Chapter 6
An Evolutionary Approach with insights of chapters 7 and 8

Companion slides of
How to Solve It: Modern Heuristics
by Zbigniew Michaelwicz and David B. Fogel
(Springer-Verlag, 2nd edition, 2004)
prepared by
José Fernando Oliveira
Maria Antónia Carravilla

Previous approaches

• Each relies on a single solution as the basis for future exploration with each iteration.
  – They either process complete solutions in their entirety, or they construct the final solution from smaller building blocks.
  – They always have a single “current best” solution stored that they try to improve in the next step.
New idea

• What would happen if we instead maintained several solutions simultaneously?
• What would happen if our search algorithm worked with a population of solutions?
  – Not a parallel search method (e.g. parallel simulated annealing)
    • With a parallel computer, where each processor would maintain a single solution and each processor would execute the same algorithm in parallel.
  – But with competition between solutions in a population.

New idea

• Let us simulate the evolutionary process of competition and selection and let the candidate solutions in our population fight for room in future generations!
• Moreover, we can use random variation to search for new solutions in a manner similar to natural evolution.

Genetic algorithms / Evolutionary approaches
Genetic algorithms

• Suppose we start with a population of initial solutions (e.g. generated by sampling randomly from the search space $S$)
  – Sometimes, to emphasize a link with genetics, these solutions are called chromosomes.

• The evaluation function can be used to determine the relative merit of each of our initial solutions. This isn't always straightforward because multiple objectives need to be combined in a useful way.
  – At least, the chosen evaluation function must be capable of differentiating between two individuals, i.e., it has to be able to rank one solution ahead of another.
  – Those solutions that are better, as determined by the evaluation function, are favored to become parents for the next generation of offspring.

Genetic algorithms

• New candidate solutions (offspring) are generated from pairs of solutions (parent solutions).
  – One way we can do this is by taking parts of two parents and putting them together to form an offspring.
  – For example, we might take the first half of one parent together with the second half of another.
  – The first half of one parent might be viewed as a characteristic and similarly so with the second half of the second parent.

• With each generation, the individuals compete – either only among themselves or also against their parents – for inclusion in the next iteration.
General implementation decisions

• How to codify a solution.
• How to generate the initial population.
• How to recombine solutions (parents) to generate new solutions (offsprings) – Crossover.
• How to introduce random variation – Mutation.
• How to evaluate solutions.
• How to generate the following generation of solutions.

EVOLUTIONARY APPROACH FOR THE SAT
Solution codification: chromosomes

• We want to use an evolutionary algorithm to generate solutions for an SAT of 100 variables.

We have a compound Boolean statement in the form
\[ F(x) = (x_{17} \lor \overline{x}_{37} \lor x_{73}) \land (\overline{x}_{11} \lor x_{56}) \land \ldots \land (x_{2} \lor x_{43} \lor \overline{x}_{77} \lor \overline{x}_{89} \lor x_{97}) \]
and we need to find a vector of truth assignments for all 100 variables \( x_i, i = 1, \ldots, 100 \), such that \( F(x) = 1 \) (TRUE).

• Since we are dealing with variables that can only take on two states, TRUE or FALSE, a natural representation to choose is a binary string (vector) of length 100.
  
  – Each component in the string signifies the truth assignment for the variable which corresponds to that location in the string.
  
  – Any binary string of variables would constitute a possible solution to the problem.

Initial population

• How many individuals do we want to have?
  
  – Let’s say that we want 30 parent vectors.

• One way to determine this collection would be to have a 50-50 chance of setting each bit in each vector to be either 1 or 0.
  
  – This would, hopefully, give us a diverse set of parents.
  
  – Typically, this diversity is desirable, although in some cases we might want to initialize all of the available parents to some best-known solution and proceed from there.
  
  – If we use this randomized procedure to determine the initial population then every trial of our evolutionary algorithm will have a potentially different initial population based on the random sampling that we generate.
Evaluation function

• It is straightforward to use $F(x)$ to check if any of our initial vectors are optimal solutions.
  – But if none of these vectors turns out to be optimal, i.e., for each randomly selected initial vector, $x$, $F(x) = 0$, this doesn't provide any useful information for determining how close a particular vector might be to the optimal solution.

Evaluation function

• Instead, what we might do is consider each part of the compound Boolean expression $F(x)$ that is separated by an “and” symbol, and count up how many of these separate subexpressions are TRUE.
  – We could then use the number of TRUE subexpressions as a evaluation function, with the goal of maximizing the number of TRUE subexpressions.
  – If all of the subexpressions are TRUE then it follows that $F(x) = 1$.

The optimal solution(s) should give the optimal evaluation(s)
Evaluating the quality of the solutions of a population is the most time-consuming part of executing a evolutionary algorithm.
Selection function

• We want to judge the quality of our 30 initial vectors
  – Let us presume that there are 20 subexpressions that comprise $F(x)$
    • the best score we can get is 20 and the worst is 0.
  – Suppose that the best vector in the population after this initial scoring has a score of 10, the worst has a score of 1, and the average score is 4.5.

• What we might do next is think about how we want to generate the individuals to survive as parents for the subsequent generation.

Roulette wheel selection

• One method: assign a probability of being selected to each individual in proportion to their relative fitness.
  – A solution with a score of 10 is 10 times more likely to be chosen than a solution with a score of 1.

• This proportional selection is also sometimes called **roulette wheel selection**
  – Because a common method for accomplishing this procedure is to think of a roulette wheel being spun once for each available slot in the next population, where each solution has a slice of the roulette wheel allocated in proportion to their fitness score.

\[
P_i = \frac{F_i}{F} \quad (e.g. \frac{10}{4.5}, \frac{1}{4.5}, \ldots)
\]
Random tournament selection

- Samples a collection of the current population
- Identifies the best individual in the sample
- Retains that individual for the next generation
- Returns the remaining individuals to the population for eventual resampling.
- Most popular tournament strategy: sample of size 2


Selection

- We might obtain multiple copies of some solutions that happened to be chosen more than once in the 30 spins.
- Some solutions very likely wouldn't be selected at all.
  - In fact, even the best solution in the current population might not be selected, just by random chance.
- The roulette wheel or tournament selection merely bias the selection toward better solutions (stochastic selection).
- The selection is applied to the previous generation (parents) together with the new generated set of solutions (offsprings) or parents are discarded and selection is applied just to offsprings to select the best ones.

Parents and offsprings or just offsprings?
Generation of new solutions

• Regardless of the resulting distribution of our solutions, we haven't generated anything new, we have only modified the distribution of our previous solutions.

• Next we need to use random variation to search for new solutions.
  – By recombination of 2 or more solutions: CROSSOVER
  – By modifying one solution: MUTATION

Crossover

• Randomly include the chance that some of our new 30 candidates will recombine.

• For each individual we might have a probability of, say, 0.9 that it will not pass unchanged into the next generation, but instead will be broken apart and reattached with another candidate solution chosen completely at random from the remaining 29 other solutions.

• This procedure is designed, hopefully, to ensure that successive components in candidate strings that tend to generate better solutions have a chance of staying together.

• Whether or not this would be effective partly depends on the degree of independence of the components.
1-point crossover and 2-point crossover


Mutation

• Recombination operator can only generate new solutions for evaluation if there is **sufficient diversity** in the current population.
  
  — If we are unlucky and generate a homogeneous population, we'd be completely stuck, even if our current best solution wasn't a local optimum.

• Therefore, we might also include another operator (**mutation**) that flips a single bit in a chosen string at random, and then **apply this with a low probability**.

• We ensure that we always have the potential to introduce novel variations.

http://www.talkorigins.org/faqs/genalg/genalg.html
EVOLUTIONARY APPROACH FOR THE TSP

Chromosomes

• Suppose that we wanted to use an evolutionary algorithm to generate solutions for a TSP of 100 cities.

• Representation of a possible solution:
  – the natural choice is to use an ordered list of the cities to be visited.
  – for example, the vector \([17, 1, 43, 4, 6, 79, \ldots, 100, 2]\) would mean that the salesman starts at city 17, then goes to city 1, then on to cities 43, 4, 6, 79, and so on through all of the cities until they pass through city 100, and lastly city 2, getting back to city 17 in the end.
Initial population

• Let us consider that we have 30 candidate tours.
• We can generate 30 random permutations of the integers [1...100] to serve as the starting population.
• If we had some hints about good solutions, maybe from some salesmen who have traveled these routes before, we could include these...

Evaluation function

• In this case, the objective of minimizing the total distance of the tour suggests using the evaluation function that assigns a score to each tour that is equivalent to its total length.
  – The longer the tour, the worse the score, and our goal then is to find the tour that minimizes the evaluation function.
• This part of the evolutionary approach appears much more straightforward for the TSP than for the SAT.
New solutions generation: crossover operator

• Just take the first part of one solution and splice on the second part of another would very likely generate a solution that had multiple instances of some cities and no instances of others.

\[
\begin{align*}
[4 & 3 | 5 & 7 & 2 & 6 & 1 ] & \rightarrow [4 & 3 | 4 & 1 & 5 & 6 & 7 ]
\end{align*}
\]

Alternative crossover operators
Partially-mapped crossover (PMX)

• This crossover builds an offspring by choosing a subsequence of a tour from one parent and preserving the order and position of as many cities as possible from the other parent.

• A subsequence of a tour is selected by choosing two random cut points, which serve as boundaries for the swapping operations.
Partially-mapped crossover (PMX)

Example

\[
P_1 = (1\ 2\ 3\ |\ 4\ 5\ 6\ 7\ |\ 8\ 9)
\]
\[
P_2 = (4\ 5\ 2\ |\ 1\ 8\ 7\ 6\ |\ 9\ 3)
\]

\[
O_1 = (x\ x\ x\ |\ 1\ 8\ 7\ 6\ |\ x\ x)
\]
\[
O_2 = (x\ x\ x\ |\ 4\ 5\ 6\ 7\ |\ x\ x)
\]

This swap also defines a series of mappings:

\[1 \leftrightarrow 4,\ 8 \leftrightarrow 5,\ 7 \leftrightarrow 6,\ \text{and}\ 6 \leftrightarrow 7\]

\[
O_1 = (x\ 2\ 3\ |\ 1\ 8\ 7\ 6\ |\ x\ 9)
\]
\[
O_2 = (x\ x\ 2\ |\ 4\ 5\ 6\ 7\ |\ 9\ 3)
\]

Fill in additional cities from the original parents for which there's no conflict.

\[
O_1 = (4\ 2\ 3\ |\ 1\ 8\ 7\ 6\ |\ 5\ 9)
\]
\[
O_2 = (1\ 8\ 2\ |\ 4\ 5\ 6\ 7\ |\ 9\ 3)
\]

The first \(x\) in the offspring \(O_1\) (which should be 1, but there was a conflict) is replaced by 4 because of the mapping \(1 \leftrightarrow 4\). Similarly for the other \(x\).

Alternative crossover operators

Order crossover (OX)

- This crossover builds offspring by choosing a subsequence of a tour from one parent and preserving the relative order of cities from the other parent.
Order crossover (OX)

Example

\[
P1 = (1\ 2\ 3\ |\ 4\ 5\ 6\ 7\ |\ 8\ 9)
\]
\[
P2 = (4\ 5\ 2\ |\ 1\ 8\ 7\ 6\ |\ 9\ 3)
\]
\[
O1 = (x\ x\ x\ |\ 1\ 8\ 7\ 6\ |\ x\ x)
\]
\[
O2 = (x\ x\ x\ |\ 4\ 5\ 6\ 7\ |\ x\ x)
\]

The symbol “x” will be interpreted as “at present unknown”.

Sequence of cities in the second parent (from the second cut point).

Sequence of cities after removing cities 4, 5, 6, and 7, which are already in the first offspring.

9 -3 -4 -5 -2 -1 -8 -7 -6
9 -3 -2 -1 -8

\[
O1 = (2\ 1\ 8\ |\ 4\ 5\ 6\ 7\ |\ 9\ 3)
\]
\[
O2 = (3\ 4\ 5\ |\ 1\ 8\ 7\ 6\ |\ 9\ 2)
\]

This sequence is placed in the first offspring (starting from the second cut point). Similarly we obtain the other offspring.

Alternative crossover operators

Cycle crossover (OX)

• This crossover builds offspring in such a way that each city and its position comes from one of the parents.
Cycle crossover (OX)
Example

P1 = (1 2 3 4 5 6 7 8 9)
P2 = (4 1 2 8 7 6 9 3 5)

O1 = (1 x x x x x x x x)
The first city from the first parent.

O1 = (1 x x 4 x x x x)
The next city to be considered must be city 4, as it's the city from parent P2 that's just “below” the selected city 1. In P1 this city is at position “4”.

O1 = (1 x x 4 x x x 8 x)
The selection of city 2 requires selecting city 1, which is already on the list. Thus, we have completed a cycle.

O1 = (1 x 3 4 x x x 8 x)

O1 = (1 2 3 4 x x x 8 x)
The remaining cities are filled in from the other parent.

The symbol “x” will be interpreted as “at present unknown”.

CX preserves the absolute position of the elements in the parent sequence.
Mutation operators

Inversion

• Simple inversion selects two points along the length of the permutation, which is then cut at these points, and the substring between these points is reversed.

\[(1 2 \mid 3 4 5 6 \mid 7 8 9)\]
\[(1 2 \mid 6 5 4 3 \mid 7 8 9)\]

Mutation operators

Insertion and Displacement

• Selects a city and inserts it in a random place:

\[(1 2 3 4 5 6 7 8 9)\]
\[(1 5 2 3 4 6 7 8 9)\]

• Selects a subtour and inserts it in a random place:

\[(1 2 3 4 5 6 7 8 9)\]
\[(1 5 6 7 2 3 4 8 9)\]
Mutation operators
Reciprocal exchange

• Swaps two cities:

(1 2 3 4 5 6 7 8 9)
(1 5 3 4 2 6 7 8 9)
Structure of an evolutionary algorithm

**procedure** evolutionary algorithm
begin
\[ t \leftarrow 0 \]
initialize \( P(t) \)
evaluate \( P(t) \)
while (not termination-condition) do
begin
\[ t \leftarrow t+1 \]
select \( P(t) \) from \( P(t-1) \)
alter \( P(t) \)
evaluate \( P(t) \)
end
end
Structure of an evolutionary algorithm

http://tim.hibal.org/blog/?p=40