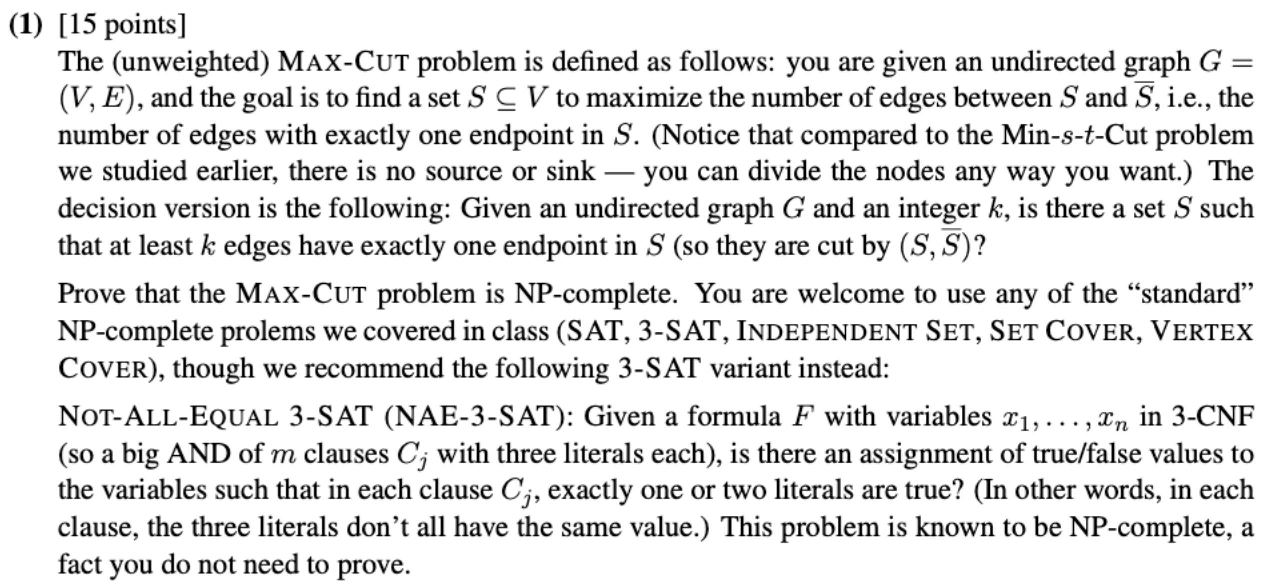
Assignment 10

CSCI 270

Paul Kim | pbkim@usc.edu | 1723717002

Question 1



(1) [15 points]
The (unweighted) MAX-CUT problem is defined as follows: you are given an undirected graph $G = (V, E)$, and the goal is to find a set $S \subseteq V$ to maximize the number of edges between S and $\overline{S}$, i.e., the number of edges with exactly one endpoint in S. (Notice that compared to the Min-s-t-Cut problem we studied earlier, there is no source or sink — you can divide the nodes any way you want.) The decision version is the following: Given an undirected graph G and an integer k, is there a set S such that at least k edges have exactly one endpoint in S (so they are cut by $(S, \overline{S})$?

Prove that the MAX-CUT problem is NP-complete. You are welcome to use any of the "standard" NP-complete prolems we covered in class (SAT, 3-SAT, INDEPENDENT SET, SET COVER, VERTEX COVER), though we recommend the following 3-SAT variant instead:

NOT-ALL-EQUAL 3-SAT (NAE-3-SAT): Given a formula F with variables $x_1, \ldots, x_n$ in 3-CNF (so a big AND of m clauses C_j with three literals each), is there an assignment of true/false values to the variables such that in each clause C_j, exactly one or two literals are true? (In other words, in each clause, the three literals don't all have the same value.) This problem is known to be NP-complete, a fact you do not need to prove.

Answer:

Unweighted Max-Cut Problem: Given an undirected graph G = (V, E), find a set S in V, to maximize the number of edges between S and S' (A Cut).

Decision Problem: Given an undirected graph G and an integer K, is there a set S such that at least K edges have exactly one endpoint in S.

Problem: Prove that the Max Cut Problem is NP-Complete.

Prove that the Decision Problem is in NP:

Claim: Decision Problem is in NP:

Proof: Certificate is an undirected graph G and an integer K. The Certifier verifies/checks that there is a cut of size K.

Runtime: This proposed algorithm runs in polynomial time because we can just use loops that checks the sets that the size of a given cut contains K such edges (It is efficient).

Proved that Decision Problem is in NP.

Prove that the Unweighted Max-Cut Problem is NP Hard, by showing that Not-All-Equal 3-SAT <= (p) Unweighted Max-Cut

*Devise some reduction to show that it is NP hard (by showing a reduction from a known NP-Hard Problem, to NAE 3-SAT).

Not-All-Equal 3-SAT. Given a formula F with variables X1,...,Xn in 3 CNF, is there an assignment of T/F to the variables such that in each clause C_j, exactly one or two literals are true, and the other is false. This problem is known to be NP-Complete.

Reduction:

*Since we know that NAE 3-SAT is NP-Hard, Max Cut must be NP-Hard too (Prove).

We want to encode variables and clauses in the graph. The standard way to think about a variation of NAE 3-SAT is to assign T/F values to each variable so that in each clause, there is an assignment of T/F to the variables such that exactly one or two literals are true.

Rough Idea: Intuitively, we model the clause and its 3 literals as 3 nodes, and the clause as a cluster of these three nodes. When making an S and S' cut, the maximum way to get the edge values is to evidently put one node on the S side of the cut, and two nodes on the S' side of the cut. To relate this intuition to the NAE 3-SAT problem, we can assign the node in the S side of the cut with a value of true, and the nodes in the S' side of the cut with values of false.

For every clause in NAE 3-SAT, produce a "triangle" (X1, X2, X3) in the graph. If the two literals in the clause are the same, give the triangle a double edge. For each literal create a node for itself and its negation X_i and X_i, and create an edge between these two, for each time X_i or its negation appear (this makes sure that the literal and negation are not in the same set).

*It is important that for every literal you create a negation, and draw an edge between the literal and negation to make sure that the literal and negation are not in the same set.

For every clause and the triangle between the vertices, we assign a capacity 1 to these edges.

To find the solution to NAE 3-SAT, we assign all vertices of the S side of the cut to true, and the others as false. Each triangle we constructed can contribute at most 2 edges to a cut. Assume we tried to find a cut from our formulation. We know that for this cut all X_i are separated from its negation by the cut. Therefore, all triangles can now be split by the cut.

Here we constructed a Graph.

Proof:

Total Total Cost of edges (Max) will be 2*M from the triangles + N from the negation edges.

Proof =>:

Supposing that NAE 3-SAT is a satisfiable assignment. The assignment correctly corresponds to a cut in G. One side of the cut consists of all vertices labeled true. The other side of the cut consists of all vertices that are assigned false in the assignment. Since exactly one of the terms Xi and its negation evaluate to true and the other false in the assignment, it means that these edges must go across to the cut, contributing to the capacity of the cut. Since the assignment satisfies NAE 3-SAT, exactly two edges in every clause must go across the cut, contributing M*2 to the capacity of the cut. The total capacity of the cut is equal to M*2 + n (where n is the number of literals edges).

<= Proof:

Suppose that G contains a cut with capacity M*2 + n as we suggested above. It must be true that two of the edges go across this cut per triangle. If the cut were to miss at least one of the edges that go across the cut, then the capacity would yield something smaller than M*2 + n. We also know that no cut can separate three edges of a triangle because we assigned the node in the S cut as true and the node on the S' cut as false, so the capacity could never be bigger than M*2 + n (cannot ever cut three edges). Since all edges go across the cut, the cut successfully corresponds to the assignment for NAE 3-SAT, while also satisfying the NAE 3-SAT requirement.

Runtime: We can create the transformation from clauses -> graphs in poly time just using loops to create such a graph with T and F assignments. We can also go from a cut to a boolean assignment simply in poly time as well. (It is efficient)

Proof Completed.

Question 2

(2) [10 points]
We call a program "boring" if it outputs the same thing for every input. Formally, we define BORING := $\{P \mid\mid P(x) = P(1)$ for all inputs $x\}$. Give a reduction showing that HALT $\leq_m$ BORING and prove the reduction correct.

[Note: this problem will likely only make sense and be solvable to you after the first class of the final week of classes.]

Answer:

HALT: Given a program P, does P halt when run on its own source code as input?

Theorem: HALT is undecidable. (Proved in class)

HALT = (P || P(P)) Terminates.) => Checks if the program ever stops.

BORING: A program P is boring if it outputs the same thing for every input.

BORING = (P || P(x) = P(1) for all inputs X).

Problem: Give a reduction showing that HALT <= (m) BORING and prove the reduction is correct.

Proof: Show that HALT <= (m) BORING. *We reduce from an undecidable problem to our new problem, to show that BORING is undecidable.

Define a function F:

Input: Input to HALT, so a program P.

Output: Input to Boring, so a program Q.

Algorithm:

```
Int Q (int X)
    Run P(P); // program runs infinitely
    Return 1; // program terminates will same output if else.
```

Properties:

If P is in HALT, then Q is in BORING.

- If P terminates when run on input P, then Q always returns 1.

IF P is not in HALT, then Q is not in BORING.

- If P does not terminate when run on input P, then Q will run forever as P does not terminate, showing that there exists at least one input X on which Q(X) does not terminate nor output the same thing for every input.

Proof:

If P terminates on any input, then Q(X) = 1 for all input X.

- If P(P) Terminates, then for all input X, Q(X) terminates and always outputs 1, so Q is in BORING.

IF P does not terminate for an input, then Q(X) does not terminate (because it gets stuck in the call to P(P).

- If P(P) runs forever, then Q will get stuck, so it does not terminate for any X, so Q is not in BORING.

The algorithm is very simple and runs in Polynomial Time.

Proof Completed. We reduced from halt to show that the BORING problem is undecidable.

Question 3

(3) [8 points]
In light of the fact that the Halting Problem is not decidable, we might wonder what would happen if we had a more powerful programming language. Specifically, imagine the language C+++, which is just like C++, except it has one more very powerful function:

```
bool doesItHalt (string c++-program, string input);
// returns whether the given C++ program will halt on the given input.
// can only be called for C++ programs, but not for C+++ programs.
```

This function `doesItHalt` allows you to figure out whether a C++ program will terminate on a given input — note, though, that it cannot be called on C+++ programs.

One would not consider this a reasonable model of computation because it is very powerful, in that it can solve a non-comptable problem. But suppose that we accepted this as our model of computing. Could we now compute all functions?

Answer this question by proving one of the following: (1) Every function (whose input is a string/number and output is a string/number) can be computed by a C+++ program, or (2) There exists a function which cannot be computed by any C+++ program.

[Note: this problem will likely only make sense and be solvable to you after the first class of the final week of classes.]

Answer:

C+++ = same language as C++ that also has a very powerful function:

//

bool doesItHalt (string C++ program, string input);

// returns whether the given C++ program will halt on the given input.

*// *Note can only be called for C++ programs, but not for C+++ programs.*

//

doesItHalt allows you to figure out whether a C++ program terminates on a given input, though, it cannot be called on C+++ programs.

*Using this, could we now compute all functions?

- Short Answer: No, it allows you to solve the problem in C++, but you still cannot run HALT on C+++.

Problem: Prove that there exists a function which cannot be computed by any C+++ program.

Intuition: You can now solve the HALTING problem in C++ but you cannot run HALT on C+++. C+++ does not know if C++ code halts or not. HALT only works for C++ code.

Prove: HALT is undecidable in C+++.

Proof:

Assume that HALT2 is decidable in C+++, that includes C+++ programs, whereas HALT itself doesn't include C+++. That is, there exists a program that outputs true if P(P) terminates, and outputs false if not.

We can create a new function that takes a C+++ program and input.

```
bool doesItHalt2 (string C+++ program, string input):
    // If C+++ terminates, return TRUE
    // If C+++ does not terminate, return false
```

We can then write another new Destroyer program that gets an input program P and does the following using the function we wrote above:

```
bool Destroyer(Program P):
    bool flag = doesItHalt2(P); // Output 1 means P(P) terminates.
    if (flag == true):
        -> enter an infinite loop;
    else:
        -> RETURN false
```

What happens if we call Destroyer on input Destroyer. (Set P = Destroyer).

1.) Destroyer(Destroyer) Terminates.

  - Then this means that doesitHalt2(Destroyer) returns TRUE, so flag = true. However by our Destroyers design, the Destroyer enters an infinite loop, so it doesn't terminate which is a contradiction.

2.) Destroyer(Destroyer) does not Terminate.

  - Then, doesitHalt2(Destroyer) returns false, so flag = false. So Destroyer will terminate and return 0 which is a contradiction.

*We always reach a contradiction, so we cannot conclude if HALT2 is not decidable or not.

So the only assumption that C+++ is computable for all function must have been wrong since we show that HALT is not decidable in C+++. So there exist a function for which cannot be computed by a C+++ program.