(phase\ n)}{Opt(phase1)+Opt(phase2)+ \dots +Opt(phase\ n)} \\ &\leq \max \frac{M(phase\ i)}{Opt(phase\ i)} \\ &\leq k \end{aligned}$$

Therefore the marking algorithm is k-competitive ratio.

Question 3

(b). Solution:

This is equivalent of asking if we can put number into fewer than $\lceil S \rceil$ bins. Assuming that we can do so, then it means the total capacity *p* of out bins is

$p < \lfloor S \rfloor \leq S$.

However, the fact is given that

$$S = \sum_{i=1}^{n} s_i ,$$

meaning the total space required to accommodate the elements is *S*, so we cannot fit them in less than S, contradicting the assumption.
Therefore the optimal number of bins required is at least $\lceil S \rceil$.

(c). Solution:

Assuming that we have one bin which is at most half full.

Then when we take next object of size at most half, by the rule of ***first-fit***, we will not put it into an unused bin since it will go to the bin of size at most half full, this is the bin we had already.

In this way we will never create another bin that is at most half full. Since we want to pack all objects into the minimum number of unit-size bins. Then we begin with fewer than two bins that are at most half full. (Because just like what we illustrated, if next object is of size more than half-full, we take a new bin, otherwise we can accommodate it in the existed bin of size at most half-full)

Thus there will never be two bins of size at most half full (and thus never two that are less than half full).

(d). Solution:

From question (c) we know there is at most one bin that is less than half full,
So we know that in all possible scenarios,

$$S > \frac{m-1}{2} ,$$

Where m is the number of bins we use. In another form,

$$2S + 1 > m,$$

So, assuming that $m > \lceil 2S \rceil$, then,

$$\begin{aligned} m &\geq \lceil 2S \rceil + 1 \\ &= \lceil 2S + 1 \rceil \\ &\geq 2S+1 \\ &> m, \end{aligned}$$

which is a contradiction.

Therefore, the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$.

(e). Solution:
From question (b), we know that the optimal number of bins required is at least $\lceil S \rceil$; and from question (d), we know the first-fit heuristic asks for using at most $\lceil 2S \rceil$bins.
Thus the number of bins we use is within a factor of $\frac{\lceil 2S \rceil}{\lceil S \rceil}$. And,

$$\frac{\lceil 2S \rceil}{\lceil S \rceil} \leq 2.$$

For the first-fit heuristic, the approximation ratio is 2.

(f). Solution:
Implementation:

Keeping an array A of bins, and an array B of objects.
The lengths of A and B is *m* and *n*, respectively.

since we know there was at most one bin that was less than half-full, and
make use of it. We can do faster.
Each time picking an object s:

    # assigning pointers
    a ← first empty bin
    b ← bin of size less than half-full
    If there is no nonempty, less than half-full bin:

```
        b = a
# Find the first bin that can accommodate the object.
If size(s) more than half-full:
        Insert s into first empty bin, a
        Increment a, i.e., a ← next empty bin
Else if size(s) less than half-full:
        Fill s in b, b is updated
        If size(b) more than half-full:
                Get rid of less than half-full pointer, b ← null
```

Running time:
This will run in time O(n) (linear time), since we save time of linearly searching suitable bin with pointers.

Question 4
(a). Solution:
Since M is a matching and P is an augmenting path with respect to M. And by definition of augmenting path here, its first edge goes to an unmatched vertex in L, then such edge can not be in M.

Similarly, last edge of P goes to an unmatched vertex in R, so the last edge cannot be in M.

Also since the edges alternate being in or not in M, there must be one more edge not in M than in M. So for the augmenting path P, it contains k edges in M and k+1 edges not in M. And it implies

$$\begin{aligned} |M \oplus P| &= |M| + |P| - 2k \\ &= |M| + 2k + 1 - 2k \\ &= |M| + 1 \end{aligned}$$

by deleting each edge of M which is in P from both M and P following definition of symmetric difference.
As required, assuming that $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M.

Let k_i be the number of edges in P_i which are in M, so that $|P_i| = 2k + i + 1$.
Thus we have

$$M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k) = |M| + |P_1| + \ldots + |P_k| - 2k_1 - 2k_2 - \ldots - 2k_k$$
$$= |M| + k.$$

In order to prove we get a matching, assuming that there was some vertex v which had at least 2 incident edges e and e' .

Since M is a matching, then they cannot both come from M. Similarly, P is simple and every other edge of P is deleted, then they can not come from P both.

Thus, $e \in M$ and $e' \in P \backslash M$. However, if $e \in M$ then $e \in P$, so $e \notin M \oplus P$, which generates a contradiction. Thus M is a matching.

In the similar procedure, we can prove the matching in the case of

$$M \oplus (P_1 \cup \ldots \cup P_k)$$

as well.

(b). Solution:
In order to prove this, we can assume that G' has some vertices of degree at least 3 first.
Since the edges of G' is from $M \oplus M^*$, then at least 2 of these edges come from the same matching, contradicting the definition of a matching.
So this is impossible.
Thus every vertex has degree at most 2, and G' is a disjoint union of simple paths and cycles.

If edge (a, b) is incident with edge (c, d) in a simple path or cycle and b=c.
By definition, two edges with the same endpoint cannot appear in a matching, they must belong alternately to M and M^* .
Also every cycle has the same number of edges in each matching and every path has at most one more edge in one matching than in the other.
Therefore, it is proved that if $|M| \leq |M^*|$, then there must be at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

(c). Solution:
From the definition of M', we know that every vertex matched by M must be incident with some edge in M' .
Since P is augmenting with respect to M' , the left endpoint of the first edge of P is incident to a vertex connected by an edge not in M' .

In other words, P begins with a vertex which is unmatched by M since every vertex of M is incident with an edge in M' .
Since P is vertex disjoint from P1, P2, . . . , Pk, any edge of P which is in M' must in fact be in M and any edge of P which is not in M' cannot be in M.
Since P has edges alternately in M' and E − M', P must in fact have edges alternately in M and E − M.
Then the last edge of P must be incident to a vertex in R which is unmatched by M'. Any vertex unmatched by M' is also unmatched by M, so P is an augmenting path for M.
P must have length at least *l* because *l* is the length of the shortest augmenting path with respect to M.
If P had length *l*, then this would contradict the fact that P1 ∪ . . . ∪ Pk is a maximal set of vertex disjoint paths of length *l* because we could add P to the set.
Thus P has more than *l* edges.

(d). Solution:
We know that any edge in M ⊕ M' is in exactly one of M or M' . Thus, the only possible contributing edges from M' are from P1 ∪ . . . ∪ Pk.
An edge from M can contribute if and only if it is not in exactly one of M and P1 ∪ . . . ∪ Pk, which means it must be in both.
Thus, the edges from M are redundant so M ⊕ M' = (P1 ∪ . . . ∪ Pk) which implies A = (P1 ∪ . . . ∪ Pk) ⊕ P.

In order to prove that P is edge disjoint from each Pi, assume that an edge e of P is also an edge of Pi for some i.
Since P is an augmenting path with respect to M' , either e ∈ M' or e ∈ E − M' .
Suppose e ∈ M' , since P is also augmenting with respect to M, we must have e ∈ M.
However, if e is in M and M' then e cannot be in any of the Pi 's by the definition of M' .

If e ∈ E −M' , then e ∈ E −M since P is augmenting with respect to M. Since e is an edge of Pi , e ∈ E −M' implies that e ∈ M, a contradiction.

Since P has edges alternately in M' and E − M' and is edge disjoint from P1 ∪ . . . ∪ Pk, P is also an augmenting path for M, which implies |P| ≥ l.

Therefore we conclude that |A| ≥ (k + 1)l since every edge in A is disjoint.

(e). Solution:
Suppose M* is a matching with strictly more than $|M| + |V|/(l + 1)$ edges.
From (b) there are strictly more than $|V|/(l + 1)$ vertex-disjoint augmenting paths with respect to M.
Each one of these contains at least l edges, so it is incident on l + 1 vertices.
Since the paths are vertex disjoint, there are strictly more than $|V|(l + 1)/(l + 1)$ distinct vertices incident with these paths, which is a contradiction.
Thus, the size of the maximum matching is at most $|M| + |V|/(l + 1)$.

(f). Solution:
After iteration number $\sqrt{|V|}$. Let M* be a maximal matching in G.
Then $|M^*| \geq |M|$.
By question (b), $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex disjoint augmenting paths with respect to M.
By part (c), each of these is also a an augmenting path for M. Since each has length $|V|$, there can be at most $\sqrt{|V|}$ such paths, so $|M^*| - |M| \leq \sqrt{|V|}$.
Thus, only $\sqrt{|V|}$ additional iterations of the repeat loop can occur, and there are at most $2\sqrt{|V|}$ iterations in total.

(g). Solution:

Implementation;
For each unmatched vertex in L, perform a BFS to find the length of the shortest path to an unmatched vertex in R. the BFS is modified to make sure that we only traverse an edge if it causes the path to alternate between an edge in M and an edge in E − M.
The first time an unmatched vertex in R is reached we know the length k of a shortest augmenting path. We can use this to stop our search early if at any point we have traversed more than that number of edges.
To find disjoint paths, start at the vertices of R which were found at distance k in the BFS.
Run a DFS backwards from these, which maintains the property that the next vertex we pick has distance one fewer, and the edges alternate between being in M and E − M.
Along with the path, mark the vertices as used so that we never traverse them again.

Running Time:
This takes O(E), thus from (f) the total runtime is O($\sqrt{V}$ E).