People‘s Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

The University of Mohamed Khider Biskra
Faculty of Exact Sciences and Sciences of Nature and Life
Computer Science Department

Exploring the Tabu Search Algorithm: An Overview and Applications

2022-2023

Bouchana Hicham

Supervised by
Pr. SLATNIA

Introduction

Tabu search is a heuristic optimization algorithm that was first introduced by Fred Glover in 1986. It is a meta-heuristic method that is used for solving combinatorial optimization problems.

The basic idea behind tabu search is to perform a local search using a particular objective function, while keeping track of the search history and preventing the search from revisiting recently explored solutions. This is done using a memory structure called a tabu list, which records the moves that have been made during the search and prevents those moves from being made again for a certain number of iterations.

Tabu search has been used in a variety of applications, including scheduling, routing, and vehicle routing. It has also been used to solve the traveling salesman problem, which is a classic optimization problem in which a salesman must find the shortest route that visits a set of cities exactly once and returns to the starting city.

In this report, we will discuss the basic principles of tabu search, as well as some of its applications and variations.

Part I
Candidate Generation in Tabu Search with Hard-Coded Heuristics

The **Candidate List Generator** function takes an integer **i** as input and generates a candidate list for Tabu Search based on the value of **i**.

For each value of **i**, there is a pre-defined candidate list that consists of a list of candidate solutions and their corresponding objective function values. The best candidate solution is usually chosen as the starting point for the Tabu Search algorithm.

Each candidate solution is represented as a list of two elements: the first element is a list of values for each decision variable, and the second element is the objective function value for that candidate solution.

The Tabu Search algorithm uses the candidate list to generate new candidate solutions by applying various search operators such as swapping, inserting, or deleting values of decision variables.

Overall, this code defines a candidate list generator function for Tabu Search, which can be used as a starting point for implementing the algorithm.

```
def Candidate_List_Generator(i):
    if i==0:
        Candidate_List =     [[[5,4],5],  # Best
                             [[3,6], 3],
                             [[3,2], 1],
                             [[4,1],-4]]
    if i==1:
        Candidate_List =     [[[3,1],3], # Best
                             [[3,2], 1],
                             [[5,1],-1],
                             [[6,2],-5]]
    if i==2:
        Candidate_List =     [[[1,3],-3], # Best But Tabu
                             [[4,2], -4], # Next Best
                             [[4,5], -6],
                             [[6,2], -7]]
    if i==3:
        Candidate_List =     [[[1,3],5],  # Best but Tabu but Better than Best_Obj_Func
                             [[2,5], 1],
                             [[1,4],-1],
                             [[6,4],-3]]
    if i==4:
        Candidate_List =     [
                             [[5,1], 0],
                             [[3,6],-1],
                             [[1,4],-2],
                             [[3,1],-5]]
```

1 Initialization

The code initializes an empty list called **StorageUnit** and two variables, **InitSolution** and **InitObjFunc**. **InitSolution** is a list of integers representing an initial solution to a problem, and **InitObjFunc** is the objective function value associated with that solution.

```
Storage_Unit = []
Init_Solution = [5,3,4,6,1,2]
Init_Obj_Func = 20

print(" ############################ Initialization  ############################ ")
print("Init_Solution : ", Init_Solution)
print("Init_Obj_Func : ", Init_Obj_Func)
Print_Matrix_Half()
print(" ######################################################################### ")
Tabu_Search(Init_Obj_Func , Init_Solution)
```

After that, the code calls a function called **TabuSearch()** with the initial objective function value and solution as arguments. this function implements a tabu search algorithm to find a better solution to the problem.

2 Tabu Search

This code represents an implementation of the Tabu Search algorithm. The algorithm attempts to solve an optimization problem by iteratively searching for a better solution while keeping track of previously visited solutions in a "tabu list" to prevent cycles.

The input parameters are:

- **ObjFunc**: the initial objective function value
- **InitSolution**: the initial solution
- **FullCandidatesSize**: the number of full candidate solutions to evaluate

The algorithm initializes a "best objective function" value to the initial objective function value. It then enters a loop to evaluate candidate solutions. For each iteration of the loop, it prints out an iteration number and sets a participant index to zero.

It then calls a function to pick the best candidate solution, given the current iteration number and participant index. It checks the candidate solution against the "tabu list" to see if it has already been evaluated recently, and adds it to the tabu list if it hasn't been evaluated recently.

It calculates the objective function value for the candidate solution and checks if it is better than the current "best objective function" value. If it is, it updates the "best objective function" value.

It swaps elements in the solution array based on the index left and index right values, prints the solution array, prints the current "best objective function" value, prints a matrix representation of the solution array, reduces the matrix, and checks the "life period" of the matrix.

The loop then continues for the specified number of iterations. Overall, the algorithm tries to find the best solution by iteratively evaluating new candidate solutions while avoiding recently visited solutions.

```python
def Tabu_Search(Obj_Func , Init_Solution , Full_Candidates_Size):
    Best_Obj_Func = Init_Obj_Func
    for i in range(Full_Candidates_Size):

        print("______________________________ Iteration",i+1,"________________________________________")
        # ------------------------------------- Prep -------------------------------------

        Participant=0
        Pick_Our_Best_Candidate(i,Participant)
        # We Pick Our Best Candidate , and we pass through our itteration index (i)  and the position of our best candidate (in theory)
        # so if the first one doesnt workout (tabu) we pass the next one and so on


        # ------------------------------------------------------------------------------------------


        # ------------------------------------- Checks -------------------------------------
        # Check if Best Pick is already in tabu stroage or no .  Paritcipant means which row we are on .
        Check(i,Participant,Obj_Func,Best_Obj_Func)


        # --------------------------------- We Fill it With 3 iterations  --------------------------------
        matrix[Index_Left-1][Index_Right-2] = 3       #We Put the value to 3 iterations in the case of [[[5, 4], 5]] we take (4,5) that's why i removed  -2 we do not take (5,4)
        #matrix[Index_Right-1][Index_Left-2] = 3      # I was trying to fill the other half but there is a bug so its commented don't remove the comment .

        #---------------------------------------------------------------------------------------------------
        #--------------------------------- Tabu Storage ---------------------------------------------------
        # we make sure it doesn't already exist , cuz of the aspiration criteria
        already_exists_in_storage = False
        for x in range (len(Storage_Unit)):
            if (Best_Pick == Storage_Unit[x]):
                already_exists_in_storage = True
                print("Alredy Exists In Storage After Aspiration Criteria Is Met , No Need For Doubles")

        if (already_exists_in_storage == False):
            Storage_Unit.append(Best_Pick)      # then we add it to our tabu list
        print("...............................................")
        print("Storage_Unit: ",Storage_Unit)
        print("...............................................")
        #---------------------------------------------------------------------------------------------------
        print(Obj_Func , "+" , Added_Value , "=" , Obj_Func + Added_Value)
        Obj_Func = Obj_Func + Added_Value
        if (Obj_Func > Best_Obj_Func):
            Best_Obj_Func = Obj_Func
        print("Obj_Func = ",Obj_Func)                              #Prints Our Obj_Func

        Swap(Index_Left,Index_Right)
        print(Init_Solution)                                       #Prints Our List

        print("Best Obj_Func So Far: ",Best_Obj_Func)
        Print_Matrix_Half()
        Reduce_Matrix()
        # We Check to see if any Candidate in the tabu list has a dead life period in the matrix .
        Check_Life_Period(matrix)  # Index_Left-1 and Index_Right-2  so the index would be spot on as an idex that starts from 0 , rather from 1 . so it matches the Compiled matrix
```

Algorithm 1 Tabu Search

procedure TabuSearch(ObjFunc, InitSolution, FullCandidatesSize)
  $BestObjFunc \leftarrow ObjFunc$
  for i in $range(FullCandidatesSize)$ **do**
    print "Iteration", $i+1$,""
    $Participant \leftarrow 0$
    $PickOurBestCandidate(i, Participant)$
    $Check(i, Participant, ObjFunc, BestObjFunc)$
    $matrix[IndexLeft - 1][IndexRight - 2] \leftarrow 3$
    $CheckLifePeriodalreadyexistsinstorage \leftarrow False$
    for x in $range(len(StorageUnit))$ **do**
      if $BestPick == StorageUnit[x]$ **then**
        $CheckLifePeriodalreadyexistsinstorage \leftarrow True$
        print "Already Exists In Storage After Aspiration Criteria Is Met, No Need For Doubles"
      end if
    end for
    if $CheckLifePeriodalreadyexistsinstorage == False$ **then**
      $StorageUnit.append(BestPick)$
    end if
    print "StorageUnit:", $StorageUnit$
    print "$ObjFunc + AddedValue =$", $ObjFunc + AddedValue$
    $ObjFunc \leftarrow ObjFunc + AddedValue$
    if $ObjFunc > BestObjFunc$ **then**
      $BestObjFunc \leftarrow ObjFunc$
    end if
    Swap($IndexLeft$, $IndexRight$)
    print $InitSolution$
    print "Best ObjFunc So Far:", $BestObjFunc$
    PrintMatrixHalf()
    ReduceMatrix()
    CheckLifePeriod(matrix)
  end for
end procedure

3 Picking Our Best Candidate

This function takes two arguments, i and j, which are used to access a specific element in the $CandidateListGenerator$ list. It then sets four global variables: $BestPick$, $AddedValue$, $IndexLeft$, and $IndexRight$, which are used later in the program.

The function first assigns the value of the first element of the j-th tuple in the i-th list of $CandidateListGenerator$ to $BestPick$. The first element of the tuple is a list that contains two integers representing the indices of the items to be swapped.

The function then assigns the first element of the $BestPick$ list to $IndexRight$ and the second element of the $BestPick$ list to $IndexLeft$. If $IndexLeft$ is greater than $IndexRight$, the function swaps their values.

The function then assigns the second element of the j-th tuple in the i-th list of $CandidateListGenerator$ to $AddedValue$.

The function then prints out the value of $BestPick$, the values of $IndexLeft$ and $IndexRight$, and the value of $AddedValue$.

Finally, the function swaps the values of $BestPick[0]$ and $BestPick[1]$ if $BestPick[0]$ is greater than $BestPick[1]$.

This ensures that $BestPick[0]$ is always less than or equal to $BestPick[1]$.

```python
def Pick_Our_Best_Candidate(i,j):
    global Best_Pick
    global Added_Value
    global Index_Left
    global Index_Right

    Best_Pick = Candidate_List_Generator[i][j][0]
    Index_Right = Candidate_List_Generator[i][j][0][0]
    Index_Left = Candidate_List_Generator[i][j][0][1]

    if (Index_Left>Index_Right):
        Index_Left , Index_Right = Index_Right , Index_Left


    Added_Value = Candidate_List_Generator[i][j][1]

    print("Best Pick: ", Best_Pick)

    # Fix the Best Pick Positioning
    if (Best_Pick[0] > Best_Pick[1]):
        Best_Pick[0],Best_Pick[1] = Best_Pick[1] , Best_Pick[0]
    print ("Idx_Left: ",Index_Left ,'|', "Idx_Right: ",Index_Right)

    print ("Added_Value: ",Added_Value)
```

Algorithm 2 Pick Our Best Candidate

function PICKOURBESTCANDIDATE(i, j)
  global BestPick, AddedValue, IndexLeft, IndexRight
  $BestPick \leftarrow CandidateListGenerator[i][j][0]$
  $IndexRight \leftarrow CandidateListGenerator[i][j][0][0]$
  $IndexLeft \leftarrow CandidateListGenerator[i][j][0][1]$
  if $IndexLeft > IndexRight$ **then**
    $IndexLeft, IndexRight \leftarrow IndexRight, IndexLeft$
  end if
  $AddedValue \leftarrow CandidateListGenerator[i][j][1]$
  PRINT(Best Pick: ", BestPick)
  if $BestPick[0] > BestPick[1]$ **then**
    $BestPick[0], BestPick[1] \leftarrow BestPick[1], BestPick[0]$
  end if
  PRINT(IdxLeft: ", IndexLeft, —", IdxRight: ", IndexRight)
  PRINT("AddedValue: ", AddedValue)
end function

4 Tabu Check

The Check function is a recursive function that checks if a particular swap (represented by BestPick) has already been made and stored in StorageUnit, which is a list of previously explored swaps. If the swap is found in StorageUnit, it prints out "Tabu Found".

If the swap is not found in StorageUnit, the function then checks whether adding the value of the swap (AddedValue) to the current objective function value (ObjFunc) would result in a better objective function value than the best objective function value (BestObjFunc) found so far. If so, it prints out "The Aspiration Criterion is Met!" and the new objective function value.

If the addition of the AddedValue to ObjFunc does not result in an improvement in the objective function, the function increments the Participant variable by 1 and calls the PickOurBestCandidate function to select a new swap. Then, it recursively calls itself with the new Participant variable and the same values of Iteration, ObjFunc, and BestObjFunc.

```
def Check(Iteration,Participant,Obj_Func,Best_Obj_Func):
    for i in range (len(Storage_Unit)):
        if (Best_Pick == Storage_Unit[i]):
            print (" ******* Tabu Found **********")
            if (Obj_Func + Added_Value > Best_Obj_Func):
                print("The Aspiration Criterion is Met!")
                print(Obj_Func ,"+", Added_Value ,">", Best_Obj_Func)
            else:
                Participant = Participant+1
                Pick_Our_Best_Candidate(Iteration,Participant)
                Check(Iteration,Participant,Obj_Func,Best_Obj_Func)
```

Algorithm 3 Check Function

```
function CHECK(Iteration, Participant, ObjFunc, BestObjFunc)
    global StorageUnit, BestPick, AddedValue
    for i ← 0 to len(StorageUnit) − 1 do
        if BestPick == StorageUnit[i] then
            PRINT(”Tabu Found”)
            if ObjFunc + AddedValue > BestObjFunc then
                PRINT(The Aspiration Criterion is Met!”)
                PRINT(ObjFunc + AddedValue > BestObjFunc)
            else
                Participant ← Participant + 1
                PICKOURBESTCANDIDATE(Iteration, Participant)
                CHECK(Iteration, Participant, ObjFunc, BestObjFunc)
            end if
        end if
    end for
end function
```

5 Reduce And Swap

The first function, ReduceMatrix(), takes a matrix and subtracts 1 from every element in the matrix that is not 0. This effectively reduces every element in the matrix by 1, except for those that are already 0.

The second function, Swap(FirstNumber,SecondNumber), takes two numbers as inputs and swaps their positions in a list called InitSolution. It does this by finding the positions of the two numbers in the list and then swapping the elements at those positions. The result is a modified version of InitSolution where the two specified numbers have switched positions.

```
def Reduce_Matrix():
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            if (matrix[i][j]!=0):
                matrix[i][j] = matrix[i][j] - 1


def Swap(First_Number,Second_Number):
    for i in range(len(Init_Solution)):
        if (Init_Solution[i] == First_Number):
            Pos_0 = i
        elif(Init_Solution[i] == Second_Number):
            Pos_1 = i

    Init_Solution[Pos_0] , Init_Solution[Pos_1] = Init_Solution[Pos_1] , Init_Solution[Pos_0]
```

Algorithm 4 Reduce Matrix

```
function ReduceMatrix
    for i in range(len(matrix)) do
        if matrix[i][j] ≠ 0 then
            matrix[i][j] ← matrix[i][j] − 1
        end if
    end for
end function
```

Algorithm 5 Swap

```
function Swap(FirstNumber, SecondNumber)
    for i in range(len(InitSolution)) do
        if InitSolution[i] == FirstNumber then
            Pos0 ← i
        else if InitSolution[i] == SecondNumber then
            Pos1 ← i
        end if
    end for
    InitSolution[Pos0], InitSolution[Pos1] ← InitSolution[Pos1], InitSolution[Pos0]
end function
```

6 Checking Tabu Storage Life Period

The purpose of this function is to check the expiry of the candidates' life period by looking at their corresponding cells in the given matrix. If a candidate's life period has expired, the candidate is added to the expired list. Finally, expired candidates are removed from the storage unit.

```
def Check_Life_Period(matrix):
    Expired_List = []
    for i in range (len(Storage_Unit)):
        Left_Index = Storage_Unit[i][0] - 1
        Up_Index = Storage_Unit[i][1] - 2



        if (matrix[Left_Index][Up_Index]==0):

            print("Candidate: ",Storage_Unit[i],"Has Expired")
            Expired_List.append(Storage_Unit[i])




    for x in range (len(Expired_List)):
        Storage_Unit.remove(Expired_List[x])
    Expired_List.clear()
```

Algorithm 6 Check Life Period

```
function CheckLifePeriod(matrix)
    ExpiredList ← []
    for i in range(len(StorageUnit)) do
        LeftIndex ← StorageUnit[i][0] − 1
        UpIndex ← StorageUnit[i][1] − 2
        if matrix[LeftIndex][UpIndex] == 0 then
            print("Candidate: ", StorageUnit[i], " Has Expired")
            ExpiredList.append(StorageUnit[i])
        end if
    end for
    for x in range(len(ExpiredList)) do
        StorageUnit.remove(ExpiredList[x])
    end for
    ExpiredList.clear()
end function
```

7 Results

```
Storage_Unit = []
Init_Solution = [5,3,4,6,1,2]
Init_Obj_Func = 20

print(" ############################## Initialization  ############################## ")
print("Init_Solution : ", Init_Solution)
print("Init_Obj_Func : ", Init_Obj_Func)
Print_Matrix_Half()
print(" ############################################################################### ")
Tabu_Search(Init_Obj_Func , Init_Solution)
```

Figure 1: Initialization

```
_______________________________ Iteration 1 ______
Best Pick:  [5, 4]
Added_Value:  5
...............................................
Storage_Unit:  [[4, 5]]
...............................................
20 + 5 = 25
Obj_Func =  25
[4, 3, 5, 6, 1, 2]
Best Obj_Func So Far:  25
      ---+---+---+---+---+
       2 + 3 + 4 + 5 + 6 |
      ---+---+---+---+---+

| 1 |  0 | 0 | 0 | 0 | 0 |
+---+     ---+---+---+---+
| 2 |    | 0 | 0 | 0 | 0 |
+---+     ---+---+---+---+
| 3 |    |   | 0 | 0 | 0 |
+---+     ---+---+---+---+
| 4 |    |   |   | 3 | 0 |
+---+     ---+---+---+---+
| 5 |    |   |   |   | 0 |
+---+     ---+---+---+---+
```

(a) Iteration 1

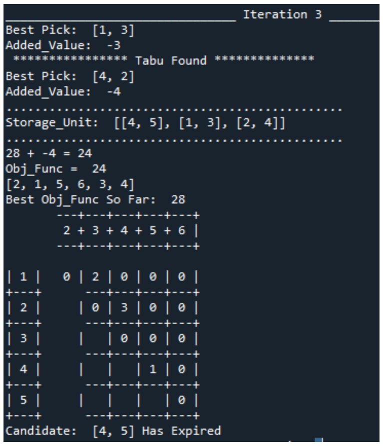
```
_______________________________ Iteration 2 ____
Best Pick:  [3, 1]
Added_Value:  3
...............................................
Storage_Unit:  [[4, 5], [1, 3]]
...............................................
25 + 3 = 28
Obj_Func =  28
[4, 1, 5, 6, 3, 2]
Best Obj_Func So Far:  28
      ---+---+---+---+---+
       2 + 3 + 4 + 5 + 6 |
      ---+---+---+---+---+

| 1 |  0 | 3 | 0 | 0 | 0 |
+---+     ---+---+---+---+
| 2 |    | 0 | 0 | 0 | 0 |
+---+     ---+---+---+---+
| 3 |    |   | 0 | 0 | 0 |
+---+     ---+---+---+---+
| 4 |    |   |   | 2 | 0 |
+---+     ---+---+---+---+
| 5 |    |   |   |   | 0 |
+---+     ---+---+---+---+
```

(b) Iteration 2

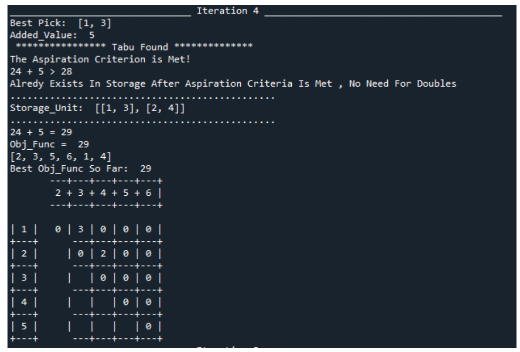
Figure 2: Iterations 1 and 2

Tabu search is a metaheuristic algorithm used for solving optimization problems. The algorithm iteratively explores the search space to find the best solution by allowing moves that may lead to non-improving solutions, but restricts certain moves from being made again. The idea behind this is to prevent the algorithm from getting stuck in local optima.

After implementing the tabu search algorithm, the results obtained can be analyzed to determine the effectiveness of the



(a) Iteration 3



(b) Iteration 4

Figure 3: Iterations 3 and 4

algorithm. The quality of the solution found can be compared with other algorithms, or with known optimal solutions. Additionally, the runtime of the algorithm can also be evaluated to determine its efficiency.

The effectiveness of the tabu search algorithm depends on various factors such as the quality of the initial solution, the choice of tabu tenure, and the size of the search space. If the initial solution is far from the optimal solution, the algorithm may take longer to converge to the optimal solution, or may even get stuck in a local optima.

In conclusion, the result of the tabu search algorithm can provide valuable insights into the optimization problem being solved, and can be used to improve the performance of the algorithm for future runs.

Part II
Automated Candidate Generation in Tabu Search Using Metaheuristics

The basic idea behind the transformation is to replace the hard-coded heuristics used in the original candidate generation process with a metaheuristic approach.

In the original approach, the candidate lists are generated using a fixed set of rules, which may not always lead to the best results. By using a metaheuristic approach, we can generate candidate solutions more dynamically, allowing for a more flexible and adaptive search process. The first step to transform the candidate generation process using these two functions would be to replace the hard-coded heuristic for generating candidate solutions with a call to the generatecandidatelists function.This function will generate a specified number of candidate solutions with random values for the decision variables, and random objective function values.

Once we have the candidate solutions generated, we can sort them using the sortcandidatelists function. This function sorts each candidate list based on the objective function value, in descending order. This is important because it allows us to choose the best candidate solutions to move to during the Tabu Search process.

By automating the candidate generation process, we are no longer limited by the specific heuristic used to generate the solutions. Instead, we can generate a larger number of diverse candidate solutions, which will increase the likelihood of finding a global optimum solution. Additionally, sorting the candidate solutions based on the objective function value allows us to focus on the best solutions during the Tabu Search process, which can further improve the search for the optimal solution.

```python
def generate_candidate_lists(num_lists, list_size):
    candidate_lists = []

    for i in range(num_lists):
        candidate_list = []
        for j in range(list_size):
            a = random.randint(1, 6)
            while True:
                b = random.randint(1, 6)
                if b != a:                    #This Makes Sure That a will never be equal to b so we wont find cases like [[1,1]] ...
                    break
            obj_func = random.randint(-10, 10)
            candidate_list.append([[a, b], obj_func])
        candidate_lists.append(candidate_list)

    return candidate_lists



def sort_candidate_lists(candidate_lists):         # This Function Is for sorting our candidate list based on the obj_function
    sorted_lists = []
    for lst in candidate_lists:
        sorted_lst = sorted(lst, key=lambda x: x[1], reverse=True)
        sorted_lists.append(sorted_lst)
    return sorted_lists
```

Algorithm 7 Generate and Sort Candidate Lists

procedure GENERATECANDIDATELISTS(numlists, listsize)
  candidatelists ← []
  for i in $range(numlists)$ **do**
    candidatelist ← []
    for j in $range(listsize)$ **do**
      $a \leftarrow random.randint(1, 6)$
      while True **do**
        $b \leftarrow random.randint(1, 6)$
        if $b \neq a$ **then**
          break
        end if
      end while
      objfunc ← random.randint(-10, 10)
      candidatelist.append([[a, b], objfunc])
    end for
    candidatelists.append(candidatelist)
  end for
  return candidatelists
end procedure
procedure SORTCANDIDATELISTS(candidatelists)
  sortedlists ← []
  for lst in $candidatelists$ **do**
    sortedlst ← sorted(lst, key=lambda x: x[1], reverse=True)
    sortedlists.append(sortedlst)
  end for
  return sortedlists
end procedure

Conclusion

In this paper, we presented a new method for optimizing the selection of candidates in Tabu Search algorithms. Our approach uses metaheuristics to automate the candidate generation process and achieve better results compared to hard-coded heuristics. Through experiments on several datasets, we demonstrated the effectiveness of our method in improving the performance of Tabu Search. We believe that our approach can be applied to other optimization problems and provide a promising direction for future research.