CSE 6140 / CX 4140 Assignment 4
due Nov 13, 2020 at 11:59pm on Canvas

Ting Liao GTID: 903278773
(collaborator: Chin Wang GTID: 903506612 / Shen-Yi Cheng GTID: 903514405)

11/17/2020

1 Local Search and Backtracking [20 pts]

The 3-COLORING problem is as follows: Given a graph G, can you find an assignment of colors to nodes using at most 3 colors, such that no two adjacent nodes have the same color.
3-COLORING is an NP-complete decision problem and hence we will use local search (LS) and backtracking. You will design a local search algorithm to try to find a feasible solution, or at least one for which the number of adjacent nodes with same color is minimized.

(a) (5 pts) Local Search: How will a candidate solution be represented and what will be the evaluation metric for a candidate solution (3 pts)?
Describe one strategy for finding an initial solution (2 pts).
Ans:
The candidate solution of the 3-coloring problem will be represented as a set of nodes (Let's say S = $\{s1, s2 \ldots\}$) and each node has different color (using 1, 2, 3 to represent the 3 color) with its adjacent nodes. For example, if we have four nodes, and the adjacency list looks like:
=> 1 : 2 3
2 : 1 3 4
3 : 1 2 4
4 : 2 3
Then, one possible candidate solution will be $\{1, 2, 3, 1\}$.

For the evaluation metric, we could check through the candidate solution and see if it met the requirement of no two adjacent nodes with same color or the number of adjacent nodes with same color is minimized. If it satisfy, we found the feasible solution. If not, then we could try another candidate solution and check again.

To find the initial solution, we could simply assign different colors into the set of nodes, such as $\{1, 2, 3, 1, 2, 3 \ldots\}$. Then, we could use the evaluation metric to verify the set.

(b) (5 pts) Local Search: Describe a neighborhood for this search problem – how do we move from one candidate solution to another (3 pts)? What is the size of the neighborhood, i.e. what is the time complexity of finding the best move from the current solution (2 pt)?
Ans:
The neighborhood of this problem can be defined by the following thought. First, we want to find no two adjacent nodes with same color or the number of adjacent nodes with same color is minimized. That means the neighborhood should have less number of adjacent nodes with same color compared with the original candidate solution. To move, we can decide to change the color of a node and compare

with changing the other nodes' color to check which has less number of adjacent nodes with same color. Then, we will move to that solution.

Generally, the neighborhood solution depends on the information about the solutions in the neighborhood of the current candidate solution. The size of the neighborhood will be the number of neighbor nodes notated as m of current candidate solution and the time complexity should be O(m) = O(n).

(c) (10 pts) If we use backtracking to solve the decision problem, what would `Expand()` do for 3-COLORING and how many children will each node have (2 pts)?
What is the worst-case running time of backtracking for 3-COLORING in asymptotic (e.g. Big-O) notation (2 pt)?
What are the possible outcomes of `Check()`? (4pts)
Why would `Check()` prune a search node when solving 3-COLORING (2 pts)?
Ans:
The Expand() function will choose an adjacent node of the current node and the number of child depends on how many adjacent nodes connected with the current node.

The worst case time complexity of 3-Coloring is O(3N) = O(N) because we have 3 colors and N nodes.

The possible outcomes of Check() are checking for solution found or not and meeting the dead end or not. If the solution found, then we check for the minimum same colors and update the upper bound based on the check. If not dead end, we check for the lower bound.

The Check() prune a search node because if the lower bound is greater than the current candidate solution, it means that this solution violate the initial setting of lower bound. Therefore, the Check() should prune the search node.

2 Optimizing Amazon's Operations (16 pts)

Amazon is considering building a set of warehouses in a new market. There are n possible locations at which a warehouse can be built, and the cost of building a warehouse at location $i \in \{1, \ldots, n\}$ is $f_i \in \mathbb{R}_{>0}$. To serve this new market efficiently, Amazon would like the warehouses to be close to the cities. In particular, there are m cities, and the distance between city $j \in \{1, \ldots, m\}$ and warehouse $i \in \{1, \ldots, n\}$ is $d(i, j) \in \mathbb{R}_{\geq 0}$. Of course, Amazon could build a warehouse at every one of the n possible locations, but that may be too expensive. Instead, the company would like to construct warehouses at a *subset* W of the n locations, $W \subseteq \{1, \ldots, n\}$, such that the sum of the total construction cost and the minimum distances to the cities is minimized. More formally, Amazon would like to find the set W that minimizes:

$$\sum_{i \in W} f_i + \sum_{j \in \{1,\ldots,m\}} \min_{i \in W} d(i, j)$$

1. Devise a branch-and-bound algorithm for this problem. This entails deciding:

   (a) What is a subproblem? (2 pts)
   Ans:
   For this problem, we can first think about the process during the program. There might be some locations have warehouses, some locations don't and some locations are not decided yet.

Suppose S is the set S[i] storing the value $\{-1, 0, 1\}$, which determine the status of each location. "0" represents no warehouse, "1" means that the location has the warehouse and "-1" means the location has not been determined yet. Also, let S_0, S_1, S_{-1} be the subset of $\{1, \ldots, n\}$ to store the index for S[i] and their union is $\{1, \ldots, n\}$. Moreover, we defined d'(j,W) as the minimum distance between city j and the set of location i in W.

Therefore, the subproblem is shown below:

$$\sum_{i \in S_{-1}} f_i + \sum_{i \in S_1} f_i + \sum_{j \in \{1,\ldots,m\}} \min \left\{ d^{'}(j, S_1), d^{'}(j, S_{-1}) \right\}$$

Then, we can use this relation to find the subset of S-1 which minimized the cost.

(b) How do you choose a subproblem to expand? (2 pts)
Ans:
We choose the subproblem with the smallest cost to expand. For example, if we choose location i ($i \in S_{-1}$) to build the warehouse and get the smallest cost compared with other locations, we could use location i as the next starting point and continue to find the next subproblem with the smallest cost.

(c) How do you expand a subproblem? (2 pts)
Ans:
After picking a location i in the subset S-1, now we can consider for the subproblem that setting S[i] to 0 or 1 (Build the warehouse on this location or not) will return the smallest cost. Therefore, we expand our subproblem.

(d) What is an appropriate lower bound? Argue why this is a valid lower bound. (3 pts)
Ans:
The lower bound would be:

$$\sum_{i \in S_1} f_i + \sum_{j \in \{1,\ldots,m\}} \min \left\{ d^{'}(j, S_1), d^{'}(j, S_{-1}) \right\}$$

The fixed cost would be the construction cost $\sum_{i \in S_1} f_i$ and its minimum distance $\sum_{j \in \{1,\ldots,m\}} \min \left\{ d^{'}(j, S_1) \right\}$. However, when we build a new warehouse in a location choosing from S_1, the minimum distance could be shortened. Therefore, we will need to take

$$\sum_{j \in \{1,\ldots,m\}} \min \left\{ d^{'}(j, S_{-1}) \right\}$$

into consideration. From the above reason, this will return the minimum cost and distance for each turn of computing and thus this is the valid lower bound.

2. Outline a simple greedy heuristic for the problem, and provide the pseudocode. (4 pts)
Explain why it finds a valid solution (1 pts) and its running time (2 pts).
Ans:
The greedy algorithm we used is to keep trying the index in S-1 and call the subproblem to check

whether it return the minimum cost.

Pseudo code :

```
// Initial setting
S0 = 0, S1 = 0, S-1 = {1, ..., n} (due to that all locations are undetermined in the beginning)
totalCost = INF
new = 0
while new < totalCost do
    for i in S_{-1} do
        if i can get the minimum cost by using the subproblem and the minimum cost is less than
         totalCost then
            S_1.add(i)
            S_{-1}.remove(i)
            totalCost = new
        end
    end
    S_0.add(the rest of items inside S_{-1})
    S_{-1}.remove(the rest of items)
    return S_1, S_0
end
```

The valid solution for this problem is that we at least find a location to build the warehouse. From our pseudo code, the initial total cost is set to INF, so we can ensure that there will be at least one location can get the minimum cost which is smaller than the total cost. Therefore, our greedy algorithm is proved to provide a valid solution.

For our time complexity, due to that we have n locations, the worst case of the outer while-loop will be O(n). Then, the inner for-loop need to go through each item in S_{-1} which contains n items, so it needs O(n) times. Besides, computing the minimum distance needs to consider all cities and locations, which requires for O(m), because the distance between city $j \in \{1, \ldots, m\}$. As a result, the total time complexity is $O(mn^2)$.

3 Approximation for Bin Packing [14 pts]

Suppose that we are given a set of n objects, where the size s_i of the i^{th} object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

This problem is NP-hard. The *first-fit heuristic* takes each object in turn and places it into the first bin that can accommodate it. Let $S = \sum_{i=1}^{n} s_i$.

(a) Argue that the optimal number of bins required is at least $\lceil S \rceil$ (3 pts).
Ans:
We know that $0 < S_i < 1$, so we can put each item in the first bin. If the item does not fit into any of the existing bins, we will open a new bin. That is, if S equals to 1, we will only need 1 bin because every item can fit into the bin. However, if $S > 1$ (let's say S = 1.1), there must exist one item which will make the existing bin become overload. Then, we must open a new bin. Therefore, the optimal number of bins required is at least $\lceil S \rceil$.

(b) Argue that the first-fit heuristic leaves at most one bin less than half full (4 pts).
Ans:
Based on the concept of first-fit heuristic, if the item, S_i, can fit into any of the existing bins, we must put S_i into the existing bin. If there exists more than one bin less than half full, it will violate the concept of first-fit heuristic. Therefore, the first-fit heuristic leaves at most one bin less than half full.

(c) Prove that the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$ (4 pts).
Ans:
Now we are arguing for the worst case of using at most bins. For the worst case, all items are greater than 0.5, which means that only one item can be placed in one bin. Let's say we have N items, so we need N bins. We have to proof that $N <= \lceil 2S \rceil$.

$$=> S = \sum(S_i) > N * 0.5$$

for all $S_i > 0.5$

So, $N < S/0.5 = 2S <= \lceil 2S \rceil$, we successfully proof that $N <= \lceil 2S \rceil$

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

(d) Prove an approximation ratio of 2 for the first-fit heuristic (3 pts).
Ans:
From part b., we know that if we use N bins, we will have at least N-1 bins more than half full.

Proof
Assume the number of bins used in the optimal solution is OPT.
$OPT >= \sum(S_i) > (N-1)/2$
$2 * OPT > N - 1 >= N$

Therefore, we proof that the approximation ration of first-fit heuristic is 2.

4 Bonus: Some DP Problem (7 bonus points)

Evelyn hosted this year's Thanksgiving feast at her place. She invited her family and friends, who all brought gifts for her children. Almost like any other siblings, Charlie and Alan were again fighting over who should get what gifts. Now, to resolve the conflict between them, Evelyn went to seek help from her housekeeper Berta. Berta suggested Charlie and Alan to play the following game:
She arranged all the n gifts in a row having values from $v(1), \cdots, v(n)$, where i^{th} gift has value $v(i)$. Assume n is even. Both of the players get their chances to play, in an alternate fashion. In each turn, a player can either select the first or the last gift, removes it from the row and gains the corresponding value of the gift. Now, as Charlie is always mean to Alan, Evelyn decides that Alan should go first. Give a dynamic programming algorithm to determine the maximum possible value that Alan can definitely accumulate.

(a) (3 pts) Give the subproblem and prove optimal substructure for this problem.

Ans:

In this problem, if we use a straight forward thought to choose the gift, it might not provide the optimal solution. For example, each turn Alan decides to choose the gift from the max(first, last) value. It seems that Alan can get a nice result. However, if the row of the gifts look like this $\{2, 4, 15, 8\}$, Alan will gets the total value of 12 (8+4) which is not the best choice (17 = 2+15).

Due to that n is even, Alan can use a strategy to pick the gifts in pairs. For instance, four gifts with value $\{2, 4, 15, 8\}$. Then, Alan can decide to pick 8+4 or 2+15 based on his opponent Charlie will definitely pick the maximum value, too.

Now we get the subproblem that we have to find the max value of the two choices:

Suppose the front index is i and the end index is j.
Assume a function T(x, y) represents the maximum value between index x and y. Then, we can get the following substructure.

1. T(i+2, j) => Both pick the front gift (When Alan picks gift v_i and Charlie picks gift v_{i+1})
2. T(i+1, j-1) => One pick the front and the other pick the end. No matter Alan picks the front or the last gift, he should always take the value of this pair of gifts into consideration.
3. T(i, j-2) => Both pick the gift in the end (When Alan picks gift v_j and Charlie picks gift v_{j-1})

With the above substructure, Alan now can consider how to choose the gifts:

1. Pick the first gift v_i and the opponent will pick the Max(v_{i+1}, v_j).
   => Then, Alan can only choose from the min(T(i+2, j), T(i+1, j-1))
2. Pick the last gift v_j and the opponent will pick the Max(v_i, v_{j-1}).
   => Then, Alan can only choose from the min(T(i+1, j-1), T(i, j-2))

As a result, Alan should choose the gifts by using the above methods. That is, Alan makes the decision based on:
=> Max(v_i + min(T(i+2, j), T(i+1, j-1)), v_j + min(T(i+1, j-1), T(i, j-2)))

Due to that each time we consider taking the max value of gifts in pairs, we can prove that we get the optimal substructure. For example, again we have the gifts row $\{2, 4, 15, 8\}$. Now implementing the above substructure,
=> Max(2+min(max(4, 15), max(15, 8)), 8+min(max(4, 15), max(2, 4))) = 17
Which provide us the maximum value of gifts and thus we get the optimal substructure.

(b) (4 pts) Write the recurrence (include base cases).
Ans:

Recurrence:

=> T(i, j) = Max(vi + min(T(i+2, j), T(i+1, j-1)), vj + min(T(i+1, j-1), T(i, j-2))) where i is the front index and j is the end index

Base case should occur when j == i+1:

=> T(i, j) = Max((v_i, v_j)