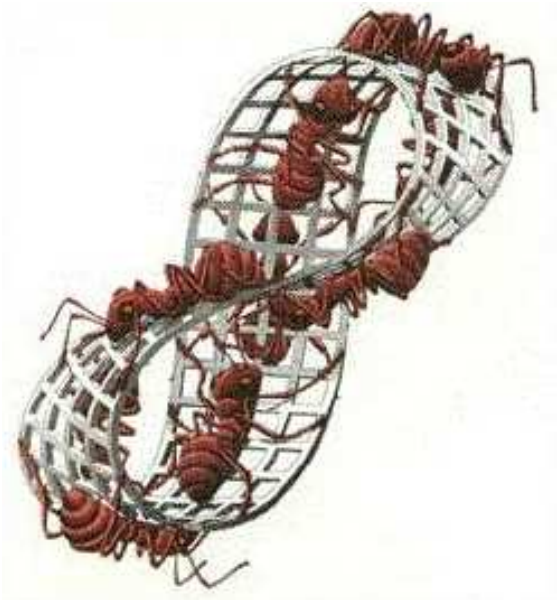
ACO and other (meta)heuristics for CO



Outline

- Notes on combinatorial optimization and algorithmic complexity
- Construction and modification metaheuristics: two complementary ways of searching a solution space
- Finally, Ant Colony Optimization explained
- Other population-based metaheuristics for optimization problems: Genetic Algorithms, Cultural Algorithms, Cross-entropy
- Single agent metaheuristics: Local Search, Tabu Search, Rollout algorithms

Ant Colony Optimization metaheuristic

- Reverse engineering of the mechanisms behind the ant colony shortest path behavior [Dorigo et al., 1991]
- ***Multi-agent architecture based on stigmergy***. The agents, called ***ants*** are an abstraction and an engineered version of real ants
- Target: solution of ***combinatorial optimization problems*** (static and dynamic, centralized and distributed)
- General idea: *repeated construction of solutions using a stochastic policy, observation of the results, and updating of real-valued variables used in turn by the decision policy in order to bias subsequent solution construction toward the most promising areas of the search space*

Before proceeding, let's understand fi rst what a metaheuristic is, and which is the class of problems ACO is intended for

Algorithmic complexity

- The *time complexity function* of an algorithm for a given problem indicates, for each input size n, the maximum time the algorithm needs to fi nd a solution to an instance of that size (worst-case time complexity). For instance, a ***polynomial time algorithm*** has time complexity $\mathcal{O}(g(n))$ where g is a polynomial function. If the time complexity cannot be bounded by a polynomial the algorithm is said ***exponential*** [Garey and Johnson, 1979]
- A problem is said ***intractable*** if there is no polynomial time algorithm able to solve it
- For the so called *NP-hard* class of optimization problems exact algorithms need, in the worst case, exponential time to fi nd the optimum (e.g., Traveling salesman, quadratic assignment, vehicle routing, graph coloring, satisfi ability. . .)
- ***Exact algorithms*** are guaranteed to fi nd the optimal solution and to prove its optimality for every fi nite size instance of the combinatorial problem in bounded, instance-dependent time

Heuristics and metaheuristics

- For most NP-hard problems of interest the performance of exact algorithms is not satisfactory. Huge computational times are required sometimes even for small instances. It is therefore necessary in practice to trade optimality for effi ciency. ***Heuristic methods*** seek to obtain good solutions at relatively low computational cost without being able to guarantee the optimality of the solution
- ***Asymptotic convergence*** can be often proved for specifi c cases. An algorithm that comes with the formal proof that it returns a solution worse than the optimal one by at most some fi xed value or factor is an ***approximation algorithm***
- A ***metaheuristic*** is a high-level strategy which guides other (problem-specifi c) heuristics to search for solutions in a possibly wide set of problem domains (e.g., [Glover and Kochenber, 2002]. A metaheuristic can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifi cations to make them adapted to the specifi c problem (e.g., ***Tabu Search, Simulated Annealing, Iterated Local Search, Evolutionary Computation, ACO***)

A trivial example of real-life metaheuristic

- Meta-strategy: Before buying a new item (e.g., a new laptop or a new pair of shoes), it is mandatory to check at least 5 shops and collect related information
- The high-level strategy is independent from the specific item and doesn't tell anything about how the shops are actually selected and how the final decision is taken. We therefore need at least two additional heuristics for the specific tasks of selecting the shop and taking the final decision

 Can you think of examples of metaheuristics that you adopt in your real-life and/or that are adopted in your scientific field of interest? Do you think it is useful in

Combinatorial problems

- *Instance of a combinatorial optimization problem*: is a pair (S, J), where S is a ***finite set of feasible solutions***, and J is a function that associates a real ***cost*** to each feasible solution, $J : S \to \mathbb{R}$. The problem consists in finding the element $s^* \in S$ which minimizes the function J:

$$s^* = \arg\min_{s \in S} J(s).$$

- Given the finiteness of the set S, the minimum of J on S indeed exists for at least one element
- In the case of continuous optimization S is a subset of $\mathbb{R}^n$ and the global minimum might not exist (if the set is open)
- *A combinatorial optimization problem* is a *set of instances* of an optimization problem, usually all sharing some core properties (e.g., species' genome and individual's DNA)
- A *solution* is a feasible solution in the set S.
- If mapping J changes during the execution of the algorithm, $J = J(s, t)$, the problem is *dynamic*

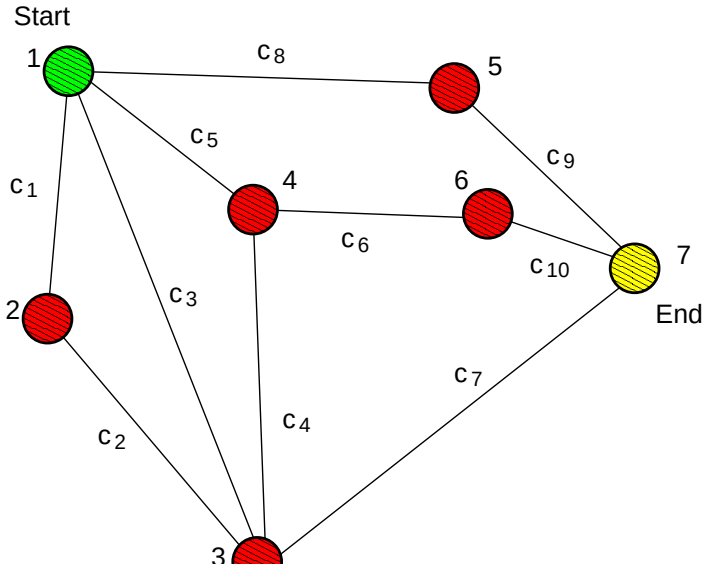
Characteristics of a solution

- Given the fi niteness of the solution set, this requires that the set of elements that can be part of a solution is also fi nite. Let's call them ***solution components***
- Example: fi nd the shortest path connecting two end-points.

$$S = \{(1237), (137), (13467), (123467), (1467), (1567)\}$$

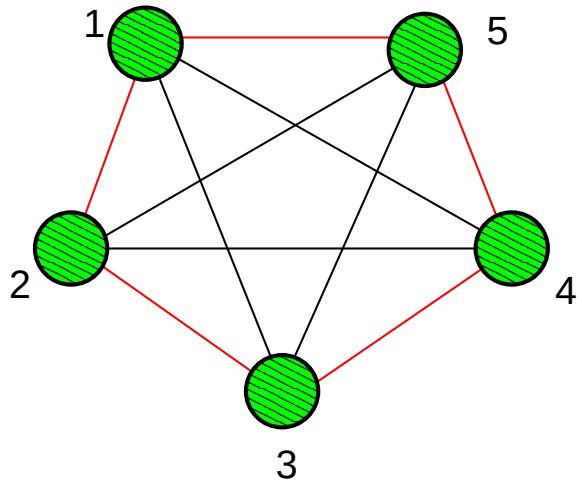
is the loopless solution set. The cost of a solution can be the sum of the costs of the single links. The set C of the solution components is:

$$C = \{1, 2, 3, 4, 5, 6, 7\}$$



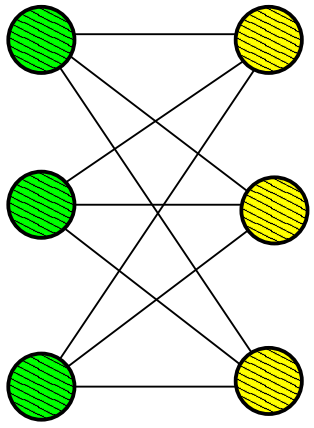
TSP: constrained shortest path

- Traveling Salesman Problem: the minimal cost closed circuit touching all the nodes
- NP-hard, very fameus, loads of applications, studied since about 1850
- For n cities $\rightarrow n!$ solutions (n=10, n! = 3,628,800)
- A solution is a ciclic permutation of cities: $s = (132451)$. $J = \sum_{\mathcal{P}} d_{ij}$, where d_{ij} are the distances, or, more in general, the link costs: $J = d_{13} + d_{32} + d_{24} + d_{45} + d_{51}$



QAP: quadratic assignment problem

- Quadratic Assignment Problem: minimizing the assignment of activities $a_i \in A$ to locations $l_j \in L$
- Extension of TSP, which can be seen as a problem of assigning a city to a position from 1 to n (bipartite assignment problems). NP-hard but more diffi cult than TSP
- A solution is a complete assignment:
  $s = (a_1, l_3), (a_2, l_4), (a_3, l_5), (a_4, l_1), (a_5, l_2)$
- Every possible pair $a_i l_j$ has a cost. The objective is to minimizing the sum of all costs: $J(s) = \sum_{a_i, l_j \in s} c_{ij}$, where c_{ij} is the cost of assigning activities a_i to location l_j.
  $J(s) = c_{13} + c_{24} + c_{35} + c_{41} + c_{52}$.



Construction heuristics

- Starting from an empty *partial solution* $x_0 = \emptyset$, a complete solution $s \in S$ is incrementally built by adding one-at-a-time a new component $c \in C$ to the partial solution.
- The generic iteration (also termed *transition*) of a construction process can be described as:

$$x_j = \{c_1, c_2, \ldots, c_j\} \rightarrow x_{j+1} = \{c_1, c_2, \ldots, c_j, c_{j+1}\}, \quad c_i \in C, \tag{1}$$

  where $x_j \in X' = \wp(C)$ is a *partial solution* of cardinality (length) $j,\ j \leq |C| < \infty$.
- Solution construction: ***sequential decision process***

Generic construction algorithm

procedure Generic_construction_algorithm$()$

$t \leftarrow 0;$

$x_t \leftarrow \emptyset;$

while $(x_t \notin S \ \vee \ \neg stopping\_criterion)$

$c_t \leftarrow$ select_component$(C|\ x_t);$

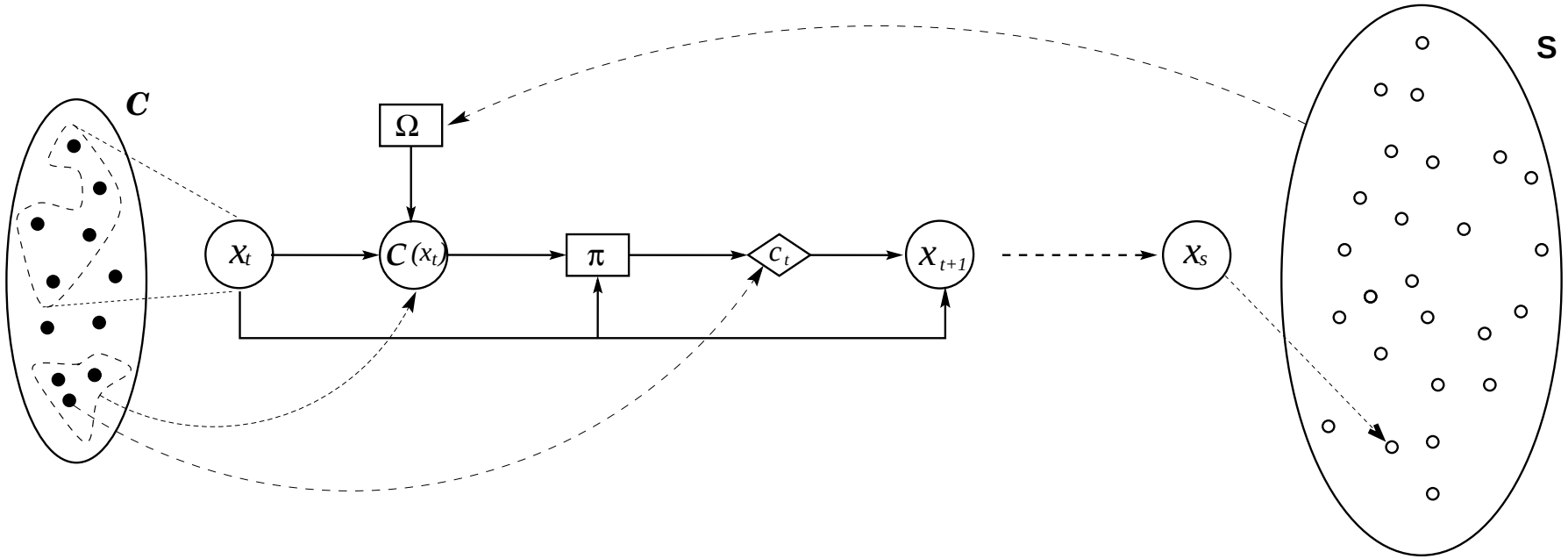
$x_{t+1} \leftarrow$ add_component$(x_t, c_t);$

$t \leftarrow t + 1;$

end while

return $x_t;$

Construction illustrated

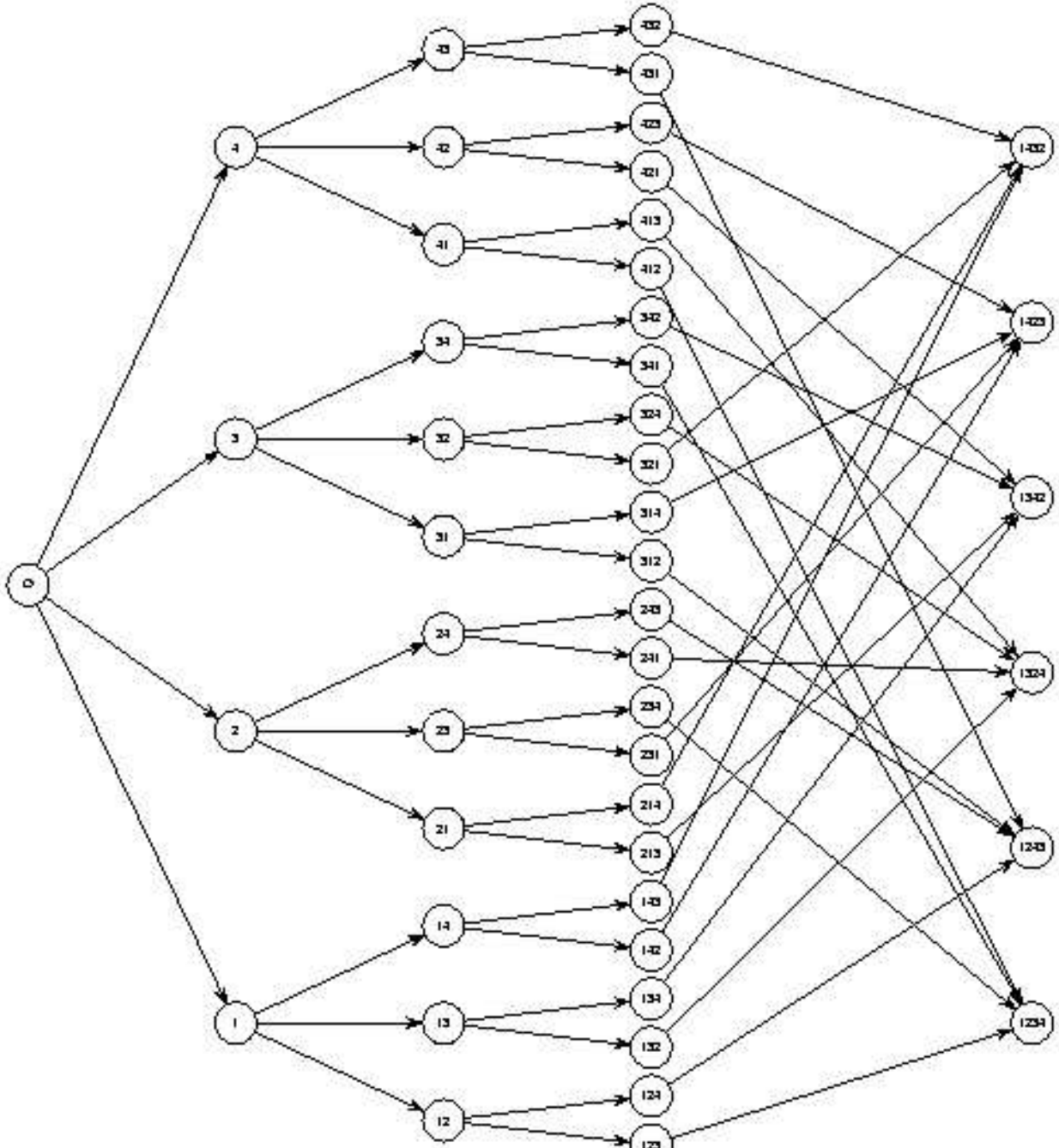


- Step-by-step dependence on all the past decisions:

$$P(c_t) = P\big(c_t \mid (c_k^1, c_{k+1}^2, \ldots, c_{k+t-1}^{t-1})\big),$$

- Appropriate when the cost $J(s)$ is a combination of contributions each one related to the fact that a particular component c_j is included into the partial solution solution x_j

State diagram of the sequential process



Modification methods

- Construction algorithms work on the set of solution components
- Modification strategies act upon the *search space of the complete solutions*: start with a complete solution and proceed by *modifications* of it
- Construction methods make use of an *incremental local view* of a solution, while modification approaches are based on a *global view* of a solution
- This class numbers some of the most effective algorithms and heuristics for combinatorial problems

Neighborhood of a solution

- There is no assigned metric as in the continuous. The notion of ***proximity*** is arbitrary
- A neighborhood is a mapping $\mathcal{N}$ that associates to each feasible solution s of an optimization problem a set $\mathcal{N}(s)$ of other feasible solutions. The mapping can be conveniently expressed in terms of a rule M that, given s, allows the defi nition of the set of solutions belonging to the neighborhood by applying some *modification* procedure on the components of s:

  $\mathcal{N}(s) = \{s' : s' \in S \ \wedge \ s'$ can be obtained from s from $M(s)\}$.

- $\mathcal{N}$ can be random but only those mappings preserving a certain degree of correlation between the value $J(s)$ associated to the point s and the values associated to the points in $\mathcal{N}(s)$ are really meaningful.
  Measure of *closeness* among the values associated to the solutions: if some components of s_t are fi xed, the values of the solutions generated by modifying other components through M should be correlated to the value of s_t.

Local search methods

- Look for a solution *locally optimal with respect to the defined neighborhood structure*
- They are based on the iteration of the process of neighborhood examination until no further improvements are found.
- The most effective heuristics [Aarts and Lenstra, 1997]

Generic local search algorithm

```
procedure Modifi cation_ heuristic()
   defi ne_ neighborhood_ structure();
   s ← get_initial_solution(S);
   s_best ← s;
   while (¬ stopping_criterion)
      s' ← select_solution_from_neighborhood(N(s));
      if (accept_solution(s'))
         s ← s';
         if (s < s_best)
            s_best ← s;
         end if
      end if
   end while
return s_best;
```

Main design issues

- Neighborhood structure
- Generation of the initial solution
- Selection of a candidate solution from the neighborhood of the current solution
- Criterion to accept or reject such selected solution