ISSN: 2087-1716
Jurnal Ilmiah ILKOM Volume 8 Nomor 1 (April 2016)

OPTIMALISASI SOLUSI TERBAIK DENGAN PENERAPAN NON-DOMINATED SORTING II ALGORITHM

Poetri Lestari Lokapitasari Belluano
poe3.setiawan@gmail.com
Universitas Muslim Indonesia

Abstrak

Non Dominated Sorting pada *Genetic Algorithm* merupakan kelas khusus dari algoritma evolusioner dengan menggunakan teknik yang terinspirasi oleh biologi evolusioner seperti warisan, mutasi, seleksi alam dan rekombinasi (*crossover*). Tahap-tahap teknik pencarian untuk menemukan penyelesaian perkiraan pada optimisasi dan masalah pencarian, sebagai solusi hasil akhir. Seleksi non-Dominasi algoritma genetik adalah sebuah algoritma optimasi multi objek/tujuan algoritma dan merupakan contoh dari Algoritma Evolusioner dari bidang Komputasi Evolusioner. NSGA merupakan perpanjangan dari Algoritma Genetika untuk optimasi fungsi tujuan ganda. Hal ini berhubungan dengan evolusioner algoritma Optimasi multi objek lainnya (EMOO) atau Beberapa Algoritma Evolusioner objektif MOEA) seperti Algoritma Vektor-Dievaluasi genetik (VEGA), Algoritma Evolusioner penguatan Pareto (SPEA), dan Strategi Evolusi Pareto Arsip (Paes). Ada dua versi algoritma, yaitu NSGA klasik dan bentuk yang terbaru saat ini kanonik NSGA-II.
Kata kunci: *non-dominated sorting, genetic algorithm*, optimasi multi objek.

1. Pendahuluan

Pemecahan masalah atau pencarian solusi optimal dalam suatu kasus permasalahan, dianggap bukan hal yang mudah. Pada bidang ilmu komputer menawarkan berbagai cara pencarian solusi optimal menggunakan metode algoritma yang berbeda-beda sesuai dengan prosedur dan analisis optimisasi permasalahan.

Algoritma Genetik sebagai cabang dari algoritma Evolusi merupakan metode *adaptive* yang bisa digunakan untuk memecahkan suatu pencarian nilai dalam sebuah masalah optimasi. Algoritma ini didasarkan pada proses genetik yang ada dalam makhluk hidup yaitu perkembangan generasi dalam sebuah populasi yang alami, secara lambat laun mengikuti seleksi alam atau "siapa yang kuat, dia yang bertahan (*survive*)". Dengan meniru teori evolusi ini, algoritma genetik digunakan untuk mencari solusi permasalahan-permasalahan dalam dunia nyata, karena dipercaya memiliki kehandalan dalam menghasilkan output yang optimal baik dari masalah dengan satu variabel bahkan multi-variabel atau multi-objek.

Untuk memecahkan masalah optimisasi dalam algoritma genetik diselesaikan dengan beberapa prosedur dalam pencarian solusi yakni salah satunya dengan metode seleksi (*sorting*) untuk menerapkan pengetahuan tentang struktur dari ruang pencarian agar dapat mengurangi banyaknya waktu yang dipakai dalam pencarian. Sedangkan pada permasalahan yang bersifat multi-variable atau multi-objek dibutuhkan pula suatu metode seleksi pencarian solusi yang tepat yakni NSGA (*Non-Dominated Sorting In Genetic Algorithm*) yang merupakan perpanjangan dari Algoritma Genetika untuk optimasi fungsi tujuan ganda dan mampu meningkatkan penyesuaian adaptif dari populasi kandidat solusi untuk sebuah front Pareto yg dibatasi oleh aturan fungsi obyektif.

2. Landasar Teori

2.1. Metode Heuristik

Merupakan teknik yang digunakan untuk meningkatkan efisiensi dari proses pencarian. Dalam pencarian *state space,* heuristik adalah aturan untuk memilih cabang-cabang yang paling mungkin menyebabkan penyelesaian permasalahan dapat diterima. Pada dasarnya tidak ada langkah-langkah pasti pencarian solusi dalam metode heuristic. Pencarian lebih bersifat intuitif berdasarkan pengamatan terhadap jalannya algoritma terhadap sebuah permasalahan. Namun ada tiga metode dasar metode heuristik yakni dengan prosedur mengurangi kemungkinan solusi, partisi masalah, dan penetapan biaya untuk sub-masalah [4].

1. Pengurangan Kemungkinan Solusi

Ide dasarnya dengan menghilangkan secara dramatis sejumlah kemungkinan solusi. Salah satu pendekatan yang dapat dilakukan adalah memperketat kriteria yang harus dipenuhi di dalam

fungsi kriteria. Bahkan dapatditambahkan fungsi kriteria lainnya untuk pengecekan kemungkinan solusi.

2. Partisi Masalah

Prosedur ini dilakukan dengan selalu berusaha untuk membagi-bagi setiap permasalahan yang ada ke dalam upa masalah yang diasumsikan lebih mudah untuk diselesaikan. Proses partisi dapat dilakukan berdasarkan prioritas masalah atau urutan waktu masalah.

3. Penetapan Biaya untuk Sub-Masalah

Dalam metode ini akan ditambahkan properti biaya ke dalam setiap kemungkinan solusi. Setiap pengambilan keputusan berdasarkan pada fungsi kriteria dan properti biaya selalu dihitung berdasarkan langkah sebelumnya. Biaya yang diambil adalah biaya yang terkecil, dengan harapan solusi akan tercapai dengan biaya yang sekecil-kecilnya.

2.2. Optimasi Fungsi

Mengatasi optimasi masalah pada komputasi Metaheuristik dan Kepakaran dibutuhkan strategi khusus untuk memberikan wawasan umum agar setiap prosedur-prosedur fungsi dapat saling berinteraksi baik secara stokastik maupun melalui optimasi fungsi.

Optimasi fungsi bersumber dari bidang Komputasi Evolusioner, *Swarm Intelligence*, dan terkait Komputasi sub-bidang Kepakaran.

2.3. Algoritma Genetika (*Genetic Algorithm*)

Penciptaan awal dari Algoritma Genetika adalah John Holland. Algoritma Genetika menggunakan analogi secara langsung dari kebiasaan yang alami yaitu seleksi alam, yang bekerja dengan sebuah populasi yang terdiri dari individu-individu. Masing-masing individu mempresentasikan sebuah kemungkinan solusi terhadap suatu persoalan/permasalahan yang ada. Individu tersebut dilambangkan dengan sebuah nilai *fitness* untuk mencari solusi terbaik.

Pertahanan yang tinggi dari individu memberikan kesempatan untuk melakukan reproduksi melalui perkawinan silang dengan individu lain dalam populasi tersebut. Individu baru yang dihasilkan disebut keturunan yang membawa beberapa sifat dari induknya. Sedangkan individu lain yang tidak masuk dalam seleksi dalam lingkup reproduksi akan mati dengan sendirinya, sehingga tidak diperhitungkan menjadi suatu tahapan solusi. Dengan begitu, beberapa generasi dengan karakteristik yang bagus akan bermunculan dalam bentukan populasi tersebut, untuk kemudian dicampur dan ditukar dengan karakter lainnya. Semakin banyak individu yang dikawinkan, maka akan semakin banyak pula kemungkinan ditemukannya solusi terbaik.

Sebelum Algoritma Genetika dapat dijalankan, maka sebuah kode yang sesuai (*representative*) harus dirancang, dimana titik solusi dalam ruang permasalahan dikodekan dalam bentuk kromosom/string yang terdiri atas komponen genetik terkecil yaitu Gen. Dengan teori evolusi dan teori genetika, di dalam penerapan Algoritma Genetika akan melibatkan beberapa operator, yaitu :

1. Operator Evolusi yang melibatkan proses seleksi (*selection*) di dalamnya.
2. Operator Genetika yang melibatkan operator pindah silang (*crossover*) dan mutasi (*mutation*).

Untuk memeriksa hasil optimasi, dibutuhkan *fitness function,* yang menandakan gambaran hasil atau solusi yang telah dikodekan. Selama berjalan, induk harus digunakan untuk reproduksi, pindah silang dan mutasi untuk menciptakan keturunan. Jika Algoritma Genetika didesain secara baik, maka populasi akan mengalami konvergensi dan akan didapatkan sebuah solusi yang optimum.

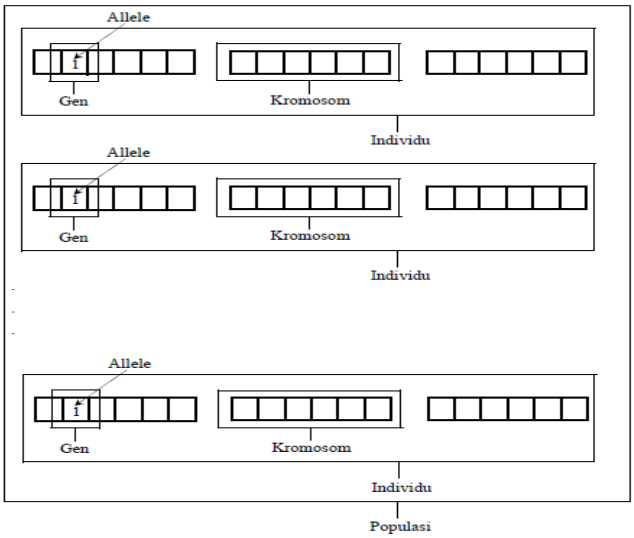
Beberapa hal yang harus dilakukan dalam Algoritma Genetika, adalah:

1. **Memdefinisikan individu**, dimana individu menyatakan salah satu kemungkinan solusi penyelesaian.
2. **Mendefinisikan nilai fitness**, yang merupakan ukuran baik-tidaknya sebuah individu atau baik-tidaknya solusi yang didapatkan. Nilai fitness yang digunakan adalah yang bernilai paling tinggi dari individu yang ada.
3. Menetukan proses **pembangkitan populasi awal**, yang biasanya dilakukan dengan menggunakan pembangkitan acak seperti *random-walk*.
4. Menentukan proses **seleksi** yang akan digunakan.
5. Menentukan proses **perkawinan silang** (*cross-over*) dan **mutasi gen** yang akan digunakan.

Istilah-istilah penting yang perlu diperhatikan dalam mendefinisikan individu, antara lain:

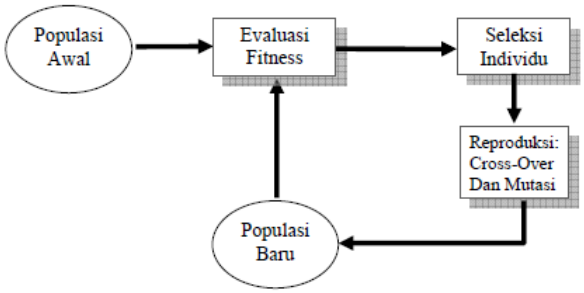
a. Genotype (gen), sebuah nilai yang menyatakan satuan dasar yang membentuk suatu arti tertentu dalam satu kesatuan gen yang dinamakan kromosom. Dalam algoritma genetika, gen bisa berupa nilai biner, float, integer maupun karakter, atau kombinatoral.
b. Allele, adalah nilai dari gen
c. Kromosom, gabungan gen-gen yang membentuk nilai tertentu.

d. Individu, menyatakan satu nilai atau keadaan yang meyatakan salah satu kemungkinan solusi dari permasalahan.
e. Populasi, merupakan sekumpulan individu yang akan diproses bersama dalam satu siklus proses evolusi.
f. Generasi, menyatakan satu siklus proses evolusi atau satu iterasi di dalam algoritma genetika.



Gambar 1. Ilustrasi Representasi Penyelesaian Permasalahan dalam Algoritma Genetika

Siklus dari algoritma Genetika pertama kali ditemukan oleh David Goldberg [1], dimana gambaran siklus dapat dilihat pada Gambar 2.

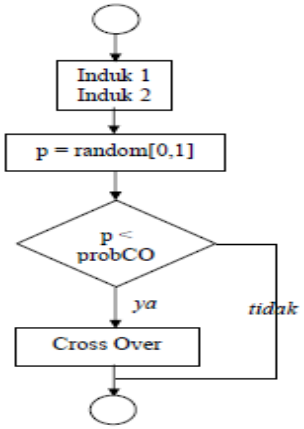


Gambar 2. Siklus Algoritma Genetika oleh David Goldberg

Komponen-komponen utama Algortima Genetika terdiri dari:

1. **Teknik Pengkodean**, yakni bagaimana mengkodekan gen dari kromosom dimana satu gen biasanya akan mewakili satu variabel. Dengan demikian kromosom dapat dipresentasikan dengan menggunakan ;
   - String bit : 10011 dan seterusnya.
   - Array bilangan real : 65.65 ; -67.98 ; 77.34 dan seterusnya.
   - Elemen permutasi : E2, E10, E5 dan seterusnya.
   - Daftar aturan : R1, R2, R3 dan seterusnya.
   - Elemen Program : pemrograman genetika
   - Struktur lainnya.
2. **Membangkitkan Populasi Awal** adalah proses membangkitkan sejumlah individu secara acak atau melalui prosedur tertentu. Ukuran populasi tergantung pada masalah yang akan diselesaikan dan jenis operator genetika yang akan diimplementasikan. Setelah ukuran populasi ditentukan, kemudian dilakukan pembangkitan populasi awal. Syarat-syarat yang harus dipenuhi untuk menunjukkan suatu solusi harus benar-benar diperhatikan dalam pembangkitan setiap individunya.
3. **Seleksi**, digunakan untuk memilih individu-individu mana saja yang akan dipilih untuk proses kawin silang dan mutasi, dan digunakan untuk mendapatkan calon induk/parent yang baik. “Induk yang baik akan menghasilkan keturunan yang baik”. Semakin tinggi nilai fitness suatu individu maka semakin besar pula kemungkinannya akan terpilih.
4. **Pindah Silang (*Cross-Over*)**, adalah operator dari algoritma genetika yang melibatkan dua individu untuk membentuk kromosom baru. Pindah silang menghasilkan titik baru dalam ruang

pencarian yang siap untuk diuji. Operasi ini tidak selalu dilakukan pada semua individu yang ada. Individu dipilih secara acak untuk dilakukan crossing dengan P_C antara 0,6 s/d 0,95. Jika pindah silang tidak dilakukan, maka nilai dari induk akan diturunkan kepada turunannya.



Gambar 3. Flowchart Proses CrossOver

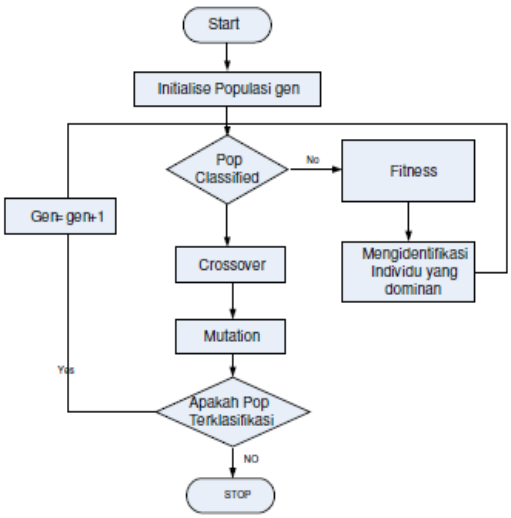
5. **Mutasi**, merupakan operator untuk menggantikan gen yang hilang dari populasi akibat proses seleksi yang memungkinkan munculnya kembali gen yang tidak muncul pada inisialisasi populasi.

2.4. Metode *Non-Dominated Sorting*

Seleksi *non-Dominasi* algoritma genetik adalah sebuah algoritma optimasi multi objek/tujuan algoritma dan merupakan contoh dari Algoritma Evolusioner dari bidang Komputasi Evolusioner. NSGA (Non-Dominated Sorting Genetic Algorithm) merupakan perpanjangan dari Algoritma Genetika untuk optimasi fungsi tujuan ganda. Ada dua versi algoritma yang banyak digunakan saat ini, yaitu NSGA klasik dan bentuk yang terbaru saat ini kanonik NSGA-II.

Tujuan dari algoritma NSGA adalah untuk meningkatkan penyesuaian adaptif dari populasi kandidat solusi untuk sebuah front Pareto yg dibatasi oleh aturan fungsi obyektif. Algoritma ini menggunakan sebuah proses evolusi dengan pengganti operator evolusi termasuk seleksi, crossover genetik, dan mutasi genetik. Populasi diurutkan menjadi hirarki sub-populasi berdasarkan urutan dominasi Pareto. Kesamaan antara anggota dari masing-masing sub-kelompok dievaluasi di bagian depan Pareto, dan kelompok hasl dan ukuran persamaan digunakan untuk mempromosikan sebuah front beragam dri solusi non-dominasi.

Adapun urutan kerja dari NSGA-II terlihat pada Gambar 4 sebagai berikut:



Gambar 4. Flowchart NSGA-II

Algoritma NSGA menyediakan daftar pseudocode dominasi non seleksi Algoritma genetik II (NSGA-II) untuk meminimalkan fungsi biaya. Pada *Sort-ByRank* dan fungsi Jarak (*Distance function order*) dari populasi menjadi sebuah hirarki non-dominasi front Pareto. Himpunan jarak (*The Crowding Distance*) bertugas menghitung jarak rata-rata antara anggota front masing-

masing di bagian front itu sendiri, sesuai referensi Deb dkk untuk sebuah presentasi yg jelas dari Pseudocode dan penjelasan fungsi-fungsi ini [3].

Algorithm : Pseudocode for NSGAII.

```
Input: Population_size, ProblemSize, P_crossover, P_mutation
Output: Children
Population ← InitializePopulation(Population_size, ProblemSize);
EvaluateAgainstObjectiveFunctions(Population);
FastNondominatedSort(Population);
Selected ← SelectParentsByRank(Population, Population_size);
Children ← CrossoverAndMutation(Selected, P_crossover, P_mutation);
while ¬StopCondition() do
    EvaluateAgainstObjectiveFunctions(Children);
    Union ← Merge(Population, Children);
    Fronts ← FastNondominatedSort(Union);
    Parents ← ∅;
    Front_L ← ∅;
    foreach Front_i ∈ Fronts do
        CrowdingDistanceAssignment(Front_i);
        if Size(Parents)+Size(Front_i) > Population_size then
            Front_L ← i;
            Break();
        else
            Parents ← Merge(Parents, Front_i);
        end
    end
    if Size(Parents)<Population_size then
        Front_L ← SortByRankAndDistance(Front_L);
        for P_1 to P_Population_size−Size(Front_L) do
            Parents ← Pi;
        end
    end
    Selected ← SelectParentsByRankAndDistance(Parents,
    Population_size);
    Population ← Children;
    Children ← CrossoverAndMutation(Selected, P_crossover,
    P_mutation);
end
return Children;
```

Crossover dan fungsi Mutasi (*Mutation Function*) melakukan crossover klasik dan operator mutasi genetik dari algoritma genetika. Kedua pilihan antara Parents berdasarkan peringkat dan jarak serta Sort-by berdasarkan peringkat dan Jarak membedakan anggota populasinya yang pertama-tama berdasarkan peringkat (urutan solusi prioritas dominasi bagian front yg dimiliki) lalu kemudian jarak pada front (dihitung dengan fungsi himpunan jarak).

NSGA dirancang agar cocok untuk beberapa fungsi kontinu masalah optimasi kasus objektif. Sebuah representasi biner dapat digunakan bersamaan dengan operator genetik klasik seperti satu titik crossover dan mutasi titik. Sebuah representasi bernilai real dianjurkan untuk fungsi kontinu masalah optimasi, pada gilirannya membutuhkan representasi operator genetik spesial seperti Crossover Biner Simulasi (SBX) dan polynomial mutasi [2].

3. Hasil

Dari prosedur algoritma NSGA-II yang telah dipaparkan pada pembahasan sebelumnya, memberikan contoh seleksi non-dominasi menurut Algoritma Genetik II (NSGA-II) yg dijalankan dalam Bahasa Pemrograman Ruby. Masalah demonstrasi adalah turunan dari beberapa tujuan berkelanjutan fungsi optimasi disebut SCH (masalah pertama [3]).

Prosedur untuk mencari nilai minimum dari dua fungsi:

$$f1 = \sum_{i=1}^{n} x_i^2 \text{ and } f2 = \sum_{i=1}^{n} (x_i - 2)^2$$

$$-10 \leq x_i \leq 10 \text{ and } n = 1.$$

Untuk menentukan solusi paling optimal untuk fungsi tersebut:

$$x \in [0, 2]$$

Algoritma merupakan implementasi dari NSGA-II berdasarkan presentasi oleh Deb dkk [3]. Algoritma ini menggunakan string biner representasi (16 bit untuk setiap objective function parameter) yang diterjemahkan dan *rescaled* ke domain fungsi.

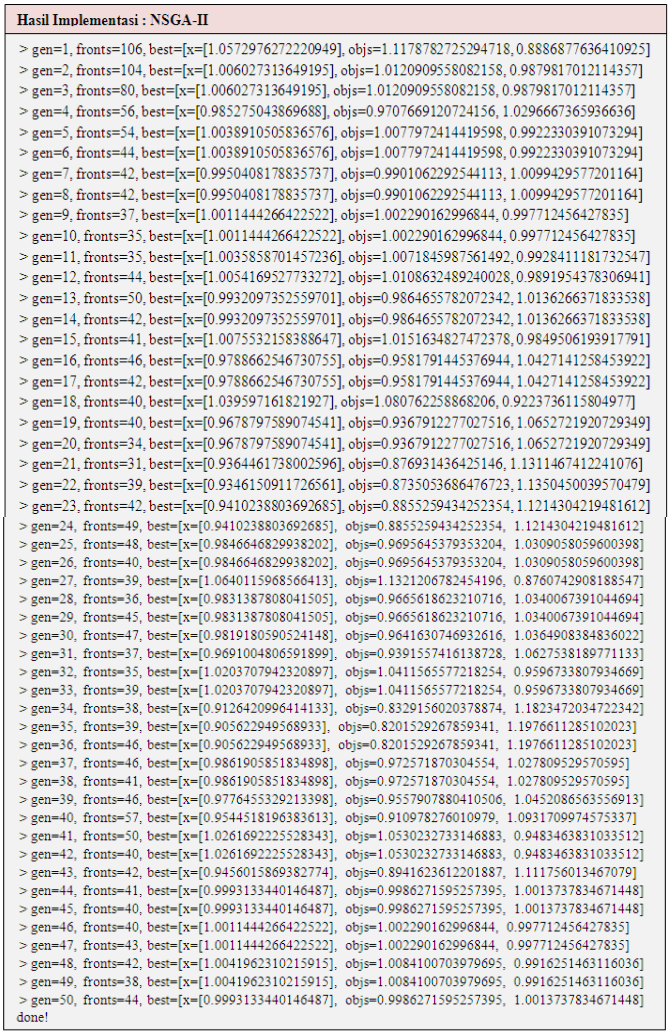
Implementasi ini menggunakan operator *crossover* yg sama dan mutasi titik dengan tingkat mutasi tetap 1/L, di mana ***L*** adalah jumlah bit dalam string biner solusi itu.

Non-Dominated Sorting in Genetic Algorithm dapat diuraikan sesuai daftar Psedocode berikut:

```ruby
def crowded_comparison_operator(x,y)
  return y[:dist]<=>x[:dist] if x[:rank] == y[:rank]
  return x[:rank]<=>y[:rank]
end

def better(x,y)
  if !x[:dist].nil? and x[:rank] == y[:rank]
    return (x[:dist]>y[:dist]) ? x : y
  end
  return (x[:rank]<y[:rank]) ? x : y
end

def select_parents(fronts, pop_size)
  fronts.each {|f| calculate_crowding_distance(f)}
  offspring, last_front = [], 0
  fronts.each do |front|
    break if (offspring.size+front.size) > pop_size
    front.each {|p| offspring << p}
    last_front += 1
  end
  if (remaining = pop_size-offspring.size) > 0
    fronts[last_front].sort! {|x,y| crowded_comparison_operator(x,y)}
    offspring += fronts[last_front][0...remaining]
  end
  return offspring
end

def weighted_sum(x)
  return x[:objectives].inject(0.0) {|sum, x| sum+x}
end

def search(search_space, max_gens, pop_size, p_cross, bits_per_param=16)
  pop = Array.new(pop_size) do |i|
    {:bitstring=>random_bitstring(search_space.size*bits_per_param)}
  end
  calculate_objectives(pop, search_space, bits_per_param)
  fast_nondominated_sort(pop)
  selected = Array.new(pop_size) do
    better(pop[rand(pop_size)], pop[rand(pop_size)])
  end
  children = reproduce(selected, pop_size, p_cross)
  calculate_objectives(children, search_space, bits_per_param)
  max_gens.times do |gen|
    union = pop + children
    fronts = fast_nondominated_sort(union)
    parents = select_parents(fronts, pop_size)
    selected = Array.new(pop_size) do
      better(parents[rand(pop_size)], parents[rand(pop_size)])
    end
    pop = children
    children = reproduce(selected, pop_size, p_cross)
    calculate_objectives(children, search_space, bits_per_param)
    best = parents.sort!{|x,y| weighted_sum(x)<=>weighted_sum(y)}.first
    best_s = "[x=#{best[:vector]}, objs=#{best[:objectives].join(', ')}]"
    puts " > gen=#{gen+1}, fronts=#{fronts.size}, best=#{best_s}"
  end
  union = pop + children
  fronts = fast_nondominated_sort(union)
  parents = select_parents(fronts, pop_size)
  return parents
end

if __FILE__ == $0
  # problem configuration
  problem_size = 1
  search_space = Array.new(problem_size) {|i| [-10, 10]}
  # algorithm configuration
  max_gens = 50
  pop_size = 100
  p_cross = 0.98
  # execute the algorithm
  pop = search(search_space, max_gens, pop_size, p_cross)
  puts "done!"
end
```

```ruby
  return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
end

def point_mutation(bitstring, rate=1.0/bitstring.size)
  child = ""
  bitstring.size.times do |i|
    bit = bitstring[i].chr
    child << ((rand()<rate) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def crossover(parent1, parent2, rate)
  return ""+parent1 if rand()>=rate
  child = ""
  parent1.size.times do |i|
    child << ((rand()<0.5) ? parent1[i].chr : parent2[i].chr)
  end
  return child
end

def reproduce(selected, pop_size, p_cross)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.modulo(2)==0) ? selected[i+1] : selected[i-1]
    p2 = selected[0] if i == selected.size-1
    child = {}
    child[:bitstring] = crossover(p1[:bitstring], p2[:bitstring], p_cross)
    child[:bitstring] = point_mutation(child[:bitstring])
    children << child
    break if children.size >= pop_size
  end
  return children
end

def calculate_objectives(pop, search_space, bits_per_param)
  pop.each do |p|
    p[:vector] = decode(p[:bitstring], search_space, bits_per_param)
    p[:objectives] = [objective1(p[:vector]), objective2(p[:vector])]
  end
end

def dominates(p1, p2)
  p1[:objectives].each_index do |i|
    return false if p1[:objectives][i] > p2[:objectives][i]
  end
  return true
end

def fast_nondominated_sort(pop)
  fronts = Array.new(1){[]}
  pop.each do |p1|
    p1[:dom_count], p1[:dom_set] = 0, []
    pop.each do |p2|
      if dominates(p1, p2)
        p1[:dom_set] << p2
      elsif dominates(p2, p1)
        p1[:dom_count] += 1
      end
    end
    if p1[:dom_count] == 0
      p1[:rank] = 0
      fronts.first << p1
    end
  end
  curr = 0
  begin
    next_front = []
    fronts[curr].each do |p1|
      p1[:dom_set].each do |p2|
        p2[:dom_count] -= 1
        if p2[:dom_count] == 0
          p2[:rank] = (curr+1)
          next_front << p2
        end
      end
    end
    curr += 1
    fronts << next_front if !next_front.empty?
  end while curr < fronts.size
  return fronts
end

def calculate_crowding_distance(pop)
  pop.each {|p| p[:dist] = 0.0}
  num_obs = pop.first[:objectives].size
  num_obs.times do |i|
    min = pop.min{|x,y| x[:objectives][i]<=>y[:objectives][i]}
    max = pop.max{|x,y| x[:objectives][i]<=>y[:objectives][i]}
    rge = max[:objectives][i] - min[:objectives][i]
    pop.first[:dist], pop.last[:dist] = 1.0/0.0, 1.0/0.0
    next if rge == 0.0
    (1...(pop.size-1)).each do |j|
      pop[j][:dist]+=(pop[j+1][:objectives][i]-pop[j-1][:objectives][i])/rge
    end
  end
end
```

Melihat uraian Psedocode tersebut, maka hasil dari metode *Non-Dominated Sorting in Genetic Algorithm* diimplementasikan sebagai berikut:



Hasil Implementasi : NSGA-II

```
> gen=1, fronts=106, best=[x=[1.0572976272220949], objs=1.1178782725294718, 0.8886877636410925]
> gen=2, fronts=104, best=[x=[1.0060273136491951], objs=1.0120909558082158, 0.9879817012114357]
> gen=3, fronts=80, best=[x=[1.0060273136491951], objs=1.0120909558082158, 0.9879817012114357]
> gen=4, fronts=56, best=[x=[0.9852750438696881], objs=0.9707669120724156, 1.0296667365936636]
> gen=5, fronts=54, best=[x=[1.0038910505836576], objs=1.0077972414419598, 0.9922330391073294]
> gen=6, fronts=44, best=[x=[1.0038910505836576], objs=1.0077972414419598, 0.9922330391073294]
> gen=7, fronts=42, best=[x=[0.9950408178835737], objs=0.9901062292544113, 1.0099429577201164]
> gen=8, fronts=42, best=[x=[0.9950408178835737], objs=0.9901062292544113, 1.0099429577201164]
> gen=9, fronts=37, best=[x=[1.0011444266422522], objs=1.002290162996844, 0.997712456427835]
> gen=10, fronts=35, best=[x=[1.0011444266422522], objs=1.002290162996844, 0.997712456427835]
> gen=11, fronts=35, best=[x=[1.0035858701457236], objs=1.0071845987561492, 0.9928411181732547]
> gen=12, fronts=44, best=[x=[1.0054169527733272], objs=1.0108632489240028, 0.9891954378306941]
> gen=13, fronts=50, best=[x=[0.9932097352559701], objs=0.9864655782072342, 1.0136266371833538]
> gen=14, fronts=42, best=[x=[0.9932097352559701], objs=0.9864655782072342, 1.0136266371833538]
> gen=15, fronts=41, best=[x=[1.0075532158388647], objs=1.0151634827472378, 0.9849506193917791]
> gen=16, fronts=46, best=[x=[0.9788662546730755], objs=0.9581791445376944, 1.0427141258453922]
> gen=17, fronts=42, best=[x=[0.9788662546730755], objs=0.9581791445376944, 1.0427141258453922]
> gen=18, fronts=40, best=[x=[1.039597161821927], objs=1.080762258868206, 0.9223736115804977]
> gen=19, fronts=40, best=[x=[0.9678797589074541], objs=0.9367912277027516, 1.0652721920729349]
> gen=20, fronts=34, best=[x=[0.9678797589074541], objs=0.9367912277027516, 1.0652721920729349]
> gen=21, fronts=31, best=[x=[0.9364461738002596], objs=0.876931436425146, 1.1311467412241076]
> gen=22, fronts=39, best=[x=[0.9346150911726561], objs=0.8735053686476723, 1.1350450039570479]
> gen=23, fronts=42, best=[x=[0.9410238803692685], objs=0.8855259434252354, 1.1214304219481612]
> gen=24, fronts=49, best=[x=[0.9410238803692685], objs=0.8855259434252354, 1.1214304219481612]
> gen=25, fronts=48, best=[x=[0.9846646829938202], objs=0.9695645379353204, 1.0309058059600398]
> gen=26, fronts=40, best=[x=[0.9846646829938202], objs=0.9695645379353204, 1.0309058059600398]
> gen=27, fronts=39, best=[x=[1.0640115968566413], objs=1.1321206782454196, 0.8760742908188547]
> gen=28, fronts=36, best=[x=[0.9831387808041505], objs=0.9665618623210716, 1.0340067391044694]
> gen=29, fronts=45, best=[x=[0.9831387808041505], objs=0.9665618623210716, 1.0340067391044694]
> gen=30, fronts=47, best=[x=[0.9819180590524148], objs=0.9641630746932616, 1.0364908384836022]
> gen=31, fronts=37, best=[x=[0.9691004806591899], objs=0.9391557416138728, 1.0627538189771133]
> gen=32, fronts=35, best=[x=[1.0203707942320897], objs=1.0411565577218254, 0.9596733807934669]
> gen=33, fronts=39, best=[x=[1.0203707942320897], objs=1.0411565577218254, 0.9596733807934669]
> gen=34, fronts=38, best=[x=[0.9126420996414133], objs=0.8329156020378874, 1.1823472034722342]
> gen=35, fronts=39, best=[x=[0.905622949568933], objs=0.8201529267859341, 1.1976611285102023]
> gen=36, fronts=46, best=[x=[0.905622949568933], objs=0.8201529267859341, 1.1976611285102023]
> gen=37, fronts=46, best=[x=[0.9861905851834898], objs=0.972571870304554, 1.027809529570595]
> gen=38, fronts=41, best=[x=[0.9861905851834898], objs=0.972571870304554, 1.027809529570595]
> gen=39, fronts=46, best=[x=[0.9776455329213398], objs=0.9557907880410506, 1.0452086563556913]
> gen=40, fronts=57, best=[x=[0.9544518196383613], objs=0.910978276010979, 1.0931709974575337]
> gen=41, fronts=50, best=[x=[1.0261692225528343], objs=1.0530232733146883, 0.9483463831033512]
> gen=42, fronts=40, best=[x=[1.0261692225528343], objs=1.0530232733146883, 0.9483463831033512]
> gen=43, fronts=42, best=[x=[0.9456015869382774], objs=0.8941623612201887, 1.111756013467079]
> gen=44, fronts=41, best=[x=[0.9993133440146487], objs=0.9986271595257395, 1.0013737834671448]
> gen=45, fronts=40, best=[x=[0.9993133440146487], objs=0.9986271595257395, 1.0013737834671448]
> gen=46, fronts=40, best=[x=[1.0011444266422522], objs=1.002290162996844, 0.997712456427835]
> gen=47, fronts=43, best=[x=[1.0011444266422522], objs=1.002290162996844, 0.997712456427835]
> gen=48, fronts=42, best=[x=[1.0041962310215915], objs=1.0084100703979695, 0.9916251463116036]
> gen=49, fronts=38, best=[x=[1.0041962310215915], objs=1.0084100703979695, 0.9916251463116036]
> gen=50, fronts=44, best=[x=[0.9993133440146487], objs=0.9986271595257395, 1.0013737834671448]
done!
```

Gambar 5. Implementasi Ruby Psedocode untuk NSGA-II

Berdasarkan implementasi dari psedocode NSGA-II terdapat beberapa komponen-komponen yang perlu diketahui, yakni:

- Gen, merupakan bentuk kromosom dari hasil mutasi genetik.
- Fronts, adalah nilai perbandingan antara dua fungsi yang dilaksanakan secara rekursif sehingga mendapatkan nilai random.
- Best, dengan kode **X** menjadi nilai akhir dari solusi paling optimal dari urutan nilai antara objek awal hingga objek akhir yang diperoleh dari proses permutasi dalam lingkup populasi.
- Objs, adalah objek untuk mengurutkan nilai terbaik yang diperoleh dari hasil permutasi.

Melihat implementasi tersebut, dijelaskan bahwa telah terjadi mutasi sejumlah individu gen/kromosom sebanyak 50 gen yang masing-masing menunjukkan nilai fitness pada fronts dengan nilai random yang diproses secara rekursif. Tampak ada beberapa Gen yang bermutasi menghasilkan fronts yang bernilai sama. Hal ini membuktikan bahwa telah terjadi mutasi gen dengan mendapatkan kemungkinan solusi optimasi dan menghasilkan individu baru sebagai parents, berdasarkan perbandingan dari dua fungsi objek hingga nilai **X** ditetapkan sebagai solusi terbaik.

4. Kesimpulan dan Saran

Metode Non-Dominated Sorting digunakan untuk mendapatkan nilai solusi yang terbaik berdasarkan nilai kemungkinan dari suatu bentuk permasalahan yakni dengan melaksanakan proses pengurutan data/nilai yang kemudian diseleksi. Data yang diseleksi merupakan gen/kromosom yang telah ditetapkan sebagai individu-individu pilihan untuk proses kawin silang dan mutasi, dan digunakan untuk mendapatkan calon induk/parent yang terbaik. Semakin tinggi nilai fitness suatu individu maka semakin besar pula kemungkinannya akan terpilih sebagai individu kromosom baru.

Metode NSGA-II ini diterapkan menggunakan algoritma genetika yang melibatkan dua individu untuk membentuk kromosom baru. Operasi Pindah silang akan menghasilkan titik baru dalam ruang

pencarian yang siap untuk diuji. Individu dipilih secara acak untuk dilakukan operasi cross-over. Jika pindah silang tidak dilakukan, maka nilai dari induk akan diturunkan kepada turunannya.

Sedangkan proses permutasi yang berfungsi sebagai operator akan dilaksanakan melalui beberapa prosedur untuk menggantikan gen yang hilang dari populasi akibat proses seleksi yang memungkinkan munculnya kembali gen yang tidak muncul pada inisialisasi populasi.

Pada dasarnya Algoritma NSGA-II ini berfungsi untuk meminimalkan fungsi biaya dengan menetukan *Sort-ByRank* dan fungsi Jarak (*Distance function order*) dari populasi sehingga menjadi sebuah hirarki non-dominasi front Pareto. Himpunan jarak (*The Crowding Distance*) bertugas menghitung jarak rata-rata antara anggota front masing-masing di bagian front itu sendiri. Sehingga untuk penelitian berikutnya dapat mengimplementasikan metode NSGA-II menggunakan objek nyata dalam menemukan optimasi solusi yang terbaik.

Daftar Pustaka

[1] _____ 2011. Algoritma Genetika. Tersdia online: lecturer.eepis-its.edu, diakses: Desember 2011, pukul 17.11

[2] K.Deb and R.B. Argawal. 1995. Simulated Binary crossover for countinuous search space. Complex systems, 9:115-148.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA–II. IEEE Transactions on Evolutionary Computation, 6(2):182–197.

[4] _____ 2012. Strategi Algoritmik, Laboratorium Ilmu dan Rekayasa Komputasi, Departemen Teknik Informatika, Institut Teknologi Bandung. www.informatika.org, diakses: januari 2012, pukul 10.40.