Solving Computationally Hard problems (optimisation problems, NP hard problems)

Submitted By:
Prakhar Jain
10400EN002 (IDD - 18)

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and I would like to extend my sincere thanks to all of them.

I am highly indebted to Prof. A. K. Agrawal for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.
I would like to express my gratitude towards my parents & friends for their kind co-operation and encouragement which help me in completion of this project.

I would like to express my special gratitude and thanks to department of Computer Science IIT BHU persons for giving me such attention and time.

Motivation

I have done semester long project of Automata under the guidance of Prof A. K. Agrawal last semester which encouraged me in going through the limits of computability. In that project, I learned about principles of Turing Machine and Complexity Theory apart from the basic classes of Languages and their application like Regular Languages, Context Free Grammars. Now after learning about the limitations of computability, I was motivated to give my times on solving problems which are hard and learning about algorithms that exist in this field.

I have a experience of programming competitions where we are given algorithmic problems to solve in time limit. So, I had a plan to take as many of such hard problems possible and give solutions using the knowledge learned from the algorithms used in solving these.

Thus, I ended up having this project of solving many hard problems and learning algorithms related to it.

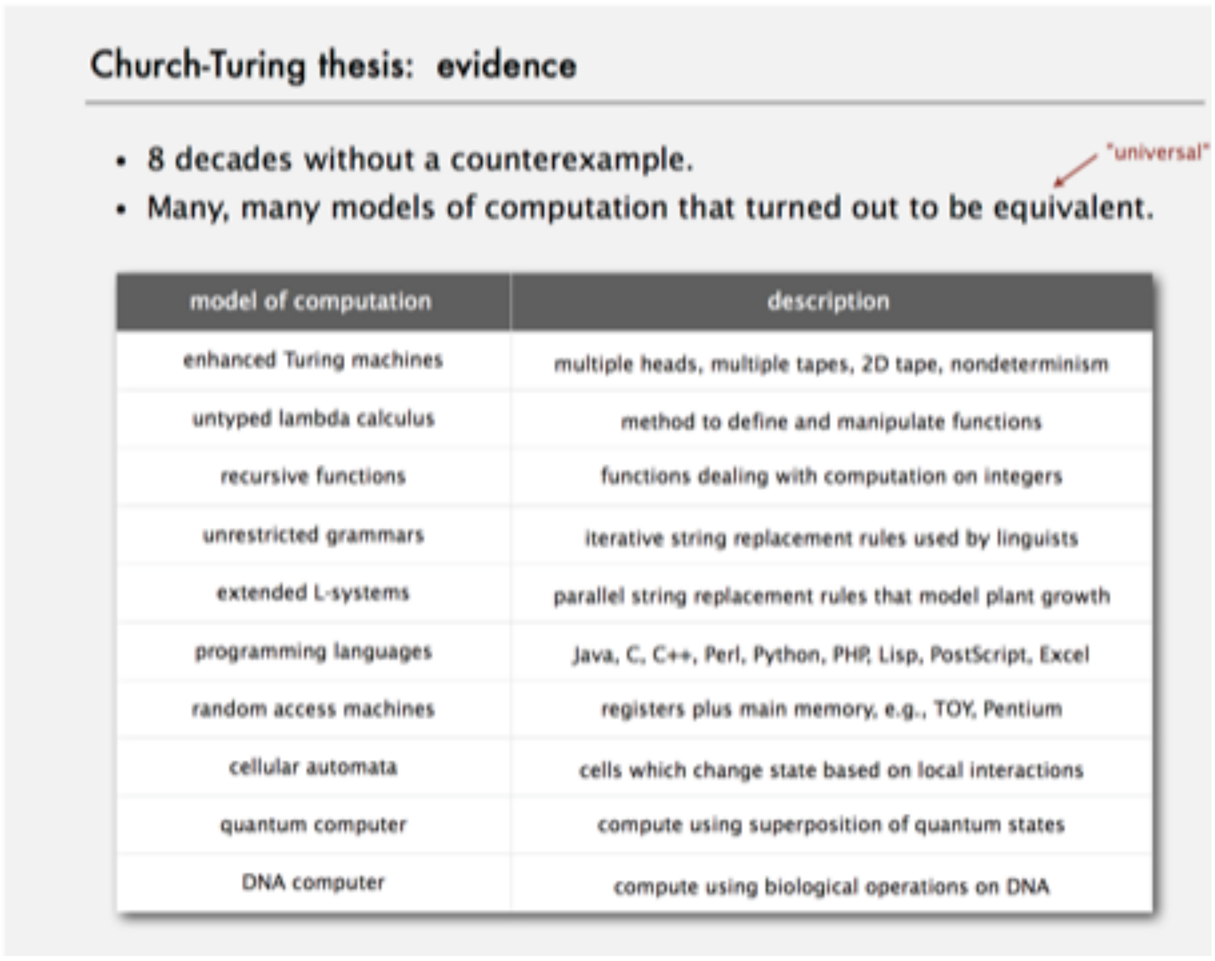
Index

Turing Machine

Turing Machine can compute any function that can be computed by a physically harness able process of the natural world

To prove that a model has more or equal power than any other model, we use simulation of one model over the other

Turing Machine is simple and universal model of computation
Many more models have been studied and are found to be equivalent



Church-Turing thesis: evidence

- 8 decades without a counterexample.
- Many, many models of computation that turned out to be equivalent.

model of computation	description
enhanced Turing machines	multiple heads, multiple tapes, 2D tape, nondeterminism
untyped lambda calculus	method to define and manipulate functions
recursive functions	functions dealing with computation on integers
unrestricted grammars	iterative string replacement rules used by linguists
extended L-systems	parallel string replacement rules that model plant growth
programming languages	Java, C, C++, Perl, Python, PHP, Lisp, PostScript, Excel
random access machines	registers plus main memory, e.g., TOY, Pentium
cellular automata	cells which change state based on local interactions
quantum computer	compute using superposition of quantum states
DNA computer	compute using biological operations on DNA

Turing Machine’s   acceptance is defined as the set of finite configurations C1,C2,…,CK.
A TM which accepts or rejects on all input is called deciders.
Multi tape TM and non deterministic TM has same power [proof Automata VL]

Turing recognizable(Recursively Enumerable) and Turing decidable(Recursive) language
Recursively Enumerable languages are not closed under complementation and difference. Recursive Languages are not closed under homomorphism
Universal Turing Machine which takes Turing Machine M and its input w and accepts only if M accepts w; UTM is recursively enumerable and not recursive language.
There are more languages than integers. But there are no more TMs than integers

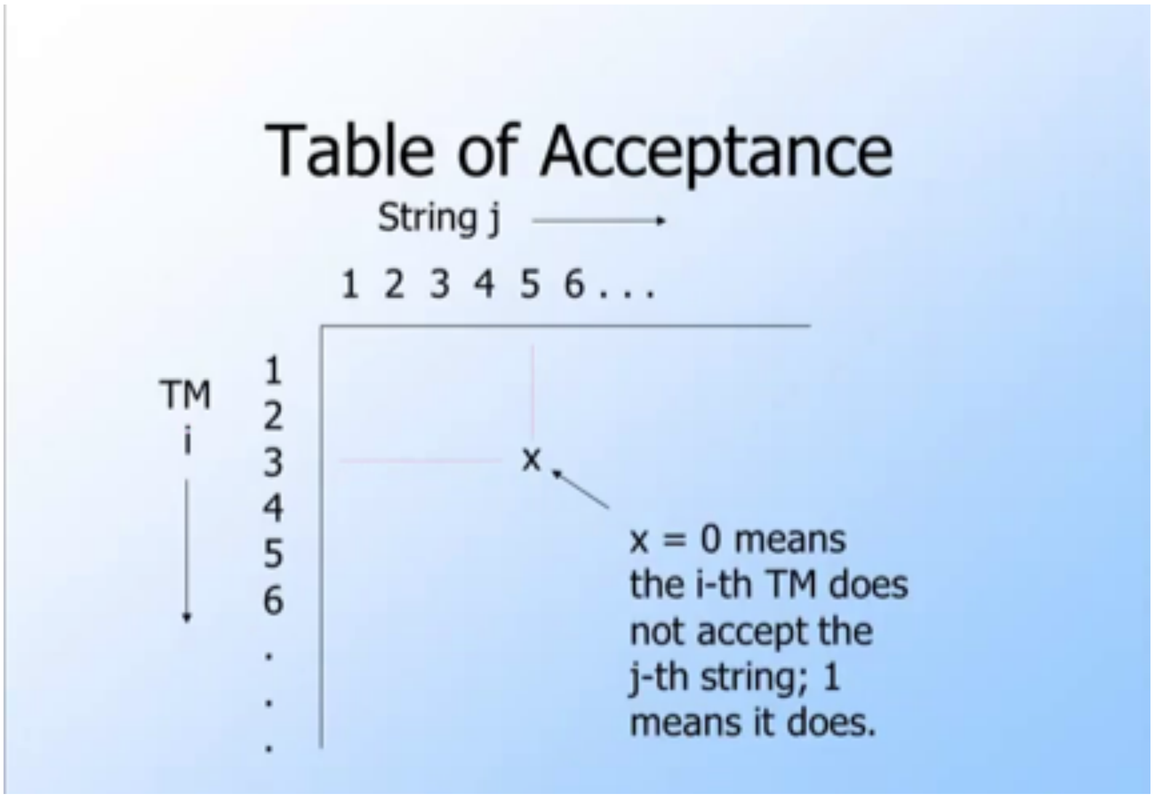
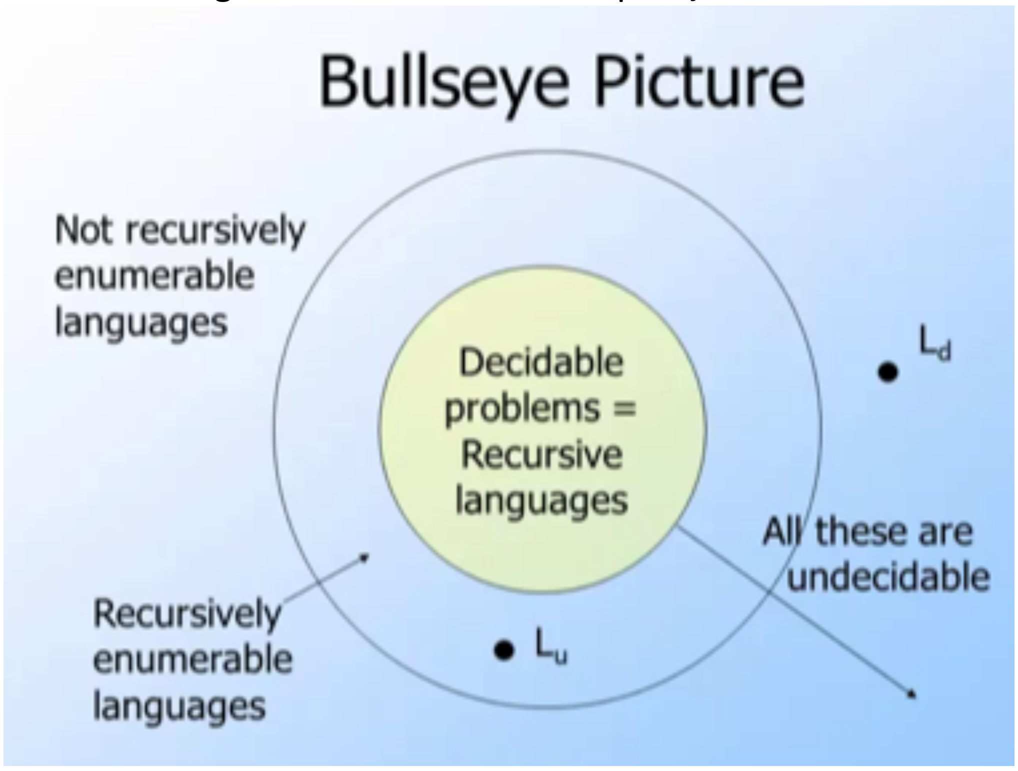


Table of Acceptance

String j
1 2 3 4 5 6 . . .
TM
i
1
2
3
4
5
6
. . .
x
x = 0 means the i-th TM does not accept the j-th string; 1 means it does.

Consider the diagonalization language Ld = {w | w is the ith string, and the i-th Turing Machine does not accept w}


Bullseye Picture
Not recursively enumerable languages
Decidable problems = Recursive languages
L_d
All these are undecidable
Recursively enumerable languages
L_u

Rice's Theorem talks about the properties of languages. This also applies to programs. That tells us that any nontrivial question about what a program does will also be undecidable. I want to emphasize "about what a program does." There are lots of questions about what a program *looks like* that are decidable, for example, I can tell whether a program uses more than 20 variable names. But that's not a question about what a program does.

NP and P

The class P of problems or languages that can be solved by a Turing machine [Deterministic Time Bounded] that runs in polynomial time, as a function of its input size.   We also meet the class NP, which is problems that can be solved by a Turing machine that is nondeterministic, but has a polynomial time bound along each branch.

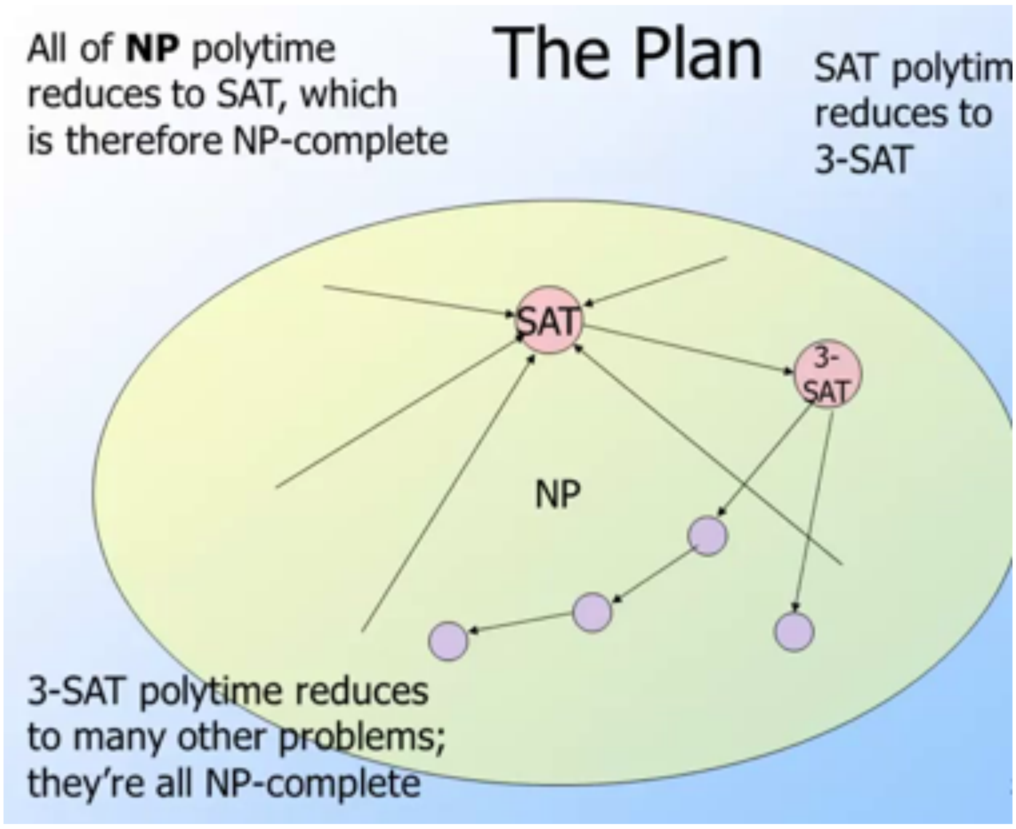
NP is defined in terms of nondeterministic Turing machines. The running time of a nondeterministic Turing machine is the maximum number of moves it takes along any branch, that is making any sequence of choices. If there is a polynomial bound on that time, then the nondeterministic machine is said to be polynomial time-bounded.

NP complete problems are in NP and if it is in P then every problem in NP is in P
There is a polytime time reduction from all language in NP to this language

NP complete problems are both NP hard and NP
Cook Theorem states that SAT problem is NP Complete
NP Hard problems if are in P then P = NP.

Confusion between NP complete and NP Hard:
A set or property of computational search problems. A problem is NP - hard if solving it in polynomial time would make it possible to solve all problems in class NP in polynomial time.
Some NP-hard problems are also in NP (these are called " NP-complete"), some are not.

Knapsack Problem is NP-Complete


The Plan
All of NP polytime reduces to SAT, which is therefore NP-complete
SAT polytim reduces to 3-SAT
SAT
3-SAT
NP
3-SAT polytime reduces to many other problems; they're all NP-complete

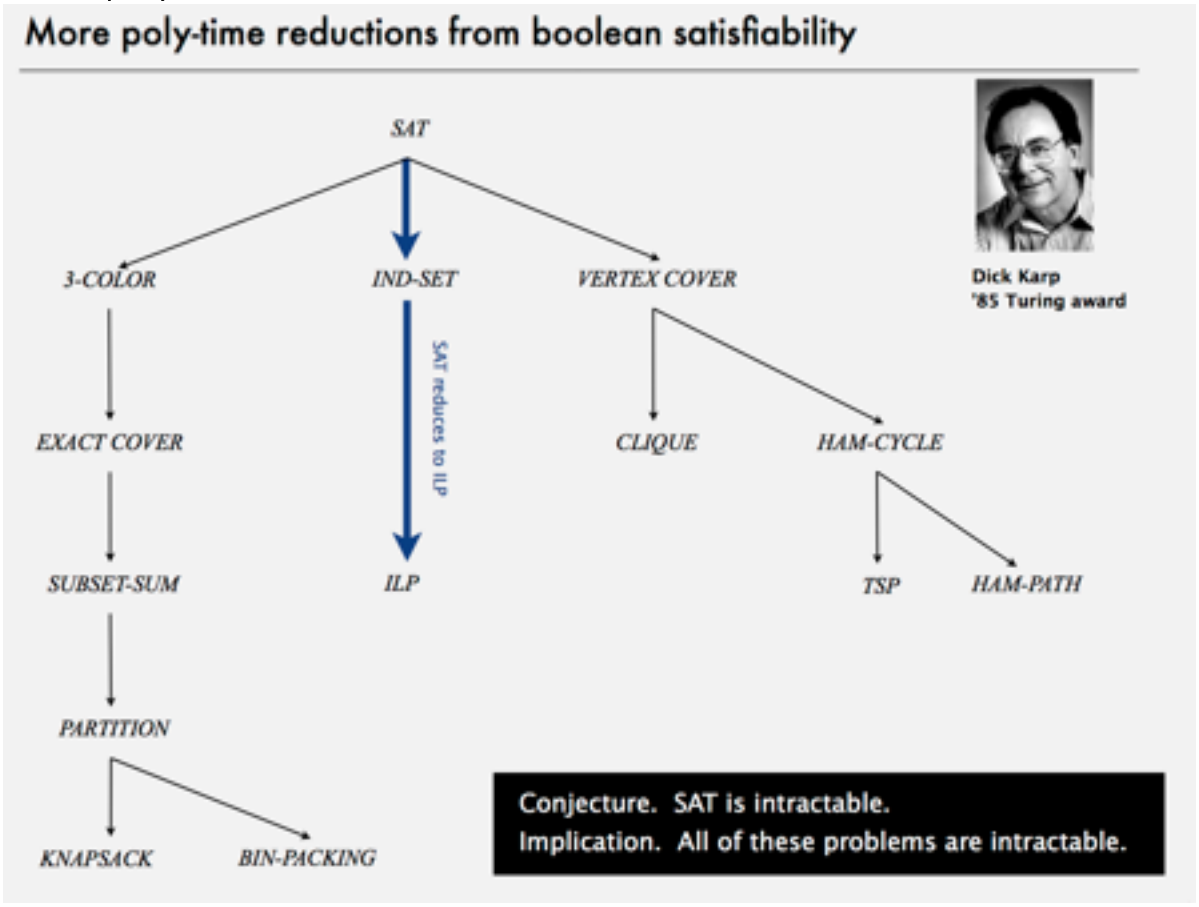
Search problems are those problems in which we are able to find solution or report none, and we can check that the solution is correct in polynomial time

NP is class of all search problem
P is class of all search problems solvable in polynomial time

NP problems can be solved by non deterministic Turing machine in polynomial time.
Non Deterministic Turing Machine:
It always guesses correctly. It was always know which state to go.

Eg:
Imagine for a set of linear inequalities, we need a 0-1 solution (Inter Linear Programming ILP)
We have n variables
With the help of ND machine, we can create an array of size n which on initialization contains solution [guesses the correct solution of all variables]
In contrast to what we do in initialization of array which initializes to all

zeroes
If we can reduce a problem P1 to another problem P2 in polytime, then we conclude that P2 is as hard as P1
Cook Reduction/ Polytime Reduction:
	Polynomial number of standard computation steps
	Polynomial number of calls to P2
If SAT polytime reduces to Y, then Y is intractable



Conjecture. SAT is intractable.
Implication. All of these problems are intractable.

When you find NP problem:

- Try to reduce it tractable parts
- Heuristics
- Solve it in exponential time but in faster way

Knapsack Problem

The problem is mathematically formulated in the following way. Given n items to choose from, each item $i \in 0...n-1$ has a value v_i and a weight w_i. The knapsack has a limited capacity K. Let x_i be a variable that is 1 if you choose to take item i and 0 if you leave item i behind. Then the knapsack problem is formalized as the following optimization problem, maximize:

$$v_i\, x_i \quad {\scriptstyle i \in 0...n-1}$$

subject to:

$$w_i x_i \leq K \quad {\scriptstyle i \in 0...n-1}$$

$$x_i \in \{0,1\} \ (i \in 0...n-1)$$

Input Example:

```
4 11
8 4
10 5
15 8
4 3
```

Output Example:

```
19 0
0 0 1 1
```

Algorithms applied:

DP

O(k,j) denotes the optimal solution to the knapsack problem with capacity k and items [1..j]

We are interested in finding out the best value O(K,n)

$O(k,j) = \max(O(k,j-1), v_j + O(k-w_j,j-1))$ if $w_j \leq k$!

$O(k,j) = O(k,j-1)$ otherwise!

$O(k,0) = 0$ for all k

Time Complexity: O(Kn)

Greedy

Idea 1:
More items is best, start with small ones and take as many as you can!
Idea 2:
Valuable items are best, start with the most valuable items!
Idea 3:
Value density! dollars per kilogram

Branch and Bound

Iterative two steps
– branching
– bounding
Branching!
– split the problem into a number of subproblems
like in exhaustive search!
Bounding!
– find an optimistic estimate of the best solution to the subproblem!
maximization: upper bound!
minimization: lower bound

Search Strategies

Search Strategies!
– depth-first, best-first, least-discrepancy! – many others!
Depth-first!
– prunes when a node estimation is worse than the best found solution!
– memory efficient!
Best-First!
– select the node with the best estimation!
Least-Discrepancy!

– trust a greedy heuristic
Assume that a good heuristic is available!
– it makes very few mistakes!
– the search tree is binary!
– following the heuristic means branching **left**!
– branching **right** means the heuristic was wrong!
Limited Discrepancy Search (LDS)!
– explore the search space in increasing order of mistakes !
– trusting the heuristic less and less

Limited Discrepancy Search (LDS)!
– explore the search space in increasing order of mistakes !
– trusting the heuristic less and less!
Explores the search space in waves!
– no mistake!
– one mistake! – two mistakes!
– ....

Branch and Bound for Mixed Integer Programming

Break the problems into n constraints each with variables having 0 or 1 values
Solve the linear relaxation
– If the linear relaxation is worse than the best solution found so far, prune this node
the associated problem is suboptimal
– If the linear relaxation is integral, we have found a feasible solution
update the best feasible solution if appropriate
– Otherwise, find an integer variable x that has a fractional value f in the linear relaxation
create two subproblems x <= floor(f) and x >= ceil(f)

Travelling Salesman Problem

The problem is mathematically formulated in the following way: Given a list of locations N = 0...n − 1 and coordinates for each location $\langle x_i, y_i \rangle$ $i \in N$. Let v_i $i \in N$ be a variable denoting the visitation order (i.e. the value of v_i is the i-th location to be visited) and let $dist(l_1, l_2)$ be the Euclidean distance between two locations. Then the traveling salesman problem is formalized as the following optimization problem,

minimize:

$$\text{Summation}_{\ i \in 0 \ldots n-1}$$

$$dist(v_i, v_{i+1}) + dist(v_n, v_0)$$

subject to:

$$v_i \text{ are a permutation of } N$$

MODEL:
Problem Formulation:

Formulate the travelling salesman problem for integer linear programming as follows:

Generate all possible trips, meaning all distinct pairs of stops.
Calculate the distance for each trip.
The cost function to minimize is the sum of the trip distances for each trip in the tour.
The decision variables are binary, and associated with each trip, where each 1 represents a trip that exists on the tour, and each 0 represents a trip that is not on the tour.
To ensure that the tour includes every stop, include the linear constraint that each stop is on exactly two trips. This means one arrival and one departure from the stop.

Calculate Distance between Points

Because there are 200 stops, there are 19,900 trips, meaning 19,900 binary variables (# variables = 200 choose 2).

Generate all the trips, meaning all pairs of stops.

Calculate all the trip distances, assuming that the earth is flat in order to use the Pythagorean rule.

With this definition of the dist vector, the length of a tour is

dist'*x

where x is the binary solution vector. This is the distance of a tour that you try to minimize

Equality Constraints

The problem has two types of equality constraints. The first enforces that there must be 200 trips total. The second enforces that each stop must have two trips attached to it (there must be a trip to each stop and a trip departing each stop).

Specify the first type of equality constraint, that you must have nStops trips, in the form Aeq*x = beq.

To specify the second type of equality constraint, that there needs to be two trips attached to each stop, extend the Aeq matrix as sparse.

Binary Bound

All decision variables are binary. Now, set the intcon argument to the number of decision variables, put a lower bound of 0 on each, and an upper bound of 1.

SOURCE CODE

```
def solve_mip(distanceMatrix, nodeCount):

    solver = pywraplp.Solver('CP is fun!',
pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING);

    print "creating variables"
    nodes = range(nodeCount)

    x = [[solver.BoolVar('x%d_%d' % (i,j)) for j in nodes] for i in nodes]

    print "enter/exit just once"
    for i in nodes:
        solver.Add ( x[i][i] == 0 )
        row = x[i]
        solver.Add ( solver.Sum(row) == 1 )
        column = [x[j][i] for j in nodes]
        solver.Add ( solver.Sum(column) == 1 )

    u = [solver.IntVar(0, nodeCount - 1, 'u%d' % i) for i in nodes]
    solver.Add (u[0] == 0)

    objective = solver.Objective()
    for i in nodes:
        for j in nodes:
            if (i != j):
                objective.SetCoefficient(x[i][j], distanceMatrix.get(i,j))
                solver.Add( u[i] - u[j] + nodeCount * x[i][j] <= (nodeCount - 1) )
```

Constraint Programming

Modelling of the problem into decision variables with a set of domains and constraints that must be satisfied in any case. Constraints are independent of each other and thus adding constraints to the problem space is easy. We need to make a choice or branch for selecting a value in the domain of the variable.
Choices may be wrong, then we need to backtrack. Branching means for e.g. selecting every possible values in the domain.
Constraints provide two purpose:
Feasibility Check
Pruning

Constraint Propogation Algorithm

```
propagate()
{
        repeat
                Select a constraint c
                if
                        c is infeasible given the domain store, then return
failure
                else
                        apply the pruning algorithm associated with c
        until no constraint can remove any value from the domain of its
variable
        return success
}
```

Some easy problems covered:

1) Map Coloring
2) N Queens

Linear Constraint over integers
a1x1 + … + anxn >= b1y1 + … + bmym
aibj >= 0 are constants
xj,yi are variables with domains D(xi), D(yj)

->Feasibility Test

a1 max(D(x1)) + … + an max (D(xn)) >= b1 min(D(y1)) + … bm min(D(ym))
L >= R

-> Pruning

aixi >= R-(L-ai max D((xj))
bjyj <= L - (R-bi min(D(yi))

Richness of Constraint Programming

1) Reification
- the ability to transform a constraint into a 0/1 variable
- the variable has the value 1 if the constraint is true and 0 otherwise

2) The basic element constraint
- the ability to index an array / matrix with a variable or an expression containing variables

3) Global Constraints
- make modelling easier and more natural
- give the ability to exploit dedicated algorithms

Eg x1={1,2} x= {1,2} x3 = {1,2}

all_different(xi)

is better algorithm than

```
for (i=1..3)
    for (j=1..3)
        if i != j
        x[i] != x[j]
```

Problems Solved:

1) Stable Marriage
2) Crypto Arithmetic

How to optimise in constraint programming?
- solve a sequence of satisfaction problems
- find a solution
- impose a constraint that the next solution must be better

Source Codes:

Knapsack Problem:

DP

```
dp[0][0] = 0;
        for (int i = 0; i < items.length; ++i) {
            for (int cur = 0; cur  <= K; ++cur) {
                if (dp[i][cur] == Integer.MIN_VALUE) continue;
                dp[i + 1][cur] = Math.max(dp[i + 1][cur], dp[i][cur]);
                int weight = cur + items[i].weight;
                if (weight <= K) {
                    dp[i + 1][weight] = Math.max(dp[i + 1][weight], dp[i][cur] +
items[i].cost);
                }
            }
        }
        for (int t : dp[items.length]) result = Math.max(result, t);
```

Greedy

```
Arrays.sort(this.items, new Comparator<Item>() {
            public int compare(Item a, Item b) {
                double here = (double) a.cost / a.weight;
                double there = (double) b.cost / b.weight;
                if (here > there) return -1;
                if (here < there) return 1;
                return 0;
            }
        });

        for (int i = 0; i < items.length; i++) {
            if (weight + items[i].weight <= K) {
                taken[items[i].ID] = 1;
                value += items[i].cost;
                weight += items[i].weight;
            } else {
                taken[items[i].ID] = 0;
            }
        }
```

Branch and Bound

```
static class Node {
        int value;
        int room;
        int estimate;
        int at;
        BitSet taken;
        static int best = 0;
        static BitSet bestTaken = null;

        Node(int value, int room, int estimate, int at) {
            this.value = value;
            this.room = room;
            this.estimate = estimate;
            this.at = at;
        }

        public void branchAndBound(Item[] items) throws Exception{
            if (this.room < 0) return;
            if (System.currentTimeMillis() - Knapsnack.startTime >= 1000 *
30) throw new Exception();
            if (this.estimate <= this.best && this.at < items.length) return;
            this.best = Math.max(best, this.value);
            if (at == items.length) {
                if (this.value == this.best) {
                    this.bestTaken = (BitSet) this.taken.clone();
                }
            } else {
                Node left = new Node(this.value + items[at].cost, this.room -
items[at].weight, this.estimate, at + 1);
                left.taken = (BitSet) this.taken.clone();
                left.taken.set(at);

                Node right = new Node(this.value, this.room, this.estimate -
items[at].cost, at + 1);
                right.taken = (BitSet) this.taken.clone();

                left.branchAndBound(items);
                right.branchAndBound(items);
            }
        }
```

```
  }
```

Stable Marriage:

```
 rankMen = ranks["rankMen"]
 rankWomen = ranks["rankWomen"]

 n = len(rankMen)

 wife = [solver.IntVar(0, n - 1, "wife[%i]" % i) for i in range(n)]
 husband = [solver.IntVar(0, n - 1, "husband[%i]" % i) for i in range(n)]

 for m in range(n):
   solver.Add(solver.Element(husband, wife[m]) == m)

 for w in range(n):
   solver.Add(solver.Element(wife, husband[w]) == w)

 rankWomen[o,husband[o]] <
 for m in range(n):
   for o in range(n):
     b1 = solver.IsGreaterCstVar(
         solver.Element(rankMen[m], wife[m]), rankMen[m][o])
     b2 = (
         solver.IsLessCstVar(
             solver.Element(rankWomen[o], husband[o]), rankWomen[o]
[m]))
     solver.Add(b1 - b2 <= 0)

 for w in range(n):
   for o in range(n):
     b1 = solver.IsGreaterCstVar(
         solver.Element(rankWomen[w], husband[w]), rankWomen[w][o])
     b2 = solver.IsLessCstVar(
         solver.Element(rankMen[o], wife[o]), rankMen[o][w])
     solver.Add(b1 - b2 <= 0)

 solution = solver.Assignment()
 solution.Add(wife)
 solution.Add(husband)

 db = solver.Phase(wife + husband,
                    solver.CHOOSE_FIRST_UNBOUND,
                    solver.ASSIGN_MIN_VALUE)
```

Crypt Arithmetic

```
  digits = range(0, 10)
  s = solver.IntVar(digits, 's')
  e = solver.IntVar(digits, 'e')
  n = solver.IntVar(digits, 'n')
  d = solver.IntVar(digits, 'd')
  m = solver.IntVar(digits, 'm')
  o = solver.IntVar(digits, 'o')
  r = solver.IntVar(digits, 'r')
  y = solver.IntVar(digits, 'y')

  letters = [s, e, n, d, m, o, r, y]

  solver.Add(1000 * s + 100 * e + 10 * n + d + 1000 * m + 100 * o + 10 * r
+ e ==
             10000 * m + 1000 * o + 100 * n + 10 * e + y)

  solver.Add(s != 0)
  solver.Add(m != 0)

  solver.Add(solver.AllDifferent(letters))

  solver.NewSearch(solver.Phase(letters,
                                solver.INT_VAR_DEFAULT,
                                solver.INT_VALUE_DEFAULT))
```

References

1) Automata course by Jeff Ullman from Stanford University (Coursera)

2) Algorithms Design and Analysis by Tim Roughgarden from Stanford University (Coursera)

3) Discrete Optimisation by Professor Pascal Van Hentenryck (Coursera)

4) https://code.google.com/p/or-tools/