**Algorithms and Complexity**

**2018**

**Mästarprov 2: Complexity**

1. To show that this problem is in NP we need to prove that an instance of the problem can be verified in polynomial time. The verification can be done easily by counting the number of edges and vertices in the set and check if the number of vertices is and the number of edges is .

If and then we are asking if there is a complete subgraph of size in the graph . Then this problem is equivalent to the clique problem which is a NP-complete problem. Thus, the original problem is a NP-complete problem as well.

1. A solution to this problem can be easily checked by first check if all small rectangles fulfills the requirements and then check if any two small rectangles overlaps each other. The time complexity here is which is a polynomial. So, the problem is a NP-problem.

To show that this problem is a NP-complete problem we shall reducing partitioning problem to this problem. Assume that the inputs to the partition problem are . Let the sum of these input be . Let the big rectangle have the size and . Let the small rectangles have the size and we want to insert the small rectangles into the big rectangle. If then the rotation does not matter. If then the small rectangle cannot be rotated since it exceeds the size of the big rectangle. So, the only problem is when . Assuming that there are small rectangles with the size . Iterate through to 0 and for each one of them, let it be . In each iteration, let the input be , and all small rectangles except of them. Stop the iteration at the situation when the iteration gives a ‘no’ as answer and iteration gives a ‘yes’ for answer. If none of the situation gives ‘yes’ as the answer, then the partition problem has answer ‘no’. If the returning is ‘yes’ at the beginning then we have two situations. If is even then the partition problem has answer ‘yes’. If is odd then we rerun the problem with the following input: , and all small rectangles except of the rectangles and two extra rectangles with size and . If the answer is ‘yes’ then the partition problem has answer ‘yes’ otherwise ‘no’. When we find the according to the criterial above, then the same procedure can be performed to check the answer. Then this is a reduction from the partition problem to the coverage problem. So the coverage problem is a NP-complete problem.

1. First, we prove that this problem belongs to NP. A solution to this problem shall include all people so that they form a OlNe-group. So, the solution can be checked by iterating through all members contained in the solution. We separate the members into two set. One of the set contains those members that only knows one person. The other group contains those members that knows more than one person. This is done in time . Then if the number of elements of any groups is not then the solution is false. Then check if all of the member in the first set knows different persons by tabling. If not, the solution is false. This is done in time . At the end, check the second group if all members know each other. If not, then the solution is false. Otherwise the solution is true. This is done in time . The total time complexity of this checking algorithm is then . This is a polynomial so the problem is in NP.

Then we try to reduce the clique problem to this problem. First, we take an input graph and an input integer for the clique problem. If then the clique problem can be solved trivially. Then we add a leaf to or remove leaves from each vertex in that do not connect to exact one leaf. Since these new leaves shall not create clique or remove clique. This process can be done in time complexity by iterating through the graph. The resulting graph can be used as the input to the OlNe-problem. If it returns ‘yes’ then the answer to the corresponding clique problem is ‘yes’ and if it returns ‘no’ then the answer to the corresponding clique problem is ‘no’. Since the clique problem is a NP-complete problem, the OlNe-problem is a NP-complete problem as well.

1. The construction of the partitioning of the books can be done in the following way. First we sort the weights in a decreasing order: . Then perform the following iteration:

For each from to :

Let , resulting\_list = empty\_list;

Call recursion\_algorithm(weight\_list, , , resulting\_list\_pointer)

Then resulting\_list contains the books for the th box.

Recursion\_algorithm(weight\_list, , , resulting\_list\_pointer):

For each weight from to in the weight\_list:

Let

If then append to resulting list;

Call recursion\_algorithm(weight\_list\_except\_, , , resulting\_list\_pointer)

The first sorting part has the complexity on average with quick-sort. In the iteration part, the recursion is called times for each and shall decrease by 1 for each iteration. The recursion algorithm calls the function times each iteration because the input which denotes the start location of the recursion always increase. So totally the algorithm is called times which make the time complexity of this part to be . If the time complexity of the algorithm is , the total complexity is then: