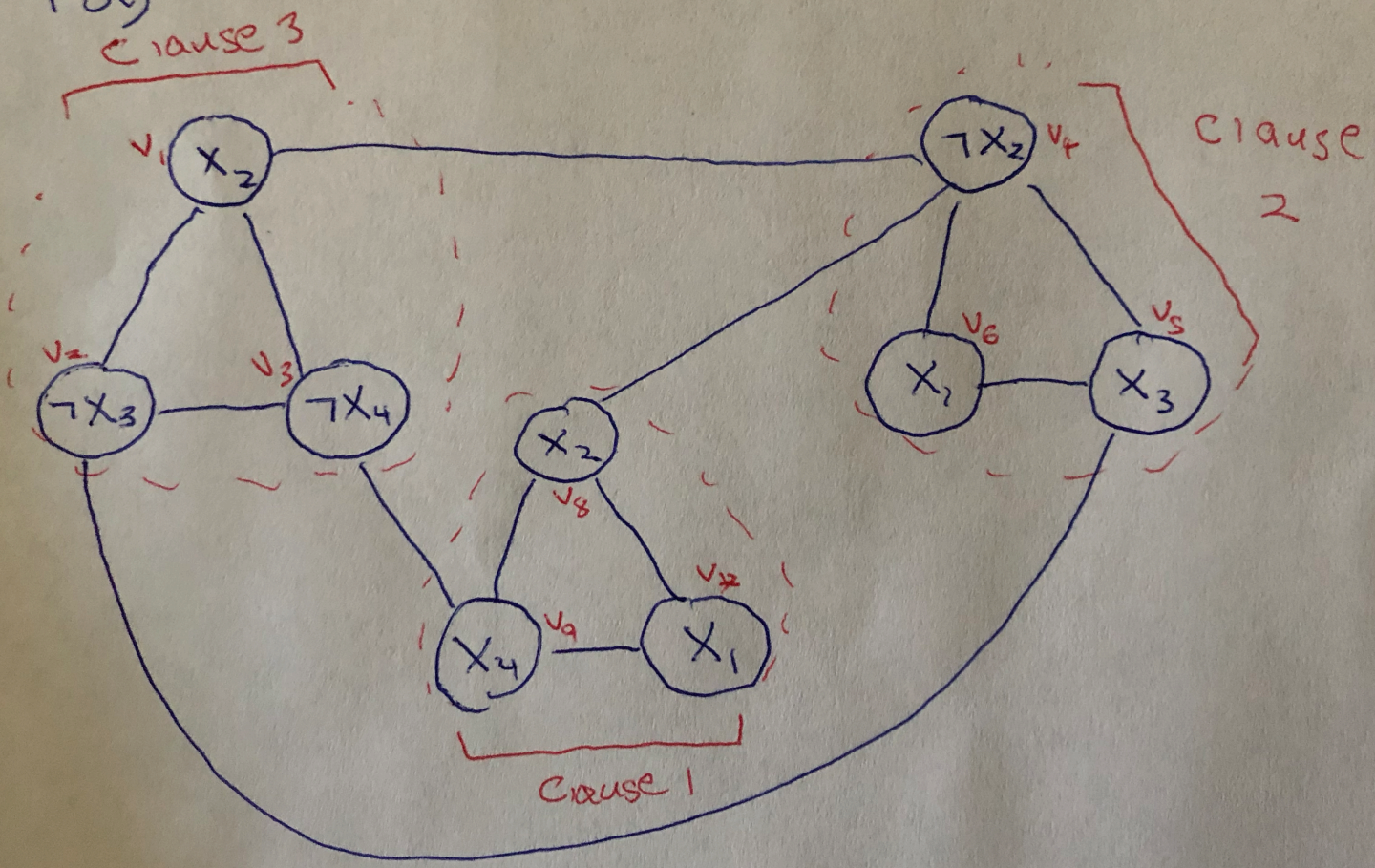
Zachary McNulty (zmcnulty, 1636402)

Problem 1a)

Problem 1a)

K = 3



CSE 417 HW 8

Zachary McNulty (zmcnulty, 1636402)

March 2019

1 Problem 1

a)
see graph picture above

b)

Given the satisfying assignment $x_1 = x_2 = x_3 = False$ and $x_4 = True$ our first task is to pick one $True$ literal from each clause. We can see that in the first clause $(x_1 \vee x_2 \vee x_4)$ only x_4 is true, so we will take the vertex corresponding to x_4 in clause one, v_9,(see graph above) to be part of our set. Next, we can see that in the second clause $(x_1 \vee \neg x_2 \vee x_3)$ that only the literal $\neg x_2$ is true, so we will choose the corresponding $\neg x_2$ vertex of clause 2, v_4, in our graph (see graph above) to be part of our set. Lastly, we can see that in clause 3 $(x_2 \vee \neg x_3 \vee \neg x_4)$ that only $\neg x_3$ is true so we will take the vertex corresponding to $\neg x_3$ of clause 3, v_2 ,in our graph (see graph above) and add it to our set. This yields the set $\{v_2, v_4, v_9\}$ of size 3 and it is easy to verify this is an independent set of the given graph. Since we had no choice at each clause about which literal to add, this is the only valid independent set associated to this satisfying assignment.

Given the independent set $\{v_2, v_4, v_9\}$, we can see this defines the partial truth assignment $x_2 = x_3 = False$ and $x_4 = True$. Note that regardless of the choice of the truth value for x_1, we will have a satisfying assignment of the variables as at least x_4 is true in clause one, at least $\neg x_2$ is true in clause two, and at least $\neg x_3$ is true in clause three. Thus, this independent set generates two satisfying assignments:

$$x_1 = x_2 = x_3 = False, x_4 = True$$
$$x_2 = x_3 = False, x_1 = x_4 = True$$

2 Problem 2

a)

Weights	x_1	x_2	x_3	x_4	$(x_1 \vee x_2 \vee x_4)$	$(x_1 \vee \neg x_2 \vee x_3)$	$(x_2 \vee \neg x_3 \vee \neg x_4)$	weight
$w(x_1)$	1	0	0	0	1	1	0	1000110
$w(\neg x_1)$	1	0	0	0	0	0	0	1000000
$w(x_2)$		1	0	0	1	0	1	100101
$w(\neg x_2)$		1	0	0	0	1	0	100010
$w(x_3)$			1	0	0	1	0	10010
$w(\neg x_3)$			1	0	0	0	1	10001
$w(x_4)$				1	1	0	0	1100
$w(\neg x_4)$				1	0	0	1	1001
$w(s_{11})$					1	0	0	100
$w(s_{12})$					1	0	0	100
$w(s_{21})$						1	0	10
$w(s_{22})$						1	0	10
$w(s_{31})$							1	1
$w(s_{32})$							1	1
C	1	1	1	1	3	3	3	1111333

The given Subset Sum problem would then be to find a subset of the weights above weights:

$$\{w(x_i), w(\neg x_i), w(s_{j1}), w(s_{j2}) : j \in \{1,2,3\}, i \in \{1,2,3,4\}\}$$

that sums to exactly $C = 1,111,333$. We can see the correspondence of each literal to the choice of weights: if $w(x_i)$ is chosen to be part of the optimal sum, then x_i is true in the satisfying assignment, but if $w(\neg x_i)$ is chosen then x_i is false. We can then see that since each of the first four digits of C are one, it cannot be true that there exists a subset that sums to C where both x_i and $\neg x_i$ are chosen or a subset where neither are chosen. Similarly, since the last three digits of C corresponding to each clause are three, but there are only two slack variables for each clause, any optimal subset that sums to C must include at least one true literal in its respective clause.

b)

Given the assignment $x_2 = x_3 = False$ and $x_1 = x_4 = True$, we can generate the corresponding subset sum solution. As x_2 and x_3 are $False$, add weights $w(\neg x_2) = 100010$ and $w(\neg x_3) = 10001$ to our subset. Similarly, as

$x_1 = x_4 = True$ add the weights $w(x_1) = 1000110$ and $w(x_4) = 1100$ to our subset. Consider the column for clause $(x_1 \vee x_2 \vee c_4)$. Since we chose $w(x_1)$ and $w(x_4)$, but not $w(x_2)$ the current column sum is two. Thus, we must add either $w(s_{11})$ or $w(s_{12})$ to our subset to get that column to sum to three (the choice is arbitrary). For the column corresponding to the second clause $(x_1 \vee \neg x_2 \vee x_3)$, since we chose $w(x_1)$ and $w(\neg x_2)$ but not $w(x_3)$, the current column sum is two. To get the column to sum to three as is necessary we must add either $w(s_{21})$ or $w(s_{22})$, the choice is arbitrary. For the column corresponding to the third clause, as we chose $w(\neg x_3)$ but not $w(\neq x_4)$ or $w(x_2)$, we must include both slacks $w(s_{31}), w(s_{32})$. Since we have two points where we could choose either of the two slacks to create a valid subset sum C, this generates the four valid subsets:

$$\{w(x_1), w(x_4), w(\neg x_2), w(\neg x_3), w(s_{11}), w(s_{21}), w(s_{31}), w(s_{32})\}$$

$$\{w(x_1), w(x_4), w(\neg x_2), w(\neg x_3), w(s_{11}), w(s_{22}), w(s_{31}), w(s_{32})\}$$

$$\{w(x_1), w(x_4), w(\neg x_2), w(\neg x_3), w(s_{12}), w(s_{21}), w(s_{31}), w(s_{32})\}$$

$$\{w(x_1), w(x_4), w(\neg x_2), w(\neg x_3), w(s_{12}), w(s_{22}), w(s_{31}), w(s_{32})\}$$

Since all these assignments select the same subsets of

$$\{w(x_i), w(\neg x_i) : i \in \{1, 2, 3, 4\}\}$$

and a literal x_i or $\neg x_i$ is true iff it is selected to be part of this subset, all the above subsets of the weight correspond to the same satisfiability problem. Since subset sum reduction requires either $w(x_i)$ or $w(\neg x_i)$ to be chosen, there are no literals left unassigned so each subset generates exactly one satisfiable assignemt, the one used to build them: $x_2 = x_3 = False$ and $x_1 = x_4 = True$.

3 Problem 3

1. (a) Prove that MaxCut is in NP
   i. Let the hint be a sequence of ones and twos where a one in position i tells us that vertex $v_i \in V_1$ and a two in position i tells us that $v_i \in V_2$.
   ii. The hint is the same length as the number of vertices in the graph, $|V|$.
   iii. First, the verifier will check that it is given a well-formed representation of the graph $G = (V, E)$. Next, it will check that the hint is a well-formed representation of a possible solution (check that $length(hint) == |V|$ and that all digits in the hint are one or two). Lastly, the verifier checks that the given hint is a solution to the MaxCut problem. It does this by looping over the set of edges (x, y), checking the part of vertex x, checking the part of vertex y, and checking if they are equal. Throughout this process, the verifier keeps track of the number of times the edge goes between the two parts V_1, V_2. Return YES iff the total count of these spanning edges is at least k. A hint will be deemed to fail to prove this given instance as a YES instance if it is not well-formed or if the count of these edges going between V_1, V_2 is less than k. A NO instance cannot fool the verifier as no well-formed input can generate a partition with less than k edges going between V_1, V_2 and convince the verifier otherwise as it checks each edge exactly once and no vertices can be in both parts.
   iv. Reading the hint and translating it to some data structure is at most an $O(|V|)$ operation. Looping over the edges requires at most $O(|E|) \leq O(|V|^2)$ operations as there are at most $O(|V|^2)$ edges in G. Checking the membership of a given vertex to either V_1 or V_2 is at most $O(|V|)$. Since the verifier checks the membership of both vertices for each edge, this yields a nested loop structure with $O(|V|^3)$ operations at the worst. More specifically, it requires $O(|E| * |V|)$ but the latter bound is sufficient as it is still polynomial (assuming you count vertices as your input size).

   (b) **Algorithm**
   There are $S(|V|, 2) = 2^{|V|-1} - 1$ partitions of the $|V|$ vertices into two parts (not quite the $2^{|V|}$ subsets of $\{1, ..., |V|\}$ as we require that the parts be nonempty and there is some symmetry between the parts: $V_1 = \{1, 2, 3\}, V_2 = \{4, ..., |V|\}$ gives the same partition as $V_1 = \{4, ..., |V|\}, V_2\{1, 2, 3\}$). For each of these possible partitions, calculate the number of edges that go between the two parts by looping over each edge (u, v) and checking the membership of both u and v.

Correctness
Since we calculate the number of bridge edges on every possible partition, we can certainly answer whether there exists a partition with at least k bridge edges.

Complexity
For each of the $2|V| - 1 - 1$ possible partitions, we have to loop over all of the $|E|$ possible edges. If we use sets to store the parts V_l, V_2, checking the membership of each vertex in the edge is a constant time operation. Thus, this yields a runtime of $O(|E|2^|V|)$. If we have a simple graph such that $|E| \leq |V|^2$ then we can see that $n^2 2^n \leq 2^n 2^n = 2^{2n} = 4^n$ and thus this algorithm is $O(4^|V|)$ giving it an exponential running time. As expected, this algorithm does not run in polynomial time.

4 Problem 4

1. (a) Prove that k-Partition Problem is in NP
   i. Let the hint be the n vector h described in the problem, a vector of length n whose entries are between one and k. Here, the entry $h[i] == j$ tells us that weight i is placed in part j.
   ii. The hint is the same size as the number of weights, n.
   iii. The verifier will first check that it is given a well-formed input (that it is given a sequence of weights $(w_1, ..., w_n)$ who are all positive integers and that k is a positive integer). Then, it will check that the hint is well-formed (that its entries are all integers between one and k inclusive and that it has exactly n entries). Lastly, the verifier will check if this hint is to a YES instance or a NO instance. To do so, create a dictionary mapping each of the numbers/parts $1, ..., k$ to zero. Reading over the hint, add w_i to the value associate to part $h[i]$. After traversing the entire hint, if the sums are all equal return YES, else return NO.
   iv. Verifying input and hint are well-formed are $O(n)$ as both are simply list traversals. Similarly, checking for a YES instance is $O(n)$, assuming we are using a data structure with $O(1)$ accesses for the weights, because we extract each weight exactly once and simply traverse the hint.

   (b) **Algorithm**
   Use a brute force approach to check all possible partitions of the n weights. We can use a recursive backtracking approach to generate these partitions. Start by choosing which of the k parts to put the first weight w_1 in. For each possible choice, recurse into its own individual subproblem. This will generate a tree with branching factor k. Then continue the process choosing which of the k parts to put weight w_2 in, and so on. Once all weights have been placed, check if the given partition is a solution to the kPP. If it is not, recurse back and keep choosing.

   Correctness
   Since we explore all possible partitions, we will certainly find a valid partition if it exists.

   Complexity
   We can see that if there are n weights, our state tree will have n levels with branching factor k. This will generate k^n possible states, although some of these will be invalid partitions as they will not have enough parts. Given a specific partition however, we already showed that we can verify if it is a solution in polynomial time. Thus, this algorithm is $O(nk^n) \leq O(k^{2n})$. Thus, this algorithm runs in exponential time.

(c) To show that $2PP \leq_p KNAP$ we just have to find a polynomial time algorithm that converts $2PP$ to $KNAP$. Consider the following conversion:

Let the set of weights $\{w_1, ..., w_n\}$ be for the $2PP$ problem. Define a $KNAP$ problem with the same set of weights and a capacity of $C = \frac{1}{2}\sum_{i=1}^{n} w_i$. In this way, we can convert between from $2PP$ to $KNAP$ in $O(n)$ time as all we have to do is calculate this capacity C by summing up the n weights. If the $2PP$ problem is a YES instance, then there is a partition into two parts such that the sum of the weights in each part $\frac{1}{2}\sum_{i=1}^{n} w_i$. Then, selecting either of these two parts forms a subset with sum $C = \frac{1}{2}\sum_{i=1}^{n} w_i$ which satisfies the condition of the associated $KNAP$ problem. Similarly, if we have a solution to the given $KNAP$ problem, we have a subset of the weights that sum to $C = \frac{1}{2}\sum_{i=1}^{n} w_i$. This implies all the weights NOT chosen to be part of this subset also sum to $\frac{1}{2}\sum_{i=1}^{n} w_i$. Thus, this forms a partition of the weights into two parts, chosen by $KNAP$ and not chosen by $KNAP$, that have the same sum and thus that partition satisfies the associated $2PP$ problem. Thus, $2PP \leq KNAP$.

(d) No it would not imply that $2PP$ is not solvable in polynomial time: we would have to show the reverse of 4*c* to prove that. With what we showed in 4*c*, the runtime of $KNAP$ just places an upperbound on the runtime for $2PP$ (up to a polynomial order) but does not say anything about a lower bound. In actuality, it might have just been a bad idea to try to convert the problem to a $KNAP$ problem.

(e) **Extra Credit:** A given $KNAP$ problem is defined by a set of positive weights $\{w_1, w_2, ..., w_n\}$ and a capacity C. Suppose $C < \sum w_i$ otherwise the $KNAP$ problem is trivial to solve. Generate a kPP problem, $k \geq 2$, with the following weights:

$$\{w_1, w_2, ..., w_n, \}$$

where w_i are the corresponding weights from the $KNAP$ problem and:

$$s_1 = C$$
$$s_2 = \sum w_i - C$$
$$s_j = \sum w_i \quad \forall j : 2 < j \leq k$$

Thus, if we are given a solution to the $KNAP$ problem $S \subseteq W = \{w_1, ..., w_n\}$ such that $\sum_{w_i \in S} w_i = C$, then we can find a solution to the corresponding kPP problem as:

$$\{\{S, s_2\}, \{W \setminus S, s_1\}, \{s_3\}, ..., \{s_k\}\}$$

This is a valid solution to the associated kPP problem as there are k disjoint parts where the sum of each part is the same: $\sum w_i$. Similarly, given a solution to this associated kPP, we could find a solution to $KNAP$ simply by taking S as we know:

$$\sum(S + s_2) = \sum w_i$$

$$\sum(S) + \sum w_i - C = \sum w_i$$

$$\sum(S) = C$$

As $k \leq n$, we can generate the associated kPP problem from the $KNAP$ problem in $O(n)$ time: simply have to traverse the weights of the $KNAP$ problem and calculate their total sum. Thus, $KNAP \leq_p kPP$ for any $k \geq 2$.

5 Problem 5

Algorithm

1. Create a dictionary *edge_counts* that maps vertices to integers and keeps track of the number of unused edges each vertex has. Additionally, create a **stack** of vertices *current_path* to store the vertices that have currently been traversed. Finally, create a list *euler_circuit* to store the final Euler Tour.
2. Choose a vertex v_0 to be the start of the algorithm arbitrarily and add it to *current_path*.
3. Choose one of the edges out of v_0, $e_1 = (v_0, v_1)$ arbitrarily. Reduce the edge counts of both v_0 and v_1 by 1 and add v_1 to *current_path*. Remove this edge from the given graph.
4. continue this process of choosing edges until you reach v_0 again. There are two possible scenarios:
   (a) All edges have been chosen and none remain in the edge list. In which case, the current sequence of edges is a Eulerian Tour and we are done.
   (b) There are unchosen edges remaining.
5. Loop through the vertices in *current_path*. As *current_path* is a stack, this will loop through the current cycle we found in reverse. For each vertex v popped from the stack, if $edge_[v] == 0$, add v to the start of *euler_circuit*. Else, v still has edges remaining so our Euler Tour is not complete. Repeat the above algorithm using v as the start vertex but using the same *edge_counts*, *current_path*, *euler_circuit* data structures.

Correctness
Firstly, we will show that each iteration of this algorithm ends at the same vertex it begins at (justifying that step 6 above in the algorithm always occurs rather than us getting "stuck" at another vertex with no more edges to choose from). Note that at any point in time, we will use exactly two edges moving through a vertex: one to enter the vertex and one to exit. Thus, whenever we travel through a vertex with even number of edges remaining, we also leave it with even number of edges remaining. However, the only way to get "stuck" in a vertex is if we take the final edge going into that vertex, implying that the vertex had an odd number of edges remaining. The only vertex for which this is the case is the start vertex as we never had to take an incoming edge to reach it. Thus, at any given iteration, we can only finish the iteration at the start vertex. While this process does not guarantee that we get the full Eulerian Tour, we can easily join cycles until we have a full tour. After a given cycle is complete, all vertices once more have an even number of edges remaining. Thus, we can

find another vertex v_n with edges remaining and use that as our start vertex. After this second iteration, since we now have two disjoint cycles including v_n, $\{v_0, v_1, ..., v_n, ..., v_0\}$ and $\{v_n, ..., v_i, ...v_n\}$, we can simply merge them to form a single larger cycle $\{v_0, v_1, ..., v_n, ..., v_i, ..., v_n, ...v_0\}$.

Since we add vertices to *current_path* in an order corresponding to a given sequence of edges, we are certain v_j follows v_i in *current_path* iff there is an edge between the two. Since we remove edges from the graph after traversing them, we will not repeat any edges. Thus, a Euler Circuit is generated.

Complexity

If we use an adjacency list where the same edge has a pointer between its location in each of the vertices that are its endpoints, we can quickly delete edges from our graph. Since we have no preference for the edges being chosen, we only traverse each edge exactly once. The amount of work we do with each edge is constant (popping off a stack, getting/updating values in a dictionary) so we perform at most $O(1)$ work for each edge when generating *current_path*, *edge_counts*, *euler_circuit*. Furthermore, the traceback part of the algorithm where we check if each vertex has any edges remaining is at most $O(E)$ as once more each pair of adjacent vertices in *current_path* is encoding an edge and we will only traverse each at most once. Overall, this yields a runtime of $O(|E|)$ which is also $O(|V| + |E|)$.