DATA STRUCTURES PROJECT REPORT

# MULTIDIMENSIONAL KNAPSACK USING TABU SEARCH

REVIEW 3

*DURBHAKA ANIL KUMAR 15BCE0666*

*VOLETI RAVI 15BCE0082*

*TANVI ANAND 15BCE0755*

SLOT-G1

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

VIT® UNIVERSITY

(Estd. u/s 3 of UGC Act 1956)

VELLORE ▪ CHENNAI

www.vit.ac.in

## WHAT IS TABU SEARCH:

Tabu search, created by Fred W. Glover in 1986 and formalized in 1989, is a metaheuristic search method employing local search methods used for mathematical optimization.

In its best known form, tabu search can be viewed as beginning in the same way as ordinary local or neighborhood search, proceeding iteratively from one point (solution) to another until a chosen termination criterion is satisfied. 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. We may contrast TS with a simple descent method where the goal is to minimize f(x). Such a method only permits moves to neighbor solutions that improve the current objective function value and ends when no improving solutions can be found. The final x obtained by a descent method is called a local optimum, since it is at least as good as or better than all solutions in its neighborhood. The evident shortcoming of a descent method is that such a local optimum in most cases will not be a global optimum, i.e., it usually will not minimize f(x) over all $x \in X$ .

## THE GENERAL FORM OF A TABU SEARCH ALGORITHM:

```
 Tabu search (Xinit , Nb_div, Nb_int, Nb_local, Nb_Drop, Lt_length, BestSol_array)

 /* X is the actual solution */

•     X=Xinit ; LtLt_length={};

•     For i=0 to Nb_div do

•         For j=0 to Nb_int do

•             X*local=X;
```

```
/* move */
•       go from X to X' by a sequence of Nb_Drop/Add;
•       If F(X')> F(X ) then X*=X' ;X*local =X';
    Else If F(X')>F(X*local ) Then X*local =X';
•       If X' is a part of the B Best solutions
    Then insert X' in the BestSol_array;
•       X = X' ; update History;
•       LtLt_length=LtLt_length+ X; /* X is Tabu*/
•       If no improvement of F(X*) during Nb-local
   iterations go to 10, Else go to 4 ;
•     Intensification(X*local, X* );
 Diversification(History, X);
```

EXPLANATION OF THE ALGORITHM IN SIMPLE LANGUAGE:

In this algorithm, Xinit is initialized as the initial solution present with each slave processor. LtLt_length will be initialized as a null array, in which new best tabu solutions found using the local searches will be added.

X' denotes the current solution and X* denotes the best solution to a search.

Now, the first For-loop contains the iterating of i from 0 to Nb_div, which will make it search all the areas for the solution even if it is a lesser promising area.

Inside this For-loop, there will be another For-loop which contains the iterating of j from 0 to Nb_int, which will make it go to the best solutions locally and the other neighbours around the current solution.

Now, we will initially set the best local solution X*local equal to the initial value X.

After searching through solution X, next we will go to X’, and keep on dropping and adding simultaneously, the solution which are less yielding, and the solutions which are of higher importance respectively, by setting the components to 1 or 0 depending on the place which is worth visiting agin or not.

Now,

```
    If F(X’)> F(X ) then X*=X’ ;X*local =X’;
    Else If F(X’)>F(X*local ) Then X*local =X’;
```

Now, using these if- else cases, if X’ is a part of the best solution of the local searches, then it will be added into the BestSol_array.

As X is the actual solution, we will equate X=X’ and increment the value of LtLt_length by writing the following line: “ LtLt_length=LtLt_length+ X; “.

Else, if there would be no improvement in the value of the best solution X*, we will go to the 11th line of the algorithm and iterate the j loop again , else we will return to the line 4, in order to repeat the steps on th same local area unless we find the best solution in that particular area.

**LOCAL SEARCH:**

Local (neighborhood) searches take a potential solution to a problem and check its immediate neighbors (that is, solutions that are similar except for one or two minor details) in the hope of finding an improved solution. Local search methods have a tendency to become stuck in suboptimal regions or on plateaus where many solutions are equally fit. 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). In addition, prohibitions (henceforth the term tabu) are introduced to discourage the search from coming back to previously-visited solutions. The implementation of tabu search uses memory structures that describe the visited solutions or user-provided sets of rules. 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. Local search consists of two ways:

**ADD:**

Here one or several components fixed at 0 are chosen and set to 1. The selected component must not be Tabu except if the aspiration criteria is realized. Adding object to the knapsack is realized until no object can be added. The intensification and diversification phases complete the local search. The role of the intensification is to drive the search towards interesting region (promising region) by taking into account characteristics found in high quality solutions in the selection of future solutions to visit. The diversification phase, on the other hand, forces the search to explore regions containing unvisited solutions.

**DROP:**

This step selects Nb_drop components fixed at 1 in the actual solution X and resets these components to 0. As in, when tabu search makes a local search and either the same solutions are coming again and again, or the area where tabu search is being done is not giving a promising solution, then the drop phase will reset these components to 0, so that the search is not performed there again.

**INTENSIFICATION:**

The idea behind the concept of search intensification is that, as an intelligent human being would probably do, one should explore more thoroughly the portions of the search space that seem "promising" in order to make sure that the best solutions in these areas are indeed found. From time to time, one would thus stop the normal searching process to perform an intensification phase. In general, intensification is based on which one records the number of consecutive iterations that various "solution components" have been present in the current solution without interruption. A typical approach to intensification is to restart the search from the best currently known solution and to "freeze" (fix) in it the components that seem more attractive. In intensification, one could therefore allow more complex insertion moves or switch to an ejection chain neighborhood structure. Also, if Add/Drop moves were used, Swap moves could be added to the neighborhood structure.

**DIVERSIFICATION:**

One of the main problems of all methods based on Local Search approaches, and this includes TS in spite of the beneficial impact of tabus, is that they tend to be too "local" (as their name implies), i.e., they tend to spend most, if not all, of their time in a restricted portion of the search space. The negative consequence of this fact is that, although good solutions may be obtained, one may fail to explore the most interesting parts of the search space and thus end up with solutions that are still pretty far from the optimal ones. Diversification is an algorithmic mechanism that tries to reduce this problem by forcing the search into previously unexplored areas of the search space. It is usually based on some form of long-term memory of the search, in which one records the total number of iterations (since the beginning of the search) that various “solution components” have been present in the current solution or have been involved in the selected moves. In cases where it is possible to identify useful “regions” of the search space, the frequency memory can be refined to track the number of iterations spent in these different regions.

## THE MASTER SLAVE PROCESS:

The master process executes an iterative program. At each (search) iteration, P slave processes are launched. To start a search, the master process must give to each slave a starting solution (an initial solution) and a search strategy (synchronous centralized communication scheme). From a functional point of view, the master process is composed of three procedures and uses one data structure. Those procedures are :

the main procedure, the strategy generation procedure (SGP) and the initial solution generation procedure (ISP). The data structure used by the master process is an array of P entries. The entry i corresponds to informations given to or by the slave processor i and contains four items: -

- The search strategy (three values) (St)
- The initial solution used by the slave (Si)
- The B best solutions found by the slave i (best )
- The score of the slave i (score)

**THE MASTER PROCESS ALGORITHM:**

```
•   Procedure Master Process(P, Nb_search_it)
•     Read and send to slaves problem data
•         For i=1 to Nb_search_it do
•             Call SGP(P, Data_struc,  ) and ISP(P, Data_struc)
•             Send Initial solutions and strategies to slaves
•             Receive from each slave its B best solutions
```

//

**FINAL CODE FOR TABU SEARCH**

```
from copy import deepcopy


class TabuList(list):


    def __init__(self, size=3):
        self.size = size
        super(TabuList, self).__init__()


    def append(self, element):
        if len(self) == self.size:
            self.pop(0)
            return super(TabuList, self).append(element)
        return super(TabuList, self).append(element)

    def __contains__(self, move):
        for i in range(len(self)):
            if move == self[i]:
                return True
        return False



class TabuSearch(object):

    def __init__(self, max_iter=2):
        self.iter_counter = 0
        self.iter_better = 0
        self.max_iter = max_iter

    def __call__(self, neighborhood_function, knapsack):
        solutions = neighborhood_function(knapsack)
        sorted_moves = self.sort_moves(solutions)
```

```
                [sorted_moves.remove(tabu.reverse()) for tabu in
knapsack.tabu_list if tabu.reverse() in sorted_moves]

    best_move = None

    best_solution = knapsack.value

    best_solution_moves = deepcopy(knapsack.moves_made)

    best_solution_items = deepcopy(knapsack.items)

    tabu_ended = False


    while self.iter_counter - self.iter_better < self.max_iter:

      self.iter_counter += 1

      if not len(sorted_moves) == 0:  # If the tabu list not eliminated
all from neighbors

              candidate_move = sorted_moves.pop(0) # Get the best
possible move

                          actual_solution = knapsack.value +
candidate_move.movement_avaliation

        knapsack.execute_movement(candidate_move)

        knapsack.tabu_list.append(candidate_move.reverse())

        if actual_solution > best_solution:

          print "Current iter %d, actual solution %d, better solution
found in %d with %d" % (self.iter_counter, actual_solution,
self.iter_better, best_solution)

          best_solution = actual_solution

                                      best_solution_moves =
deepcopy(knapsack.moves_made)
```

```
                best_solution_items = deepcopy(knapsack.items)
                self.iter_better = self.iter_counter
            else: # If you eliminated all neighbors
                # Find the taboo with better improvement
                # if len(knapsack.tabu_list) == 0:
                    # return False
                best_tabu = reduce(lambda x, y: x if x.movement_avaliation > y.movement_avaliation else y, knapsack.tabu_list)
                if best_tabu.movement_avaliation > 0: # If it present a real improvement in the current solution
                    actual_solution = knapsack.value + best_tabu.movement_avaliation
                    if actual_solution > best_solution:
                        print "[TABU] Current iter %d, actual solution %d, better solution found in %d with %d" % (self.iter_counter, actual_solution, self.iter_better, best_solution)
                        best_solution = actual_solution
                        best_solution_moves = deepcopy(knapsack.moves_made)
                        best_solution_items = deepcopy(knapsack.items)
                        self.iter_better = self.iter_counter
                    knapsack.execute_movement(best_tabu)
            solutions = neighborhood_function(knapsack)
            sorted_moves = self.sort_moves(solutions)
```

```
                [sorted_moves.remove(tabu.reverse()) for tabu in knapsack.tabu_list if tabu.reverse() in sorted_moves]
            # print "Current iter %d, actual solution %d, better solution found in %d with %d" % (self.iter_counter, actual_solution, self.iter_better, best_solution)


        print "Better solution found in %d with %d" % (self.iter_better, best_solution)
        print knapsack.value
        print best_solution
        knapsack.value = best_solution
        knapsack.items = best_solution_items
        knapsack.moves_made = best_solution_moves


        print 'Script ran with tabu search using a max of %d iterations and a tabu list with size %d.' % (self.max_iter, knapsack.tabu_list.size)
        if not tabu_ended:
            print 'Ended by iteration limit.'
        return False


    def sort_moves(self, moves):
        return sorted(moves, key=lambda x: x.movement_avaliation, reverse=True)
```

**// FINAL CODE FOR LOCAL SEARCH:**

```
def best_improving(solutions, knapsack):
    sorted_solutions = sorted(solutions, key=lambda move: move.movement_avaliation, reverse=True)
    if len(sorted_solutions) == 0:
        return False
    if any(map(lambda solution: solution.movement_avaliation > 0, sorted_solutions)):
        return sorted_solutions[0]
    return False


def first_improving(solutions, knapsack):
    for solution in solutions:
        if solution.movement_avaliation > 0:
            return solution
    return False
```

**// FINAL CODE FOR NEIGHBOURHOD SEARCH:**

```
from knapsack import Movement
from random import choice, shuffle
from copy import deepcopy


def add_and_or_remove_neighborhood(knapsack):
    neighborhood = []
    improving = []
    shuffle(knapsack.items)
    for item in knapsack.items:
        if knapsack.can_add_item(item):
            movement = Movement(add_items=[item,])
            neighborhood.append(movement)
        else:
            weight_to_lose = item.weight
            volume_to_lose = item.volume
            items = deepcopy(knapsack.items)
            to_remove = choice(items)
            items.remove(to_remove)
            while not knapsack.can_swap(to_remove, item):
                print to_remove
                print item
                print '-'*80
                if len(items) == 0:
                    break
                to_remove = choice(items)
                items.remove(to_remove)
            else:
                                movement = Movement(add_items=[item,],
remove_items=[to_remove,])
```

```
            neighborhood.append(movement)
        print neighborhood
    return neighborhood



def all_neighborhood(knapsack):
    neighborhood = []
    improving_solutions = []
    shuffle(knapsack.all_items)
    for item in knapsack.all_items:
        i = 0
        actual_value = knapsack.value
        shuffle(knapsack.items)

        to_remove = []
        volume_to_lose = item.volume
        weight_to_lose = item.weight
        while volume_to_lose > 0 and weight_to_lose > 0:
            solution_item = choice(knapsack.items)
            to_remove.append(solution_item)
            weight_to_lose -= solution_item.weight
            volume_to_lose -= solution_item.volume
```

```
      i += 1
      movement = Movement(add_items=[item,], remove_items=to_remove)
      neighborhood.append(movement)

    for solution_item in knapsack.items:
      if knapsack.can_add_item(item):
        movement = Movement(add_items=[item,])
        neighborhood.append(movement)
      elif knapsack.can_swap(solution_item, item):
        new_value = knapsack.evaluate_swap(solution_item, item)
        movement = Movement(add_items=[item,], remove_items=[solution_item,])
        neighborhood.append(movement)
      else:
        continue
  return neighborhood

def first_improving_neighborhood(knapsack):
  neighborhood = []
  improving_solution = None
  for item in knapsack.sorted_items(knapsack.all_items):
    actual_value = knapsack.value
```

```
        for solution_item in knapsack.sorted_items(knapsack.items):
            if knapsack.can_swap(solution_item, item):
                new_value = knapsack.evaluate_swap(solution_item, item)
                movement = Movement(add_items=[item,], remove_items=[solution_item,])
                if new_value > knapsack.value:
                    improving_solution = movement
                    neighborhood.append(movement)
                    return neighborhood
                neighborhood.append(movement)
            else:
                pass
    return neighborhood
```

| WEEK | WORK DONE |
|---|---|
| | |

| | |
|---|---|
| Date:16-4-2016 | Code regarding the project that is multidimensional knapsack using tabu |
| Date:23-4-2016 | Code regarding the project that is multidimensional knapsack using tabu |
| Date:30-4-2016 | Final touch up on code and its debugging. |

**REFERENCES:**

[1] Battiti.R and G.Tecchiolli. The reactive tabu search. ORSA Jour. on Comp., 6(2), 1994.

[2] Chakrapani.j and J.Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. Annals of Op. res., (41), 1993.

[3] Demmeyer.F and S.Voss. Dynamic tabu list managementusing the reverse elimination method. Annals of OR, (41):31–46, 1993.

[4] Fre´ville.A and G.Plateau. Hard 0-1 test problems for size reduction methods. Investigacion Operativa, (1):251–270, 1990.

[5] Glover.F. Tabu search part 1. ORSA Journal on Computing, 1(3), 1989.

[6] Glover.F andG.Kochenberger. Critical event tabu search for multidimensional knapsack problems. In Osman and Kelly, editors, Meta-Heuristics : Theory and applications. Kluwer Academic Pub, 1996.

[7] Hanafi.S and A.Freville. An efficient tabu search approach for the 0-1mkp. To appear in European Journal ofOR, 1996.

[8] Martello.S and P.Toth. Knapsack Problems:Algorithms and computer implementation. Wiley & Sons, 1990.

[9] Niar.S and A.Freville. Parallel resolution of the 0-1 MKP.

Technical Report 1997 01, LIMAV University of Valenciennes,

1997.

[10] Taillard.E. Parallel iterative searchmethods for vehicle routing

problems. Networks, (23):661–673, 1993.

[11] Toulouse.M, T.G.Crainic, and M.Gendreau. Communication

issues in designing cooperative multithread parallel

searches. In Osman and Kelly, editors, Meta-Heuristics :

Theory and applications. Kluwer Academic Pub, 1996