ALGORITMA MEMETIKA DAN GRASP UNTUK MENYELESAIKAN *PERMUTATION FLOW SHOP SCHEDULING PROBLEM*

NOLA MARINA

030401705Y

UNIVERSITAS INDONESIA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

DEPARTEMEN MATEMATIKA

DEPOK

2008

ALGORITMA MEMETIKA DAN GRASP UNTUK MENYELESAIKAN *PERMUTATION FLOW SHOP SCHEDULING PROBLEM*

Skripsi diajukan sebagai salah satu syarat
Untuk memperoleh gelar Sarjana Sains

Oleh:
NOLA MARINA
030401705Y

DEPOK
2008

SKRIPSI : ALGORITMA MEMETIKA DAN GRASP UNTUK MENYELESAIKAN *PERMUTATION FLOW SHOP SCHEDULING PROBLEM*

NAMA : NOLA MARINA

NPM : 030401705Y

SKRIPSI INI TELAH DIPERIKSA

DEPOK, 17 JULI 2008

DR. ZUHERMAN RUSTAM,DEA
PEMBIMBING I

DR. YUDI SATRIA,MT
PEMBIMBING II

Tanggal lulus Ujian Sidang Sarjana : 17 Juli 2008

Penguji I : Dr Zuherman Rustam, DEA.

Penguji II : Dra Suarsih Utama.

Penguji III : Dra Siti Nurrohmah, M.Si.

KATA PENGANTAR

Alhamdulillahi rabbil ‘aalamiin. Puji syukur hanya kepada ALLAH SWT, Yang Maha Pengasih, sehingga skripsi ini dapat diselesaikan dengan baik. Shalawat dan salam kerinduan kepada sebaik-baik teladan Rasulullah SAW.

Selesainya skripsi ini tidak terlepas dari bantuan, bimbingan, dorongan, dan do’a yang tulus dari banyak pihak. Penulis mengucapkan terima kasih kepada Bapak Dr. Zuherman Rustam , DEA dan Dr. Yudi Satria ,MT, selaku dosen pembimbing yang telah bersedia meluangkan waktu untuk memberikan pengarahan, diskusi dan bimbingan serta persetujuan sehingga skripsi ini dapat selesai dengan baik.

Terimakasih juga untuk seluruh dosen dan karyawan departemen Matematika atas segala ilmu dan bantuan teknis yang penulis peroleh selama menjadi mahasiswa Matematika UI.

Terkhusus untuk kedua orangtua penulis, yang telah mendidik penulis dengan cara istimewa. Terimakasih atas kepercayaan dan do’a tiada henti yang kalian berikan. Semoga Allah senantiasa menjaga kalian. Salam sayang untuk Uda Wel, Sil, dan Ayub. Juga kepada keluarga besar penulis yang sangat mendukung dan memotivasi penulis.

Untuk warga ‘*Happy family’*, terimakasih atas pengertian, pengorbanan dan keriangan yang senantiasa menyambut penulis di kontrakan tercinta. Adel, Fina, Ira, Lisa, dan fatma, kebersamaan ini tak kan terlupa.

Terimakasih untuk tim yang kompak selalu, Kak TB, Lismanto, dan Luqita. Tanpa kalian, semua ini sangat terasa berat.

Tak lupa penulis juga ucapkan terimakasih kepada Kak Doddy 00 dan Kak Onggo 02 atas bimbingan dan keikhlasannya untuk diganggu oleh penulis.

Spesial buat teman-teman angkatan 2004. Ajat, Avi, Bong, Dewi, Dina, Edi, Ega, Eny, Erma, Echa, Ias, Iif, Intan, Wanto, Ivan, Johan, Milka, Lisa, Murni, Nadya, Nabung, Novi, Nuri, Mita, Valdo, Reza, Riska, Rimbun, Manap, Siska, Spina, Handi, Vajar, Rini, Harry, Leli. Terimakasih telah menjadi teman 'belajar' penulis selama 4 tahun. Ganbatte kudasai...

Terakhir, terimakasih untuk semua pribadi yang secara sadar ataupun tidak telah menjadi 'guru' dalam kehidupan penulis. Semoga Allah mengganjar setiap keikhlasan dari setiap amal shaleh kalian.

Semoga skripsi ini dapat berguna bagi siapa saja yang mengkajinya, serta dapat dikembangkan dan disempurnakan agar lebih bermanfaat untuk kepentingan orang banyak.

"Barangsiapa yang menjadikan akhirat sebagai harapannya, maka Allah akan memberikan kepuasan dalam hatinya, menghimpun sgala impiannya, dan dunia akan mendatanginya. Dan barangsiapa yang dunia sebagai cita-citanya, maka Allah akan menjadikan kemiskinan di depan matanya, membuyarkan sgala impiannya, dan dunia tak akan mendatanginya melainkan apa yang tlah ditentukan baginya. [HR. Tirmidzi]"

Nola Marina

2008

ABSTRAK

Flowshop Scheduling Problem (FSP) adalah masalah penjadwalan yang berkaitan dengan pengurutan pemrosesan *n* pekerjaan pada *m* mesin, dimana setiap pekerjaan harus diproses tepat satu kali pada setiap mesin dalam urutan yang sama, dengan waktu proses tertentu. *Permutation Flowshop Scheduling Problem* (PFSP) adalah kasus khusus dalam FSP, dimana *n* pekerjaan diproses dalam urutan yang sama pada setiap mesin.

Pada tugas akhir ini akan dilihat kinerja kombinasi Algoritma Memetika (AM) dan metode *Greedy Randomized Adaptive Search Procedure* (GRASP) dalam menyelesaikan PFSP dengan tujuan meminimumkan *makespan*. Kinerja metode AM dan GRASP dilihat dari kedekatan solusi yang dihasilkan dengan *Best Known Solution* (BKS) pada *Taillard's Benchmark* dan dari waktu komputasinya. Berdasarkan pengujian, disimpulkan bahwa metode AM dan GRASP cukup kompetitif dalam meyelesaikan PFSP dengan *error* relatif tidak lebih dari 2 %. Selain itu, metode AM dan GRASP lebih cepat konvergen ke solusi optimal dibandingkan dengan metode AM dan metode GRASP sendiri-sendiri.

Kata kunci: GRASP; *Iterated Local Search*; Algoritma Memetika; *Permutation Flowshop Scheduling Problem*

ix + 68 hlm.; lamp.

Bibliografi : 11(1996 - 2007)

DAFTAR ISI

KATA PENGANTAR.....i

ABSTRAK.....iii

DAFTAR ISI.....iv

DAFTAR GAMBAR.....vi

DAFTAR TABEL.....vii

DAFTAR LAMPIRAN.....viii

DAFTAR SINGKATAN.....ix

BAB I PENDAHULUAN.....1

1.1 Latar Belakang.....1

1.2 Tujuan.....5

1.3 Pembatasan Masalah.....5

1.4 Sistematika Penulisan.....5

BAB II LANDASAN TEORI.....7

2.1 *Shop Scheduling Problem* (SP).....7

2.2 *Permutation Flow Shop Scheduling Problem* (PFSP).....10

2.3 GRASP.....16

2.3.1 NEH.....17

2.3.2 *Iterated Local Search* (ILS).....19

2.3.3 *Path Relinking* (PR)…….....21

2.4 Algoritma Memetika(AM).....24

2.4.1 Skema Pengkodean Kromosom.....28

2.4.2 Pembentukan Populasi…….....30

2.4.3 *Local Search* ……..31

2.4.4 Evaluasi Nilai *Fitness*.....32

2.4.5 Seleksi.....33

2.4.6 *Crossover*.....35

2.4.7 Mutasi.....38

BAB III PENERAPAN ALGORITMA MEMETIKA DAN GRASP DALAM MENYELESAIKAN PFSP…….....41

3.1 GRASP Dalam Menyelesaikan PFSP.....42

3.2 AM Dalam Meyelesaikan PFSP.....45

3.2.1 Skema Pengkodean Kromosom.....45

3.2.2 Pembentukan Populasi Awal.....46

3.2.3 *Local Search* ……..46

3.2.4 Evaluasi Nilai *Fitness*.....47

3.2.5 Seleksi.....48

3.2.6 *Path Crossover*(PX).....49

3.2.7 Mutasi.....51

3.2.8 *Cold Restart*.....51

BAB IV IMPLEMENTASI DAN PENGUJIAN.....55

4.1 Implementasi.....55

4.2 Hasil Percobaan…….58

4.3 Hasil Perbandingan.....62

BAB V KESIMPULAN.....65

DAFTAR ACUAN.....67

DAFTAR GAMBAR

Gambar 2.1 Jenis grafik *Gantt Chart*……..……..13

Gambar 2.2 Contoh perhitungan *makespan* dengan *Machine Oriented Gantt Chart*.....14

Gambar 2.3 Contoh pembentukan solusi NEH.....18

Gambar 2.4 Prinsip ILS.....21

Gambar 2.5 Contoh prosedur PR.....23

Gambar 2.6 *Flowchart* AM.....27

Gambar 3.1 *Flowchart* AM dan GRASP.....41

Gambar 3.2 *Roulette wheel* 49

Gambar 3.3 Contoh prosedur *Path Crossover*.....51

Gambar 4.1 Tampilan output program untuk masalah 'contoh'.....58

Gambar 4.2 Tampilan output grafik hasil percobaan terbaik masalah ta03(a), ta08(b), dan ta33(c).....61

Gambar 4.3 Perbandingan *error* relatif metode AM, metode GRASP dengan metode AM dan GRASP untuk data ta03, ta08, dan ta33…….63

DAFTAR TABEL

Tabel 1.1 Waktu proses untuk pekerjaan pada tiap mesin…….. 14

Tabel 3.1 Perhitungan nilai *fitness*.....47

Tabel 3.2 *Makespan*, nilai *fitness* dan persentase nilai *fitness*.....48

Tabel 4.1 Nilai *makespan* dan waktu komputasi dari hasil percobaan AM dan GRASP dalam menyelesaikan PFSP.....60

Tabel 4.2 Nilai *makespan* dari hasil percobaan metode AM dan metode GRASP dalam menyelesaikan PFSP.....63

DAFTAR LAMPIRAN

1. Contoh Data Masalah Penguji.....69

2. Listing Program AM Dan GRASP.....70

DAFTAR SINGKATAN

SP	*Shop scheduling Problem*
FSP	*Flow Shop scheduling Problem*
PFSP	*Permutation Flow Shop scheduling Problem*
GRASP	*Greedy Randomized Adaptive Search Procedure*
AG	Algoritma Genetika
AM	Algoritma Memetika
NEH	Nawaz, Enscore, and Ham
ILS	*Iterated Local Search*
LS	*Local Search*
PR	*Path Relinking*
BKS	*Best Known Solution*

BAB I

PENDAHULUAN

1.1 Latar Belakang

Dalam bidang industri, keuangan, perdagangan dan berbagai aktivitas lain dibutuhkan suatu perencanaan yang baik. Paling tidak untuk memperoleh keuntungan yang tinggi, kualitas yang baik, biaya yang rendah, atau efisiensi sumber daya. Masalah penjadwalan menjadi aspek yang sangat penting, karena penjadwalan merupakan salah satu elemen perencanaan.

Penjadwalan secara umum merupakan suatu proses pengaturan sumber daya yang tersedia untuk menyelesaikan suatu pekerjaan, agar tujuan yang ingin dicapai terpenuhi.

Salah satu masalah penjadwalan yang sering muncul adalah *Shop scheduling Problem* (SP), yaitu masalah penjadwalan yang berkaitan dengan pengurutan pemrosesan sejumlah pekerjaan pada sejumlah mesin. SP merupakan masalah optimisasi yang memiliki karakteristik, antara lain terdiri dari *m* mesin dan *n* pekerjaan. Setiap pekerjaan harus diproses pada setiap mesin. Masing-masing mesin dapat memroses paling banyak satu pekerjaan pada suatu waktu. Setiap pekerjaan harus diproses pada suatu mesin selama suatu periode waktu tertentu tanpa interupsi. Dengan demikian dua pekerjaan

tidak bisa dijadwalkan pada waktu yang sama jika keduanya memerlukan mesin yang sama. Setiap pekerjaan hanya dapat diproses oleh satu mesin dalam satu waktu.

Tujuan dari SP antara lain untuk meminimumkan waktu tunggu pekerjaan, meminimumkan waktu senggang mesin, dan atau meminimumkan keseluruhan waktu penyelesaian dari semua pekerjaan (*makespan*).

Flow Shop scheduling Problem (FSP) adalah salah satu kelas dari SP yang memiliki syarat, setiap pekerjaan harus diproses tepat satu kali pada setiap mesin dengan urutan pemrosesan yang sama.

Pada FSP terdapat kasus khusus dengan syarat tambahan, yaitu urutan pemrosesan pekerjaan pada setiap mesin juga harus sama, yang disebut dengan *Permutation Flow Shop scheduling Problem* (PFSP). PFSP merupakan masalah kombinatorial yang cukup rumit untuk diselesaikan. PFSP juga telah dikenal sebagai masalah *NP-hard*, jika mesin yang digunakan lebih dari 3 [RAV06]. Sehingga, metode eksak seperti *dynamic programming, integer programming,* dan *mixed integer programming* tidak mampu menyelesaikan masalah ini. Untuk itu, masalah PFSP lebih efektif diselesaikan menggunakan metode heuristik, karena kemampuannya dalam mendapatkan solusi yang mendekati global optimal dengan waktu komputasi yang relatif singkat.

Metode heuristik yang telah digunakan dalam penelitian- penelitian terkait PFSP antara lain *local search, greedy randomized adaptive search procedure* (GRASP)*, ant system, tabu search, simulated annealing*, algoritma

genetika, dan algoritma memetika. Algoritma memetika telah terbukti lebih efektif diantara metode-metode tersebut [RMA06].

Jadwal yang baik diperoleh dari algoritma yang baik pula, maka perlu untuk menentukan suatu algoritma yang tepat untuk suatu masalah penjadwalan. Hingga saat ini, berbagai penelitian terkait PFSP masih terus dikembangkan untuk menemukan metode yang semakin baik untuk menyelesaikan PFSP.

Pada tugas akhir ini akan dibahas mengenai PFSP yang akan diselesaikan dengan menggunakan kombinasi Algoritma Memetika (AM) dengan metode GRASP yang diperkenalkan oleh Ravetti et al [RAV06]. Ide dari metode ini adalah metode GRASP membentuk kumpulan solusi yang cukup baik terlebih dahulu, untuk kemudian digunakan oleh AM sebagai populasi awal, sehingga AM hanya memerlukan operator yang sederhana dan iterasi yang sedikit untuk konvergen lebih cepat ke solusi optimal.

GRASP adalah metode heuristik yang menggunakan proses iteratif dalam pencarian solusi. Ide dari GRASP adalah setiap iterasinya terdiri dari 2 tahap, yaitu pembentukan solusi awal dan peningkatan kualitas solusi dengan menerapkan metode *Iterated Local Search* (ILS) dan *Path Relinking* (PR). Solusi yang memenuhi kriteria pada setiap iterasinya, akan dimasukkan ke dalam kumpulan solusi yang disebut ‘solusi elit’ atau *Pool*.

ILS adalah metode heuristik yang secara iteratif melakukan perbaikan terhadap solusi sekarang (*current solution*) dengan cara menerapkan *Local Search* terhadap modifikasi dari solusi sekarang.

PR adalah metode heuristik untuk meningkatkan kualitas solusi dengan cara mengeksplorasi subruang solusi atau *path* diantara dua solusi yang baik yang sudah ada.

AM merupakan penggabungan dari Algoritma Genetika dengan *Local search*. Algoritma Genetika (AG) adalah metode heuristik yang banyak digunakan untuk menyelesaikan masalah optimisasi yang berdasarkan pada proses evolusi biologi. AG diajukan oleh John Holland pada tahun 1975. Kandidat solusi dari suatu masalah direpresentasikan sebagai sekumpulan gen yang disebut kromosom. AG diawali dengan pembentukan kumpulan koromosom yang disebut dengan populasi. Masing-masing kromosom pada populasi akan dievaluasi menggunakan fungsi *fitness*. Kemudian secara iteratif akan dibentuk populasi baru dengan menerapkan operator-operator genetika hingga mencapai kriteria berhenti. Operator dasar pada AG adalah seleksi, *crossover,* dan mutasi.

Local Search pada AM bertujuan untuk melakukan perbaikan lokal yang dapat diterapkan sebelum dan atau sesudah proses seleksi, *cross over* dan mutasi.

Untuk menguji kinerja metode AM dan GRASP dalam menyelesaikan PFSP, digunakan data penguji yang diambil dari *Taillard's Benchmark*. Kinerja metode AM dan GRASP akan dilihat dari kedekatan solusi yang dihasilkan dengan *Best Known Solution* (BKS) pada *Taillard's Benchmark* dan dari waktu komputasinya.

1.2 Tujuan

Tugas akhir ini bertujuan untuk melihat kinerja metode AM dan GRASP dalam menyelesaikan PFSP, diukur dari kedekatan solusi yang diperoleh dengan BKS dari *Taillard's Benchmark* dan dari waktu komputasinya.

1.3 Pembatasan Masalah

Pembatasan masalah dalam penulisan skripsi ini adalah:

- Permasalahan yang dibahas dibatasi hanya untuk PFSP dengan fungsi tujuan meminimumkan *makespan*.
- Data masalah yang digunakan untuk pengujian diambil dari *Taillard's Benchmark* dengan ukuran (format: jumlah pekerjaan x jumlah mesin) 20 x 5, 20 x10, 20 x 20, 50 x 5, 50 x 10, dan 100 x 5.

1.4 Sistematika Penulisan

Skripsi ini terdiri dari 5 bab dengan sistematika penulisan:
Pada BAB I dibahas tentang latar belakang, tujuan, pembatasan masalah, dan sistematika penulisan. Teori tentang SP secara umum, PFSP, metode GRASP , ILS, PR dan AM dijelaskan pada BAB II. Kemudian, pada BAB III

dibahas tentang teori dan prosedur AM dan GRASP, khusus untuk menyelesaikan PFSP. BAB IV berisi implementasi dari AM dan GRASP dalam menyelesaikan PFSP beserta hasil pengujian dan analisanya. Dan BAB V merupakan bagian akhir dari tugas akhir ini yang memuat pernyataan singkat dan tepat yang dijabarkan dari hasil studi literatur dan pengujian.

BAB II

LANDASAN TEORI

Pada Bab II dijelaskan landasan teori yang digunakan untuk mendukung tugas akhir ini. Subbab 2.1 membahas teori SP secara umum, kemudian Subbab 2.2 lebih khusus membahas PFSP. Pada Subbab 2.3 dan 2.4 dibahas landasan teori dari metode yang digunakan untuk menyelesaikan masalah pada tugas akhir ini, yaitu GRASP dan Algoritma Memetika.

2.1 *Shop Scheduling Problem (*SP)

Penjadwalan secara umum merupakan suatu proses pengaturan sumber daya yang tersedia untuk menyelesaikan suatu himpunan pekerjaan dalam periode waktu tertentu. Penjadwalan dapat muncul dalam berbagai bidang, seperti penjadwalan produksi, penjadwalan tenaga kerja, perencanaan transportasi, logistik, komunikasi, *computer design*, dan lain-lain. Tujuan penjadwalan adalah untuk mengoptimalkan satu atau beberapa fungsi tujuan, dimana fungsi tujuannya bisa bermacam-macam tergantung dari aplikasi masalahnya. Sebagai contoh, dalam lingkungan industri, fungsi tujuan biasanya berupa total waktu proses. Kegagalan dalam membuat jadwal atau menjalankan suatu jadwal yang salah dapat memperlambat selesainya pekerjaan dan menghabiskan biaya yang banyak. Untuk itu,

masalah penjadwalan menjadi aspek yang sangat penting, karena penjadwalan merupakan salah satu elemen perencanaan agar tercapai efisiensi dalam suatu pekerjaan.

Disiplin ilmu tentang penjadwalan di bidang industri muncul pada awal abad ke- 20, pertama kali diperkenalkan oleh Henry Gantt.

Salah satu masalah penjadwalan yang sering muncul adalah SP, yaitu masalah penjadwalan yang berkaitan dengan pengurutan pemrosesan sejumlah pekerjaan pada sejumlah mesin. SP merupakan masalah optimisasi yang memiliki karakteristik sebagai berikut:

- Terdiri dari *m* mesin dan *n* pekerjaan.
- Setiap pekerjaan harus diproses pada setiap mesin.
- Masing-masing mesin dapat memroses paling banyak satu pekerjaan pada suatu waktu.
- Setiap pekerjaan harus diproses pada suatu mesin selama suatu periode waktu tertentu tanpa interupsi.
- Setiap pekerjaan hanya dapat diproses oleh satu mesin dalam satu waktu.

Tujuan dari SP antara lain untuk meminimumkan waktu tunggu pekerjaan, meminimumkan waktu senggang mesin, dan meminimumkan keseluruhan waktu penyelesaian dari semua pekerjaan (*makespan*).

Secara umum, SP terdiri dari beberapa kelas, diantaranya:

- *Job Shop scheduling Problem* (JSP)

  Setiap pekerjaan harus diproses tepat satu kali pada mesin sesuai dengan urutannya masing-masing.

- *Open Shop scheduling Problem* (OSP)

  Urutan pemrosesan setiap pekerjaan tidak diberikan, sehingga tujuannya adalah mencari urutan yang tepat untuk masing-masing pekerjaan.

- *Flow Shop scheduling Problem* (FSP)

  Setiap pekerjaan harus diproses tepat satu kali pada setiap mesin, dimana urutan mesin yang dilalui setiap pekerjaan harus sama.

NP-Hard adalah suatu kelompok masalah dimana tidak ada algoritma yang dapat menemukan solusi optimal untuk masalah tersebut dalam waktu polynomial. Kalaupun ada algoritma yang dapat menyelesaikan secara eksak, ini akan menghabiskan waktu yang sangat lama , terutama ketika ukuran masalah semakin besar. Sehingga sebagai alternatifnya disarankan untuk menggunakan metode heuristik yang efisien untuk menemukan solusi yang cukup baik.

FSP termasuk kedalam kelas *NP-Hard* jika mesin yang digunakan lebih dari 3 [RAV06]. Metode eksak seperti *dynamic programming, integer programming, dan mixed integer programming* tidak mampu menyelesaikan masalah ini, kecuali untuk ukuran masalah yang kecil. Untuk itu, masalah FSP lebih efektif diselesaikan dengan menggunakan metode heuristik.

Metode heuristik adalah metode yang mulai dari sebuah atau sekumpulan solusi awal, kemudian melakukan pencarian terhadap solusi yang lebih baik, hingga mendekati solusi optimal. Metode ini baik digunakan untuk menyelesaikan masalah optimisasi kombinatorial yang rumit, dimana algoritma eksak sudah tidak mampu menyelesaikannya atau masalahnya sulit untuk dipahami dan diformulasikan. Kelebihan metode heuristik adalah tidak perlu menganalisa semua kemungkinan solusi untuk mendapatkan solusi optimal seperti yang dilakukan oleh algoritma eksak dan mampu mendapatkan solusi yang mendekati solusi optimal dengan waktu komputasi yang relatif singkat.

Pada FSP terdapat kasus khusus dengan syarat tambahan, yaitu urutan pemrosesan pekerjaan pada setiap mesin juga harus sama, yang disebut dengan *Permutation Permutation Flow Shop scheduling Problem* (PFSP). PFSP adalah masalah yang akan diselesaikan pada tugas akhir ini, yang akan dijelaskan pada Subbab 2.2.

2.2 *Permutation Flow Shop Scheduling Problem* (PFSP)

PFSP adalah salah satu kasus dalam SP, dimana *n* pekerjaan harus diproses pada *m* mesin. Setiap pekerjaan melalui *m* mesin dengan urutan yang sama, yaitu melalui mesin *1*, kemudian mesin *2*, dan seterusnya hingga terakhir mesin *m*. Dan *n* pekerjaan diproses dalam urutan yang sama pada setiap mesin. Karena pada PFSP diketahui bahwa pemrosesan *n* pekerjaan

pada setiap mesin memiliki urutan yang sama, maka cara penempatan pekerjaan pada setiap mesin adalah sama, yaitu *n!*. Sehingga ruang solusi untuk PFSP adalah sebesar *n!*.

Berdasarkan hal diatas, kandidat solusi dari PFSP adalah jadwal yang direpresentasikan oleh urutan/barisan *n* pekerjaan. Misalkan, jika ada *5* pekerjaan dan *3* mesin, maka salah satu jadwal yang dapat dibentuk adalah:

1-3-4-2-5

Representasi ini menunjukkan pekerjaan *1* diproses pertama kali pada semua mesin dimulai dari mesin *1*, mesin *2*, dan mesin*3*. Kemudian dilanjutkan dengan pekerjaan *3* pada mesin *1*, mesin *2*, dan mesin *3*, jika pekerjaan *1* telah selesai diproses pada mesin tersebut, demikian seterusnya hingga pekerjaan *5* selesai pada mesin *3*.

Beberapa asumsi yang dipakai dalam PFSP ini adalah:

- Jumlah pekerjaan *n* dan jumlah mesin *m* berhingga.
- Setiap pekerjaan *independent* satu sama lain.
- Waktu proses pekerjaan *i* pada mesin *j* diberikan.
- Setiap mesin dalam keadaan siap dipakai dan selalu ada selama periode penjadwalan.
- Waktu *set up* termasuk dalam waktu proses, dengan kata lain diabaikan.
- Sumber daya selalu tersedia selama periode penjadwalan

Tujuan dari PFSP yang dibahas pada tugas akhir ini adalah menemukan urutan *n* pekerjaan yang meminimumkan total waktu penyelesaian semua pekerjaan atau m*akespan* yang dilambangkan dengan ***C_{max}***.

PFSP dapat dimodelkan dalam bentuk model matematis sebagai berikut :

Fungsi tujuan : meminimumkan $C_{\max}(\pi)$

$C_{\max}(\pi) = C(\pi_n, m)$

Didapatkan dari fungsi rekursif berikut:

$$C(\pi_1,1) = p(\pi_1,1)$$
$$C(\pi_i,1) = C(\pi_{i-1},1) + p(\pi_i,1)$$
$$C(\pi_1, j) = C(\pi_1, j-1) + p(\pi_1, j) \qquad i = 2,..,n$$
$$C(\pi_i, j) = \max\{C(\pi_{i-1}, j), C(\pi_i, j-1)\} + p(\pi_i, j) \qquad j = 2,..,m$$

Tujuan PFSP adalah menemukan permutasi π^* dalam $\prod$ atau himpunan semua permutasi π, sedemikian sehingga meminimumkan *makespan* atau :

$$C_{\max}(\pi^*) \le C_{\max}(\pi) \qquad \forall \pi \in \prod$$

Indeks pekerjaan = i , i =1,..,n

Indeks mesin = j, j = 1,…,m

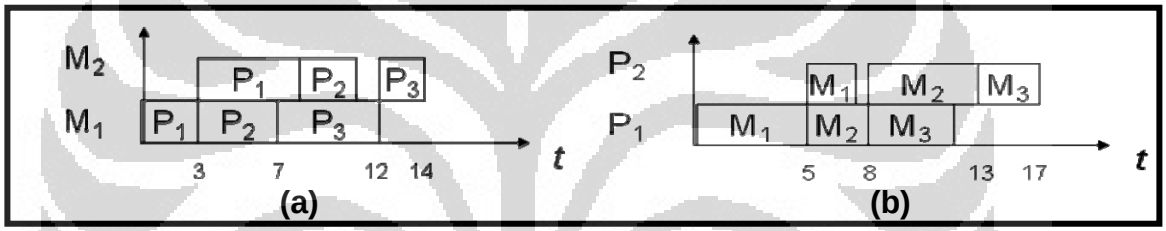
p(i,j) = waktu proses pekerjaan i pada mesin j

π_i . = pekerjaan pada posisi ke i, i = 1,….,n

π = $\{\pi_1, \pi_2, .., \pi_n\}$ = Urutan pekerjaan

. $C(\pi_i, j)$ = waktu penyelesaian hingga selesai pekerjaan pada posisi ke *i* pada mesin *j*. [SED07]

Untuk mempermudah perhitungan *makespan* suatu jadwal, digunakan grafik *Gantt Chart* , yang diperkenalkan oleh Henry Gantt. *Gantt Chart* terdiri dari 2 jenis, yaitu *Machine Oriented Gantt Chart* (Gambar 2.1 (a)) dan *Job Oriented Gantt Chart* (Gambar 2.1 (b)) yang ditunjukkan oleh gambar di bawah ini.



Gambar 2.1 Jenis grafik *Gantt Chart*

Berikut diberikan contoh masalah PFSP beserta penggunaan *Gantt Chart* untuk menghitung *makespan* dari jadwal-jadwal yang terbentuk.

Contoh 2.1

Sebuah Perusahaan pembuatan pakaian akan memproduksi 3 jenis pakaian , yaitu pakaian 1, pakaian 2, dan pakaian 3, yang akan dikerjakan dengan 2 mesin, yaitu mesin potong dan mesin jahit. Berikut adalah daftar waktu proses setiap pakaian pada mesin potong dan mesin jahit.

Tabel 1.1 Waktu proses untuk pekerjaan pada tiap mesin

Mesin / Pekerjaan	Waktu proses	
	M1	M2
P1	3	5
P2	4	3
P3	5	2

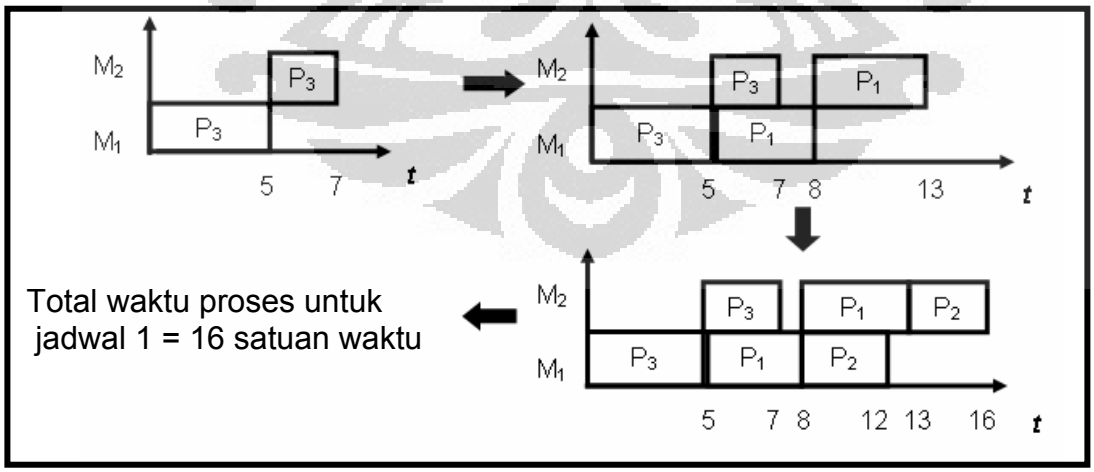
P_i = pakaian ke- i, i= 1,2,3

M_j = mesin ke- j, j= 1,2 bertutur-turut adalah mesin potong dan mesin jahit

Setiap pakaian harus diproses berurutan pada mesin potong kemudian mesin jahit, tepat 1 kali. Untuk efesiensi dan keteraturan, kedua mesin di*setting* untuk memroses pakaian yang sama sebelum di *setting* untuk memroses pakaian berikutnya. Perusahan ingin menjadwalkan pemrosesan ketiga pakaian tersebut, untuk meminimumkan total waktu prosesnya agar biaya produksi lebih rendah.

Berikut adalah contoh perhitungan *makespan* salah satu jadwal dari masalah di atas, menggunakan *Machine Oriented Gantt Chart*.

Jadwal 1 : 3, 1, 2



Gambar 2.2 Contoh perhitungan *makespan* dengan *Machine Oriented Gantt Chart*

Setiap jadwal dapat menghasilkan nilai *makespan* yang berbeda-beda. Jadwal terbaik untuk contoh di atas adalah jadwal dengan *makespan* terkecil dari 3 ! atau 6 jadwal yang dapat dibentuk.

Perkembangan penelitian terhadap PFSP sangat berguna, karena implementasinya yang sederhana dan telah diketahui bahwa solusi PFSP merupakan pendekatan yang baik untuk solusi FSP [RAV06].

Metode heuristik yang telah digunakan dalam penelitian-penelitian terkait PFSP antara lain Metode NEH oleh Nawaz, Enscore, and Ham 1983, *Simulated Annealing* oleh Ogbu dan Smith 1990, *Tabu search* oleh Taillard 1990, Algoritma Genetika oleh Murata et al 1996, *Hybrida Genetic Algorithm* (HGA) oleh El Bouri, Algoritma *Genetic Local Search* (GLS) oleh Yamada et al, 1997, Algoritma Robust Genetik (RGA) oleh Ruiz et al, *Iterated Local Search* oleh Thomas Stutzle 1998*, Greedy Randomized Adaptive Search Procedure,* dan algoritma memetika oleh Pablo Moscato.

Pada tugas akhir ini, PFSP akan diselesaikan dengan menggunakan kombinasi Algoritma Memetika (AM) dan metode GRASP.

Selanjutnya, teori mengenai metode GRASP akan dibahas pada Subbab 2.3.

2.3 GRASP

Sebelum dijelaskan mengenai teori dasar dari metode GRASP yang akan digunakan pada tugas akhir ini, yaitu metode NEH, ILS, dan PR, akan dijelaskan mengenai metode GRASP standar terlebih dahulu.

Metode GRASP telah berhasil menyelesaikan banyak masalah optimisasi kombinatorial secara efisien. GRASP mulai dikembangkan pada tahun 1980 an dan singkatan tersebut diciptakan oleh Feo & Resende 1995. GRASP adalah algoritma iteratif yang biasanya terdiri dari tahap pembentukan solusi dan tahap perbaikan solusi pada setiap iterasinya. Pada algoritma GRASP standar , dibentuk solusi awal yang baik menggunakan algoritma *greedy randomized adaptive function* dan solusi awal tersebut diperbaiki menggunakan *Local Search .* Kemudian solusi yang baik dihasilkan pada setiap iterasinya dan dimasukkan ke dalam kumpulan solusi atau '*Pool*'. Kunci kesuksesan algoritma *Local Search* tergantung dari pilihan struktur *neighborhood* yang cocok, keefisienan pencarian *neighborhood* dan pemilihan solusi awal. Maka, solusi awal dari GRASP memiliki peran penting untuk kunci terakhir, karena GRASP menghasilkan solusi awal yang baik untuk *Local Search. Pseudocode* untuk metode GRASP standar :

```
Prosedur GRASP standar
for i = 1 sampai maksimum iterasi
     Bentuk solusi awal
     Lakukan local search  pada solusi awal, hingga
     ditemukan solusi terbaik.
     Update solusi dengan solusi hasil local search.
end for
```

GRASP telah dikembangkan dengan menambahkan berbagai macam metode heuristik lainnya, misalnya *Path Relinking*. Penambahan *Path Relinking* pada algoritma dasar GRASP terkenal dapat menghasilkan peningkatan yang cukup signifikan pada kualitas solusi [ABR01].

Pada metode GRASP yang digunakan dalam tugas akhir ini, solusi awal dibentuk dengan metode NEH, kemudian pada setiap iterasinya dilakukan peningkatan kualitas solusi terhadap solusi sekarang dengan menerapkan metode ILS dan PR. Metode NEH dijelaskan pada Sub-subbab 2.3.1, kemudian metode ILS dan PR dibahas pada Sub- Subbab 2.3.2 dan 2.3.3.

2.3.1. NEH

Pada tahun 1983, Nawaz, Enscore, and Ham telah memperkenalkan suatu metode heuristik NEH untuk masalah FSP. NEH mudah dibentuk dan menghasilkan solusi yang bagus untuk kebanyakan masalah FSP. Ide dari metode NEH adalah pekerjaan dengan total waktu proses pada semua mesin lebih besar, seharusnya diberi bobot yang lebih tinggi untuk dimasukkan terlebih dahulu ke dalam jadwal, sehingga dapat diraih *makespan* yang minimum. Hingga saat ini, NEH merupakan salah satu heuristik terbaik untuk masalah meminimumkan *makespan* [RMA 2005].

Pseudocode dari metode NEH untuk PFSP :

```
1    Urutkan pekerjaan berdasarkan jumlah waktu proses
     secara menurun.
2    Buat jadwal dari urutan 2 pekerjaan pertama, yang
     meminimumkan jadwal parsial.
3    for i = 3 sampai n
     Tempatkan pekerjaan ke- i pada salah satu posisi i
     yang mungkin pada solusi parsial, yang meminimumkan
     jadwal parsial.
     end for
```

Untuk lebih jelasnya, diberikan contoh penggunaan algoritma NEH pada sebuah masalah PFSP sebagai berikut:

Langkah 1: Urutkan pekerjaan berdasarkan jumlah waktu proses secara menurun.

Pekerjaan	Mesin			
	1	2	3	$\sum p_{ij}$
1	6	1	15	25
2	7	5	8	20
3	2	13	4	19
4	4	2	7	13

Langkah 2: Menemukan urutan terbaik dari pekerjaan *1* dan *2*

Pekerjaan yang diuji : *1* dan *2*
Jadwal parsial : belum ada

	Makespan	
(1) *1 – 2*	**30**	→ lebih baik
(2) *2 – 1*	35	

Langkah 3: Ambil pekerjaan berikutnya, dan letakkan pada posisi terbaik. Ulangi hingga semua pekerjaan masuk ke dalam jadwal.

Pekerjaan yang diuji : *3*
Jadwal parsial : *1 – 2*

	Makespan	
(1) ***3*** *– 1 – 2*	42	
(2) *1 –* ***3*** *– 2*	**34**	→ lebih baik
(3) *1 – 2 –* ***3***	35	

Pekerjaan yang diuji : *4*
Jadwal parsial : *1 – 3 – 2*

	Makespan	
(1) ***4*** *– 1 – 3 – 2*	**40**	→ lebih baik
(2) *1 –* ***4*** *– 3 – 2*	42	jadi, **solusi NEH adalah**
(3) *1 – 3 –* ***4*** *– 2*	41	*4 – 1 – 3 – 2*
(4) *1 – 3 – 2 –* ***4***	41	

Gambar 2.3 Contoh pembentukan solusi NEH

2.3.2. *Iterated Local Search* (ILS)

ILS adalah metode heuristik sederhana dan kuat yang secara iteratif menerapkan *Local Search* terhadap modifikasi dari solusi sekarang.

Local Search adalah metode heuristik untuk memperbaiki atau meningkatkan kualitas solusi sekarang dengan menelusuri *neighborhood*nya hingga ditemukan solusi yang lebih baik. Jika solusi yang lebih baik ditemukan, solusi sekarang di *update*, kemudian dilakukan lagi pencarian *neighborhood*nya. Jika tidak ada lagi peningkatan kualitas solusi, pencarian dihentikan, dan berarti solusi optimal lokal telah ditemukan.

ILS disebut sederhana karena hanya perlu menambahkan beberapa baris dalam *listing code* algoritma yang sudah ada. Dan ILS disebut kuat karena ILS telah berhasil menunjukkan hasil yang terbaik pada *Traveling Salesman Problem* dan sangat kompetitif dalam menyelesaikan berbagai masalah lain [STU98].

Ada empat komponen dari ILS, yaitu :

1. Pembentukan solusi awal

   Dapat dilakukan dengan beberapa metode, diantaranya metode *random*, *greedy construction heuristic*, dan NEH. Kemudian, solusi awal diperbaiki dengan menggunakan *Local Search*

2. *Local Search*

   Prosedur pencarian *neighborhood* yang digunakan untuk *Local Search* adalah *insertion neighborhood* dimana suatu pekerjaan pada posisi *i*

dipindahkan dan dimasukkan kembali pada semua kemungkinan posisi, $j \neq i$. Jika ditemukan peningkatan kualitas solusi, solusi di *update* dan prosedur diatas diulang kembali. Hal ini dilakukan untuk semua pekerjaan hingga pencarian selesai, dimana tidak ada lagi peningkatan kualitas solusi. Prosedur *insertion* pada *insertion neighborhood* disini adalah *shift insertion* . Contoh : nilai 6 dimasukkan ke posisi *2* adalah seperti berikut:

3. *Perturbation*

   Perturbation adalah prosedur memodifikasi solusi sekarang dengan cara melakukan satu kali perpindahan.

   Perturbation dapat dilakukan beberapa prosedur, diantaranya :

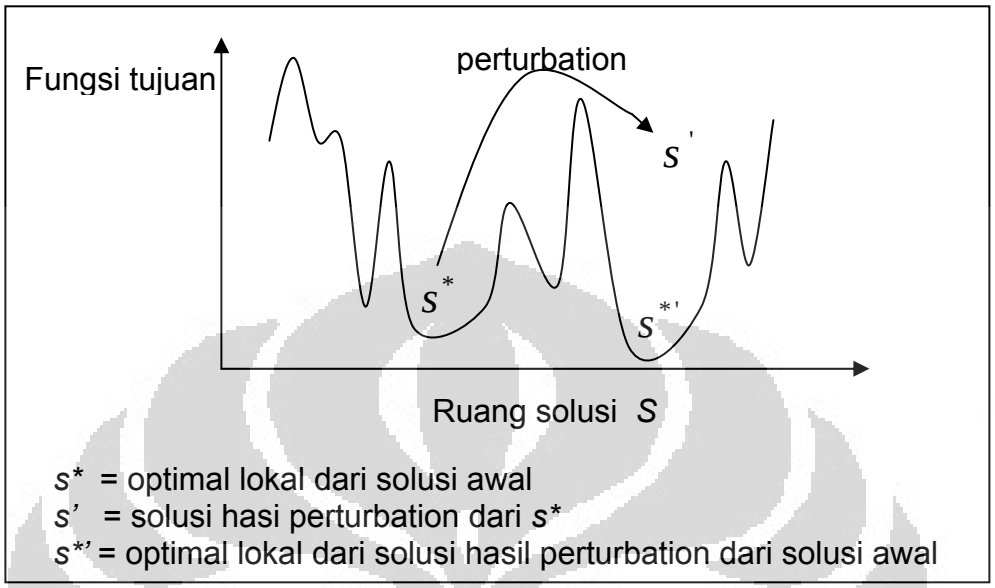
   - *Swap* 2 pekerjaan yang berurutan.
   - *Swap* 2 pekerjaan yang dipilih secara *random*.

   Perubahan kecil ini dilakukan untuk mencari optimal lokal yang lebih baik dari optimal lokal solusi sekarang. Solusi hasil *Perturbation* ini kemudian diperbaiki lagi dengan menggunakan *Local Search.*

4. Kriteria penerimaan solusi

   Tujuan diterapkannya kriteria penerimaan solusi ini adalah untuk menghindari penurunan kualitas solusi. Biasanya, solusi *Local Search* diterima untuk diterapkan pada iterasi selanjutnya, jika terjadi peningkatan kualitas solusi.

Prinsip ILS dapat dilihat pada gambar berikut:



Gambar 2.4 Prinsip ILS

Pseudocode dari metode ILS :

```
Prosedur  Iterated Local Search
      s0 = Solusi awal
      s* = LocalSearch (s0)
      repeat
            s' = Perturbation (s*)
            s*' = LocalSearch (s')
            s* = Kriteria penerimaan solusi (s*,s*')
      Until kondisi berhenti
end
```

2.3.3. *Path relinking* (PR)

PR pertama kali dikenalkan oleh Glover dan Laguna (1997). PR adalah metode heuristik untuk pencarian solusi yang lebih intensif. Tujuan dari PR adalah menemukan solusi baru yang lebih baik dengan cara

memeriksa ruang pencarian atau *"path"* diantara 2 solusi yang sudah baik pada kumpulan solusi atau*"Pool"*.

Ide dari metode ini adalah membawa atau mengarahkan solusi pertama yang disebut "*initial solution*" menjadi semakin dekat atau sama dengan solusi kedua yang disebut "*guiding solution*". Solusi-solusi yang terbentuk selama proses tersebut, disebut dengan "*path solution".* Solusi PR adalah solusi terbaik diantara semua "*path solution"* tersebut.

Prosedur PR terdiri dari beberapa tahap, dimana setiap tahap terdiri dari beberapa *path solution. Path solution* pada suatu tahap adalah solusi yang terbentuk dari hasil satu kali *swap* pada solusi awal di tahap tersebut. Mekanismenya adalah nilai dari solusi awal dan *guiding solution* pada suatu posisi dibandingkan. Jika berbeda, maka *swap* kedua nilai yang berbeda tersebut pada solusi awal, sehingga terbentuk satu *path solution.* Pada setiap tahap, dipilih solusi terbaik dari semua *path solution* yang terbentuk, untuk menjadi solusi awal pada tahap berikutnya.

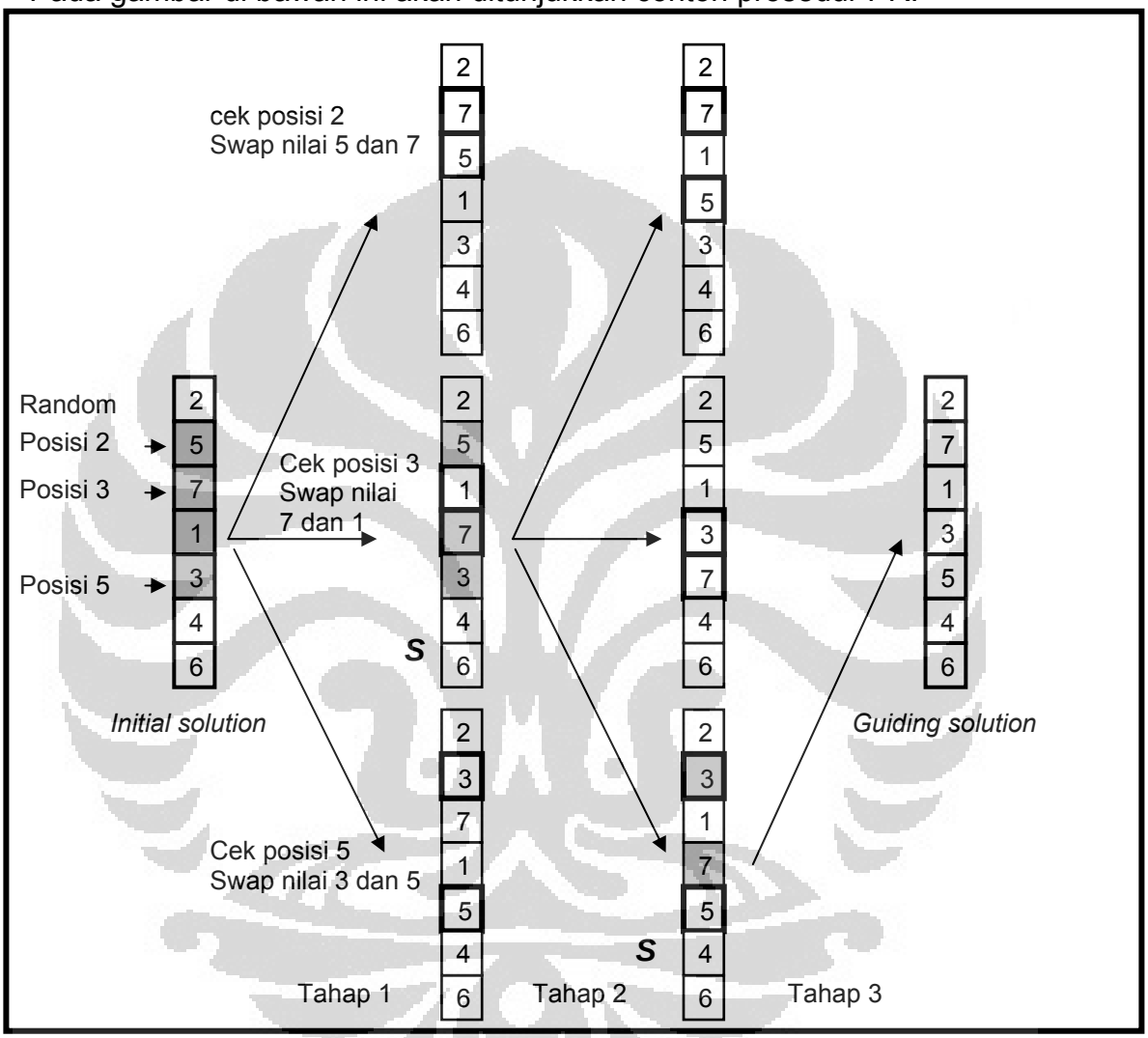
Prosedur ini dimulai dari "*initial solution*", dan berhenti jika telah menemukan solusi yang sama dengan "*guiding solution*" atau telah mencapai jumlah tahap maksimum.

Solusi PR akan menggantikan solusi terburuk dalam *"Pool",* jika solusi hasil PR lebih baik dan tidak sama dengan solusi yang sudah ada dalam *"Pool".* Ada 3 versi terkait pada solusi mana PR akan diterapkan, yaitu:

- Antara 2 solusi terbaik dari *"Pool"*

- Antara 2 solusi yang dipilih secara acak dari *"Pool".*
- Antara solusi terbaik dan solusi yang dipilih secara acak dari *ool".*

Pada gambar di bawah ini akan ditunjukkan contoh prosedur PR.



Gambar 2.5 Contoh prosedur PR

Gambar 2.5 adalah contoh prosedur PR. Pada tiap tahap hanya diizinkan 3 kali *swap* dari semua kemungkinan *swap* yang dapat dilakukan, maka dihasilkan 3 *path solution*. Lambang "***S***" menunjukkan solusi terbaik pada tiap tahap dan akan menjadi solusi awal untuk tahap berikutnya.

Perhatikan bahwa jumlah *path solution* pada suatu tahap dapat berjumlah kurang dari 3, jika jumlah posisi yang nilainya berbeda dengan "*guiding solution*" kurang dari 3. Pada contoh di atas, prosedur PR sudah berhenti pada tahap 3, karena solusi yang dihasilkan sudah sama dengan "*guiding solution*".

Selanjutnya akan dijelaskan mengenai Algoritma Memetika secara umum pada Subbab 2.4.

2.4 Algoritma Memetika

Algoritma Memetika (AM) adalah perluasan dari Algoritma Genetika. Algoritma Genetika (AG) merupakan salah satu metode heuristik yang banyak digunakan untuk menyelesaikan masalah optimisasi kombinatorial yang sulit. AG diajukan oleh John Holland pada tahun 1960-an yang didasari oleh proses evolusi biologi. Dalam ilmu Biologi, sekumpulan individu yang sama, hidup dan berkembang bersama dalam suatu area, disebut dengan populasi. Konsep yang penting dalam AG adalah hereditas dan seleksi alam. Hereditas merupakan sebuah ide bahwa sifat-sifat individu dapat dikodekan dengan cara tertentu, sehingga sifat- sifat itu dapat diturunkan kepada generasi berikutnya. Sifat-sifat individu tersebut dikodekan kedalam sekumpulan *gen-gen* yang disebut kromosom. Perbedaan susunan *gen-gen* dalam suatu kromosom ini akan membawa sifat yang berbeda-beda dan unik pada individu. Untuk itu dapat dikatakan bahwa sebuah kromosom

melambangkan satu individu. Sedangkan ide dari seleksi alam adalah individu-individu yang tidak bisa beradaptasi akan mati, sedangkan individu – individu yang mampu beradaptasi akan hidup dan menghasilkan keturunan. Dalam waktu yang cukup lama, individu-individu dapat melakukan perkawinan dan menghasilkan keturunan yang membawa sifat kedua orang tuanya. Dan dalam penurunan sifat ini dapat terjadi kesalahan yang menyebabkan terjadinya perubahan hereditas pada generasi berikutnya.

Pada AG, kandidat solusi (selanjutnya disebut 'solusi') dari suatu masalah direpresentasikan sebagai kromosom. AG diawali dengan pembentukan kumpulan kromosom yang disebut dengan populasi. Masing-masing kromosom pada populasi akan dievaluasi menggunakan fungsi *fitness, yaitu* fungsi yang mengukur secara kuantitatif kemampuan suatu kromosom untuk bertahan dalam populasi. Kemudian secara iteratif akan dibentuk populasi baru yang lebih baik dari populasi sebelumnya dengan menerapkan operator-operator genetika hingga mencapai kriteria berhenti. Operator dasar pada AG adalah seleksi, *crossover*, dan mutasi. Seleksi yaitu operator untuk memilih 2 kromosom sebagai orang tua. *Crossover* adalah operator untuk mengawinkan 2 kromosom orang tua, sehingga menghasilkan keturunan atau anak untuk generasi berikutnya. Dan mutasi adalah operator yang mengubah urutan *gen* pada kromosom. Khusus untuk masalah FSP, penerapan AG pertama kali digunakan oleh Werner (1984).

Kelebihan dari AG adalah pada cara kerjanya yang *parallel*. AG bekerja dalam ruang pencarian yang menggunakan banyak individu

sekaligus, sehingga kemungkinan AG untuk terjebak pada ekstrim lokal lebih kecil dibandingkan metode lain. AG juga mudah diimplementasikan. Karena jika suatu algoritma AG sudah dapat diimplementasikan, kita juga dapat menyelesaikan masalah lain hanya dengan membuat kromosom dan fungsi *fitness* yang baru, sehingga AG bisa dijalankan.

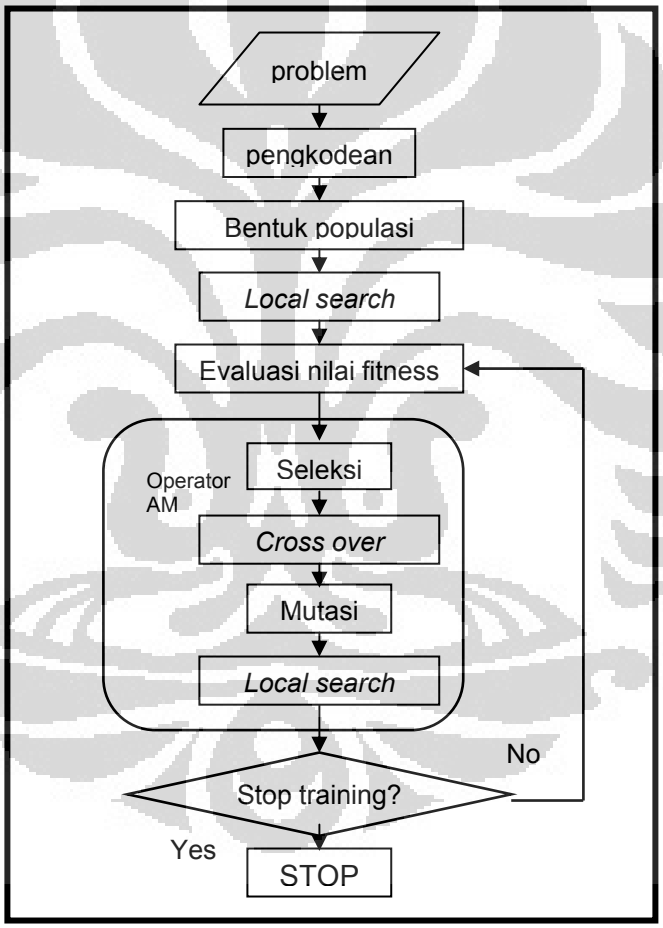
Kekurangan dari AG adalah dalam hal waktu komputasi karena harus melakukan evaluasi *fitness* pada semua solusi di setiap iterasinya. Sehingga AG bisa lebih lambat dibandingkan dengan metode lain. Namun karena kita dapat mengakhiri komputasi pada waktu yang diinginkan, proses yang lama ini dapat diatasi. [OBI98]

Namun telah diketahui bahwa AG tidak cukup baik untuk mencari solusi yang sangat dekat dengan solusi optimal. Kekurangan ini dapat diakomodasi dengan cara menambahkan metode *Local Search* pada AG [GAL97]. Hasil penggabungan ini dikenal dengan sebutan Algoritma Memetika.

AM didasari oleh Teori evolusi Biologi *Neo-Darwinian* dan pendapat Dawkin tentang *meme* sebagai unit evolusi kultural yang mampu melakukan perbaikan terhadap dirinya sendiri. AM adalah suatu metode pencarian heuristik yang memiliki karakteristik yang sama dengan AG dikombinasikan dengan metode '*Local Search'*, yang secara bersama-sama dapat meningkatkan kualitas pencarian solusi (Moscato, 1989).

Pada AM, *Local Search* bertujuan untuk melakukan perbaikan lokal yang dapat diterapkan sebelum dan atau sesudah proses seleksi, *crossover* dan mutasi.

Local Search juga sangat berguna untuk mengontrol besarnya ruang pencarian solusi. AM dapat memberikan hasil yang lebih baik daripada AG , namun memerlukan waktu komputasi yang lebih lama. Berikut adalah *flowchart* dari AM standar.



Gambar 2.6 *Flowchart* AM

Pada Sub-Subbab 2.4.1 sampai 2.4.7 akan dibahas komponen-komponen dalam AM. Akan ditemukan beberapa istilah yang mengadopsi istilah evolusi biologi. Contoh diantaranya:

- 'Kromosom' atau individu untuk merepresentasikan 'solusi'.
- 'Generasi' untuk menggantikan istilah iterasi
- 'Orangtua' adalah kromosom pada generasi sekarang yang terpilih untuk menghasilkan kromosom anak.
- 'Anak' adalah kromosom hasil perkawinan kromosom orangtua pada generasi sekarang yang akan menjadi kromosom pada generasi berikutnya.
- 'Proses evolusi' adalah proses yang terjadi pada tiap generasi yang meliputi proses seleksi, *crossover*, mutasi dan *local search*.

2.4.1 Skema Pengkodean Kromosom

Untuk bisa menggunakan AG, yang harus dilakukan sebelumnya adalah menentukan pengkodean kromosom yang cocok untuk masalah yang akan diselesaikan. Karena kromosom harus membawa informasi dari solusi yang ia representasikan. Secara umum, suatu kromosom berbentuk:

$$\text{Kromosom} = (gen_1, gen_2, \ldots\ldots, gen_n)$$

Dimana *n* adalah jumlah *gen* dalam suatu kromosom dan setiap *gen* diisi oleh nilai dari *gen* atau disebut juga *allele*.

Ada banyak cara pengkodean kromosom. Terdapat tiga skema pengkodean kromosom yang paling umum digunakan, yaitu:

- *Binary encoding*

  Binary encoding adalah pengkodean yang paling sering digunakan. Karena penelitian pertama kali tentang AG, menggunakan pengkodean ini dan pemakaiannya relatif sederhana. Pada *Binary encoding* nilai *gen* adalah 0 atau 1.

  Contoh kromosom dengan *Binary encoding:*

  Kromosom A

1	0	0	1	1	0	0	0	1

  Kromosom B

0	1	0	1	0	0	1	0	1

  Contoh masalah yang menggunakan *Binary encoding* : *Knapsack problem*

- *Value encoding*

  Value encoding digunakan untuk masalah yang sulit jika direpresentasikan dengan *binary encoding* . Pada *value encoding*, setiap kromosom adalah barisan dari beberapa nilai. Nilai disini dapat disesuaikan dengan masalahnya, misalnya bilangan *real*, huruf, atau objek tertentu.

  Contoh kromosom dengan *value encoding:*

  Kromosom A

1.2324	0.3647	1.0000	0.4556	0.1329

  Kromosom B ABDJEIFJDHDIERJFDLDFLFEGT

  Kromosom C *(back), (back), (right), (forward), (left)*

Contoh masalah yang menggunakan *value encoding* : Menentukan bobot untuk *neural network*.

- *Permutation encoding*

  Permutation encoding dapat digunakan pada *ordering problem.* Pada *permutation encoding*, setiap kromosom terdiri dari angka yang merepresentasikan urutan pengerjaan atau aktivitas.

  Contoh kromosom dengan *permutation encoding:*

Kromosom A	1	5	3	2	7	9	6	4	8
Kromosom B	8	5	6	2	7	1	3	4	9

  Contoh masalah yang menggunakan *value encoding* : *Travelling Salesman Problem* dan *Permutation Flow Shop Scheduling Problem.*

2.4.2. Pembentukan Populasi Awal

Telah diketahui bahwa AG adalah algoritma heuristik yang bekerja pada populasi, yaitu kumpulan kromosom atau solusi yang akan diperbaharui pada setiap generasinya. Maka tahap pertama dari AG adalah pembentukan populasi awal yang berisi kumpulan kromosom sebanyak ukuran populasi atau *popsize*. Pembentukan populasi awal biasanya dilakukan secara acak. Namun dalam perkembangannya, pembentukan populasi awal dapat dilakukan dengan berbagai cara yang dapat membuat AG lebih cepat konvergen ke solusi optimal. Misalnya metode NEH yang telah dijelaskan pada Sub-Subbab 2.3.1.

Selanjutnya, *Local Search* dilakukan terhadap semua kromosom pada populasi awal.

2.4.3. *Local Search*

Local Search pada AM memiliki prosedur yang sama dengan *Local Search* pada GRASP yang telah dijelaskan sebelumnya pada Subbab 2.3.2. Pada AM, *Local Search* diterapkan pada 2 tempat, yaitu:

- Sebelum masuk ke proses evolusi, *Local Search* diterapkan terhadap semua kromosom pada populasi awal.
- Pada proses evolusi, *Local Search* diterapkan sebelum atau sesudah proses seleksi, *crossover* atau mutasi. Tapi tidak diterapkan untuk semua kromosom dalam populasi, melainkan dilakukan dengan probabilitas tertentu yang disebut dengan *LS_rate*.

Setelah dilakukan *Local Search* terhadap semua kromosom, populasi awal digantikan oleh populasi hasil *Local Search.* Kemudian, AM masuk ke proses evolusi yang dimulai dari generasi pertama hingga generasi maksimum. Proses ini dimulai dengan evaluasi individu atau kromosom dengan menggunakan nilai *fitness*.

2.4.4. Evaluasi Nilai *Fitness*

Di setiap generasi, kromosom dievaluasi dengan menggunakan fungsi *fitness.* Dalam evolusi alam, individu yang bernilai *fitness* tinggi akan bertahan hidup dan individu yang bernilai *fitness* rendah akan mati. Nilai *fitness* adalah nilai yang mengukur secara kuantitatif kemampuan dari suatu individu untuk bertahan dalam populasi, sehingga bisa juga diartikan sebagai ukuran baik tidaknya suatu solusi. Nilai *fitness* sangat penting untuk mengarahkan AG dalam pencarian solusi.

Pada kebanyakan aplikasi optimisasi, fungsi *fitness* dibentuk berdasarkan fungsi tujuannya. Jika masalah optimisasinya adalah memaksimumkan, maka fungsi tujuan dapat langsung menjadi fungsi *fitness.* Namun untuk masalah dengan fungsi tujuan meminimumkan, fungsi *fitness* harus disesuaikan agar tetap berlaku individu yang lebih baik adalah individu yang memiliki nilai *fitness* lebih tinggi. Bentuk penyesuaian ini dapat dilakukan dengan berbagai cara, contohnya:

- Fungsi *fitness didefinisikan sebagai* invers dari fungsi tujuan,

$$fitness_i = \frac{1}{f_i + h}$$

  Dimana f_i adalah nilai dari fungsi tujuan dan h adalah bilangan yang sangat kecil, untuk menghindari pembagian dengan nol.

- Fungsi *fitness* diperoleh dengan mengurangi nilai fungsi tujuan terbesar dalam populasi dengan nilai fungsi tujuan suatu

kromosom, kemudian ditambah dengan bilangan kecil yang merupakan nilai *fitness* terkecil yang diinginkan.

$$fitness_i = (f_w - f_i) + small\_\text{int}$$

- Fungsi *fitness* didefinisikan :

$$fitness_i = \frac{(f_w - f_i)^2}{\sum_{i=1}^{Npop} (f_w - f_i)^2}$$

Dimana f_w adalah nilai fungsi tujuan terbesar dalam populasi dan f_i adalah nilai fungsi tujuan kromosom *i*.

Fungsi *fitness* yang digunakan dapat bermacam-macam sesuai dengan masalah. Penentuan fungsi *fitness* sangat berpengaruh pada perfomansi AM secara keseluruhan. Pada tugas akhir ini, fungsi *fitness* yang digunakan adalah jenis yang kedua dari contoh di atas.

Setelah pembentukan populasi awal dan perhitungan nilai *fitness* dilakukan, maka langkah AM selanjutnya adalah seleksi.

2.4.5. Seleksi

Seleksi adalah operator AM yang dilakukan pada setiap generasi untuk memilih kromosom-kromosom dari populasi yang akan menjadi orangtua bagi generasi berikutnya. Berdasarkan teori evolusi, individu yang menghasilkan keturunan adalah individu yang terbaik dalam populasi. Untuk

itu, pemilihan kromosom yang akan menjadi orangtua dilakukan secara proporsional sesuai dengan nilai *fitness*. Sehingga, semakin besar nilai *fitness* suatu kromosom, maka kemungkinan terpilih sebagai orangtua semakin besar juga. Terdapat beberapa metode dalam pemilihan kromosom terbaik. Diantaranya adalah :

- *Roulette wheel selection*

  Masing-masing kromosom menempati potongan lingkaran pada *roulette wheel* dengan ukuran yang proporsional, sebanding dengan nilai *fitness*nya. Semakin besar nilai *fitness* suatu kromosom, maka semakin besar kemungkinan kromosom itu untuk terpilih.

- *Rank selection*

  Dengan menggunakan *rank selection*, semua kromosom dalam populasi di *ranking* dari yang terburuk sampai yang terbaik. Setelah di *ranking* , dilakukan prosedur yang sama dengan *roulette wheel selection,* dimana kromosom terburuk mendapat 1 bagian potongan, kromosom terburuk kedua mendapat 2 bagian potongan , dan seterusnya hingga kromosom terbaik mendapat *N* (jumlah kromosom dalam populasi) bagian potongan. Namun, metode ini membuat AM lambat konvergensinya, karena kromosom terbaik memiliki kemungkinan yang tidak terlalu berbeda dengan beberapa kromosom lain.

- *Tournament selection*

  Diambil sekelompok kecil kromosom secara *random*, kemudian dipilih satu kromosom yang memiliki nilai *fitness* paling tinggi untuk menjadi orangtua. Hal ini dilakukan berulang-ulang hingga didapatkan jumlah orangtua yang diinginkan.

- *Truncation selection*

  Semua kromosom dalam populasi diurutkan berdasarkan nilai *fitness*nya, kemudian dipilih sejumlah kromosom terbaik.

Pada tugas akhir ini, jenis operator seleksi yang digunakan adalah *r*oulette *wheel selection* yang akan dibahas lebih lanjut pada Sub-Subbab 3.2.4.

Kromosom-kromosom yang terpilih menjadi orangtua akan masuk ke tahap selanjutnya, yaitu *crossover*.

2.4.6. *Crossover*

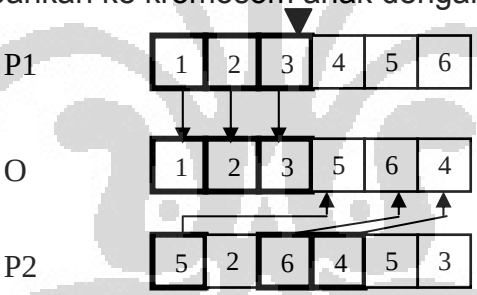
Crossover adalah operator yang mengawinkankan dua kromosom orangtua yang telah terpilih pada proses seleksi, sehingga menghasilkan kromosom anak yang mewarisi ciri-ciri dasar dari kromosom orangtua. *Crossover* diharapkan dapat menghasilkan kromosom anak yang lebih baik dengan cara menurunkan ciri-ciri terbaik dari orangtuanya, walaupun tidak semua jenis operator c*rossover* memperhatikan hal ini. Jenis operator

c*rossover* harus disesuaikan dengan tipe masalah, agar tetap menjaga kelayakan dari kromosom anak yang dihasilkan.

Berikut diberikan beberapa jenis operator c*rossover* beserta prosedurnya. Contoh akan menggunakan kromosom dengan 6 *gen*. P1 dan P2 adalah kromosom orangtua, sedangkan O adalah kromosom anak.

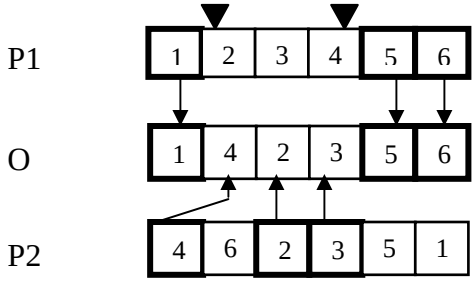
- *One point crossover*

  Satu titik dipilih secara *random* untuk membagi orangtua yang pertama, kemudian *gen* pertama hingga titik yang terpilih di turunkan ke kromosom anak sesuai posisinya, sedangkan orangtua yang satu lagi dicek, jika ada nilai *gen* yang belum ada pada anak, maka ditambahkan ke kromosom anak dengan cara *left-to-right*.



- *Two point crossover*

  Dua titik dipilih secara *random* untuk membagi orangtua yang pertama. Nilai *gen* di luar sisi dua titik yang terpilih, diturunkan ke kromosom anak, sesuai posisinya. Dari orangtua kedua, dilihat nilai *gen* yang lain, kemudian ditambahkankan ke kromosom anak .

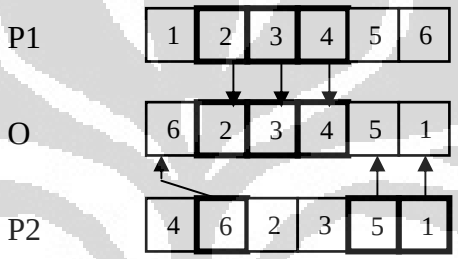


- *Uniform crossover*

  Untuk memilih *gen* dari orangtua mana yang akan diturunkan ke kromosom anak, dilakukan dengan menggunakan rasio.

- *Subsequence preservation crossover*

  Pilih subbarisan dari orangtua pertama untuk diturunkan ke kromosom anak. Kemudian masukkan nilai *gen* yang lain dari orangtua kedua ke kromosom anak dengan cara *left-to-right* .



Pada tugas akhir ini, jenis operator *crossover* yang digunakan adalah *Path crossover* yang akan dijelaskan pada Sub-Subbab 3.2.6.

Crossover rate adalah parameter yang menunjukkan seberapa sering c*rossover* dilakukan. Jika *crossover* dilakukan, maka kromosom anak terbentuk dari bagian kromosom kedua orangtuanya. Jika tidak dilakukan *crossover*, maka kromosom anak merupakan copyan langsung dari orangtuanya. Jika *crossover rate* = 0 %, maka semua generasi baru terbentuk dari pengopyan kromosom dari populasi generasi sebelumnya. Jika *crossover rate* = 100%, maka semua kromosom anak merupakan hasil dari *crossover. Crossover rate* biasanya sekitar 70%-95%

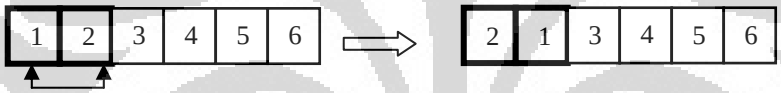
Selanjutnya, kromosom anak hasil *crossover,* masuk ke tahap mutasi.

2.4.7. Mutasi

Mutasi adalah operator dasar genetika yang mengubah posisi *gen* dalam kromosom. Tujuan mutasi adalah memelihara keanekaragaman kromosom pada populasi untuk mencegah terbentuknya populasi yang homogen, sehingga dapat menghindari konvergen ke optimal lokal. Mutasi biasanya dilakukan terhadap kromosom hasil *crossover*. Terdapat beberapa jenis operator mutasi, diantaranya:

- *Swap mutation*

  Adalah menukar 2 *gen* yang berurutan.

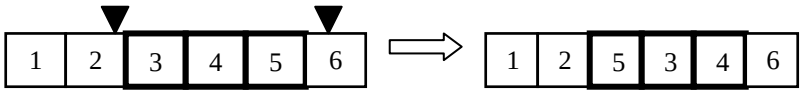


- *Exchange mutation*

  Adalah menukar 2 *gen* yang dipilih secara *random*.

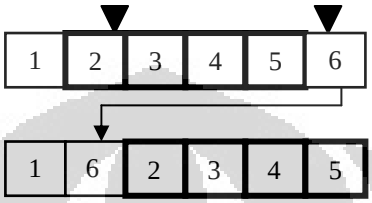
- *Scramble sublist mutation*

  Adalah memilih 2 titik secara *random*, kemudian mengubah susunan *gen* diantara 2 titik tersebut secara *random*.



- *Shift mutation*

  Adalah memilih satu *gen* secara *random* kemudian memasukkannya ke posisi yang dipilih secara *random* juga.



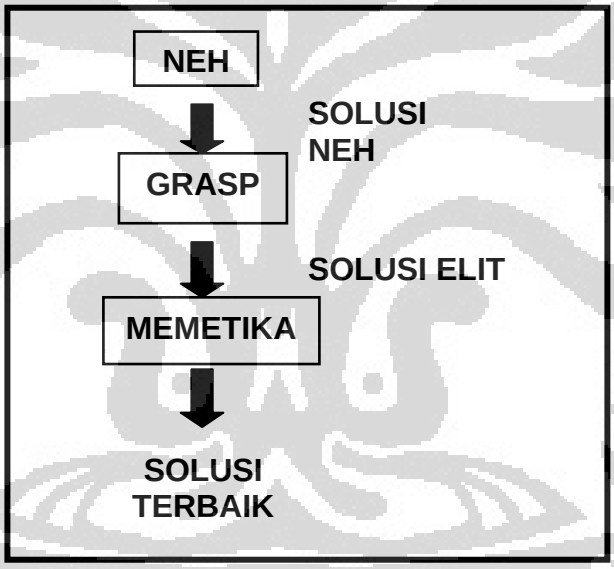
Jenis operator mutasi yang akan digunakan pada tugas akhir ini adalah *Exchange mutation.*

Mutasi sebaiknya tidak sering dilakukan. *Mutation rate* biasanya sekitar 0.05% - 1%**.** *Mutation rate* adalah parameter yang menunjukkan seberapa sering mutasi dilakukan. Jika tidak dilakukan mutasi, maka kromosom anak tidak mengalami perubahan setelah *crossover.* Jika mutasi dilakukan maka satu atau beberapa kromosom mengalami perubahan. Jika *mutation rate* = 0 %, maka tidak ada perubahan pada semua kromosom, dan jika *mutation rate* = 100%, maka semua kromosom anak berubah.

BAB III

PENERAPAN ALGORITMA MEMETIKA DAN GRASP DALAM MENYELESAIKAN PFSP

Prosedur AM dan GRASP dalam menyelesaikan PFSP dapat digambarkan oleh *flowchart* berikut:



Gambar 3.1 *Flowchart* AM dan GRASP

Biasanya, populasi awal pada AM dibangun secara acak. Hal ini mengakibatkan AM memerlukan iterasi yang cukup banyak untuk konvergen ke solusi optimal. Pada tugas akhir ini diusulkan untuk membangun populasi awal yang cukup baik, sehingga AM hanya memerlukan operator yang sederhana dan iterasi yang sedikit. Hal inilah yang menjadi latar belakang adanya penggabungan dua metode, yaitu AM dan GRASP. Dimana GRASP

membentuk kumpulan solusi yang cukup baik terlebih dahulu, untuk kemudian digunakan oleh AM sebagai populasi awal.

Metode GRASP dalam menyelesaikan PFSP dijelaskan pada Subbab 3.1 dan Algoritma Memetika dalam meyelesaikan PFSP dijelaskan pada Subbab 3.2.

3.1 GRASP dalam Menyelesaikan PFSP

Pada metode GRASP ini, digunakan metode NEH untuk membentuk solusi awal sebelum masuk ke proses iterasi. Prosedur NEH telah dijelaskan pada Sub-Subbab 2.2.1. Solusi dari metode NEH dimasukkan kedalam kumpulan solusi atau *Pool.* Kemudian dilakukan LS terhadap solusi NEH, dan solusi LS yang dihasilkan dimasukkan kedalam *Pool* dengan syarat solusi tersebut tidak terlalu mirip dengan solusi yang sudah ada dalam *Pool*. Parameter yang digunakan untuk kemiripan ini adalah *proximity*.

Proximity adalah batas minimal perbedaan komponen suatu solusi dengan semua solusi yang sudah ada dalam *Pool,* yang menjadi syarat dimasukkannya solusi tersebut kedalam *Pool.* Contoh : 2 solusi memiliki p*roximity* sebesar 5 artinya, terdapat 5 buah *allele* yang posisinya berbeda antara 2 solusi tersebut.

Pada Algoritma GRASP standar, pada setiap iterasi dilakukan pembentukan solusi baru. Sehingga *Pool* untuk setiap iterasi saling *independent*. Sedangkan pada metode GRASP ini, kita menjaga suatu

kumpulan solusi dengan kualitas yang baik, untuk kemudian diterapkan prosedur *Path Relinking* (PR) pada kumpulan solusi tersebut.

Proses iterasi pada metode ini, menggunakan metode ILS dan PR untuk perbaikan solusi, dengan prosedur sebagai berikut:

1. Solusi awal

   Solusi awal yang digunakan adalah *Current Solution*. Untuk iterasi pertama, *Current Solution* adalah solusi terbaik dari *Pool.* Untuk iterasi selanjutnya, *Current Solution* selalu di *update* dengan solusi hasil ILS dan PR.

2. *Perturbation*

   Dilakukan terhadap solusi awal, dengan melakukan *swap* terhadap 2 posisi yang dipilih secara acak.

3. *Local Search*

   Diterapkan pada solusi hasil dari *perturbation*, dengan prosedur *insertion neighborhood.* Prosedur *insertion* pada *insertion neighborhood* disini, sama dengan *shift mutation* pada AM yang telah dijelaskan pada Sub-Subbab 2.4.7.

   Kemudian solusi *Local Search* dimasukkan ke *Pool* dengan syarat tidak terlalu mirip dengan solusi yang sudah ada dalam *Pool* dan *makespan* nya lebih kecil daripada solusi terbaik.

4. PR

   Diterapkan pada 2 solusi dalam *Pool*, yaitu solusi terbaik dan solusi yang dipilih acak dari *Pool*. PR tidak dilakukan pada setiap iterasi, tetapi

dengan frekuensi tertentu yang disebut dengan *Path Relinking Frequency* (PRF) .Solusi PR dimasukkan kedalam *Pool* dengan syarat yang sama dengan syarat masuknya solusi *Local Search* di atas.

5. Kriteria penerimaan solusi PR

   Jika ditemukan peningkatan kualitas solusi, *Current Solution* diganti dengan solusi PR. Jika tidak ditemukan peningkatan, *Current Solution* diganti dengan solusi PR dengan probabilitas $p_T > 0$. Jika *n* adalah jumlah pekerjaan, *m* adalah jumlah mesin, dan p_{ij} adalah waktu proses pekerjaan *i* pada mesin *j*, maka *temperature* T adalah

$$T = 0.5 \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij}}{n.m.10}$$

   Probabilitas penerimaan solusi adalah:

   p_T = *exp* (- *makespan* solusi PR - *makespan Current solution*) / T

6. Kriteria penerimaan solusi *Local Search*

   Jika ditemukan peningkatan kualitas solusi, *Current Solution* diganti dengan solusi *Local Search*. Jika tidak ditemukan peningkatan, *Current Solution* diganti dengan solusi *Local Search* dengan probabilitas $p_T > 0$.

   p_T = *exp*(- *makespan* solusi LS - *makespan Current solution*) / T

Hasil dari metode GRASP adalah kumpulan solusi elit, yang selanjutnya akan digunakan oleh AM sebagai populasi awal.

3.2 AM dalam Menyelesaikan PFSP

Setelah pembentukan populasi awal, semua solusi pada populasi diperbaiki dengan menggunakan *Local Search*. Kemudian dilakukan proses evolusi. Kromosom orangtua diseleksi dengan menggunakan *Roulette wheel selection,* untuk dikawinsilangkan menggunakan operator *Path Crossover.* Setelah itu dilakukan mutasi terhadap solusi terbaik dalam populasi dan *Local Search* diterapkan lagi pada populasi yang telah dihasilkan. Kemudian populasi sekarang digantikan oleh populasi baru hasil evolusi. Prosedur ini dilakukan terus - menerus pada setiap generasi. Jika sampai sejumlah generasi tertentu tidak terjadi peningkatan kualitas populasi secara berturut-turut, maka semua solusi pada populasi diganti dengan menggunakan *Cold Restart*. Setelah penggantian seluruh solusi dalam populasi, prosedur AM dilanjutkan hingga mencapai kriteria berhenti. Kriteria berhenti yang akan digunakan pada tugas akhir ini adalah maksimum generasi. Komponen AM yang digunakan dijelaskan pada Sub-Subbab 3.2.1 sampai 3.2.8

3.2.1 Skema Pengkodean kromosom

Jenis pengkodean kromosom yang digunakan adalah *permutation encoding*. Dimana kromosom menggambarkan urutan pekerjaan, *gen-gen* melambangkan posisi-posisi pekerjaan tersebut, sedangkan *allele* menggambarkan pekerjaan-pekerjaan yang memiliki posisi-posisi tersebut.

Misalnya, untuk PFSP dengan 10 pekerjaan dan 5 mesin, salah satu kromosom yang dapat dibentuk adalah:

7	5	4	2	1	9	6	3	8	10

3.2.2 Pembentukan Populasi Awal

Tahap pertama dari AM adalah membentuk populasi awal yang berisi kumpulan kromosom/solusi sebanyak ukuran populasi atau *popsize* yang diinginkan. Untuk AM yang akan digunakan dalam metode heuristik pada tugas akhir ini, populasi awal yang digunakan adalah kumpulan solusi atau *Pool* yang telah dihasilkan pada metode GRASP sebelumnya, yaitu sebanyak 15 solusi terbaik, kemudian ditambah dengan sejumlah solusi yang dibangun secara *random* hingga jumlahnya sesuai dengan *popsize* yang diinginkan.

3.2.3 *Local Search*

Pada AM untuk menyelesaikan PFSP dalam skripsi ini, *Local Search* dilakukan terhadap semua solusi pada populasi awal dan setelah proses mutasi pada setiap iterasi dengan *LS_rate* yang relatif kecil.

3.2.4 Evaluasi Nilai *fitness*

Pada tugas akhir ini, nilai *fitness* yang digunakan pada AM untuk menyelesaikan PFSP adalah

$$fitness_i = (f_w - f_i) + c$$

Dimana f_w adalah nilai *makespan* terbesar dalam populasi, f_i adalah nilai *makespan* kromosom *i*, dan *c* adalah bilangan kecil sebagai nilai *fitness* terkecil yang diinginkan.

Berikut diberikan contoh perhitungan nilai *fitness*.

Contoh 3.1

Diketahui populasi dengan *popsize* 5 dan *makespan* masing-masing kromosom telah diberikan. Maka nilai *fitness*nya diberikan oleh tabel berikut:

f_w = 55

Pilih bilangan kecil, c =1

Tabel 3.1 Perhitungan nilai *fitness*

	Makespan	$(f_w - f_i) + c$	nilai *fitness*
K1	55	(55-55) +1	1
K2	53	(55-53) +1	3
K3	51	(55-51)+1	5
K4	54	(55-54)+1	2
K5	52	(55-52)+1	4
Total			15

3.2.5 *Roulette Wheel Selection*

Untuk menyelesaikan PFSP pada tugas akhir ini, pemilihan orangtua pada AM dilakukan dengan menggunakan *Roulette wheel selection*. Sesuai dengan namanya, metode ini menggunakan prinsip pada permainan *roulette wheel*. Masing-masing kromosom menempati potongan lingkaran pada *roulette wheel* dengan ukuran yang proporsional, sebanding dengan nilai *fitness*nya. Pemilihan kromosom dilakukan dengan cara memutar *roulette wheel*. Semakin besar nilai *fitness* suatu kromosom, maka semakin besar kemungkinan kromosom itu untuk terpilih. Namun tidak tertutup kemungkinan komosom dengan nilai *fitness* yang kecil bisa terpilih.

Contoh 3.2:

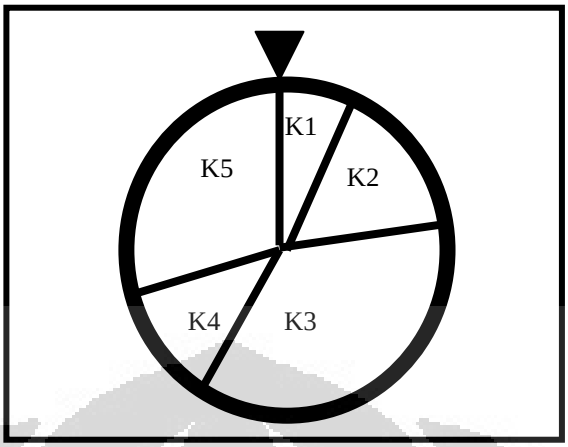
Diketahui ukuran populasi adalah 5 dengan nilai *makespan* dan *fitness* diberikan oleh tabel 3.2.

Tabel 3.2 *Makespan*, nilai *fitness* dan persentase nilai *fitness*

Kromosom	*Makespan*	Nilai *fitness*	% nilai *fitness*
K1	55	1	6,7
K2	53	3	20
K3	51	5	33,33
K4	54	2	13,33
K5	52	4	26,67
Total		15	100

Dapat dilihat dari tabel di atas bahwa, K1 memiliki peluang untuk terpilih sebagai orangtua sebesar 6,7% , K2 sebesar 20%, dan seterusnya.

Roulette wheel untuk populasi diatas ditunjukkan oleh gambar 3.2



Gambar 3.2 *Roulette wheel*

Misalkan pemutaran *roulette wheel* dilakukan sebanyak 2 kali, putaran pertama jatuh pada K3 dan putaran kedua pada K5. Maka didapat 2 kromosom orangtua, yaitu K3 dan K5 yang akan masuk kedalam tahap *crossover* untuk menghasilkan keturunan.

3.2.6 *Path Crossover* (PX)

Path crossover pertama kali diperkenalkan oleh Glover [1994], dimana prosedurnya mirip dengan *path relinking*. Kelebihan PX dibandingkan dengan operator *crossover* biasa adalah PX menjamin kromosom anak tidak akan lebih buruk daripada kromosom orangtua. Prosedur PX adalah sebagai berikut:

- Pilih 2 kromosom orangtua, kemudian pencarian dilakukan dengan cara membentuk *path* antara kedua orangtua.
- Dimulai dari posisi *random* sampai mencapai posisi ini kembali.

Allele dari kedua orangtua pada posisi *random* dibandingkan. Jika sama, semua *allele* diturunkan ke anak. Jika berbeda, lakukan *swap* antara *allele* yang berbeda tersebut pada masing-masing kromosom orangtua, sehingga dihasilkan 2 kromosom anak. Hal ini dilakukan untuk posisi selanjutnya dengan prosedur yang sama, hingga sampai ke posisi *random* yang dipilih pertama kali.

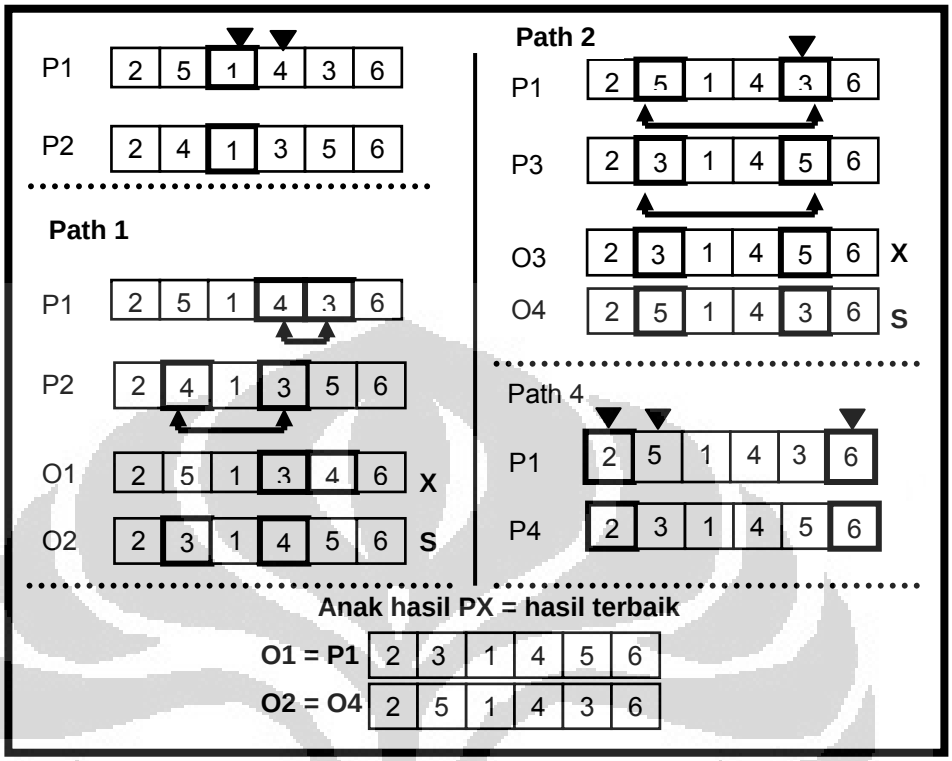
- Pada setiap *path*, algoritma ini memilih kromosom hasil *swap* dengan *makespan* yang lebih kecil daripada kromosom orangtua,. Kromosom anak hasil dari *crossover* ini adalah solusi terbaik yang ditemukan selama membentuk *path*.

Contoh Penggunaan PX:

Misalnya n=6, Kromosom orangtua yang terpilih:

P1	2	5	1	4	3	6
P2	2	4	1	3	5	6

Misalkan posisi pertama yang terpilih adalah 3. Maka dicek *allele* kedua orangtua pada posisi 3. Karena sama, maka dilanjutkan ke posisi 4. Dicek lagi *allele* kedua orangtua pada posisi 4, karena beda, *swap allele* yang berbeda tersebut pada masing-masing orangtua. Kemusdian dilanjutkan lagi untuk posisi selanjutnya hingga sampai ke posisi 2.



Gambar 3.3 Contoh prosedur *Path Crossover*

3.2.7 Mutasi

Mutasi disini bekerja seperti *perturbation* pada GRASP. Jenis operator mutasi yang digunakan adalah *Exchange mutation* yang diterapkan hanya untuk solusi terbaik.

3.2.8 *Cold restart*

Cold restart adalah proses penggantian populasi, dimana semua kromosom digantikan sekaligus oleh N kromosom baru. Ada dua masalah

yang menyebabkan penggantian populasi perlu dilakukan. Pertama, rendahnya variasi atau keanekaragaman pada populasi yang dapat menyebabkan AM konvergensi prematur ke solusi yang tidak cukup baik.. Dan kedua, ketika AM telah mencapai stagnansi, dimana tidak ada lagi peningkatan berarti setelah generasi tertentu, namun mungkin saja populasinya masih cukup bervariasi, sehingga seharusnya bisa menemukan solusi yang baik.

Proses *cold restart* adalah sebagai berikut. *Makespan* terbaik dari setiap generasi disimpan dan dibandingkan dengan *makespan* terbaik generasi sebelumnya. Setiap kali terjadi *makespan* terbaik generasi sekarang tidak lebih baik dari *makespan* terbaik generasi sebelumnya, maka dilakukan perhitungan 1 generasi. Jika setelah beberapa generasi berturut-turut tidak ada peningkatan *makespan*, maka dibentuk populasi baru dengan cara mengganti populasi sebelumnya. Batas banyak generasi sebagai syarat untuk dilaksanakannya *cold restart* dinamakan dengan generasi *cold restart* (C_r). Dengan demikian, setiap kali *makespan* terkecil dari populasi tidak berubah setelah lebih dari C_r generasi, *cold restart* dilakukan.

Prosedur penggantian populasi *cold restart* pada tugas akhir ini adalah sebagai berikut:

- Tiga solusi terbaik tetap dipertahankan pada populasi baru.
- 40 % dari populasi dihasilkan dari hasil mutasi terhadap tiga solusi terbaik di atas.

- Sisanya dibentuk secara random.

Berikut adalah *Pseudocode* metode AM dan GRASP untuk menyelesaikan PFSP:

```
Begin
    Tentukan parameter GRASP dan Algoritma Memetika
    Baca data (waktu proses setiap pekerjaan pada setiap
    mesin);
    ********* GRASP *********
    Terapkan metode NEH;
    Masukkan solusi NEH ke Pool;
    Terapkan Local Search pada solusi NEH;
    Masukkan solusi Local Search ke Pool;
    CurrSolution = BestSolution;
    for iter= 1 sampai dengan maxIter do
          Terapkan ILS ;
          Masukkan solusi ILS ke Pool (jika memenuhi
          kriteria);
          Terapkan PR (jika memenuhi kriteria);
          Masukkan solusi PR ke Pool (jika memenuhi
          kriteria);
          Update CurrSolution jika solusi PR lebih baik
          Update CurrSolution jika solusi ILS lebih baik
          Restart Pool (jika tidak ada peningkatan
          BestSolution setelah sejumlah iterasi tertentu)
    endfor;
    *********Algoritma Memetika*********
    Bentuk populasi awal sesuai ukuran popsize;
    Terapkan Local Search untuk semua solusi dalam
    populasi awal
    for gen = 1 sampai dengan maxGen do
          Hitung nilai fitness tiap kromosom pada
          populasi;
          Bentuk populasi baru dengan seleksi;
          Terapkan crosssover dan mutasi (jika memenuhi
          kriteria);
          Terapkan Local Search (jika memenuhi kriteria);
          Terapkan cold restart scheme (jika memenuhi
          kriteria);
    endfor;
end;
```

BAB IV
IMPLEMENTASI DAN HASIL PENGUJIAN

Pada Bab IV ini, implementasi dari metode AM dan GRASP untuk menyelesaikan PFSP dibahas pada Subbab 4.1. Pengujian dilakukan dengan melakukan percobaan terhadap 12 masalah penguji, dengan hasil percobaan diberikan pada Subbab 4.2. Kemudian melakukan percobaan terhadap 3 masalah penguji untuk melihat kinerja AM dan GRASP dibandingkan dengan AM saja dan GRASP saja, dengan hasil perbandingan ditunjukkan pada Subbab 4.3.

4.1 Implementasi

Pada tugas akhir ini, AM dan GRASP untuk menyelesaikan PFSP diimplementasikan dalam sebuah program dengan menggunakan MATLAB 7. Program dijalankan pada *Personal Computer* (PC) dengan prosesor Intel Celeron 2.26GHz, memori 256 Mb, dan sistem operasi *Microsoft Windows XP Professional version 2002* .

Masukan dari program adalah sebagai berikut:

- Nama *file input*

  File input memberi informasi banyak pekerjaan, banyak mesin, dan matriks waktu proses setiap pekerjaan pada setiap mesin.

- Dua parameter GRASP , yaitu maksimum iterasi (MaxIter) dan PRF.
- Lima parameter AM, yaitu maksimum generasi (MaxGen), ukuran populasi (UkPop), probabilitas *crossover* (co_rate), probabilitas mutasi (mu_rate), probabilitas *Local Search* (LS_rate), serta jumlah generasi untuk melakukan *cold restart* (Cr).

Program akan berhenti apabila maksimum generasi yang dimasukkan telah tercapai. Kemudian program akan mengeluarkan solusi terbaik yang ditemukan, waktu komputasinya dan grafik yang menampilkan perubahan *makespan* terbaik terhadap pertambahan generasi.

Berikut adalah fungsi-fungsi yang digunakan pada program beserta penjelasannya :

- GRASP*Memetic*

  Merupakan fungsi utama yang berisi metode GRASP, dengan memanggil fungsi –fungsi yang terkait, yaitu NEH, LS, PR, *perturbation*, dan *proximity*. Setelah dihasilkan solusi GRASP, kemudian dipanggil fungsi *Memetic*.

- *Memetic*

  Fungsi ini adalah fungsi utama untuk AM, yang akan memanggil beberapa fungsi operator AM , yaitu evaluasi individu(fungsi *fitness*), *rhoulette wheel*, PX, mutasi, *Local Search*, dan *cold restart*.

- *Makespan*

  Fungsi untuk menghitung *makespan* dari suatu kromosom

- NEH

  Fungsi untuk membentuk kromosom berdasarkan prosedur NEH.

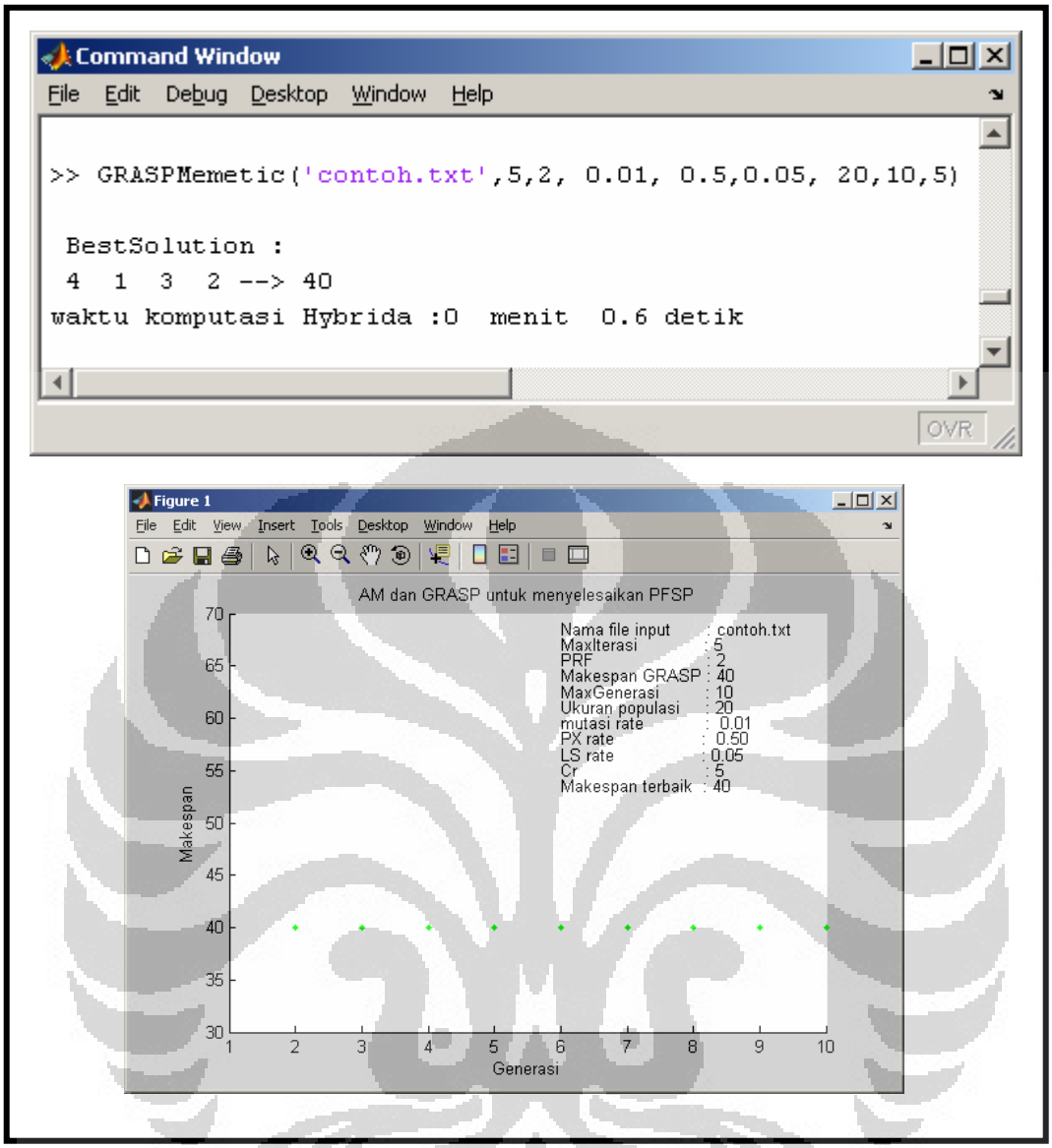
- *Makespan*_NEH

  Fungsi untuk menghitung *makespan* parsial dari kromosom pada pembentukan kromosom berdasarkan prosedur NEH.

Contoh penggunaan program ini diberikan pada contoh 4.1

Contoh 4.1

Diberikan data contoh masalah PFSP dengan 4 pekerjaan dan 3 mesin yang termuat dalam file 'contoh' yang digunakan untuk mencari solusi NEH pada Sub-Subbab 2.3.1. Ketik GRASPMemetic pada *command window* , kemudian masukkan data nama file, parameter GRASP dan parameter AM. Output yang dihasilkan adalah sebagai berikut:



Gambar 4.1 Tampilan output program untuk masalah 'contoh'

Pada gambar 4.1 bagian atas, *BestSolution* menunjukkan solusi terbaik yang ditemukan, yaitu 4-1-3-2 dengan *makespan* 40 dan waktu komputasinya 0,6 detik CPU *time.* Gambar 4.2 bagian bawah menunjukkan *makespan* terbaik dari setiap generasi pada AM.

4.2 Hasil percobaan

Pada Subbab ini akan diberikan beberapa hasil percobaan untuk melihat kinerja AM dan GRASP dalam menyelesaikan masalah PFSP dengan ukuran yang bervariasi. Masalah penguji yang digunakan adalah 12 data masalah pengujian dengan ukuran yang bervariasi, yaitu 20 x 5, 20 x 10, 20 x 20 dan 50 x 5, 50 x 10, dan 100 x 5 diambil dari *Taillard's Benchmark*.

Contoh data masalah diberikan pada Lampiran 2.Percobaan AM dan GRASP berikut ini menggunakan nilai parameter sebagai berikut:

Max Iterasi : 50
PRF : 2
Max generasi :50 atau 100
Ukuran populasi : 20
Mu_rate : 0.01
Co_rate : 0.5
LS_rate : 0.05
Cr : 20

Hasil percobaan ditampilkan pada Tabel 4.1 dimana kolom nama data menyatakan nama *file* dari masalah yang diuji, kolom ukuran menyatakan ukuran data, yaitu jumlah pekerjaan x jumlah mesin, kolom percobaan yang terdiri dari *makespan* dan waktu komputasi (dalam *second*) dari 10 kali percobaan. Kolom *makespan* terbaik adalah *makespan* dari solusi terbaik yang didapat dari 10 kali percobaan, kolom BKS (*Best Known Solution*) menyatakan *makespan* dari solusi terbaik yang didapatkan hingga saat ini

[*Taillard's Benchmark*], dan baris *error* relatif menyatakan *error* rata-rata terhadap nilai BKS.

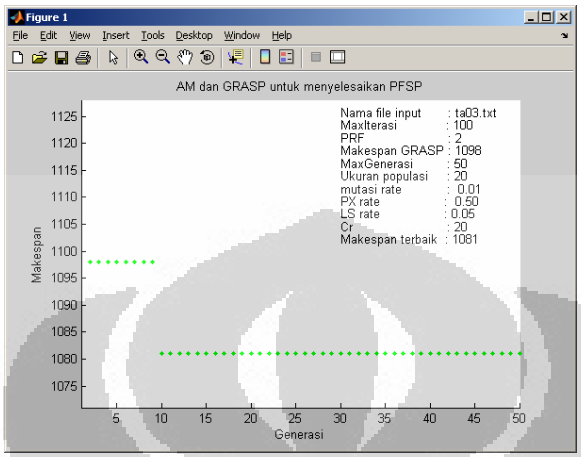
Berdasarkan tabel 4.1 dapat dilihat bahwa *error* relatif yang dihasilkan tidak lebih dari 2 %. 8 masalah diantaranya memiliki *error* relatif yang kurang dari 1 % dan hanya 4 masalah yang memiliki *error* relatif antara 1% - 2%.

Dapat dilihat juga bahwa untuk mencapai *error* relatif tersebut, waktu komputasi rata-rata cenderung meningkat seiring dengan peningkatan ukuran data.

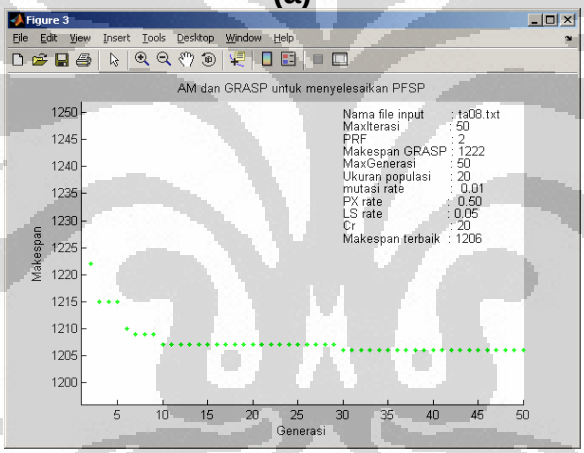
Tabel 4.1 Nilai *Makespan* dan Waktu Komputasi dari Hasil Percobaan AM dan GRASP untuk Menyelesaikan

Nama Data	Ukuran	*makespan* dan waktu komputasi (*second*) dari percobaan ke-										*Makespan* terbaik dan Waktu rata-rata	BKS	*error* relatif
		1	2	3	4	5	6	7	8	9	10			
ta03	20 x 5	1081	1089	1081	1088	1085	1088	1088	1081	1081	1084	**1081**	1081	**0,0033**
		6.6	4.7	5.9	6.1	6.8	7.4	7.2	7.1	6.4	5.1	6,33		
ta08	20 x 5	1211	1206	1206	1206	1206	1206	1206	1211	1212	1206	**1206**	1206	**0,0013**
		8.6	7.6	7.3	7.9	6.5	7.7	6.7	8.5	8.1	7.1	7,6		
ta09	20 x 5	1230	1235	1236	1230	1236	1236	1238	1238	1239	1239	**1230**	1230	**0,0046**
		7.9	9.2	10.4	9.5	7.5	6.8	6.8	9.1	7.3	8.1	8,26		
ta10	20 x 5	1108	1108	1112	1108	1111	1109	1111	1113	1111	1112	**1108**	1108	**0,0021**
		10	9.2	8.3	8.5	7.5	6.3	7.4	6.6	7.4	10.3	8,15		
ta12	20 x 10	1686	1684	1691	1691	1693	1691	1695	1694	1689	1695	**1684**	1659	**0,0192**
		9	8.9	12.1	9.5	12.3	12.1	9.4	10.2	9.4	12.8	10,57		
ta13	20 x 10	1513	1513	1519	1518	1517	1518	1509	1514	1518	1513	**1509**	1496	**0,0128**
		9.4	10	9.8	10.9	13.6	11.4	11.4	10.2	7.8	8.1	10,26		
ta22	20 x 20	2132	2122	2120	2128	2124	2126	2120	2120	2120	2120	**2120**	2099	**0,0115**
		9.1	10.4	11.8	7.1	6.3	9.6	9	9.3	10	10.3	9,29		
ta26	20 x 20	2245	2250	2252	2250	2245	2249	2243	2244	2244	2254	**2243**	2226	**0,0097**
		10.2	9.1	9.9	7.1	5.9	12.1	8.6	10.7	10.6	11.2	9,54		
ta31	50 x 5	2729	2724	2729	2724	2729	2729	2724	2724	2729	2724	**2724**	2724	**0,0009**
		90.4	20.8	23.8	19.4	90.2	92.2	29.8	22.1	20.4	90.9	50		
ta33	50 x 5	2621	2627	2631	2621	2623	2627	2625	2621	2623	2622	**2621**	2621	**0,0012**
		19,7	25,8	24,6	29.6	26	29,6	21.3	98	28,2	25,6	34,68		
ta44	50 x 10	3123	3123	3097	3103	3119	3108	3104	3102	3105	3103	**3097**	3063	**0,0149**
		82.7	80.5	76.5	60.4	174.6	95	75.6	63	176.6	61.7	94,66		
ta63	100 x 5	5206	5213	5213	5213	5213	5193	5193	5205	5213	5205	**5193**	5175	**0,0061**
		283.8	297.4	231.7	181.2	145.6	269.3	221.5	205.2	266.4	195.9	229,8		

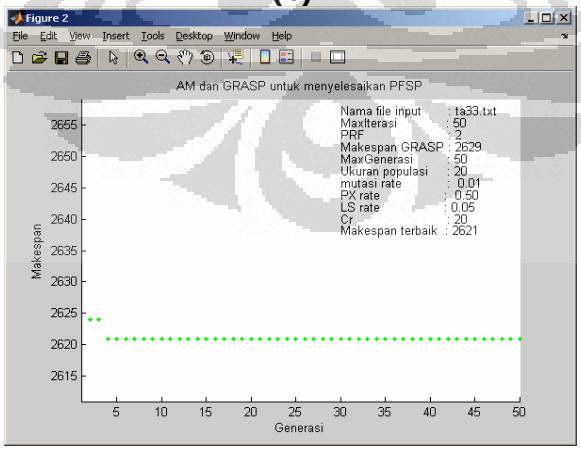
Grafik output hasil terbaik yang diperoleh dari beberapa data masalah diberikan pada Gambar 4.2.



(a)



(b)



(c)

Gambar 4.2 Tampilan output grafik hasil perobaan terbaik masalah ta03(a), ta08(b), dan ta33(c)

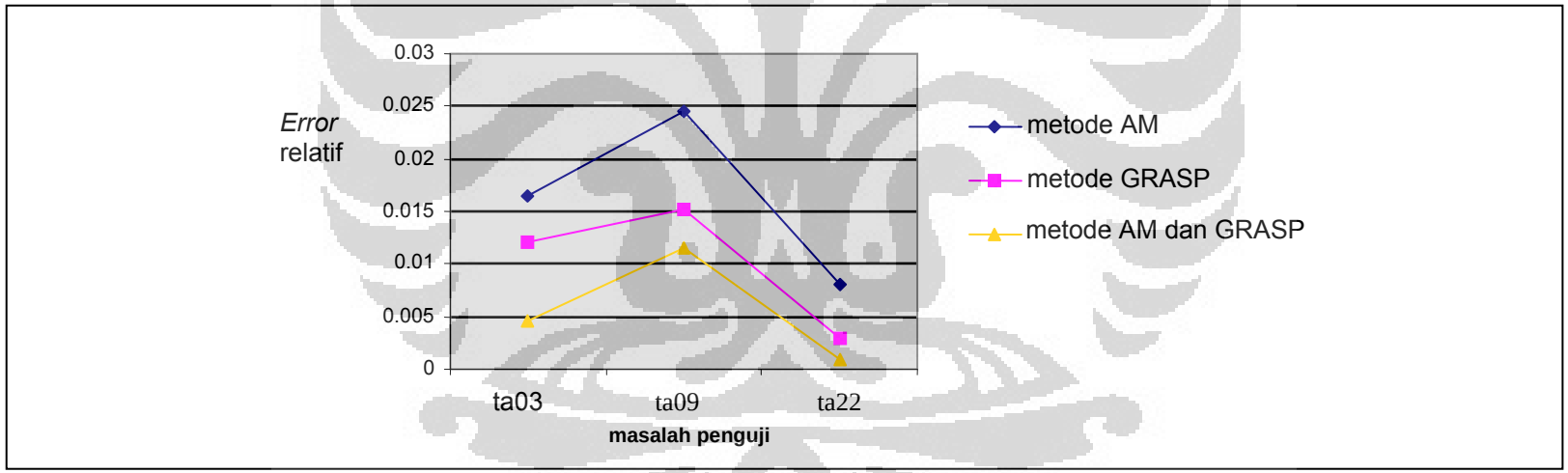
4.3 Hasil Perbandingan

Pada Subbab ini akan diberikan beberapa hasil percobaan yang akan digunakan untuk melihat perbandingan kinerja metode AM dan metode GRASP dengan kombinasi AM dan GRASP. Percobaan untuk metode kombinasi AM dan GRASP telah dilakukan pada Subbab 4.2. Pada Subbab ini dilakukan percobaan metode AM dan metode GRASP untuk 3 masalah penguji, ta09, ta22, dan ta33, dengan kriteria berhentinya adalah CPU *time* dengan besar yang sama dengan rata-rata waktu komputasi yang diperoleh metode AM dan GRASP pada tabel 4.1 untuk masing-masing masalah tersebut.

Hasil percobaan ini diberikan pada tabel 4.2 dan gambar 4.2. Dari hasil percobaan dapat dilihat bahwa rata-rata *error* metode AM dan GRASP lebih baik daripada metode AM dan metode GRASP sendiri-sendiri. Rata-rata *error* metode AM dan metode GRASP sebenarnya masih bisa dikecilkan lagi, jika waktu komputasinya ditambah, Hal ini berarti metode AM dan GRASP lebih cepat konvergen ke solusi optimal dibandingkan dengan metode AM dan metode GRASP sendiri-sendiri.

Tabel 4.2 Nilai *makespan* dari hasil percobaan metode AM dan metode GRASP dalam menyelesaikan PFSP

Nama Data	Metode	*Makespan* dari percobaan ke-										Waktu	BKS	*error* relatif	*error* relatif AM &GRASP
		1	2	3	4	5	6	7	8	9	10				
ta09	GRASP	1252	1255	1247	1240	1252	1237	1233	1253	1240	1240	8,26	1230	0,0121	**0,0046**
	AM	1240	1255	1230	1253	1257	1252	1261	1247	1255	1253			0,0165	
ta22	GRASP	2120	2130	2137	2136	2136	2137	2136	2120	2137	2121	9,29	2099	0,0152	**0,0115**
	AM	2106	2139	2142	2168	2146	2150	2175	2175	2151	2152			0,0245	
ta33	GRASP	2623	2634	2624	2637	2631	2621	2630	2621	2628	2637	34,68	2621	0,0029	**0,001**
	AM	2628	2656	2627	2691	2627	2640	2623	2634	2639	2655			0,0080	



Gambar 4.3 Perbandingan *error* relatif metode AM, metode GRASP dengan metode AM dan GRASP untuk data ta03, ta08, dan ta33

BAB V
KESIMPULAN

Berdasarkan hasil percobaan yang telah dilakukan pada beberapa data masalah penguji dari *Taillard's Benchmark*, disimpulkan bahwa metode AM dan GRASP cukup baik untuk menyelesaikan PFSP. Metode AM dan GRASP menghasilkan jadwal dengan *makespan* yang memiliki *error* relatif tidak lebih dari 2 %. Berdasarkan hasil perbandingan, didapatkan bahwa kombinasi AM dan GRASP lebih cepat konvergen ke solusi optimal dibandingkan metode GRASP atau Algoritma Memetika. Sehingga dapat disimpulkan, kombinasi metode AM dan GRASP menghasilkan kinerja yang lebih baik daripada metode AM dan metode GRASP sendiri-sendiri.

DAFTAR ACUAN

[ABR01] Aiex, R.M., Binato,S., Resende, M.G.C., 2001.*Parallel GRASP with Path Relinking for Job Shop Scheduling,* AT&T Labs Research Technical Report, USA. To appear in Parallel Computing.

[AHM08] El-Bouri, Ahmed. "*A Hybrid Genetic Algorithm for Flowshop Scheduling*." Retrieved 2008 from http://www.Umoncton.ca/cie/conferences/29thconf/29thICCIE/Papers/paper04.pdf

[DGL03] Digaspero, L., 2003. *Local Search Techniques for Scheduling Problems: Algorithms and Software Tools*. Ph.D. Thesis 1061-1071.

[GAL97] Yamada, T., Reeves, C.,1997. *Permutation Flowshop , Scheduling by Genetic Local Search.* IEE conference publication, 232-238, IEE

[MIT96] Murata, T., Ishibuchi, H. & Tanaka, H.,1996. *Genetic algorithms for flowshop scheduling problems*, Computers Ind. Engng., 30(4).

[OBI98] Obitko, M., 1998.*Introduction to Genetic Algorithm*. http://www.obitko.com/tutorials/genetic-algorithms/

[RAV06] Ravetti,M.G Nakamura, F.G. Meneses,C.N. Resende,M.G.C. Mateus, and G.R.Pardalos,P. ,2006.*Hybrid Heuristics for the Permutation Flow Shop Problem*. AT&T Labs Research Technical Report TD-6V9MEV, Shannon Laboratory, Florham Park, NJ 07932 USA.

[RMA06] Ruiz, R., Maroto, C., Alcaraz, J., 2006. *Two new robust genetic algorithms for the flowshop scheduling problem.* OMEGA, the International Journal of Management Science 34, 461–476.

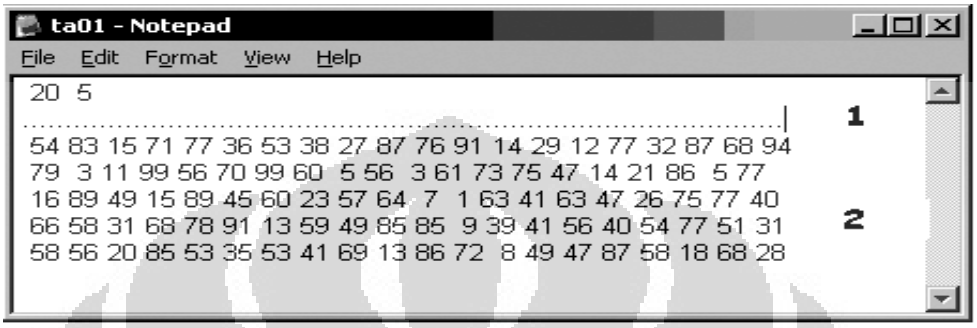
[RYA98] Reeves,C. R., Yamada, T.,1998. *Genetic Algorithms, Path Relinking and the Flowshop Sequencing Problem.* Evolutionary Computation journal (MIT press), Vol.6 No.1, pp. 230

[SED07] Šeda ,Miloš. 2007. *Mathematical Models Of Flow Shop And Job Shop Scheduling Problems.* Proceedings Of World Academy Of Science, Engineering And Technology Volume 25 November 2007 Issn 1307-6884

[STU98] Stutzle,Thomas, 1998.*Applying Iterated Local Search to the Flow Shop Problem.* Technical Report, AIDA-98-04, Darmstad University of Technology, Computer Science Department, Intellctics Group, Darmstad, Germany

LAMPIRAN 1

Contoh File Data Masalah



Contoh di atas adalah salah satu masalah penguji yang diambil dari *Taillard's Benchmark,* yaitu data masalah ta01. Bagian pertama menyatakan ukuran masalah berupa jumlah pekerjaan dan jumlah mesin. Dalam *file* di atas jumlah pekerjaan adalah 20 dan jumlah mesin adalah 5.

Bagian kedua mulai baris kedua hingga baris keenam, menyatakan matriks waktu proses setiap pekerjaan pada masing-masing mesin. Dimana kolom pertama menyatakan waktu proses pekerjaan pertama pada mesin 1 sampai 5, begitu juga dengan kolom kedua hingga kolom ke duapuluh.

LAMPIRAN 2

Listing Program AM Dan GRASP

Untuk Menyelesaikan *Permutation Flow Shop Scheduling Problem*

1. **Fungsi Makespan**

```
function MS = makespan(n,m,job_time,kromosom)
MSpan = zeros( m, 1 );
for i=1:n
    for j=1:m
        if( j < 2 )
              MSpan (j,1) = MSpan(j,1) +
              job_time(kromosom(i),j);
        else
              if(MSpan (j,1) > MSpan (j-1,1))
                   MSpan (j,1) = MSpan(j,1) + job_time(
                   kromosom(i),j);
              else
                   MSpan (j,1) = job_time(kromosom(i),j) +
                   MSpan (j-1,1) ;
              end
        end
    end
end
   MS = MSpan ( m );
```

2. **Fungsi NEH**

```
function kromosom= NEH(n,m,job_time2)
for i=2:n
    if(i == 2)
        % urutan : 1 - 2
        urut = [1;2];
        MS1 = makespan_part(m,job_time2,urut);
        urut_tetap = urut;
        % urutan : 2 - 1
        urut = [2;1];
        MS2 = makespan_part(m,job_time2,urut);
        if(MS2 < MS1)
            urut_tetap = urut;
        end
    else
        for j=1:i
            urut(j) = i;
            l = 1;
            for k=1:i
                if(k ~= j)
                    urut(k) = urut_tetap(l);
                    l = l + 1;
                end
            end
            if(j > 1)
```

```
                if(MS > makespan_part(m,job_time2,urut))
                    MS= makespan_part(m,job_time2,urut);
                    calon_urut_tetap = urut;
                end
            else
                MS= makespan_part(m,job_time2,urut);
                calon_urut_tetap = urut;
            end
        end
        urut_tetap = calon_urut_tetap;
    end
end
kromosom = urut_tetap';
```

3. **Makespan_part untuk NEH**

```
function MS = makespan_part(m,job_time,urut)
% n = Jumlah Pekerjaan
% m = Jumlah Mesin
n = size(urut);
n = n(1);

MS1 = zeros( m, 1 );
for i=1:n
    for j=1:m
        if( j < 2 )
             MS1(j,1) = MS1(j,1) + job_time(urut(i),j  );
        else
            if( MS1(j,1) > MS1(j-1,1))
                MS1(j,1) = MS1(j,1) + job_time(urut(i),j);
            else
                MS1(j,1) =job_time(urut(i),j)+ MS1(j-1,1) ;
            end
        end
    end
end
MS = MS1(end);
```

4. **Fungsi utama GRASPMemetic**

```
function Sol = GRASPMemetic (file_input,MaxIter,PRF,
mu_rate, co_rate,LS_rate, UkPop, MaxG,Cr)
tic
fid  = fopen(file_input,'r');
n    = fscanf(fid,'%d',[1,1]);          % banyak pekerjaan
m    = fscanf(fid,'%d',[1,1]);          % banyak mesin
job_time = fscanf(fid,'%d',[n,m]);      % waktu proses
fclose(fid);

%Sort the job by decreasing sum of processing time

total_time = sum(job_time,2);
[total_time,jobsorted] = sort(total_time,'descend');
job_time2(:,:) = job_time(jobsorted,:);
```

```
% Temperature

T=0.5*(sum(total_time,2))/(n*m*10);

% NEH

Sol_NEH = NEH(n,m,job_time2);
Solution = Sol_NEH;
POOL(1,:)= Sol_NEH;
MS(1)= makespan(n,m,job_time,POOL(1,:));

% Local Search

SolLS = LS(Solution,n,m,job_time);
Solution = SolLS;

% LSInsertion(Solution)

POOL(2,:)= SolLS;
MS(2)= makespan(n,m,job_time,POOL(2,:));

BestSolution = POOL(1,:);
if MS(2)< MS(1)
    BestSolution = POOL(2,:);
end
MS_Best = makespan(n,m,job_time,BestSolution);

CurrSol = BestSolution;
MS=makespan(n,m,job_time,CurrSol);
index_pool = 2;

NonImproves = 0;

% Main Iteration of GRASP

for i=1:MaxIter

    % Perturbation

    Solution = Perturbation(CurrSol);

    % Iterated Local Search

    SolLS = LS(Solution,n,m,job_time);
    MS_LS = makespan(n,m,job_time,SolLS);
    Solution= SolLS;

    % PoolInsertion(Solution, Proximity)

    if (proximity(Solution, POOL) == 1 & MS_LS < MS_Best)
        index_pool = index_pool+1;
        POOL(index_pool,:) = Solution;
    end

    % Path Relinking dan Acceptance Criterion
```

```
    acak= floor((index_pool)*rand(1)+1);
    PoolSolution =POOL(acak,:);
    if mod(i, PRF)==0,

SolPR=PR(BestSolution,PoolSolution,n,m,job_time,total_time)
;
        MS_PR = makespan(n,m,job_time,SolPR);
        if (proximity(SolPR, POOL) == 1 & MS_PR < MS_Best)
            index_pool = index_pool+1;
            POOL(index_pool,:) = SolPR;
        end

    % PR Acceptance Criterion

    if MS_PR<MS
        CurrSol = SolPR;
    else
        random=rand();
        if random < ((exp(-MS_PR-MS))/T)
            CurrSol = SolPR;
        end
    end
    end
    MS=makespan(n,m,job_time,CurrSol);

    % ILS Acceptance Criterion

    if MS_LS<MS
        CurrSol = SolLS;
    else
        random=rand();
        if random < ((exp(-MS_LS-MS))/T)
            CurrSol = SolILS;
        end
    end
    MS=makespan(n,m,job_time,CurrSol);
    if MS < MS_Best
        NonImproves = 0;
    else
        NonImproves = NonImproves + 1;
    end
    if NonImproves == 10

    %restartPOOL

    % NEH

    Sol_NEH = NEH(n,m,job_time2);
    Solution = Sol_NEH;
    POOL(1,:)= Sol_NEH;
    MS(1)= makespan(n,m,job_time,POOL(1,:));

    % Local Search
```

```
    SolLS = LS(Solution,n,m,job_time);
    Solution = SolLS;

    % LSInsertion(Solution)

    POOL(2,:)= SolLS;
    MS(2)= makespan(n,m,job_time,POOL(2,:));

    BestSolution = POOL(1,:);
    if MS(2)< MS(1)
        BestSolution = POOL(2,:);
    end
    MS_Best = makespan(n,m,job_time,BestSolution);

    CurrSol = BestSolution;
    MS=makespan(n,m,job_time,CurrSol);
    index_pool = 2;

    end
end


for j=1:index_pool
    MSpan(j) = makespan(n,m,job_time,POOL(j,:));
end
[MSpan,Ind] = sort(MSpan, 'ascend');
POOL2(:,:) = POOL(Ind,:);
MS_GRASP = MSpan(1);
Sol = Memetic(POOL2,MS_GRASP, file_input, m,
job_time,MaxIter, PRF, mu_rate, co_rate,LS_rate, UkPop,
MaxG,Cr);
finished = toc;
fprintf('waktu komputasi Hybrida :%d  menit  %.1f detik\n',
round(finished/60), mod(finished,60));
```

5. **Fungsi Iterated Local Search**

Local Search

```
function SolLS = LS(Solution,n,m,job_time)

MS = makespan(n,m,job_time,Solution);
improv=MS;

r = 1;
InitialSolution = Solution;
while r <= n
   for i=1:n
       Solution = LSI(Solution,InitialSolution(r),i);
       MS = makespan(n,m,job_time,Solution);
       if  MS >improv
           ;
       else
           InitialSolution = Solution;
```

```
            improv = MS;
            break;
        end
    end
    r = r+1;
end
    SolLS = InitialSolution;
```

Perturbation

```
function Solution = Perturbation(Solution)
    a = 1;
    b = 1;

    while a==b
        a = floor(length(Solution)*rand(1))+1;
        b = floor(length(Solution)*rand(1))+1;
    end
    SolutionP=Solution;
    Sol = SolutionP(a);
    SolutionP(a)=SolutionP(b);
    SolutionP(b)=Sol;
    Solution = SolutionP;
```

Local Search Insertion

```
function SolLSI = LSI(P,pindah,j)
i = find(P == pindah);
if i > j
    P(j+1:i) = P(j:i-1);
    P(j) = pindah;
elseif i < j
    P(i:j-1) = P(i+1:j);
    P(j) = pindah;
end
SolLSI = P;
```

Proximity

```
function prox = proximity(P, POOL)
[c n] = size(POOL);
beda1 = find(P ~= POOL(1,:));
beda = length(beda1);

i = 2;
batas = floor(0.2*n);
kosong = sum(POOL(i,:) ~= zeros(1,n));
while (kosong ~= 0) & (beda > batas) &(i <= c)
    beda1 = find(P ~= POOL(i,:));
    beda = length(beda1);
    i = i+1;
    if i <= c
        kosong = sum(POOL(i,:) ~= zeros(1,n));
    end
end
```

```
if beda > batas
    prox = 1;
else
    prox = 0;
end
```

6. **Fungsi Path Relinking**

```
function
SolPR=PR(BestSolution,PoolSolution,n,m,job_time,total_time)
MS1 = makespan(n,m,job_time,BestSolution);
sama = (PoolSolution == BestSolution);
solusi = BestSolution;
span_solusi = 1;
if(n - sum(sama) >= 3 && sum(sama) < n)
    i = 3;
elseif(n - sum(sama) < 3)
    i = 1;
else
    i = 0;
end;

posisi = [];
for j=1:n
    if sama(i) == 0
        posisi = [posisi j];
    end
end

k = 1;
while k <= 5 & i ~= 0 & length(posisi) ~= 0
    tukar1=1;
    tukar11=1;
    while i > 0
        while tukar11==tukar1
            tukar11 = randsrc(1,1,posisi);
        end
        tukar1=tukar11;
        tukar2 = find(BestSolution ==
PoolSolution(tukar1));
        solusi1(i,:) = Swap(BestSolution, tukar1, tukar2);
        span_solusi1(i,:) =
makespan(n,m,job_time,solusi1(i,:));
        i = i-1;
    end
    %Cari yang terbaik
    index = find(span_solusi1 == min(span_solusi1));
    index = index(1);
    solusi(k,:) = solusi1(index,:);
    BestSolution = solusi(k,:);
    span_solusi(k,:) = makespan(n,m,job_time,solusi(k,:));
```

```
    sama = (PoolSolution == BestSolution);
    if(n - sum(sama) >= 3 && sum(sama) < n)
        i = 3;
    elseif(n - sum(sama) < 3)
        i = 1;
    else
        i = 0;
    end;

    posisi = [];
    for j=1:n
        if sama(i) == 0
            posisi = [posisi j];
        end
    end
    k = k+1;
end

index = find(span_solusi == min(span_solusi));
index = index(1);
SolPR = solusi(index,:);
MS=makespan(n,m,job_time,SolPR);
```

Swap

```
function Swapkrom = Swap(kromosom,a,b)

dummy = kromosom(a);
kromosom(a) = kromosom(b);
kromosom(b) = dummy;

Swapkrom = kromosom;
```

7. **Fungsi Algoritma Memetika**

```
function P = Memetic(POOL,MS_GRASP,
file_input,m,job_time,MaxIter, PRF, mu_rate,
co_rate,LS_rate, UkPop, MaxG,Cr)
 [index_pool n] = size(POOL);

%Populasi awal

if index_pool<=15
    for j=1:index_pool
        P(j,:) = POOL(j,:);
    end
    for i = index_pool+1:UkPop
        P(i,:) = randperm(n);
    end
else
    for j=1:15
        P(j,:) = POOL(j,:);
    end
    for i = 16:UkPop
        P(i,:) = randperm(n);
```

```
    end
end
for j=1:UkPop
    MSpan(j) = makespan(n,m,job_time,P(j,:));
end
B = min(MSpan)+30;

%Local Search (Insertion Mutation)
for i = 1:UkPop
   P(i,:) = LS(P(i,:), n, m, job_time);
end

for i=1:UkPop
    MSpan(i) = makespan(n,m,job_time,P(i,:));
end
min_MSpan1 = min(MSpan);
count = 0;

% Loop evolusi

    for iter=1:MaxG;
        fitness(1) = EvaluasiIndividu(MSpan, MSpan(1));
        MaxF = fitness(1);
        MinF = fitness(1);
        BestIndex = 1;
        BestX = P(1,:);

        for j=2:UkPop
            kromosom = P(j,:);
            fitness(j) = EvaluasiIndividu(MSpan, MSpan(j));

            if(fitness(j) > MaxF)
                MaxF = fitness(j);
                BestIndex = j;
                BestX = kromosom;
            end

            if(fitness(j)< MinF)
                MinF = fitness(j);
            end
        end

        TempP = P;

        % Elitisme:

        if mod(UkPop, 2)== 0,       % ukuran populasi genap
            IterasiMulai = 3;
            TempP(1,:) = P(BestIndex,:);
            TempP(2,:) = P(BestIndex,:);
        else                        % ukuran populasi ganjil
            IterasiMulai = 4;
            TempP(1,:) = P(BestIndex,:);
            TempP(2,:) = P(BestIndex,:);
        end
```

```
        LinearFitness =
LinearFitnessRanking(UkPop,fitness,MaxF,MinF);

        % Roulette-wheel selection dan Path Crossover

        for jj = IterasiMulai:2:UkPop,
            IP1 = 1;
            IP2 = 1;
            while IP1 == IP2
            IP1 = RouletteWheel(UkPop,LinearFitness);
            IP2 = RouletteWheel(UkPop,LinearFitness);
            end
            rand_co=rand(1);
            if (rand_co < co_rate),
                [Anak1 Anak2]= PX(n, m,
job_time,P(IP1,:),P(IP2,:));
                TempP(jj, :) = Anak1(1,:);
                TempP(jj+1,:) = Anak2(1,:);
            else
                TempP(jj,:) = P(IP1,:);
                TempP(jj+1,:) = P(IP2,:);

            end

         end

        % Mutasi dilakukan pada kromosom terbaik

         rand_mu = rand(1);
         if(rand_mu < mu_rate)
             P(2,:) = Perturbation(TempP(2,:));
         end

        %Local Search

        rand_LS = rand(1);
        if(rand_LS < LS_rate)
            for i = 1:UkPop
                TempP(i,:) = LS(P(i,:), n, m, job_time);
            end
        end


        P=TempP;

        %Cold Restart Scheme

        min_makespan(iter) = min(MSpan);
        if(iter > 1)
            if(min_makespan(iter) == min_makespan(iter-1))
                count = count + 1;
            end
            if(count > Cr)
                [MSpan,j] = sort(MSpan);
```

```
                TempP(:,:) = P(j,:);
                P = ColdRestart (TempP, MSpan,UkPop);
            end
        end

        for j=1:UkPop
            MSpan(j) = makespan(n,m,job_time,P(j,:));
        end
        min_MSpan(iter) = min(MSpan);
    end
    MinSpan = min(MSpan);
    BestIndex = find(MSpan == MinSpan);
    BestIndex = BestIndex(1);
    fprintf('\n BestSolution :\n');
    for m=1:n
    fprintf(' %d ', P(BestIndex,m));
    end
    fprintf('--> %d\n',MSpan(BestIndex));
    MS_best = min(MSpan);
    C = B - MSpan_best;
```

Evaluasi Individu

```
function Fitness = EvaluasiIndividu(M,Mi)
    max_length = max(M);
    small_int = min(M);

      Fitness  = (max_length - Mi) + small_int;
```

Linear Fitness Ranking

```
function LFR =
LinearFitnessRanking(UkPop,Fitness,MaxF,MinF)
[SF,IndF] = sort(Fitness);

% LinearFitness = nilai fitness baru hasil penskalaan
for rr=1:UkPop,
  LFR(IndF(UkPop-rr+1)) = MaxF-(MaxF-MinF)*((rr-1)/(UkPop-
1));
end
```

Roulette Wheel

```
function Pindex = RouletteWheel(UkPop, LinearFitness)

JumFitness = sum(LinearFitness);
KumulatifFitness  = 0;
RN = rand;
j = 1;
Pindex = 1;
while j <= UkPop,
    KumulatifFitness = KumulatifFitness + LinearFitness(j);

    if (KumulatifFitness/JumFitness) > RN,
        Pindex = j;
```

```
        break;
    end
    j = j + 1;
end
```

8. **Path Crossover**

```
function [Anak1 Anak2] = PX(n,m,job_time,P1,P2)
a=floor(n*rand(1))+1;
i = a;
while i <= n+a-1
    O1=P1;
    O2=P2;
    if i<=n
        if P1(i)==P2(i)
            ;
        else
            Index1 = find(P1 == P2(i));
            Index2 = find(P2 == P1(i));
            O1 = Swap(P1, Index1, Index2);
            O2 = Swap(P2, Index1, Index2);
        end
    else
        k=i-n;
        if P1(k)==P2(k)
            ;
        else
            Index1 = find(P1 == P2(k));
            Index2 = find(P2 == P1(k));
            O1 = Swap(P1, Index1, Index2);
            O2 = Swap(P2, Index1, Index2);
        end
    end
    MP1 = makespan(n,m,job_time,P1);
    MP2 = makespan(n,m,job_time,P2);
    MO1 = makespan(n,m,job_time,O1);
    MO2 = makespan(n,m,job_time,O2);

    if MO1<=MP1
        P1=O1;
    end
    if MO2<=MP2
        P2=O2;
    end
    i = i+1;
end
Anak1=P1;
Anak2=P2;
```

9. **Mutasi**

```
function Solution = Perturbation(Solution)
a = 1;
b = 1;
```

```
while a==b
      a = floor(length(Solution)*rand(1))+1;
      b = floor(length(Solution)*rand(1))+1;
end
     SolutionP=Solution;
     Sol = SolutionP(a);
     SolutionP(a)=SolutionP(b);
     SolutionP(b)=Sol;
     Solution = SolutionP;
```

10. **Cold Restart**

```
function TempP = ColdRestart (TempP, MSpan,UkPop)
   [m n] = size(TempP);
   [MSpan,j] = sort(MSpan);
   TempP_new(:,:) = TempP(j,:);

   for i=1:3
       TempP(i,:) = TempP_new(i,:);
   end

   for j=4:4+round(0.45*UkPop)
       acak = floor(3*rand(1))+1;
       randP = TempP(acak,:);
       TempP(j,:) = Perturbation(randP);
   end

   for i=j+1:UkPop
       TempP(i,:) = randperm(n);
   end
```