CSE417_HW8

3.

(b)

Max Cut Algorithm	

Def MaxCut(G(V,E)):
    V1 is a subset of V
    V2 = V – V1
    change recorder = False
    while change recorder == False:
            change recorder = True   # recorder reset
            Go through all nodes in V1 and check if I put it into the other side will the number of cross edges increase?
                    If cross edge number increases:
                            Put the node into V2
                            Change recorder = False
            Go through all nodes in V2 and check if I put it into the other side will the number of cross edges increase?
                    If cross edge number increases:
                            Put the node into V1
                            Change recorder = False
    end while
return V1
	
	O(n)
	
	O(n)

Time complexity:

In this algorithm, I set up sets V1 and V2 and check if I put a node from this part to the other part, will it increase the number of cross edges? And this while loop will terminate when all nodes stay in the same place (number of cross edges will no longer increase even I switch every node to the other side). And this process will lead to $O(n^n)$ time complexity.

Correctness:

In the beginning, I will randomly set up sets V1 and V2 belong to V and V-V1=V2. And then make it check if there is any u belongs to V1 when change to V2 that will increase the number of cross edges then I will move the node u to the other side. In this procedure, the number of cross edges will increase at least by 1 and not 0. By doing this procedure every time, even though I will spend a lot of time search and compare the edge numbers from each node, in the end of the procedure, I will still reach to value of max number of cross edges. Thus I can return the resulting set either V1 or V2 as the answer.

5.

Euler Tour Algorithm	Runtime
Def EulerTour(G (V,E))	
Empty stack	
Set of Original Vertices = V	
Set of Original Edges = E	
Euler path edge = []	
Current Node = None	
Previous Node = None	
If all vertices have even edges:	O(E+V)
Choose any one of them to be Start Node O	
Current Node = O	
Elif exactly 2 vertices have two odd degree:	O(E+V)
Choose either of them to be the Start Node O	
Current Node = O	
Else:	O(1)
Print(“No Euler tour path for this graph”)	
While Current Node has no neighbor and Stack is Empty:	O(1)
If Current Node has no out-going edges :	O(V+E)
Previous Node = Current Node	
Current Node = Stack.pop()	O(E)
Add edge (Previous node, Current Node) to Euler path edge	
Else:	
Stack.append(Current Node)	
Previous Node = Current Node	O(1)
Current Node = one of Current Node’s neighbors	
Delete the edge (Previous Node, Current Node) from E	
Return Euler path edge	

Runtime Analysis:

$O(E+V)+O(E+V)+O(1)+O(1)*(O(V+E)+O(E)+O(1)) = O(V+E)$

Correctness:

From this Algorithm, in the beginning, I will check if all vertices have even degree, in this situation, there must be a solution for euler path in it. And I will pick one of them to be the starter.

And if not, I will also check if there are exactly two vertices have two odd number of edges, in this sense, I can still pick either one to be the start and in the end, the other one would be the end.

If all of them doesn’t meet, then this graph contains no euler path.

After I checked that the graph within a solution, I start to use the starter to look around his neighbors, if there are any neighbors around it, I will add the current vertex to the stack and take any of its neighbors and removethe edge between selected neighbor and that vertex and set the neighbor as the current vertex. And if current vertex has no neighbors I will remove the last vertex from the stack and set it as current vertex and add a Euler path edge between the current vertex and previous one. By doing so till the current vertex has no more neighbors and the stack is empty and then return the Euler Path edges (it’s a reversed edge).

4.

(b)

K partition problem Algorithm	Runtime
Input S is a set of integers	
n is number of items in S	
k is the target value of partition	
Answer = KPP(S,n,k)	
Def KPP(S, n, k):	
if n<k:	O(1)
return None	
SOS = sum of S from S[1] to S[n]	O(n)
sumLeft = []	
For i from 0 to k:	O(k)
sumLeft[i] = SOS/k	
bool res = !(SOS % k) && subsetSum(S,n-1,sumLeft,A,k)	T(n-1)
if (!res):	O(1)
print(“kPP isn’t possible”)	
return	
for i from 0 to k:	O(k)
print(“Partition”, i , “ is”)	
for j from 0 to n:	O(n)
if A[j] == i + 1	O(1)
print(“ this is ”,S[j])	
Def subsetSum(S, n, sumLeft, A, k):	
If checkSum(sumLeft,k):	O(k)
Return True	
If n < 0 :	O(1)
Return False	
bool res = False	
for i from 0 to k:	O(k)
if (!res && (sumLeft[i] – S[n]) >=0):	O(1)
A[n] = i + 1	O(1)
sumLeft[i] = sumLeft[i] – S[n]	O(1)
res = subsetSum(S,n-1,sumLeft,A,k)	T(n-1)
sumLeft[i] = sumLeft[i] + S[n]	O(1)
return res	

Def checksum(sumLeft, k):	
r = True	O(1)
for i from 0 to k :	O(k)
if sumLeft[i] != 0	O(1)
r = false	
return r	

Runtime analysis:

For subsetSum it's T(n) = T(n-1) and O(k)=> $O(k*n^2)$

For total KPP it's O(k)+O(n)+$O(k*n^2)$+O(k)*O(n) = $O(k* n^2)$

Correctness:

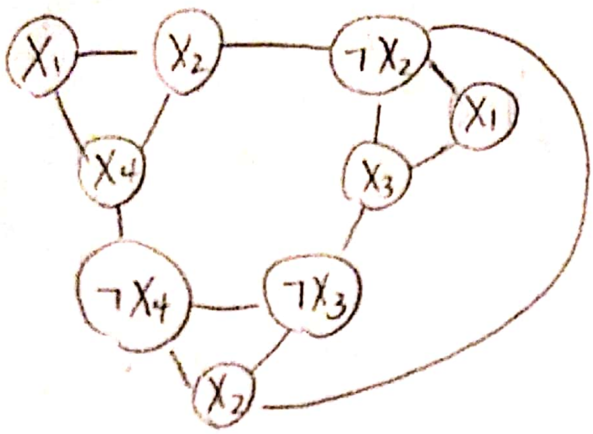
I start by calculating the sum of all elements in the set. If sum is not divisible by k, we cannot divide the array into k subsets with equal sum. However, if sum is divisible by k, we still need to check if k subsets with sum of elements equal to (sum/k) exists or not. We can find this by considering each item in the given array one by one and for each item we include it in the i'th subset and recurse for remaining items with remaining sum. We backtrack if solution is not found by including current item in i'th subset and try for (i+i)th subset.

In the end, we will return true and print the subsets when k subsets each with zero sum are found (termination step). For printing the partitions, I maintain anseperate array A[] to keep track of subsets elements. If the value of A[i] is k, then it means that i'th item of S is part of k'th subset.

CSE 417

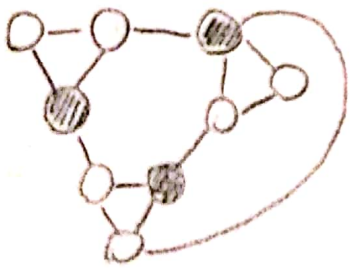
1. $(x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$

(a)



interger $k = 3$

(b) $x_1 = x_2 = x_3 = \text{False}$, $x_4 = \text{True}$



independent set

$\rightarrow$ $\boxed{\neg x_2,\ x_4,\ \neg x_3}$

$\downarrow$

corresponding assignments

① $[\neg x_2,\ x_4,\ \neg x_3,\ x_1]$

② $[\neg x_2,\ x_4,\ \neg x_3,\ \neg x_1]$

Yes, they are both satisfying assignments.

2. $(x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$

		Variables				Clauses		
		x_1	x_2	x_3	x_4	$(x_1 \vee x_2 \vee x_4)$	$(x_1 \vee \neg x_2 \vee x_3)$	$(x_2 \vee \neg x_3 \vee \neg x_4)$
Literals	W_1 (x_1)	1	0	0	0	1	1	
	W_2 $(\neg x_1)$	1	0	0	0	0	0	1
	W_3 (x_2)		1	0	0	1	0	0
	W_4 $(\neg x_2)$		1	0	0	0	1	1
	W_5 (x_3)			1	0		1	0
	W_6 $(\neg x_3)$			1	0		0	0
	W_7 (x_4)				1	1		1
	W_8 $(\neg x_4)$				1	0		
slack	W_9 (S_{11})					1	0	0
	W_{10} (S_{12})					1	0	0
	W_{11} (S_{21})						1	0
	W_{12} (S_{22})						1	0
	W_{13} (S_{31})							1
	W_{14} (S_{32})							1
		1	1	1	1	3	3	3

$\left(\begin{array}{l} x_2 = x_3 = \text{False} \\ x_1 = x_4 = \text{True} \end{array} \right.$

corresponding subset

$[x_1, \neg x_2, \neg x_3]$

$[x_1, x_1, \neg x_3]$

$[x_4, \neg x_2, \neg x_3]$

$[x_4, x_1, \neg x_3]$

corresponding assignments are: satisfying ?

$[X_1, \neg X_2, \neg X_3] \rightarrow \left(\begin{array}{l} [X_1, \neg X_2, \neg X_3, X_4] \\ [X_1, \neg X_2, \neg X_3, \neg X_4] \end{array} \right.$ ✓ ✓

$[X_1, X_1, \neg X_3] \rightarrow \left(\begin{array}{l} [X_1, \neg X_3, X_2, X_4] \\ [X_1, \neg X_3, \neg X_2, \neg X_4] \\ [X_1, \neg X_3, X_2, \neg X_4] \\ [X_1, \neg X_3, \neg X_2, X_4] \end{array} \right.$ ✓ ✓ ✓ ✓

$[X_4, X_2, X_3] \rightarrow \left(\begin{array}{l} [X_4, X_2, X_3, X_1] \\ [X_4, X_2, X_3, \neg X_1] \end{array} \right.$ ✓ ✓

$[X_4, X_1, \neg X_3] \rightarrow \left(\begin{array}{l} [X_4, X_1, \neg X_3, X_2] \\ [X_4, X_1, \neg X_3, \neg X_2] \end{array} \right.$ ✓ ✓

Yes, they are all satisfying.

3. Max Cut problem: known: $G=(V,E)$, k.

problem: Is there a partition of the vertices into two (nonempty, nonoverlapping) subsets V_1 and V_2 so that k or more edges have one end in V_1 and the other end in V_2.

Input: An undirected graph $G=(V,E)$ and integer k

Output: A cut $(V_1; V_2)$ where $V_1 \subset V$, $V_2 \subset V$, $V_1 \neq \phi$, $V_2 \neq \phi$, $V_1 \cap V_2 = \phi$ so that $|E(V_1, V_2)| \geq k$ where $E(V_1,V_2) = \{(u,v) \mid u \in V_1, v \in V_2\}$

Complexity	procedure $m(X, h)$
	if
$O(1)$	X is a well-formed representation of a graph $G=(V,E)$ and an integer k
	and
$O(1)$	h is a well-formed representation of a cut (V_1, V_2)
	and
$O(1)$	$E(V_1,V_2) = \{(u,v) \mid u \in V_1, v \in V_2\}$, $\lvert E(V_1,V_2)\rvert \geq k$
	and
$O(n^2)$	$V_1 \subset V$, $V_2 \subset V$, $V_1 \neq \phi$, $V_2 \neq \phi$, $V_1 \cap V_2 = \phi$.
	and
$O(n^2)$	V_1 is a vertex set of G, V_2 is a vertex set of G.
	then
$O(1)$	output "Yes"
	else
$O(1)$	output "I'm not convinced"

3. (i) the certificate is the cut (V_1, V_2)

(ii) $2 \le$ the length of the certificate (the cut (V_1, V_2)) $\le |V|$

(iii)

- By checking the set of edges $E(V_1, V_2) = \{(u,v) \mid u \in V_1, v \in V_2\}$,

  ① check the number of $|E(V_1, V_2)| \geq k$ to make sure there are k or more edges have one end in V_1 and one end in V_2.

  ② check $V_1 \subset V$, $V_2 \subset V$, $V_1 \neq \phi$, $V_2 \neq \phi$, $V_1 \cap V_2 = \phi$, $V_1 \cup V_2 = V$ to make sure the subsets V_1, V_2 of V to be nonempty, nonoverlapping subsets of V.

  By doing this, I can make sure this is a "Yes" instance of MaxCut.

- If the $|E(V_1, V_2)| < k$ or $V_1 \not\subset V$ or $V_2 \not\subset V$ or $V_1 = \phi$ or $V_2 = \phi$ or $V_1 \cap V_2 \neq \phi$ then it won't output "Yes".

  If the "no" instance appears, at this time, if it meet the criteria to be "Yes" instance, then there shouldn't be a "no" instance, $\Rightarrow\Leftarrow$.

  On the other hand, it it doesn't meet the criteria then it will be a "no" instance.

  Thus, there is no way for a no instance to fool the verifier to say "Yes".

  Q.E.D

3. (iv) MaxCut instances : $\left| \begin{array}{l} k = q \\ G = (V, E) \end{array} \right.$ $V_1 \cap V_2 \subseteq V$

$E(V_1, V_2) = \{(u, v) \mid u \in V_1, v \in V_2\}$.

$\subseteq E$

length of instances : number of vertices and edges $= n + m$

length of the hint : g, where $2 \le g \le n$ (where $0 < m < n^2$)

(g . m is bounded by n)

Asyptotically, as a function of length of MaxCut instance n

In the line of $V_1 \subset V$, $V_2 \subset V$, $V_1 \neq \phi$, $V_2 \neq \phi$, $V_1 \cap V_2 = \phi$

this will take me $O(n^2) + O(n^2) + O(1) + O(1) + O(n^2) = O(n^2)$

and In the line of $|E(V_1, V_2)| \geq k$

this will take me $O(1)$ time

so, totally it would $O(n^2)$ time

thus, I find a verifier that can examine

the solution to be feasible or not in polynomial time,

$\Rightarrow$ It's a NP problem

QED

4. k-Partition Problem

Problem: Is it possible to partition a sequence of positive integers into k groups having equal sums

Input: a sequence of positive integers $(w_1, w_2, \ldots, w_n)$

output: an n-vector h each of whose entries is integer in the range $1, \ldots, k$

procedure $V(x, h)$

if:

x is sequence of integers $(w_1, w_2, \cdots, w_n)$ and an integer k

and

h is an n-vector whose each entries is positive integer in the range $1, \ldots, k$

and

$$\sum_{i \in h^{-1}[j]} w_i = \frac{1}{k} \sum_{i=1}^{n} w_i \quad (1 \le j \le k)$$

and

$$h^{-1}[j] = \{ i \mid h[i] = j \} , (1 \le j \le k)$$

then

output "Yes"

else:

output "I'm unconvinced"

4.

(a)

(i) the "hint" is the n-vector h

(ii) the length of h is n

(iii) the verifier will check all entries of h to be positive

and check the summantion of both sides to be the same

$$\left(\sum_{i \in h^{-1}[j]} W_i \right) = \frac{1}{k} \sum_{i=1}^{n} W_i$$

and check $h^{-1}[j] = \{ i \mid h[i] = j \}$

then verifier will output "Yes"

For those instances who is not all entries to be positive integer

or the summantion of both side not the same

$$\left(\sum_{i = h^{-1}[j]} W_i \right) \neq \frac{1}{k} \sum_{i=1}^{n} W_i$$

or $h^{-1}[j] \neq \{ i \mid h[i] = j \}$.

then it will fail to output "Yes"

(iv) length of "hint" is k (k groups having equal sum)

length of intances is n $(W_1, W_2 \cdots, W_n)$

the runtime of the verifier would be $O(nk)$

runtime $O(k)$ to check each entry of h to be positive integer

runtime $O(k)$ to check the summantions match

runtime $O(n)$ to check $\sum_{i=h^{-1}[j]} W_i = \frac{1}{k} \sum_{i=1}^{n} W_i$

verify in polynomial time

$\Rightarrow$ It's a NP problem

#

4\. (C) Show that 2-PP $\le_p$ KNAP (right now $k=2$)

(proof) Given a set of n positive integers $N = \{s_1, s_2, \cdots s_n\}$

create the following instance of 0-1 knapsack:

- $a_j = s_j$ • $b = \frac{1}{2} \sum_{j \in N} s_j$
- $c_j = s_j$ • $k = \frac{1}{2} \sum_{j \in N} s_j$

That is, the 0-1 knapsack instance is to decide if the following problem is feasible:

$$\sum_{j \in N} s_j x_j \ge \frac{1}{2} \sum_{j \in N} s_j \qquad \text{—(1)}$$

$$\sum_{j \in N} s_j x_j \le \frac{1}{2} \sum_{j \in N} s_j \qquad \text{—(2)}$$

$$x_j \in \{0, 1\}$$

Given N, we need to make $2(n-1)$ addition and 2 division operations, and store $(2n+2)$ numbers to represent the resulting 0-1 knapsack instance. Clearly, the time to carry this out is polynomially bounded in n.

Right now, Let's move on to show the correctness of Reduction.

☆ Show that N is a "yes" instance of 2-Partition if and only if the reduction produces a "yes" instances of 0-1 knapsack

(look at the back)

① Correctness of the Reduction : "Only If" Direction

Now suppose that the 0-1 Knapsack problem instance resulting from the reduction is a "yes" instance and let $\hat{x}$ be a feasible solution ;

Since $\sum_{j\in N} S_j \hat{x}_j \geq k$ and $\sum_{j\in N} S_j \hat{x}_j \leq b = k$, it must be the case that $\sum_{j\in N} S_j \hat{x}_j = k = \frac{1}{2}\sum_{j\in N} a_j$. Therefore $S = \{j\in N : \hat{x}_j = 1\}$ gives us a partition of N s.t. $\sum_{j\in S} S_j = \sum_{j\in N\setminus S} S_j$.

② Correctness of the Reduction "If Direction"

Suppose that N is a "yes" instance of 2-partition problem. This implies that there are exists at least one subsets $S\subset N$, such that

$$\sum_{j\in S} S_j = \sum_{j\in N\setminus S} S_j = \frac{1}{2}\sum_{j\in N} S_j = b = k$$

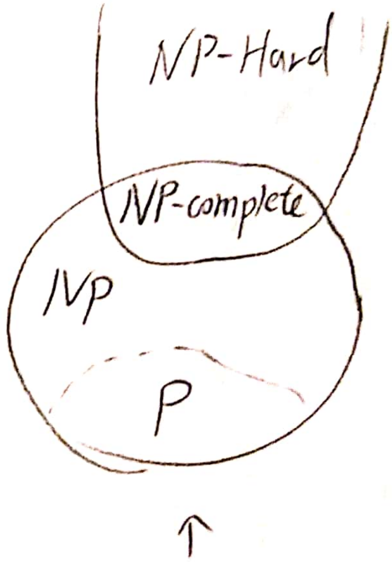
Thus, the solution $x_j = 1$ for all $j\in S$ and $x_j = 0$ $\forall j \notin S$ is feasible for (1) and (2). Therefore, the reduction maps "yes" instances of 2-PP to "yes" instances of 0-1 knapsack.

③ Since N is a yes instance of 2-PP if and only if the reduction produces a "yes" instance of 0-1 knapsack, and since the reduction is polynomial, it follows that 2-PP is polynomially reducible to 0-1 knapsack. $\Rightarrow$ 2-PP $\leq_p$ 0-1 knapsack

QED

4. (d) Suppose there is no polynomial time algorithm for KNAP. Even though in 4.C I showed that 2-PP $\leq_p$ KNAP, I still cannot imply that there is no polynomial time algorithm for 2-PP. Because if I say 2-PP is polynomial reducible to a NP problem, it doesn't contradict to the hypothesis of 2-pp being able to polynomial reducible to a P problem.

Thus, it is still possible for 2-pp to have a polynomial time algorithm for it.



relation of P. NP

NP complete, NP Hard