



# Heuristic Optimization Algorithms

Group 8

KN Vardhan

Tata Sai Manoj

VKS Deepak Reedy

# What is a Heuristic Algorithm?

- A heuristic algorithm is a method for finding near-optimal solutions to a problem of optimization.
- Heuristics are a commonly utilized technique for a variety of reasons.

# What is a Heuristic Algorithm?

- Problems for which there is no exact answer or whose formulation is unknown, or if it is computationally intensive.
- However, the trade-off is in terms of optimality **(or)** correctness in favor of speed.

# Travelling Salesman Problem

- We have to find the shortest tour that visits each city exactly once and then return to the starting city.
- Belongs to the class of NP-hard optimization problems.
- The Hamiltonian cycle problem is a well-known NP-complete problem, which implies the NP-hardness of TSP.

# Reducing Hamiltonian Cycle to TSP

- If we consider a graph  $G$ , we can find the complement of the graph in polynomial time.
- Assign weights of '0' to the old edges, and weights of '1' to the new edges.
- If we find a Hamiltonian cycle of cost  $= 0$ , then the cycle only consists of the edges present in the original graph. This implies that there is a solution to the Travelling Salesman Problem for that graph.



# Formulation of TSP

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

# Simulated Annealing

- Simulated Annealing is the heuristic we apply to find a near-optimal solution for the TSP.
- The term “annealing” refers to a process in thermodynamics in which metal is heated to a high temperature and then cooled gradually to achieve the desired shape.

# Simulated Annealing

- Similarly, our algorithm begins with a initial solution (generated), and then continues to generate candidate solutions until the stopping criterion is satisfied.
- As the number of iterations continue to grow, only highly optimum choices are made to the best solution. This is indicated by the decreases in temperature, which increases the value of “rho”, in turn making it harder for the probability distribution to pick an non-optimal choice.



# High Level Description (Pseudocode)

- The following [link](#) is the animation of the Travelling Salesman Problem with Simulated Annealing.

**Variables:** Initial Temperature  $T_0$ , Minimum Temperature  $T_{min}$ , Maximum number of iterations  $K_{max}$ , Probability of temperature drop  $\rho$ , 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  $\Delta f \geq 0$  then
        |  $\rho = e^{(\frac{\Delta f}{t})}$ ;
        | if  $rand(0, 1) \geq \rho$  then
        | |  $x_{curr} = x_{new}$ ;
        | end
    end
    end
     $i = i + 1$ ;
     $T_i = \rho \times T_i$ 
end
```

# Optimizations in the Algorithm

- The temperature can be changed in a variety of ways (other than the one given in the description).
- Linear cooling cannot capture the fluctuation efficiently.
- Linear cooling : -  $T(k) = T_{max} - \alpha k$

Natural Logarithmic cooling : -  $T(k) = T_{max} \alpha^k$

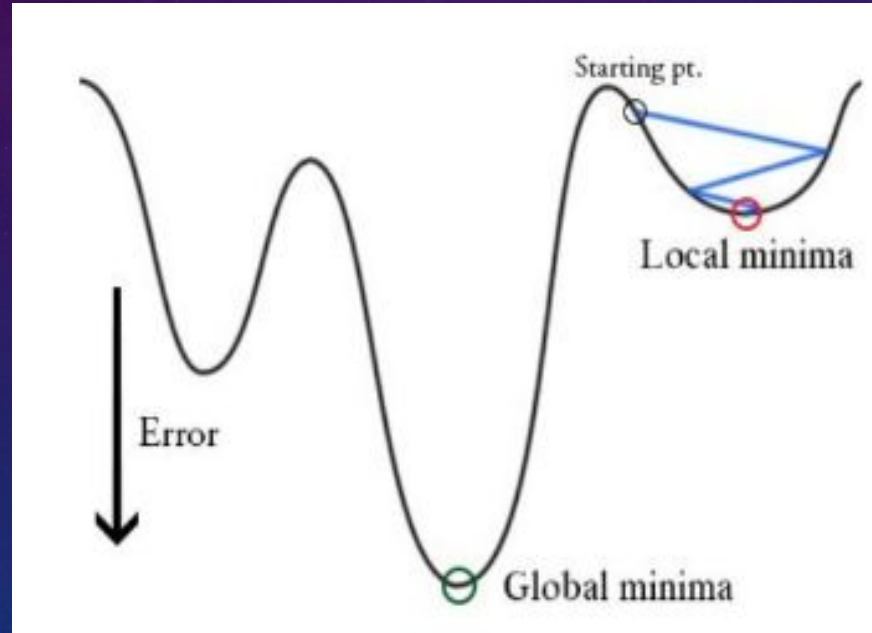
Exponential cooling : - 
$$= \frac{T_{max}}{1 + \alpha \log(k + 1)}$$

Quadratic cooling : - 
$$T(k) = \frac{T_{max}}{1 + \alpha k^2}$$

# Tabu Search

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

# Tabu search for non-convex problems



# Motive for Tabu Search Heuristic

- Tabu search enhances the performance of local search by relaxing its basic rule.
- First, at each step worsening moves can be accepted if no improving move is available (like when the search is stuck at a strict local minimum).
- If a potential solution has been previously visited within a certain short-term period or if it has violated a rule, it is marked as "tabu" (forbidden) so that the algorithm does not consider that possibility repeatedly. Tabu Search uses its memory to search beyond local optima.



# Single Machine Total Weighted Tardiness Problem (SMTWTP)

- Consider a set of jobs that need to be processed on a single machine that can handle at most one job at a time. Each job is assigned to a processing time that describes the time that is needed to process job, a due date that describes the time point when the processing of job should have been finished, and a weight that represents the priority of job.
- Given such a set of jobs, we need to find the schedule with minimum total weighted tardiness.
- This is a non-convex, combinatorial, NP-hard problem, we demonstrate the tabu search algorithm based on the SMTWTP.

# Mathematical modeling

Number of jobs  $= n \in \mathbb{N}$

Processing time of job  $j = p_j$

Due date of job  $j = d_j$

Weight of job  $j = w_j$

$\pi$  is a permutation of length  $n$  which is a bijective mapping

$\pi : [1, 2, \dots, n] \rightarrow [1, 2, \dots, n]$

we represent a permutation  $\pi$  as the  $n$ -tuple  $(\pi(1), \dots, \pi(n))$

Completion time of job  $j = C_j$

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

Tardiness of job  $j = T_j$

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

# Mathematical modeling

Objective : to minimize the total tardiness of all jobs

Total tardiness is give by  $\sum_{j=1}^n w_j T_j$

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

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

# Algorithm Demonstrating Tabu search on SMTWTP

- Start with an initial schedule and initialize current schedule to initial schedule. This can be any schedule that fits the criteria for an acceptable solution. And initialize best schedule to initial schedule.
- Generate a set of neighbouring schedules to the current schedule. From this set, the schedules that are formed by moves in the **Tabu List** are removed with the exception of the schedules that fit the **Aspiration Criteria**.
- Choose the best schedule out of this modified neighbourhood. If this schedule is better than the current best schedule, update the current best schedule to this. Then, regardless if it is better than current schedule, we update current schedule to this schedule.

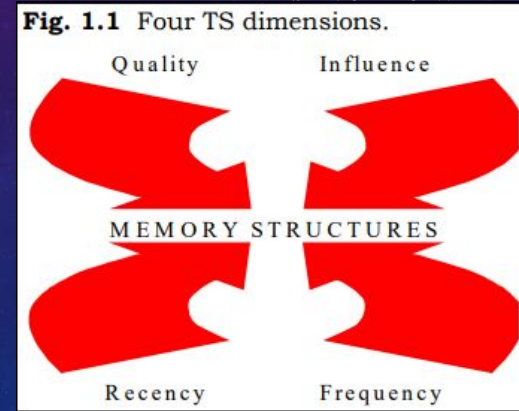
# Algorithm Demonstrating Tabu search on SMTWTP

- Update the Tabu List by removing all moves that are expired past the **Tabu Tenure** and add the new move( from previous current schedule to present )to the Tabu List. If frequency memory is used, then also increment the **frequency memory counter** with the new move.
- If the Termination Criteria, number of iterations reached maximum iteration, is met, then the search stops or else it will move onto the next iteration.



# Key features of Tabu Search

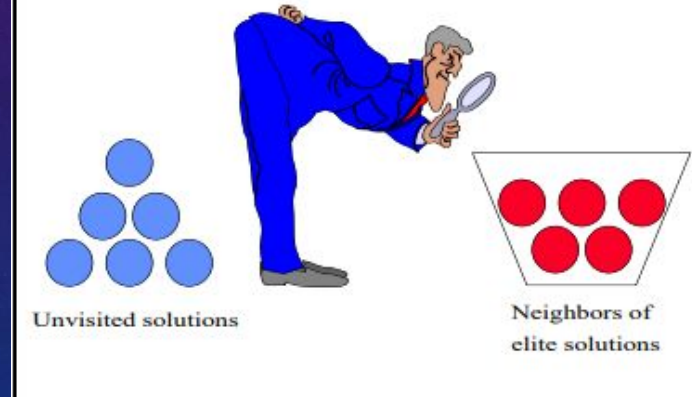
- *Recency based memory structure* keeps track of solutions attributes that have changed during the recent past.
- *Tabu tenure* is the iteration up which certain move in tabu table is restricted.
- *Frequency based memory structures* keep track of moves that occur with certain frequency in previous iteration.

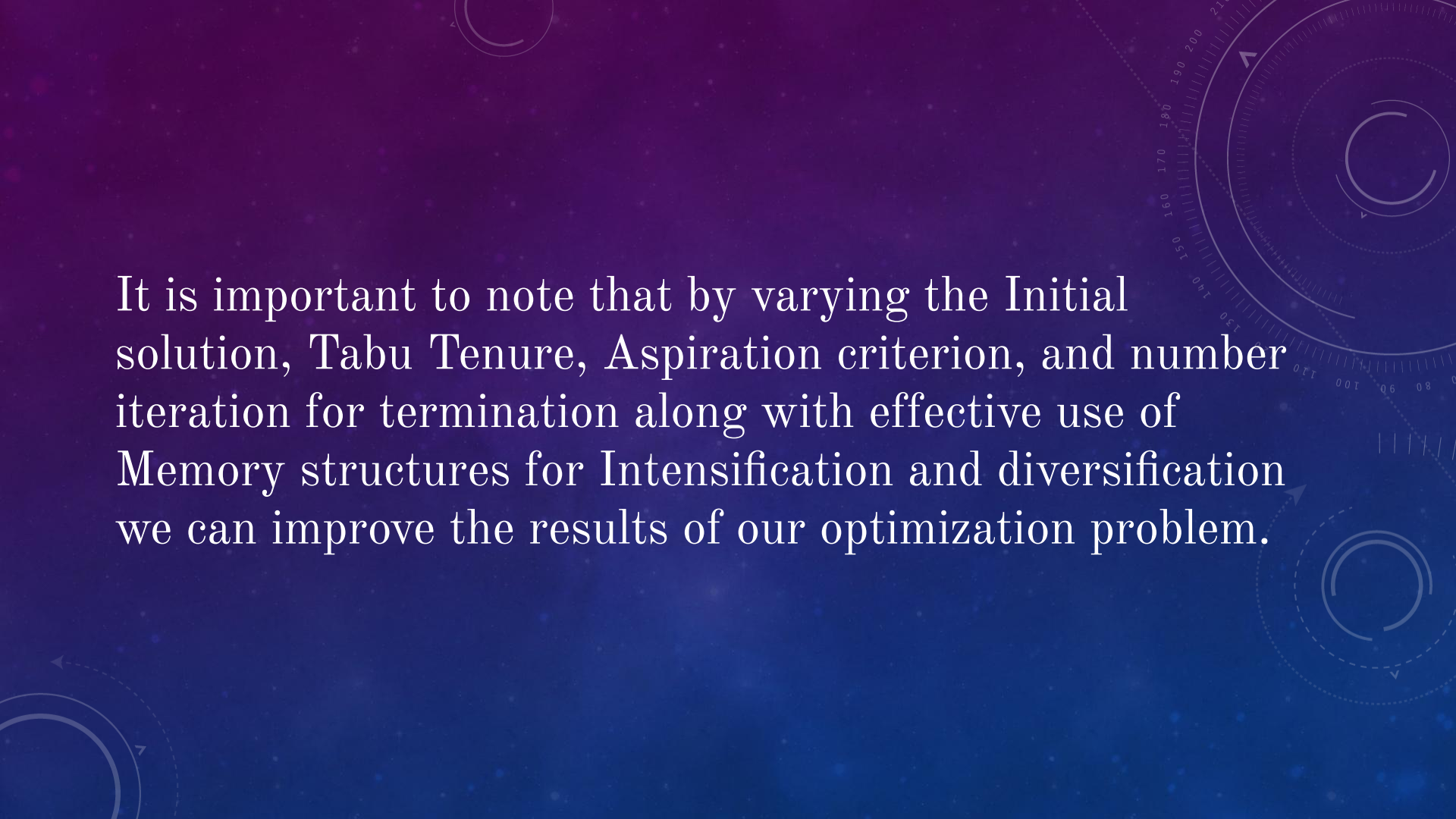


# Key features of Tabu Search

- *Diversification* drives the search into regions dissimilar to those already examined.
- *Intensification* is to initiate a return to regions in the configuration space in which some stored elite solutions lie, these regions can then be searched more thoroughly.
- *Aspiration Criteria*(optional) cancel out the Tabu and the move can be made even if it's in the Tabu List.

**Fig. 1.2** Intensification and diversification.

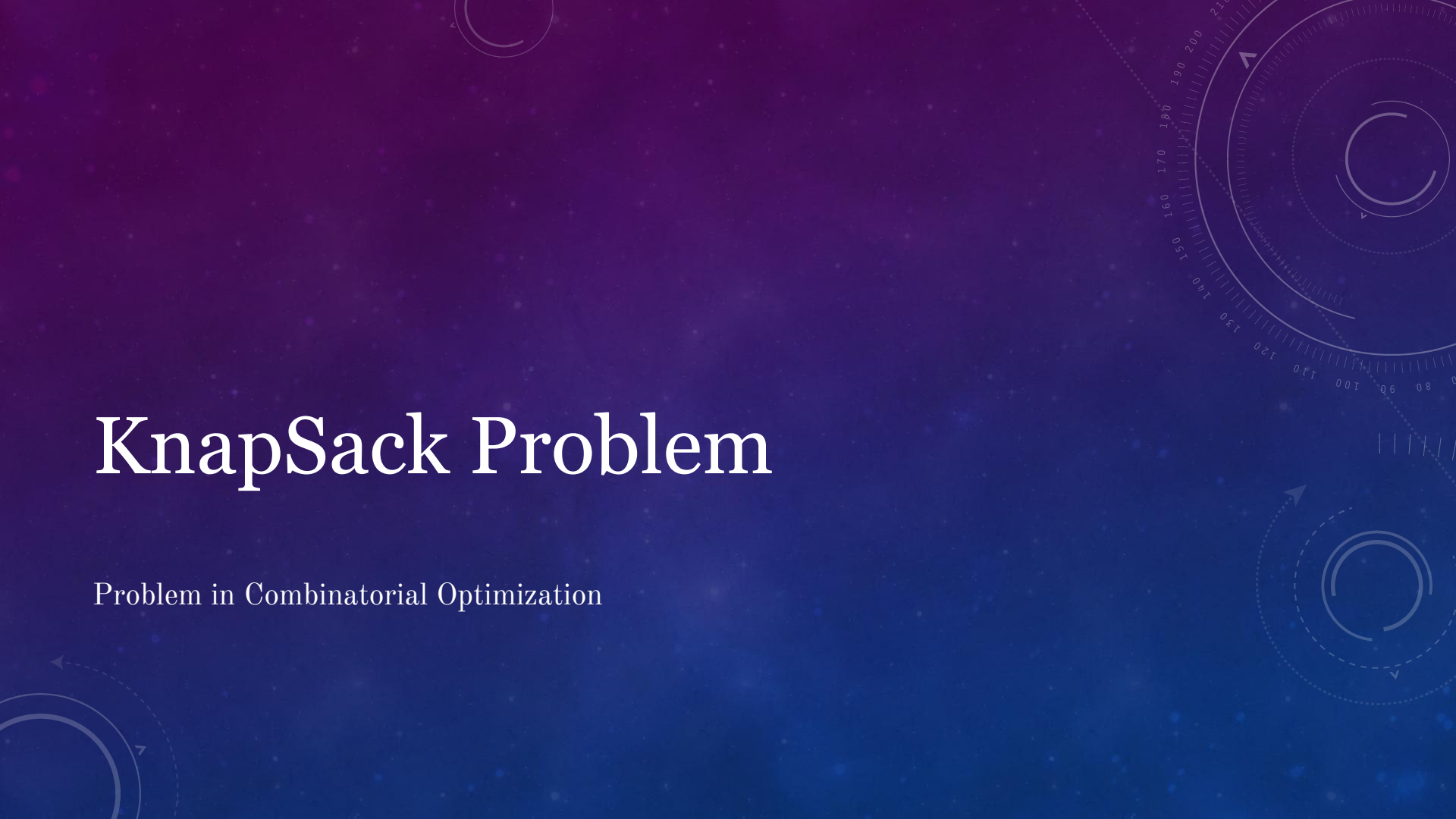




It is important to note that by varying the Initial solution, Tabu Tenure, Aspiration criterion, and number iteration for termination along with effective use of Memory structures for Intensification and diversification we can improve the results of our optimization problem.

# KnapSack Problem

Problem in Combinatorial Optimization



# Knapsack Problem

- A heuristic algorithm sacrifices optimality, accuracy, precision to solve a problem faster and more efficiently than standard approaches. When approximate answers are satisfactory and getting exact solutions become computationally difficult, we use heuristic algorithms.
- One of the most common use of heuristic comes in the Knapsack Problem. The heuristic algorithm for this problem used is *Greedy Approximation Algorithm*. The *Greedy Approximation Algorithm* sorts the items based on their value per mass and adds the items with the highest ratio (value / mass) as long as there is space remaining.
- Here, we try to find the best possible approximate answer using the *Ant Colony Optimization (ACO)* for the 0-1 Knapsack Problem.



# What is a Knapsack Problem

- The knapsack problem is described as;
  - Given a set of items,
  - Each item has a certain value “v”, and weight “w”.
  - Goal is to make  $V (= \sum v)$  as maximum as possible with  $W (= \sum w) \leq K$ .
  - With the above goal (constraint), we need to find list of such items.
- Different types of knapsack problems:
  - 0-1 Knapsack Problem
  - Bounded Knapsack Problem (BKP)
  - Unbounded Knapsack Problem (UKP)

# Why are knapsack problems of special interest?

- The decision problem form of the knapsack problem is NP-Complete. Thus, there is no known algorithm for differentiating various 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. Thus, there is no concreteness to prove that a given solution is optimal or not. Given on the above queries, solving knapsack problem is quite challenging.

# Before going to ACO Algorithm

```
val = []      # list of items' values
wt = []      # list of items' weights
W          # maximum weight for the knapsack
n = len(val)
t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

def knapsack(wt, val, W, n):
    .....    # function defined next to the main code

print(knapsack(wt, val, W, n))
```

```
def knapsack(wt, val, W, n):
    if n == 0 or W == 0: return 0
    if t[n][W] != -1: return t[n][W]
    if wt[n-1] <= W:
        t[n][W] = max(
            val[n-1] + knapsack(
                wt, val, W-wt[n-1], n-1),
            knapsack(wt, val, W, n-1))
    return t[n][W]
    elif wt[n-1] > W:
        t[n][W] = knapsack(wt, val, W, n-1)
    return t[n][W]
```

# Ant Colony Optimization Algorithm

- An evolutionary strategy developed after observing how ants find the shortest & quickest path between their colony and a food source.
- Ants are formed randomly on nodes and migrate stochastically from a starting node to viable adjacent node.
- Each ant builds a complete solution using several steps, with an intermediate/partial solution at each step.

# Ant Colony Optimization Algorithm

- An ant  $k$  develops an intermediate solution by moving from node  $i$  to node  $j$  in each step.
- The ant analyzes a set of possible expansions from its present node at each step and goes to one of them with highest probability of success.
- The best solution outcome from every ant's solution will be taken as our most approximate optimal solution.



The background is a gradient of deep blue and purple, speckled with white dots resembling a starry sky. Overlaid on this are several faint, white geometric patterns. In the top right, there is a large circular scale with degree markings from 0 to 210 and concentric circles. In the bottom right, there are dashed circular lines with arrows indicating a clockwise direction. In the bottom left, there are solid circular lines with an arrow indicating a counter-clockwise direction. In the top left, there is a small circular arc with an arrow.

THANK YOU