Institut
Teknologi
Sepuluh Nopember

TUGAS AKHIR — IF184802

DESAIN DAN ANALISA ALGORITMA STRATEGI PERMAINAN CATUR DENGAN METODE DIVIDE AND CONQUER

YUSTIAN
NRP 05111540000058

Dosen Pembimbing
RULLY SOELAIMAN, S.Kom., M.Kom.
RIZKY JANUAR AKBAR, S.Kom., M.Eng.

DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya 2019

[Halaman ini sengaja dikosongkan]

TUGAS AKHIR — IF184802

DESAIN DAN ANALISA ALGORITMA STRATEGI PERMAINAN CATUR DENGAN METODE DIVIDE AND CONQUER

YUSTIAN
NRP 05111540000058

Dosen Pembimbing
RULLY SOELAIMAN, S.Kom., M.Kom.
RIZKY JANUAR AKBAR, S.Kom., M.Eng.

DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya 2019

[Halaman ini sengaja dikosongkan]

UNDERGRADUATE THESIS — IF184802

DESIGN AND ANALYSIS OF CHESS GAME STRATEGY ALGORITHM USING DIVIDE AND CONQUER METHOD

YUSTIAN
NRP 05111540000058

Supervisors
RULLY SOELAIMAN, S.Kom., M.Kom.
RIZKY JANUAR AKBAR, S.Kom., M.Eng.

INFORMATICS DEPARTMENT
Information Technology and Communication Faculty
Institut Teknologi Sepuluh Nopember
Surabaya 2019

[Halaman ini sengaja dikosongkan]

LEMBAR PENGESAHAN

DESAIN DAN ANALISA ALGORITMA STRATEGI PERMAINAN CATUR DENGAN METODE DIVIDE AND CONQUER

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Algoritma Pemrograman
Program Studi S-1 Departemen Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh
Yustian
NRP. 05111540000058

Disetujui oleh Pembimbing Tugas Akhir:

1. Rully Soelaiman, S.Kom., M.Kom. .....
   NIP. 197002131994021001 (Pembimbing 1)
2. Rizky Januar Akbar, S.Kom., M.Eng. .....
   NIP. 198701032014041001 (Pembimbing 2)

SURABAYA
JANUARI 2019

[Halaman ini sengaja dikosongkan]

DESAIN DAN ANALISA ALGORITMA STRATEGI PERMAINAN CATUR DENGAN METODE DIVIDE AND CONQUER

Nama	: Yustian
NRP	: 0511 1540 0000 58
Departemen	: Departemen Informatika FTIK-ITS
Pembimbing 1	: Rully Soelaiman, S.Kom., M.Kom.
Pembimbing 2	: Rizky Januar Akbar, S.Kom., M.Eng.

Abstrak

Permasalahan yang dibahas pada Tugas Akhir ini terkait pada cara menentukan kondisi permainan dan jumlah langkah valid yang dapat dilakukan berdasarkan konfigurasi dari bidak catur pada papan catur yang diberikan, dengan jumlah bidak dan lebar papan yang jauh lebih banyak dan besar dibanding permainan catur pada umumnya. Karena kondisi yang tidak lazim ini perlu dilakukan pemecahan masalah menjadi sub-problem yaitu pencarian bidak terdekat yang dapat dilakukan dengan pencarian lower bound dan upper bound dari posisi sekarang. Untuk dapat mendukung pencarian bidak terdekat, informasi dari bidak disimpan dalam bentuk map-set yang dikelompokkan berdasarkan posisi x, posisi y, atau kombinasi dari x dan y.

Tugas Akhir ini memberikan sebuah penyelesaian terhadap permasalahan yang dideskripsikan pada paragraf pertama dengan pendekatan divide and conquer yang digabungkan dengan strategi permainan catur. Solusi dari permasalahan dilakukan dengan pertama menentukan untuk tiap bidak raja apakah berada dalam kon-

disi sekak, kemudian mencari jumlah langkah valid yang dapat dilakukan, dari kedua hasil proses sebelumnya kondisi permainan dapat ditentukan.

Solusi yang digunakan dalam menyelesaikan permasalahan ini memiliki kompleksitas O(p log q log q + 18q log q) dimana p adalah jumlah bidak dalam satu papan dan q adalah nilai minimal dari lebar papan dan jumlah bidak dalam papan.

Kata Kunci: Catur, Divide and Conquer, Lower Bound, Map, Set, Upper Bound

DESIGN AND ANALYSIS OF CHESS GAME STRATEGY ALGORITHM USING DIVIDE AND CONQUER METHOD

Name	: Yustian
NRP	: 0511 1540 0000 58
Major	: Informatics Department FTIK-ITS
Supervisor 1	: Rully Soelaiman, S.Kom., M.Kom.
Supervisor 2	: Rizky Januar Akbar, S.Kom., M.Eng.

Abstract

The problem imposed in this Final Project is how to determine the condition of chess game and the number of plausible move given a configuration of chess piece in a chess board with the number of chess piece and the length of board are far bigger than the one in the normal game of chess. Because of this abnormality there is a need to divide this problem to sub-problems which in this case is finding the closest chess pieces which can be achieved by finding the lower bound and upper bound of the current position. To be able to support the finding of closest chess pieces, the information of the chess piece should be saved in form of map-set where it is grouped based on their x position, y position, or the combination of both x and y.

This Final Project gives a solution to the problem described in the previous paragraph with divide and conquer approach combined with chess game strategy. The solution is done by firstly determine for each king whether it is in check, after that find the plausible move, based on the result of previous two process the game condition can be determined.

The solution used to solve this problem had O(p log q log q + 18q log q) where p is the number of chess piece on one board and q is the minimum of board length and number of piece.

Keyword: Chess, Divide and Conquer, Lower Bound, Map, Set, Upper Bound

KATA PENGANTAR

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa. Melalui berkat, rahmat, dan bimbingan-Nya, penulis dapat menyelesaikan Tugas Akhir yang berjudul:

DESAIN DAN ANALISA ALGORITMA STRATEGI PERMAINAN CATUR DENGAN METODE DIVIDE AND CONQUER

Penelition Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember.

Penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama penulis mengerjakan Tugas Akhir maupun selama menempuh masa studi antara lain:

- Kedua orang tua dan keluarga penulis yang selalu mendoakan dan memberikan dorongan kepada penulis untuk menyelesaikan Tugas Akhir ini.
- Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing yang telah banyak meluangkan waktu untuk memberikan ilmu, nasihat, motivasi, masukan, dan bimbingan kepada penulis baik selama penulis menempuh masa kuliah maupun selama perngerjaan Tugas Akhir ini.
- Bapak Rizky Januar Akbar, S.Kom., M.Eng. selaku Dosen Pembimbing yang telah meluangkan waktu, memberikan ilmu dan masukan kepada penulis.

- Brian, Bram, Steven, Iyem, Yuuta, Raca, Chasni, Frieda, Irsyad, Cynthia, dan Nuzul telah banyak membantu penulis selama perkuliahan dan penulisan buku ini dalam keilmuan maupun dukungan moral.
- Seluruh tenaga pengajar, asisten dosen, dan karyawan Departemen Informatika ITS yang telah memberikan ilmu dan waktunya demi berlangsungnya kegiatan belajar mengajar di Departemen Informatika ITS.
- Seluruh anggota *competitive programming* yang telah memberikan motivasi dalam belajar, ilmu, dan pengalaman kepada penulis.
- Pihak-pihak lain yang tidak dapat disebutkan disini yang telah membantu penulis selama berkuliah di Departemen Informatika.

Penulis mohon maaf apabila masih ada kekurangan pada Tugas Akhir ini. Penulis juga mengharapkan kritik dan saran yang membangun untuk pembelajaran dan perbaikan di kemudian hari. Semoga Tugas Akhir ini dapat menjadi kontribusi dan bermanfaat untuk perkembangan ilmu pengetahuan kedepannya.

Surabaya, Januari 2019

Yustian

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR KODE SUMBER

BAB 1

PENDAHULUAN

Bab ini meliputi latar belakang, masalah masalah apa saja yang dapat dirumuskan, tujuan, manfaat, metodologi apa saja yang dipakai, serta bagaimana Tugas Akhir dirumuskan dalam buku ini.

1.1 Latar Belakang Permasalahan

Catur adalah permainan papan dimana dua orang bergantian menggerakan bidak milik sendiri dengan tujuan akhir yaitu memojokan bidak raja lawan sehingga pihak lawan tidak dapat melakukan pergerakan yang valid lagi.

Dalam dunia komputer, telah banyak permasalahan yang berhungan dengan catur yang muncul seperti, penentuan kondisi papan permainan, penentuan jumlah langkah valid yang dapat dilakukan, penentuan jumlah langkah maksimal dengan kondisi permainan tertentu, hingga membuat mesin catur yang dapat bermain melawan manusia dalam turnamen catur.

Solusi untuk permasalahan di atas pun telah ada salah satunya adalah *bitboard* dimana kondisi dari tiap petak dalam papan catur direpresentasikan dalam kumpulan nilai biner. Namun solusi *bitboard* ini dibuat dengan memanfaatkan fakta bahwa papan catur berukuran 8x8 dan dapat direpresentasikan ke dalam 64-bit integer[4].

Pada tugas akhir ini, permasalahan yang diangkat adalah permasalahan penentuan kondisi permainan catur dan penentuan jumlah langkah valid pada permainan catur menggunakan papan berukuran besar seperti yang terdapat pada permasalahan SPOJ Klasik 32092 berjudul *Chessboard State 3: Intergalactic Chess Tournament*. Latar belakang permasalahan ini mengangkat permasalahan untuk menentukan kondisi papan permainan catur dan menentukan jumlah langkah yang valid yang dapat dilakukan pada T buah papan catur berukuran NxN yang berisikan P bidak catur dengan konfigurasi tertentu. Dimana T adalah bilangan bulat antara 1 sampai dengan 20000, N adalah bilangan bulat antara 8 sampai dengan 10^{18}, dan P adalah bilangan bulat antara 2 dan 200000. Tiap bidak pada papan diberikan dengan format x y c dimana x dan y adalah koordinat bidak pada papan catur dan c merepresentasikan tipe bidak yang mencakup bidak raja, bidak ratu, bidak menteri, bidak kuda, dan bidak prajurit

Dengan menggunakan pendekatan naif, dimana papan catur disimpan dalam bentuk *array* dua dimensi, maka dengan batasan lebar papan yang diberikan (lebar papan maksimal 10^{18}) akan melebihi batasan memori yang diperbolehkan. Hal yang sama juga terjadi dari segi batasan waktu eksekusi. Pada pendekatan naif untuk mencari bidak terdekat dari posisi tertentu dilakukan pengecekan ke petak demi petak hingga menemukan bidak, waktu eksekusi terburuk untuk proses ini adalah $O(n)$ dengan n adalah lebar papan, akan jauh melebihi batasan waktu yang diberikan.

Maka dari itu, penulis akan mencoba menyelesaikan permasalahan ini dengan menggabungkan metode *divide and conquer* dan startegi permainan catur dalam penentuan kondisi papan permainan dan penentuan jumlah langkah valid yang dapat dilakukan pada papan

catur dengan ukuran yang besar.

Hasil tugas akhir ini diharapkan dapat menentukan struktur data dan algoritma yang tepat untuk dapat menyelesaikan permasalahan diatas dengan optimal dan diharapkan dapat memberikan kontribusi pada perkembangan ilmu pengetahuan dan teknologi informasi.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam Tugas Akhir ini adalah sebagai berikut:

1. Bagaimanan menganalisa dan menentukan algoritma dan struktur data yang tepat dalam menyelesaikan permasalahan penentuan kondisi permainan dan jumlah langkah valid pada permainan catur berdasarkan konfigurasi *P* bidak pada papan berukuran *NxN*?
2. Bagaimana mengimplementasikan algoritma dan struktur data yang digunakan dalam menyelesaikan permasalahan penentuan kondisi permainan dan jumlah langkah valid pada permainan catur berdasarkan konfigurasi *P* bidak pada papan berukuran *NxN*?
3. Bagaimana hasil dari kinerja algoritma dan struktur data yang digunakan dalam menyelesaikan permasalahan penentuan kondisi permainan dan jumlah langkah valid pada permainan catur berdasarkan konfigurasi *P* bidak pada papan berukuran *NxN*?

1.3 Batasan Masalah

Permasalahan yang dibahas dalam Tugas Akhir ini memiliki berbagai macam batasan, yakni:

1. Implementasi daripada algoritma menggunakan bahasa pemrograman C++.
2. Uji coba kebenaran dilakukan dengan uji *submission* ke dalam situs penilaian daring SPOJ.
3. Batas maksimum jumlah papan dalam satu kasus uji adalah 20000.
4. Batas maksimum lebar papan adalah 10^{18}.
5. Batas maksimum jumlah bidak pada satu papan adalah $2x10^5$.
6. Batas maksimum jumlah bidak pada satu kasus uji adalah 10^6.
7. Dataset yang digunakan untuk pengujian kinerja algoritma adalah dataset pada permasalahan 32092 berjudul *Chessboard State 3: Intergalactic Chess Tournament*.

1.4 Tujuan

Tujuan dari Tugas Akhir ini adalah sebagai berikut:

1. Melakukan analisa dan desain algoritma untuk menyelesaikan permasalahan penentuan kondisi permainan dan jumlah langkah valid pada permainan catur berdasarkan konfigurasi *P* bidak pada papan berukuran *NxN*.
2. Melakukan evaluasi hasil dari kinerja algoritma dalam menyelesaikan permasalahan penentuan kondisi permainan dan jumlah langkah valid pada permainan catur berdasarkan konfigurasi *P* bidak pada papan berukuran *NxN*.

1.5 Manfaat

Tugas akhir ini diharapkan dapat membantu menyelesaikan permasalahan penentuan kondisi permainan dan jumlah langkah valid

pada permainan catur berdasarkan konfigurasi *P* bidak pada papan berukuran *NxN*.

1.6 Metodologi

Dalam mengerjakan Tugas Akhir, berikut ini adalah tahapan tahapan yang sudah dilalui oleh penulis:

1. Penyusunan Proposal Tugas Akhir
   Pada tahap ini, disusun proposal daripada Tugas Akhir yang berisikan gambaran umum terhadap analisis dan desain penyelesaian untuk permasalahan *Chessboard State 3: Intergalactic Chess Tournament*.
2. Studi Literatur
   Pada tahap ini, dilakukan pembelajaran terhadap segala metode, pendekatan, maupun teori teori yang berhubungan langsung dengan permasalahan Tugas Akhir. Sumber yang digunakan diambil dari internet, buku teks, serta jurnal yang pernah membahas permasalahan ini.
3. Desain Algoritma
   Pada tahap ini, proses desain mengenai kebutuhan apa saja yang dibutuhkan agar permasalahan *Chessboard State 3: Intergalactic Chess Tournament* dapat terselesaikan dan mendapatkan kinerja yang optimal.
4. Implementasi Algoritma
   Pada tahap ini, segala kebutuhan yang telah dapat teridentifikasikan pada tahap sebelumnya diimplementasikan dan digarap sedemikian rupa dengan menggunakan teknik-teknik pemrograman sehingga dihasilkan sebuah program yang bersesuaian.
5. Uji coba dan Evaluasi
   Pada tahap ini, uji coba dan evaluasi dieksekusi menggunakan bantuan daring SPOJ yang bersesuaian dengan masalah

yang ditujukan, untuk diuji apakah luaran dari program telah sesuai.

6. Penyusunan Buku Tugas Akhir
   Pada tahap ini, buku Tugas Akhir disusun sebagai bentuk dokumentasi daripada seluruh proses pengerjaan Tugas Akhir.

1.7 Sistematika Penulisan

Berikut sistematika yang digunakan dalam penulisan buku Tugas Akhir:

1. BAB I: PENDAHULUAN
   Bab ini berisi latar belakang permasalahan, rumusan, batasan masalah, tujuan Tugas Akhir, manfaat Tugas Akhir, metodologi, serta sistematika penulisan Tugas Akhir.
2. BAB II: DASAR TEORI
   Bab ini berisikan landasan-landasan teori mengenai pendekatan dan implementasi algoritma yang digunakan untuk menyelesaikan Tugas Akhir.
3. BAB III: DESAIN ALGORITMA
   Bab ini menjelaskan bagaimana algoritma didesain terlebih dahulu dengan melakukan analisa kebutuhan untuk permasalahan *Chessboard State 3: Intergalactic Chess Tournament* yang ada pada daring SPOJ.
4. BAB IV: IMPLEMENTASI ALGORITMA
   Bab ini berisi implementasi-implementasi dari algoritma yang digunakan untuk menyelesaikan tiga permasalahan yang terdapat pada daring SPOJ dengan mengikuti kebutuhan-kebutuhan yang telah dispesifikasikan pada bab sebelumnya.
5. BAB V: UJI COBA DAN EVALUASI
   Bab ini berisikan uji coba dan evaluasi terhadap implementasi yang telah dijelaskan pada bab sebelumnya.

6. BAB VI: KESIMPULAN
   Bab ini berisikan kesimpulan yang dapat diambil dari hasil uji coba dan evaluasi yang telah dibahas pada bab sebelumnya.

[Halaman ini sengaja dikosongkan]

BAB 2

DASAR TEORI

2.1 Deskripsi Umum Permasalahan

Diberikan *T* buah papan catur berukuran *NxN* dimana tiap papan catur berisikan *P* bidak dan tiap bidak diberikan dalam format *x y c* yang berarti bidak dengan tipe *c* berada pada koordinat *x, y* pada papan permainan yang bersangkutan. Tipe bidak mencakup raja (K/k), ratu(Q/q), menteri(B/b), kuda(H/h), benteng(R/r), dan prajurit(P/p) di mana huruf kecil menandakan bidak berwarna putih dan huruf besar menandakan bidak berwarna hitam.

Untuk setiap papan yang diberikan berlaku aturan sebagai berikut yaitu pada awal permainan bidak putih menempati area atas papan dan bidak hitam menempati area bawah papan. Hal ini menyebabkan bidak prajurit putih bergerak menuju arah bawah (arah *y* positif) dan bidak prajurit hitam bergerak ke arah atas (arah *y* negatif) dengan acuan posisi (1,1) papan catur berada pada pojok kiri atas. Selain itu pada papan catur yang diberikan akan terdapat minimal dua buah bidak yaitu bidak raja putih dan raja hitam dan perpindahan posisi tiap bidak bergantung pada cara bergerak masing-masing bidak.

Dalam permasalahan ini pergerakan yang berlaku adalah pergerakan dasar dari tiap bidak catur, secara spesifik

- Bidak raja dapat bergerak ke semua arah dengan jarak pergerakan sama dengan satu.
- Bidak ratu dapat bergerak ke semua arah dengan sembarang

jarak pergerakan atau hingga menemui bidak lain atau ujung papan.

- Bidak menteri dapat bergerak ke arah diagonal dengan sembarang jarak pergerakan atau hingga menemui bidak lain atau ujung papan.

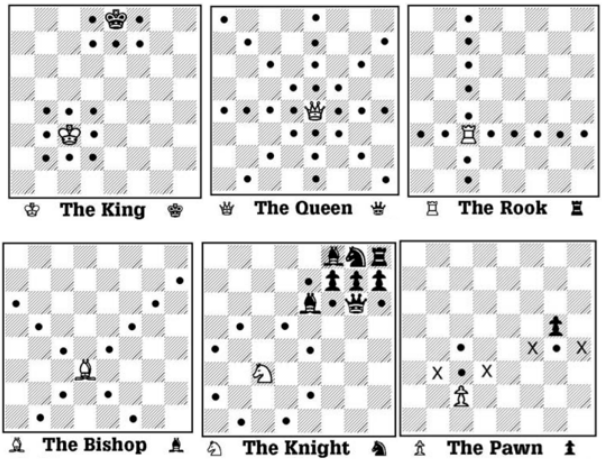
- Bidak benteng dapat bergerak ke arah horisontal atau vertikal dengan sembarang jarak pergerakan atau hingga menemui bidak lain atau ujung papan.

- Bidak prajurit dapat bergerak satu langkah ke arah depan. Arah depan di sini tergantung dari warna bidak, dimana dalam permasalahan ini prajurit putih bergerak ke arah bawah (y positif) dan prajurit hitam bergerak ke arah atas (y negatif) dari papan catur.

- Bidak kuda dapat bergerak ke salah satu dari delapan posisi yang terdiri dari gabungan dua langkah secara horisontal ditambah satu langkah secara vertikal, atau satu langkah secara horisontal dan dua langkah secara vertikal.

Selain bergerak bidak juga dapat memakan bidak lawan. Memakan bidak lawan dapat dilakukan bila bidak lawan berada dalam jangkauan bidak sendiri dengan memindahkan bidak kawan tersebut ke petak yang berisikan bidak lawan, bidak lawan yang dimakan kemudian dikeluarkan dari papan permainan. Cara memakan bidak ini berlaku untuk semua jenis bidak kecuali bidak prajurit. Bidak prajurit memiliki cara memakan yang berbeda dari cara bergerak yaitu dapat memakan pada bidak yang berada di depan prajurit pada arah dia- gonal berjarak satu. Ilustrasi untuk pergerakkan catur dapat dilihat pada Gambar 2.1.

Dengan mengacu pada aturan di atas diminta untuk menentukan kondisi permainan berdasarkan masukan yang diberikan yang mencakup:



Gambar 2.1: Pergerakan dan Cara Makan Bidak Catur

- ”Safe” bila kedua raja tidak dalam kondisi sekak.
- ”Impossible” bila kedua raja dalam kondisi sekak.
- ”*White/Black* Check - *m* Plausible Moves” bila salah satu raja dalam kondisi sekak, dan ada *m* langkah valid untuk keluar dari kondisi sekak.
- ”*White/Black* Checkmate” bila salah satu dari raja dalam kondisi sekak, dan tidak ada langkah valid yang dapat dilakukan lagi.

2.2 Deskripsi Umum

Pada subbab ini, berbagai landasan teori umum yang digunakan untuk melakukan pendekatan terhadap permasalahan ini.

2.2.1 Slider

Slider dalam permainan catur didefinisikan sebagai bidak yang dapat bergerak dalam satu garis horisontal, vertikal, atau diagonal dengan jumlah langkah berapapun hingga mencapai ujung papan catur atau terhalang bidak lain. Bidak yang termasuk *slider* adalah benteng, menteri, dan ratu.

2.2.2 Pinned

Pinned dalam permainan catur adalah kondisi suatu bidak yang dimana jika bidak tersebut digerakkan akan menyebabkan bidak raja milik sendiri berada dalam kondisi sekak. Hal ini menyebabkan pergerakkan dengan menggunakan bidak yang berada dalam kondisi pinned menjadi tidak valid.

2.2.3 Sekak

Sekak (*Check*) adalah kondisi dimana bidak raja pemain berada dalam jangkauan serang bidak lawan yang memungkinkan lawan untuk memakan (*capture*) bidak raja pada giliran berikutnya. Pemain dapat keluar dari kondisi sekak dengan melakukan *intercepting* atau menghalangi pergerakan bidak penyerang menuju bidak raja, memakan bidak penyerang, atau menggerakkan bidak raja menuju posisi dimana bidak raja tidak lagi berada dalam kondisi sekak. Jika tidak memungkinkan untuk keluar dari kondisi sekak, maka permainan berakhir dan pemain dinyatakan kalah.

2.2.4 Divide and Conquer

Divide and conquer adalah desain algoritma yang memecah suatu permasalahan kompleks menjadi permasalahn yang lebih sederhana sehingga dapat diselesaikan dengan lebih mudah [2]. Salah satu contoh algoritma *Divide and Conquer* adalah *binary search*.

2.2.5 Red-Black Tree

Red-Black Tree adalah variasi dari *self balancing tree* dimana tiap elemen dari *tree* tersebut berisi informasi ekstra berupa warna yang berguna untuk menjaga keseimbangan dari *tree* [1], sehingga proses penambahan, penghapusan, dan pencarian elemen pada *tree* tersebut dapat dilakukan dalam *O(log n)* di mana *n* adalah jumlah elemen pada *red-black tree*.

2.2.6 Set

Set adalah struktur data yang digunakan untuk menyimpan elemen yang tidak berulang dengan urutan tertentu. Elemen pada set tidak dapat dimodifikasi, namun dapat dilakukan penambahan dan penghapusan elemen. Implementasi dari Set pada bahasa pemrograman C++ adalah dengan menggunakan *red-black tree*.

2.2.7 Unordered Map

Unordered Map adalah struktur data yang dibentuk dari kombinasi pasangan *key value* dan *mapped value*. Struktur data ini digunakan untuk memetakan suatu nilai unik (*key value*) ke objek (*mapped value*) yang bersesuaian. Dalam *unordered map* elemen disimpan tanpa urutan tertentu, namun dikelompokkan berdasarkan nilai *hash* masing-masing elemen sehingga memungkinkan untuk melakukan akses ke elemen dengan *key value* tertentu dalam waktu konstan.

2.2.8 Lower Bound

Lower bound adalah elemen pada suatu kumpulan elemen yang memiliki nilai kurang dari atau sama dengan nilai tertentu yang digunakan sebagai acuan. Pada bahasa pemrograman C++ pencarian *lower bound* diimplementasikan dengan menggunakan modifikasi dari *binary search* [2], sehingga memiliki kompleksitas pencarian *O(log n)* di mana *n* adalah jumlah anggota kumpulan elemen[3].

Pencarian ini hanya dapat diaplikasikan pada kumpulan elemen yang telah terurut.

2.2.9 Upper Bound

Upper bound adalah elemen pada suatu kumpulan elemen yang memiliki nilai lebih besar dari suatu nilai tertentu yang digunakan sebagai acuan. Pada bahasa pemrograman C++ pencarian *upper bound* diimplementasikan dengan menggunakan modifikasi dari *binary search* sehingga memiliki kompleksitas pencarian *O(log n)* di mana *n* adalah jumlah anggota kumpulan elemen[3]. Pencarian ini hanya dapat diaplikasikan pada kumpulan elemen yang telah terurut[2].

2.3 Strategi Penyelesaian

Pada subbab ini akan dipaparkan mengenai strategi penyelesaian pada permasalahan klasik pada daring SPOJ dengan kode CSTATE3. Strategi penyelesaian yang digunakan didasarkan pada penggunaan metode *divide and conquer* dan digabungkan dengan strategi permainan catur untuk memecah permasalahan besar pencarian kondisi dan jumlah langkah valid pada suatu konfigurasi papan menjadi permasalahan kecil yaitu sejumlah pencarian bidak terdekat dengan menggunakan pencarian *lower bound* dan *upper bound* dari posisi tertentu ditambah dengan sejumlah perbandingan untuk menyelesaikan permasalahan besar di awal. Secara singkat, strategi penyelesaian permasalahan dari CSTATE3 ini terbagi menjadi empat bagian besar yaitu:

1. Representasi bidak catur relatif terhadap posisi pada papan permainan.
2. Pembangunan struktur data map-set untuk menampung informasi dari posisi bidak.
3. Struktur data pendukung sebagai komplemen dari struktur data map-set.

4. Penentuan raja dalam kondisi sekak.
5. Penentuan jumlah langkah valid.
6. Penentuan kondisi permainan

2.3.1 Representasi Bidak Catur Relatif Terhadap Posisi Pada Papan Permainan

Papan catur dapat dipandang melalui empat sudut pandang yaitu secara horizontal, vertikal, main diagonal, dan anti diagonal. Ketika papan dipandang melalui salah satu sudut pandang tersebut, akan terlihat kelompok bidak yang terdiri dari bidak yang berada pada satu garis lurus pada sudut pandang itu. Sebagai contoh, jika dilihat melalui sudut pandang horizontal maka bidak yang berada pada satu garis horizontal yang sama akan membentuk satu kelompok bidak. Atau secara formal,

- Secara horizontal bidak dikelompokkan berdasarkan posisi y bidak.
- Secara vertikal bidak dikelompokkan berdasarkan posisi x bidak.
- Secara main diagonal bidak dikelompokkan berdasarkan selisih posisi x dan posisi y bidak.
- Secara anti diagonal bidak dikelompokkan berdasarkan jumlah posisi x dan posisi y bidak.

Sebagai contoh jika pada papan terdapat bidak raja pada posisi (3,4), bidak ratu pada posisi (6,4), dan bidak kuda pada posisi (6,1) maka secara horizontal akan terdapat dua kelompok bidak berdasarkan posisi y tiap bidak yaitu kelompok bidak y = 1 dan kelompok bidak y = 4. Secara vertikal terdapat dua kelompok bidak berdasarkan posisi x tiap bidak yaitu kelompok bidak x = 3 dan kelompok bidak x = 6. Secara main diagonal terdapat tiga kelompok bidak berdasarkan posisi x - y tiap bidak yaitu kelompok bidak

x - y = -1, kelompok bidak x - y = 2, dan kelompok bidak dengan x - y = 5. Secara anti diagonal terdapat dua kelompok bidak berdasarkan posisi x + y tiap bidak yaitu kelompok bidak x + y = 7 dan kelompok bidak x + y = 10.

2.3.2 Pembangunan Struktur Data Map-Set Untuk Menampung Informasi dari Bidak

Dengan menggunakan cara pengelompokkan pada subbab 2.3.1, dapat digunakan struktur data *map* untuk memetakan nilai yang menjadi dasar pengelompokkan (misal: posisi y pada sudut pandang horizontal) terhadap suatu kelompok nilai yang merepresentasikan posisi bidak. Informasi yang disimpan tidak dalam bentuk informasi bidak lengkap (pasangan nilai x dan y dari bidak) namun berupa informasi parsial, dimana untuk mendapat informasi lengkap digunakan kombinasi dari informasi yang disimpan dalam kelompok nilai yang dipetakan (*mapped value*) dengan nilai yang digunakan untuk memetakan (*key value*) pada struktur data *map*. Secara singkat,

- Secara horisontal informasi yang disimpan adalah kelompok nilai x dari bidak yang memiliki nilai y yang sama. Informasi lengkap didapat dari gabungan nilai x yang disimpan dan nilai y yang digunakan sebagai *key value*.

- Secara vertikal informasi yang disimpan adalah kelompok nilai y dari bidak yang memiliki nilai x yang sama. Informasi lengkap didapat dari gabunagan nilai x yang digunakan sebagai *key value* dan nilai y yang disimpan.

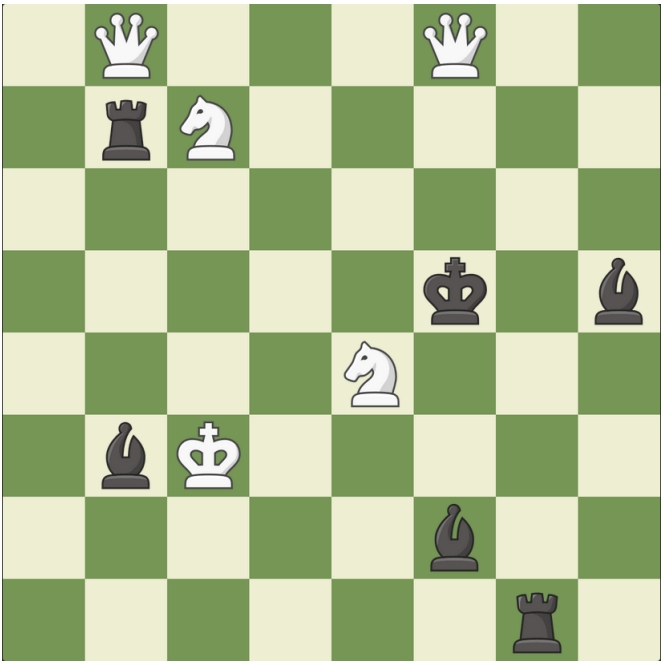
- Secara *main diagonal* informasi yang disimpan adalah kelompok nilai x dari bidak yang memiliki selisih nilai x dan y yang sama. Informasi lengkap didapat dari gabungan nilai x yang disimpan dan nilai y yang didapat dengan mencari selisih dari nilai x yang disimpan dengan nilai *key* yang digunakan.

- Secara *anti diagonal* informasi yang disimpan adalah kelompok nilai x dari bidak yang memiliki jumlah nilai x dan y yang sama. Informasi lengkap didapat dari gabungan nilai x yang disimpan dan nilai y yang didapat dengan mencari selisih dari nilai *key* yang digunakan dengan nilai x yang disimpan.

Struktur data yang digunakan untuk menampung kelompok nilai yang dideskripsikan adalah STL(*standard template library*) *set*. *Set* digunakan dengan pertimbangan bahwa posisi bidak yang diberikan melalui masukan adalah unik dan tidak terurut, sehingga untuk pembangunan awal struktur data *map of set* ini memiliki kompleksitas *O(p log q log q)* dimana *p* adalah jumlah bidak keseluruhan dan q adalah nilai minimal antara jumlah bidak dan besar papan, dan untuk perubahan posisi bidak pada proses selanjutnya memiliki kompleksitas *O(log n)*. Sebagai Contoh dari struktur data *map-set* untuk papan catur pada Gambar 2.2 dapat dibangun empat struktur data *map-set* yang masing-masing merepresentasikan sudut pandang yang berbeda yang dapat dilihat pada Gambar 2.3 untuk sudut pandang horisontal, Gambar 2.4 untuk sudut pandang vertikal, Gambar 2.5 untuk sudut pandang *main diagonal*, dan Gambar 2.6 untuk sudut pandang *anti diagonal*.

2.3.3 Struktur Data Pendukung Sebagai Komplemen dari Struktur Data Map-Set

Pada subbab 2.3.2 terlihat bahwa informasi yang tersimpan hanya berupa koordinat dari bidak. Maka perlu struktur data tambahan untuk menentukan bidak pada posisi *(x, y)* merupakan bidak dengan tipe apa, sehingga perlu ditambahkan STL *map* yang memetakan koordinat ke tipe bidak. Selain itu untuk mempermudah operasi pada subbab 2.4 dan 2.5 diperlukan struktur data tambahan berupa STL *unordered map* yang memetakan tipe bidak ke *vector* yang berisi objek bidak (objek bidak berisikan koordinat bidak dan tipe bidak). Sebagai contoh untuk bidak pada Gambar 2.2 akan di-



Gambar 2.2: Contoh Konfigurasi Bidak Pada Papan Catur

```
horizontal = {
        # pos_y: set<pos_x>
        (1): [2, 6],
        (2): [2, 3],
        (4): [6, 8],
        (5): [5],
        (6): [2, 3],
        (7): [6],
        (8): [7]
}
```

Gambar 2.3: *Map-Set* Horisontal

```
vertical = {
        # pos_x: set<pos_y>
        (2): [1, 2, 6],
        (3): [2, 6],
        (5): [5],
        (6): [1, 4, 7],
        (7): [8],
        (8): [4]
}
```

Gambar 2.4: *Map-Set* Vertikal

```
main_diagonal = {
        # (pos_x - pos_y): set<pos_x>
        (0): [2, 5],
        (1): [2, 3],
        (2): [6],
        (4): [8],
        (5): [6],
        (-1): [6, 7],
        (-3): [3],
        (-4): [2]
}
```

Gambar 2.5: *Map-Set Main Diagonal*

```
anti_diagonal = {
        # (pos_x + pos_y): set<pos_x>
        (3): [2],
        (4): [2],
        (5): [3],
        (7): [6],
        (8): [2],
        (9): [3],
        (10): [5, 6],
        (12): [8],
        (13): [6],
        (15): [7]
}
```

Gambar 2.6: *Map-Set Anti Diagonal*

```
{
        # tipe : vector<(pos_x, pos_y)>
        WHITE_QUEEN: [(2,1,WHITE_QUEEN),
                (6,1,WHITE_QUEEN)],
        BLACK_ROOK: [(2,2,BLACK_ROOK),
                (7,8,BLACK_ROOK)],
        WHITE_HORSE: [(3,2,WHITE_HORSE),
                (5,5,WHITE_HORSE)],
        BLACK_KING: [(6,4,BLACK_KING)],
        BLACK_BISHOP: [(8,4,BLACK_BISHOP),
                (2,6,BLACK_BISHOP),
                (6,7,BLACK_BISHOP)],
        WHITE_KING: [(3,6,WHITE_KING)]
}
```

Gambar 2.7: Struktur Data *Map* Tipe Bidak Menuju Objek Bidak

hasilkan struktur data tambahan sebagaimana terlihat pada Gambar 2.8 untuk *map* koordinat bidak ke tipe bidak dan Gambar 2.7 untuk *unordered map* tipe bidak ke *vector* berisikan objek bidak dengan tipe yang menjadi *key value*.

```
{
        # (pos_x, pos_y) : tipe
        (2,1): WHITE_QUEEN,
        (6,1): WHITE_QUEEN,
        (2,2): BLACK_ROOK,
        (7,8): BLACK_ROOK,
        (3,2): WHITE_HORSE,
        (5,5): WHITE_HORSE,
        (6,4): BLACK_KING,
        (8,4): BLACK_BISHOP,
        (2,6): BLACK_BISHOP,
        (6,7): BLACK_BISHOP,
        (3,6): WHITE_KING
}
```

Gambar 2.8: Struktur Data Map Koordinat Menuju Tipe Bidak

2.4 Penentuan Raja Dalam Kondisi Sekak

Untuk setiap bidak raja pada papan catur yang diberikan dilakukan empat evaluasi sebagai berikut,

1. Pencarian bidak terdekat ke delapan arah dari posisi raja.
2. Pencarian bidak pada delapan petak yang dapat dituju dengan cara bergerak kuda.
3. Pencarian bidak pada dua petak yang dapat dituju dengan cara bergerak prajurit.
4. Pencarian bidak pada delapan petak disekitar raja.

2.4.1 Pencarian Bidak Terdekat Delapan Arah dari Posisi Raja

Pencarian dilakukan dengan tujuan untuk menentukan apakah ada *slider* lawan pada delapan arah dari raja yang dapat bergerak ke petak raja. *Slider* dibagi menjadi tiga yaitu *slider* yang dapat bergerak secara horisontal atau vertikal (bidak benteng), *slider* yang dapat bergerak secara diagonal (bidak menteri) dan keduanya (bidak

ratu). Posisi bidak terdekat dari bidak raja dapat ditentukan dengan mencari nilai *upper bound* dan *lower bound* pada struktur data *map-set* yang telah dibangun pada subbab 2.3.2, pada *set* yang bersesuaian dengan *key value* yang didapat dari posisi raja pada sudut pandang yang bersesuaian, atau dengan lebih terperinci,

- Posisi bidak pada arah utara dan selatan dari raja didapat dengan mencari nilai upper bound (selatan) dan lower bound (utara) pada kelompok bidak dengan *key value* sama dengan posisi x raja dari sudut pandang vertikal.
- Posisi bidak pada arah timur dan barat dari raja didapat dengan mencari nilai upper bound (barat) dan lower bound (timur) pada kelompok bidak dengan *key value* sama dengan posisi y raja dari sudut pandang horizontal.
- Posisi bidak pada arah tenggara dan barat laut dari raja didapat dengan mencari nilai upper bound (barat laut) dan lower bound (tenggara) pada kelompok bidak dengan *key value* sama dengan jumlah nilai posisi x dan y raja dari sudut pandang *anti diagonal*.
- Posisi bidak pada arah timur laut dan barat daya dari raja didapat dengan mencari nilai upper bound (barat daya) dan lower bound (timur laut) pada kelompok bidak dengan *key value* sama dengan selisih dari nilai posisi x dan y raja dari sudut pandang *main diagonal*.

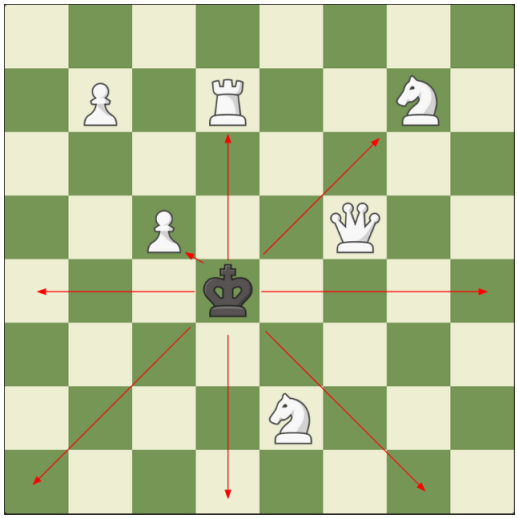
Untuk setiap nilai *upper bound* dan *lower bound* yang ditemukan pada tiap sudut pandang di atas dilakukan pengecekan dengan menggunakan struktur data *map* (subbab 2.3.3) untuk menentukan tipe bidak pada posisi yang ditemukan. Berdasarkan tipe bidak tersebut ditentukan apakah bidak merupakan bidak lawan dan tergolong *slider* yang dapat bergerak ke arah raja, jika ya maka raja dikatakan berada dalam kondisi sekak dari arah tersebut. Jika bidak

yang ditemukan merupakan bidak kawan, maka dilakukan pencarian bidak terdekat kedua (dengan mencari *upper bound* atau *lower bound* dari posisi bidak yang ditemukan) jika bidak terdekat kedua merupakan bidak lawan dan tergolong *slider* yang dapat bergerak ke arah raja maka, bidak terdekat pertama (yang merupakan bidak kawan), diberi status *pinned* yang berarti jika bidak tersebut digerakkan maka raja akan berada dalam kondisi sekak (langkah tidak valid).

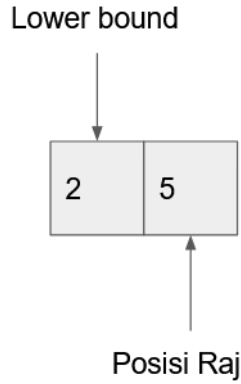
Sebagai contoh pada Gambar 2.9 jika dilihat secara vertikal, maka pencarian dilakukan pada set yang bersesuaian dengan posisi *x* dari bidak raja. Hasil pencarian *lower bound* mendapatkan hasil 2 terlihat pada Gambar 2.10. Nilai 2 ini merupakan posisi *y* dari bidak terdekat dengan bidak raja. Koordinat dari bidak didapat melalui gabungan nilai yang didapat dari pencarian yaitu 2 dan *key value* dari set dimana pencarian dilakukan, dalam contoh ini *key value* yang digunakan adalah sama dengan posisi *x* dari bidak raja yaitu 4. Sehingga didapat koordinat bidak terdekat *(x,y) = (4,2)* dimana pada posisi tersebut dengan menggunakan *map* koordinat ke tipe bidak (subbab 2.3.3) diketahui bahwa bidak pada koordinat tersebut adalah bidak benteng putih. Ilustrasi pencarian *lower bound* yang dijelaskan dapat dilihat pada Gambar 2.10.

2.4.2 Pencarian Bidak Kuda Lawan Pada Delapan Petak yang Dapat Dituju dengan Cara Bergerak Kuda

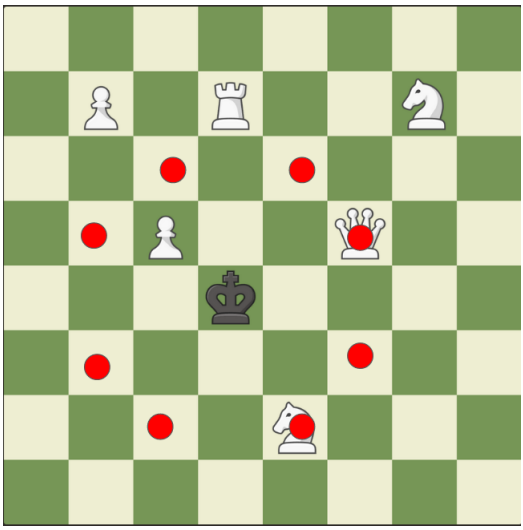
Pencarian dilakukan untuk menentukan apakah ada bidak kuda lawan yang dapat bergerak ke petak raja dengan cara melihat apakah pada delapan petak yang dapat dituju dari posisi raja sekarang dengan menggunakan pergerakan kuda pada permainan catur berisikan bidak dan apakah bidak tersebut merupakan bidak kuda lawan, jika kondisi tersebut benar untuk satu atau lebih petak maka raja dikatakan dalam kondisi sekak. Ilustrasi untuk pencarian ini dapat dilihat pada Gambar 2.11.



Gambar 2.9: Pencarian ke Delapan Arah dari Raja



Gambar 2.10: Pencarian *Lower Bound* vertikal pada Gambar 2.9 dengan *key value = 4*



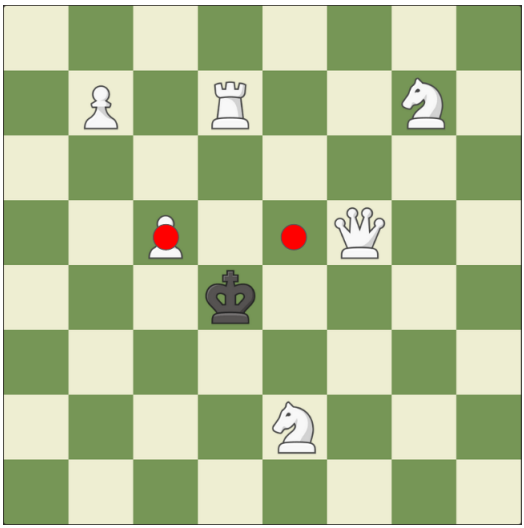
Gambar 2.11: Pencarian ke Delapan Petak Pergerakkan Kuda

2.4.3 Pencarian Bidak Prajurit Lawan ke Dua Arah Cara Makan Prajurit

Pencarian dilakukan untuk menentukan apakah ada bidak prajurit lawan yang dapat bergerak ke petak raja dengan cara melihat pada dua petak di mana dari petak tersebut prajurit lawan dapat berpindah ke petak raja. Jika pada salah satu atau kedua petak tersebut terdapat prajurit lawan maka, raja dikatakan dalam kondisi sekak. Ilustrasi untuk pencarian ini dapat dilihat pada Gambar 2.12.

2.4.4 Pencarian Bidak Raja Lawan ke Delapan Petak di sekitar Bidak Raja

Pencarian dilakukan untuk menentukan apakah ada bidak raja lawan yang dapat bergerak ke petak raja sendiri dengan cara melihat pada delapan petak di sekitar bidak raja sendiri sesuai cara bergerak bidak raja. Jika pada salah satu petak tersebut terdapat raja lawan maka,



Gambar 2.12: Pencarian ke Dua Petak Cara Makan Prajurit

raja sendiri dikatakan dalam kondisi sekak. Ilustrasi untuk pencarian ini dapat dilihat pada Gambar 2.13.

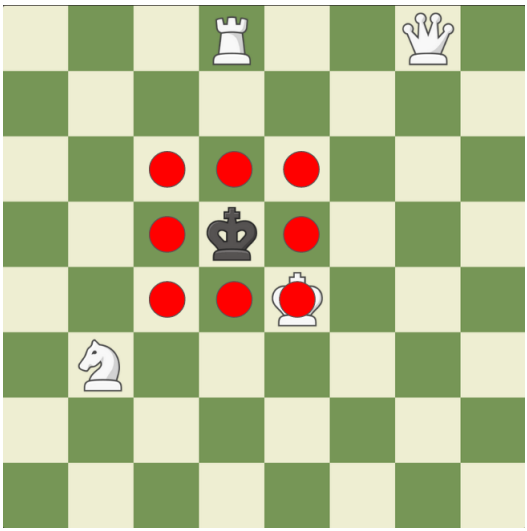
2.5 Penentuan Jumlah Langkah Valid

Jika bidak raja dalam kondisi sekak, maka langkah yang dapat dilakukan pemain adalah langkah yang dapat membawa raja keluar dari kondisi sekak. Jumlah langkah valid total didapatkan dengan menggabungkan jumlah langkah yang didapat dari tiga cara di bawah yaitu

1. Menggerakkan raja yang berada dalam kondisi sekak.

2. Mencari bidak teman yang dapat memakan bidak penyerang.

3. Mencari bidak teman yang dapat melakukan intersepsi bidak penyerang.



Gambar 2.13: Pencarian ke Delapan Petak Cara Gerak Raja

2.5.1 Jumlah Langkah dari Pergerakkan Bidak Raja

Jumlah langkah dari pergerakan raja didapatkan dengan menggerakkan raja ke petak yang dapat dicapai dari posisi raja sekarang sesuai dengan cara gerak raja yaitu delapan petak di sekitar raja sekarang. Raja dapat berpindah ke petak yang dituju jika pada petak tujuan kosong atau bidak yang berada pada petak tersebut adalah bidak lawan. Untuk setiap posisi baru raja, dilakukan pengecekan kondisi sekak (subbab 2.4) jika hasil pengecekan adalah raja tidak dalam kondisi sekak pada posisi baru maka jumlah langkah valid bertambah satu.

2.5.2 Jumlah Langkah dari Bidak Kawan Memakan Bidak Lawan

Memakan bidak, dalam permainan catur dilakukan dengan cara menggerakkan salah satu bidak kawan sesuai dengan cara gerak bidak yang bersangkutan, menuju petak dimana bidak lawan berada. Jumlah dari pergerakan bidak kawan untuk memakan bidak penyerang (bidak lawan yang menyekak bidak raja sendiri) dilakukan jika jumlah bidak penyerang sama dengan satu. Untuk mendapatkan jumlah bidak kawan yang dapat memakan bidak penyerang adalah dengan menggunakan penentuan kondisi sekak pada bidak penyerang (menganggap bidak penyerang sebagai bidak raja), jumlah bidak yang dapat menyebabkan sekak pada bidak penyerang adalah jumlah langkah valid yang dapat dilakukan. Dengan catatan bidak yang digunakan untuk menyebabkan sekak pada bidak penyerang tidak boleh berada dalam kondisi *pinned*.

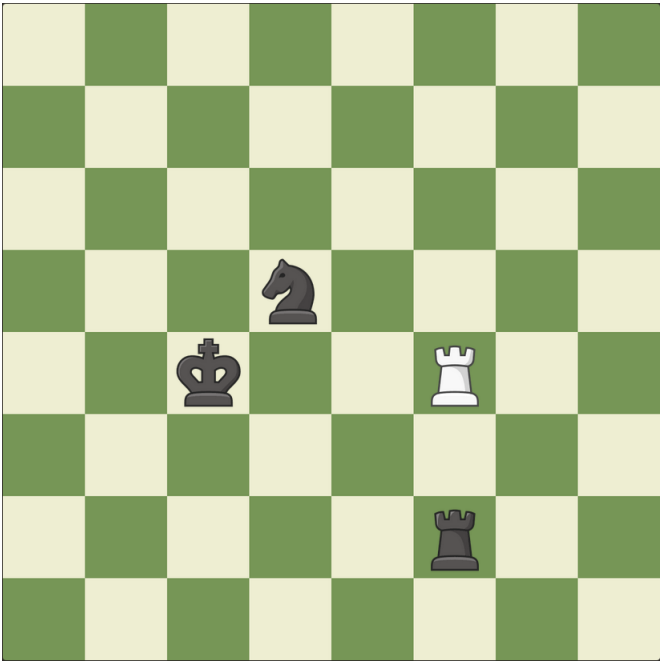
Pada Gambar 2.14 bidak raja hitam diserang oleh benteng putih (bidak penyerang), dengan mencari jumlah sekak pada posisi benteng putih (subbab 2.4), didapatkan jumlah bidak hitam yang dapat berpindah ke posisi benteng putih.

2.5.3 Jumlah Langkah dari Intersepsi Bidak Kawan

Intersepsi pada permainan catur dilakukan dengan cara menggerakkan bidak kawan ke petak yang berada pada jalur pergerakkan dari bidak penyerang menuju bidak raja. Penentuan jumlah langkah melalui intersepsi serangan oleh bidak kawan dilakukan jika jumlah bidak penyerang sama dengan satu dan bidak penyerang tersebut tergolong *slider*. Cara penentuan jumlah langkah dapat dibagi menjadi dua bagian yaitu

1. Intersepsi oleh bidak kawan yang tergolong *slider*.

2. Intersepsi oleh bidak kawan yang tergolong bukan *slider*.



Gambar 2.14: Memakan Bidak Penyerang

2.5.3.1 Intersepsi oleh Bidak Kawan yang Tergolong Slider

Jumlah intersepsi oleh bidak kawan yang tergolong *slider* dilakukan dengan mencari bidak terdekat dari setiap petak yang berada di antara bidak raja dan bidak penyerang, dan ada bidak yang memiliki kemungkinan dapat berpindah menuju petak tersebut. Atau dengan kata lain untuk setiap kelompok bidak pada tiap sudut pandang (horisontal, vertikal, *main diagonal*, dan *anti diagonal*) yang berada di antara bidak raja dan bidak penyerang, dilakukan pencarian bidak terdekat dari petak yang berada pada jalur antara bidak raja dan bidak penyerang, untuk setiap bidak terdekat yang ditemukan dan merupakan bidak kawan yang tergolong *slider* maka jumlah langkah valid bertambah satu.

Sebagai contoh pada Gambar 2.15 perhitungan jumlah intersepsi dilakukan dengan cara,

- Pada petak A, tidak dilakukan pencarian apapun karena tidak ada kelompok bidak pada enam arah (arah yang darimana serangan datang tidak diperhitungkan) dari posisi petak A.
- Pada petak B, dilakukan pencarian bidak terdekat secara vertikal dan *anti diagonal*. Pencarian secara vertikal mendapatkan dua buah bidak yaitu kuda hitam pada koordinat (3,3) dan benteng hitam pada koordinat (3,7). Untuk kuda hitam dapat diabaikan karena tidak tergolong *slider*, untuk benteng hitam karena merupakan slider dan dapat bergerak menuju petak B, maka jumlah intersepsi bertambah satu. Pencarian secara *anti diagonal* mendapatkan bidak menteri hitam pada koordinat (6,2) yang merupakan *slider* dan dapat bergerak ke petak B sehingga jumlah intersepsi bertambah satu.
- Pada petak C, dilakukan pencarian bidak terdekat secara *main diagonal* dimana mendapatkan bidak menteri hitam pada koordinat (7,8) yang merupakan *slider* dan dapat bergerak ke petak C sehingga jumlah intersepsi bertambah satu.

- Pada petak D, dilakukan pencarian bidak terdekat secara *main diagonal* dan *anti diagonal*. Pencarian secara *main diagonal* mendapatkan bidak kuda hitam pada koordinat (3,3) yang dapat diabaikan karena bukan merupakan *slider*. Pencarian secara *anti diagonal* mendapatkan bidak benteng hitam pada koordinat (3,7) yang merupakan *slider* namun tidak dapat bergerak menuju petak D sehingga jumlah intersepsi tidak bertambah.

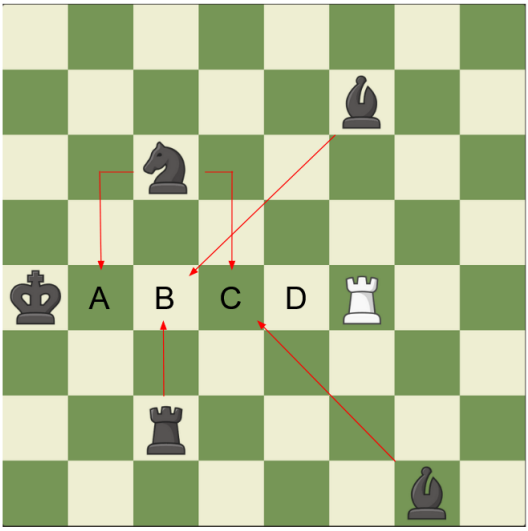
2.5.3.2 Intersepsi oleh Bidak Kawan yang Tergolong Bukan Slider

Bidak kawan yang tergolong non slider adalah kuda dan prajurit, pencarian jumlah langkah valid yang dapat dilakukan didapat dengan cara, menggerakkan setiap bidak kuda dan prajurit sesuai dengan cara gerak masing-masing, untuk setiap posisi baru dari cara gerak bidak (delapan posisi untuk kuda dan dua posisi untuk prajurit) jika posisi tersebut berada di jalur antara bidak raja dan bidak penyerang maka jumlah langkah valid bertambah satu. Sebagai contoh pada Gambar 2.15 untuk bidak kuda hitam pada koordinat (3,3) dilakukan pengecekan ke delapan kemungkinan gerak yaitu [(1,2), (2,1), (4,1), (5,2), (5,4), (4,5), (2,5), (1,4)]. Dari ke delapan kemungkinan pergerakan tersebut, yang jatuh pada jalur antara bidak raja dan bidak penyerang hanya koordinat (2,5) dan (4,5) yaitu pada petak A dan C, sehingga jumlah intesepsi bertambah dua.

2.5.4 Penentuan Kondisi Permainan

Kondisi permainan ditentukan berdasarkan hasil dari subbab 2.4 dan 2.5 di mana

- Jika pada subbab 2.4 kedua raja tidak dalam kondisi sekak maka kondisi permainan adalah “Safe”.



Gambar 2.15: Intersepsi oleh Bidak Kawan

- Jika pada subbab 2.4 kedua raja dalam kondisi sekak maka kondisi permainan adalah “Impossible”.
- Jika pada subbab 2.4 salah satu raja dalam kondisi sekak dan pada subbab 2.5 jumlah langkah valid yang dapat dilakukan lebih besar dari nol maka kondisi permainan adalah “*warna* Check - *m* Plausible moves” di mana *warna* di isi dengan warna dari raja yang terkena sekak dan *m* di isi dengan jumlah langkah valid yang dapat dilakukan.
- Jika pada subbab 2.4 salah satu raja dalam kondisi sekak dan pada subbab 2.5 jumlah langkah valid yang dapat dilakukan sama dengan nol maka kondisi permainan adalah “*warna* Checkmate” di mana *warna* di isi dengan warna dari raja yang terkena sekak.

BAB 3

DESAIN ALGORITMA

Bab ini akan mendeskripsikan mengenai bagaimana algoritma untuk permasalahan SPOJ Klasik 32092 berjudul *Chessboard State 3: Intergalactic Chess Tournament* didesain untuk diselesaikan.

3.1 Desain Umum

Program akan diawali dengan menerima masukan berupa nilai *T* di mana *T* adalah banyaknya papan catur dalam suatu kasus uji. Untuk tiap papan catur, program akan menerima masukan berupa dua nilai *N* dan *P* di mana *N* adalah lebar papan (papan berukuran *NxN*) dan *P* adalah jumlah bidak pada papan tersebut. *P* baris berikutnya akan berisi tiga buah nilai *x y c* di mana *x* dan *y* adalah koordinat bidak pada papan catur dan *c* adalah tipe bidak. Setelah menerima masukan, masukan tersebut akan diolah dan hasilnya ditampilkan pada layar yaitu kondisi permainan berdasarkan konfigurasi dari bidak yang diberikan. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.1.

3.2 Desain Tipe Data Agregat Piece

Tipe data agregat ini adalah tipe data yang digunakan untuk merepresentasikan sebuah bidak. Tipe data agregat ini terdiri dari tiga variabel yaitu dua buah integer yang merupakan koordinat bidak pada papan catur dan sebuah karakter yang berisikan tipe dari bidak tersebut. Pseudocode dari desain tipe data agregat ini dapat dilihat pada Gambar 3.2.

```
integer t <- getInput()
for i <- 1 to t do
        integer board_size <- getInput()
        integer num_of_piece <- getInput()
        map char to vector of Piece piece_list <- null
        map pair of integer to char piece_coord <- null

        for j <- 1 to num_of_piece do
                integer x_pos <- getInput()
                integer y_pos <- getInput()
                char piece_type <- getInput()

                Piece p <- create_piece(x_pos, y_pos,
                    piece_type)
                piece_coord[(x_pos, y_pos)] <- piece_type
                piece_list[piece_type].push_back(p)
        end for

        solve(piece_list, piece_coord)

end for
```

Gambar 3.1: Pseudocode Fungsi Main

```
Piece():
        integer x_pos
        integer y_pos
        char piece_type
```

Gambar 3.2: Tipe Data Agregat *Piece*

```
1 Checker():
2         set of integer horizontal
3         set of integer vertical
4         set of integer main_diagonal
5         set of_integer anti_diagonal
```

Gambar 3.3: Desain Tipe Data *Agregat Checker*

3.3 Desain Tipe Data Agregat Checker

Tipe data agregat ini adalah tipe data yang digunakan untuk menyimpan kelompok bidak yang berada pada delapan arah dari suatu posisi. Tipe data ini merupakan bentuk simplifikasi dengan menggunakan bidak atau posisi tertentu sebagai pusat, dari struktur data *map-set* yang dibangun pada subbab 2.3.2 yang menyimpan informasi untuk keseluruhan papan permainan. Tipe data agregat ini terdiri dari empat buah struktur data *set* yaitu

1. Horizontal, yang berisikan bidak yang berada pada satu garis horizontal dengan posisi tertentu.

2. Vertikal, yang berisikan bidak yang berada pada satu garis vertikal dengan posisi tertentu.

3. Main diagonal, yang berisikan bidak yang berada pada satu garis main diagonal dengan posisi tertentu.

4. Anti diagonal, yang berisikan bidak yang berada pada satu garis anti diagonal dengan posisi tertentu.

Pseudocode dari desain tipe data agregat ini dapat dilihat pada Gambar 3.3.

3.4 Desain Tipe Data Agregat Check Count

Tipe data agregat ini merupakan tipe data yang digunakan untuk menampung jumlah bidak yang membuat raja berada dalam kondisi

```
1 Check_Count():
2         integer eight_dir
3         integer horse
4         integer pawn
5         integer total_checked
```

Gambar 3.4: Desain Tipe Data Agregat *Check Count*

sekak. Tipe data agregat ini terdiri dari empat integer yang masing-masing berisikan jumlah sekak oleh slider lawan, berisikan jumlah sekak oleh kuda lawan, berisikan jumlah sekak oleh prajurit lawan, dan gabungan ketiga jumlah sekak sebelumnya. Pseudocode dari desain tipe data agregat ini dapat dilihat pada Gambar 3.4.

3.5 Desain Fungsi Slider Checker

Fungsi ini adalah fungsi untuk menentukan apakah bidak terdekat dari suatu posisi yang menjadi acuan dapat berpindah ke posisi tersebut. Fungsi ini membutuhkan parameter bidak dengan posisi yang menjadi acuan, hasil pencarian *upper bound* atau *lower bound*, kelompok bidak untuk arah tertentu, *map* koordinat bidak ke tipe bidak, warna bidak yang akan dievaluasi, fase pengecekan, penunjuk nilai yang dicari adalah *upper bound* atau *lower bound*, nilai yang digunakan sebagai acuan, dan arah pencarian.

Fungsi ini akan dipanggil untuk tiap arah pada delapan arah dari posisi acuan. Untuk bidak terdekat pada tiap arah, fungsi ini akan menentukan apakah bidak terdekat tersebut dapat berpindah ke posisi acuan. Berdasarkan fase pengecekan yang sedang dilakukan fungsi ini melakukan hal tambahan yang berbeda, ketika fase pengecekan kondisi sekak raja fungsi ini akan melakukan penentuan bidak kawan mana saja yang memiliki status pinned dan apakah bidak terdekat termasuk bidak penyerang, ketika fase pencarian bidak kawan yang dapat bergerak ke posisi bidak penyerang fungsi ini akan melakukan pengecekan apakah bidak kawan yang digerakkan

memiliki status *pinned* atau tidak. Fungsi ini akan mengembalikan boolean yang menandakan raja di sekak atau tidak dari suatu arah. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.5

3.6 Desain Fungsi Horse Checker

Fungsi ini adalah fungsi untuk mencari jumlah kuda lawan yang dapat berpindah ke posisi tertentu. Fungsi ini membutuhkan parameter bidak dengan posisi yang akan menjadi acuan, *map* koordinat bidak ke tipe bidak, warna bidak yang akan dievaluasi, dan fase pengecekan.

Fungsi ini akan mencari untuk setiap petak yang dapat dituju dari posisi awal dengan menggunakan cara bergerak bidak kuda pada permainan catur, apakah pada petak tersebut terdapat kuda lawan. Selain itu berdasarkan fase pengecekan fungsi ini akan melakukan hal tambahan yang berbeda, ketika fase pengecekan kondisi raja fungsi ini akan menentukan bidak lawan mana yang merupakan bidak penyerang, ketika fase pencarian bidak kawan yang dapat bergerak ke posisi bidak penyerang fungsi ini akan melakukan pengecekan apakah bidak kawan yang digerakkan memiliki status pinned. Fungsi ini akan mengembalikan sebuah integer yang menunjukkan jumlah kuda lawan yang dapat berpindah ke posisi bidak yang dievaluasi. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.6

3.7 Desain Fungsi Pawn Checker

Fungsi ini adalah fungsi untuk mencari jumlah prajurit lawan yang dapat berpindah ke posisi tertentu. Fungsi ini membutuhkan parameter bidak dengan posisi yang akan menjadi acuan, *map* koordinat bidak ke tipe bidak, warna dari bidak yang akan dievaluasi, dan fase pengecekan.

Fungsi ini akan mencari untuk setiap petak yang dari petak tersebut

```
1  int enemy_color <- king_color^1
2  coord <- coord_function(king.x_pos, king.y_pos,
       searched_value, direction)
3  result <- checker_function(piece_coord[
       searched_piece_coord], king_color, direction)
4
5  if phase = INITIAL_CHECK_PHASE then
6          if result = friend piece then
7                  if move_dir = upper then
8                          searched_value++
9                  else
10                         searched_value--
11                 end if
12
13                 next_coord <- coord_function(king.x_pos,
                       king.y_pos, searched_value)
14                 if piece_coord[next_closest_coord] = enemy
                        slider then
15                         pinned_list[king_color].push_back(
                               searched_piece_coord)
16                 end if
17
18         else if result = enemy piece then
19                 attacker_piece <- create_piece(coord.x_pos
                       , coord.y_pos, piece_coord[coord])
20                 attacker_list[king_color].push_back(
                       attacker_piece)
21         end if
22
23 else if result = enemy piece and phase = EATING_PHASE and
       is_pinned(coord) then
24         result <- 0
25 end if
26
27 return (result = 1)
```

Gambar 3.5: Pseudocode Fungsi Slider Checker

```
1  integer checked_count <- 0
2  integer enemy_color <- king_color^1
3  char enemy_horse = HORSES[enemy_color]
4
5  for all move in HORSE_MOVE do
6          pair of integer pos <- (king.x_pos + move.first,
               king.y_pos + move.second)
7          if piece_coord[pos] = enemy_horse then
8
9                  if phase = EATING_PHASE and is_pinned(pos,
                       enemy_color) = true then
10                         continue
11                 end if
12
13                 checked_count <- checked_count + 1
14
15                 if phase = INITIAL_CHECKING_PHASE then
16                         Piece attacker <- create_piece(pos
                               .first, pos.second,
                               other_piece[pos])
17                         attacker_list[king_color].
                               push_back(attacker)
18                 end if
19         end if
20 end for
21
22 return checked_count
```

Gambar 3.6: Pseudocode Fungsi Horse Check

prajurit lawan dapat memakan bidak yang dievaluasi, apakah pada petak tersebut terdapat prajurit lawan. Selain itu berdasarkan fase pengecekan fungsi ini akan melakukan hal tambahan yang berbeda, ketika fase pengecekan kondisi raja fungsi ini akan menentukan bidak lawan mana yang merupakan bidak penyerang, ketika fase pencarian bidak kawan yang dapat bergerak ke posisi bidak penyerang fungsi ini akan melakukan pengecekan apakah bidak kawan yang digerakkan memiliki status pinned. Fungsi ini akan mengembalikan sebuah integer yang menunjukkan jumlah prajurit lawan yang dapat berpindah ke posisi bidak yang dievaluasi. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.7.

3.8 Desain Fungsi King Checker

Fungsi ini adalah fungsi untuk mencari jumlah raja lawan yang dapat berpindah ke posisi tertentu. Fungsi ini membutuhkan parameter bidak dengan posisi yang akan dievaluasi, *map* koordinat bidak ke tipe bidak, dan warna bidak yang dievaluasi.

Fungsi ini akan mencari untuk setiap petak yang dapat dituju melalui pergerakan raja, apakah pada petak tersebut terdapat raja lawan. Fungsi ini akan mengembalikan sebuah integer yang menunjukkan jumlah raja lawan yang dapat berpindah ke posisi bidak yang dievaluasi. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.8.

3.9 Desain Fungsi Penghitungan Jumlah Sekak

Fungsi ini adalah fungsi untuk mencari jumlah bidak yang dapat berpindah ke posisi tertentu. Fungsi ini membutuhkan parameter bidak dengan posisi yang akan dievaluasi, *map* koordinat bidak ke tipe bidak, hasil dari pengecekan ke delapan arah dari bidak yang akan dievaluasi, warna dari bidak yang akan dievaluasi, dan fase pengecekan.

```
integer checked_count <- 0
integer enemy_color <- king_color^1
char enemy_pawn = PAWNS[enemy_color]

for all move in PAWN_MOVE[enemy_color] do
        pair of integer pos <- (king.x_pos + move.first,
            king.y_pos + move.second)

        if piece_coord[pos] = enemy_pawn then

                if phase = EATING_PHASE and is_pinned(pos,
                    enemy_color) = true then
                        continue
                end if

                checked_count <- checked_count + 1

                if phase = INITIAL_CHECKING_PHASE then
                        Piece attacker <- create_piece(pos
                            .first, pos.second,
                            other_piece[pos])
                        attacker_list[king_color].
                            push_back(attacker)
                end if
        end if
end for

return checked_count
```

Gambar 3.7: Pseudocode Fungsi Pawn Checker

```
integer checked_count <- 0
integer enemy_color <- king_color^1
char enemy_king = KINGS[enemy_color]
for all move in KING_MOVE do
        pair of integer pos <- (king.x_pos + move.first,
            king.y_pos + move.second)
        if piece_coord[pos] = enemy_king then
                checked_count <- checked_count + 1
        end if
end for

return checked_count
```

Gambar 3.8: Pseudocode Fungsi King Checker

Fungsi ini akan menentukan jumlah bidak yang dapat berpindah ke posisi bidak yang dievaluasi berdasarkan parameter hasil dari pengecekan ke delapan arah, dan memanggil fungsi pencarian kuda lawan, prajurit lawan, dan raja lawan yang dapat berpindah ke posisi bidak yang dievaluasi, kecuali untuk fase pencarian bidak kawan yang dapat bergerak ke posisi bidak penyerang pencarian untuk raja lawan tidak perlu dilakukan. Fungsi ini akan mengembalikan tipe data agregat Checker Count yang berisikan hasil yang didapat dari pemanggilan fungsi di atas. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.9.

3.10 Desain Fungsi Menggerakkan Bidak Raja

Fungsi ini adalah fungsi untuk mencari jumlah langkah valid yang didapat dari menggerakkan raja ke posisi baru. Fungsi ini membutuhkan parameter bidak raja, *map-set* dari keseluruhan bidak, *map* koordinat bidak ke tipe bidak, hasil dari pengecekan ke delapan arah dari raja, dan warna bidak raja.

Fungsi ini akan mencari untuk setiap posisi baru hasil pergerakan raja, apakah pada posisi tersebut raja berada dalam kondisi sekak. Fungsi ini akan mengembalikan integer yang menunjukkan jumlah petak posisi baru dimana raja tidak lagi dalam kondisi sekak atau dengan kata lain jumlah langkah valid yang dapat diambil dari menggerakkan raja. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.10.

3.11 Desain Fungsi Memakan Bidak Penyerang

Fungsi ini adalah fungsi untuk mencari jumlah langkah valid yang dapat dilakukan dengan menggunakan bidak kawan untuk memakan (bergerak ke petak) bidak penyerang. Fungsi ini membutuhkan parameter bidak penyerang yang akan dievaluasi, *map-set* yang merepresentasikan bidak pada papan secara keseluruhan, *map* koordinat bidak ke tipe bidak, dan warna dari bidak penyerang.

```
1  integer enemy_color <- king_color^1
2  char enemy_horse <- HORSES[enemy_color]
3  char enemy_pawn <- PAWNS[enemy_color]
4  char enemy_king <- KINGS[enemy_color]
5
6  integer slider_count <- 0
7  for i <- 0 to 7 do
8          if slider_check_result[i] = true
9                  slider_count <- slider_count + 1
10         end if
11 end for
12 integer horse_count <- horse_checker(king, piece_coord,
       king_color, phase)
13 integer pawn_count <- pawn_checker(king, piece_coord,
       king_color, phase)
14 integer king_count <- 0
15 if phase != EATING_PHASE
16         king_count <- king_checker(king, piece_coord,
               king_color)
17 end if
18 integer total_count = slider_count + horse_count +
       pawn_count + king_count
19
20 Check_Count checker
21 checker.eight_dir <- slider_count
22 checker.horse <- horse_count
23 checker.pawn <- pawn_count
24 checker.total_checked <- total_count
25
26 return checker
```

Gambar 3.9: Pseudocode Fungsi Pencarian Jumlah Sekak

```
1  integer king_possible_move <- 0
2  integer enemy_color <- king_color^1
3  char enemy_horse <- HORSES[enemy_color]
4  char enemy_pawn <- PAWNS[enemy_color]
5  char enemy_king <- KINGS[enemy_color]
6
7  for all move in MOVE_8_DIR do
8          Piece new_king <- create_piece(king.x_pos +
               KING_MOVE[move].first, king.y_pos + KING_MOVE[
               move].second, king.piece_type)
9          pair of integer new_king_coord <- (new_king.x_pos,
                new_king.y_pos)
10
11         if out_of_bound(new_king_coord) = true or is_enemy
               (piece_coord[new_king_coord], INVERSE_COLORS[
               king_color]) = false then
12             continue
13         end if
14
15         Checker king_checker <- new_king_checker(king,
               new_king)
16         piece_coord[new_king_coord] <- king.piece_type
17
18         vector of boolean check_result <-
               get_slider_result(new_king, king_checker,
               piece_coord, king_color, KING_MOVING_PHASE)
19         Check_Count king_checked_count <- get_check_count(
               new_king, piece_coord, check_result,
               king_color, KING_MOVING_PHASE)
20
21         if king_checked_count.total_checked = 0 then
22                 king_possible_move <- king_possible_move +
                       1
23         end if
24
25 end for
26
27 return king_possible_move
```

Gambar 3.10: Pseudocode Fungsi Menggerakkan Bidak Raja

```
integer enemy_color <- attacker_color^1
char enemy_horse <- HORSES[enemy_color]
char enemy_pawn <- PAWNS[enemy_color]
char enemy_king <- KINGS[enemy_color]

Checker attacker_checker <- create_checker(attacker)

vector of boolean check_result <- get_slider_result(
    attacker, attacker_checker, piece_coord,
    attacker_color, EATING_PHASE)
Check_Count attacker_checked_count <- get_check_count(
    attacker, piece_coord, check_result, attacker_color,
    EATING_PHASE)

return attacker_checked_count.total_checked
```

Gambar 3.11: Psudocode Fungsi Memakan Bidak Penyerang

Fungsi ini dalam prosesnya akan membuat tipe data agregat Checker yang bersesuaian untuk bidak penyerang dan kemudian mengaplikasikan fungsi untuk menentukan kondisi raja. Di mana hasil dari fungsi tersebut adalah tipe data agregat Check Count yang berisikan jumlah bidak yang dapat berpindah ke posisi yang diberikan sebagai parameter fungsi atau pada kasus ini berarti jumlah bidak yang dapat memakan bidak penyerang. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.11.

3.12 Desain Fungsi Intersepsi Slider

Fungsi ini adalah fungsi untuk mencari jumlah langkah yang dapat dilakukan dengan menggerakkan bidak slider teman sehingga menutupi jalur bidak penyerang menuju raja. Fungsi ini membutuhkan parameter bidak raja, bidak penyerang, *map-set* yang berisi informasi bidak pada papan secara keseluruhan, *map* posisi bidak ke tipe bidak, arah dari mana bidak penyerang datang relatif terhadap bidak raja, dan warna bidak raja.

```
integer intercept_number <- 0
for all direction in SLIDER_DIRECTIONS do
        if direction = checked_from then
                continue
        end if
        for all iterator in piece_set[direction] do
                integer key <- iterator.first
                if in_between(king, attacker, direction,
                    key, checked_from) = true then
                        integer intersect <-
                            get_intersection(king, key,
                            direction, checked_from)
                        intercept_number <-
                            intercept_number +
                            intercept_processor(intersect,
                             direction, key, piece_set[
                            direction][key], piece_coord,
                            king_color)
                end if
        end for
end for

return intercept_number
```

Gambar 3.12: Desain Fungsi Intersepsi Slider

Fungsi ini akan melakukan untuk setiap sudut pandang, perulangan terhadap semua *key value* dari struktur data *map-set* yang telah dibangun, jika *key value* tersebut berada di antara bidak raja dan bidak penyerang maka dilakukan pencarian bidak terdekat dari posisi yang berada pada jalur antara bidak raja dan bidak penyerang. Untuk setiap bidak terdekat yang ditemukan yang merupakan bidak teman yang tegolong *slider* dan tidak memiliki status *pinned*, jumlah langkah valid bertambah satu. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.12.

3.13 Desain Fungsi Intersepsi Kuda

Fungsi ini adalah fungsi untuk mencari jumlah langkah yang dapat dilakukan dengan menggerakkan bidak kuda teman sehingga menu-

tupi jalur bidak penyerang menuju raja. Fungsi ini membutuhkan parameter bidak raja, bidak penyerang, *map* tipe bidak ke kumpulan koordinat bidak, *map* koordinat bidak ke tipe bidak, warna bidak raja, dan arah dari mana bidak penyerang datang relatif terhadap bidak raja.

Fungsi ini akan melakukan untuk setiap bidak kuda teman, menggerakkan bidak kuda tersebut ke segala kemungkinan pergerakan bidak kuda. Untuk setiap posisi akhir pergerakkan yang berada pada jalur antara bidak raja dan bidak penyerang, dan bidak kuda yang digunakan tidak memiliki status *pinned* maka jumlah langkah valid bertambah satu. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.13.

3.14 Desain Fungsi Intersepsi Prajurit

Fungsi ini adalah fungsi untuk mencari jumlah langkah yang dapat dilakukan dengan menggerakkan bidak prajurit teman sehingga menutupi jalur bidak penyerang menuju raja. Fungsi ini membutuhkan parameter bidak raja, bidak penyerang, *map* tipe bidak ke kumpulan koordinat bidak, *map* koordinat bidak ke tipe bidak, warna bidak raja, dan arah dari mana bidak penyerang datang relatif terhadap bidak raja.

Fungsi melakukan untuk setiap bidak prajurit teman, menggerakkan bidak prajurit tersebut ke segala kemungkinan pergerakan bidak prajurit. Untuk setiap posisi akhir pergerakan yang berada pada jalur antara bidak raja dan bidak penyerang, dan bidak prajurit yang digunakan tidak memiliki status *pinned* maka jumlah langkah valid bertambah satu. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.14.

```
1  integer intercept_number <- 0
2  char ally_horse = HORSES[king_color]
3  integer enemy_color = king_color^1
4  for all horse in piece_list[ally_horse] do
5          pair of integer horse_coord <- (horse.x_pos, horse
               .y_pos)
6          if is_pinned(horse_coord, king_color) = false then
7                  for all move in HORSE_MOVE do
8                          pair of integer new_coord <- (
                               horse.x_pos + move.first,
                               horse.y_pos + move.second)
9                          if is_enemy(piece_coord[new_coord
                               ], INVERSE_COLORS[king_color])
                                = false then
10                                 continue
11                         end if
12                         if out_of_bound(new_coord) = false
                                and in_line(king, new_coord,
                               checked_from) = true and
                               in_middle(king, attacker,
                               new_coord, checked_from) =
                               true then
13                                 intercept_number <-
                                       intercept_number + 1
14                         end if
15                 end for
16         end if
17 end for
18
19 return intercept_number
```

Gambar 3.13: Desain Fungsi Intersepsi Kuda

```
integer intercept_number <- 0
char ally_pawn <- PAWNS[king_color]
int enemy_color <- king_color^1
pair of integer white_move <- (0, 1)
pair of integer black_move <- (0, -1)
vector of pair of integer pawn_moves <- [white_move,
    black_move]

for all pawn in piece_list[ally_pawn] do
        pair of integer pawn_coord <- (pawn.x_pos, pawn.
            y_pos)
        if is_pinned(pawn_coord, king_color) = false then
                pair of integer move = pawn_moves[
                    king_color]
                pair of integer new_coord <- (pawn.x_pos +
                     move.first, pawn.y_pos + move.second)
                if is_enemy(piece_coord[new_coord],
                    INVERSE_COLORS[king_color]) = false
                    then
                        continue
                end if
                if out_of_bound(new_coord) = false and
                    in_line(king, new_coord, checked_from)
                     = true and in_middle(king, attacker,
                    new_coord, checked_from) = true then
                        intercept_number <-
                            intercept_number + 1
                end if
        end if
end for

return intercept_number
```

Gambar 3.14: Desain Fungsi Intersepsi Prajurit

3.15 Desain Fungsi Solve

Fungsi ini adalah fungsi untuk melakukan penentuan kondisi permainan berdasarkan konfigurasi bidak pada papan catur yang diberikan. Fungsi ini membutuhkan parameter *map* tipe bidak ke kumpulan bidak dan *map* koordinat bidak ke tipe bidak.

Fungsi akan menentukan kondisi permainan berdasarkan hasil pencarian kondisi sekak dan pencarian jumlah langkah valid yang dapat dilakukan bila salah satu bidak raja berada dalam kondisi sekak. Fungsi ini kemudian akan menampilkan pada layar *console* kondisi dari permainan sesuai yang didefinisikan pada subbab 2.5.4. Pseudocode dari desain fungsi ini dapat dilihat pada Gambar 3.15.

```
1  Piece black_king <- pieces_list[BLACK_KING][0]
2  Piece white_king <- pieces_list[WHITE_KING][0]
3  Checker b_checker <- create_checker(black_king,
       pieces_list)
4  Checker w_checker <- create_checker(white_king,
       pieces_list)
5
6  vector of boolean b_result <- get_slider_result(black_king
       , b_checker, piece_coord, BLACK, INITIAL_CHECK_PHASE)
7  vector of boolean w_result <- get_slider_result(white_king
       , w_checker, piece_coord, WHITE, INITIAL_CHECK_PHASE)
8  Check_Count b_count <- get_check_count(black_king,
       piece_coord, b_result, BLACK, INITIAL_CHECK_PHASE)
9  Check_Count w_count <- get_check_count(white_king,
       piece_coord, w_result, INITIAL_CHECK_PHASE)
10
11 boolean black_checked = black_checked_count.total_checked
       != 0
12 boolean white_checked = white_checked_count.total_checked
       != 0
13
14 if black_checked = true and white_checked = true then
15         print("Impossible")
16 else if black_checked = false and !white_checked = false
       then
17         print("Safe")
18 else if black_checked = true then
19         integer possible_move <- only_one_check(black_king
               , piece_coord, pieces_list, b_result, b_count,
                BLACK)
20         if possible_move = 0 then
21                 print("Black Checkmate")
22         else
23                 print("Black Check - $s Plausible Moves",
                       possible_move)
24         end if
25 else if white_checked = true then
26         llu possible_move <- only_one_check(white_king,
               piece_coord, pieces_list, w_result, wh_count,
               WHITE)
27         if possible_move = 0)
28                 print("White Checkmate")
29         else
30                 print("White Check - $s Plausible Moves",
                       possible_move)
31         end if
32 end if
```

Gambar 3.15: Desain Fungsi Solve

[Halaman ini sengaja dikosongkan]

BAB 4

IMPLEMENTASI ALGORITMA

Bab ini akan menjabarkan mengenai bagaimana permasalahan SPOJ Klasik 32092 *Chessboard State 3: Intergalactic Chess Tournament* diimplementasikan dengan menggunakan desain fungsi yang telah dijelaskan pada bab sebelumnya.

4.1 Lingkungan Implementasi

Implementasi yang dilakukan dalam Tugas Akhir ini memiliki dua komponen penting, yakni perangkat keras dan perangkat lunak. Berikut ini merupakan spesifikasi daripada perangkat keras maupun perangkat lunak yang digunakan untuk implementasi Tugas Akhir.

1. Perangkat Keras
    - Processor: Intel Core i7 8^{th} Gen
    - Memori: 16 GB
2. Perangkat Lunak
    - Sistem Operasi: Windows 10 Home
    - IDE : *Dev-C++*
    - Compiler: gcc 7.3 C++14

4.2 Rancangan Data

Subbab ini akan mendeskripsikan bagaimana format data masukan serta format data keluaran yang terdapat pada setiap permasalahan yang ada dalam Tugas Akhir ini.

4.2.1 Data Masukan

Data masukan dalam Tugas Akhir ini didefinisikan sebagai data-data yang akan digunakan oleh program sebagai masukan untuk diselesaikan oleh algoritma yang ada.

Data pada permasalahan CSTATE3 diawali dengan sebuah bilangan bulat *T*. Untuk setiap *T* diberikan dua buah bilangan bulat *N* dan *P*. Untuk *P* baris berikutnya berisikan 2 buah bilangan bulat *x* dan *y*, dan sebuah karakter *c* yang merupakan data bidak pada papan permainan berukuran *NxN* yang akan diolah.

4.2.2 Data Keluaran

Keluaran data yang dihasilkan oleh program adalah berupa *T* buah *string* dimana tiap *string* merupakan kondisi permainan berdasarkan konfigurasi bidak pada papan catur yang diberikan.

4.3 Implementasi Algoritma

Bagian ini akan menjabarkan mengenai implementasi daripada fungsi-fungsi yang telah didesain dari bab sebelumnya.

4.3.1 Header yang diperlukan

Implementasi algoritma penentuan kondisi permainan catur berdasarkan konfigurasi bidak pada papan catur untuk menyelesaikan permasalahan SPOJ Klasik 32092 *Chessboard State 3: Intergalactic Chess Tournament* membutuhkan *header* yaitu cstdio, map, vector, set, algorithm, utility, dan unordered map seperti yang terlihat pada Kode Sumber 4.1.

Header cstdio berisikan modul untuk menerima masukan dan memberi keluaran. *Header* map berisikan struktur data yang digunakan untuk memetakan koordinat dari suatu bidak ke tipe bidak. *Header*

vector digunakan untuk menyimpan cara bergerak bidak kuda, prajurit, dan raja. *Header* set berisikan struktur data yang digunakan untuk menyimpan informasi bidak relatif terhadap sudut pandang tertentu. *Header* utility berisikan tipe data pair yang digunakan untuk merepresentasikan koordinat dari bidak. *Header* unordered map berisikan struktur data yang digunakan untuk memetakan identitas kumpulan bidak yang berada pada satu garis ke kumpulan informasi bidak yang bersesuaian dan memetakan tipe bidak ke kumpulan bidak.

```
#include <cstdio>
#include <map>
#include <vector>
#include <set>
#include <utility>
#include <unordered_map>
```

Kode Sumber 4.1: Header yang diperlukan

4.3.2 Pre-Processor

Pre-Processor mendeskripsikan mengenai tipe-tipe data (*typedef*) yang didefinisikan ulang dengan tujuan untuk mempermudah penulisan kode sumber yang terdapat dalam tugas akhir ini. Pada bagian ini *llu* mendefinisikan tipe data *long long int*, *pii* mendefiniskan tipe data *pair of integer and integer*, *pll* mendefinisikan tipe data *pair of llu and llu*, *map_tree* yang mendefinisikan tipe data *unordered map of llu to set of llu*, map_list yang mendefinisikan tipe data unordered map of char to vector of Piece, dan map_coord yang mendefinisikan tipe data *map of pll to char*. *Piece* pada tipe data map_list merupakan tipe data agregat yang dijelaskan pada subbab 3.2. *map_tree, map_list,* dan *map_coord* merupakan struktur data yang telah dibahas pada subbab 2.3.2 dan subbab 2.3.3. Kode sumber untuk bagian pre-processor dapat dilihat pada Kode Sumber 4.2.

```
#define llu long long int
#define pii pair<int, int>
#define pll pair<llu, llu>
#define map_tree unordered_map<llu, set<llu>>
#define map_list unordered_map<char, vector<Piece>>
#define map_coord map<pll, char>
```

Kode Sumber 4.2: Pre-Processor

4.3.3 Tipe Data Agregat

Bagian ini mengimplementasikan tipe data agregat yang digunakan untuk mempermudah penulisan kode sumber dengan mengelompokkan beberapa informasi menjadi sebuah tipe data agregat. Terdapat tiga tipe data agregat yang digunakan yaitu

1. Tipe data agregat Piece (subbab 3.2)
2. Tipe data agregat Checker (subbab 3.3)
3. Tipe data agregat Check_Count (subbab 3.4)

Dengan kode sumber untuk masing-masing tipe data agregat dapat dilihat pada Kode Sumber 4.3, Kode Sumber 4.4, dan Kode Sumber 4.5.

```
typedef struct piece
{
        llu x_pos;
        llu y_pos;
        char piece_type;
} Piece;
```

Kode Sumber 4.3: Tipe Data Agregat Piece

```
typedef struct checker
{
        set<llu> horizontal;
        set<llu> vertical;
        set<llu> main_diagonal;
        set<llu> anti_diagonal;
} Checker;
```

Kode Sumber 4.4: Tipe Data Agregat Checker

```
typedef struct check_count
{
        int eight_dir;
        int horse;
        int pawn;
        int total_checked;
} Check_Count;
```

Kode Sumber 4.5: Tipe Data Agregat Checker_Count

4.3.4 Konstanta

Bagian ini mengimplementasikan mengenai konstanta-konstanta yang digunakan dalam kode sumber. Pendefinisian konstanta disini bertujuan mempermudah pembacaan kode sumber dan mengurangi kesalahan dalam pembuatan kode sumber. Beberapa konstanta yang paling penting adalah *WHITE_RANGE* dan *BLACK_RANGE* yang masing-masing berisikan *pair of integer and integer* yang merupakan batas atas dan batas bawah nilai ascii dari tipe bidak putih dan tipe bidak hitam. *HORSE_MOVE* dan *KING_MOVE* yang masing-masing merupakan *vector* berisikan cara bergerak bidak kuda dan bidak raja dalam permainan catur dalam bentuk *pair of integer and integer*. *BLACK_PAWN_MOVE* dan *WHITE_PAWN_MOVE* yang berisikan cara makan dari bidak prajurit hitam dan bidak prajurit putih dalam permainan catur dalam bentuk *pair of integer and integer*. Kode Sumber dari daftar konstanta yang digunakan dapat dilihat pada Kode Sumber 4.6 dan 4.7

```
const int BLACK = 1;
const int WHITE = 0;
const char BLACK_KING = 'K';
const char WHITE_KING = 'k';
const char BLACK_QUEEN = 'Q';
const char WHITE_QUEEN = 'q';
const char BLACK_BISHOP = 'B';
const char WHITE_BISHOP = 'b';
const char BLACK_ROOK = 'R';
const char WHITE_ROOK = 'r';
const char BLACK_HORSE = 'H';
const char WHITE_HORSE = 'h';
const char BLACK_PAWN = 'P';
const char WHITE_PAWN = 'p';
const vector<char> HORSES = {WHITE_HORSE, BLACK_HORSE};
const vector<char> PAWNS = {WHITE_PAWN, BLACK_PAWN};
const vector<char> KINGS = {WHITE_KING, BLACK_KING};
const vector<char> BLACK_TYPE = {BLACK_KING, BLACK_QUEEN,
    BLACK_HORSE, BLACK_BISHOP, BLACK_ROOK, BLACK_PAWN};
const vector<char> WHITE_TYPE = {WHITE_KING, WHITE_QUEEN,
    WHITE_HORSE, WHITE_BISHOP, WHITE_ROOK, WHITE_PAWN};
const vector<vector<char>> type_list = {WHITE_TYPE,
    BLACK_TYPE};
const pii WHITE_RANGE = pii(97, 122);
const pii BLACK_RANGE = pii(65, 90);
const vector<pii> INVERSE_COLORS = {BLACK_RANGE, WHITE_RANGE
    };
const vector<char> HORIZONTAL_VERTICAL_PIECE = {'Q', 'R', 'q'
    , 'r'};
const vector<char> DIAGONAL_PIECE = {'Q', 'B', 'q', 'b'};
const int NORTH = 0;
const int NORTH_EAST = 1;
const int EAST = 2;
const int SOUTH_EAST = 3;
const int SOUTH = 4;
const int SOUTH_WEST = 5;
const int WEST = 6;
const int NORTH_WEST = 7;
```

Kode Sumber 4.6: Definisi Konstanta

```
const vector<pii> HORSE_MOVE = {pii(-2, -1), pii(-2, 1), pii
    (1, -2), pii(-1, -2), pii(2, -1), pii(2, 1), pii(1, 2),
    pii(-1, 2)};
const vector<pii> WHITE_PAWN_MOVE = {pii(1, 1), pii(-1, 1)};
const vector<pii> BLACK_PAWN_MOVE = {pii(1, -1), pii(-1, -1)
    };
const vector<vector<pii>> PAWN_MOVE = {BLACK_PAWN_MOVE,
    WHITE_PAWN_MOVE};
const vector<pii> KING_MOVE = {pii(0, -1), pii(1, -1), pii(1,
     0), pii(1, 1), pii(0, 1), pii(-1, 1), pii(-1, 0), pii
    (-1, -1)};
const vector<int> MOVE_8_DIR = {NORTH, NORTH_EAST, EAST,
    SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST};
const int HORIZONTAL = 2;
const int VERTICAL = 0;
const int MAIN_DIAGONAL = 3;
const int ANTI_DIAGONAL = 1;
const vector<int> SLIDER_DIRECTIONS = {HORIZONTAL,
    ANTI_DIAGONAL, VERTICAL, MAIN_DIAGONAL};
const int INITIAL_CHECK_PHASE = 0;
const int KING_MOVING_PHASE = 1;
const int EATING_PHASE = 2;
```

Kode Sumber 4.7: Definisi Konstanta Lanjutan

4.3.5 Variabel Global

Bagian ini mengimplementasikan mengenai variabel global apa saja yang digunakan untuk menyelesaikan permasalahan *CSTATE3*. Implementasi daripada deklarasi variabel global dapat dilihat pada kode sumber 4.8. Penjelasan variabel dapat dilihat pada Tabel 4.1

```
llu board_size, num_of_piece;
vector<pll> black_pinned_list, white_pinned_list;
vector<Piece> black_attacker_list, white_attacker_list;
vector<vector<pll>> pinned_list = {white_pinned_list,
    black_pinned_list};
vector<vector<Piece>> attacker_list = {white_attacker_list,
    black_attacker_list};
```

Kode Sumber 4.8: Variabel Global

Tabel 4.1: Penjelasan Variabel Global

No	Variabel	Tipe Data	Penjelasan
1	board_size	llu	Besar papan yang didapat melalui masukan.
2	num_of_piece	llu	Jumlah bidak pada papan permainan yang didapat melalui masukan.
3	black_pinned_list	vector< pll >	Struktur data yang menampung bidak hitam dengan kondisi *pinned*.
4	white_pinned_list	vector< pll >	Struktur data yang menampung bidak putih dengan kondisi *pinned*.
5	white_attacker_list	vector< Piece >	Struktur data yang menampung bidak hitam yang menyebabkan raja putih berada dalam kondisi sekak.
6	black_attacker_list	vector< Piece >	Struktur data yang menampung bidak putih yang menyebabkan raja hitam berada dalam kondisi sekak.
7	pinned_list	vector< vector < pll > >	Wrapper untuk variabel *white_pinned_list* dan *black_pinned_list*.
8	attacker_list	vector< vector < pll > >	Wrapper untuk variabel *white_attacker_list* dan *black_attacker_list*.

4.3.6 Implementasi Fungsi Main

Fungsi *main* adalah algoritma yang yang dirancang pada subbab 3.1. Implementasi dapat dilihat pada Kode Sumber 4.9. Penjelasan

variabel dapat dilihat pada Tabel 4.2.

```
int main()
{
        int t;
        scanf("%d",&t);
        for (int i = 0; i < t; i++)
        {
                scanf("%lld %lld", &board_size,&num_of_piece)
                    ;
                map_list pieces_list;
                map_coord piece_coord;

                pinned_list[BLACK].clear();
                pinned_list[WHITE].clear();
                attacker_list[BLACK].clear();
                attacker_list[WHITE].clear();

                for (int j = 0; j < num_of_piece; j++)
                {
                        llu x_pos, y_pos;
                        char piece_type;

                        scanf("%lld %lld %c", &x_pos, &y_pos,
                             &piece_type);
                        getchar();
                        Piece other_piece = create_piece(
                            x_pos, y_pos, piece_type);
                        piece_coord[pll(x_pos, y_pos)] =
                            piece_type;
                        pieces_list[piece_type].push_back(
                            other_piece);
                }
                solve(pieces_list, piece_coord);
        }
}
```

Kode Sumber 4.9: Implementasi Fungsi Main

Tabel 4.2: Penjelasan Variabel Pada Fungsi Main

No	Variabel	Tipe Data	Penjelasan
1	t	int	Jumlah papan dalam satu kasus uji.
2	pieces_list	map_list	Struktur data yang berisikan pemetaan tipe bidak ke kumpulan bidak yang bersesuaian.
3	piece_coord	map_coord	Struktur data yang berisikan pemetaan koordinat bidak ke tipe bidak.

4.3.7 Implementasi Fungsi Slider Checker

Fungsi slider_checker merupakan implementasi dari desain yang dibuat pada subbab 3.5 yang digunakan untuk menentukan apakah raja di sekak dari arah tertentu. Kode sumber untuk fungsi ini dapat dilihat pada Kode Sumber 4.10 dan 4.11 dengan penjelasan variabel pada Tabel 4.3.

Fungsi *slider_checker* ini akan dipanggil oleh fungsi *get_slider_result* untuk kedelapan arah dari posisi bidak raja dan hasil dari kedelapan arah tersebut akan dikembalikan dalam bentuk *vector of boolean*. Kode sumber untuk fungsi *get_slider_result* dapat dilihat pada Kode Sumber 4.12 dan 4.13 dengan penjelasan variabel pada Tabel 4.4.

Pada Kode Sumber 4.10 dapat dilihat bahwa fungsi *slider_checker* membutuhkan beberapa fungsi utilitas untuk mendukung proses yang dilakukan. Kode sumber untuk fungsi-fungsi utilitas ini dapat dilihat pada Kode Sumber 4.14, 4.15, 4.16, 4.17, 4.18, dan 4.19.

```
bool slider_checker(Piece king, set<llu>::iterator &parent_it
    , set<llu> &dir_checker, map_coord &other_coord, int
    king_color, int phase, int dir_to_move, llu search_val,
    int direction)
{
        set<llu>::iterator it = parent_it;
        int enemy_color = king_color^1;
        pll coord;
        int res;

        if (dir_checker.size() == 1 || *it == search_val)
                return 0;
        if (it != dir_checker.end() && *it != search_val) {
                coord = slider_checker_coord(king.x_pos, king
                    .y_pos, *it, direction);
                res = (other_coord.find(coord) == other_coord
                    .end()) ? 0 : viewpoint_checker(
                    other_coord[coord], king_color,
                    direction);
        }
        else res = 0;

        if (phase == INITIAL_CHECK_PHASE)
        {
                if (res == 2)
                {
                        if (dir_to_move) it++;
                        else it--;
                        if (it != dir_checker.end() && *it !=
                             search_val) {
                                pll new_coord =
                                    slider_checker_coord(
                                    king.x_pos, king.y_pos,
                                    *it, direction);
...
```

Kode Sumber 4.10: Implementasi Fungsi *slider_checker*

```
...
                                int pinned = (other_coord.
                                    find(new_coord) ==
                                    other_coord.end()) ? 0 :
                                     viewpoint_checker(
                                    other_coord[new_coord],
                                    king_color, direction);
                                if (pinned == 1)
                                pinned_list[king_color].
                                    push_back(coord);
                        }
                }
                else if (res == 1)
                {
                        Piece attacker = create_piece(coord.
                            first, coord.second, other_coord
                            [coord]);
                        attacker_list[king_color].push_back(
                            attacker);
                }

        }
        else if (phase == EATING_PHASE && res == 1 &&
            is_pinned(coord, enemy_color))
        {
                res = 0;
        }

        return (res == 1);
}
```

Kode Sumber 4.11: Lanjutan Implementasi Fungsi *slider_checker*

Tabel 4.3: Penjelasan Variabel Pada Fungsi *slider_checker*

No	Variabel	Tipe Data	Penjelasan
1	it	set< llu >::iterator	Iterator yang menunjukkan posisi dari hasil pencarian *lower bound* atau *upper bound* pada set.
2	enemy_color	int	Warna dari bidak yang berlawanan dengan bidak raja yang sedang dievaluasi.
3	coord	pll	Koordinat dari bidak terdekat hasil pencarian *lower bound* atau *upper bound* pada arah tertentu.
4	res	int	Hasil dari proses pengecekan apakah bidak hasil pencarian menyebabkan sekak pada bidak raja yang dievaluasi.
5	new_coord	pll	Koordinat dari bidak kedua terdekat hasil pencarian *lower bound* atau *upper bound* pada arah tertentu.
6	pinned	bool	Penanda apakah bidak terdekat memiliki status *pinned*.
7	attacker	Piece	Bidak penyerang penyebab sekak pada bidak raja yang dievaluasi.

```
vector<bool> get_slider_result(Piece king, Checker &
    king_checker, map_coord &other_coord, int king_color,
    int phase)
{
        vector<bool> check_result(8, false);
        set<llu>::iterator right;
        set<llu>::iterator left;
        int enemy_color = king_color^1;

        vector< set<llu> *> viewpoint = {&king_checker.
            horizontal, &king_checker.vertical, &
            king_checker.main_diagonal, &king_checker.
            anti_diagonal};
        vector< llu > view_search_val = {king.x_pos, king.
            y_pos, king.x_pos, king.x_pos};
        vector< int > view_upper = {EAST, SOUTH, SOUTH_EAST,
            NORTH_EAST};
        vector< int > view_lower = {WEST, NORTH, NORTH_WEST,
            SOUTH_WEST};
        vector< int > directions = {HORIZONTAL, VERTICAL,
            MAIN_DIAGONAL, ANTI_DIAGONAL};

        int i = 0;
        for (set<llu>* view : viewpoint)
        {
                if (!view->empty())
                {
                        right = view->upper_bound(
                            view_search_val[i]);
                        check_result[view_upper[i]] =
                            slider_checker(king, right, *
                            view, other_coord, king_color,
                            phase, 1, view_search_val[i],
                            directions[i]);
...
```

Kode Sumber 4.12: Implementasi fungsi *get_slider_result*

```
...
                        left = view->lower_bound(
                            view_search_val[i]);
                        left--;
                        check_result[view_lower[i]] =
                            slider_checker(king, left, *view
                            , other_coord, king_color, phase
                            , 0, view_search_val[i],
                            directions[i]);
                }
                i++;
        }

        return check_result;
}
```

Kode Sumber 4.13: Lanjutan Implementasi Fungsi *get_slider_result*

Tabel 4.4: Penjelasan Variabel Pada Fungsi *get_slider_result*

No	Variabel	Tipe Data	Penjelasan
1	check_result	vector < bool >	Hasil penentuan kondisi sekak ke delapan arah dari bidak raja.
2	right	set < llu >::iterator	Iterator yang menunjukkan posisi dari hasil pencarian *upper bound*.
3	left	set < llu >::iterator	Iterator yang menunjukkan posisi dari hasil pencarian *lower bound*.
4	enemy_color	int	Warna dari bidak yang berlawanan dengan bidak raja yang dievaluasi.
5	viewpoint	vector < set < llu > >	Vector yang berisikan referensi ke set yang berisi kumpulan bidak yang terletak pada satu garis dengan bidak raja.
6	view_search_val	vector < llu >	Vector yang berisikan nilai yang digunakan sebagai acuan pencarian *upper bound* dan *lower bound*.
7	view_upper	vector < int >	Vector yang berisikan arah pencarian dari *upper bound*.
8	view_lower	vector < llu >	Vector yang berisikan arah pencarian dari *lower bound*.
9	directions	vector < llu >	Vector yang berisikan sudut pandang yang digunakan dalam pencarian *upper bound* dan *lower bound*.

```
pll slider_checker_coord(llu x, llu y, llu z, int direction)
{
        if (direction == HORIZONTAL) return pll(z, y);
        else if (direction == VERTICAL) return pll(x, z);
        else if (direction == MAIN_DIAGONAL) return pll(z, (z
             - x + y));
        else if (direction == ANTI_DIAGONAL) return pll(z, (x
             + y - z));
}
```

Kode Sumber 4.14: Implementasi Fungsi Utilitas *slider_checker_coord*

```
int viewpoint_checker(char piece_type, int king_color, int
    direction)
{
        if (direction == HORIZONTAL || direction == VERTICAL)
                return horizontal_vertical_checker(piece_type
                    , king_color);
        else if (direction == MAIN_DIAGONAL || direction ==
            ANTI_DIAGONAL)
                return diagonal_checker(piece_type,
                    king_color);
}
```

Kode Sumber 4.15: Implementasi Fungsi Utilitas *viewpoint_checker*

```
int horizontal_vertical_checker(char piece_type, int
    king_color)
{
        if (is_enemy(piece_type, INVERSE_COLORS[king_color]))
        {
                if (find_in(HORIZONTAL_VERTICAL_PIECE,
                    piece_type))
                {
                        return 1;
                }
                return 0;
        }
        return 2;
}
```

Kode Sumber 4.16: Implementasi Fungsi Utilitas *horizontal_vertical_checker*

```
int diagonal_checker(char piece_type, int king_color)
{
        if (is_enemy(piece_type, INVERSE_COLORS[king_color]))
        {
                if (find_in(DIAGONAL_PIECE, piece_type))
                {
                        return 1;
                }
                return 0;
        }
        return 2;
}
```

Kode Sumber 4.17: Implementasi Fungsi Utilitas *diagonal_checker*

```
Piece create_piece(llu x, llu y, char type)
{
        Piece my_piece;
        my_piece.x_pos = x;
        my_piece.y_pos = y;
        my_piece.piece_type = type;

        return my_piece;
}
```

Kode Sumber 4.18: Implementasi Fungsi Utilitas *create_piece*

```
bool is_pinned(pll coord, int king_color)
{
        for (pll pinned_coord : pinned_list[king_color])
        {
                if (pinned_coord == coord)
                {
                        return true;
                }
        }
        return false;
}
```

Kode Sumber 4.19: Implementasi Fungsi Utilitas *is_pinned*

4.3.8 Implementasi Fungsi Horse Checker

Fungsi *horse_checker* merupakan implementasi dari desain fungsi pada subbab 3.6 yang digunakan untuk menentukan jumlah bidak kuda lawan yang menyebabkan bidak raja yang dievaluasi berada dalam kondisi sekak. Kode sumber dari implementasi fungsi ini dapat dilihat pada Kode Sumber 4.20 dengan penjelasan variabel pada Tabel 4.5.

```
int horse_checker(Piece king, map_coord &other_piece, int
    king_color, int phase)
{
        int checked_count = 0;
        int enemy_color = king_color^1;
        char enemy_horse = HORSES[enemy_color];
        for (pii move : HORSE_MOVE)
        {
                pll pos(king.x_pos + move.first, king.y_pos +
                     move.second);
                if (other_piece.find(pos) != other_piece.end
                    ())
                {
                        if (other_piece[pos] == enemy_horse)
                        {
                                if (phase == EATING_PHASE &&
                                    is_pinned(pos,
                                    enemy_color)) {
                                        continue;
                                }
                                checked_count += 1;
                                if (phase ==
                                    INITIAL_CHECK_PHASE) {
                                        Piece attacker =
                                            create_piece(pos
                                            .first, pos.
                                            second,
                                            other_piece[pos
                                            ]);
                                        attacker_list[
                                            king_color].
                                            push_back(
                                            attacker);
                                }
                        }
                }
        }

        return checked_count;
}
```

Kode Sumber 4.20: Implementasi Fungsi *horse_checker*

Tabel 4.5: Penjelasan Variabel Pada Fungsi *horse_checker*

No	Variabel	Tipe Data	Penjelasan
1	checked_count	int	Jumlah bidak kuda yang menyebabkan sekak pada bidak raja.
2	enemy_color	int	Warna dari bidak yang berlawanan dengan warna bidak raja yang dievaluasi.
3	enemy_horse	char	Tipe bidak dari bidak kuda lawan.

4.3.9 Implementasi Fungsi Pawn Checker

Fungsi *pawn_checker* merupakan implementasi dari desain fungsi pada subbab 3.7 yang digunakan untuk menentukan jumlah bidak prajurit lawan yang menyebabkan bidak raja yang dievaluasi berada dalam kondisi sekak. Kode sumber dari implementasi fungsi ini dapat dilihat pada Kode Sumber 4.21 dengan penjelasan variabel pada Tabel 4.6.

```
int pawn_checker(Piece king, map_coord &other_piece, int
    king_color, int phase)
{
        int enemy_color = king_color^1;
        char enemy_pawn = PAWNS[enemy_color];
        int checked_count = 0;
        for (pii move : PAWN_MOVE[enemy_color])
        {
                pll pos(king.x_pos + move.first, king.y_pos +
                     move.second);
                if (other_piece.find(pos) != other_piece.end
                    ())
                {
                        if (other_piece[pos] == enemy_pawn)
                        {
                                if (phase == EATING_PHASE &&
                                    is_pinned(pos,
                                    enemy_color)) {
                                        continue;
                                }
                                checked_count += 1;
                                if (phase ==
                                    INITIAL_CHECK_PHASE) {
                                        Piece attacker =
                                            create_piece(pos
                                            .first, pos.
                                            second,
                                            enemy_pawn);
                                        attacker_list[
                                            king_color].
                                            push_back(
                                            attacker);
                                }
                        }
                }
        }

        return checked_count;
}
```

Kode Sumber 4.21: Implementasi Fungsi *pawn_checker*

Tabel 4.6: Penjelasan Variabel Pada Fungsi *pawn_checker*

No	Variabel	Tipe Data	Penjelasan
1	checked_count	int	Jumlah bidak prajurit yang menyebabkan sekak pada bidak raja.
2	enemy_color	int	Warna dari bidak yang berlawanan dengan warna bidak raja yang dievaluasi.
3	enemy_pawn	char	Tipe bidak dari bidak prajurit lawan.

4.3.10 Implementasi Fungsi King Checker

Fungsi *king_checker* merupakan implementasi dari desain fungsi pada subbab 3.8 yang digunakan untuk menentukan apakah bidak raja lawan menyebabkan bidak raja sendiri dalam kondisi sekak. Kode sumber dari implementasi fungsi ini dapat dilihat pada Kode Sumber 4.22 dengan penjelasan variabel pada Tabel 4.7.

```
int king_checker(Piece king, map_coord &other_piece, int
    king_color)
{
        int checked_count = 0;
        int enemy_color = king_color^1;
        char enemy_king = KINGS[enemy_color];
        for (pii move : KING_MOVE)
        {
                pll pos(king.x_pos + move.first, king.y_pos +
                     move.second);
                if (other_piece.find(pos) != other_piece.end
                    ())
                {
                        if (other_piece[pos] == enemy_king)
                        {
                                checked_count += 1;
                        }
                }
        }

        return checked_count;
}
```

Kode Sumber 4.22: Implementasi Fungsi *king_checker*

Tabel 4.7: Penjelasan Variabel Pada Fungsi *king_checker*

No	Variabel	Tipe Data	Penjelasan
1	checked_count	int	Jumlah bidak raja lawan yang menyebabkan sekak pada bidak raja sendiri.
2	enemy_color	int	Warna dari bidak yang berlawanan dengan warna bidak raja yang dievaluasi.
3	enemy_king	char	Tipe bidak dari bidak raja lawan.

4.3.11 Implementasi Fungsi Penghitungan Jumlah Sekak

Fungsi *get_check_count* adalah implementasi dari desain fungsi pada subbab 3.9 yang digunakan untuk mencari jumlah total bidak yang menyebabkan sekak pada bidak raja. Kode sumber dari implementasi fungsi ini dapat dilihat pada Kode Sumber 4.23 dengan penjelasan variabel pada Tabel 4.8.

```
Check_Count get_check_count(Piece king, map_coord &
    piece_coord, vector<bool> &check_8_dir_result, int
    king_color, int phase)
{
        int enemy_color = king_color^1;
        char enemy_horse = HORSES[enemy_color];
        char enemy_pawn = PAWNS[enemy_color];
        char enemy_king = KINGS[enemy_color];

        int slider_count = 0;
                for (int i=0; i < 8; i++) {
                if (check_8_dir_result[i] == true) {
                        slider_count += 1;
                }
        }
        int horse_count = horse_checker(king, piece_coord,
            king_color, phase);
        int pawn_count = pawn_checker(king, piece_coord,
            king_color, phase);
        int king_count = 0;
        if (phase != EATING_PHASE)
        king_count = king_checker(king, piece_coord,
            king_color);
        int total_count = slider_count + horse_count +
            pawn_count + king_count;

        Check_Count checkers;
        checkers.eight_dir = slider_count;
        checkers.horse = horse_count;
        checkers.pawn = pawn_count;
        checkers.total_checked = total_count;

        return checkers;
}
```

Kode Sumber 4.23: Implementasi Fungsi *get_check_result*

Tabel 4.8: Penjelasan Variabel Pada Fungsi *get_check_count*

No	Variabel	Tipe Data	Penjelasan
1	enemy_color	int	Warna dari bidak yang berlawanan dengan warna bidak raja yang dievaluasi.
2	enemy_horse	char	Tipe bidak dari kuda lawan.
3	enemy_pawn	char	Tipe bidak dari prajurit lawan.
4	enemy_king	char	Tipe bidak dari bidak raja lawan.
5	slider_count	int	Jumlah sekak oleh bidak *slider* lawan.
6	horse_count	int	Jumlah sekak oleh bidak kuda lawan.
7	pawn_count	int	Jumlah sekak oleh bidak prajurit lawan.
8	king_count	int	Jumlah sekak oleh bidak raja lawan.
9	total_count	int	Gabungan semua jumlah sekak yang dialami oleh bidak raja.
10	checkers	Check_Count	Tipe data agregat yang menyimpan informasi dari tiap jumlah sekak yang dialami oleh bidak raja.

4.3.12 Implementasi Fungsi Menggerakkan Bidak Raja

Fungsi *try_moving_king* adalah implementasi dari desain fungsi pada subbab 3.10 yang digunakan untuk mencari jumlah langkah

valid yang didapat dari memindahkan posisi bidak raja. Kode sumber dari implementasi fungsi ini dapat dilihat pada Kode Sumber 4.24 dengan penjelasan variabel pada Tabel 4.9 dan Tabel 4.10.

Pada fungsi ini untuk setiap posisi baru, informasi dari bidak yang berada pada satu garis dengan bidak raja untuk tiap sudut pandang harus diperbaharui mengikuti posisi baru dari bidak raja. Hal ini dilakukan melalui pemanggilan fungsi *new_king_checker* yang akan menyalin informasi kumpulan bidak yang segaris dengan posisi baru bidak raja dari variabel *piece_set* yang berisikan informasi lengkap bidak pada papan, kemudian melakukan *insertion* bidak raja pada posisi baru dan menghapus bidak raja pada posisi awal. Kode sumber dari fungsi *new_king_checker* dapat dilihat pada Kode Sumber 4.26.

```
int try_moving_king(Piece king, vector<map_tree> &piece_set,
    map_coord &piece_coord, vector<bool> &checked_from, int
    king_color)
{
        int king_possible_move = 0;
        int enemy_color = king_color^1;
        char enemy_horse = HORSES[enemy_color];
        char enemy_pawn = PAWNS[enemy_color];
        char enemy_king = KINGS[enemy_color];

        for (int move : MOVE_8_DIR)
        {
                Piece new_king;
                new_king.x_pos = king.x_pos + KING_MOVE[move
                    ].first;
                new_king.y_pos = king.y_pos + KING_MOVE[move
                    ].second;
                new_king.piece_type = king.piece_type;
                pll king_coord(new_king.x_pos, new_king.y_pos
                    );
                int inv_move = (move <= 3) ? (move + 4) : (
                    move % 4);
                if (out_of_bound(king_coord) || checked_from[
                    inv_move])
                {
                        continue;
                }
...
```

Kode Sumber 4.24: Implementasi Fungsi *try_moving_king*

```
...
                if (checked_from[move]) {
                        if (piece_coord.find(king_coord) ==
                            piece_coord.end()) {
                                continue;
                        }
                }
                if (piece_coord.find(king_coord) !=
                    piece_coord.end()) {
                        char destination_piece = piece_coord[
                            king_coord];
                        if (!is_enemy(destination_piece,
                            INVERSE_COLORS[king_color])) {
                                continue;
                        }
                }

                Checker king_checker = new_king_checker(king,
                     new_king, piece_set, move);

                map_coord new_piece_coord = piece_coord;
                new_piece_coord[king_coord] = king.piece_type
                    ;

                vector<bool> king_check_result =
                    get_slider_result(new_king, king_checker
                    , new_piece_coord, king_color,
                    KING_MOVING_PHASE);
                Check_Count king_checked_count =
                    get_check_count(new_king,
                    new_piece_coord, king_check_result,
                    king_color, KING_MOVING_PHASE);

                if (king_checked_count.total_checked == 0)
                {
                        king_possible_move += 1;
                }
        }

        return king_possible_move;
}
```

Kode Sumber 4.25: Implementasi Fungsi *try_moving_king*

Tabel 4.9: Penjelasan Variabel Pada Fungsi *try_moving_king*

No	Variabel	Tipe Data	Penjelasan
1	enemy_color	int	Warna dari bidak yang berlawanan dengan warna bidak raja yang dievaluasi.
2	enemy_horse	char	Tipe bidak dari kuda lawan.
3	enemy_pawn	char	Tipe bidak dari prajurit lawan.
4	enemy_king	char	Tipe bidak dari bidak raja lawan.
5	king_possible_move	int	Jumlah langkah valid yang didapat dengan memindahkan posisi bidak raja.
6	new_king	Piece	Bidak raja pada posisi baru.
7	king_coord	pll	Koordinat baru dari bidak raja.
8	inv_move	int	Arah berlawanan darimana raja di sekak oleh bidak *slider* lawan.
9	king_checker	Checker	Kumpulan bidak yang berada pada satu garis horisontal, vertikal, main diagonal, dan anti diagonal dengan bidak raja pada posisi baru.

Tabel 4.10: Penjelasan Variabel Pada Fungsi *try_moving_king*

No	Variabel	Tipe Data	Penjelasan
10	new_piece_coord	map_coord	Struktur data yang berisi pemetaan koordinat bidak ke tipe bidak yang telah diperbaharui setelah bidak raja berpindah posisi.
11	king_check_result	vector < bool >	Vector yang berisikan penanda apakah bidak raja di sekak dari arah tertentu pada posisi baru.
12	king_checked_count	Check_Count	Informasi jumlah sekak yang dialami bidak raja pada posisi baru.

```
Checker new_king_checker(Piece old_king, Piece new_king,
    vector<map_tree> &piece_set, int move)
{
        Checker king_checker;
        king_checker.horizontal = add_piece(new_king.x_pos,
            piece_set[HORIZONTAL][new_king.y_pos]);
        king_checker.vertical = add_piece(new_king.y_pos,
            piece_set[VERTICAL][new_king.x_pos]);
        king_checker.main_diagonal = add_piece(new_king.x_pos
            , piece_set[MAIN_DIAGONAL][new_king.x_pos -
            new_king.y_pos]);
        king_checker.anti_diagonal = add_piece(new_king.x_pos
            , piece_set[ANTI_DIAGONAL][new_king.x_pos +
            new_king.y_pos]);

        if (move % 4 == 0)
        king_checker.vertical.erase(old_king.y_pos);
        if (move % 4 == 1)
        king_checker.anti_diagonal.erase(old_king.x_pos);
        if (move % 4 == 2)
        king_checker.horizontal.erase(old_king.x_pos);
        if (move % 4 == 3)
        king_checker.main_diagonal.erase(old_king.x_pos);

        return king_checker;
}
```

Kode Sumber 4.26: Implementasi Fungsi *new_king_checker*

4.3.13 Implementasi Fungsi Memakan Bidak Penyerang

Fungsi *try_eating_piece* merupakan implementasi dari desain fungsi pada subbab 3.11 yang digunakan untuk menentukan jumlah langkah valid yang didapat dengan memakan bidak penyerang. Pada fungsi ini dilakukan pembangunan *Checker* dengan bidak penyerang sebagai acuan, namun proses pembangunan *Checker* penyerang berbeda dengan pembangunan *Checker* ketika memindahkan posisi bidak raja. Pada pembangunan *Checker* ketika memindahkan bidak raja perlu dilakukan penambahan dan penghapusan bidak raja pada posisi baru dan posisi awal, sedangkan pada pembangunan *Checker* untuk bidak penyerang hanya perlu menyalin informasi dari *piece_set* yang telah ada. Kode sumber dari fungsi ini dapat

dilihat pada Kode Sumber 4.27 dengan penjelasan variabel pada Tabel 4.11.

```
int try_eating_piece(Piece attacker, vector<map_tree> &
    piece_set, map_coord &piece_coord, int attacker_color)
{
        int enemy_color = attacker_color^1;
        char enemy_horse = HORSES[enemy_color];
        char enemy_pawn = PAWNS[enemy_color];
        char enemy_king = KINGS[enemy_color];

        Checker attacker_checker;
        attacker_checker.horizontal = piece_set[HORIZONTAL][
            attacker.y_pos];
        attacker_checker.vertical = piece_set[VERTICAL][
            attacker.x_pos];
        attacker_checker.main_diagonal = piece_set[
            MAIN_DIAGONAL][attacker.x_pos - attacker.y_pos];
        attacker_checker.anti_diagonal = piece_set[
            ANTI_DIAGONAL][attacker.x_pos + attacker.y_pos];

        vector<bool> attacker_result = get_slider_result(
            attacker, attacker_checker, piece_coord,
            attacker_color, EATING_PHASE);
        Check_Count attacker_count = get_check_count(attacker
            , piece_coord, attacker_result, attacker_color,
            EATING_PHASE);

        return attacker_count.total_checked;
}
```

Kode Sumber 4.27: Implementasi Fungsi *try_eating_piece*

Tabel 4.11: Penjelasan Variabel Pada Fungsi *try_eating_piece*

No	Variabel	Tipe Data	Penjelasan
1	enemy_color	int	Warna dari bidak yang berlawanan dengan warna bidak raja yang dievaluasi.
2	enemy_horse	char	Tipe bidak dari kuda lawan.
3	enemy_pawn	char	Tipe bidak dari prajurit lawan.
4	enemy_king	char	Tipe bidak dari bidak raja lawan.
5	attacker_checker	Checker	Kumpulan bidak yang berada pada satu garis horisontal, vertikal, main diagonal, dan anti diagonal dengan bidak penyerang.
6	attacker_result	vector $<$ bool $>$	Vector yang berisikan penanda apakah bidak attacker dapat dimakan dari arah tertentu.
7	attacker_count	Check_Count	Informasi jumlah bidak yang dapat memakan bidak penyerang.

4.3.14 Implementasi Fungsi Intersepsi Slider

Fungsi *intercept_slider* adalah implementasi dari desain fungsi pada subbab 3.12 yang digunakan untuk menentukan jumlah langkah valid yang didapat dengan melakukan intersepsi serangan dari bidak penyerang dengan bidak *slider* kawan atau dengan kata lain mencari jumlah bidak *slider* kawan yang dapat berpindah ke jalur antara

penyerang dan bidak raja. Kode sumber dari fungsi *intercept_slider* dapat dilihat pada Kode Sumber 4.28 dengan penjelasan variabel pada Tabel 4.12.

Seperti yang telah dijelaskan pada subbab 3.12 untuk setiap *key value* dari *map-set* yang berada antara bidak raja dan bidak penyerang akan dilakukan pencarian interseksi antara garis yang dibuat antara bidak raja dan bidak penyerang, dan garis yang dibentuk oleh *map-set* dengan *key value* yang ditemukan. Posisi dari interseksi ini dicari dengan memanggil fungsi *get_intersection* yang akan menentukan posisi interseksi berdasarkan bidak penyerang dan arah bidak slider relatif terhadap bidak raja. Kode sumber dari fugsi *get_intersection* dapat dilihat pada Kode Sumber 4.29.

Setelah mendapat posisi interseksi, fungsi *intercept_slider* akan memanggil fungsi *intercept_processor* yang digunakan untuk menentukan jumlah slider yang dapat bergerak ke posisi interseksi pada *key value* tertentu. Kode sumber dari fungsi ini dapat dilihat pada Kode Sumber 4.30 dengan penjelasan variabel pada Tabel 4.13.

Selain itu dibutuhkan juga fungsi utilitas untuk menentukan apakah *key value* berada di antara bidak raja dan bidak penyerang (fungsi *in_between*), menentukan apakah slider yang ditemukan pada fungsi *intercept_processor_helper* dapat berpindah ke posisi intersepsi (fungsi *intercept_processor_helper*), dan menentukan koordinat bidak yang ditemukan berdasarkan *key value* dan hasil pencarian *lower bound* dan *upper bound* pada fungsi *intercept_processor* (fungsi *slider_intercept_coord*). Kode sumber dari fungsi-fungsi utilitas ini dapat dilihat pada Kode Sumber 4.31, 4.33, dan 4.34.

```
int intercept_slider(Piece king, Piece attacker, vector<
    map_tree> &piece_set, map_coord &piece_coord, int
    checked_from_dir, int king_color)
{
        int intercept_number = 0;
        for (int dir : SLIDER_DIRECTIONS)
        {
                if (dir == checked_from_dir)
                {
                        continue;
                }
                for (auto it : piece_set[dir])
                {
                        llu key = it.first;
                        if (in_between(king, attacker, dir,
                            key, checked_from_dir))
                        {
                                llu intersect =
                                    get_intersection(king,
                                    key, dir,
                                    checked_from_dir);
                                int num_of_intercept =
                                    intercept_processor(
                                    intersect, dir, key,
                                    piece_set[dir][key],
                                    piece_coord, king_color)
                                    ;
                                intercept_number +=
                                    num_of_intercept;
                        }
                }
        }

        return intercept_number;
}
```

Kode Sumber 4.28: Implementasi Fungsi *intercept_slider*

Tabel 4.12: Penjelasan Variabel Pada Fungsi *intercept_slider*

No	Variabel	Tipe Data	Penjelasan
1	intercept_number	int	Total jumlah intersepsi yang dapat dilakukan bidak kawan.
2	key	llu	*Key value* dari *map-set* sebagai mana dijelaskan pada subbab 2.3.1 dan 2.3.2.
3	intersect	llu	lokasi interseksi antara garis dengan *key value* tertentu dengan garis yang dibentuk antara bidak raja dan bidak penyerang.

```
llu get_intersection(Piece king, llu key, int direction, int
    checked_from)
{
        llu intersect = -1;
        if (checked_from == VERTICAL) intersect = king.x_pos;
        else if (checked_from == HORIZONTAL)
        {
        if (direction == VERTICAL) intersect = king.y_pos;
        else if (direction == ANTI_DIAGONAL) intersect = key
            - king.y_pos;
        else if (direction == MAIN_DIAGONAL) intersect = key
            + king.y_pos;
        }
        else if (checked_from == ANTI_DIAGONAL)
        {
        if (direction == VERTICAL) intersect = king.x_pos +
            king.y_pos - key;
        else if (direction == HORIZONTAL) intersect = king.
            x_pos + king.y_pos - key;
        else if (direction == MAIN_DIAGONAL) intersect = (
            king.x_pos + king.y_pos + key) / 2;
        }
        else if (checked_from == MAIN_DIAGONAL)
        {
        if (direction == VERTICAL) intersect = key - king.
            x_pos + king.y_pos;
        else if (direction == HORIZONTAL) intersect = king.
            x_pos - king.y_pos + key;
        else if (direction == ANTI_DIAGONAL) intersect = (
            king.x_pos - king.y_pos + key) / 2;
        }

        return intersect;
}
```

Kode Sumber 4.29: Implementasi Fungsi *get_intersection*

```
int intercept_processor(llu intersect, int direction, llu key
    , set<llu> &piece_set, map_coord &piece_coord, int
    king_color)
{
        int intercept_count = 0;
        set<llu>::iterator upper_value, lower_value;
        set<llu> pieces = piece_set;
        if (pieces.empty()) {
                return intercept_count;
        }
        pieces.insert(intersect);

        upper_value = pieces.upper_bound(intersect);
        if (upper_value != pieces.end()){
                if (*upper_value != intersect) {
                        intercept_count +=
                            intercept_processor_helper(
                            piece_coord, *upper_value, key,
                            king_color, direction);
                }
        }

        lower_value = pieces.lower_bound(intersect);
        if (lower_value != pieces.end())
        {
                lower_value--;
                if (lower_value != pieces.end())
                {
                        if (*lower_value != intersect) {
                                intercept_count +=
                                    intercept_processor_helper
                                    (piece_coord, *
                                    lower_value, key,
                                    king_color, direction);
                        }
                }
        }

        return intercept_count;
}
```

Kode Sumber 4.30: Implementasi Fungsi *intercept_processor*

Tabel 4.13: Penjelasan Variabel Pada Fungsi *intercept_processor*

No	Variabel	Tipe Data	Penjelasan
1	intercept_count	int	Jumlah intersepsi pada *key value* tertentu.
2	upper_value	set < llu >::iterator	Iterator menuju hasil pencarian *upper bound* pada *set* dengan *key value* tertentu.
3	lower_value	set < llu >::iterator	Iterator menuju hasil pencarian *lower bound* pada *set* dengan *key value* tertentu.

```
bool in_between(Piece king, Piece attacker, int direction,
    llu key, int checked_from)
{
        if (direction == VERTICAL)
        {
                llu lo = min(king.x_pos, attacker.x_pos);
                llu hi = max(king.x_pos, attacker.x_pos);
                if (key > lo && key < hi) return true;
                return false;
        }
...
```

Kode Sumber 4.31: Implementasi Fungsi Utilitas *in_between*

```
...
        else if (direction == ANTI_DIAGONAL)
        {
                llu lo = min(king.x_pos + king.y_pos,
                    attacker.x_pos + attacker.y_pos);
                llu hi = max(king.x_pos + king.y_pos,
                    attacker.x_pos + attacker.y_pos);
                if (key > lo && key < hi)
                {
                        if (checked_from == MAIN_DIAGONAL) {
                                if ((lo + key) % 2 == 0)
                                    return true;
                                else return false;
                        }
                        return true;
                }
                return false;
        }
        else if (direction == HORIZONTAL)
        {
                llu lo = min(king.y_pos, attacker.y_pos);
                llu hi = max(king.y_pos, attacker.y_pos);
                if (key > lo && key < hi) return true;
                return false;
        }
        else if (direction == MAIN_DIAGONAL)
        {
                llu lo = min(king.x_pos - king.y_pos,
                    attacker.x_pos - attacker.y_pos);
                llu hi = max(king.x_pos - king.y_pos,
                    attacker.x_pos - attacker.y_pos);
                if (key > lo && key < hi)
                {
                        if (checked_from == ANTI_DIAGONAL) {
                                if ((lo + key) % 2 == 0)
                                    return true;
                                else return false;
                        }
                        return true;
                }
                return false;
        }
}
```

Kode Sumber 4.32: Implementasi Fungsi Utilitas *in_between*

```
int intercept_processor_helper(map_coord &piece_coord, llu
    val, llu key, int king_color, int direction)
{
        int intercept_count = 0;
        int enemy_color = king_color^1;
        pll interceptor = slider_intercept_coord(val, key,
            direction);
        if (piece_coord.find(interceptor) != piece_coord.end
            ()) {
                char piece_type = piece_coord[interceptor];
                if (!out_of_bound(interceptor) && !is_pinned(
                    interceptor, king_color)) {
                        intercept_count = (viewpoint_checker(
                            piece_type, enemy_color,
                            direction) % 2);
                }
        }

        return intercept_count;
}
```

Kode Sumber 4.33: Implementasi Fungsi Utilitas *intercept_processor_helper*

```
pll slider_intercept_coord(llu val, llu key, int direction)
{
        if (direction == VERTICAL) return pll(key, val);
        else if (direction == ANTI_DIAGONAL) return pll(val,
            key - val);
        else if (direction == HORIZONTAL) return pll(val, key
            );
        else if (direction == MAIN_DIAGONAL) return pll(val,
            val - key);
}
```

Kode Sumber 4.34: Implementasi Fungsi Utilitas *slider_intercept_coord*

4.3.15 Implementasi Fungsi Intersepsi Kuda

Fungsi *intercept_horse* adalah implementasi dari desain fungsi pada subbab 3.13 yang digunakan untuk menentukan jumlah langkah valid yang didapat dengan melakukan intersepsi serangan dari bidak

penyerang dengan menggunakan bidak kuda kawan atau dengan kata lain mencari jumlah bidak kuda kawan yang dapat berpindah ke petak yang berada pada jalur antara bidak penyerang dan bidak raja. Kode sumber dari fungsi ini dapat dilihat pada Kode Sumber 4.35 dengan penjelasan variabel pada Tabel 4.14.

Dalam prosesnya fungsi *intercept_horse* membutuhkan beberapa fungsi utilitas seperti fungsi *in_line* untuk menentukan apakah suatu koordinat berada pada satu garis dengan bidak raja dan bidak penyerang, dan fungsi *in_middle* untuk menentukan apakah suatu koordinat berada diantara bidak penyerang dan bidak raja. Kode sumber untuk fungsi-fungsi utilitas ini dapat dilihat pada Kode Sumber 4.37, 4.38, dan 4.39.

```
int intercept_horse(Piece king, Piece attacker, map_list &
    piece_list, map_coord &piece_coord, int king_color, int
    checked_from)
{
        int intercept_number = 0;
        char ally_horse = HORSES[king_color];
        int enemy_color = king_color^1;

        for (Piece horse: piece_list[ally_horse]) {
                pll horse_coord = pll(horse.x_pos, horse.
                    y_pos);
                if (!is_pinned(horse_coord, king_color)){
                        for (pii move: HORSE_MOVE) {
                                pll new_coord = pll(horse.
                                    x_pos + move.first,
                                    horse.y_pos + move.
                                    second);
                                if (piece_coord.find(
                                    new_coord) !=
                                    piece_coord.end()) {
...
```

Kode Sumber 4.35: Implementasi Fungsi *intercept_horse*

```
...
                                        if (!is_enemy(
                                            piece_coord[
                                            new_coord],
                                            INVERSE_COLORS[
                                            king_color])) {
                                                continue;
                                        }
                                    }
                                    if (!out_of_bound(new_coord)
                                        && in_line(king,
                                        new_coord, checked_from)
                                         && in_middle(king,
                                        attacker, new_coord,
                                        checked_from)) {
                                            intercept_number +=
                                                1;
                                    }
                                }
                            }
        }

        return intercept_number;
}
```

Kode Sumber 4.36: Implementasi Fungsi *intercept_horse*

Tabel 4.14: Penjelasan Variabel Pada Fungsi *intercept_horse*

No	Variabel	Tipe Data	Penjelasan
1	intercept_number	int	Jumlah intersepsi pada oleh bidak kuda kawan.
2	ally_horse	char	Tipe bidak kuda kawan.
3	enemy_color	int	Warna dari bidak lawan.
4	horse_coord	pll	Koordinat dari bidak kuda.
5	new_coord	pll	Koordinat dari bidak kuda setelah melakukan pergerakan.

```
bool in_line(Piece king, pll interceptor_coord, int
    checked_from)
{
        if (checked_from == VERTICAL)
                return (interceptor_coord.first == king.x_pos
                    );
        else if (checked_from == ANTI_DIAGONAL)
                return (interceptor_coord.first +
                    interceptor_coord.second == king.x_pos +
                     king.y_pos);
        else if (checked_from == HORIZONTAL)
                return (interceptor_coord.second == king.
                    y_pos);
        else if (checked_from == MAIN_DIAGONAL)
                return (interceptor_coord.first -
                    interceptor_coord.second == king.x_pos -
                     king.y_pos);
        }
```

Kode Sumber 4.37: Implementasi Fungsi Utilitas *in_line*

```
bool in_middle(Piece king, Piece attacker, pll
    interceptor_coord, int checked_from)
{
        if (checked_from == VERTICAL)
        {
                pll lohi = pll(min(king.y_pos, attacker.y_pos
                    ), max(king.y_pos, attacker.y_pos));
                if (interceptor_coord.second > lohi.first &&
                    interceptor_coord.second < lohi.second)
                    {
                        return true;
                }
                return false;
        }
...
```

Kode Sumber 4.38: Implementasi Fungsi Utilitas *in_middle*

```
...
        else
        {
                pll lohi = pll(min(king.x_pos, attacker.x_pos
                    ), max(king.x_pos, attacker.x_pos));
                if (interceptor_coord.first > lohi.first &&
                    interceptor_coord.first < lohi.second) {
                        return true;
                }
                return false;
        }
}
```

Kode Sumber 4.39: Lanjutan Implementasi Fungsi Utilitas *in_middle*

4.3.16 Implementasi Fungsi Intersepsi Prajurit

Fungsi *intercept_pawn* adalah implementasi dari desain fungsi pada subbab 3.14 yang digunakan untuk menentukan jumlah langkah valid yang didapat dengan melakukan intersepsi serangan dari bidak penyerang dengan menggunakan bidak prajurit kawan atau dengan kata lain mencari jumlah bidak prajurit kawan yang dapat berpindah ke petak yang berada pada jalur antara bidak penyerang dan bidak raja. Kode sumber dari fungsi ini dapat dilihat pada Kode Sumber 4.40 dengan penjelasan variabel pada Tabel 4.15.

Fungsi *intercept_pawn* juga membutuhkan fungsi utilitas yang sama dengan fungsi *intercept_horse* yaitu fungsi *in_line* dan fungsi *in_middle* yand dapat dilihat pada Kode Sumber 4.37 dan 4.38.

```
int intercept_pawn(Piece king, Piece attacker, map_list &
    piece_list, map_coord &piece_coord, int king_color, int
    checked_from)
{
        int intercept_number = 0;
        int ally_pawn = PAWNS[king_color];
        int enemy_color = king_color^1;
        pii black_pawn_move = pii(0, -1);
        pii white_pawn_move = pii(0, 1);
        vector<pii> pawn_moves = {white_pawn_move,
            black_pawn_move};

        for (Piece pawn: piece_list[ally_pawn]) {
                pll pawn_coord = pll(pawn.x_pos, pawn.y_pos);
                if (!is_pinned(pawn_coord, king_color)) {
                        pii move = pawn_moves[king_color];
                        pll new_coord = pll(pawn_coord.first
                            + move.first, pawn_coord.second
                            + move.second);
                        if (piece_coord.find(new_coord) !=
                            piece_coord.end()) {
                                continue;
                        }
                        if (!out_of_bound(new_coord) &&
                            in_line(king, new_coord,
                            checked_from) && in_middle(king,
                             attacker, new_coord,
                            checked_from)) {
                                intercept_number += 1;
                        }
                }
        }

        return intercept_number;
}
```

Kode Sumber 4.40: Implementasi Fungsi *intercept_pawn*

Tabel 4.15: Penjelasan Variabel Pada Fungsi *intercept_pawn*

No	Variabel	Tipe Data	Penjelasan
1	intercept_number	int	Jumlah intersepsi pada oleh bidak kuda kawan.
2	ally_pawn	char	Tipe bidak prajurit kawan.
3	enemy_color	int	Warna dari bidak lawan.
4	black_pawn_move	pii	Cara bergerak dari bidak prajurit hitam.
5	white_pawn_move	pii	Cara bergerak dari bidak prajurit putih.
6	pawn_moves	vector < pii >	Wrapper white_pawn_move dan black_pawn_move.
7	pawn_coord	pll	Koordinat dari bidak prajurit.
8	new_coord	pll	Koordinat dari bidak prajurit setelah melakukan pergerakan.

4.3.17 Implementasi Fungsi Solve

Fungsi *solve* adalah implementasi dari desain fungsi pada subbab 3.15 yang merupakan fungsi utama untuk menyelesaikan permasalahan CSTATE3. Kode sumber dari fungsi ini dapat dilihat pada Kode Sumber 4.41 dan 4.42 dengan penjelasan variabel pada Tabel 4.16 dan Tabel 4.17.

Fungsi *solve* dalam prosesnya akan memanggil fungsi *only_one_check* ketika hanya salah satu bidak raja yang berada dalam kondisi sekak. Fungsi *only_one_check* ini adalah fungsi yang akan memanggil fungsi-fungsi lain yang telah dijelaskan pada subbab sebelumnya untuk menentukan jumlah langkah valid yang dapat di-

ambil. Kode sumber dari fungsi ini dapat dilihat pada Kode Sumber 4.43 dan 4.44 dengan penjelasan variabel pada Tabel 4.18.

```
void solve(map_list &pieces_list, map_coord &piece_coord)
{
        Piece black_king = pieces_list[BLACK_KING][0];
        Piece white_king = pieces_list[WHITE_KING][0];

        Checker black_king_checker = create_checker(
            black_king, pieces_list);
        Checker white_king_checker = create_checker(
            white_king, pieces_list);

        vector<bool> black_check_result = get_slider_result(
            black_king, black_king_checker, piece_coord,
            BLACK, INITIAL_CHECK_PHASE);
        vector<bool> white_check_result = get_slider_result(
            white_king, white_king_checker, piece_coord,
            WHITE, INITIAL_CHECK_PHASE);

        Check_Count black_checked_count = get_check_count(
            black_king, piece_coord, black_check_result,
            BLACK, INITIAL_CHECK_PHASE);
        Check_Count white_checked_count = get_check_count(
            white_king, piece_coord, white_check_result,
            WHITE, INITIAL_CHECK_PHASE);

        bool black_checked = black_checked_count.
            total_checked != 0;
        bool white_checked = white_checked_count.
            total_checked != 0;
...
```

Kode Sumber 4.41: Implementasi Fungsi *solve*

```
...
        if (black_checked && white_checked)
        {
                printf("Impossible\n");
        }
        else if (!black_checked && !white_checked)
        {
                printf("Safe\n");
        }
        else
        {
                if (black_checked)
                {
                        int possible_move = only_one_check(
                            black_king, piece_coord,
                            pieces_list, black_check_result,
                             black_checked_count, BLACK);
                        if(possible_move == 0) {
                                printf("Black Checkmate\n");
                        }
                        else {
                                printf("Black Check - %d
                                    Plausible Moves\n",
                                    possible_move);
                        }
                }
                else
                {
                        int possible_move = only_one_check(
                            white_king, piece_coord,
                            pieces_list, white_check_result,
                             white_checked_count, WHITE);
                        if(possible_move == 0) {
                                printf("White Checkmate\n");
                        }
                        else {
                                printf("White Check - %d
                                    Plausible Moves\n",
                                    possible_move);
                        }
                }
        }
}
```

Kode Sumber 4.42: Lanjutan Implementasi Fungsi *solve*

Tabel 4.16: Penjelasan Variabel Pada Fungsi *solve*

No	Variabel	Tipe Data	Penjelasan
1	black_king	Piece	Bidak raja hitam.
2	white_king	Piece	Bidak raja putih.
3	black_king_checker	Checker	Kumpulan bidak yang berada pada satu garis horisontal, vertikal, *main diagonal*, dan *anti diagonal* dengan bidak raja hitam.
4	white_king_checker	Checker	Kumpulan bidak yang berada pada satu garis horisontal, vertikal, *main diagonal*, dan *anti diagonal* dengan bidak raja putih.
5	black_check_result	vector $<$ bool $>$	Vector yang berisikan penanda apakah bidak raja hitam dapat dimakan pada arah tertentu.
6	white_check_result	vector $<$ bool $>$	Vector yang berisikan penanda apakah bidak raja putih dapat dimakan pada arah tertentu.
7	black_checker_count	Checker_Count	Informasi jumlah bidak yang menyebabkan sekak pada bidak raja hitam.
8	white_checker_count	Checker_Count	Informasi jumlah bidak yang menyebabkan sekak pada bidak raja putih.

Tabel 4.17: Penjelasan Variabel Pada Fungsi *solve*

No	Variabel	Tipe Data	Penjelasan
9	black_checked	bool	Penanda apakah bidak raja hitam berada dalam kondisi sekak.
10	white_checked	bool	Penanda apakah bidak raja putih berada dalam kondisi sekak.
11	possible_move	int	Jumlah langkah valid yang dapat dilakukan ketika bidak raja berada dalam kondisi sekak.

```
int only_one_check(Piece king, map_coord &piece_coord,
    map_list &piece_list, vector<bool> check_result,
    Check_Count checked_count, int king_color)
{
        vector<map_tree> piece_set = create_piece_set(
            piece_list);
        int enemy_color = king_color^1;
        int king_possible_move = 0;
        int eater_possible_move = 0;
        int intercept_possible_move = 0;

        king_possible_move = try_moving_king(king, piece_set,
             piece_coord, check_result, king_color);

        if (checked_count.total_checked == 1 && attacker_list
            [king_color].size() == 1)
        {
                Piece attacker = attacker_list[king_color
                    ][0];
                eater_possible_move = try_eating_piece(
                    attacker, piece_set, piece_coord,
                    enemy_color);
...
```

Kode Sumber 4.43: Implementasi Fungsi *only_one_check*

```
...
                if (checked_count.eight_dir == 1)
                {
                        int checked_dir = get_checked_dir(
                            check_result);
                        intercept_possible_move =
                            try_intercepting(king, attacker,
                             piece_set, piece_coord,
                            piece_list, checked_dir,
                            king_color);
                }
        }
        int all_possible_move = king_possible_move +
            eater_possible_move + intercept_possible_move;

        return all_possible_move;
}
```

Kode Sumber 4.44: Lanjutan Implementasi Fungsi *only_one_check*

Tabel 4.18: Penjelasan Variabel Pada Fungsi *only_one_check*

No	Variabel	Tipe Data	Penjelasan
1	piece_set	vector < map_tree >	Informasi kumpulan bidak yang berada pada satu garis dikelompokkan berdasarkan *key value* yang bersesuaian (subbab 2.3.2).
2	enemy_color	int	Warna dari bidak lawan.
3	king_possible_move	int	Jumlah langkah valid yang didapat dari menggerakkan bidak raja ke posisi yang tidak terkena sekak.
4	eater_possible_move	int	Jumlah langkah valid yang didapat dari memakan bidak penyerang yang menyebabkan sekak.
5	intercept_possible_move	int	Jumlah langkah valid yang didapat dari intersepsi jalur antara bidak penyerang dan bidak raja.
6	all_possible_move	int	Jumlah total dari langkah valid yang dapat dilakukan.

BAB 5

UJI COBA DAN EVALUASI

Bagian ini akan menjelaskan mengenai uji coba dan evaluasi daripada implementasi yang telah dilakukan pada bagian sebelumnya.

5.1 Lingkungan Uji Coba

Lingkungan uji coba pada daring penilaian *Sphere Online Judge* memiliki spesifikasi sebagai berikut:

1. Perangkat Keras:
    - *Processor* Intel(R) Pentium G860 CPU @ 3GHz.
    - Memori 1536 MB
2. Perangkat Lunak:
    - *Compiler* g++ 6.3 C++14

Uji coba tidak hanya dilakukan pada daring penilaian *Sphere Online Judge*. Uji coba dilakukan pada komputer lokal dengan spesifikasi sebagai berikut:

1. Perangkat Keras
    - Processor: Intel Core i7 8^{th} Gen
    - Memori: 16 GB
2. Perangkat Lunak
    - Sistem Operasi: Windows 10 Home
    - IDE : *Dev-C++*
    - Compiler: gcc 7.3 C++14

Gambar 5.1: Hasil Umpan Balik Solusi pada Daring SPOJ dengan Waktu Terbaik

File #0: Result is AC, time = 0.000000, mem = 15280
File #1: Result is AC, time = 0.800000, mem = 16104
File #2: Result is AC, time = 0.940000, mem = 17128
File #3: Result is AC, time = 1.310000, mem = 33512
File #4: Result is AC, time = 0.740000, mem = 17040
File #5: Result is AC, time = 1.900000, mem = 55280
File #6: Result is AC, time = 1.320000, mem = 25384
File #7: Result is AC, time = 1.290000, mem = 74496
File #8: Result is AC, time = 2.590000, mem = 33464
File #9: Result is AC, time = 2.720000, mem = 86656
File #10: Result is AC, time = 2.340000, mem = 22840
File #11: Result is AC, time = 4.180000, mem = 85184
File #12: Result is AC, time = 2.060000, mem = 50920
File #13: Result is AC, time = 4.340000, mem = 22096
File #14: Result is AC, time = 0.660000, mem = 16104
File #15: Result is AC, time = 0.650000, mem = 15472

Gambar 5.2: Detail Hasil Umpan Balik Solusi pada Daring SPOJ dengan Waktu Terbaik

5.2 Uji Kebenaran Program

Uji coba kebenaran program dilakukan dengan cara mengirimkan kode sumber ke dalam situs penilaian *Sphere Online Judge*. Sistem pada situs penilaian tersebut akan menjalankan program yang dikirim oleh *user* dan hasilnya kemudian akan dicocokkan dengan file keluaran yang telah disediakan oleh pembuat soal. Permasalahan yang menjadi perhatian dalam Tugas Akhir ini adalah *SPOJ Klasik 32092 Chessboard State 3: Intergalactic Chess Tournament*. Hasil uji coba dengan waktu terbaik ditunjukkan pada Gambar 5.1 dengan perincian waktu eksekusi untuk tiap kasus uji ditunjukkan pada Gambar 5.2.

Selain itu juga dilakukan pengujian sebanyak 12 kali pada situs penilaian daring SPOJ untuk melihat variasi waktu dan memori yang dibutuhkan program. Hasil ujicoba sebanyak 12 kali dapat dilihat pada Gambar lampiran A.1. Dari hasil uji coba dapat ditarik beberapa informasi seperti yang tertera pada Tabel 5.1.

Tabel 5.1: Kecepatan dan Memori Maksimal, Minimal, dan Rata-Rata dari Hasil Uji Coba Pengumpulan 12 Kali Pada Situs Pengujian Daring SPOJ

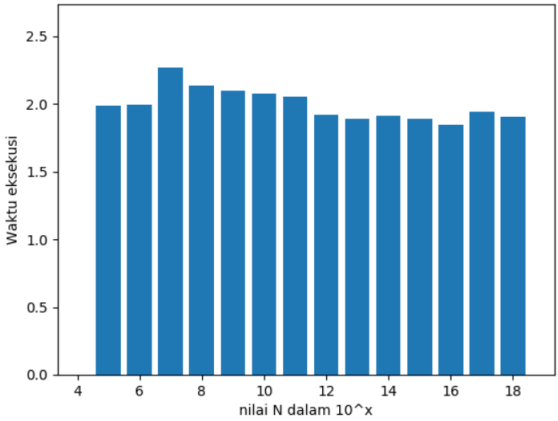
Waktu Maksimal	29,44 detik
Waktu Minimal	27,84 detik
Waktu Rata-Rata	28,60 detik
Memori Maksimal	85 MB
Memori Minimal	75 MB
Memori Rata-Rata	84 MB

Berdasarkan Tabel 5.1 dari percobaan yang dilakukan didapatkan waktu eksekusi rata-rata 28,60 detik dan waktu maksimal adalah 29,44 detik pada daring SPOJ. Pada permasalahan ini terdapat 27 pengumpulan solusi dan hanya 4 pengguna saja yang berhasil lolos dari kasus uji yang diberikan pembuat permasalahan. Implementasi ini mendapat peringkat tertinggi dari segi waktu dibandingkan dengan implementasi lain yaitu di waktu terendah 27,84 detik. Umpan balik dari pengumpulan berkas ke daring SPOJ dapat dilihat pada Gambar lampiran A.1.

5.3 Uji Kinerja Program

Uji kinerja daripada program dilakukan dengan cara mengeksekusi solusi yang telah dibangun yang memiliki kompleksitas $O(p \log q \log q + 18q \log q)$ dimana,

- p adalah jumlah bidak dalam satu papan.



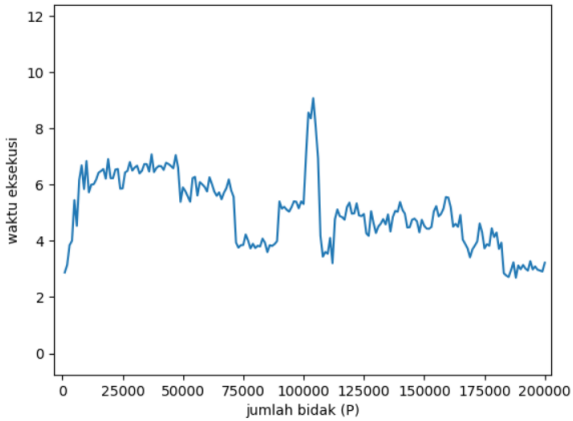
Gambar 5.3: Hasil Eksekusi untuk Besar Papan 10^5 hingga 10^{18}

- q merupakan nilai minimal dari n (besar papan) dan p (jumlah bidak).
- t dan p memiliki hubungan $t*p=10^6$.

Solusi tersebut akan dieksekusi dengan kasus uji yang dibangkitkan secara acak di komputer lokal sebagai masukan. Data yang dibangkitkan secara acak dibagi menjadi dua macam yaitu data dengan besar papan acak dan data dengan jumlah bidak acak.

Pengujian kinerja untuk solusi diatas dilakukan dengan membangkitkan 30 kasus untuk nilai N=10^5 hingga N=10^{18} dengan kenaikan dikalikan 10 dan 30 kasus uji untuk P=1000 hingga P=200000 dengan kenaikan 1000. Untuk satu nilai N dan/atau P, akan terdapat 30 waktu yang tercatat oleh sistem yang kemudian 30 nilai tersebut kemudian dicari rata-ratanya sehingga muncul 1 nilai yang mewakili waktu eksekusi.

Grafik yang ditunjukkan pada Gambar 5.3 dan 5.4 menunjukkan bahwasanya:



Gambar 5.4: Hasil Eksekusi untuk Jumlah Bidak antara 1000 hingga 200000

1. Jumlah bidak merupakan variabel yang memiliki efek dominan pada kompleksitas.

2. Ukuran papan merupakan variabel yang memiliki efek minimal pada kompleksitas.

3. Waktu eksekusi tertinggi didapat pada kisaran P=10^5 dan T=10.

4. Jumlah bidak yang terlalu kecil meningkatkan kemungkinan terjadinya kondisi ”Safe” dimana kedua bidak raja tidak dalam kondisi sekak. Hal ini menyebabkan waktu eksekusi yang lebih cepat dikarenakan tidak perlu melakukan pencarian jumlah langkah valid.

5. Jumlah bidak yang terlalu besar meningkatkan kemungkinan terjadinya kondisi ”Impossible” dimana kedua bidak raja berada dalam kondisi sekak. Hal ini menyebabkan waktu eksekusi yang lebih cepat dikarenakan tidak perlu melakukan pencarian jumlah langkah valid.

6. Tipe bidak juga dapat mempengaruhi waktu eksekusi dikarenakan biaya pengecekan untuk bidak *slider* jauh lebih tinggi dibandingkan bidak *non-slider*. Hal ini terlihat pada grafik untuk nilai P yang relatif dekat dapat terjadi fluktuasi waktu eksekusi.

Tabel data uji performa dapat dilihat pada bagian Lampiran B dan Lapiran C (Tabel B.1 dan Tabel C.1 hingga Tabel C.8)

BAB 6

KESIMPULAN

6.1 Kesimpulan

Beberapa kesimpulan yang dapat ditarik dari pengerjaan Tugas Akhir ini, khususnya pada pengimplementasian solusi program untuk permasalahan *SPOJ Klasik 32092 Chessboard State 3: Intergalactic Chess Tournament* adalah sebagai berikut:

1. Konsep *Divide and Conquer* dapat digunakan untuk memecah permasalahan besar mencari kondisi permainan catur dan jumlah langkah valid menjadi permasalahan sederhana yaitu sejumlah pencarian bidak terdekat dengan menggunakan *lower bound* dan *upper bound*.

2. Penentuan kondisi permainan untuk satu kasus uji memiliki kompleksitas *O(p log q log q + 18q log q)* dengan waktu yang dibutuhkan program untuk menyelesaikan permasalahan *SPOJ Klasik 32092 Chessboard State 3: Intergalactic Chess Tournament* dengan waktu minimum 27,84 detik, maksimum 29,44 detik, rata-rata 28,60 detik dan memori minimum 75 MB, maksimum 85 MB, rata-rata 84 MB untuk 15 kasus uji.

3. Jumlah bidak yang terlalu banyak atau terlalu sedikit pada suatu papan akan memiliki waktu eksekusi yang cukup rendah dikarenakan kemungkinan untuk muncul kondisi permainan ”Safe” atau ”Impossible” meningkat.

4. Selain dari jumlah bidak kompleksitas juga dipengaruhi tipe bidak pada papan catur dimana biaya perhitungan untuk bidak *slider* lebih besar daripada bidak *non-slider*.

5. Penggunaan struktur data *map of set* sudah tepat untuk permasalahan ini dikarenakan pencarian bidak terdekat dapat dilakukan dengan kompleksitas yang bergantung pada jumlah bidak dan bukan pada besar papan.

6.2 Saran

Pada Tugas Akhir ini, tentu tidak terlepas daripada kekurangan serta hal-hal penting yang dapat menjadi pertimbangan kelak. Berikut saran-saran yang dapat diambil dari Tugas Akhir ini:

1. Cara merepresentasikan bidak dan pencarian bidak terdekat pada tugas akhir ini dapat menjadi referensi untuk permasalahan serupa untuk kedepannya.

2. Metode yang diimplementasikan pada Tugas Akhir ini dapat menjadi bahan riset untuk permasalahan penentuan kondisi dan/atau jumlah langkah valid pada permainan papan dengan ukuran papan yang besar.

DAFTAR PUSTAKA

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, **Introduction To Algorithm, 2nd ed**. Cambridge, Massachusetts London, England: The MIT Press, 2001.

[2] S. Halim and F. Halim, **Competitive Programming 3**.Singapore, 2013.

[3] Anonim. **Lower and Upper Bound** [Online]. Tersedia: http://www.cplusplus.com/reference/algorithm/lower_bound/, http://www.cplusplus.com/reference/algorithm/upper_bound/ [Diakses 1 September 2018]

[4] Vuckovic, Vladan. 2012. **An Alternative Chessboard Representation Based On 4-bit Piece Coding**. 1: 269

[Halaman ini sengaja dikosongkan]

LAMPIRAN A

UJI COBA CSTATE3

22936363	☐	2018-12-24 13:39:18	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	29.09	80M	CPP14
22936358	☐	2018-12-24 13:38:02	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	29.12	75M	CPP14
22936350	☐	2018-12-24 13:36:24	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	29.25	85M	CPP14
22936344	☐	2018-12-24 13:35:05	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	28.65	85M	CPP14
22936339	☐	2018-12-24 13:33:21	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	29.44	85M	CPP14
22911075	☐	2018-12-20 04:27:28	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	28.17	84M	CPP14
22911071	☐	2018-12-20 04:24:56	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	28.14	85M	CPP14
22889715	☐	2018-12-16 14:45:47	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	27.84	85M	CPP14
22889688	☐	2018-12-16 14:41:28	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	28.43	85M	CPP14
22889277	☐	2018-12-16 12:47:22	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	27.98	84M	CPP14
22888676	☐	2018-12-16 10:44:07	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	27.98	85M	CPP14
22578147	☐	2018-10-25 14:31:33	Chessboard State 3: Intergalactic Chess Tournament	**accepted** edit ideone it	29.03	85M	CPP14

Gambar A.1: Hasil Umpan Balik Pengumpulan Kode Sumber Sebanyak 12 kali

[Halaman ini sengaja dikosongkan]

LAMPIRAN B

TABEL DATA UJI KINERJA

Tabel B.1: Hasil Uji Coba 1 Pada Mesin Lokal

N	Waktu Eksekusi (detik)
10^5	1.98439
10^6	1.99705
10^7	2.26688
10^8	2.13212
10^9	2.10172
10^{10}	2.07671
10^{11}	2.05392
10^{12}	1.92201
10^{13}	1.88855
10^{14}	1.91118
10^{15}	1.89033
10^{16}	1.84647
10^{17}	1.94019
10^{18}	1.90227

[Halaman ini sengaja dikosongkan]

LAMPIRAN C

TABEL DATA UJI KINERJA 2

Tabel C.1: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
1000	1000	2.87379
500	2000	3.14598
333	3000	3.84887
250	4000	4.0016
200	5000	5.45016
166	6000	4.53631
142	7000	6.16388
125	8000	6.68479
111	9000	5.84385
100	10000	6.83902
90	11000	5.72568
83	12000	6.00031
76	13000	6.0139
71	14000	6.17457

Tabel C.2: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
66	15000	6.4216
62	16000	6.48352
58	17000	6.55447
55	18000	6.21376
52	19000	6.90916
50	20000	6.23156
47	21000	6.22887
45	22000	6.53529
43	23000	6.55699
41	24000	5.85555
40	25000	5.86621
38	26000	6.43187
37	27000	6.4895
35	28000	6.80133
34	29000	6.50001
33	30000	6.60847
32	31000	6.67256
31	32000	6.39795
30	33000	6.49299
29	34000	6.73311
28	35000	6.72889

Tabel C.3: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
27	36000	6.46601
27	37000	7.07602
26	38000	6.44624
25	39000	6.59192
25	40000	6.66453
24	41000	6.65381
23	42000	6.5243
23	43000	6.77763
22	44000	6.73531
22	45000	6.66343
21	46000	6.58228
21	47000	7.05168
20	48000	6.60058
20	49000	5.38894
20	50000	5.90143
19	51000	5.76738
19	52000	5.58421
18	53000	5.3939
18	54000	6.22844
18	55000	6.27607
17	56000	5.61425
17	57000	6.08944
17	58000	6.0092
16	59000	5.91842
16	60000	5.75801
16	61000	6.26089
16	62000	6.02498

Tabel C.4: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
15	63000	5.75265
15	64000	5.60123
15	65000	5.72499
15	66000	5.48131
14	67000	5.71327
14	68000	5.87325
14	69000	6.18558
14	70000	5.79631
14	71000	5.56653
13	72000	3.93981
13	73000	3.76109
13	74000	3.84618
13	75000	3.85429
13	76000	4.22535
12	77000	4.01372
12	78000	3.73298
12	79000	3.89761
12	80000	3.74452
12	81000	3.82673
12	82000	3.80351
12	83000	4.07828
11	84000	3.93484
11	85000	3.59966
11	86000	3.84657
11	87000	3.82041
11	88000	3.88823
11	89000	3.98841
11	90000	5.40254

Tabel C.5: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
10	91000	5.14654
10	92000	5.21091
10	93000	5.10758
10	94000	5.0394
10	95000	5.19842
10	96000	5.40572
10	97000	5.38846
10	98000	5.15833
10	99000	5.40244
10	100000	5.31125
9	101000	7.01664
9	102000	8.5616
9	103000	8.35796
9	104000	9.07405
9	105000	8.08473
9	106000	6.93486
9	107000	4.17865
9	108000	3.43908
9	109000	3.60723
9	110000	3.5432
9	111000	4.10253
8	112000	3.20624
8	113000	4.76979
8	114000	5.11683
8	115000	4.88019
8	116000	4.8447
8	117000	4.75274

Tabel C.6: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
8	118000	5.20446
8	119000	5.36085
8	120000	4.96881
8	121000	4.97725
8	122000	5.33577
8	123000	4.90316
8	124000	4.88141
8	125000	4.95492
7	126000	4.27007
7	127000	4.17848
7	128000	5.05296
7	129000	4.63858
7	130000	4.27888
7	131000	4.50986
7	132000	4.61429
7	133000	4.77385
7	134000	4.58536
7	135000	4.94017
7	136000	4.33498
7	137000	4.8483
7	138000	5.06624
7	139000	5.033
7	140000	5.3813
7	141000	5.10402
7	142000	4.96373
6	143000	4.47413
6	144000	4.48652
6	145000	4.75423

Tabel C.7: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
6	146000	4.79044
6	147000	4.69325
6	148000	4.30374
6	149000	4.7502
6	150000	4.53835
6	151000	4.43771
6	152000	4.42774
6	153000	4.49665
6	154000	5.05189
6	155000	5.23482
6	156000	4.872
6	157000	4.96137
6	158000	5.14792
6	159000	5.55885
6	160000	5.53733
6	161000	5.19524
6	162000	4.50639
6	163000	4.60685
6	164000	4.50108
6	165000	4.92132
6	166000	4.04385
5	167000	3.89376
5	168000	3.73012
5	169000	3.41493
5	170000	3.69892
5	171000	3.83319
5	172000	3.9823

Tabel C.8: Hasil Uji Coba 2 Pada Mesin Lokal

T	P	Waktu Eksekusi (detik)
5	173000	4.61998
5	174000	4.31477
5	175000	3.7322
5	176000	3.88301
5	177000	3.82723
5	178000	4.4494
5	179000	4.13707
5	180000	4.29647
5	181000	3.71394
5	182000	3.93644
5	183000	2.8499
5	184000	2.76962
5	185000	2.71352
5	186000	2.95889
5	187000	3.23667
5	188000	2.68955
5	189000	3.1249
5	190000	2.99057
5	191000	3.1434
5	192000	3.01294
5	193000	2.94547
5	194000	3.27959
5	195000	2.98435
5	196000	3.0905
5	197000	2.97447
5	198000	2.94522
5	199000	2.91058
5	200000	3.22298

BIODATA PENULIS

Yustian lahir di Surabaya, 25 Februari 1997. Penulis telah menempuh 12 tahun pendidikan formal di TK Petra (2004-2006), SDK Santo Xaverius Surabaya (2006-2009), SMPK Angelus Custos (2009-2012), dan SMAK Frateran (2012-2015). Penulis kemudian melanjutkan pendidikannya di Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember Surabaya.

Selama masa studinya di Jurusan Teknik Informatika ITS Surabaya, penulis mengambil bidang minat Algoritma Pemrograman. Penulis juga aktif mengikuti kompetisi dalam bidang pemrograman tingkat nasional. Di dalam kampus pula, penulis pernah menjadi Asisten Dosen mata kuliah Struktur Data. Dalam kegiatan keorganisasian, penulis terlibat langsung sebagai anggota tim soal *National Programming Competition* 2016, tim soal *National Programming Competition* 2017, panitia Pemusatan Latihan Nasional 2 TOKI 2016. Penulis dapat dihubungi melalui surel `yustian9722583@gmail.com`.