Homework 5

CIS-675 DESIGN AND ANALYSIS OF ALGORITHMS

PROF. IMANI PALMER

6/11/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

Question 1:

The Subset Sum problem is as follows: Given a set of integers *S* and an integer *t*, is there a subset *S'* of *S* such that the sum of all elements in *S'* is equal to *t*? The Equal Partition problem is as follows: Given a set of integers *K*, is it possible to split *K* into two sets such that two subsets have the same sum? Show that Equal Partition reduces to Subset Sum.

The Subset Sum problem can be solved by finding the sum of every possible combination of elements of S and comparing them to integer *t*. To find every possible combination, a binary table was used.

Suppose the input array was {1, 2, 3} with a search value of 6.

Because the array is of size 3, the following binary table would be generated:

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Each of the 8 binary elements would be multiplied by the respective element in the input array and the product added to the sum for that iteration. If the sum equals the search value, *t*, then a 1 is returned. The implementation is written in C below.

```c
#include <stdio.h>
#include <stdlib.h>
#define N 4

// function that returns base^exponent
int power_func(int base, int exponent){
   int i;
   int answer = base;
   for(i = 1; i < exponent; i++){
      answer *= base;
   }
   return(answer);
}

//---------------------------------- algorithm ------------------------------
---------------//
// Does there exist a subset of array s.t. the sum of the subset equals value?
// Return 1 for TRUE and 0 for FALSE
int SubsetSum(int* array, int value){
   int i, j;
   int range = power_func(2, N); // 2^N
   int** binary_array = (int**)malloc(range * sizeof(int*));
   int sum = 0;
   for(i = 0; i < range; i++){
```

```
      binary_array[i] = (int*)malloc(N*sizeof(int));
   }

   // construct binary table
   for(i = 0; i < range; i++){
      for(j = 0; j < N; j++){
         int num1 = power_func(2, j+1);
         int num2 = num1/2;
         if((i%num1) >= num2){
            binary_array[i][j] = 1;
         }
         else{
            binary_array[i][j] = 0;
         }
         sum += (array[j] * binary_array[i][j]);
      }
      if(sum == value){
         return(1);
      }
      sum = 0;
   }
   return(0);
}

int main(){
   int array[N] = {1, 2, 0, -2};
   int number = 3;
   int boolean_value = SubsetSum(array, number);
   if(boolean_value == 1){
      printf("TRUE\n");
   }
   else{
      printf("FALSE\n");
   }
}
```

The output of the code is: `TRUE`

To solve the Equal Partition problem, calculate the sum of the elements in set *K*, called SUM. If SUM is odd, then *K* cannot possibly be split into two subsets of the same sum (return FALSE). If SUM is even, then search for a subset *K'* whose sum equals SUM/2. The Subset Sum problem mirrors this part of the Equal Partition problem. If *K'* is found, return TRUE. Else return FALSE.

The implementation is written in C and is below.

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#define N 4

// function that returns base^exponent
int power_func(int base, int exponent){
   int i;
   int answer = base;
   for(i = 1; i < exponent; i++){
      answer *= base;
   }
   return(answer);
}

//------------------------- algorithm ---------------------------------//
// Does there exist a subset of array s.t. the sum of the subset equals value?
// Return 1 for TRUE and 0 for FALSE.
int SubsetSum(int* array, int value){
   int i, j;
   int range = power_func(2, N); // 2^N
   int** binary_array = (int**)malloc(range * sizeof(int*));
   int sum = 0;
   for(i = 0; i < range; i++){
      binary_array[i] = (int*)malloc(N*sizeof(int));
   }

   // construct binary table
   for(i = 0; i < range; i++){
      for(j = 0; j < N; j++){
         int num1 = power_func(2, j+1);
         int num2 = num1/2;
         if((i%num1) >= num2){
            binary_array[i][j] = 1;
         }
         else{
            binary_array[i][j] = 0;
         }
         sum += (array[j] * binary_array[i][j]);
      }
      if(sum == value){
         return(1);
      }
      sum = 0;
   }
   return(0);
}
```

```
//---------------------algorithm-------------------------//
// Can array be split into 2 subsets where their cumulative sums are equal?
// Return 1 for TRUE and 0 for FALSE.
int EqualPartition(int* array){
   int i;
   int SUM = 0;
   int half_SUM = 0;
   for(i = 0; i < N; i++){
      SUM += array[i];
   }

   half_SUM = SUM/2;

   // if half the cumulative sum is odd, return FALSE.
   if((half_SUM % 2) != 0){
      return 0;
   }

   return(SubsetSum(array, half_SUM));
}


int main(){
   int array[N] = {5, 2, 0, -3};
   int number = 3;
   int boolean_value = EqualPartition(array);
   if(boolean_value == 1){
      printf("TRUE\n");
   }
   else{
      printf("FALSE\n");
   }
}
```

The output of the code is: TRUE

Question 2:

Vertex Cover Problem: A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either u or v is in vertex cover. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.

Set Cover Problem: Given a set of elements $\cup$ (called the universe) and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest subcollection of S union equals the universe.

For example, consider the universe $\cup = (1, 2, 3, 4, 5)$ and the collection of sets $S = (1, 2, 3), (2, 4), (3, 4), (4, 5)$. Clearly the union of S is $\cup$. However, we can cover all of the lements with the following smaller number of sets:(1,2,3),(4,5)

Prove the Vertex Cover problem reduces to the Set Cover Problem

The Set Cover problem is solved by searching through the collection of sets '*S*' for the set with the most unique elements in U. Call this set "Best," and add it to the set cover. Remove those elements in Best from U. While U is not empty, continue searching for optimal sets in S who contain the most elements left in U. Add it/them to the set cover.

The Vertex Cover problem is solved by searching through the undirected graph for the vertex touching the most edges. Call this vertex "Best," and add it to the vertex cover. Consider the set of all edges in the graph, E. Remove from E all edges touching the Best vertex. While E is not empty, continue searching for vertices who touch the most edges remaining in E. Add these vertices to the vertex cover.

The Vertex Cover problem reduces to the Set Cover problem.

Set Cover Sudo Code:

```
Set Cover = {}
U = {all elements}
While U is non-empty:
        Best = S[0] // initialize Best with first set in S
        For(i = 1; i < size_of_S; i++): // find the most optimal set
                If S[i] has more unique elements in U than Best:
                        Best = S[i]
        U.remove(elements in Best)
        Set Cover.add(Best)
return(Set Cover)
```

Vertex Cover Sudo Code:

```
Undirected Graph G;
Vertex Cover = {}
E = {all edges}
While E is non-empty:
```

```
        Best = any vertex in G
        For(all vertices in G):
                if a vertex 't' in G has more edges in E than Best:
                        Best = t
        E.remove(all elements in Best)
        Vertex Cover.add(Best)
return(Vertex Cover)
```

Question 3:

We discussed the Hamiltonian Cycle problem for undirected graphs. Define an equivalent problem, called D-Hamiltonian Cycle, for directed graphs (i.e., given a directed graph, does it contain a cycle that vists every node exactly once?). Show that D-Hamiltonian Cycle is NP-Complete.

To show that D-Hamiltonian Cycle is NP-Complete, it must be shown that it is NP Hard and can be solved with a non-deterministic polynomial time algorithm.

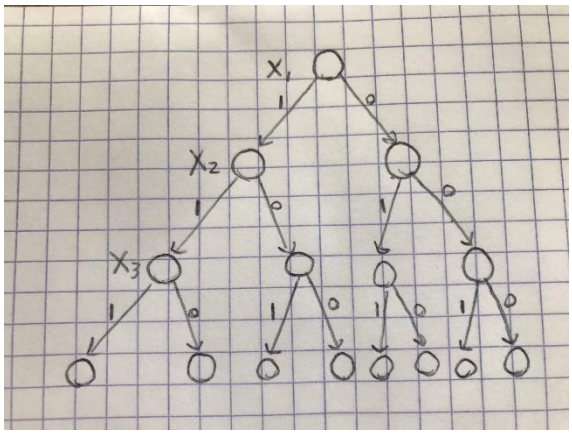
To prove that it is NP-Hard, compare it to the Boolean Satisfiability problem, which is a well-known NP-Complete problem. Show that the Boolean Satisfiability problem reduces to the D-Hamiltonian Cycle problem.

The Boolean Satisfiability problem is below.

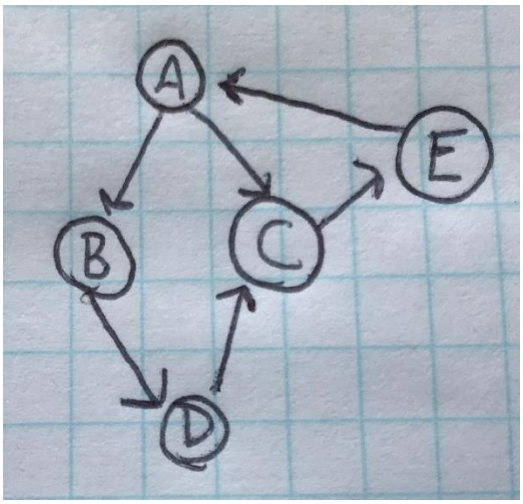
Suppose there is a formula $(X_1 \vee not(X_2) \vee X_3) \wedge (X_1 \vee X_2 \vee not(X_3))$. Determine the values of X_i which produce the result TRUE. To find these values, all possible combinations must be tested.

X_1	X_2	X_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Space-State Tree of Boolean Satisfiability problem:



The problem takes exponential time to solve: $O(2^n)$.



In the above example, the D-Hamiltonian Cycle is ABDCEA.

From the State-Space Tree of the Boolean Satisfiability problem, one can visually see that the process will be similar searching through the directed graph for the D-Hamiltonian Cycle. The D-Hamiltonian Cycle problem can be solved using a Depth-First Search from a starting vertex. Whenever a vertex has two or more outgoing edges, the algorithm must travel all edges to find a path back to the starting vertex (cycle). If there exists a cycle such that all vertices are visited exactly once, a D-Hamiltonian Cycle exists. The two problems both utilize the same logic of testing all possible combinations (traveling all paths) until a solution is found. Because the D-Hamiltonian Cycle problem reduces to the Boolean Satisfiability problem, D-Hamiltonian Cycle is NP-Hard.

To show that there is a non-deterministic polynomial-time algorithm, consider if every edge explored by the algorithm was “correct,” which is to say is an edge existing in the D-Hamiltonian Cycle. If this were the case, the algorithm would find the D-Hamiltonian Cycle in linear time because no time would be wasted exploring incorrect paths.

Because it is NP-Hard and there exists a non-deterministic polynomial-time algorithm to solve it, the D-Hamiltonian Cycle problem is NP-Complete.

Question 4:

The Subset Sum problem is as follows: Given a set of numbers $S = (s_1, s_2, ..., s_n)$ and a target value t, is there some subset S' of such that the sum of numbers in S' is equal to t? We saw earlier that this problem is NP-Complete. Design a reasonable backtracking algorithm to solve the Subset Sum problem. As always, make sure to include enough detail for us to implement your algorithm.

The backtracking algorithm will subtract elements from the desired input value 't'. If the difference equals zero, then a boolean value of 1 is returned. If the difference is greater than 0, then the next element from the set is analyzed. If the difference is less than 0, the element is skipped, and the next element is analyzed. See implementation below. Runtime is $O(2^n)$.

```
#include <stdio.h>
#include <stdlib.h>
#define N 4

int Backtracking_algorithm(int* array, int value, int index){
   int i, answer;
   for(i = index; i < N; i++){
      if(value - array[i] == 0){
         return(1);
      }
   }
   if(index+1 < N){
      if(value-array[index] > 0){
         return Backtracking_algorithm(array, value-array[index], index+1);
      }
      if(value-array[index] < 0){
         return Backtracking_algorithm(array, value, index+1);
      }
   }
   return(0);
}

int main(){
   int array[N] = {1, 0, 1, 1};
   int number = 3;
   int answer = Backtracking_algorithm(array, number, 0);
   if(answer == 1){
      printf("Sum found.\n");
   }
   else{
      printf("Sum not found.\n");
   }
```

```
}
```

The output of the program is below.

```
Sum found.
```

5 Question

Consider the N Queens problem, where we had to place N queens on an N-by-N chess board such that no two queens were in the same row, column, or diagonal (i.e., no two queens are threatening each other). We designed a simple local search algorithm for this problem as follows:

First, because we know that each column can contain only one queen, assign each queen to her own column. Place each queen at random location within her column. Calculate the number of pairs of queens that are threatening each other (e.g., if Q is threatened by both Q_2 and Q_3, this counts as 2 threats). For each queen Q, consider moving queen Q, to a different location in the same column, and calculate the new number of conflicts that would exist if you performed that move. Once you have performed these calculations, perform the move that results in the greatest reduction in the current number of conflicts (subject to the constraint that each queen must remain in her assigned column). If there is no move that reduces the current number of conflicts, then terminate. Repeat until termination.

Give an example showing that this algorithm may fail to find a correct solution. (Hint: try $N = 4$. Show that (a) there is a solution in which all queens are safe, and (b) there is some starting position and some sequence of moves following the above description that fails to find such a solution.)

A) The solution where all the queens are safe is below.

		Q_3	
Q_1			
			Q_4
	Q_2		

B) Consider the starting position below.

Q_1			
		Q_3	
			Q_4
	Q_2		

There is one conflict present, and it is between Q_3 and Q_4. The algorithm described will attempt to find a move that reduces the number of conflicts from 1 to 0. There is no such move, so the algorithm will terminate. When a queen is placed in a corner position, the rest of the queens have a 3x3 plot on which to move. It is impossible to successfully place the queens in a 3x3 board where no conflicts are present.