Due 23:59 Nov 22 (Tuesday). There are 100 points in this assignment.
Submit your answers (**must be typed**) in pdf file to CourSys
`https://coursys.sfu.ca/2022fa-cmpt-705-x1/`. Submissions received after 23:59 of Nov 22 will get penalty of reducing points: 20 and 50 points deductions for submissions received at $[00:00, 00:10]$ and $(00:10, 00:30]$ of Nov 22, respectively; no points will be given to submissions after $00:30$ of Nov 22.

1. (Chapter 8 Problem 4 of the text book) 20 points

   A system consists of n processors and m resources; each processor requests a subset of resources. An assignment of resources to processors is to allocate each resource to at most one processor. A processor is active in an assignment if all resources it requests are allocated to it, otherwise blocked. The resource reservation problem is that given n processors, m resources, a set of requested resources for each processor, and an integer $k > 0$, is it possible to allocate resources to processors so that at least k processors are active. For each of the following resource reservation problems, give a polynomial time algorithm or prove it is NP-complete.

   (a) The general resource reservation problem as above.

   (b) The special case of the problem with $k = 2$.

   (c) The special case that there are two types of resources (e.g., hard disks and GPUs) and each processor requests at most one resource of each type.

   (d) The special case that each resource is requested by at most two processors.

Answer

For each process, it is either active or it is blocked by the OS. Active processes are the ones for which all of the resources are available for use and Blocked processes in this case are those which are blocked by the OS because of contention of resources.

(a) This problem can be boiled down to decision problem, rather than a optimization problem. This is because of the reasoning provided in the preface to this question. So for this we can consider a graph which has the nodes representing the processes running on the processors, and edges can be resources available for each process. As this is a decision problem, we try finding another well known decision NP complete problem and map it to our problem of Resource Allocation. Firstly, the problem is an NP complete problem as we can query the processors to tell which resources are requested by more than one processors, or which resources are still up for grab. We now need to show that another problem $B <=_p A$, where A is our Resource Allocation problem and B is any other NP complete problem that we know of. So, if we look at Bipartite Independent Set problem, which we know to be NP complete. For this we have processors $p_1, p_2, ...p_k$ and the edges we have are the requirement for resources we have. If we have a request by two or more processors for a single resource we draw a edge between them. One of the processors among the connected ones will get activated, and the rest of them will get blocked, hence maintaining the integrity of the problem. Hence we can safely say that the problem described in the question is NP complete problem.

(b) For the edge case we have of $k = 2$, where two processors are at the most contention for a resource. We can have two independent sets, as in the previous part, and we can check if there is contention between the processors in polynomial time. Hence, this special version can be solved within polynomial time.

(c) This special case can also be solved under polynomial time, this can be proved by assuming that we have two resources HDD, GPU. In the question itself we have the assumption that each processor requests at most one resource of each type. This makes the problem easy enough as there won't be any contention for resources. This means that there will be two disjoint sets. So to prove if we have k active processes, we just need to check if there is k resources of each HDD and GPU. This can be done in under polynomial time.

(d) If we have the special case that each resource can be contented by at the most two processes, we can easily check if the resource (edges) are in contended by the processors (vertices). This can be done in polynomial time in the number of resources, m.

2. (Chapter 8 Problem 6 of the text book) 15 points

   A CNF F on Boolean variables $x_1, .., x_n$ is monotone if each literal in each clause of F is the form of x_i (the negation $\bar{x}_i$ does not appear in any clause for any i). Given a monotone CNF F, the monotone satisfiability with few true variable problem asks that whether there is a satisfiable assignment for F in which at most k of $x_1, .., x_n$ are set to 1? Prove this problem is NP-complete.

Answer

Conjunctive Normal Form is a type of formulation to express Boolean expressions, literals are $x, \bar{y}, z, etc.$, clauses are $(x \vee y), (x \wedge \bar{y}), etc.$. A CNF Formula is satisfiable if there is a truth assignment such that the formula equates to $True$. It takes polynomial time to equate literals in a CNF formula for it to equate to $True$, hence it is NP problem. A Boolean CNF formulation is a Monotone if there are no not operations in the expressions. We can say that the Boolean CNF satisfiability (SAT) is at least as hard as Monotone satisfiability. The CNF satisfiability problem is a decision problem, because it can have say "at most k literals to True or 1". Now we have a CNF satisfiability problem, which has say k clauses like: $C_1, C_2, ..., C_k$. We can base the proof on the statement "if a variable x_i and its negation $\bar{x_i}$ exists in any one of the clauses that means that there is a conflict". To change this CNF satisfiability to a monotone satisfiability, we reduce the CNF SAT to monotone SAT. We can do that by, (i) Search for non-conflicting examples in the negative state and flip their values (there can be k of these). (ii) For each negated literal we find, $\bar{x_i}$, we can replace them with a new variable, say y_1, which is now a positive literal. By doing these reduction we have now achieved the Monotone satisfiability problem, which is what we were after. Since, CNF SAT is a well known NP complete problem, we can safely assume that Monotone satisfiability is at least as hard as the CNF SAT, So monotone satisfiability is also a NP hard problem.

---

3. (Chapter 8 Problem 21 of the text book) 15 points

   The fully compatible configuration (FCC) problem is defined as follows: An instance of FCC consists of disjoint sets of $A_1, .., A_k$, each A_i is a set of options, and a set P of incompatible pairs (x, y), where $x \in A_i$ and $y \in A_j$ with some $1 \leq i \neq j \leq k$. The FCC problem is to decide whether there is fully configuration which is a selection of one element from each A_i $(1 \leq i \leq k)$ so that no pair of selected elements is in P. Prove the problem is NP-complete.

Answer

As this question is not detailed enough to be understood, and needs more explanation. In the problem 21 (chapter 8) of the book, we can see that we are required to find out if all of the pairing are compatible or not. We have a set of disjoint options for our computer which are $A_1, A_2, A_3, ..., A_k$ and there is a set of incompatible pairs in P. So we are required to find if by selecting each option from the A set, we have no option pair in the set P. The example given in the book with $k = 3$ and the choice of software, OS and email client can be worked out by using the same problem in the previous solution of Boolean satisfiability (SAT). For this we can say that 3-SAT is at least as hard as our FCC problem and it is a well known NP hard problem. So, for this we can have a k-SAT problem, and map it to the FCC problem and we then have to show that $SAT <_p FCC$. If we start from SAT problem and have a option set for each clause given to us and each set can have k options corresponding to the initial clauses we have. We can now have options A_i and A_j which are compatible. Now when we choose a term from each of the clauses (as from the previous question) but then we should ignore its negations, this would result in no incompatibilities. With this we have mapped the SAT problem to our FCC problem. This would now satisfy the SAT problem integrity. With this we can conclude that, if SAT problem is a NP hard problem, so FCC should be at least as hard as itself. This makes the FCC problem also a NP complete problem.

---

4. (Chapter 11 Problem 1 of the text book) 15 points

   There are n containers $1, .., n$ of weights $w_1, .., w_n$ and some trucks, each truck can hold at most K units of weight (w_i and K are positive integers, $w_i \leq K$). Multiple containers can be put on a truck subject to the weight restriction K. The problem is to use a minimum number of trucks to carry all containers. A greedy algorithm for this problem as follows: start with an empty truck and put containers $1, .., j$ on the truck to have this truck loaded (i.e., $\sum_{1 \leq i \leq j} w_i \leq K$ and $\sum_{1 \leq i \leq j+1} w_i > K$); then put containers $j+1, j+2, ...$ on a new empty truck to have this truck loaded; continue this process until all containers are carried.

   (a) Give an example of a set of containers and a value K to show that the greedy algorithm above does not give an optimal solution.

   (b) Prove the greedy algorithm is a 2-approximation algorithm.

Answer

(a) This question is very similar to one that we did in one of the previous assignments, where greedy algorithm does not work at all. The problem was to minimize the number of boxes visible by placing smaller boxes into other bigger boxes. If we have three containers with the weight $2, 4, 2$ and say the maximum weight that the truck can hold is 4. The greedy algorithm will go on and select the container 1 in one truck, container 2 in the second truck and the container 3 in the third truck. Whereas, the optimal solution is to put the first and the third container in one truck and the second container into the second truck. Hence we can say that the greedy algorithm does not perform optimally, as it uses 3 trucks compared to the 2 trucks used by the optimal solution.

(b) 2 Approximation Algorithm like for vertex cover is where we choose edges (add them to a list of vertex cover) and remove their incident edges, then we run this again till the graph has no more edges, this is a 2 approximation algorithm. Here we take the ratio of the result of our algorithm with the optimal result $\alpha = \frac{Algo(I)}{Opt(I)}$. Now onto the problem at hand, If we have a number, say K, which represents how much a truck can hold and we have the total weights of all of the trucks, as W. We can say that we will require at least W/K trucks to move the containers (lower bound). If we say that the number of trucks used by the greedy algorithm is an odd number m, and odd numbers can be represented by the formula $m = 2q + 1$, we can divide the number of trucks assigned into groups of 2. There will be a total of $q + 1$ groups. For the first q groups we can safely assume that the weight of containers assigned to the trucks is greater than K. Otherwise the second truck will not start its journey. We can deduce by this that $W > q \times K$, hence $\frac{W}{K} > q$. With this we can say that the optimal solution, as described above, $q + 1$ trucks which is lower bounded by m by a factor of 2, $m = 2q + 1$. This means that this greedy solution is a 2-approximation algorithm. The same thing can be done if the greedy algorithm assigns an even number of trucks $m = 2q$. Same logic can be applied to that, and we will reach the same outcome.

---

5. (Chapter 11 Problem 3 of the text book) 15 points

   Given a set of positive integers $A = \{a_1, .., a_n\}$, positive integer B and a subset $S \subseteq A$, $t(S) = \sum_{a_i \in S} a_i$ is called the total sum of S. A subset S is called a feasible set if $t(S) \leq B$. The maximum feasible set problem is find a feasible set S with $t(S)$ maximized.

   (a) A simple greedy algorithm works as follows:

```
Initially $S = \emptyset$; $t(S) = 0$;
for $i = 1, 2, .., n$ do
    if $t(S) + a_i \leq B$ then $S = S \cup \{a_i\}$ and $t(S) = t(S) + a_i$;
endfor
```

   Give an instance that the algorithm returns a feasible set S with $t(S) < (1/2)t^*$, where t^* is the maximum total sum of a feasible set.

   (b) Give an $O(n \log n)$ time (1/2)-approximate algorithm for the problem.

Answer

(a) We have an positive integer array of A, and we have a positive number B. If we have an array A with 2 elements, such that $A = 1, 50$, and the integer B equals 50, such that $B = 50$. The algorithm will return a feasible set S with only a_1, where the optimal solution would choose a_2 and it holds the constraint that we were given with $t(S) < (1/2)t^*$ or $1 < (1/2)50$.

(b) If we are considering the algorithm suggested in the question, we can firstly delete the numbers which exceed B, just as we delete the objects which exceed the total weight capacity in Knapsack Algorithm, because they do not contribute in any sort to the solution. Now, we will iterate through the array A till we reach the point where the if conditional becomes false. The one object before this happens would still be in the bound B. We now have two sets where we can draw the conclusion from: $\{a_1, a_2, ...a_{i-1}\}$ and a_i, where i is the iteration value where the conditional broke down. We can say that one of the two sets we have, mentioned in the previous line, are at most B and at least $B/2$ (in the range $B/2 - -B$). We can select whichever one this set is to be our optimal desired value. We can see that it is (1/2)-approximate approximation algorithm. As far as time complexity is concerned, for the first deletion which takes at most n time steps, lastly we run through the list again taking at most n time steps. We can assume that the time complexity of this algorithm is $O(n)$.

---

6. (Chapter 11 Problem 5 of the text book) 20 points

   (a) Consider a load balancing problem instance of 10 machines and n jobs $S = \{1, .., n\}$ with $1 \leq t_i \leq 50$ for every i and $\sum_{i=1}^{n} t_i \geq 3000$. Prove that the greedy algorithm discussed in class finds a solution of makespan T with $T \leq (1.2)(\sum_{1\leq i\leq n} t_i)/10$ for this instance.

   (b) Implement the greedy algorithm to find a solution for the load balancing problem instance in (a) and compare the solution with the lower bound $(\sum_{1\leq i\leq n} t_i)/10$ on the makespan of the instance (each t_i can be generated randomly). Submit the results of the greedy algorithm and the lower bound for some instances of about $30 \sim 50$ jobs.

Answer

(a) If we have n jobs and m machines (10 in our case). If for a machine i, assume i had the smallest load we have a job t_i and the previous job run on the machine was j. Its load, before assigning job j, can be thought of as $T - t_j$ (T being the total load on the machine). This can be written as $T - t_j < T_k$ (T_k is loads on the machines $1 \leq k \leq m$). We can now divide the expression by total number of machines m.

$$T - t_j \leq \frac{1}{m}\Sigma_m T_k$$

$$T - t_j = \frac{1}{m}\Sigma_m t_k$$

$$T - t_j \leq T^*$$

This expression tells us that T is our makespan with the job t_j then the optimal makespan is T^*. Now we also know that: $T^* \geq \frac{1}{m}\Sigma_j t_j$. We can now put the values we have $m = 10, \Sigma_j t_j = 3000$

$$T^* \geq \frac{1}{m}\Sigma_j t_j \geq \frac{1}{10}3000 = 300$$

The average load is 300 in our case, and our optimal makespan is at least the average load. Now the question in the book asks that prove that the greedy algorithm will find a solution, which has a makespan of at most 20% above the average load. For this we can use $\frac{T-T^*}{T^*} \leq \frac{t_j}{T^*}$

$$\frac{T - T^*}{T^*} \leq \frac{t_j}{T^*} \leq \frac{50}{300} = \frac{1}{6} \leq 20\%$$

(b) For this part I randomly generated 30 jobs which have t_j are within the range 1-50, with 10 machines. I got the following jobs $[43, 23, 19, 10, 30, 27, 38, 39, 9, 40, 4, 25, 36, 12, 47, 29, 45, 24, 28, 13, 17, 31, 32, 44, 41, 48, 21, 26, 35, 2]$. When I assigned the jobs based on the load. I can see that the jobs get assigned to the machines in the order $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 3, 8, 2, 1, 5, 4, 2, 3, 6, 7, 9, 0, 8, 6, 2, 5, 7, 3, 1]$. With this I calculated the load on each machine to be $[75, 72, 103, 98, 75, 77, 92, 82, 93, 71]$. As part (b) of this question is independent of part (a), my sum of all the jobs amount to $\Sigma_j t_j = 838$. This means that my optimal makespan is:

$$T^* \geq \frac{1}{m}\Sigma_j t_j \geq \frac{1}{10}838 = 83.8$$

$$\frac{T - T^*}{T^*} \leq \frac{t_j}{T^*} \leq \frac{50}{83.8} = \frac{1}{6} = 59\%$$

With this stated the load on each machine in my case are: $[75, 72, 103, 98, 75, 77, 92, 82, 93, 71]$. With this we can say that the greedy algorithm performs quite well. The ratio of the difference of our makespan and the optimal makespan to the optimal makespan comes out to be 59%.