Selected Topics in Nature-inspired Algorithms

Seminar

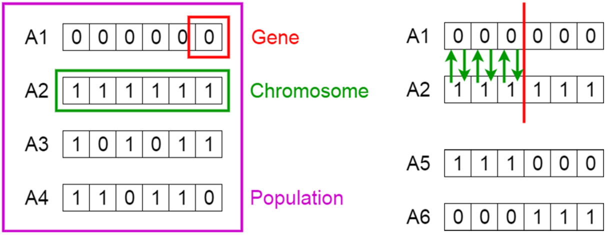
Summer 2018

Nico Potyka

Genetic Algorithms

Genetic Algorithms: Idea

- search algorithms inspired by genetics and natural selection
- genetic algorithms evolve population of solutions
- solutions are represented as chromosomes
- recombination operators model reproduction
- mutation operators model mutation



Basic Genetic Algorithm

$\text{population} \leftarrow \text{initialize}(\text{N})$

do

  $\text{mating\_pool} \leftarrow \text{select}(\text{population})$

  $\text{offspring} \leftarrow \text{recombine}(\text{mating\_pool})$

  $\text{offspring} \leftarrow \text{mutate}(\text{offspring})$

  $\text{population} \leftarrow \text{replace}(\text{population}, \text{offspring})$

until *termination condition reached*

return $\text{best}(\text{population})$

Example: Knapsack

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Chromosome Representation: 0-1 arrays

Population size: N = 5

Initialization: Random

0	0	0	0	0
0	1	0	1	0
1	0	1	0	1
0	0	1	1	0
1	1	1	1	1

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Fitness Function:

0 if weight > W, (infeasible solutions have fitness 0)

1 + value otherwise (feasible solutions have positive fitness)

0	0	0	0	0	**Fitness: 1**
0	1	0	1	0	**Fitness: 7001**
1	0	1	0	1	**Fitness: 0**
0	0	1	1	0	**Fitness: 9001**
1	1	1	1	1	**Fitness: 0**

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Selection: Fitness-proportionate

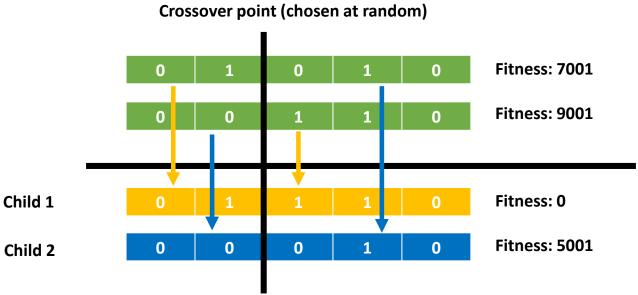
0	0	0	0	0	**Fitness: 1**	**Probability: 1/16003 ≈ 0**
0	1	0	1	0	**Fitness: 7001**	**Probability: 7001/16003 ≈ 0.44**
1	0	1	0	1	**Fitness: 0**	**Probability: 0**
0	0	1	1	0	**Fitness: 9001**	**Probability: 9001/16003 ≈ 0.56**
1	1	1	1	1	**Fitness: 0**	**Probability: 0**

Cumulated Fitness: 16003

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Recombination: 1-point crossover



Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Mutation: bit-flip mutation

Child 1	0	1	1	1	0	**Fitness: 0**
Child 2	0	0	0	1	0	**Fitness: 5001**

Child 1	0	1	0	1	1	**Fitness: 12000**
Child 2	1	0	0	1	0	**Fitness: 6001**

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Replacement: Elitist (Survival of the fittest)

0	0	0	0	0	Fitness: 1
0	1	0	1	0	Fitness: 7001
~~1~~	~~0~~	~~1~~	~~0~~	~~1~~	Fitness: 0
0	0	1	1	0	Fitness: 9001
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	Fitness: 0
0	1	0	1	1	Fitness: 12000
1	0	0	1	0	Fitness: 6001

0	0	0	0	0
0	1	0	1	0
0	0	1	1	0
0	1	0	1	1
1	0	0	1	0

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Termination: when 100 generations have been created

0	0	0	0	0
0	1	0	1	0
0	0	1	1	0
0	1	0	1	1
1	0	0	1	0

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Algorithm Analysis

before implementing your solution, think about alternatives

- what are potential problems with your algorithm?
- could they be solved by replacing some building blocks?

Algorithm Analysis

before implementing your solution, think about alternatives

- what are potential problems with your algorithm?
- could they be solved by replacing some building blocks?

potential problems for our Knapsack solution

- algorithm may create many infeasible solutions
- potential solutions
    1. change solution representation and objective function
    2. change reproduction and mutation operation
    3. add a repair operation that fixes infeasible solutions
    4. ...

Example: Knapsack 2

Change Solution Representation

- previously, we viewed problem as an assignment problem: i-th gene indicates that i-th item is in knapsack
- rephrase problem as a permutation problem
  - chromosomes are permutations of item indices
  - go through items according to permutation order
    1. if item is within weight limit, add item to solution
    2. otherwise, omit item
  - no infeasible solutions will be generated anymore

Chromosome Representation: {1, 2, 3, 4, 5} permutations

Population size: N = 5

Initialization: Random

1	2	3	4	5
3	1	2	5	4
2	4	1	3	5
1	5	2	3	4
5	4	3	2	1

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

From permutation to solution

S = {}
w = 0

1	2	3	4	5

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

From permutation to solution

S = {1}
w = 10

1	2	3	4	5

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

From permutation to solution

S = {1,2}
w = 110

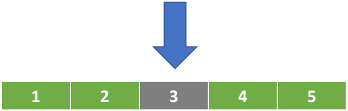
1	2	3	4	5

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

From permutation to solution

S = {1,2}
w = 110



Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

From permutation to solution

S = {1,2,4}
w = 111

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

From permutation to solution

S = {1,2,4,5}
w = 311

1	2	3	4	5

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Fitness Function: value of permutation

1	2	3	4	5	**Fitness: 13000**
3	1	2	5	4	**Fitness: 10000**
2	4	1	3	5	**Fitness: 13000**
1	5	2	3	4	**Fitness: 13000**
5	4	3	2	1	**Fitness: 13000**

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Selection: Fitness-proportionate

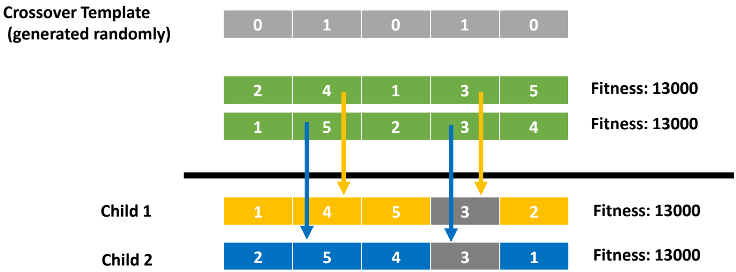
1	2	3	4	5	**Fitness: 13000**	**Probability: 13/62 ≈ 0.21**
3	1	2	5	4	**Fitness: 10000**	**Probability: 10/62 ≈ 0.16**
2	4	1	3	5	**Fitness: 13000**	**Probability: 13/62 ≈ 0.21**
1	5	2	3	4	**Fitness: 13000**	**Probability: 13/62 ≈ 0.21**
5	4	3	2	1	**Fitness: 13000**	**Probability: 13/62 ≈ 0.21**

Cumulated Fitness: 62000

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Recombination: uniform order-based crossover



Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Mutation: swap mutation

Child 1	1	4	5	3	2	Fitness: 13000
Child 2	2	5	4	3	1	Fitness: 13000

Child 1	1	3	5	4	2	Fitness: 10000
Child 2	1	5	4	3	2	Fitness: 13000

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Replacement: Elitist

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Termination: when 100 generations have been created

1	2	3	4	5
2	4	1	3	5
1	5	2	3	4
5	4	3	2	1
1	5	4	3	2

Item	1	2	3	4	5
Weight	10	100	300	1	200
Value	1000	2000	4000	5000	5000

W = 400

Algorithm Analysis

what is the tradeoff?

- no infeasible solutions anymore
- building up solution from chromosome is slightly more complicated
- run experiments with both implementations to evaluate

Design Choices

```
population ← initialize(N)
do
  mating_pool ← select(population)
  offspring ← recombine(mating_pool)
  offspring ← mutate(offspring)
  population ← replace(population, offspring)
until termination condition reached
return best(population)
```

Solution Representation

Some choices

- assignment problem: genes correspond to values of variables
- permutation problem: chromosomes represent some ordering

population $\leftarrow$ **initialize(N)**

do

  mating_pool $\leftarrow$ select(population)

  offspring $\leftarrow$ recombine(mating_pool)

  offspring $\leftarrow$ mutate(offspring)

  population $\leftarrow$ replace(population, offspring)

until *termination condition reached*

return best(population)

Initialization

Some choices

- initialize at random
- use fast heuristic to initialize non-naive initial population

Adding extreme solutions initially can be a good idea

- 0- and 1-assignment in binary assignment problems
- ascending and descending order in permutation problems

```
population ← initialize(N)
do
  mating_pool ← select(population)
  offspring ← recombine(mating_pool)
  offspring ← mutate(offspring)
  population ← replace(population, offspring)
until termination condition reached
return best(population)
```

Selection

- fitness-proportionate selection: pick chromosome at random

$$P(c \text{ is selected}) = \frac{f(c)}{\sum_{c' \in \mathrm{Population}} f(c')}$$

- basic tournament selection with parameter s
  1. choose s chromosomes at random
  2. select fittest chromosome from chosen ones

- truncated selection with parameter s
  1. pick $\frac{N}{s}$ fittest chromosomes
  2. every picked chromosome gets s copies in mating pool

population ← initialize(N)

do

    mating_pool ← select(population)

    offspring ← **recombine(mating_pool)**

    offspring ← mutate(offspring)

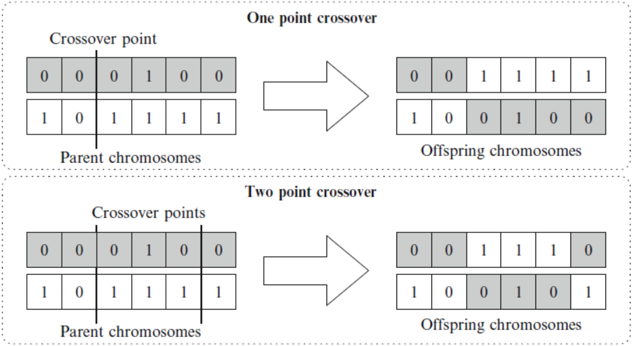
    population ← replace(population, offspring)

until *termination condition reached*

return best(population)

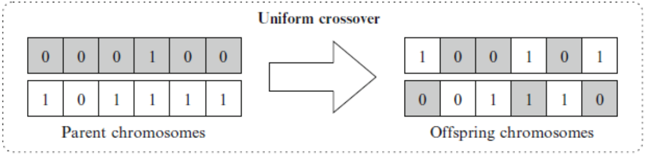
Recombination

- k-point crossover (assignment problem)
    1. pick k crossover points at random
    2. assign blocks from parents to children alternatingly



Recombination

- uniform crossover (assignment problem)
  - for each gene, child 1 inherits gene from parent 1 with probability p and from parent 2 with probability $1 - p$
  - usually, $p = 0.5$



Recombination

- uniform order-based crossover (permutation problem)
  1. generate 0-1 template at random
  2. for all 1-indices, copy values of *i*-th parent to *i*-th child
  3. add remaining values according to order given by other parent

Parent P_1	A	B	C	D	E	F	G
Parent P_2	E	B	D	C	F	G	A
Template	0	1	1	0	0	1	0
Child C_1	E	B	C	D	G	F	A
Child C_2	A	B	D	C	E	G	F

```
population ← initialize(N)
do
  mating_pool ← select(population)
  offspring ← recombine(mating_pool)
  offspring ← mutate(offspring)
  population ← replace(population, offspring)
until termination condition reached
return best(population)
```

Mutation

- bit-flip mutation (assignment problem): change each gene to a random value with probability p_m
- swap mutation (permutation problem): pick two indices at random and swap genes, repeat m times

```
population ← initialize(N)
do
   mating_pool ← select(population)
   offspring ← recombine(mating_pool)
   offspring ← mutate(offspring)
   population ← replace(population, offspring)
until termination condition reached
return best(population)
```

Replacement

- Elitist: keep N best solutions from both current population and mating pool
- Delete all: replace current population with mating pool
- Steady-state: replace n chromosomes from current population with n new members from mating pool
  - delete/pick worst/best
  - delete/pick anti-fitness-proportionally/fitness-proportionally

Remark: Keeping the currently best solution is always a good idea

```
population ← initialize(N)
do
  mating_pool ← select(population)
  offspring ← recombine(mating_pool)
  offspring ← mutate(offspring)
  population ← replace(population, offspring)
until termination condition reached
return best(population)
```

Termination Condition

- maximum number of generations
- maximum number of unchanged generations
- maximum number of non-improving generations
- time limit
- fitness threshold reached

population $\leftarrow$ initialize(N)

do

  mating_pool $\leftarrow$ select(population)

  offspring $\leftarrow$ recombine(mating_pool)

  offspring $\leftarrow$ mutate(offspring)

  population $\leftarrow$ replace(population, offspring)

until *termination condition reached*

return **best(population)**

Return value

- return (one) best solution
- return all best solutions
- apply local search algorithm to best solutions and return
  - (one) best improved solution
  - all best improved solutions

Hybrid Genetic Algorithms

- hybrid algorithms combine multiple algorithms
- some hybridization ideas for genetic algorithms
  - apply fast local search algorithm to final generation after termination
  - apply fast local search algorithm to mutated offspring in every iteration

Possible Parameters

- population size
- mating pool size
- mutation probability
- other parameters of submodules

population $\leftarrow$ initialize(N)

do

  mating_pool $\leftarrow$ select(population)

  offspring $\leftarrow$ recombine(mating_pool)

  offspring $\leftarrow$ mutate(offspring)

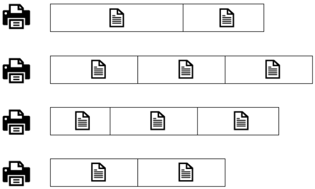
  population $\leftarrow$ replace(population, offspring)

until *termination condition reached*

return best(population)

Programming Task

Makespan Problem



- Problem: given jobs $1, \ldots, n$ with processing time $p_1, \ldots, p_n$ and m machines, assign jobs to machines in a way that minimizes time to finish all jobs (makespan)
- Examples
  - job shop scheduling
  - multiprocessor scheduling (assign threads to processors)
  - project management (assign tasks to team members)

Makespan Problem Formalized

- given m identical machines and
- n jobs $1, \ldots, n$ with processing time $p_1, \ldots, p_n$
- assign jobs to machines in a way that minimizes makespan
- optimization problem
  - Variables: n variables with domain $\{1, \ldots, m\}$ ($x_j = k$ iff we assign job j to machine k)
  - no constraints
  - Objective function: $f(x_1, \ldots, x_n) = \max\{T_1, \ldots T_m\}$, where $T_k = \sum_{x_j = k} p_j$ (time needed by machine k)

Makespan Programming Task

- decompose genetic algorithm into independent modules
  - Initializer
  - Selector
  - Recombiner
  - Mutator
  - Replacer
- for each module, implement two variants
- divide the work among group members
  (agree on data structures and interfaces first)

Makespan Programming Task

- make a few experiments with different modules and different parameter settings for all benchmark problems (classes)
  (try different population size, mating pool size, etc.)
- document your findings and prepare a small presentation (5-10 minutes)
- send me a compressed archive containing
  1. slides (structure findings in table or other visualization)
  2. source files/ notebook
  3. assignment of tasks to group members

Problem Decomposition

- the simplest way to 'modularize' the problem is to create different functions and case differentiations

```
if(initializer1) {

    population = initialize1();

}
else if(initializer2) {

    population = initialize2();

}
```

- however, the code will become messy and difficult to maintain
- if you have time, try to apply object-oriented or functional programming techniques

Object-oriented Problem Decomposition

- create an (abstract) class for each module
- variants inherit from (or implement) these classes

```
public class GeneticAlgorithm {

    private Initializer initializer;
    private Selector selector;
    private Recombiner recombiner;
    private Mutator mutator;
    private Replacer replacer;

    public GeneticAlgorithm(Initializer initializer, Selector selector, Recombiner recombiner, Mutator mutator, Replacer replacer) {
        this.initializer = initializer;
        this.selector = selector;
        this.recombiner = recombiner;
        this.mutator = mutator;
        this.replacer = replacer;
    }

  ...
```

```
public interface Initializer {

    public Collection<Chromosome> initializePopulation(Problem p, int pop_size);

}
```

```
public interface Recombiner {

    public Collection<Chromosome> recombine(Collection<Chromosome> mating_pool);

}
```

Object-oriented Problem Decomposition

- implementation uses only the functionality provided by the (abstract) class

```
public Assignment search(Problem p, int pop_size, int pool_size, double mutation_prob, long time_limit, boolean verbose) {

    long time_spent = 0;
    long noIterations = 0;
    long startTime = System.currentTimeMillis();
    Collection<Chromosome> population = initializer.initializePopulation(p, pop_size);

    if(verbose) {
        printInfo(p, population, noIterations, time_spent);
    }

    do {

        Collection<Chromosome> mating_pool = selector.select(p, population, pool_size);
        Collection<Chromosome> offspring = recombiner.recombine(mating_pool);
        mutator.mutate(p, offspring, mutation_prob);
        population = replacer.replace(p, population, offspring);

        time_spent = System.currentTimeMillis() - startTime;

        if(verbose) {
            noIterations++;
            if(noIterations%10 == 0) {
                printInfo(p, population, noIterations, time_spent);
            }
        }

    }
    while(time_spent < time_limit);

    return getFittest(p, population).getAssignment();

}
```

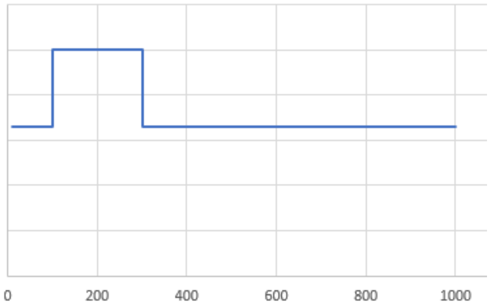
Pseudocode

```
public Assignment search(time_limit) {

    population = initializer.initializePopulation

    do {

        mating pool = selector.select
        offspring = recombiner.recombine
        mutator.mutate
        population = replacer.replace

    }
    while(time_spent < time_limit);

    return getFittest

}
```

Benchmark Problem 1 (Uniformly Randomized)

Consider class of randomly generated makespan problems with

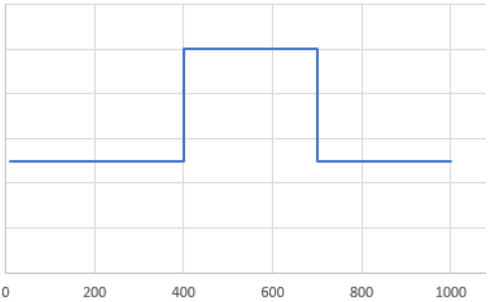
- 20 machines
- 300 jobs overall
- 200 jobs with random processing time between 10 and 1000
- 100 jobs with random processing time between 100 and 300



Benchmark Problem 2 (Uniformly Randomized)

Consider class of randomly generated makespan problems with

- 20 machines
- 300 jobs overall
- 150 jobs with random processing time between 10 and 1000
- 150 jobs with random processing time between 400 and 700



Benchmark Problem 3 (Highly Specialized)

Consider makespan problem with

- 50 machines
- 101 jobs overall
- 3 jobs with processing time 50
- and 2 jobs with processing time $51, 52, 53, \ldots, 99$ each, i.e.,
  - 2 jobs with time 51
  - 2 jobs with time 52
  - 2 jobs with time 53
  - ...
  - 2 jobs with time 99