**EXPERIMENT-10**

**Title: Branch and Bound**

1. **The Hamiltonian cycle problem is NP-complete.**

A language is in NP if a proposed solution can be verified in polynomial time. In this case a proposed solution is simply a listing of the vertices in the order they appear in the claimed cycle. This list is the so-called certificate.

To check if this list is actually a solution to the Hamiltonian cycle problem, one counts the vertices to make sure they are all there, then checks that each is connected to the next by an edge, and that the last is connected to the first.

This takes time proportional to nn, because there are nn vertices to count and nn edges to check. nn is a polynomial, so the check runs in polynomial time.

Note that this only shows that Hamiltonian Cycle is in NP, not that it is NP-complete.

1. **The sum of subset problem using NP-complete**

If you express the inputs in **unary** you get a different running time than if you express them in a higher base (binary, most commonly).

So the question is, for subset sum, what base is appropriate? In computer science we normally default to the following:

* If the input is a **list** or **collection**, we express its size as the number of items
* If the input is an **integer**, we express its size as the number of bits (binary digits)

The intuition here is that we want to take the more "compact" representation.

So for subset sum, we have a list of size nn and a target integer of value tt. Therefore it's common to express the input size as nn and t=2kt=2k where kk is the number of bits needed to express tt. So the running time is O(n2k)O(n2k) which is exponential in kk.

But one could also say that tt is given in unary. Now the size of tt is tt, and the running time is O(nt)O(nt), which is polynomial in nn and tt.

In reductions involving subset sum (and other related problems like partition, 3-partition, etc) we **must** use a non-unary representation if we want to use it as an NP-Hard problem to reduce from.

An approximate version of the subset sum would be: given a set of *N* numbers *x*1, *x*2, ..., *xN* and a number *s*, output

* yes, if there is a subset that sums up to *s*;
* no, if there is no subset summing up to a number between (1 − *c*)*s* and *s* for some small *c* > 0;
* any answer, if there is a subset summing up to a number between (1 − *c*)*s* and *s* but no subset summing up to *s*.

If all numbers are non-negative, the approximate subset sum is solvable in time polynomial in *N* and 1/*c*.

The solution for subset sum also provides the solution for the original subset sum problem in the case where the numbers are small (again, for nonnegative numbers). If any sum of the numbers can be specified with at most *P* bits, then solving the problem approximately with *c* = 2−*P* is equivalent to solving it exactly. Then, the polynomial time algorithm for approximate subset sum becomes an exact algorithm with running time polynomial in *N* and 2*P* (i.e., exponential in *P*).

The algorithm for the approximate subset sum problem is as follows:

initialize a list *S* to contain one element 0.

for each *i* from 1 to *N* do

let *T* be a list consisting of *xi* + *y*, for all *y* in *S*

let *U* be the union of *T* and *S*

sort *U*

make *S* empty

let *y* be the smallest element of *U*

add *y* to *S*

for each element *z* of *U* in increasing order do

//trim the list by eliminating numbers close to one another

//and throw out elements greater than *s*

if *y* + *cs*/*N* < *z* ≤ *s*, set *y* = *z* and add *z* to *S*

if *S* contains a number between (1 − *c*)*s* and *s*, output *yes*, otherwise *no*

The algorithm is polynomial time because the lists *S*, *T* and *U* always remain of size polynomial in *N* and 1/*c* and, as long as they are of polynomial size, all operations on them can be done in polynomial time. The size of lists is kept polynomial by the trimming step, in which we only include a number *z* into *S* if it is greater than the previous one by *cs*/*N* and not greater than *s*.

This step ensures that each element in *S* is smaller than the next one by at least *cs*/*N* and do not contain elements greater than *s*. Any list with that property consists of no more than *N*/*c* elements.

The algorithm is correct because each step introduces an additive error of at most *cs*/*N* and *N* steps together introduce the error of at most *cs*.