BAB 4

DESAIN DAN PENGEMBANGAN ALGORITMA

Bab ini menguraikan desain dan pengembangan dari dua algoritma utama untuk menyelesaikan permasalahan penjadwalan. Algoritma pertama merupakan algoritma yang ditujukan untuk mampu menghasilkan solusi *feasible* dengan memenuhi semua *hard constraint* dari suatu permasalahan penjadwalan. Sementara itu, algoritma kedua merupakan algoritma yang bertujuan untuk mengoptimasi solusi *feasible* yang telah dihasilkan oleh algoritma pertama. Proses optimasi dijalankan dengan mengurangi jumlah pelanggaran *soft constraint* dalam batas waktu yang ditentukan dan tetap menjaga solusi dalam keadaan *feasible*.

Kedua algoritma dikembangkan berdasarkan pendekatan hiper-heuristik *selection-perturbation*, dengan menggunakan bentuk dasar dari algoritma Iterated Local Search (ILS). Pada algoritma untuk mencari solusi *feasible*, pengembangan dilakukan dengan mendesain algoritma untuk dapat terus menjalankan proses pencarian tanpa adanya penolakan terhadap solusi baru. Berdasarkan ide tersebut algoritma ini dinamakan sebagai algoritma Progressive Acceptance Iterated Local Search (PA-ILS). Sementara itu pada algoritma untuk proses optimasi, algoritma didesain dengan menggabungkan strategi nilai ambang batas (*threshold*) dengan algoritma ILS. Kelebihan utama pada algoritma ini ada pada aspek generalitasnya dengan nilai *threshold* yang mampu beradaptasi dengan permasalahan yang ada, sehingga tidak membutuhkan pengaturan nilai parameter. Berdasarkan karakteristik tersebut, algoritma ini dinamakan Adaptive Threshold-Iterated Local Search (AT-ILS).

Untuk penjelasan lebih detail dari kedua algoritma tersebut, Subbab 4.1, 4.2, dan 4.3 menjabarkan rasionalitas dan desain pengembangan algoritma. Selain itu, pembahasan *Low Level Heuristic* (LLH) yang digunakan oleh kedua algortima dibahas pada Subbab

4.4. Terakhir, pembahasan proses pengaturan nilai parameter untuk algoritma PA-ILS dijabarkan dalam Subbab 4.5.

4.1 Rasionalitas Desain Algoritma

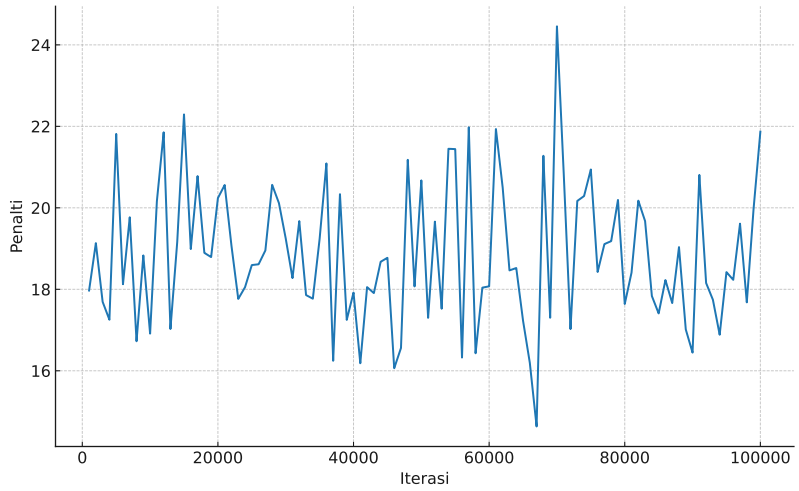
Bagian ini menjelaskan rasionalitas dari desain algotrima yang dikembangkan. Subbab 4.1.1 menjelaskan pentingnya keseimbangan eksplorasi dan eksploitasi yang menjadi ide dasar dari pengembangan algoritma ini. Selanjutnya, Subbab 4.1.2 dan 4.1.3 menjelaskan alasan pemilihan metode dan pendekatan hiper-heuristik *Selection-Perturbation* serta pemilihan algoritma ILS yang digunakan sebagai bentuk dasar dalam pengembangan algoritma. Terakhir, Subbab 4.1.4 menjelaskan pendekatan dalam mendesain algoritma yang dikembangkan pada penelitian ini.

4.1.1 Keseimbangan antara Eksplorasi dan Eksploitasi

Dalam mengembangkan algoritma untuk permasalahan optimasi, dibutuhkan keseimbangan antara proses eksplorasi dan eksploitasi dalam proses pencarian solusi. Keseimbangan ini tidak berarti bahwa jumlah proses eksplorasi dan eksploitasi harus sama, tetapi kedua proses tersebut mampu menyesuaikan dengan karakteristik permasalahan.

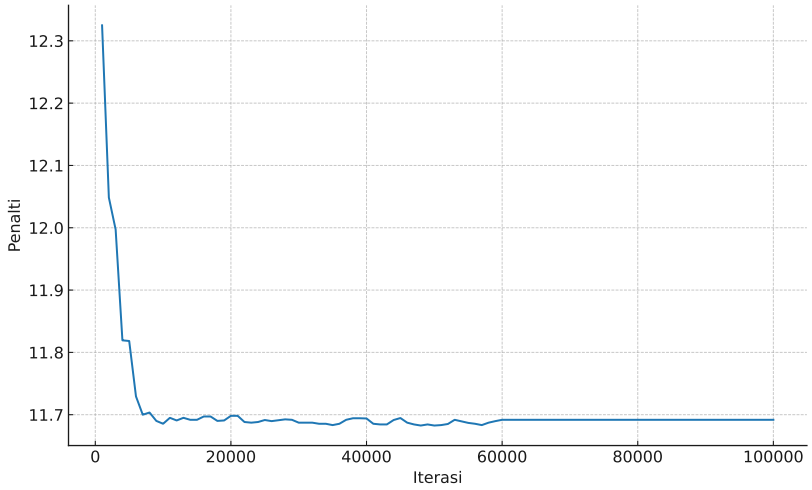
Algoritma yang terlalu banyak melakukan eksplorasi dapat menghambat konvergensi solusi menuju hasil yang lebih optimal. Hasilnya solusi akan terjebak dalam rentang tertentu tanpa peningkatan signifikan. Sebagai ilustrasi, Gambar 4.1.1 menunjukkan proses pencarian solusi pada algoritma Threshold Acceptance dengan nilai ambang batas yang terlalu besar, yang menyebabkan proses eksplorasi menjadi berlebihan. Akibatnya, pencarian solusi menjadi stagnan dalam suatu rentang tertentu dan sulit diarahkan menuju hasil yang lebih baik. Hal ini terjadi karena saat algoritma mencoba menurunkan nilai penalti, proses eksplorasi yang berlebihan sering kali menyebabkan nilai penalti kembali meningkat, sehingga pencarian solusi hanya berfluktuasi dalam rentang tertentu tanpa mencapai perbaikan signifikan.

Sebaliknya, algoritma yang terlalu banyak melakukan eksploitasi cenderung terjebak pada kondisi yang disebut *local optima*. Kondisi ini terjadi ketika algoritma tidak mengizinkan penerimaan solusi yang sedikit lebih buruk untuk eksplorasi ruang pencarian. Namun, algoritma tidak mampu menemukan solusi yang lebih baik. Hal ini menyebabkan



Gambar 4.1.1: Ilustrasi Proses Pencarian dengan Tahapan Eksplorasi yang Berlebihan

solusi berhenti pada titik tertentu hingga akhir proses pencarian. Gambar 4.1.2 menunjukkan contoh algoritma Threshold Acceptance dengan nilai ambang batas yang terlalu kecil, yang menyebabkan minimnya proses eksplorasi. Pada ilustrasi ini, solusi tidak bergerak sejak iterasi sekitar 60000. Algoritma tidak dapat menemukan solusi yang lebih baik, namun algoritma juga tidak mengizinkan penerimaan solusi yang sedikit lebih buruk untuk keluar dari titik tersebut. Hasilnya proses pencarian terjebak dalam kondisi *local optima*.



Gambar 4.1.2: Ilustrasi Proses Pencarian dengan Tahapan Eksploitasi yang Berlebihan

4.1.2 Penggunaan Pendekatan Hiper-Heuristik *Selection-Perturbation*

Berdasarkan hasil pemeringkatan penelitian terdahulu pada Subbab 2.2, pendekatan metaheuristik (S) menunjukkan hasil paling baik dan konsisten. Dalam permasalahan penjadwalan, pendekatan metaheuristik (S) dimulai dengan memodifikasi satu solusi menggunakan salah satu *neighborhood operator* yang umumnya dipilih secara acak. Solusi baru yang dihasilkan kemudian dievaluasi untuk menentukan apakah solusi tersebut akan diterima. Jika diterima, solusi ini disimpan dan digunakan dalam iterasi berikutnya. Jika tidak, solusi tersebut dibuang dan solusi sebelumnya tetap dipertahankan.

Secara umum, pendekatan metaheuristik (S) dalam menyelesaikan permasalahan penjadwalan tidak berbeda secara signifikan dari pendekatan hiper-heuristik (S-P). Pendekatan hiper-heuristik (S-P) memiliki tiga komponen utama, yaitu:

- kumpulan LLH: kumpulan dari LLH yang dapat digunakan untuk memodifikasi solusi.
- LLH *selection*: strategi dalam memilih LLH pada setiap iterasi.
- *move acceptance*: strategi untuk menentukan penerimaan solusi.

Ketiga komponen pada hiper-heuristik (S-P) pada dasarnya menyerupai komponen pada pendekatan metaheuristik (S). Pada pendekatan hiper-heuristik (S-P), komponen *move acceptance* umumnya menggunakan algoritma yang sama dengan algoritma pada pendekatan metaheuristik (S) seperti Simulated Annealing, ILS, dan Great Deluge (Demeester et al., 2012; Soria-Alcaraz et al., 2016). Sementara itu pada komponen LLH *selection*, pemilihan LLH dapat dilakukan dengan beberapa strategi tertentu atau bisa dilakukan secara acak (Pillay, 2016*b*). Penggunaan strategi pemilihan LLH secara acak merupakan strategi yang serupa dengan pendekatan metaheuristik (S), yang juga cenderung memilih *neighborhood operator* secara acak. Istilah LLH dan *neighborhood operator* dalam algoritma penjadwalan umumnya merujuk pada konsep yang sama yaitu metode untuk melakukan perubahan solusi. Hal ini menunjukan pendekatan metaheuristik (S) dan hiper-heuristik (S-P) pada dasarnya menjalankan proses serupa dalam konteks penjadwalan.

Namun, secara konseptual terdapat perbedaan antara pendekatan metaheuristik (S) dan hiper-heuristik (S-P). Pada pendekatan metaheuristik (S), pengembangan desain

algoritma dilakukan secara menyeluruh. Sementara pada pendekatan hiper-heuristik (S-P), desain algoritma dibagi menjadi tiga komponen utama seperti yang telah dijabarkan sebelumnya. Struktur desain algoritma yang dibagi menjadi tiga komponen memberikan keunggulan berupa kemudahan pemahaman serta fleksibilitas yang lebih tinggi dalam pengembangan lebih lanjut. Berdasarkan pertimbangan tersebut, penelitian ini memilih untuk mengembangkan desain algoritma menggunakan pendekatan hiper-heuristik (S-P).

4.1.3 Pemilihan Algoritma ILS

Algoritma ILS menjalankan tiga tahapan utama secara berulang, yaitu *perturbation*, *local search*, dan *move acceptance*. Algoritma ILS sebenarnya lebih tepat disebut sebagai sebuah kerangka kerja, karena algoritma ini hanya menyediakan struktur dasar berupa tiga tahapan tersebut tanpa menjelaskan secara rinci bagaimana masing-masing tahapan, seperti *perturbation* dan *local search*, harus dijalankan atau strategi apa yang diterapkan dalam *move acceptance*. Oleh karena itu, algoritma ILS sangat cocok dijadikan dasar dalam pengembangan algoritma yang dikembangkan dalam penelitian ini.

Hasil penelitian sebelumnya menunjukkan bahwa pengembangan penelitian menggunakan algoritma ILS menunjukkan hasil yang menjanjikan. Dalam aspek pencarian solusi *feasible*, studi oleh Song et al. (2018) dan Goh et al. (2020) menunjukkan bahwa algoritma ini dapat menghasilkan solusi terbaik dibandingkan dengan penelitian lainnya. Sementara itu, pada aspek optimasi, penelitian oleh Soria-Alcaraz et al. (2016) menunjukkan hasil yang sangat menjanjikan pada *benchmark International Timetabling Competition* (ITC) 2007 *Track* 3 dengan memperoleh peringkat terbaik dari 7 penelitian. Temuan-temuan ini mendukung pemilihan ILS sebagai dasar pengembangan algoritma dalam penelitian ini, yang memungkinkan untuk menghasilkan algoritma yang efektif dalam menghasilkan solusi *feasible* dan juga mengoptimasi solusi.

4.1.4 Pendekatan Desain Algoritma PA-ILS dan AT-ILS

Dalam permasalahan menemukan solusi *feasible*, algoritma memiliki tujuan untuk menghasilkan solusi dengan tidak ada satupun pelanggaran terhadap *hard constraint*. Tujuan dalam permasalahan mencari solusi *feasible* merupakan tujuan yang jelas dan dapat diukur. Sementara itu, pada proses optimasi, tujuan utamanya adalah meminimalkan jumlah pelanggaran *soft constraint*. Namun, nilai akhir yang menjadi tujuan dalam proses

optimasi tidak diketahui akibat dari permasalahan penjadwalan yang tergolong sebagai permasalahan NP-Hard. Hal ini menyebabkan tidak adanya nilai yang jelas untuk digunakan sebagai target dari proses optimasi.

Pada proses pencarian solusi *feasible*, solusi yang tidak *feasible* umumnya tidak digunakan dan tidak dipertimbangkan lebih lanjut. Sebagai contoh, jika terdapat dua solusi yang tidak *feasible*, solusi pertama gagal menjadwalkan tiga mahasiswa dan solusi kedua gagal menjadwalkan enam mahasiswa, maka umumnya kedua solusi tersebut akan diabaikan dan kembali mencoba untuk menemukan solusi yang *feasible*. Sebaliknya, dalam proses optimasi, semua solusi yang dihasilkan tetap memungkinkan untuk dipertimbangkan dan digunakan karena solusi tersebut *feasible*.

Perbedaan ini mempengaruhi pendekatan yang digunakan dalam mengembangkan algoritima PA-ILS dan AT-ILS. Algoritma PA-ILS didesain untuk mampu mencapai sasaran yang konkret yaitu menemukan solusi dengan tidak ada satupun pelanggaran terhadap *hard constraint*. Sementara algoritma AT-ILS didesain untuk lebih fleksibel dengan mampu mengurangi pelanggaran terhadap *soft constraint* tanpa memiliki nilai target yang pasti.

4.1.4.1 Pendekatan Desain Algoritma PA-ILS

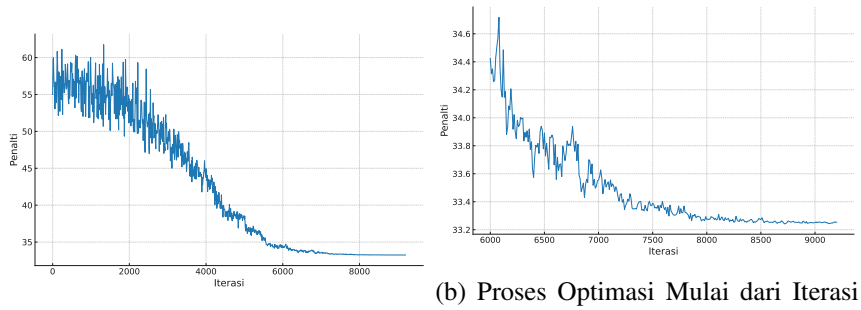
Dalam mengembangkan algoritma PA-ILS untuk mencari solusi *feasible*, algoritma didesain dengan strategi pencarian yang tidak terpaku pada satu titik solusi yang baik. Hal ini diakibatkan solusi yang tidak *feasible* tidak dapat digunakan walaupun hanya beberapa kegiatan saja yang memiliki konflik atau belum terjadwalkan. Berdasarkan hal tersebut, algoritma PA-ILS didesain untuk melakukan eksplorasi secara kontinu dalam proses pencarian solusi dan tidak terpaku pada satu solusi yang baik namum tidak *feasible*.

Untuk menerapkan strategi eksplorasi yang kontinu, algoritma dikembangkan dengan hanya menggunakan tahapan *perturbation* dan *local search*, tanpa menggunakan tahapan *move acceptance*. Dengan menghilangkan tahapan *move acceptance*, setiap solusi yang dihasilkan dari *perturbation* dan *local search* selalu diterima. Sementara itu, untuk mengontrol agar perubahan solusi tidak terlalu menjauhi dari target, perubahan solusi pada tahap *perturbation* tidak dilakukan secara acak sepenuhnya. Strategi pada tahap ini memungkinkan menjalankan dua strategi yang berbeda. Strategi pertama memilih slot

waktu secara acak seperti tahapan *perturbation* pada umumnya. Strategi kedua memilih slot waktu dengan konflik paling sedikit. Pemilihan ini didasarkan pada nilai probabilitas yang akan dijelaskan lebih lanjut pada bagian 4.2 dan 4.2.3. Dengan desain strategi ini, tahapan *perturbation* lebih sering menghasilkan solusi yang mendekati kondisi *feasible*, meskipun sesekali solusi dapat bergerak agak jauh untuk mencegah pencarian terjebak di tempat yang sama.

4.1.4.2 Pendekatan Desain Algoritma AT-ILS

Penelitian sebelumnya menunjukkan bahwa algoritma Simulated Annealing merupakan algortima yang menunjukkan hasil yang paling menjanjikan pada berbagai jenis *benchmark*. Untuk mengetahui mengapa algoritma Simulated Annealing mampu menghasilkan hasil yang baik, penelitian ini melakukan pengujian awal untuk memahami kelebihan dari algoritma Simulated Annealing. Pengujian dilakukan dengan melihat bagaimana proses pencarian pada algortima Simulated Annealing dan juga pada algoritma lainnya yaitu Late Acceptance Hill Climbing (LAHC) dan Threshold Acceptance. Gambar 4.1.3 menunjukkan ilustrasi proses optimasi dari algoritma Simulated Annealing dan Gambar 4.1.4 menunjukkan ilustrasi proses optimasi pada algoritma LAHC dan Threshold Acceptance.

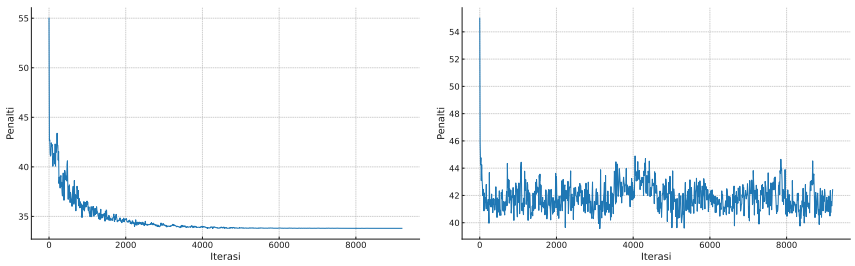


(a) Seluruh Proses Optimasi

(b) Proses Optimasi Mulai dari Iterasi 6000

Gambar 4.1.3: Ilustrasi dari Proses Optimasi Pada Algoritma Simulated Annealing

Dari ilustrasi ini, algoritma Simulated Annealing memiliki proses eksplorasi dan eksploitasi yang lebih berimbang. Dalam setiap rentang iterasi, proses eksplorasi dan eksploitasi dilakukan namun solusi akan ditekan untuk menurun secara perlahan. Pada Gambar 4.1.3b menunjukkan bahwa dari iterasi ke-6000 hingga akhir proses pencarian, proses eksplorasi dan eksploitasi tetap dilakukan namun solusi tetap terus ditekan turun secara perlahan.



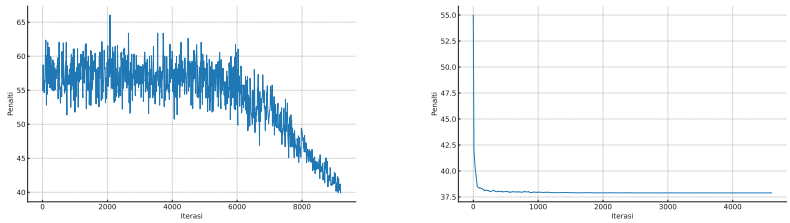
(a) LAHC (b) *Threshold Acceptance*

Gambar 4.1.4: Ilustrasi dari Proses Optimasi Pada Algoritma LAHC dan Threshold Acceptance

Berbeda dengan algoritma Simulated Annealing, algoritma LAHC melakukan penekanan penurunan solusi yang lebih besar pada awal iterasi dan terus menekan solusi hingga akhir iterasi. Walaupun pada algoritma LAHC proses eksplorasi dan eksploitasi tetap ada pada setiap rentang iterasi, namun proses ini jelas lebih didominasi oleh proses eksploitasi dengan solusi yang turun sangat cepat dibandingkan algoritma Simulated Annealing. Sementara pada algoritma Threshold Acceptance, penekanan solusi di awal iterasi sangat cepat menunjukkan dominasi proses eksploitasi. Namun pada saat mencapai nilai tertentu, solusi hanya bergerak pada rentang penalti tertentu. Hal ini diakibatkan dari proses eksplorasi yang terlalu banyak sehingga solusi tidak dapat ditekan turun.

Dari perbandingan ini, algoritma Simulated Annealing menunjukkan strategi yang lebih seimbang dalam menjalankan eksplorasi dan eksploitasi. Namun permasalahan pada algoritma Simulated Annealing ada pada dua nilai parameter yaitu *temperature* dan *cooling rate* yang harus dilakukan pengaturan nilai dengan tepat agar bisa memberikan hasil yang maksimal. Ketidaktepatan nilai parameter akan menyebabkan algoritma terlalu banyak melakukan proses eksplorasi yang berakibat solusi terjebak pada rentang tertentu seperti yang diilustrasikan pada Gambar 4.1.5a. Sebaliknya, ketika algoritma menjalankan terlalu banyak menjalankan proses eksploitasi, solusi akan terjebak dalam kondisi *local optima* seperti yang diilustrasikan pada Gambar 4.1.5b.

Berdasarkan hal tersebut, algoritma AT-ILS didesain dengan mengupayakan strategi ekplorasi dan eskploitasi yang seimbang menyerupai algoritma Simulated Annealing. Proses eksplorasi dan eksploitasi didesain agar mampu berjalan bergantian dan secara keseluruan mampu menekan solusi untuk mengurangi penalti secara perlahan. Algoritma ini didesain dengan tidak membutuhkan satupun pengaturan nilai parameter. Hal ini



(a) Terlalu Banyak Explorasi Solusi (b) Terlalu Banyak Exploitasi Solusi

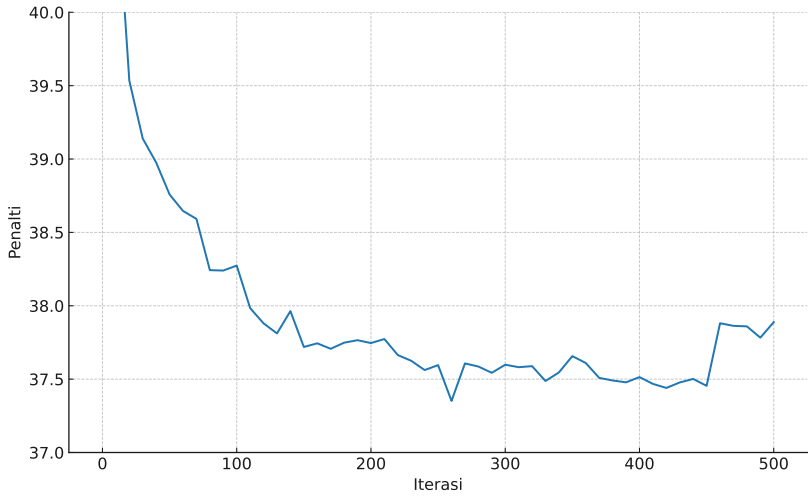
Gambar 4.1.5: Ilustrasi dari Algoritma Simulated Annealing yang Tidak Menggunakan Nilai Parameter yang Tepat

memperbaiki kekurangan pada algoritma Simulated Annealing dan juga meningkatkan tingkat generalitas dari algoritma yang dikembangkan.

Selain itu, dari pengembangan yang sudah dilakukan pada penelitian sebelumnya (Premananda, Muklason and Hutama, 2022; Premananda, Tjahyanto and Muklason, 2022), terdapat kemungkinan pencarian solusi bergerak ke arah yang menjauhi dari solusi terbaik yang berhasil ditemukan. Pencarian solusi juga memungkinkan tidak pernah kembali kearah solusi terbaik yang pernah ditemukan hingga akhir iterasi. Sebagai contoh Gambar 4.1.6 mengilustrasikan bagaimana nilai terbaik didapatkan berkisar pada iterasi ke-250, namun setelah itu proses pencarian tidak pernah berhasil mendekati nilai terbaik yang ditemukan. Bahkan diakhir iterasi, solusi semakin bergerak menjauhi solusi terbaik. Berdasarkan hal tersebut, algoritma didesain untuk dapat kembali pada titik solusi terbaik ditemukan. Hal ini untuk mencegah algoritma bergerak terlalu jauh dari solusi terbaik dan diharapkan algortima bisa menghasilkan solusi terbaik baru.

4.2 Desain Algoritma PA-ILS

Desain algoritma PA-ILS ditunjukan oleh Algoritma 2. Untuk menghasilkan proses pencarian solusi yang terus berjalan, algoritma didesain hanya memiliki dua tahapan utama, yaitu tahapan *perturbation* dan *local search*. Tahapan *perturbation* menghasilkan solusi baru yang memungkinkan untuk menghasilkan solusi yang lebih buruk. Sebaliknya, tahapan *local search* bertujuan untuk menghasilkan solusi yang lebih baik atau setidaknya setara dengan solusi sebelumnya. Suatu solusi dianggap lebih baik jika dapat menjadwalkan lebih banyak kegiatan tanpa melanggar satupun *hard constraint*. Sementara solusi baru dikategorikan lebih buruk jika jumlah kegiatan yang berhasil dijadwalkan lebih



Gambar 4.1.6: Ilustrasi Solusi yang Menjauh dari Nilai Solusi Terbaik

sedikit dari solusi sebelumnya. Dengan desain ini, algoritma selalu menerima solusi baru yang dihasilkan oleh tahapan *perturbation* dan *local search*.

Algoritma PA-ILS dikembangkan dengan selalu menerima solusi yang dihasilkan oleh tahapan *perturbation* dan *local search*. Untuk memastikan proses pencarian solusi tidak terlalu eksploratif, satu variabel dikembangkan yang bernama `probabilityNonRandomTimeSlot` yang dihitung melalui fungsi `calculateProbability`. Variabel ini bertujuan untuk membatasi eksplorasi pada tahapan *perturbation*. Variabel ini akan mengatur pemilihan slot waktu yang dapat digunakan berdasarkan jumlah konflik yang dihasilkan.

Penjelasan detail mengenai aspek-aspek utama pada algoritma PA-ILS dijabarkan pada subbab-subbab berikutnya. Subbab 4.2.1 membahas perhitungan nilai variabel `probabilityNonRandomTimeSlot`. Subbab 4.2.2 membahas proses penjadwalan kegiatan dan tahapan untuk menghapus konflik yang mungkin terjadi. Selanjutnya, tahapan *perturbation* dan *local search* dijabarkan pada Subbab 4.2.3 dan 4.2.4. Terakhir, pembahasan parameter yang terdapat pada algoritma PA-ILS dijabarkan pada Subbab 4.2.5

4.2.1 Perhitungan Variabel `probabilityNonRandomTimeSlot`

Variabel `probabilityNonRandomTimeSlot` merupakan variabel yang mengatur nilai probabilitas, sehingga nilai variabel ini berada diantara 0 hingga 1. Nilai

Algoritma 2 PA-ILS

```
1: procedure SEARCHFEASIBLESOLUTION( decreasingValue, constantFactor,
   limitUpdateProbabilityNonRandomTimeSlot, limitStuck, limitShuffle )
2:     calculateProbability(decreasingValue, constantFactor)
3:     iteration ← 0
4:     while not allEventScheduled() do
5:         perturbationPhase(probabilityNonRandomTimeSlot)
6:         localSearchPhase(limitStuck, limitShuffle)
7:         if iteration MOD limitUpdateProbabilityNonRandomTimeSlot = 0 then
8:             calculateProbability(decreasingValue, constantFactor)
9:         end if
10:        iteration ← iteration +1
11:    end while
12: end procedure
```

dari variabel dihitung dengan mengurangi nilai 1 dengan `decreasingValue`, seperti pada Persamaan 4.1. Nilai `decreasingValue` diperoleh dengan mengalikan nilai `decreasingValue` saat ini dengan `constantFactor`, sebagaimana diperlihatkan pada Persamaan 4.2. Ketiga variabel—`probabilityNonRandomTimeSlot`, `decreasingValue`, dan `constantFactor`—dibatasi dalam rentang 0 hingga 1.

Kedua persamaan ini dieksekusi ketika metode `calculateProbability` dipanggil. Pertama, Persamaan 4.2 dijalankan untuk menghitung nilai terbaru dari `decreasingValue`. Selanjutnya, Persamaan 4.1 digunakan untuk menghitung `probabilityNonRandomTimeSlot`. Pendekatan ini dirancang agar nilai `probabilityNonRandomTimeSlot` secara bertahap meningkat, sehingga meningkatkan kemungkinan pemilihan slot waktu dengan konflik minimal. Jika nilai `probabilityNonRandomTimeSlot` melebihi 0,99, prosedur pengulangan diaktifkan untuk mengatur ulang variabel `decreasingValue` ke nilai awal.

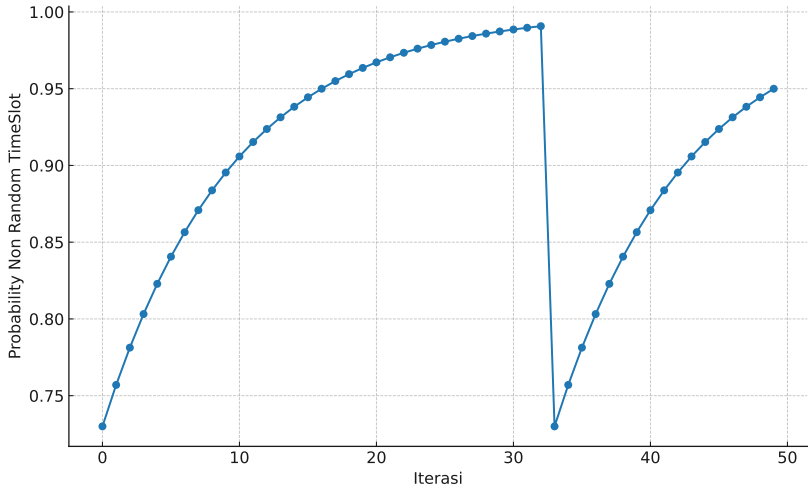
$$probabilityNonRandomTimeSlot = 1 - decreasingValue \tag{4.1}$$

$$decreasingValue = decreasingValue \cdot constantFactor \tag{4.2}$$

Untuk memahami cara kerja dari kedua persamaan, Gambar 4.2.1 mengilustrasikan perubahan nilai variabel `probabilityNonRandomTimeSlot` seiring dengan pertambahan iterasi. Dalam contoh ini, nilai awal `decreasingValue` ditetapkan sebesar 0,3 dan `constantFactor` sebesar 0,9, dengan asumsi bahwa fungsi `calculateProbability` dipanggil pada setiap iterasi.

Pada iterasi ke-0, persamaan 4.2 dijalankan dengan mengalikan `decreasingValue` (0,3) dan `constantFactor` (0,9), sehingga dihasilkan nilai 0,27. Nilai ini kemudian digunakan sebagai `decreasingValue` untuk persamaan 4.1, sekaligus menjadi nilai awal pada iterasi berikutnya. Setelah itu, persamaan 4.1 dieksekusi dengan mengurangi 1 dengan 0,27, menghasilkan nilai `probabilityNonRandomTimeSlot` sebesar 0,73. Proses perhitungan ini berulang pada setiap iterasi dan menyebabkan nilai `probabilityNonRandomTimeSlot` terus meningkat.

Ketika hasil persamaan pertama melebihi 0,99—sebagaimana ditunjukkan pada iterasi ke-32—kedua persamaan dihitung ulang dengan menggunakan nilai awal `decreasingValue` sebesar 0,3. Akibatnya, `probabilityNonRandomTimeSlot` akan memiliki nilai 0,73 sama dengan nilai pada iterasi ke-0.



Gambar 4.2.1: Grafik Perubahan Nilai Variabel `probabilityNonRandomTimeSlot`

4.2.2 Proses Menjadwalkan Kegiatan

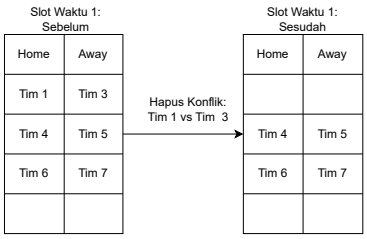
Pada tahapan *local search* dan *perturbation*, proses penjadwalan kegiatan dilakukan dengan terlebih dahulu menghapus kegiatan-kegiatan yang berkonflik dengan kegiatan yang akan dijadwalkan akibat adanya pelanggaran terhadap *hard constraint*. Setelah tidak ada konflik yang tersisa, kegiatan tersebut kemudian dijadwalkan. Untuk menggambarkan proses ini, Gambar 4.2.2 menunjukkan contoh penjadwalan kegiatan (dalam ilustrasi ini berupa penjadwalan kompetisi olahraga) dengan menampilkan beberapa tahapan yang terlibat.

Dalam contoh ilustrasi ini, pertandingan antara Tim 1 dan Tim 2 akan dijadwalkan pada slot waktu ke-1. Namun, pada slot waktu ke-1, telah dijadwalkan pertandingan antara Tim 1 dan Tim 3. Kondisi ini menimbulkan pelanggaran terhadap *hard constraint*, yang mensyaratkan bahwa satu tim hanya dapat bermain satu kali dalam satu slot waktu. Oleh karena itu, pertandingan antara Tim 1 dan Tim 3 harus dihapus dari slot waktu tersebut. Gambar 4.2.2a mengilustrasikan kondisi sebelum dan sesudah penyelesaian konflik ini.

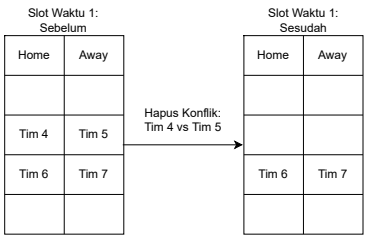
Selain itu, penjadwalan pertandingan antara Tim 1 dan Tim 2 juga melanggar batasan *hard constraint* lain. Dalam contoh ini, *hard constraint* membatasi hanya satu tim yang dapat bermain tandang pada slot waktu ke-1, dengan pilihan antara Tim 2 atau Tim 5. Akibatnya, untuk mengatasi konflik ini, diperlukan untuk menghapus pertandingan antara Tim 4 dan Tim 5 dari slot waktu ke-1. Gambar 4.2.2b mengilustrasikan kondisi sebelum dan sesudah penyelesaian konflik kedua ini. Setelah semua konflik terselesaikan, pertandingan antara Tim 1 dan Tim 2 berhasil dijadwalkan pada slot waktu ke-1, seperti yang ditunjukkan pada Gambar 4.2.2c.

4.2.3 Tahapan *Perturbation*

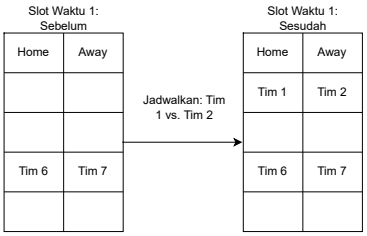
Tahapan *perturbation* didesain seperti yang ditampilakan pada Algoritma 3. Tahapan *perturbation* dimulai dengan memilih secara acak kegiatan yang belum dijadwalkan. Selanjutnya, dilakukan perbandingan antara nilai acak yang dihasilkan (dalam rentang 0 hingga 1) dengan nilai variabel `probabilityNonRandomTimeSlot`. Jika nilai acak tersebut lebih kecil dari `probabilityNonRandomTimeSlot`, strategi yang digunakan adalah menjadwalkan kegiatan pada slot waktu dengan konflik paling sedikit. Jika terdapat beberapa slot waktu dengan jumlah konflik minimum yang sama, slot waktu



(a) Hapus Konflik Pertama



(b) Hapus Konflik Kedua



(c) Menjadwalkan Permainan antara Tim 1 vs Tim 2

Gambar 4.2.2: Ilustrasi dari Proses Penjadwalan Permainan antara Tim 1 vs Tim 2

tersebut akan dipilih secara acak. Sebaliknya, jika nilai acak lebih besar atau sama dengan `probabilityNonRandomTimeSlot`, maka slot waktu akan dipilih secara acak tanpa mempertimbangkan jumlah konflik.

Algoritma 3 Perturbation Phase

```
1:  procedure PERTURBATIONPHASE(probabilityNonRandomTimeSlot)
2:      x ← getRandomUnscheduledEvent()
3:      u ← getRandomFloat()                          ▷ returns a value in [0, 1)
4:      if u < probabilityNonRandomTimeSlot then
5:          y ← getBestTimeSlot(x)
6:      else
7:          y ← getRandomTimeSlot(x)
8:      end if
9:      x ← scheduleEvent(x, y)
10: end procedure
```

4.2.4 Tahapan *Local Search*

Tahapan *local search* ditunjukan pada Algoritma 4. Tahapan ini terdiri dari proses iteratif yang mencakup dua langkah utama. Langkah pertama bertujuan untuk menjadwalkan semua kegiatan yang belum terjadwal. Sebuah kegiatan memenuhi syarat untuk dijadwalkan pada tahapan ini jika tersedia slot waktu yang tidak memiliki konflik

dengan kegiatan lain atau slot waktu dengan konflik maksimal dengan satu kegiatan yang telah terjadwal. Dengan demikian, jumlah kegiatan yang dijadwalkan akan tetap atau meningkat. Langkah kedua adalah menjalankan proses pengacakan dengan menerapkan LLH pada solusi saat ini. Namun penerapan LLH harus tetap menjaga jumlah kegiatan yang dijadwalkan agar tidak berubah. Tujuan utama proses pengacakan ini adalah untuk meningkatkan peluang keberhasilan dalam menjadwalkan kegiatan yang belum terjadwal pada iterasi berikutnya.

Tahapan *local search* akan berakhir ketika salah satu dari kondisi terpenuhi. Kondisi pertama adalah ketika variabel `stuck` melebihi ambang batas yang ditentukan, yaitu sebesar nilai pada variabel `limitStuck`. Kondisi kedua terjadi ketika tidak ada peningkatan dalam jumlah kegiatan yang berhasil dijadwalkan dan proses pengacakan gagal menghasilkan solusi baru.

Algoritma 4 Local Search Phase

```
1:  procedure LOCALSEARCHPHASE(limitStuck, limitShuffle)
2:      currentBest ← numberUnscheduledEvent()
3:      stuck ← 0
4:      succeedShuffle ← true
5:      improvement ← false
6:      while (stuck ≤ limitStuck) AND (succeedShuffle OR improvement) do
7:          improvement ← false
8:          shuffleUnscheduledEvent(x, y)
9:          for each unscheduledEvent in getUnscheduledEvents() do
10:             timeSlot ← getBestTimeSlot(unscheduledEvent)
11:             if calculateConflict(unscheduledEvent, timeSlot) < 2 then
12:                 scheduleEvent(unscheduledEvent, timeSlot)
13:                 improvement ← true
14:             end if
15:         end for
16:         if currentBest < numberUnscheduledEvent() then
17:             stuck ← 0
18:             currentBest ← numberUnscheduledEvent()
19:         else
20:             stuck ← stuck + 1
21:         end if
22:         succeedShuffle ← shuffle(limitShuffle)
23:     end while
24: end procedure
```

Dalam proses pengacakan, tahapan yang dilakukan mengikuti langkah-langkah pada Algoritma 5. Proses pengacakan dimulai dengan memilih suatu kegiatan secara acak,

kemudian menerapkan LLH yang juga dipilih secara acak. Jika proses ini berhasil, maka prosedur pengacakan dianggap selesai. Namun, jika belum berhasil, proses akan diulang hingga jumlah iterasi mencapai batas yang ditentukan oleh variabel `limitShuffle`.

Algoritma 5 Shuffle Procedure

```
1:  procedure SHUFFLE(limitShuffle)
2:      succeedShuffle ← false
3:      for i ← 1 to limitShuffle do
4:          x ← getRandomUnscheduledEvent()
5:          succeedShuffle ← doLLH(selectRandomLLH(),x)
6:          if succeedShuffle then
7:              return true
8:          end if
9:      end for
10:     return false
11: end procedure
```

4.2.5 Parameter

Berdasarkan desain algoritma yang telah dijelaskan, terdapat lima parameter yang berperan pada algoritma PA-ILS, seperti yang ditampilkan pada Tabel 4.2.1. Parameter `decreasingValue` dan `constantFactor` mempengaruhi intensitas tahapan eksploitasi dalam proses pencarian solusi. Kedua parameter ini memiliki rentang nilai antara lebih dari 0 hingga kurang dari 1.

Sementara itu, tiga parameter lainnya, berfungsi sebagai batasan untuk berbagai prosedur dalam algoritma. Parameter `limitStuck` dan `limitShuffle` berperan dalam membatasi prosedur pada tahapan *local search*. Parameter `limitUpdateProbabilityNonRandomTimeSlot` menentukan kapan probabilitas akan diperbarui melalui fungsi `calculateProbability`. Parameter `limitStuck` digunakan untuk menentukan kapan tahapan *local search* akan berakhir, sementara `limitShuffle` membatasi jumlah iterasi dalam proses pengacakan pada solusi.

Tabel 4.2.1: Daftar Parameter

Parameter	Rentang Nilai
decreasingValue	$(0, 1)$
constantFactor	$(0, 1)$
limitUpdateProbabilityNonRandomTimeSlot	> 0 (bilangan bulat positif)

(Lanjutan) Daftar Parameter

Parameter	Rentang Nilai
limitStuck	> 0 (bilangan bulat positif)
limitShuffle	> 0 (bilangan bulat positif)

4.3 Desain Algoritma AT-ILS

Pada desain algoritma AT-ILS, terdapat tiga aspek pengembangan utama yang menjadi kunci dari algoritma ini. Pengembangan pertama adalah penggunaan strategi nilai ambang batas pada tiga tahapan algoritma ILS yaitu *perturbation*, *local search*, dan *move acceptance*. Pada tahapan *perturbation*, nilai ambang batas akan menjadi batas seberapa besar solusi buruk yang memungkinkan untuk diterima.

Pada tahapan *local search*, penggunaan nilai ambang batas ditujukan untuk dapat menghasilkan proses pencarian solusi yang memiliki keseimbangan tahapan eksplorasi dan eksploitasi yang menyerupai algoritma Simulated Annealing. Penerapan nilai ambang batas memungkinkan proses pencarian menghasilkan solusi yang lebih buruk selama masih dibawah nilai ambang batas. Namun secara keseluruhan, proses pencarian pada tahapan *local search* akan mampu menurunkan nilai penalti seiring dengan nilai ambang batas yang akan terus menurun.

Sementara pada tahapan *move acceptance*, nilai ambang batas bekerja untuk memberikan peluang menerima solusi yang lebih buruk. Nilai ambang batas menjadi batasan seberapa buruk nilai solusi yang masih bisa diterima dalam tahapan *move acceptance*.

Selanjutnya, pengembangan kedua adalah desain strategi kembali ke solusi terbaik. Strategi ini diterapkan terhadap tahapan *move acceptance*. Ketika solusi baru tidak diterima pada tahapan ini, strategi ini memungkinkan untuk menggunakan solusi terbaik sebagai solusi yang digunakan dalam tahapan iterasi selanjutnya. Hal ini didesain untuk mengatasi permasalahan pencarian solusi yang memungkinkan untuk menjauhi solusi terbaik hingga akhir iterasi.

Pengembangan ketiga adalah algoritma AT-ILS didesain tanpa membutuhkan

pengaturan nilai parameter. Hal ini ditujukan untuk mengatasi kelemahan Algoritma Simulated Annealing yang sensitif terhadap perubahan nilai parameter. Strategi ini dikembangan dengan mengembangkan daftar nilai ambang batas yang berisi selisih nilai solusi saat dilakukan penerapan LLH.

Untuk menjelaskan desain pengembangan algoritma AT-ILS secara detail, Subbab 4.3.1 menjelaskan struktur algoritma ini secara umum. Selanjutnya, Subbab 4.3.2 menjelaskan bagaimana konsep dari daftar nilai ambang batas dan proses melakukan perhitungan nilai ambang batas. Subbab 4.3.3, 4.3.4 dan 4.3.5 menjelaskan ketiga tahapan utama dalam algoritma AT-ILS.

4.3.1 Desain Umum Algoritma AT-ILS

Desain algoritma AT-ILS dikembangkan dengan menggunakan tiga tahapan dalam algoritma ILS sebagai struktur dasar. Secara keseluruhan, Algoritma 6 menunjukan desain dari algoritma AT-ILS. Algoritma didesain menerima dua input. Input pertama berupa solusi awal yang *feasible* (`initialSol`), yang disimpan pada variabel `currentSol`. Input kedua berupa `eventList`, yang berisi daftar kegiatan dalam solusi awal. Selain itu, terdapat dua variabel yang diinsiasikan yaitu `currentSol` dan `bestSol`. Kedua variabel diinisiasikan dengan nilai input `initialSol`.

Tahapan selanjutnya, algoritma ini melakukan iterasi hingga waktu yang telah ditentukan tercapai. Dalam setiap iterasi, tiga tahapan utama dijalankan yaitu tahapan *perturbation*, *local search*, dan *move acceptance*. Selain itu dua fungsi lainnya yaitu untuk menghitung nilai dari variabel `thresholdList` dan `thresholdValue` diperbarui disetiap awal iterasi.

4.3.2 Fungsi dan Perhitungan Nilai Ambang Batas

Pada algoritma AT-ILS, terdapat dua variabel yang berkaitan dengan nilai ambang batas yaitu `thresholdList` dan `thresholdValue`. Variabel `thresholdValue` merupakan variabel yang digunakan sebagai nilai ambang batas pada tiga tahapan dalam algoritma AT-ILS. Nilai dari variabel `thresholdValue` didapatkan dengan menghitung nilai rata-rata dari nilai yang terdapat pada variabel `thresholdList`. Dalam proses penerapannya, daftar nilai pada variabel `thresholdList` akan dihapus secara perlahan

Algoritma 6 AT-ILS

```
1:  procedure OPTIMIZESOLUTION(initialSol,eventList)
2:      currentSol ← initialSol
3:      bestSol ← currentSol
4:      while within duration of running time do
5:          thresholdList ← calculateThreshold(currentSol, eventList)
6:          thresholdValue ← updateThreshold(thresholdList)
7:          newSol ← currentSol
8:          perturbationPhase(newSol, eventList, thresholdValue)
9:          localSearchPhase(newSol, eventList, thresholdList)
10:         moveAcceptancePhase(newSol, currentSol, bestSol)
11:     end while
12: end procedure
```

mulai dari nilai terbesar. Hal ini menyebabkan nilai pada variabel `thresholdValue` perlahan akan berkurang. Strategi ini ditujukan untuk mengarahkan solusi pada konvergensi terutama dalam tahapan *local search*.

Pengembangan variabel `thresholdList` ditujukan untuk mengetahui perkiraan jangkauan solusi yang dapat berubah saat LLH diterapkan terhadap solusi terkini. Setiap permasalahan memungkinkan memiliki jangkauan yang berbeda walaupun diterapkan LLH yang sama. Selain itu, pada dataset yang sama, kondisi solusi saat ini juga mampu mempengaruhi jangkauan yang bisa dihasilkan saat LLH diterapkan. Sebagai contoh, penerapan LLH dapat menghasilkan solusi baru dengan perbedaan nilai penalti yang besar ketika nilai penalti dari solusi saat ini masih tinggi. Namun ketika nilai penalti dari solusi saat ini sudah jauh lebih rendah, maka memungkinkan untuk menghasilkan solusi baru dengan perbedaan penalti yang lebih kecil. Hal ini mungkin berlaku sebaliknya, bergantung dari jenis dan studi kasus yang akan diselesaikan. Oleh karena itu, menggunakan variabel `thresholdList` untuk menghitung nilai ambang batas dapat menghasilkan nilai ambang batas yang lebih adaptif terhadap permasalahan dan studi kasus yang akan diselesaikan.

Secara detail, proses penghitungan nilai variabel `thresholdList` dengan menjalankan fungsi `calculateThreshold` ditunjukan pada algoritma 7. Fungsi `calculateThreshold` membutuhkan dua input berupa solusi saat ini (`currentSol`) dan daftar kegiatan yang dijadwalkan (`eventList`). Selanjutnya terdapat dua variabel yang perlu dideklarasi dan diinsialisasikan. Pertama, variabel `currentPenalty` dideklarasikan dan diinisiasikan dengan menghitung nilai penalti saat ini. Kedua, variabel

`thresholdList` diinisiasikan menjadi variabel dengan daftar nilai yang kosong.

Selanjutnya, proses perulangan dijalankan dengan menerapkan LLH pada seluruh daftar kegiatan. Jika hasil dari penerapan LLH ini *feasible* dan memiliki nilai penalti yang berbeda dari solusi saat ini, maka perbedaan tersebut dicatat dalam variabel `thresholdList`. Perubahan yang dilakukan selalu dikembalikan ke kondisi awal pada setiap iterasi agar tidak mempengaruhi proses optimasi. Jika `thresholdList` tetap kosong setelah perulangan, algoritma akan mengulang proses ini hingga `thresholdList` memiliki nilai. Setelah perulangan selesai, `thresholdList` diurutkan dan hasil akhirnya dikembalikan kepada pemanggil fungsi.

Algoritma 7 Calculate Threshold

```
1:  procedure CALCULATETHRESHOLD(currentSolution, eventList)
2:      currentPenalty ← calculatePenalty(currentSolution)
3:      thresholdList ← an empty list
4:      while thresholdList is empty do
5:          for each event in eventList do
6:              chosenLLH ← selectRandomLLH()
7:              doLLH(chosenLLH, event)
8:              if checkFeasibility(currentSolution) is true then
9:                  newPenalty ← calculatePenalty(currentSolution)
10:                 if newPenalty ≠ currentPenalty then
11:                     difference ← |currentPenalty − newPenalty|
12:                     append(thresholdList, difference)
13:                 end if
14:             end if
15:             undoChanges(event)
16:         end for
17:     end while
18:     sort(thresholdList)
19:     return thresholdList
20: end procedure
```

4.3.3 Tahapan *Perturbation*

Tahapan *perturbation* merupakan tahapan yang bertujuan untuk mengeksplorasi solusi. Pada tahapan ini, proses eksplorasi didesain seperti yang ditampilkan pada Algoritma 8. Tahapan ini dimulai dengan menerima input berupa solusi baru saat ini (`newSol`), daftar kegiatan (`eventList`) dan nilai ambang batas (`thresholdValue`). Langkah pertama adalah mendeklarasikan variabel `perturbation` dengan nilai awal `true`, yang menandakan bahwa tahapan *perturbation* sedang berjalan.

Selanjutnya, variabel `currentPenalty` dideklarasikan dan diinisialisasi dengan nilai penalti dari solusi pada variabel `newSol`. Algoritma kemudian memasuki perulangan di mana, pada setiap iterasi, LLH diterapkan pada kegiatan yang dipilih secara acak dari kegiatan dalam variabel `eventList`. Solusi baru diterima jika solusi tersebut *feasible* tanpa mempertimbangkan nilai penaltinya. Jika solusi tidak *feasible*, maka solusi dikembalikan ke kondisi awal menggunakan fungsi `undoChanges`.

Tahapan *perturbation* akan berakhir ketika nilai penalti dari solusi saat ini (`newPenalty`) melebihi nilai `currentPenalty` ditambah `thresholdValue`. Variabel `perturbation` diubah menjadi `false` yang menyatakan proses *perturbation* berakhir. Penggunaan variabel `thresholdValue` ditujukan untuk memastikan dampak dari proses eksplorasi solusi tidak terlalu kecil. Hal ini diharapkan mampu menghasilkan solusi baru yang lebih baik saat diterapkan tahapan *local search*.

Algoritma 8 Perturbation Phase

```
1:  procedure PERTURBATIONPHASE(newSol, evenList, thresholdValue)
2:      perturbation ← true
3:      currentPenalty ← calculatePenalty()
4:      while perturbation do
5:          chosenLLH ← selectRandomLLH()
6:          doLLH(chosenLLH, event.get(random))
7:          if checkFeasibility() then
8:              newPenalty ← calculatePenalty()
9:              if newPenalty > currentPenalty + thresholdValue then
10:                 perturbation ← false
11:             end if
12:         else
13:             undoChanges()
14:         end if
15:     end while
16: end procedure
```

4.3.4 Tahapan *Local Search*

Tahapan *local search* merupakan tahapan yang berfokus untuk mengeksploitasi solusi untuk dapat menghasilkan solusi terbaik baru. Tahapan ini didesain seperti yang ditunjukan pada Algoritma 9. Tahapan ini dimulai dengan menerima input berupa solusi baru dari tahapan *perturbation* (`newSol`), daftar kegiatan (`eventList`), nilai ambang batas (`thresholdValue`), serta daftar nilai ambang batas (`thresholdList`). Selanjutnya, algoritma melakukan deklarasi dan inisialisasi beberapa variabel sebagai

berikut:

- `localSearch`: variabel ini berfungsi untuk menandai kapan tahapan *local search* akan berakhir. Variabel ini diinisialisasi dengan nilai awal `true`.
- `currentPenalty`: variabel untuk mencatat nilai penalti solusi saat ini. Variabel ini diinisialisasi dengan menghitung nilai penalti dari `newSol`.
- `localBest`: variabel ini berfungsi untuk mencatat nilai penalti terbaik yang ditemukan selama tahapan *local search*. Variabel ini diinisialisasi menggunakan nilai dari variabel `currentPenalty`.
- `thresholdLocal`: variabel ini digunakan untuk membatasi penerimaan solusi dalam tahapan *local search*. Variabel ini diinisialisasi dengan nilai dari `currentPenalty`.
- `improve`: variabel ini menandakan apakah terdapat peningkatan solusi selama penerapan fungsi `ApplyHeuristicToAllEvents`. Ketika variabel ini bernilai `true` maka menandakan terjadinya peningkatan solusi. Sebaliknya, ketika variabel ini bernilai `false`, maka manandakan tidak adanya peningkatan solusi. Variabel ini diinisialisasi dengan nilai awal `false`.
- `improveLocalBest`: variabel ini menandakan apakah terjadi peningkatan pada variabel `localBest` selama penerapan fungsi `ApplyHeuristicToAllEvents`. Ketika variabel ini bernilai `true` maka menandakan terjadinya peningkatan solusi pada variabel `localBest`. Sebaliknya, ketika variabel ini bernilai `false` manandakan tidak adanya peningkatan solusi pada variabel `localBest`. Variabel ini diinisialisasi dengan nilai awal `false`.
- `amountRemoved`: variabel ini menentukan jumlah elemen yang akan dihapus dari `eventList` dalam fungsi `UpdateThresholdList`. Variabel ini diinisialisasi dengan nilai 1. Nilai pada variabel ini akan bertambah seiring berjalannya tahapan *local search*. Hal ini bertujuan untuk meningkatkan secara lebih masif penghapusan pada variabel `eventList` yang berdampak pada semakin kecilnya nilai ambang batas pada variabel `thresholdValue`.

Setelah deklarasi dan inisialisasi variabel, tahapan *local search* dimulai melalui proses iteratif. Dalam proses ini, Terdapat tiga langkah utama. Langkah pertama adalah

pembaruan nilai variabel `thresholdLocal` melalui pemanggilan fungsi `updatethresholdLocal`. Detail mengenai proses pembaruan ini dijelaskan pada Subbab 4.3.4.1. Langkah kedua adalah menerapkan perubahan pada solusi dengan menjalankan fungsi `ApplyHeuristicToAllEvents`. Dalam fungsi ini, perubahan yang diterima tidak boleh melebihi batas nilai yang ditentukan oleh variabel `thresholdLocal`. Penjelasan lebih lanjut tentang fungsi ini dijelaskan pada Subbab 4.3.4.2.

Langkah ketiga, algoritma memperbarui nilai variabel `thresholdList` dan `thresholdValue` dengan menjalankan fungsi `UpdateThresholdList`. Fungsi ini juga memeriksa apakah tahapan *local search* akan berakhir atau dilanjutkan. Penjelasan lengkap mengenai fungsi ini dijabarkan pada Subbab 4.3.4.1. Setelah iterasi berakhir, solusi terkini akan dikembalikan menjadi solusi terbaik yang ditemukan dalam tahapan *local search* dengan menjalankan fungsi `useLocalBest`.

4.3.4.1 Fungsi *UpdateThresholdLocal*

Dalam tahapan *local search*, variabel `thresholdLocal` merupakan variabel yang menjadi nilai ambang batas dalam proses penerimaan solusi baru. Variabel `thresholdLocal` dihitung dengan menjalankan fungsi `UpdateThresholdLocal` yang melibatkan variabel `thresholdList` dan `thresholdValue`. Fungsi ini didesain seperti yang ditampilkan pada Algoritma 10.

Proses dimulai dengan memeriksa apakah variabel `thresholdList` kosong atau tidak. Jika variabel `thresholdList` kosong, maka variabel `thresholdLocal` akan diatur ke nilai penalti saat ini. Jika variabel `thresholdList` tidak kosong, maka langkah berikutnya adalah mengecek apakah hasil pengurangan variabel `thresholdLocal` dengan nilai pada indeks pertama dari variabel `thresholdList` tetap lebih besar dari nilai penalti saat ini. Jika benar, maka variabel `thresholdLocal` diperbarui dengan nilai pengurangan tersebut. Namun, jika hasilnya sama atau lebih kecil dari nilai penalti saat ini, maka variabel `thresholdLocal` akan diatur sama dengan nilai penalti saat ini. Penggunaan nilai indeks pertama dari variabel `thresholdList` bertujuan untuk memberikan ruang pada solusi untuk melakukan eksplorasi.

Selain itu, terdapat pengecekan tambahan untuk memastikan solusi tidak terjebak

Algoritma 9 Local Search Phase

```
1:  procedure LOCALSEARCHPHASE(newSol, eventList, thresholdValue, thresholdList)
2:      localSearch ← true
3:      currentPenalty ← calculatePenalty(newSol)
4:      localBest ← currentPenalty
5:      thresholdLocal ← currentPenalty
6:      improve ← false
7:      improveLocalBest ← false
8:      amountRemoved ← 1
9:      while localSearch do
10:         thresholdLocal ← UpdateThresholdLocal(thresholdLocal,
11:                 thresholdList, currentPenalty, thresholdValue, improve)
12:         improveLocalBest ← ApplyHeuristicToAllEvents(
13:                 eventList, thresholdLocal,
14:                 currentPenalty, localBest)
15:         if NOT improve then
16:             localSearch ← UpdateThresholdList(thresholdList,
17:                     amountRemoved, improveLocalBest,
18:                     localBest, currentPenalty)
19:         end if
20:     end while
21:     useLocalBest()
22: end procedure
```

dalam kondisi *local optima*. Jika solusi tidak mengalami peningkatan dari iterasi sebelumnya, maka nilai dari variabel `thresholdLocal` akan ditambah dengan nilai dari variabel `thresholdValue`. Jika terjadi peningkatan, maka variabel `improve` diatur menjadi `false` untuk melakukan pengecekan peningkatan pada iterasi berikutnya.

4.3.4.2 Fungsi *ApplyHeuristicToAllEvents*

Bagian ini bertujuan untuk meningkatkan solusi dalam setiap iterasinya. Proses dalam fungsi ini ditunjukkan pada Algoritma 11. Tahapan dimulai dengan mengacak urutan dalam variabel `eventList` dan menginisialisasi variabel `improveLocalBest` dengan nilai `false`. Langkah selanjutnya menerapkan LLH yang dipilih secara acak pada setiap kegiatan dalam `eventList` untuk menghasilkan solusi baru.

Setiap solusi baru yang dihasilkan akan dilakukan pengecekan untuk memutuskan apakah solusi baru akan digunakan atau tidak. Jika solusi yang dihasilkan *feasible*, maka dilanjutkan dengan pengecekan nilai penalti. Jika solusi tidak *feasible*, solusi akan dikembalikan ke kondisi sebelumnya melalui fungsi `undoChanges`. Pada pengecekan

Algoritma 10 Update Threshold Local

```
1: procedure UPDATETHRESHOLDLOCAL(thresholdLocal, thresholdList,
   currentPenalty, thresholdValue, improve)
2:     if thresholdList is not empty then
3:         firstElement ← the first (or smallest) element of thresholdList
4:         if (thresholdLocal − firstElement) > currentPenalty then
5:             thresholdLocal ← thresholdLocal − firstElement
6:         else
7:             thresholdLocal ← currentPenalty
8:         end if
9:     else
10:        thresholdLocal ← currentPenalty
11:    end if
12:    if not improve then
13:        thresholdLocal ← thresholdLocal + thresholdValue
14:    else
15:        improve ← false
16:    end if
17:    return thresholdLocal
18: end procedure
```

nilai penalti, terdapat dua persyaratan yang menggunakan logika `OR`, yang berarti solusi akan diterima jika salah satu persyaratan terpenuhi. Persyaratan pertama adalah solusi baru memiliki nilai penalti yang lebih kecil dari nilai pada variabel `thresholdLocal`. Persyaratan kedua adalah nilai penalti solusi baru sama dengan nilai pada variabel `thresholdLocal` dan nilai penalti pada variabel `currentPenalty` juga sama dengan nilai variabel `thresholdLocal`. Jika solusi diterima dan memiliki nilai lebih kecil dari variabel `thresholdLocal`, maka variabel `improve` akan diubah menjadi `true` untuk menunjukan adanya peningkatan solusi. Selain itu, variabel `improveLocalBest` akan diperbarui jika ditemukan solusi terbaik lokal yang baru.

Penggunaan dua persyaratan untuk penerimaan solusi dari sisi nilai penalti ini bertujuan untuk mencegah solusi yang sudah meningkat kembali ke nilai penalti yang sama dengan nilai `thresholdLocal`. Dalam fungsi ini, nilai ambang batas tidak mengalami perubahan nilai. Jika syarat penerimaan hanya diterapkan pada solusi baru yang lebih kecil atau sama dengan ambang batas, terdapat kemungkinan solusi kembali ke nilai yang sama dengan nilai ambang batas. Sebagai contoh, LLH *swap* diterapkan pada kegiatan 1 dengan kegiatan 3 dan mampu menghasilkan nilai penalti yang lebih baik, sehingga solusi ini diterima. Namun saat LLH *swap* diterapkan pada kegiatan 3 dan 1

dalam beberapa iterasi berikutnya, terdapat kemungkinan akan menghasilkan nilai penalti yang sama sebelum LLH *swap* diterapkan pada kegiatan 1 dengan kegiatan 3. LLH *swap* yang awalnya menukar kan kegiatan 1 dengan kegiatan 3 pada iterasi berikutnya kembali menukarkan kegiatan 3 dan 1 yang menghasilkan tidak adanya perubahan solusi. Untuk menghindari kondisi tersebut, dua syarat penerimaan yang telah dijelaskan sebelumnya digunakan.

Algoritma 11 Apply Heuristic to Events

```
1: procedure APPLYHEURISTICTOEVENTS(eventList, thresholdLocal, currentPenalty,
   localBest)
2:     shuffle(eventList)
3:     improveLocalBest ← false
4:     for each event in eventList do
5:         chosenLLH ← selectRandomLLH()
6:         doLLH(chosenLLH, event)
7:         if isFeasible() then
8:             newPenalty ← calculatePenalty()
9:             if (newPenalty < thresholdLocal) OR ((newPenalty = thresholdLocal)
   AND (currentPenalty = thresholdLocal)) then
10:                currentPenalty ← newPenalty
11:                if currentPenalty < thresholdLocal then
12:                    improve ← true
13:                end if
14:                if currentPenalty < localBest then
15:                    localBest ← currentPenalty
16:                    saveBestLocalSol()
17:                    improveLocalBest ← true
18:                end if
19:            else
20:                undoChanges(event)
21:            end if
22:        else
23:            undoChanges(event)
24:        end if
25:    end for
26:    return improveLocalBest
27: end procedure
```

4.3.4.3 Fungsi *UpdateThresholdList*

Bagian ini merupakan fungsi untuk memperbarui nilai dari `thresholdList`. Perbaruan dilakukan dengan mengurangi jumlah elemen pada `thresholdList` untuk menurunkan nilai `thresholdValue`. Pengurangan ini bertujuan untuk mengurangi

eksplorasi dan memfokuskan proses pada tahapan eksploitasi sebelum solusi akhir diserahkan pada tahapan *move acceptance*. Algoritma 12 menunjukkan desain dari proses ini. Pada tahapan pertama, proses pengecekan terhadap jumlah elemen dalam `thresholdList` dilakukan. Jika `thresholdList` tidak memiliki satupun elemen, maka tidak ada elemen yang dapat dihapus. Hal ini menandakan akhir dari tahapan *local search*. Jika hal ini terjadi, fungsi `UpdateThresholdList` akan mengembalikan nilai `false`. Jika `thresholdList` masih memiliki elemen, maka proses penghapusan diterapkan.

Proses penghapusan dimulai dengan mengecek apakah terdapat peningkatan pada `localBest` pada tahapan `ApplyHeuristicToAllEvents` yang ditandai dalam variabel `improveLocalBest`. Jika ada, maka jumlah penghapusan akan diubah menjadi 1 dan variabel `improveLocalBest` diatur menjadi `false` untuk digunakan pada iterasi berikutnya. Jika tidak ada peningkatan, maka jumlah penghapusan akan ditingkatkan dengan menambah nilai 1 pada variabel `amountRemoved`. Tahap selanjutnya adalah menghapus elemen dari `thresholdList` sebanyak nilai `amountRemoved`. Penghapusan dimulai dari nilai terbesar yang terdapat pada variabel `thresholdList`.

Setelah penghapusan selesai, solusi mungkin akan dikembalikan ke kondisi `localBest` berdasarkan persyaratan berikut. Jika nilai penalti solusi saat ini lebih besar dari `localBest` ditambah salah satu nilai dari kuartil pertama pada `thresholdList`, maka solusi saat ini akan diatur kembali ke solusi terbaik lokal.

Langkah terakhir adalah memperbarui `thresholdValue`. Jika `thresholdList` masih memiliki elemen setelah proses penghapusan, maka nilai `thresholdValue` diperbarui melalui fungsi `updateThreshold`. Jika tidak ada elemen yang tersisa, maka `thresholdValue` akan diatur menjadi 0.

4.3.5 Tahapan *Move Acceptance*

Tahapan *move acceptance* bertujuan untuk memutuskan apakah solusi baru akan digunakan pada iterasi berikutnya. Algoritma 13 menunjukkan desain dari tahapan ini. Pertama, tiga solusi — solusi saat ini, solusi baru dan solusi terbaik — akan dihitung nilai penaltinya. Selanjutnya, jika nilai penalti solusi baru lebih kecil atau sama dengan penalti

Algoritma 12 Update Threshold List

```
1: procedure UPDATETHRESHOLDLIST(thresholdList, amountRemoved, improveLocalBest, localBest, currentPenalty)
2:     if thresholdList is empty then
3:         return false                          ▷ Indicate the end of the local search phase
4:     else
5:         if improveLocalBest then
6:             amountRemoved ← 1
7:             improveLocalBest ← false
8:         else
9:             amountRemoved ← amountRemoved + 1
10:        end if
11:        for k ← 1 to amountRemoved do
12:            if thresholdList is empty then
13:                break                         ▷ No more thresholds to remove
14:            end if
15:            removeLastElement(thresholdList)
16:        end for
17:        if (thresholdList is not empty) AND (currentPenalty > localBest + getRandomQuartil1(thresholdList)) then
18:            useLocalBest()
19:            currentPenalty ← localBest
20:        end if
21:        if thresholdList has more than one element then
22:            thresholdValue ← updateThreshold(thresholdList)
23:        else
24:            thresholdValue ← 0
25:        end if
26:    end if
27:    return true                               ▷ Continue the local search
28: end procedure
```

solusi saat ini, maka solusi baru akan menggantikan solusi saat ini dan solusi tersebut digunakan pada iterasi berikutnya. Selain itu, dilakukan pengecekan apakah nilai penalti solusi baru lebih kecil daripada solusi terbaik. Jika iya, maka solusi baru juga akan menggantikan solusi terbaik.

Sementara itu, jika solusi baru memiliki penalti lebih tinggi daripada solusi saat ini, maka dijalankan proses pengecekan berikutnya. Sebelum melakukan pengecekan lebih lanjut, `thresholdList` yang telah kosong setelah tahapan *local search* akan diisi ulang dengan memanggil fungsi `calculateThreshold`. Selanjutnya, dalam pengecekan kedua, jika nilai penalti solusi baru lebih kecil dari solusi terbaik yang ditambahkan dengan salah satu nilai kuartil pertama dari `thresholdList` yang diambil secara acak, maka solusi baru akan digunakan sebagai solusi saat ini. Sebaliknya, jika syarat ini tidak terpenuhi, solusi terbaik akan digunakan untuk iterasi berikutnya. Penggunaan solusi terbaik dibandingkan solusi saat ini bertujuan agar proses pencarian tidak semakin menjauh dari kondisi terbaik yang sudah ditemukan.

4.4 LLH

Dalam pengembangan suatu algoritma heuristik, LLH berfungsi sebagai operator untuk merubah solusi. Pada desain algoritma ini, tiga jenis LLH yang digunakan adalah *move*, *swap* dan *kempe chain*. Pemilihan ketiga LLH tersebut didasarkan pada tujuan algoritma, yaitu untuk menguji tingkat generalitasnya. Oleh karena itu, LLH yang diterapkan merupakan LLH yang sederhana dan cukup fleksibel untuk diterapkan pada berbagai jenis permasalahan penjadwalan. Penjelasan masing-masing LLH dijabarkan dalam Subbab 4.4.1, 4.4.2, dan 4.4.3.

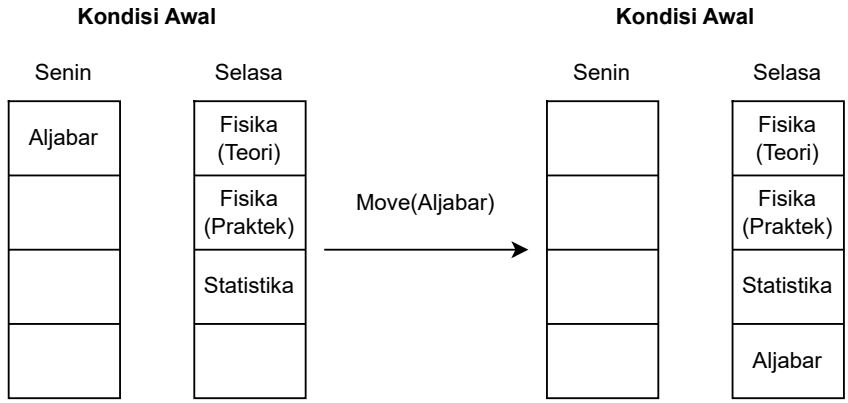
4.4.1 *Move*

LLH *move* adalah operator yang memindahkan jadwal dari satu kegiatan ke slot waktu lain. Pada kasus penjadwalan mata kuliah, operator ini memungkinkan digunakan untuk memindahkan jadwal dari suatu kelas ke slot waktu lain. Gambar 4.4.1 memperlihatkan ilustrasi LLH *move* yang diterapkan pada untuk memindahkan kelas Aljabar. Pada kondisi awal, kelas Aljabar berada pada slot waktu ke-1 di hari Senin. Dengan menggunakan operator *move*, jadwal kelas Aljabar dipindahkan ke slot waktu ke-4 di hari Selasa,

Algoritma 13 Move Acceptance

```
1: procedure MOVEACCEPTANCE(newSol, currentSol, bestSol, eventList)
2:     newPenalty ← calculatePenalty(newSol)
3:     currentPenalty ← calculatePenalty(currentSol)
4:     bestPenalty ← calculatePenalty(bestSol)
5:     if currentPenalty ≥ newPenalty then
6:         currentSol ← newSol
7:         if newPenalty < bestPenalty then
8:             bestSol ← newSol
9:         end if
10:    else
11:        thresholdList ← calculateThreshold(bestSol, eventList)
12:        if newPenalty < bestPenalty + GetRandomFromTopQuartile(thresholdList)
   then
13:            currentSol ← newSol
14:        else
15:            useBestSol()
16:        end if
17:    end if
18: end procedure
```

sehingga slot waktu ke-1 hari Senin menjadi kosong, sementara slot waktu ke-4 di hari Selasa kini terisi oleh kelas Aljabar.

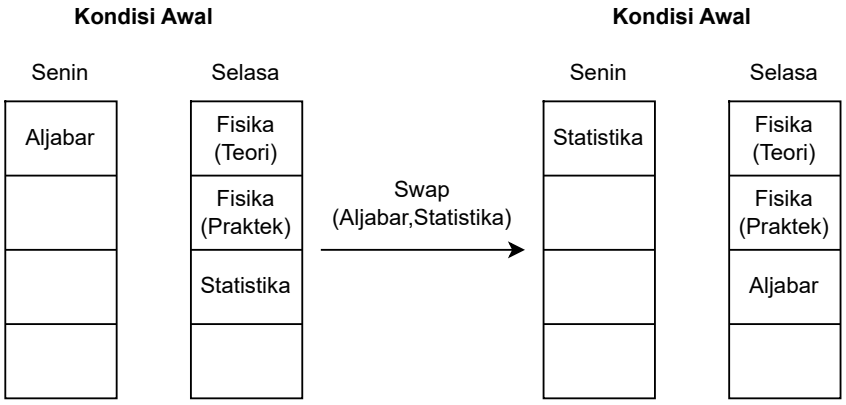


Gambar 4.4.1: Ilustrasi Operator *Move*

4.4.2 *Swap*

LLH *swap* adalah operator yang menukar jadwal antara dua kegiatan. Pada kasus penjadwalan, operator *swap* dapat diterapkan seperti pada ilustrasi dalam Gambar 4.4.2. Pada ilustrasi ini LLH *swap* diterapkan untuk menukar jadwal kelas Aljabar di slot waktu

ke-1 pada hari Senin dengan kelas Statistika yang berada di slot waktu ke-3 pada hari Selasa. Hasil dari penerapan operator ini kelas Aljabar berada pada slot waktu ke-3 di hari Selasa dan kelas Statistika berada pada slot waktu ke-1 di hari Senin.



Gambar 4.4.2: Ilustrasi Operator *Swap*

4.4.3 *Kempe Chain*

LLH *kempe chain* adalah operator yang memperluas konsep *swap* atau *move* dengan cara mengelompokkan kegiatan yang saling terkait dan kemudian memindahkan seluruh kelompok ini untuk memenuhi batasan penjadwalan. Gambar 4.4.3 menunjukan ilustrasi penggunaan operator *kempe chain*. Dalam contoh ini kondisi penjadwalan awal mata kuliah sebagai berikut:

- Kelas Aljabar dijadwalkan pada hari Senin, Slot ke-1.
- Kelas Fisika Teori dijadwalkan pada hari Selasa, Slot ke-1.
- Kelas Praktikum Fisika dijadwalkan pada hari Selasa, Slot ke-2.

Dalam contoh ini terdapat *hard constraint* berupa kelas Fisika Teori dan Praktikum Fisika harus dijadwalkan pada hari yang sama. Ketika kita ingin menukar jadwal kelas Aljabar dengan kelas Fisika Teori, penukaran ini akan menyebabkan kondisi tidak *feasible*, karena kelas Fisika Teori (yang berpindah ke Senin) akan terpisah dari kelas Praktikum Fisika, yang seharusnya berada pada hari yang sama. Untuk menyelesaikan masalah ini, kita menerapkan LLH *kempe chain* dengan langkah berikut:

1. Menukar Kelas Aljabar dan Kelas Fisika Teori. Kelas Aljabar dipindahkan ke hari

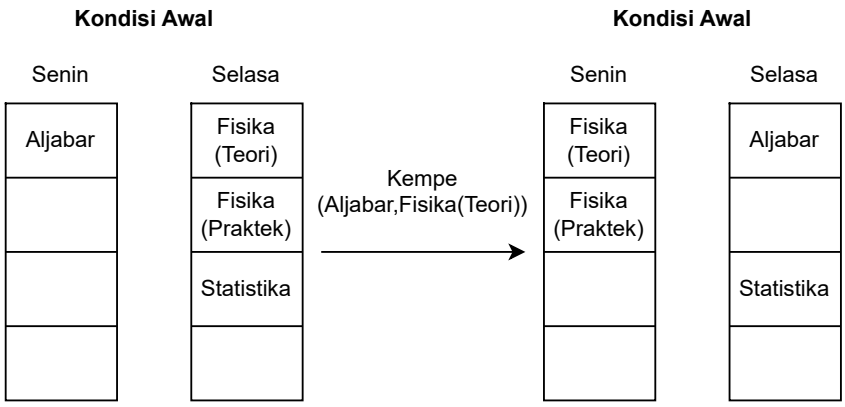
Selasa pada slot ke-1 dan kelas Fisika Teori dipindahkan ke hari Senin pada slot waktu ke-1.

2. Karena kelas Fisika Teori memiliki ketergantungan dengan kelas Praktikum Fisika, maka kelas Praktikum Fisika juga perlu dipindahkan untuk tetap berada di hari yang sama dengan kelas Fisika Teori. Maka kelas Praktikum Fisika dipindahkan ke hari Senin pada slot waktu ke-2.

Hasil akhir setelah penerapan LLH *kempe chain*:

- Kelas Aljabar dipindahkan ke hari Selasa, Slot 1.
- Kelas Fisika Teori dipindahkan ke hari Senin, Slot 1.
- Kelas Praktikum Fisika dipindahkan ke hari Senin, Slot 2.

Dengan demikian, operator kempe chain ini mempertahankan keterkaitan antara Fisika Teori dan Praktikum Fisika, sehingga tetap memenuhi batasan jadwal yang mengharuskan kedua kelas tersebut berada pada hari yang sama. Penggunaan kempe chain efektif untuk mengatasi ketergantungan antar kegiatan, terutama dalam situasi penjadwalan kompleks yang melibatkan beberapa batasan terkait.



Gambar 4.4.3: Ilustrasi Operator Kempe

4.5 Pengaturan Parameter

Berdasarkan penjelasan pada Subbab 4.2.5, algoritma PA-ILS memiliki lima parameter. Bagian ini akan menentukan nilai dari kelima parameter yang akan digunakan. Penentuan

nilai ini dilakukan secara sederhana berdasarkan percobaan pada beberapa nilai dan juga analisis dari strategi yang dikembangkan. Percobaan ini tidak ditujukan untuk menemukan nilai parameter paling optimal, melainkan hanya untuk menentukan nilai parameter yang cukup baik karena pengujian algoritma ditujukan dengan fokus generalitas.

Untuk mewujudkan hal tersebut, uji coba hanya dijalankan pada *benchmark* ITC 2021 dengan memilih tiga dataset secara acak yang terdiri dari Early 3, Middle 10, dan Late 1. Proses pengujian dijalankan pada tiga dataset dengan pengujian sebanyak sepuluh kali untuk setiap dataset pada setiap nilai parameter yang diuji. Metrik tingkat keberhasilan, rata-rata waktu untuk menemukan solusi *feasible* dan konsistensi waktu dalam bentuk Standar Deviasi (SD) digunakan sebagai penentu nilai yang akan digunakan.

Percobaan pertama dilakukan terhadap parameter `decreasingValue`. Parameter ini menentukan pemilihan strategi dalam tahapan *perturbation*. Nilai parameter ini akan menjadi batas bawah dari probabilitas dalam memilih strategi penjadwalan (lihat Subbab 4.2). Semakin besar nilai `decreasingValue` maka semakin besar faktor acak dalam menentukan strategi yang digunakan. Semakin kecil nilai `decreasingValue`, maka semakin besar probabilitas untuk memilih strategi pemilihan slot waktu secara acak. Variabel ini memiliki rentang nilai 0 hingga 1. Pada percobaan ini lima nilai diuji coba yang terdiri dari 0, 0.3, 0.6, 0.9, dan 1. Sementara untuk parameter lain diterapkan nilai yang sama yakni parameter `constantFactor` akan diisi dengan nilai 0.5, `limitUpdateProbabilityNonRandomSlot` dan `limitShuffle` dengan nilai 100, dan `limitStuck` dengan nilai 10.

Tabel 4.5.1 menunjukkan hasil dari percobaan ini. Tabel menampilkan rata-rata dan standar deviasi dari waktu yang dibutuhkan untuk menemukan solusi *feasible*. Sementara pada tingkat keberhasilan menghasilkan solusi *feasible,* seluruh ujicoba pada ketiga dataset berhasil menghasilkan solusi *feasible*, sehingga tidak dicantumkan dalam tabel. Dari hasil ini, nilai `decreaseRate` $= 0,3$ merupakan pilihan paling optimal karena memberikan keseimbangan antara performa dan konsistensi di semua dataset. Meskipun `decreaseRate` $= 0,6$ menunjukkan rata-rata terendah untuk dataset Middle 10, namun variabilitasnya lebih tinggi, terutama pada Late 1. Sebaliknya, `decreaseRate` $= 0,3$ memiliki nilai rata-rata yang kompetitif serta standar deviasi terendah di semua dataset, yang menunjukkan hasil yang konsisten dan stabil. Nilai lainnya, yakni $0,0,9$, dan 1,

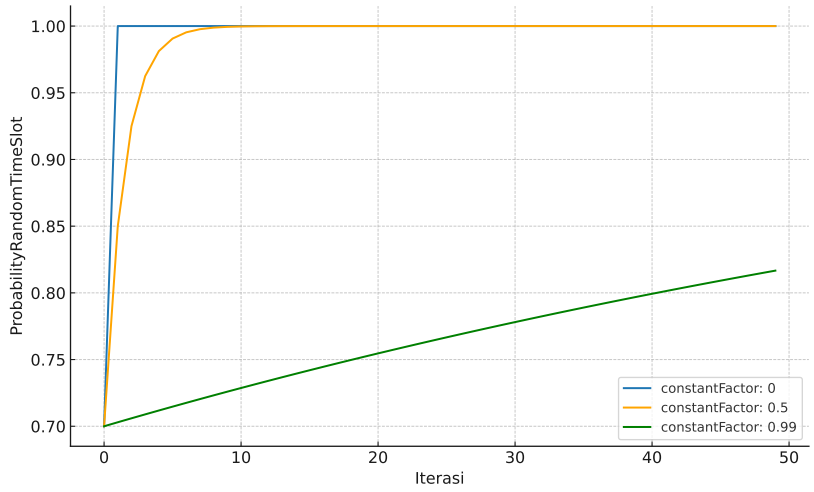
menunjukkan waktu rata-rata dan standar deviasi yang lebih tinggi. Berdasarkan hasil dan analisis tersebut, nilai 0,3 dipilih untuk digunakan pada uji coba selanjutnya.

Tabel 4.5.1: Hasil Uji Coba Nilai Parameter *decreaseRate*

Nilai	Dataset	Rata-rata	SD
0	Early 3	0,11	0,16
	Middle 10	10,10	7,03
	Late 1	7,05	2,81
0,3	Early 3	0,05	0,02
	Middle 10	6,53	3,92
	Late 1	5,69	2,83
0,6	Early 3	0,12	0,08
	Middle 10	4,31	3,32
	Late 1	5,85	4,89
0,9	Early 3	0,04	0,01
	Middle 10	9,90	5,25
	Late 1	10,52	5,67
1	Early 3	0,043	0,02
	Middle 10	7,68	6,95
	Late 1	4,63	2,56

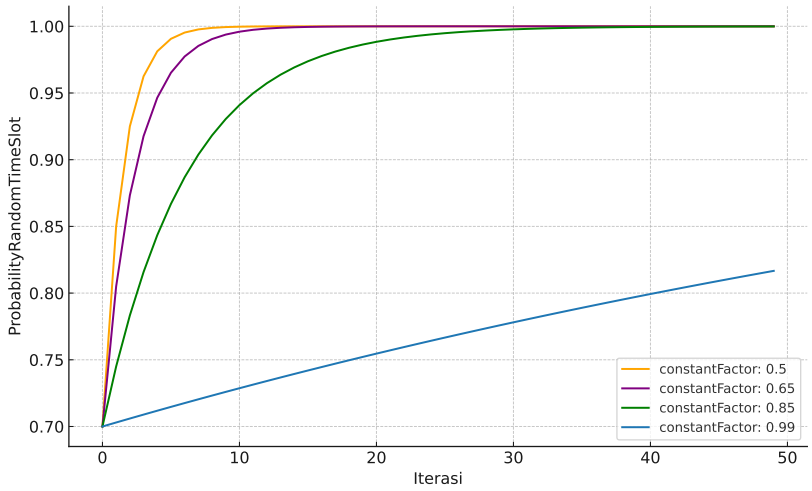
Uji coba berikutnya pada parameter `constantFactor`. Dalam pengembangan algoritma untuk mencari solusi *feasible*, probabilitas pemilihan strategi dalam tahapan *perturbation* tidaklah konstan melainkan bergerak secara perlahan menuju nilai mendekati satu. Parameter `constantFactor` yang menjadi parameter yang menentukan seberapa cepat pergerakan tersebut. Nilai parameter `constantFactor` yang lebih kecil menandakan pergerakan yang semakin cepat dan sebaliknya nilai parameter `constantFactor` yang lebih besar menandakan pergerakan yang semakin lambat. Gambar 4.5.1 menunjukkan bagaimana efek dari nilai parameter *constantFactor* terhadap probabilitas dalam pemilihan slot waktu. Berdasarkan hasil nilai `decreasingValue` 0,3, maka nilai probabilitas akan berkisar antara 0,7 sampai 1 (lihat persamaan 4.2). Penggunaan nilai `constantFactor` 0 akan menyebabkan nilai probabilitas langsung

mendekati nilai 1 pada iterasi berikutnya. Hal ini tentu tidak sesuai dengan strategi yang diharapkan karena strategi yang dikembangkan ingin melakukan perubahan nilai probabilitas secara perlahan. Sementara pada nilai 0,5, perubahan terjadi lebih perlahan dimana nilai probabilitas mendekati 1 baru diperoleh setelah beberapa iterasi. Sementara penggunaan nilai 0,99 menghasilkan pergerakan yang sangat pelan bahkan dalam ilustrasi ini setelah 50 iterasi, masih belum mencapai nilai probabilitas mendekati 1. Berdasarkan ilustrasi ini, maka uji coba pada variabel `constrantFactor` diuji coba dengan nilai 0,5, 0,65, 0,85, dan 0,99.



Gambar 4.5.1: Ilustrasi Perbandingan Nilai pada Parameter constantFactor

Hasil dari uji coba ditampilkan pada Tabel 4.5.2. Pada hasil ini seluruh uji coba menghasilkan solusi *feasible*. Dari segi waktu rata-rata menemukan solusi *feasible* dan konsistensinya, nilai 0,85 menunjukkan hasil yang paling optimal, sehingga nilai ini akan digunakan dalam tahapan berikutnya. Nilai 0,85 menjadi nilai yang paling baik dikarenakan nilai ini memiliki pergerakan yang berada di antara nilai 0,5 dan 0,99. Gambar 4.5.2 menunjukkan perbandingan pergerakan nilai probabilitas pada nilai `constrantFactor` yang diuji. Dari ilustrasi ini, nilai 0,85 menunjukkan pergerakan yang lebih seimbang dibandingkan nilai lainnya dimana nilai ini menghasilkan perubahan nilai probabilitas yang perlahan, namun tidak terlalu lama untuk mencapai nilai mendekati satu seperti pada nilai 0,99.



Gambar 4.5.2: Ilustrasi Perbandingan Nilai pada Parameter *constantFactor*

Tabel 4.5.2: Hasil Uji Coba Nilai Parameter *constantFactor*

Nilai decreasingRate	Dataset	Rata-rata	SD
0,50	Early 3	0,05	0,02
	Middle 10	6,53	3,92
	Late 1	5,69	2,83
0,65	Early 3	0,10	0,14
	Middle 10	7,42	5,60
	Late 1	5,52	2,99
0,85	Early 3	0,04	0,01
	Middle 10	6,58	5,34
	Late 1	5,42	3,91
0,99	Early 3	0,03	0,03
	Middle 10	10,58	7,50
	Late 1	12,31	10,39

Uji coba selanjutnya dilakukan pada tiga parameter lainnya yaitu parameter `limitUpdateProbabilityNonRandomSlot`, `limitShuffle`, dan `limitStuck`. Pada ketiga parameter ini hanya metrik tingkat keberhasilan solusi yang

dapat digunakan. Sementara waktu menemukan solusi tidak dapat digunakan sebagai perbandingan karena semakin besar nilai ketiga parameter tersebut akan menyebabkan waktu menemukan solusi yang lebih lama. Namun nilai yang lebih besar akan memberikan kesempatan proses pencarian solusi yang lebih detail dalam beberapa tahapan. Parameter `limitUpdateProbabilityNonRandomSlot` menentukan berapa batas iterasi solusi tidak mengalami peningkatan sebelum dilakukan perubahan probabilitas. Semakin besar nilai `limitUpdateProbabilityNonRandomSlot` akan memberikan kesempatan pencarian solusi yang lebih banyak pada setiap nilai probabilitas. Hal ini juga berlaku pada `limitShuffle` yang menentukan batas kegagalan pada proses pengacakan solusi dan `limitStuck` membatasi jumlah batas tidak meningkatnya solusi pada tahapan *local search*.

Uji coba pada ketiga parameter dijalankan dengan beberapa nilai. Parameter `limitUpdateProbabilityNonRandomSlot` diuji dengan nilai 100, 1000, 10000, dan 100000. Parameter `limitShuflle` diuji pada nilai 100, 1000, dan 10000. Nilai `limitStuck` diuji dengan nilai 10, 20, dan 100. Hasil uji coba menemukan seluruh uji coba menghasilkan solusi *feasible*. Pada parameter `limitUpdateProbabilityNonRandomSlot`, nilai 100000 dipilih untuk digunakan dengan pertimbangan untuk memberikan kesempatan lebih dalam proses pencarian terutama untuk permasalahan yang sulit seperti dataset Middle 2 dan Middle 3 pada *benchmark* ITC 2021. Sementara nilai `limitShuffle` dipilih menggunakan nilai 100. Hal ini dikarenakan dalam strategi yang dikembangkan, proses pengacakan solusi akan dihentikan ketika ada satu saja solusi yang berhasil diacak. Dari pengujian sangat jarang menemukan solusi yang gagal diacak dalam 100 kali percobaan. Namun dengan pertimbangan untuk memberikan kesempatan lebih ketika proses pengacakan mengalami banyak kegagalan, maka nilai tersebut dipilih untuk digunakan. Sementara pada nilai `limitStuck`, nilai 20 digunakan dengan tujuan untuk memberikan kesempatan lebih dalam tahapan *local search*. Nilai 100 tidak dipilih karena nilai ini memberikan peningkatan waktu yang signifikan dalam menghasilkan solusi *feasible*. Secara keseluruhan nilai parameter yang akan digunakan pada lima parameter untuk uji coba pada tiga *benchmark* dan satu studi kasus ditampilkan dalam Tabel 4.5.3.

Tabel 4.5.3: Daftar Nilai Parameter

Parameter	Nilai
decreasingValue	0,30
constantFactor	0,85
limitUpdateProbabilityNonRandomTimeSlot	100000
limitStuck	20
limitShuffle	100