Heuristic Algorithms for Optimization
AI2101 - Group 8

1st Member : K.N Vardhan
Mathematics and Computing
MA20BTECH11006
ma20btech11006@iith.ac.in

2nd Member : Tata Sai Manoj
Mathematics and Computing
MA20BTECH11018
ma20btech11018@iith.ac.in

3rd Member : VKS Deepak Reddy
Mathematics and Computing
MA20BTECH11019
ma20btech11019@iith.ac.in

***Abstract*—This paper covers three heuristic algorithms in optimization, consisting of the Travelling Salesman Problem using a heuristic approach (Simulated Annealing), Tabu-Search heuristic and Ant Colony Optimization (ACO) heuristic for the Knapsack Problem.**

I. Introduction

A heuristic algorithm is a procedure that produces near-optimal solutions to an optimization problem. However, such solutions are achieved by trading optimality, completeness, accuracy, or precision for speed.

II. The Travelling Salesman Problem

A. Introduction to the problem

The Travelling Salesman Problem (also denoted by **TSP** (or) **TSP-OPT**) is a well-known optimization problem. However, the **TSP-OPT** is NP-hard. We do not know an efficient way to verify a solution for **TSP-OPT**, so we cannot say that it is NP-complete.

B. Problem Description

The problem can be simply stated as follows : For a given list of cities and the distances between pairs of cities, we have to find the shortest possible route that goes through each city exactly once. A naive approach would be to calculate the distance for each possible route, which would involve calculating distances for $n!$ routes. However, this is extremely inefficient, and the time required for computation blows up.

C. Formulation of the problem

The **TSP** can be formulated as an integer linear programming problem. Here, N represents the number of cities and obj_{ij} represents the cost of travelling from city i to city j. The decision variables are x_{ij}, which can take a value of 0 or 1, depending on the existence of an edge between city i and city j.

$$\begin{aligned} &\text{minimize} && \Sigma_{i\in n}\Sigma_{j\in n}(obj_{ij}) \times (x_{ij}) \\ &\text{subject to} && \Sigma_{i\in n}x_{ij} = 1 \ \forall j \in N \\ & && \Sigma_{j\in n}x_{ij} = 1 \ \forall i \in N \\ & && \Sigma_{i\in n}x_{ii} = 0 \end{aligned}$$

D. Why is TSP NP-hard?

We claim without proof, that any instance of the Travelling Salesman Problem can be reduced to verifying if a graph has a Hamiltonian Cycle. It is well known that this problem is NP-hard.

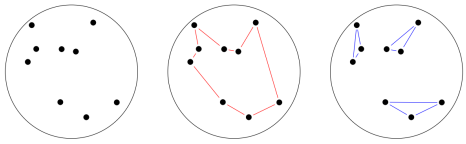

Fig. 1. Intuition for the NP-hardness of TSP-OPT

The third diagram satisfies the conditions of the given problem, but it is not "connected". We have to avoid such "local optima" and satisfy the global constraint of staying "connected".

III. Heuristic Approach : Simulated Annealing

The heuristic that we use to find a near-optimal solution for the **TSP** is Simulated Annealing (**SA**). The word "annealing" is an analogy to a thermodynamic process in which metal is raised to a high temperature and then gradually decreased to obtain it in a desired state. Similarly, our algorithm starts from a initial solution (say x), and keeps generating candidate solutions (say y_i) until the stopping criterion for i is reached.

A. Why this heuristic?

Simulated Annealing heuristic is only dependent on the following conditions :-

- The initial solution (x)
- The choice of each candidate solution (y_i)
- The stopping criterion

B. How the algorithm works?

- The **moving direction** must be determined in a probabilistic way at each step because we do not want to get stuck at a local optimum.
- The **search step** is gradually reduced in size (after every iteration), because the choices we make as we approach the optimal result must be highly optimized.

C. Details of the algorithm

The following are the steps performed by the algorithm : -

1) Create an initial solution (x) by obtaining a random shuffle of the cities. This is our initial route navigating through the cities.
2) For every iteration of the algorithm, two of the cities are swapped. This is continued until the stopping criterion is satisfied.
3) If the overall cost function is reduced, the swap is kept, and a new candidate solution (y_i) is generated.
4) If the cost function increases, the swap is not **rejected**. The swap may be included with certain probability.
5) Now, the temperature is updated after the iteration, slowly "cooling" down (this affects the chances of rejected swaps in the candidate solution).

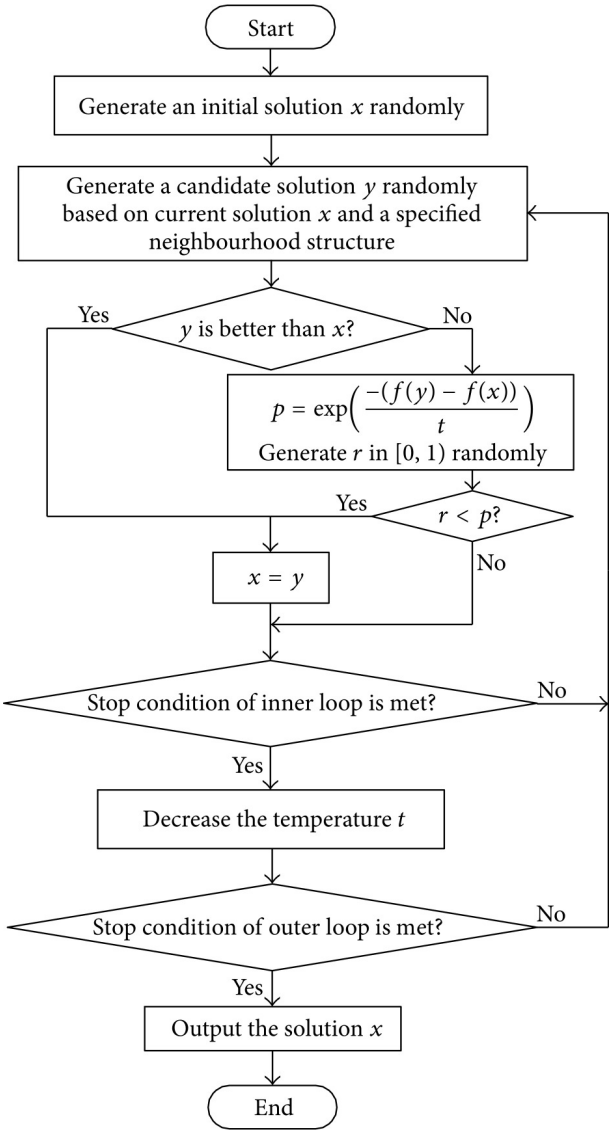

Fig. 2. A general SA algorithm

D. High Level Algorithm Description

Variables: Initial Temperature T_0, Minimum Temperature T_{min}, Maximum number of iterations K_{max}, Probability of temperature drop ρ, Objective Function obj

Output: Generating the optimal solution

- Generate an initial solution x_0
- $x_{opt} = x_0$
- Compute $obj(x_0)$ and $obj(opt)$
- $T_{opt} = T_0$

while $T_{opt} > T_0$ **do**
  $\Delta f = f(x_{curr}) - f(x_{opt})$
  if $\Delta f < 0$ **then**
    $x_{opt} = x_{curr}$;
  end
  if $\Delta f >= 0$ **then**
    $\rho = e^{(\frac{\Delta f}{t})}$;
    if *rand(0, 1)* $\geq \rho$ **then**
      $x_{curr} = x_{new}$;
    end
  end
  $i = i + 1$;
  $T_i = \rho \times T_i$
end

E. Some more possible optimizations

- The probability function that is used in the description of the algorithm is the random function, which selects a value between $[0, 1]$. We can use better probabilistic distributions which give a more optimal solution in lesser number of iterations.
- The temperature can be updated in multiple different ways (other than the one given in the description). Quadratic cooling, exponential cooling and logarithmic cooling can capture the variation better than linear cooling.

F. Visualization

The following **link** is an animation of the Travelling Salesman Problem with Simulated Annealing.

Tabu Search Heuristic

I. Introduction

When we are working on convex optimization problems, the local optima is always the global optima, so the local optimization techniques converge to local optimum but we get still get the global optima. However, in case of non-convex problems, when we use convex optimization techniques we might end up with some local optima which is not the global optima as such a method only permits moving to neighboring solutions that improve the current objective function value and ends when no improving solutions can be found. So, we need techniques that can escape from local optima in order to search the solution space beyond local optima, more extensively and effectively to reach the global minima.

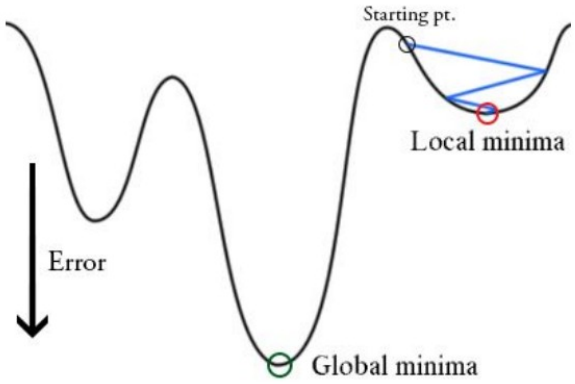

Fig. 3. Local Optima and Global Optima

Also, when solving a real-world problem, finding the optimal global solution in reasonable amount of time is necessary . However, finding exact solutions is difficult because resources are constrained, and most of the optimization problems are complex. Heuristic algorithms can solve this difficulty by offering approximate solutions. Different Heuristic algorithms are pretty extensive to various applications to solve nonlinear non-convex and combinatorial optimization problems.

Tabu search is one such Heuristic algorithm, which incorporates an iterative, adaptive memory-based neighborhood-search method. It can used on non-convex problems to get an approximate globally optimal solution.

II. Understanding Tabu Search

Tabu search basically begins in the same way as an ordinary local or neighborhood search, proceeding iteratively from one point (solution) to another until a chosen termination criterion is satisfied.

A. Neighbourhood construction

Consider X to be the search space, each solution x has an associated neighborhood $N(x) \subset X$, and each solution $x \in N(x)$ is reached from x by an operation called a move. Tabu search permits moves that deteriorate the current objective function value and selects the moves from a modified neighborhood $N^*(x)$, which is the result of maintaining a selective history of the states encountered during the search.

Such construction of the neighborhood reinforce the primary interest of TS, which is to define neighborhoods in dynamic ways that can allow search to move beyond local optima and include simultaneous consideration of multiple types of moves.

B. Recency based memory and Frequency based memory

The Selective history uses attributive memory, the type of memory records information about solution properties (attributes) that change in moving from one solution to another rather than the solution itself to save space.

There are two types of **attributive memory**, Recency based memory structures and frequency based memory structures. **Recency based memory structure** keeps track of solutions attributes that have changed during the recent past. Selected attributes that occur in solutions recently visited are labeled tabu-active and these changes of attributes are restricted up to certain iteration (**Tabu tenure**). This prevents certain solutions from the recent past from belonging to $N^*(x)$ and hence from being revisited.

Frequency based memory structures keep track of solution attributes that occur with certain frequency in previous iterations and make such changes in attributes, tabu active. Both of these are short-term memories and encourage **Diversification** (drive the search into regions dissimilar to those already examined) which is particularly helpful when better solutions can be reached only by crossing barriers or "humps" in the solution space topology.

Intensification (Jump to or initiate a return to regions in the configuration space in which some stored elite solutions lie: these regions can then be searched more thoroughly) strategies require a means for identifying a set of elite solutions as basis for incorporating good attributes into newly created solutions. To store these we need Long term memory structures. A key element of the adaptive memory framework of tabu search is to create a balance between search intensification and diversification.

Moves or solutions that are part of the **Aspiration Criteria** (optional) cancel out the Tabu and the move can be made even if it's in the Tabu List. This can also be used to prevent **stagnation** in cases where all possible moves are prohibited by the Tabu List.

III. Algorithm for Tabu Search

A. Steps in the Algorithm

Step 1: Start with an initial solution $x = x_0$. This can be any solution that fits the criteria for an acceptable solution. And initialize best solution $x_b = x_0$.

Step 2: Generate a set of neighbouring solutions to the current solution x labeled $N(x)$. From this set of solutions, the solutions that are in the Tabu List are removed with the exception of the solutions that fit the Aspiration Criteria. This new set of results is the new $N^*(x)$.

Step 3: Choose the best solution out of $N^*(s)$ and label this new solution x'. If the solution x' is better than the current best solution, update the current best solution, $x_b = x'$. Then, regardless if x' is better than x, we update x to be x'.

Step 4: Update the Tabu List $T(x)$ by removing all moves that are expired past the Tabu Tenure and add the new move x' to the Tabu List. Additionally, update the set of solutions that fit the Aspiration Criteria $A(x)$. If frequency memory is used, then also increment the frequency memory counter with the new solution.

Step 5: If the Termination Criteria are met, then the search stops or else it will move onto the next iteration. Termination Criteria is dependent upon the problem at hand but some possible examples are:

1) a max number of iterations
2) if the best solution found is better than some threshold

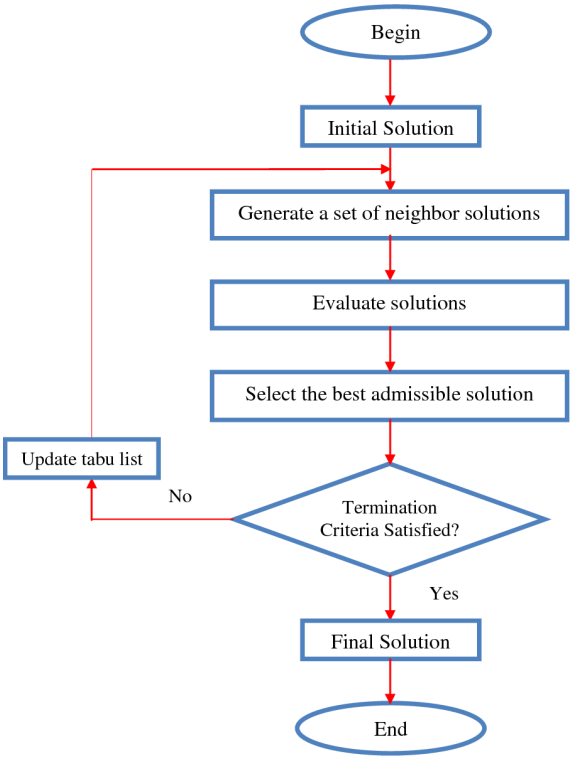

Fig. 4. Tabu Search Flowchart

B. Single Machine Total Weighted Tardiness Problem (SMTWTP)

Consider a set of $n \in N$ jobs that need to be processed on a single machine that can handle at most one job at a time. Each job j is assigned to a processing time $p_j N \geq 0$ that describes the time that is needed to process job j, a due date $d_j \in N \geq 0$ that describes the time point when the processing of job j should have been finished, and a weight $w_j \in N \geq 0$ that represents the priority of job j.

Given such a set of n jobs, a schedule π is a permutation of length n, i.e., a bijective mapping

$$\pi\colon [1, 2, ..., n] \rightarrow [1, 2, ..., n]$$

that assigns to each place i in the queue a job $\pi(i)$. For the sake of a clear notation, we represent a permutation π as the n-tuple $(\pi(1), ..., \pi(n))$. Clearly, a schedule π defines a total order in which the n jobs are processed on a single machine with $\pi(1)$ (respectively $\pi(n)$) being the first (respectively last) job in π.

For a given schedule π, the completion time C_j of a job j is the time that is needed to complete job j in π, i.e.,

$$C_j = \Sigma_{i=1}^{i \leq \pi^-(j)} p_{\pi(i)}$$

where $\pi^-(j)$ denotes the position of job j in π. The tardiness T_j of a job j is defined as

$$T_j := max\{C_j - d_j, 0\}$$

Note that the tardiness cannot be negative and thus, it can be seen as a penalty for completing a job after its due date. Given a set of n jobs, SMTWTP aims to find a schedule of all n jobs that minimizes the weighted tardiness of all jobs, i.e., it aims to minimize the objective function.

$$\Sigma_{j=1}^{n} w_j T_j$$

The expression of a schedule π is also called the total weighted tardiness of π. In terms of only initially given variables our objective is

$$\min(\Sigma_{j=1}^{n} w_j * max\{\Sigma_{i=1}^{i \leq \pi^-(j)} p_{\pi(j)} - d_j, 0\})$$

C. Pseudocode for Tabu Search Algorithm

```
TabuSearch()

    START while termination
        criterion is false
        SET value = BestMove ()
        MAKE best_move
        MAKE best_move tabu
        if value < global_best
            global_best = value
        END if
    END while
END TabuSearch

BestMove()

    START for i less than n
        START for j = i + 1 till n
            swap (sequence[i,j])
            value = objective(sequence)
            if tabu[i][j] &&
               value > global_best
                continue
            END if
            if value < best_so_far
                best_so_far = value
                best_move = [i,j]
            END if
        END for
    END for
    return best_so_far

END BestMove
```

Knapsack Problem

I. Introduction

A heuristic algorithm sacrifices optimality, accuracy, precision, or completeness for speed to solve a problem faster and more efficiently than standard approaches. When approximate answers are satisfactory but exact solutions are computationally costly, heuristic algorithms are frequently used.

One of the common use of heuristics is to solve the Knapsack problem. The knapsack problem is a problem in combinatorial optimization. Given a set of items, each with a weight and a value, determine the number of each item included in a collection. The total weight is less than or equal to a given limit, and the real value is as significant as possible.

II. Knapsack Problem

The heuristic algorithm for this problem used is *Greedy Approximation Algorithm*, which sorts the items based on their value per unit mass and adds the items with the highest v/w (value/weight) as long as there is still space remaining. There exist various kinds of knapsack problems.

Different Knapsack Problems: 0-1 knapsack problem, bounded knapsack problem (BKP), unbounded knapsack problem (UKP).

The knapsack problems are interesting from different perspectives.

- The decision problem form of the knapsack problem is NP-complete. Thus, there is no known algorithm for different test cases, both correct and fast.
- There is a pseudo-polynomial time algorithm using dynamic programming.
- While the decision problem is NP-complete, the optimization problem is not, and there is no known polynomial algorithm that can tell, given a solution, whether it is optimal

In order to obtain the solution of a 0-1 Knapsack problem in a short time period, we use heuristic algorithms. Here, we see the Ant Colony Optimization (ACO) algorithms to provide better performance for solving the 0-1 Knapsack Problem. These algorithms are a class of algorithms inspired by observing the real ants. Real ants can locate the quickest path between their colony and a food source through biological principles.

A. Mathematical formulation

In the 0-1 knapsack problem, the given elements are:

n objects,
i^{th} object has weight W_i,
P_i is the profit of i^{th} object,
M as the weight limit of the knapsack.

Our *objective* is to maximize the profit P_{net} under the *constraint* that the total weight W_{total} is at most equal to M.

Thus, it can be formulated as

$$Maximize\ z = \sum_{i=1}^{n} P_i X_i,\ i = 1, 2, \ldots n$$

And the constraint being

$$\sum_{i=1}^{n} W_i X_i \leq M,\ i = 1, 2, \ldots n; \quad X_i \in \{0, 1\}$$

where X_i is the object variable to include or not in the knapsack.

B. ACO Algorithm

This is a cooperative evolutionary strategy where numerous generations of artificial ants hunt for suitable solutions. Ants are formed randomly on nodes and migrate stochastically from a starting node to viable adjacent nodes.

The proposed algorithms are shown here, including a flowchart as well for better understanding.

while *ACO hasn't stopped* **do**
  Ants-generation-activity()
  Pheromone-evaporation()
  Daemon-actions()
end

Each ant builds a complete solution to the problem in several stages, with an intermediate or partial solution at each step. The ant k develops a new intermediate solution by moving from node i to node j in each step. The solution will is obtained in a specific number of steps. Each ant k analyzes a set of possible expansions from its present node at each step and goes to one of them with a high probability.

The three functions Ants_generation_activity(), Pheromone_evaporation(), and Daemon_actions() do specific actions to build up the optimization.

1) **Ants_generation_activity()**
   The Ants find solutions starting from a start node and moving to feasible neighbor nodes. During this process, the data collected by the ant will be stored in pheromone trails. The search for neighbor nodes is driven stochastically by an ant-decision rule, that is made up of pheromone and heuristic data.
2) **Pheromone_evaporation()**
   Here, we decrease the intensities of pheromone trails over time. The pheromone amount diminishes for all objects, which is represented by evaporation. The quantity of the pheromone rises when an additional quantity is added to all objects, which constitutes the best solution. A higher concentration of the pheromone suggests that ants are more likely to choose the object to search for the optimum solution to the problem.
   This process is done to avoid convergence from various ants and to explore more search areas.
3) **Daemon_actions()**
   It is optional for ant colony optimization, and they are often used to collect useful global data by depositing additional pheromone.

In ACO, the Ant starts from a random start node and successively visits other nodes until all of them are visited. The Ant-decision table for the Ant k doing the activity

node i to node j is governed by

$$\pi(j) = \begin{cases} \arg\{\max\limits_{i=allowed_k(t)} [\tau_{ij}(t) \cdot \eta_{ij}^{\beta}]\} & q \leq q_0 \\ S & otherwise \end{cases}$$

where

- η_{ij} is the heuristic information and is the set as the highest value of d_j
  $(\eta_{ij} = 1/d_j)$
- β is a parameter representing the importance of heuristic information
- q is a random number uniformly distributed in $[0, 1]$
- q_0 is a pre-specified parameter $(o \leq q_0 \leq 1)$
- $allowed_k(t)$ is the set of all feasible nodes that are not yet visited by k
- t is at that particular time
- S is an index of the node selected from $allowed_k(t)$

Here, in finding a feasible solution, Ants perform step-by-step pheromone updates as:

$$\tau_{ij} = \tau_{ij} + \Delta\tau_{ij}$$

where $\Delta\tau_{ij}$ is the amount of pheromone deposited from the ant while moving from node i to node j. The above step is repeated until all the ants have found a feasible solution.

The intensity of evaporation is controlled by the parameter ρ. The quantity of the pheromone on each node is updated at the end of every cycle. The pattern of updation of quantity of pheromone on each object at the end of every cycle is

$$\tau = \rho\tau \quad \rho \in (0, 1]$$

The above algorithm is repeated until the stop criteria are met.

Depicting the above algorithm in a flowchart manner.

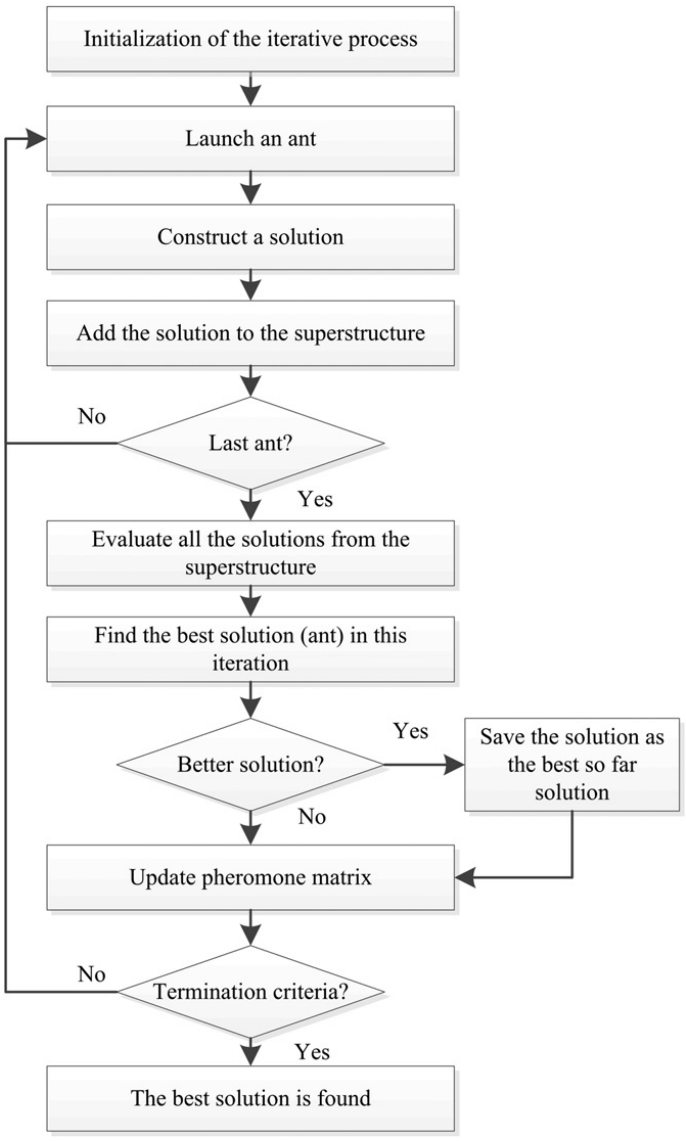

Fig. 5. Knapsack Flowchart

REFERENCES

I. TRAVELLING SALESMAN PROBLEM

[1] Travelling Salesman Problem
Wikipedia: TSP Problem
[2] Simulated Annealing
TSP with Simulated Annealing
[3] More on Simulated Annealing
Comparison with other methods
[4] ** Credit to Perry Geo for Visualize TSP with Simulated Annealing
Local Host animated TSP

II. KNAPSACK PROBLEM

[5] Knapsack Problem
Wikipedia: knapsack problem
[6] A New Approach for Solving 0-1 Knapsack Problem
2006 IEEE Internation Conference on Systems, Man, and Cybernetics
[7] The 0-1 Knapsack Problem
Ant Colony Optimization Algorithm

III. TABU SEARCH ALGORITHM

[8] Tabu Search
Wikipedia: Tabu Search
[9] Tabu Search
Research Paper on Tabu Search : Fred Glover, Rafael Marti
[10] Metaheuristic in Optimization : Algorithmic Perspective
Metaheuristic in Optimization
[11] Tabu Search meta-heuristic: Motivations and basic ideas
Heuristic : Tabu Search
[12] Tabu Search : A comparative study
Comparative Study
[13] Tabu Search : A brief survey and some real-life applications
Applications