8. Ant Colony Optimization

8.5 Implementing ACO Algorithms

Fall 2010

Instructor: Dr. Masoud Yaghini

Outline

- Data Structures
- The Algorithm
- Changes for Other ACO Algorithms
- References

Data Structures

Data Structures

- We mainly focus on **AS** and indicate, where appropriate, the necessary changes for implementing other ACO algorithms.
- Data Structures
  - Problem Representation
    - Intercity Distances
    - Nearest-Neighbor Lists
    - Pheromone Trails
    - Combining Pheromone and Heuristic Information
  - Representing Ants
    - Ant's Memory Storing (partial) Tours
    - Visited Cities

Main data structures

```
% Representation of problem data

integer dist[n][n]              % distance matrix
integer nn_list[n][nn]          % matrix with nearest neighbor lists of depth nn
real    pheromone[n][n]         % pheromone matrix
real    choice_info[n][n]       % combined pheromone and heuristic information

% Representation of ants

structure single_ant
begin
   integer tour_length          % the ant's tour length
   integer tour[n + 1]          % ant's memory storing (partial) tours
   integer visited[n]           % visited cities
end

single_ant ant[m]               % structure of type single_ant
```

Intercity Distances

- Often a symmetric TSP instance is given as the coordinates of a number of *n* points.
- In this case, one possibility would be to store the *x* and *y* coordinates of the cities in two arrays and then compute on the fly the distance between the cities as needed, however, this leads to a significant computational overhead.
- It is more reasonable to pre-compute all intercity distances and to store them in a symmetric distance matrix with n^2 entries, integer dist[n][n].

Intercity Distances

- For symmetric TSPs we only need to store **n(n-1)/2** distinct distances,
- It is more efficient to use an n^2 matrix to avoid performing additional operations to check whether, when accessing a generic distance $d(i, j)$, entry (i, j) or entry(j, i) of the matrix should be used.

Intercity Distances

- It is also important to know that, for historical reasons, in almost all the TSP literature, the distances are stored as integers.
- In fact, in old computers integer operations used to be much faster than operations on real numbers, so that by setting distances to be integers, much more efficient code could be obtained.

Nearest-Neighbor Lists

- In addition to the distance matrix, it is convenient to store for each city a list of its **nearest neighbors**.
- Let d_i be the list of the distances from a city i to all cities j, with $j = 1,\ldots, n$ and $i \# j$
- The nearest-neighbor list of a city i is obtained by sorting the list d_i according to nondecreasing distances, obtaining a sorted list d'_i , ties can be broken randomly.

Nearest-Neighbor Lists

- The position *r* of a city *j* in city *i*'s nearest-neighbor list nn_list[i] is the index of the distance d_{ij} in the sorted list d'_i
- nn_list[i][r] gives the identifier (index) of the *r*-th nearest city to city *i*
  - i.e., nn_list[i][r] = *j*
- You have to repeat a sorting algorithm over *n* - 1 cities for each city

Nearest-Neighbor Lists

- An enormous speedup is obtained for the solution construction in ACO algorithms.
- If the nearest-neighbor list is cut *nn* after a constant number *nn* of nearest neighbors, where typically *nn* is a small value ranging between 15 and 40.
- In this case, an ant located in city *i* chooses the next city among the *nn* nearest neighbors of *i*
- In case the ant has already visited all the nearest neighbors, then it makes its selection among the remaining cities

Nearest-Neighbor Lists

- It should be noted that the use of truncated nearest-neighbor lists can make it impossible to find the optimal solution.

Pheromone Trails

- In addition to the instance-related information, we also have to store for each connection (i, j) a number τ_{ij} corresponding to the pheromone trail associated with that connection.
- For symmetric TSPs this requires storing $n(n-1)/2$ distinct pheromone values, because we assume that $\tau_{ij} = \tau_{ji}$, for all (i, j)
- Again, as was the case for the distance matrix, it is more convenient to use some redundancy and to store the pheromones in a symmetric n^2 matrix.

Combining Pheromone and Heuristic Information

- When constructing a tour, an ant located on city *i* chooses the next city *j* with a probability which is proportional to the value of $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$.
- Because these very same values need to be computed by each of the *m* ants, computation times may be significantly reduced by using an additional matrix *choice_info*[*n*][*n*]
- Each entry *choice_info*[*i*][*j*] stores the value $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$.

Combining Pheromone and Heuristic Information

- Again, in the case of a symmetric TSP instance, only $n(n-1)/2$ values have to be computed, but it is convenient to store these values in a redundant way as in the case of the pheromone and the distance matrices.
- Additionally, one may store the $[\eta_{ij}]^{\beta}$ values in a further matrix heuristic to avoid recomputing these values after each iteration, because the heuristic information stays the same throughout the whole run of the algorithm

Combining Pheromone and Heuristic Information

- Finally, if some distances are zero, which is in fact the case for some of the benchmark instances in the TSPLIB, then one may set them to a very small positive value to avoid division by zero.

Representing Ants

- An ant is a simple computational agent which
  - **constructs a solution** to the problem at hand, and
  - may **deposit an amount of pheromone $\Delta\tau$** on the arcs it has traversed.
- To do so, an ant must be able to
  - (1) store the partial solution it has constructed so far,
  - (2) determine the feasible neighborhood at each city, and
  - (3) compute and store the objective function value of the solutions it generates.

Ant's memory storing (partial) tours

- The first requirement can be satisfied by storing the partial tour in a sufficiently large array.
- For the TSP we represent tours by arrays of length n + 1, integer tour[n + 1], where at position n + 1 the first city is repeated.
- This choice makes easier some of the other procedures like the computation of the tour length.

Visited Cities

- The knowledge of the partial tour at each step is sufficient to allow the ant to determine whether a city *j* is in its feasible neighborhood:
  - it is enough to scan the partial tour for the occurrence of city *j*.
  - If city j has not been visited yet, then it is member of the feasible neighborhood; otherwise it is not.
- Unfortunately, this simple way of determining the feasible neighborhood involves a high computational overhead.

Visited Cities

- The simplest way around this problem is to associate with each ant an additional array visited whose values are set to visited[j] = 1 if city j has already been visited by the ant, and to visited[j] = 0 otherwise.
- This array is updated by the ant while it builds a solution.
- The array visited, part of the data structure single_ant, is declared of type integer; however, to save memory, it could be declared of type Boolean

Tour Length

- Finally, the computation of the tour length, stored by the ant in the tour_length variable, can be done by summing the length of the n arcs in the ant's tour.

Overall Memory Requirement

- For representing all the necessary data for the problem we need:
  - four matrices of dimension n x n for representing **the distance matrix**, **the pheromone matrix**, **the heuristic information matrix**, and **the choice_info** matrix, and
  - a matrix of size n x nn for the **nearest-neighbor lists**.
  - two arrays of size (n +1) and n to store the **tour** and the **visited cities**
  - an integer for storing the **tour's length**.
  - a variable for representing each of the m ants
  - the best solution found so far
  - statistical information about the algorithm performance

The Algorithm

The Algorithm

- The main tasks to be considered in an ACO algorithm are:
  - the solution construction,
  - the management of the pheromone trails, and
  - the additional techniques such as local search.
- In addition, the data structures and parameters need to be initialized and some statistics about the run need to be maintained.
- In this section we give some details on how to implement the different procedures of AS in an efficient way.

The Algorithm

- A high-level view of the algorithm:

procedure ACOforTSP
  InitializeData
  while (**not** terminate) **do**
    ConstructSolutions
    LocalSearch
    UpdateStatistics
    UpdatePheromoneTrails
  end-while
end-procedure

Data Initialization

- **Data initialization:**
  - (1) the instance has to be read;
  - (2) the distance matrix has to be computed;
  - (3) the nearest-neighbor lists for all cities have to be computed;
  - (4) the pheromone matrix and the choice_info matrix have to be initialized;
  - (5) the ants have to be initialized;
  - (6) the algorithm's parameters must be initialized; and
  - (7) some variables that keep track of statistical information, such as the used CPU time, the number of iterations, or the best solution found so far, have to be initialized.

Data Initialization

- A possible organization of these tasks into several data initialization procedures is indicated in the figure:

```
procedure InitializeData
    ReadInstance
    ComputeDistances
    ComputeNearestNeighborLists
    ComputeChoiceInformation
    InitializeAnts
    InitializeParameters
    InitializeStatistics
end-procedure
```

Termination Condition

- The program stops if at least one **termination condition** applies. Possible termination conditions are:
  - (1) the algorithm has found a solution within a predefined distance from a lower bound on the optimal solution quality;
  - (2) a maximum number of tour constructions or a maximum number of algorithm iterations has been reached;
  - (3) a maximum CPU time has been spent; or
  - (4) the algorithm shows stagnation behavior.

Solution Construction

```
procedure ConstructSolutions
    for k = 1 to m do
        for i = 1 to n do
            ant[k].visited[i] ← false
        end-for
    end-for
    step ← 1
    for k = 1 to m do
        r ← random{1, . . . , n}
        ant[k].tour[step] ← r
        ant[k].visited[r] ← true
    end-for
```

Solution Construction

while $(step < n)$ **do**
    $step \leftarrow step + 1$
    for $k = 1$ **to** m **do**
        ASDecisionRule$(k, step)$
    end-for
end-while
for $k = 1$ **to** m **do**
    $ant[k].tour[n + 1] \leftarrow ant[k].tour[1]$
    $ant[k].tour\_length \leftarrow$ ComputeTourLength(k)
end-for
end-procedure

Solution Construction

- The solution construction requires the following phases.

1. First, the ants' memory must be emptied.
   - This is done in lines 1 to 5 of procedure ConstructSolutions by marking all cities as unvisited, that is, by setting all the entries of the array ants.visited to false for all the ants.
2. Second, each ant has to be assigned an initial city.
   - One possibility is to assign each ant a random initial city.
   - This is accomplished in lines 6 to 11 of the procedure.
   - The function random returns a random number chosen according to a uniform distribution over the set {1,…, n}.

Solution Construction

3. Next, each ant constructs a complete tour.
   - At each construction step the ants apply the AS action choice rule (line 12-17).
   - The procedure ASDecisionRule implements the action choice rule and takes as parameters the ant identifier and the current construction step index; this is discussed later in more detail.
4. Finally, the ants move back to the initial city and the tour length of each ant's tour is computed.
   - This is done in lines 18 to 21.
   - for the sake of simplicity, in the tour representation we repeat the identifier of the first city at position n + 1; this is done in line 19.

Solution Construction

- As stated above, the solution construction of all of the ants is synchronized in such a way that the ants build solutions in parallel.
- The same behavior can be obtained, for all **AS variants**, by ants that construct solutions sequentially, because the ants do not change the pheromone trails at construction time
- This is not the case for ACS, in which case the **sequential** and **parallel** implementations give different results.

Solution Construction

- In the action choice rule an ant located at city i probabilistically chooses to move to an unvisited city j based on the pheromone trails $[\tau_{ij}]^{\alpha}$ and the heuristic information $[\eta_{ij}]^{\beta}$ by see equation:

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}, \quad \text{if } j \in \mathcal{N}_i^k$$

ASDecisionRule Procedure

- AS without candidate lists: pseudo-code for the action choice rule.

```
procedure ASDecisionRule(k, i)
    input  k   % ant identifier
    input  i   % counter for construction step
    c ← ant[k].tour[i − 1]
    sum_probabilities = 0.0
    for j = 1 to n do
        if ant[k].visited[j] then
            selection_probability[j] ← 0.0
        else
            selection_probability[j] ← choice_info[c][j]
            sum_probabilities ← sum_probabilities + selection_probability[j]
        end-if
    end-for
```

ASDecisionRule Procedure

```
    r ← random[0, sum_probabilities]
    j ← 1
    p ← selection_probability[j]
    while (p < r) do
        j ← j + 1
        p ← p + selection_probability[j]
    end-while
    ant[k].tour[i] ← j
    ant[k].visited[j] ← true
end-procedure
```

ASDecisionRule Procedure

- The procedure works as follows:
- 1. First, the current city c of ant k is determined (line 1).
- 2. The probabilistic choice of the next city then works analogously to the **roulette wheel selection** procedure of evolutionary computation
  - each value choice_info[c][j] of a city j that ant k has not visited yet determines a slice on a circular roulette wheel, the size of the slice being proportional to the weight of the associated choice (lines 2–10).

Solution Construction

- 3. the wheel is spun and the city to which the marker points is chosen as the next city for ant k (lines 11–17).
- This is implemented by:
  - summing the weight of the various choices in the variable sum_probabilities,
  - drawing a uniformly distributed random number r from the interval [0, sum_probabilities],
  - going through the feasible choices until the sum is greater or equal to r.

Solution Construction

- 4. Finally, the ant is moved to the chosen city, which is marked as visited (lines 18 and 19).

Solution Construction

- When exploiting candidate lists, the procedure ASDecisionRule needs to be adapted, resulting in the procedure NeighborListASDecisionRule
- A first change is that when choosing the next city, one needs to identify the appropriate city index from the candidate list of the current city c.
- This results in changes of the maximum value of index j is changed from n to nn in line 3 and the test performed in line 4 is applied to the j-th nearest neighbor given by nn_list[c][j].

Solution Construction

- A second change is necessary to deal with the situation in which all the cities in the candidate list have already been visited by ant k.
- In this case, the variable sum_probabilities keeps its initial value 0.0 and one city out of those not in the candidate list is chosen.
- The procedure ChooseBestNext is used to identify the city with maximum value of $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$ as the next to move to.

Solution Construction

- AS with candidate lists: pseudo-code for the action choice rule.

```
procedure NeighborListASDecisionRule(k, i)
    input  k   % ant identifier
    input  i   % counter for construction step
    c ← ant[k].tour[i − 1]
    sum_probabilities ← 0.0
    for j = 1 to nn do
        if ant[k].visited[nn_list[c][j]] then
            selection_probability[j] ← 0.0
        else
            selection_probability[j] ← choice_info[c][nn_list[c][j]]
            sum_probabilities ← sum_probabilities + selection_probability[j]
        end-if
    end-for
```

Solution Construction

```
    if (sum_probabilities = 0.0) then
        ChooseBestNext(k, i)
    else
        r ← random[0, sum_probabilities]
        j ← 1
        p ← selection_probability[j]
        while (p < r) do
            j ← j + 1
            p ← p + selection_probability[j]
        end-while
        ant[k].tour[i] ← nn_list[c][j]
        ant[k].visited[nn_list[c][j]] ← true
    end-if
end-procedure
```

Solution Construction

```
procedure ChooseBestNext(k, i)
   input  k   % ant identifier
   input  i   % counter for construction step
   v ← 0.0
   c ← ant[k].tour[i − 1]
   for j = 1 to n do
      if not ant[k].visited[j] then
         if choice_info[c][j] > v then
            nc ← j          % city with maximal τ^α η^β
            v ← choice_info[c][j]
         end-if
      end-if
   end-for
   ant[k].tour[i] ← nc
   ant[k].visited[nc] ← true
end-procedure
```

Local Search

- Once the solutions are constructed, they may be improved by a local search procedure.
- While a simple 2-opt local search can be implemented in a few lines, the implementation of an efficient variant is somewhat more involved.
- Since the details of the local search are not important for understanding how ACO algorithms can be coded efficiently, we refer to the accompanying code (available at www.aco-metaheuristic.org/aco-code/) for more information on the local search implementation.

Pheromone Update

- The last step in an iteration of AS is the pheromone update.
- This is implemented by the procedure ASPheromoneUpdate, which comprises two pheromone update procedures: **pheromone evaporation** and **pheromone deposit**.
- The first one, Evaporate decreases the value of the pheromone trails on all the arcs (i, j) by a constant factor r.

Pheromone Update

- The second one, DepositPheromone, adds pheromone to the arcs belonging to the tours constructed by the ants.
- Additionally, the procedure ComputeChoiceInformation computes the matrix choice_info to be used in the next algorithm iteration.
- Note that in both procedures care is taken to guarantee that the pheromone trail matrix is kept symmetric, because of the symmetric TSP instances.

Pheromone Update

- AS: management of the pheromone updates.

procedure ASPheromoneUpdate
  Evaporate
  for $k = 1$ **to** m **do**
    DepositPheromone(k)
  end-for
  ComputeChoiceInformation
end-procedure

Pheromone Update

- AS: implementation of the pheromone evaporation procedure.

```
procedure Evaporate
  for i = 1 to n do
    for j = i to n do
      pheromone[i][j] ← (1 − ρ) · pheromone[i][j]
      pheromone[j][i] ← pheromone[i][j]    % pheromones are symmetric
    end-for
  end-for
end-procedure
```

Pheromone Update

- AS: implementation of the pheromone deposit procedure.

procedure DepositPheromone(k)
    input k % ant identifier
    $\Delta\tau \leftarrow 1/ant[k].tour\_length$
    for $i = 1$ **to** n **do**
        $j \leftarrow ant[k].tour[i]$
        $l \leftarrow ant[k].tour[i + 1]$
        $pheromone[j][l] \leftarrow pheromone[j][l] + \Delta\tau$
        $pheromone[l][j] \leftarrow pheromone[j][l]$
    end-for
end-procedure

Pheromone Update

- When attacking **large TSP instances**, profiling the code showed that the pheromone evaporation and the computation of the choice_info matrix for **AS** can require a **considerable** amount of computation time.
- But in **ACS** only the pheromone trails of arcs that are crossed by some ant have to be changed and the number of ants in each iteration is a low constant.

Statistical Information

- The last step in the implementation of AS is to store statistical data on algorithm behavior such as:
  - the best-found solution since the start of the algorithm run,
  - the iteration number at which the best solution was found
- Details about these procedures are available at www.aco-metaheuristic.org/aco-code/.

Changes for Other ACO Algorithms

Changes for Other ACO Algorithms

- Some of the necessary adaptations are described when implementing AS variants, in the following:
- 1. When depositing pheromone, the solution may be given some weight, as is the case in **EAS** and **AS$_{\text{rank}}$**. This can be accomplished by simply adding a weight factor as an additional argument of the procedure DepositPheromone.
- 2. **MMAS** has to keep track of the pheromone trail limits. The best way to do so is to integrate this into the procedure ASPheromoneUpdate.

Changes for Other ACO Algorithms

- 3. Finally, the search control of some of the AS variants may need minor changes. Examples are occasional pheromone trail **reinitializations** or the schedule for the frequency of the best-so-far update in MMAS.

Changes for Other ACO Algorithms

- Unlike AS variants, the implementation of ACS requires more significant changes, as listed in the following:
- 1. The implementation of the pseudorandom proportional action choice rule requires the generation of a random number q uniformly distributed in the interval [0, 1] and the application of the procedure ChooseBestNext if $q < q_0$, or of the procedure ASDecisionRule otherwise.

Changes for Other ACO Algorithms

- 2. The local pheromone update can be managed by the procedure ACSLocalPheromoneUpdate (see the figure) that is always invoked immediately after an ant moves to a new city.
- 3. The implementation of the global pheromone trail update is similar to the procedure for the local pheromone update except that pheromone trails are modified only on arcs belonging to the best-so-far tour.

Changes for Other ACO Algorithms

- Implementation of the local pheromone update in ACS.

procedure ACSLocalPheromoneUpdate(k, i)

input k % ant identifier

input i % counter for construction step

$h \leftarrow ant[k].tour[i-1]$

$j \leftarrow ant[k].tour[i]$

$pheromone[h][j] \leftarrow (1-\xi)pheromone[h][j] + \xi\tau_0$

$pheromone[j][h] \leftarrow pheromone[h][j]$

$choice\_info[h][j] \leftarrow pheromone[h][j] \cdot \exp(1/dist[h][j], \beta)$

$choice\_info[j][h] \leftarrow choice\_info[h][j]$

end-procedure

Changes for Other ACO Algorithms

- 4. The integration of the computation of new values for the matrix choice_info into the local and the global pheromone trail update procedures.

References

References

- M. Dorigo and T. Stützle. **Ant Colony Optimization**, MIT Press, Cambridge, 2004.

The End