# Optimización por colonias de hormigas

Metaheurísticas

Moisés Ramírez

10 de enero de 2017

## Introducción

- Ant colony optimization is a **probabilistic technique** for optimization that was introduced in the early 1990's.
- The inspiring source of ant colony optimization is the foraging (search widely for food or provisions) behavior of real ant colonies.
- Searching for optimal path in the graph based on behaviour of ants seeking a path between their colony and source of food.

- A combinatorial optimization problem is one in which the solution consists of a combination of unique components selected from a typically finite, and often small, set.
- The objective is to find the optimal combination of components.

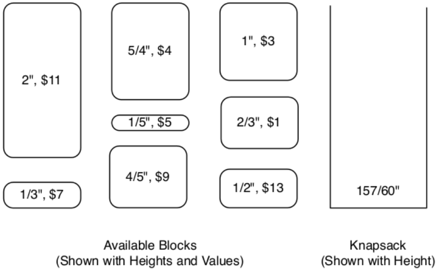
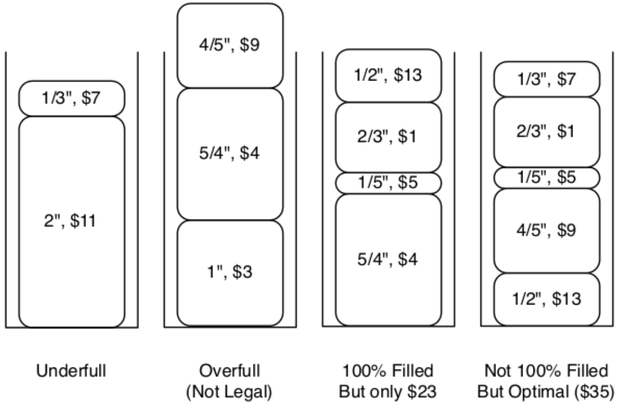


*Figure 54* A knapsack problem. Fill the knapsack with as much value ($$$) without exceeding the knapsack's height.

- Figure shows various combinations of blocks in the knapsack



- knapsack problems show up in the processor queues of operating systems; in allocations of delivery trucks along routes; traveling salesman problem (or TSP).
- **Costs and Values:** While the TSP has cost (the edge weights) which must be minimized, Knapsack instead has value ($) which must be maximized.
  Knapsack does have one thing the TSP doesn't have: it has additional weights (the block heights) and a maximum *weight* which must not be exceeded. The TSP has a different notion of infeasible solutions than simply ones which exceed a certain bound.

## General-Purpose Optimization and Hard Constraints

- As an example, consider the use of a boolean vector in combination with a metaheuristic such as simulated annealing or the genetic algorithm.
- Each slot in the vector represents a component, and if the slot is true, then the component is used in the candidate solution.
- For example, in Figure have blocks of height 2, $1/3$ , $5/4$ , $1/5$ , $4/5$ , 1, $2/3$ , and $1/2$ . A candidate solution to the problem in this Figure would be a vector of eight slots. The optimal answer shown in Figure previous would be $(\textit{false}, \textit{true}, \textit{false}, \textit{true}, \textit{true}, \textit{false}, \textit{true}, \textit{true})$, representing the blocks $1/3$ , $1/5$ , $4/5$ , $2/3$ , and $1/2$.
- The problem with this approach is that it's easy to create solutions which are infeasible.

## hard constraints

- But in a problem like the TSP, a boolean vector might consist of one slot per edge in the TSP graph. It's easy to create infeasible solutions for the TSP which are simply nonsense: how do we assess the *quality* of a candidate solution whose TSP solution isn't even a tour?
- These kind of problems, as configured, have **hard constraints**: There are large regions in the search space which are invalid.
- It would be nice to have a solution which is feasible; and during the search process it'd be nice to have feasible candidate solutions so we can actually think of a way to assign them quality assessments! There are two parts to this: **initialization** (construction) of a candidate solution from scratch, and **Tweaking** a candidate solution into a new one.

## Construction

Iterative construction of components within hard constraints is sometimes straight- forward and sometimes not. Often it's done like this:

1. Choose a component. For example, in the TSP, pick an edge between two cities A and B. In Knapsack, it's an initial block. Let our current (partial) solution start with just that component.
2. Identify the subset of components that can be concatenated to components in our partial solution. In the TSP, this might be the set of all edges going out of A or B. In Knapsack, this is all blocks that can still be added into the knapsack without going over.
3. Tend to discard the less desirable components. In the TSP, we might emphasize edges that are going to cities we've not visited yet if possible.
4. Add to the partial solution a component chosen from among

## Construction

Iterative construction of components within hard constraints is sometimes straight- forward and sometimes not. Often it's done like this:

4. Add to the partial solution a component chosen from among those components not yet discarded.
5. Quit when there are no components left to add. Else go to step 2.

This is an intentionally vague description because iterative construction is almost always highly problem-specific and often requires a lot of thought.

## Tweaking

The Tweak operator can be even harder to do right, because in the solution space feasible solutions may be surrounded on all sides by infeasible ones

- Invent a closed Tweak operator which automatically creates feasible children. This can be a challenge to do, particularly if you're including crossover. And if you create a closed operator, can it generate all possible feasible children? Is there a bias?
- Repeatedly try various Tweaks until you create a child which is feasible. This is relatively easy to do, but it may be computationally expensive.

## Tweaking

- Allow infeasible solutions but construct a quality assessment function for them based on their distance to the nearest feasible solution or to the optimum. This is easier to do for some problems than others. For example, in the Knapsack problem it's easy: the quality of an overfull solution could be simply based on how overfull it is (just like underfull solutions)
- Assign infeasible solutions a poor quality. This essentially eliminates them from the popula- tion; but of course it makes your effective population size that much smaller. It has another problem too: moving just over the edge between the feasible and infeasible regions in the space results in a huge decrease in quality.

## Greedy Randomized Adaptive Search Procedures (GRASP)

- GRASP [1] is a single-state metaheuristic which is built on the notions of constructing and Tweaking feasible solutions, but which doesn't use any notion of component-level *historical quality*.
- The idea is to create a feasible solution by constructing from among highest value (lowest cost) components and then do some hill-climbing on the solution.

[1]Feo, T.A., Resende, M.G.C. *Greedy Randomized Adaptive Search Procedures* Journal of Global Optimization (1995) 6: 109. doi:10.1007/BF01096763

## GRASP

```
1: $C \leftarrow \{C_1, ..., C_n\}$ components
2: $p \leftarrow$ percentage of components to include each iteration
3: $m \leftarrow$ length of time to do hill-climbing

4: $Best \leftarrow \square$
5: repeat
6:     $S \leftarrow \{\}$                                                    ▷ Our candidate solution
7:     repeat
8:         $C' \leftarrow$ components in $C - S$ which could be added to $S$ without being infeasible
9:         if $C'$ is empty then
10:            $S \leftarrow \{\}$                                                 ▷ Try again
11:        else
12:            $C'' \leftarrow$ the $p\%$ highest value (or lowest cost) components in $C'$
13:            $S \leftarrow S \cup \{$component chosen uniformly at random from $C''\}$
14:    until $S$ is a complete solution
15:    for $m$ times do
16:        $R \leftarrow$ Tweak(Copy($S$))    ▷ Tweak must be closed, that is, it must create feasible solutions
17:        if Quality($R$) $>$ Quality($S$) then
18:            $S \leftarrow R$
19:    if $Best = \square$ or Quality($S$) $>$ Quality($Best$) then
20:        $Best \leftarrow S$
21: until $Best$ is the ideal solution or we have run out of time
22: return $Best$
```

## Ant Colony Optimization Algorithm (ACO)

- ACO is an approach to combinatorial optimization, it simply assembles candidate solutions by selecting components which compete with one another for attention.
- Is population-oriented. there is a set of components that make up a candidate solutions to the problem.
- The “fitness” (called the pheromone) of the components in the population is adjusted as time goes on.
- Each generation one or more candidate solutions are built - ant trails. Solutions are built by selecting components one by one based, in part, on their pheromones.
- Evaluation of the fitness of each trail is done for each trail, each of the components in that trail is then updated based on that fitness: a bit of the trail’s fitness is rolled into each component’s pheromone.

## ACO

*An Abstract Ant Colony Optimization Algorithm (ACO)*

```
1: $C \leftarrow \{C_1, ..., C_n\}$ components
2: $popsize \leftarrow$ number of trails to build at once  ▷ "ant trails" is ACOspeak for "candidate solutions"
3: $\vec{p} \leftarrow \langle p_1, ..., p_n \rangle$ pheromones of the components, initially zero
4: $Best \leftarrow \Box$
5: **repeat**
6:     $P \leftarrow popsize$ trails built by iteratively selecting components based on pheromones and costs or values
7:     **for** $P_i \in P$ **do**
8:         $P_i \leftarrow$ Optionally Hill-Climb $P_i$
9:         **if** $Best = \Box$ or Fitness($P_i$) $>$ Fitness($Best$) **then**
10:            $Best \leftarrow P_i$
11:    Update $\vec{p}$ for components based on the fitness results for each $P_i \in P$ in which they participated
12: **until** $Best$ is the ideal solution or we have run out of time
13: **return** $Best$
```