

SKRIPSI

PENCARIAN JUMLAH KAMERA STATIS MINIMUM DALAM SUATU RUANGAN MENGGUNAKAN LINEAR PROGRAMMING



Prayogo Cendra

NPM: 2014730033

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI DAN SAINS
UNIVERSITAS KATOLIK PARAHYANGAN
2018

UNDERGRADUATE THESIS

**FINDING MINIMUM STATIC CAMERA IN A ROOM USING
LINEAR PROGRAMMING**



Prayogo Cendra

NPM: 2014730033

**DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHNOLOGY AND SCIENCES
PARAHYANGAN CATHOLIC UNIVERSITY
2018**

LEMBAR PENGESAHAN

PENCARIAN JUMLAH KAMERA STATIS MINIMUM DALAM SUATU RUANGAN MENGGUNAKAN LINEAR PROGRAMMING

Prayogo Cendra

NPM: 2014730033

Bandung, 18 Mei 2018

Menyetujui,

Pembimbing

Claudio Franciscus, M.T.

Ketua Tim Penguji

Anggota Tim Penguji

«penguji 1»

«penguji 2»

Mengetahui,

Ketua Program Studi

Mariskha Tri Adithia, P.D.Eng

PERNYATAAN

Dengan ini saya yang bertandatangan di bawah ini menyatakan bahwa skripsi dengan judul:

PENCARIAN JUMLAH KAMERA STATIS MINIMUM DALAM SUATU RUANGAN MENGGUNAKAN LINEAR PROGRAMMING

adalah benar-benar karya saya sendiri, dan saya tidak melakukan penjiplakan atau pengutipan dengan cara-cara yang tidak sesuai dengan etika keilmuan yang berlaku dalam masyarakat keilmuan.

Atas pernyataan ini, saya siap menanggung segala risiko dan sanksi yang dijatuhkan kepada saya, apabila di kemudian hari ditemukan adanya pelanggaran terhadap etika keilmuan dalam karya saya, atau jika ada tuntutan formal atau non-formal dari pihak lain berkaitan dengan keaslian karya saya ini.

Dinyatakan di Bandung,
Tanggal 18 Mei 2018

Meterai Rp. 6000

Prayogo Cendra
NPM: 2014730033

ABSTRAK

Kamera CCTV merupakan kamera yang digunakan untuk memantau suatu lokasi dengan tujuan pengawasan dan keamanan. Kamera CCTV pada umumnya dipasang di tempat-tempat strategis sehingga mendapatkan jangkauan yang baik. Penempatan kamera CCTV di ruangan yang berbentuk sederhana (persegi panjang) relatif tidak sulit. Kamera CCTV yang dibutuhkan pada umumnya berjumlah dua buah dan dipasang di kedua sudut ruangan sehingga saling berhadapan. Namun, jika ruangan berukuran besar, maka tujuan penggunaan kamera CCTV bukan hanya untuk mendeteksi adanya orang, melainkan juga untuk mengenali orang tersebut. Hal ini dapat menyebabkan kesulitan dalam menentukan jumlah minimum dan lokasi penempatan kamera CCTV yang dapat mencakup seluruh ruangan.

Pada skripsi ini, masalah akan akan dipelajari lebih lanjut dengan memahami setiap elemen pembentuk masalah. Selanjutnya, masalah ini akan dirumuskan lebih lanjut agar menjadi lebih konkret. Untuk menyelesaikan masalah ini, penulis menggunakan metode linear programming karena metode ini dapat menyelesaikan masalah optimasi. Masalah ini termasuk ke dalam jenis masalah optimasi karena solusi yang diharapkan harus bersifat paling optimal, yaitu penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh ruangan.

Selain merumuskan masalah, penulis juga membangun perangkat lunak yang dapat mensimulasikan masalah. Perangkat lunak ini dapat menerima masukan-masukan masalah dan menyelesaikannya menggunakan metode linear programming. Tidak hanya menyelesaikannya saja, perangkat lunak juga dapat memvisualisasikan solusinya sehingga penempatan-penempatan kamera CCTV dapat dipahami dengan lebih mudah.

Kata-kata kunci: cctv, linear programming

ABSTRACT

CCTV cameras are cameras used to monitor a location with the purpose of surveillance and security. CCTV cameras are generally installed in strategic places to get a good coverage. The placement of CCTV cameras in a simple room (rectangle) is relatively not difficult. CCTV cameras that are needed in general amount to two pieces and installed in both corners of the room so they are facing each other. However, if the room is large, then the purpose of using CCTV cameras is not only to detect people, but also to recognize the person. This can cause difficulties in determining the minimum number and location of CCTV camera placement that can cover the entire room.

In this thesis, the problem will be studied further by understanding every problem-forming element. Furthermore, this problem will be formulated further to be more concrete. To solve this problem, the author uses linear programming method because this method can solve the optimization problem. This problem belongs to the type of optimization problem because the expected solution should be the most optimal, that is the minimum placements of CCTV camera that can cover the entire room.

In addition to formulating the problem, the authors also build software that can simulate the problem. This software can receive input problems and solve them using linear programming method. Not only solve it, the software can also visualize the solution so that the placement of CCTV cameras can be understood more easily.

Keywords: cctv, linear programming

«kepada siapa anda mempersembahkan skripsi ini...?»

KATA PENGANTAR

«Tuliskan kata pengantar dari anda di sini ...»

Bandung, Mei 2018

Penulis

DAFTAR ISI

KATA PENGANTAR	xv
DAFTAR ISI	xvii
DAFTAR GAMBAR	xix
DAFTAR TABEL	xxi
1 PENDAHULUAN	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	2
1.3 Tujuan	2
1.4 Batasan Masalah	2
1.5 Metodologi	2
1.6 Sistematika Pembahasan	3
2 LANDASAN TEORI	5
2.1 <i>Linear Programming</i>	5
2.1.1 Karakteristik	5
2.1.2 Daerah <i>Feasible</i> dan Solusi Optimal	6
2.2 <i>Binary Integer Programming</i>	7
2.2.1 Algoritma <i>Balas's Additive</i>	8
2.3 Penelitian Terkait	9
3 ANALISIS	13
3.1 Pemodelan Masalah	13
3.1.1 Ruang	13
3.1.2 Kamera CCTV	15
3.1.3 Cakupan Kamera CCTV	15
3.2 Penyelesaian Masalah	18
3.2.1 Variabel	18
3.2.2 Fungsi Tujuan	19
3.2.3 Batasan	19
3.2.4 Model Masalah <i>Binary Integer Programming</i>	19
3.3 Analisis Kebutuhan Perangkat Lunak	20
3.3.1 Diagram <i>Use Case</i> dan Skenario	20
3.3.2 Kebutuhan Masukan Perangkat Lunak	20
3.3.3 Kebutuhan Keluaran Perangkat Lunak	21
3.3.4 Diagram Kelas Sederhana	21
4 PERANCANGAN	23
4.1 Perancangan Antarmuka	23
4.2 Perancangan Kelas	24

4.2.1	Kelas <i>Angle</i>	27
4.2.2	Kelas <i>Point</i>	28
4.2.3	Kelas <i>CameraSpecification</i>	29
4.2.4	Kelas <i>CameraPlacement</i>	29
4.2.5	Kelas <i>Dimension</i>	30
4.2.6	Kelas <i>Cell</i>	30
4.2.7	Kelas <i>CellMatrix</i>	31
4.2.8	Kelas <i>Room</i>	32
4.2.9	Kelas <i>MinimumCameraPlacementSolver</i>	33
4.2.10	Kelas <i>MinimumCameraPlacementSolverBalasAdditive</i>	34
4.2.11	Kelas <i>Variable</i>	34
4.2.12	Kelas <i>BIPFunction</i>	35
4.2.13	Kelas <i>BIPConstraint</i>	35
4.2.14	Kelas <i>BIPProblem</i>	36
4.2.15	Kelas <i>BIPFeasibilityCheckResult</i>	36
4.2.16	Kelas <i>Node</i>	37
4.2.17	Kelas <i>NodeStatus</i>	38
4.2.18	Kelas <i>BalasAdditiveBIPSolver</i>	38
4.2.19	Kelas <i>BalasAdditiveBIPSolverResult</i>	39
5	IMPLEMENTASI DAN PENGUJIAN	41
5.1	Lingkungan Implementasi Perangkat Keras	41
5.2	Lingkungan Implementasi Perangkat Keras	41
5.3	Implementasi Antarmuka	41
5.4	Pengujian	42
5.4.1	Pengujian-1	42
5.4.2	Pengujian-2	43
6	KESIMPULAN DAN SARAN	45
6.1	Kesimpulan	45
6.2	Saran	45
	DAFTAR REFERENSI	47
	A KODE PROGRAM	49
	B HASIL EKSPERIMEN	51

DAFTAR GAMBAR

2.1	Contoh masalah <i>linear programming</i> dengan daerah <i>feasible</i>	6
2.2	<i>Corner points</i> pada daerah <i>feasible</i>	7
2.3	Contoh masalah <i>linear programming</i> tanpa daerah <i>feasible</i>	7
2.4	Pemodelan daerah cakupan kamera	10
3.1	Pemodelan ruangan	14
3.2	Pemecahan ruangan menjadi matriks <i>cell</i>	14
3.3	Pemodelan kamera CCTV	15
3.4	Penempatan kamera CCTV dalam ruangan	16
3.5	Cakupan kamera CCTV yang terdiri dari kumpulan <i>cell</i>	16
3.6	<i>Overlap cell</i> dan <i>out of bound cell</i>	18
3.7	Diagram <i>use case</i>	20
3.8	Diagram kelas sederhana untuk <i>package</i> model	21
3.9	Diagram kelas sederhana untuk <i>package</i> bip dan <i>subpackage</i> <i>balasadditive</i>	22
4.1	Perancangan antarmuka penerima masukan	23
4.2	Perancangan antarmuka penempatan kamera CCTV	24
4.3	Diagram kelas rinci untuk <i>package</i> model	25
4.4	Diagram kelas rinci untuk <i>package</i> bip dan <i>subpackage</i> <i>balasadditive</i>	26
4.5	Diagram kelas <i>Angle</i>	27
4.6	Diagram kelas <i>Point</i>	28
4.7	Diagram kelas <i>CameraSpecification</i>	29
4.8	Diagram kelas <i>CameraPlacement</i>	29
4.9	Diagram kelas <i>Dimension</i>	30
4.10	Diagram kelas <i>Cell</i>	30
4.11	Diagram kelas <i>CellMatrix</i>	31
4.12	Diagram kelas <i>Room</i>	32
4.13	Diagram kelas <i>MinimumCameraPlacementSolver</i>	33
4.14	Diagram kelas <i>MinimumCameraPlacementSolverBalasAdditive</i>	34
4.15	Diagram kelas <i>Variable</i>	34
4.16	Diagram kelas <i>BIPFunction</i>	35
4.17	Diagram kelas <i>BIPConstraint</i>	35
4.18	Diagram kelas <i>BIPProblem</i>	36
4.19	Diagram kelas <i>BIPFeasibilityCheckResult</i>	36
4.20	Diagram kelas <i>Node</i>	37
4.21	Diagram kelas <i>NodeStatus</i>	38
4.22	Diagram kelas <i>BalasAdditiveBIPSolver</i>	38
4.23	Diagram kelas <i>BalasAdditiveBIPSolverResult</i>	39
5.1	Antarmuka penerima masukan	42
5.2	Antarmuka penempatan kamera CCTV	42
5.3	Diagram hubungan jumlah <i>cells</i> terhadap jumlah variabel	43

5.4	Diagram perbandingan jumlah iterasi antara <i>Balas's additive</i> dengan <i>exhaustive search</i> (2^n)	44
-----	--	----

DAFTAR TABEL

BAB 1

PENDAHULUAN

1.1 Latar Belakang

Kamera merupakan alat optik yang digunakan untuk mengambil gambar. Salah satu penggunaan kamera dalam kehidupan sehari-hari adalah kamera CCTV (*closed-circuit television*). Kamera CCTV digunakan untuk memantau suatu lokasi dengan tujuan pengawasan dan keamanan. Kamera CCTV pada umumnya dipasang di tempat yang strategis agar mendapat cakupan seefektif mungkin. Kamera CCTV bekerja dengan cara mengirimkan sinyal video menuju monitor khusus CCTV yang pada umumnya berada di tempat yang berbeda. Monitor ini akan dipantau oleh petugas keamanan sehingga petugas keamanan dapat memantau lokasi tersebut tanpa perlu mendatanginya secara langsung.

Penempatan kamera CCTV dalam ruangan yang berbentuk sederhana (persegi panjang) relatif tidak sulit. Kamera CCTV yang dibutuhkan pada umumnya berjumlah dua buah dan dipasang di kedua sudut ruangan yang merupakan satu diagonal sehingga saling berhadapan. Namun, jika ruangan berukuran besar, maka penempatan kamera CCTV dengan cara tersebut menjadi tidak efektif karena kamera CCTV tidak dapat memantau bagian ruangan yang berjarak terlalu jauh dari lokasi penempatan kamera CCTV. Untuk mengatasi masalah ini, pada ruangan tersebut dapat ditambahkan kamera-kamera CCTV hingga seluruh bagian dalam ruangan tercakup oleh kamera CCTV. Hal ini menyebabkan masalah lainnya, yaitu ketika menentukan jumlah minimum kamera CCTV yang dibutuhkan beserta dengan lokasi penempatannya. Apabila kamera CCTV yang dipasang berjumlah terlalu banyak, maka terdapat bagian ruangan yang setidaknya dicakup oleh 2 atau lebih kamera CCTV sehingga tidak efisien.

Pada penelitian ini, masalah tersebut akan dianalisis lebih lanjut dan diselesaikan. Solusi yang diharapkan dari masalah ini adalah penempatan-penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan. Terdapat metode yang digunakan untuk menyelesaikan masalah ini, yaitu dengan menggunakan metode *binary integer programming*. *Binary integer programming* merupakan metode untuk mendapatkan solusi terbaik dari suatu masalah dengan cara memodelkannya ke dalam bentuk matematika. Setelah memodelkan masalah ke dalam bentuk *binary integer programming*, hasil pemodelan tersebut akan diselesaikan menggunakan algoritma *Balas's additive*. *Balas's additive* merupakan algoritma yang dapat mencari solusi terbaik bagi masalah *binary integer programming* yang dilakukan secara efisien tanpa melibatkan *exhaustive search*. Dengan demikian, masalah ini dapat diselesaikan dan solusinya dapat ditemukan.

Pada penelitian ini juga akan dibangun sebuah perangkat lunak yang dapat menyelesaikan masalah tersebut. Analisis pemodelan dan penyelesaian masalah yang dilakukan akan diterapkan dalam perangkat lunak ini sehingga perangkat lunak dapat menerima masukan masalah dan menghasilkan solusinya. Perangkat lunak yang dibangun akan menggunakan tampilan antarmuka grafis dengan tujuan memvisualisasikan solusi penempatan kamera CCTV dalam ruangan. Dengan visualisasi, solusi penempatan kamera CCTV dalam ruangan akan lebih mudah untuk dipahami.

1.2 Rumusan Masalah

Berdasarkan latar belakang masalah yang telah dibahas sebelumnya, ditetapkan rumusan masalah sebagai berikut:

- Bagaimana cara memodelkan masalah ini ke dalam bentuk masalah *binary integer programming*?
- Bagaimana cara menyelesaikan masalah ini dalam bentuk masalah *binary integer programming* dengan menggunakan algoritma *Balas's additive*?
- Bagaimana cara membangun perangkat lunak yang dapat menyelesaikan masalah ini?

1.3 Tujuan

Berdasarkan rumusan masalah yang telah dijabarkan sebelumnya, ditetapkan tujuan penelitian sebagai berikut:

- Menganalisis cara memodelkan masalah ini ke dalam bentuk masalah *binary integer programming*.
- Menganalisis cara menyelesaikan masalah ini dalam bentuk masalah *binary integer programming* dengan menggunakan algoritma *Balas's additive*.
- Membangun perangkat lunak yang menerapkan analisis pemodelan dan penyelesaian masalah sehingga dapat menerima masukan masalah dan menyelesaikannya.

1.4 Batasan Masalah

Dalam pembahasan masalah ini, terdapat batasan-batasan sebagai berikut:

- Pemodelan masalah dilakukan pada bidang 2 dimensi.
- Kamera CCTV yang digunakan merupakan kamera statis.
- Ruang yang digunakan berbentuk persegi panjang yang terdiri dari ukuran panjang dan ukuran lebar. Kedua ukuran tersebut dinyatakan dalam satuan sentimeter (cm).
- Daerah cakupan kamera CCTV berbentuk sebagian lingkaran di mana lokasi penempatan kamera CCTV merupakan titik pusat sebagian lingkaran.
- Spesifikasi kamera CCTV yang digunakan terdiri dari jarak pandang dan besar sudut pandang. Keduanya digunakan untuk menentukan daerah cakupan kamera CCTV di mana jarak pandang menunjukkan jari-jari sebagian lingkaran dan sudut pandang menunjukkan besar sudut sebagian lingkaran. Parameter jarak pandang dinyatakan dalam satuan sentimeter (cm) dan parameter sudut pandang dinyatakan dalam satuan sudut derajat ($^{\circ}$).

1.5 Metodologi

Berikut ini merupakan langkah-langkah yang dalam melakukan penelitian ini:

1. Melakukan studi pustaka mengenai metode *linear programming* untuk memahami dasar *linear programming* sebelum mempelajari *binary integer programming*.

2. Melakukan studi pustaka mengenai metode *binary integer programming* untuk memodelkan masalah ini ke dalam bentuk masalah *binary integer programming*.
3. Melakukan studi pustaka mengenai algoritma *Balas's additive* untuk melakukan penyelesaian masalah *binary integer programming* menggunakan algoritma *Balas's additive*.
4. Melakukan analisis pemodelan masalah ke dalam bentuk *binary integer programming*.
5. Melakukan perancangan dan pengimplementasian perangkat lunak.
6. Melakukan pengujian perangkat lunak.
7. Membuat kesimpulan.

1.6 Sistematika Pembahasan

- **Bab 1 Pendahuluan**

Pada bagian ini dijelaskan latar belakang masalah yang diangkat dalam penelitian ini. Berdasarkan latar belakang masalah tersebut, ditentukan rumusan masalah dan tujuan dalam penelitian ini. Selain itu, terdapat batasan yang ada pada penelitian ini. Setiap langkah yang dilakukan dalam penelitian ini dibahas pada bagian metodologi.

- **Bab 2 Landasan Teori**

Pada bagian ini terdapat pembahasan mengenai teori-teori yang digunakan dalam penelitian ini. Terdapat teori *linear programming* dan teori *binary integer programming* yang digunakan untuk memodelkan masalah ini ke dalam bentuk masalah *binary integer programming*. Penyelesaian masalah *binary integer programming* akan dibahas pada bagian algoritma *Balas's additive*. Selain itu, terdapat juga penjelasan mengenai penelitian terkait yang telah dilakukan sebelumnya.

- **Bab 3 Analisis**

Pada bagian ini terdapat penjelasan mengenai analisis pemodelan masalah dan analisis penyelesaian masalah menggunakan metode *binary integer programming*. Selain itu, terdapat analisis kebutuhan perangkat lunak yang terdiri dari diagram *use case* dan diagram kelas sederhana.

- **Bab 4 Perancangan**

Pada bagian ini dibahas mengenai perancangan antarmuka dan perancangan kelas diagram rinci yang akan digunakan untuk membangun perangkat lunak.

- **Bab 5 Implementasi dan Pengujian**

Pada bagian ini akan dibahas mengenai hasil implementasi dan pengujian perangkat lunak yang dilakukan.

- **Bab 6 Kesimpulan dan Saran**

Pada bagian ini terdapat kesimpulan yang dihasilkan melalui penelitian ini beserta dengan saran untuk penelitian selanjutnya.

BAB 2

LANDASAN TEORI

2.1 *Linear Programming*

Linear programming adalah sarana untuk menyelesaikan masalah optimasi [1]. Optimasi merupakan usaha untuk memilih solusi terbaik dari suatu kumpulan solusi yang ada. Setiap masalah optimasi perlu diubah terlebih dahulu ke dalam bentuk masalah *linear programming* agar dapat diselesaikan menggunakan *linear programming*. Pada bagian ini, akan dibahas karakteristik dan cara menyelesaikan masalah *linear programming*.

2.1.1 Karakteristik

Masalah *linear programming* memiliki karakteristik sebagai berikut:

- **Variabel keputusan**

Dalam masalah *linear programming*, terdapat variabel keputusan yang mendeskripsikan keputusan yang akan diambil. Contoh:

$$\begin{aligned}x_1 &= \text{jumlah produk A yang diproduksi} \\x_2 &= \text{jumlah produk B yang diproduksi}\end{aligned}\tag{2.1}$$

- **Fungsi Tujuan**

Dalam masalah *linear programming*, terdapat suatu keuntungan yang ingin dimaksimalkan atau suatu kerugian yang ingin diminimalkan dengan menggunakan fungsi tujuan yang terdiri dari variabel-variabel keputusan. Contoh:

$$\max z = 3x_1 + 2x_2\tag{2.2}$$

- **Batasan**

Dalam masalah *linear programming*, terdapat batasan-batasan yang membuat variabel keputusan memiliki nilai yang dibatasi. Batasan juga membuat suatu variabel keputusan berkaitan dengan variabel keputusan lainnya. Contoh:

$$\begin{aligned}2x_1 + x_2 &\leq 100 \\x_1 + x_2 &\leq 80 \\x_1 &\leq 40\end{aligned}\tag{2.3}$$

- **Batasan non-negatif**

Dalam masalah *linear programming*, batasan non-negatif ($x_i \geq 0$) pada variabel keputusan menyatakan bahwa variabel keputusan tersebut tidak dapat bernilai negatif. Dengan batasan

non-negatif, variabel keputusan diharuskan untuk bernilai 0 atau positif. Contoh:

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned} \tag{2.4}$$

Dengan keempat karakteristik di atas, setiap masalah optimasi dapat diubah ke dalam masalah *linear programming* untuk diselesaikan. Berikut contoh masalah *linear programming*:

$$\begin{aligned} \max \quad & z = 3x_1 + 2x_2 \\ \text{s.t.} \quad & 2x_1 + x_2 \leq 100 \\ & x_1 + x_2 \leq 80 \\ & x_1 \leq 40 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{aligned} \tag{2.5}$$

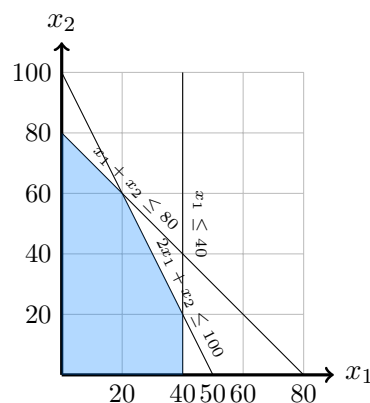
Tulisan "*s.t.*" atau "*subject to*" menandakan bahwa nilai dari setiap variabel keputusan harus memenuhi setiap batasan dan batasan non-negatif.

2.1.2 Daerah *Feasible* dan Solusi Optimal

Dalam masalah *linear programming*, terdapat daerah yang bernama daerah *feasible*. Daerah *feasible* dalam suatu masalah *linear programming* merupakan himpunan yang terdiri dari seluruh titik yang memenuhi setiap batasan dan batasan non-negatif [1]. Dengan demikian, setiap titik yang berada di dalam daerah *feasible* merupakan solusi terhadap permasalahan tersebut. Apabila dipilih suatu titik yang berada di luar daerah *feasible*, maka titik ini disebut dengan titik *infeasible*. Titik *infeasible* bukan merupakan solusi bagi permasalahan karena titik ini tidak memenuhi setiap batasan dan batasan non-negatif. Untuk menggambarkan daerah *feasible*, digunakan contoh batasan-batasan sebagai berikut:

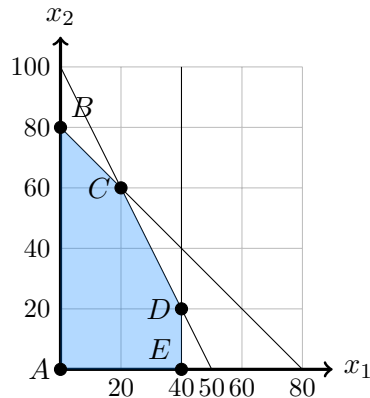
$$\begin{aligned} 2x_1 + x_2 &\leq 100 \\ x_1 + x_2 &\leq 80 \\ x_1 &\leq 40 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned} \tag{2.6}$$

Pada contoh masalah tersebut, hanya terdapat 2 variabel keputusan sehingga dapat digambarkan dalam diagram kartesius. Pada gambar 2.1 terlihat bahwa batasan-batasan membentuk daerah *feasible*.



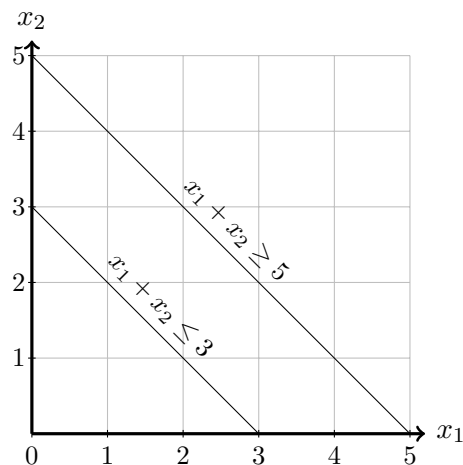
Gambar 2.1: Contoh masalah *linear programming* dengan daerah *feasible*

Agar suatu masalah *linear programming* memiliki solusi optimal, maka daerah *feasible* harus berbentuk *convex set*. Suatu himpunan titik S merupakan *convex set* apabila segmen garis yang menghubungkan setiap pasang titik dalam himpunan S berada dalam himpunan S [1]. Apabila masalah *linear programming* memiliki daerah *feasible*, maka daerah tersebut berbentuk *convex set*. Solusi optimal dari masalah tersebut berada pada salah satu *corner point* pada daerah *feasible*. *Extreme point* adalah titik perpotongan antar batasan dan *corner point* adalah *extreme point* yang berada dalam daerah *feasible*. Dengan adanya *corner points*, solusi optimal dapat dicari karena berjumlah terhingga, sehingga tidak perlu memeriksa seluruh kemungkinan solusi masalah yang berjumlah tak hingga. Pada gambar 2.2, titik A, B, C, D, dan E merupakan titik *corner points* yang salah satu di antaranya akan menghasilkan solusi optimal.



Gambar 2.2: *Corner points* pada daerah *feasible*

Tidak semua masalah *linear programming* memiliki daerah *feasible*. Apabila batasan-batasan dalam masalah tidak dapat membentuk daerah *feasible*, maka masalah tersebut tidak memiliki solusi apapun, sehingga solusi optimal dari masalah tersebut tidak dapat dicari. Gambar 2.3 menunjukkan contoh masalah *linear programming* yang tidak memiliki daerah *feasible*.



Gambar 2.3: Contoh masalah *linear programming* tanpa daerah *feasible*

2.2 Binary Integer Programming

Binary integer programming adalah lanjutan dari *linear programming* (2.1) di mana setiap variabel keputusan pada solusi optimal harus bernilai 0 atau 1. Bentuk umum masalah *binary integer*

programming adalah seperti berikut:

$$\begin{aligned} \min \quad & z = C^T x \\ \text{s.t.} \quad & Ax \geq b \\ & x \in \{0, 1\} \end{aligned} \tag{2.7}$$

Terdapat sebuah algoritma yang diciptakan oleh Egon Balas [2] bernama *Balas's additive* yang digunakan untuk menyelesaikan masalah *binary integer programming*. Algoritma *Balas's additive* dibahas lebih lanjut pada 2.2.1.

2.2.1 Algoritma *Balas's Additive*

Setiap masalah *binary integer programming* memiliki kemungkinan solusi berjumlah 2^n karena setiap variabel hanya dapat bernilai 0 atau 1 saja. Apabila menggunakan algoritma *Balas's additive*, maka solusi optimal dari masalah *binary integer programming* dapat ditemukan tanpa perlu memeriksa seluruh 2^n kemungkinan. Dengan demikian, algoritma *Balas's additive* dapat digunakan untuk menemukan solusi optimal bagi masalah *binary integer programming* secara efisien.

Untuk menyelesaikan masalah *binary integer programming* menggunakan algoritma *Balas's additive*, maka masalah perlu diubah terlebih dahulu berdasarkan ketentuan-ketentuan berikut ini:

- Fungsi tujuan dinyatakan dalam bentuk $\min z = \sum_{j=1}^n c_j x_j$.
- Sebanyak m batasan harus dinyatakan dalam bentuk pertidaksamaan $\sum a_{ij} x_j \geq b_i$ untuk $i = 1, 2, \dots, m$.
- Setiap variabel x_j dimana $j = 1, 2, \dots, n$ harus merupakan variabel biner (variabel yang hanya dapat bernilai 0 atau 1).
- Koefisien dari setiap variabel pada fungsi tujuan tidak bernilai negatif.
- Setiap variabel pada fungsi tujuan diurutkan menaik berdasarkan koefisiennya sehingga $0 \leq c_1 \leq c_2 \leq \dots \leq c_n$.

Pengurutan variabel secara menaik berdasarkan koefisiennya bertujuan agar nilai z meningkat seminimum mungkin pada setiap iterasinya. Mulanya algoritma *Balas's additive* akan membuat seluruh variabel bernilai 0 agar mendapatkan nilai z yang paling minimum. Apabila solusi bersifat *infeasible*, maka algoritma secara beriterasi akan membuat variabel-variabel terdekat dengan indeks 1 untuk bernilai 1. Dengan demikian, nilai z akan meningkat dengan nilai seminimum mungkin pada setiap iterasinya karena variabel telah terurut menaik berdasarkan koefisiennya.

Algoritma *Balas's additive* menggunakan paradigma algoritma *branch and bound* [3] untuk mendapatkan solusi optimal dari seluruh kemungkinan solusi yang ada. *Branch and bound* merupakan algoritma yang dapat mencari solusi terbaik dari suatu himpunan solusi tanpa melibatkan *exhaustive search*. *Branch and bound* bekerja dengan cara membagi dan memeriksa subhimpunan solusi secara rekursi hingga mendapatkan subhimpunan yang memiliki solusi yang paling baik. Setiap subhimpunan memiliki nilai *bound* yang menyatakan kualitas dari solusi yang dimiliki oleh masing-masing subhimpunan. Nilai *bound* ini menjadi pembanding sebelum melakukan pembagian subhimpunan. Apabila nilai *bound* suatu subhimpunan lebih buruk daripada solusi yang telah ditemukan sebelumnya, maka pembagian subhimpunan tidak dilakukan karena telah dipastikan bahwa subhimpunannya tidak akan dapat menghasilkan solusi yang lebih baik. *Branch and bound* menggunakan struktur data *tree* di mana *root* menunjukkan himpunan keseluruhan solusi dan cabang menunjukkan subhimpunannya. Terdapat 2 proses utama dalam *branch and bound*, yaitu *branching* dan *bounding*. *Branching* berfungsi untuk membagi himpunan solusi menjadi beberapa subhimpunan yang lebih kecil. *Bounding* berfungsi untuk menetapkan nilai *bound* dari suatu subhimpunan.

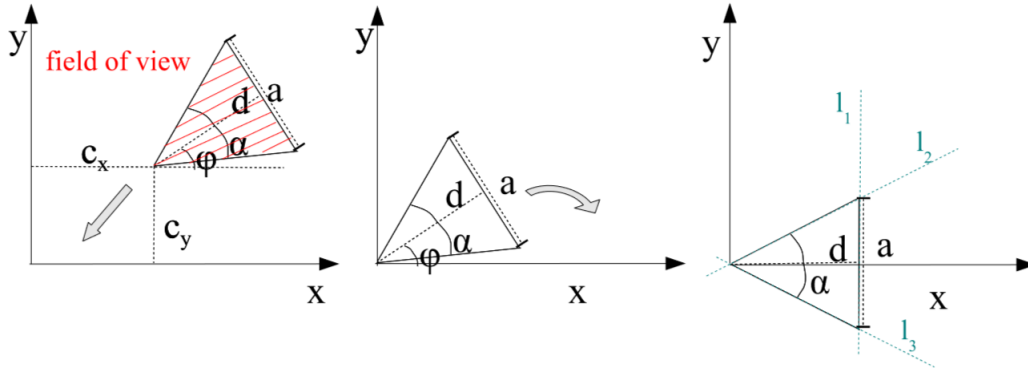
Branch and bound pada algoritma *Balas's additive* menggunakan struktur data *binary tree* karena setiap percabangan menghasilkan 2 *node* di mana salah satu *node* menunjukkan variabel bernilai 0 dan *node* lainnya menunjukkan variabel bernilai 1. Setiap *node* menunjukkan nilai dari variabel yang ditujunya sesuai dengan tingkat kedalaman *node* pada pohon biner. *Branching* terhadap suatu *node* akan menghasilkan 2 *node* baru, yaitu *node* $x_N = 0$ dan *node* $x_N = 1$ di mana x_N merupakan variabel x yang diacu oleh *node* pada saat ini yang berbeda dengan variabel terakhir (x_n) dalam kumpulan x . *Bounding* pada algoritma *Balas's additive* dibedakan menjadi 2 berdasarkan nilai x_N , yaitu:

- Apabila $x_N = 1$, maka algoritma berasumsi bahwa *node* pada saat ini dapat menghasilkan solusi *feasible* sehingga nilai *bound* untuk *node* ini adalah $\sum_{j=1}^N c_j x_j$. Karena variabel telah diurutkan sebelumnya, maka nilai *bound* ini dipastikan menjadi yang paling rendah untuk saat ini.
- Apabila $x_N = 0$, maka nilai *bound* dari *node* ini adalah $\sum_{j=1}^N c_j x_j + c_{N+1}$. *Node* ini terbentuk karena *node* sebelumnya memiliki solusi yang bersifat *infeasible*. Hal tersebut disebabkan oleh adanya batasan \geq yang tidak terpenuhi karena total ruas kiri yang lebih kecil daripada ruas kanan. Apabila variabel saat ini bernilai 0, maka batasan tersebut tetap tidak akan terpenuhi sehingga setidaknya harus terdapat 1 variabel lainnya yang bernilai 1, yaitu x_{N+1} . Variabel tersebut dipilih untuk memastikan bahwa nilai *bound* pada saat ini merupakan yang paling rendah.

Setiap *node* menunjukkan solusi yang belum tentu bersifat *feasible* sehingga perlu dilakukannya pengecekan. Untuk menentukan apakah solusi bersifat *feasible*, maka akan diasumsikan bahwa variabel setelah x_N (saat $x_N = 1$) atau x_{N+1} (saat $x_N = 0$) bernilai 0. Selanjutnya, solusi ini akan diuji terhadap setiap batasan. Apabila tidak terdapat batasan yang dilanggar, maka solusi bersifat *feasible* dan *node* dinyatakan *fathomed* sehingga tidak perlu dilakukan *branching*. *Branching* tidak perlu dilakukan karena turunannya tidak akan menghasilkan solusi yang lebih baik. Hal ini telah dipastikan ketika melakukan proses *bounding*. Solusi *feasible* ini kemudian dibandingkan dengan *incumbent* yang telah ditemukan sebelumnya. Apabila lebih baik, maka solusi ini menjadi *incumbent* yang baru. Apabila *node* memiliki solusi yang bersifat *infeasible*, maka akan dilakukan *branching*. Namun sebelum melakukan *branching*, akan dilakukan pengecekan untuk mengetahui apakah turunan dari *node* dapat menghasilkan solusi *feasible*. Pengecekan dilakukan dengan memastikan bahwa setiap batasan memiliki kemungkinan untuk dipenuhi, yaitu dengan memastikan bahwa ruas kiri dapat menghasilkan total yang tidak lebih kecil daripada ruas kanan. Total ruas kiri terbesar dapat dicari dengan menjumlahkan ruas kiri untuk variabel yang telah didapatkan hingga saat ini dengan total penjumlahan nilai koefisien positif pada variabel tersisa. Apabila terdapat 1 atau lebih batasan di mana total ruas kiri terbesar bernilai lebih kecil dibandingkan dengan nilai ruas kanan, maka *node* tersebut dinyatakan *fathomed* sehingga tidak perlu dilakukan *branching*. *Branching* tidak perlu dilakukan karena batasan tersebut tidak akan dapat dipenuhi dengan berapa pun nilai dari variabel yang tersisa. Sebaliknya, apabila setiap batasan memiliki kemungkinan untuk dipenuhi, maka *branching* akan dilakukan pada *node* tersebut dan seluruh proses sebelumnya akan kembali dilakukan.

2.3 Penelitian Terkait

Penelitian untuk mencari penempatan sensor visual secara optimal telah dilakukan sebelumnya oleh Horster dan Lienhart [4]. Masalah yang dibahas pada penelitian mereka adalah cara menentukan jumlah sensor visual minimum beserta dengan lokasi penempatan dan arah pandangnya. Masalah ini dibatasi pada bidang 2D dan ruangan yang digunakan diasumsikan berbentuk persegi panjang.



Gambar 2.4: Pemodelan daerah cakupan kamera

Pada penelitian ini, daerah cakupan kamera dimodelkan dalam bentuk segitiga seperti pada gambar 2.4. Kamera ditempatkan pada posisi c_x, c_y dan menghadap ke arah φ . Untuk mendapatkan daerah cakupan kamera, mulanya kamera ditranslasi ke titik *origin* pada sistem koordinat:

$$x' = x - c_x \quad (2.8)$$

$$y' = y - c_y \quad (2.9)$$

Selanjutnya, daerah cakupan diputar sehingga sumbu pandang menjadi paralel terhadap sumbu x:

$$x'' = \cos(\varphi) \cdot x' + \sin(\varphi) \cdot y' \quad (2.10)$$

$$y'' = -\sin(\varphi) \cdot x' + \cos(\varphi) \cdot y' \quad (2.11)$$

Daerah cakupan kamera dapat ditentukan dengan garis l_1, l_2, l_3 :

$$l_1 : x'' \leq d \quad (2.12)$$

$$l_2 : y'' \leq \frac{a}{2d} \cdot x'' \quad (2.13)$$

$$l_3 : y'' \geq -\frac{a}{2d} \cdot x'' \quad (2.14)$$

Dengan substitusi, didapatkan daerah cakupan kamera berdasarkan tiga persamaan berikut ini:

$$\cos(\varphi) \cdot (x - c_x) + \sin(\varphi) \cdot (y - c_y) \leq d \quad (2.15)$$

$$-\sin(\varphi) \cdot (x - c_x) + \cos(\varphi) \cdot (y - c_y) \leq \frac{a}{2d} \cdot (\cos(\varphi) \cdot (x - c_x) + \sin(\varphi) \cdot (y - c_y)) \quad (2.16)$$

$$-\sin(\varphi) \cdot (x - c_x) + \cos(\varphi) \cdot (y - c_y) \geq -\frac{a}{2d} \cdot (\cos(\varphi) \cdot (x - c_x) + \sin(\varphi) \cdot (y - c_y)) \quad (2.17)$$

Ruangan dimodelkan menggunakan *grid point* 2 dimensi dengan jarak antar titik sebesar f_a . Kamera hanya dapat diletakkan pada titik *grid point* dan cakupan kamera hanya diperiksa terhadap titik *grid point*. Ruang berbentuk persegi panjang sehingga ukuran ruangan terdiri dari lebar w dan tinggi h . Di dalam ruangan tidak terdapat penghalang apapun.

Titik penempatan kamera dinyatakan dalam himpunan s_x dan himpunan s_y sesuai dengan dimensi x - dan y - pada *grid point*. Selain itu, terdapat himpunan s_φ yang menyatakan arah pandang yang memungkinkan bagi setiap kamera. Kamera yang ditempatkan pada (c_x, c_y) dengan

arah pandang φ dapat mencakup *grid point* (x, y) jika pernyataan 2.15, 2.16, dan 2.17 dipenuhi.

Masalah dimodelkan ke dalam bentuk masalah *binary integer programming* dengan tujuan mendapatkan jumlah kamera minimum berdasarkan *grid point* dan model kamera yang diberikan dengan tetap memastikan bahwa setiap titik pada *grid point* tercakup oleh setidaknya satu kamera. Berikut ini merupakan definisi variabel biner $x_{ij\varphi}$ yang digunakan dalam model masalah *binary integer programming*:

$$x_{ij\varphi} = \begin{cases} 1 & \text{jika kamera ditempatkan pada } \textit{grid point} (i, j) \text{ dengan arah pandang } \varphi \\ 0 & \text{jika sebaliknya} \end{cases} \quad (2.18)$$

Total kamera (N) didapatkan dengan fungsi tujuan berikut ini:

$$N = \sum_{\varphi=0}^{s_{\varphi}-1} \sum_{i=0}^{s_i-1} \sum_{j=0}^{s_j-1} x_{ij\varphi} \quad (2.19)$$

Untuk menentukan ketercakupan titik, dibentuk fungsi biner $c(i1, j1, \varphi1, i2, j2)$:

$$c(i1, j1, \varphi1, i2, j2) = \begin{cases} 1 & \text{jika kamera yang ditempatkan pada } \textit{grid point} (i1, j1) \\ & \text{dengan arah pandang } \varphi1 \text{ dapat mencakup } \textit{grid point} (i2, j2) \\ 0 & \text{jika sebaliknya} \end{cases} \quad (2.20)$$

Dengan demikian masalah dapat diformulasikan ke dalam bentuk masalah *binary integer programming* berikut:

$$\begin{aligned} \min \quad & \sum_{\varphi=0}^{s_{\varphi}-1} \sum_{i=0}^{s_i-1} \sum_{j=0}^{s_j-1} x_{ij\varphi} \\ \text{s.t.} \quad & \sum_{\varphi1=0}^{s_{\varphi}-1} \sum_{i1=0}^{s_i-1} \sum_{j1=0}^{s_j-1} x_{i1,j1,\varphi1} \cdot c(i1, j1, \varphi1, i2, j2) \geq 1 \\ & 0 \leq i2 \leq (s_x - 1), 0 \leq j2 \leq (s_y - 1) \end{aligned} \quad (2.21)$$

Batasan pada 2.21 memastikan bahwa setiap titik pada *grid point* akan dicakup oleh setidaknya 1 kamera. Untuk memastikan bahwa hanya terdapat maksimal 1 kamera yang dapat ditempatkan pada setiap titik, maka dapat ditambahkan batasan berikut ini:

$$\begin{aligned} & \sum_{\varphi=0}^{s_{\varphi}-1} x_{ij\varphi} \leq 1 \\ & 0 \leq i \leq (s_x - 1), 0 \leq j \leq (s_y - 1) \end{aligned} \quad (2.22)$$

Jumlah variabel $x_{ij\varphi}$ dalam model *binary integer programming* ini adalah sebesar $s_x \times s_y \times s_{\varphi}$. Jika ukuran *grid point* diperbesar, maka jumlah variabel dan batasan pada model *binary integer programming* juga akan semakin besar. Karena masalah ini merupakan masalah *binary integer programming* (2.2), maka masalah ini dapat diselesaikan dengan menggunakan algoritma *Balas's additive* (2.2.1).

BAB 3

ANALISIS

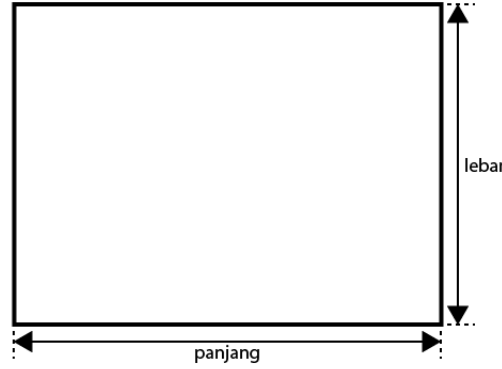
Pada bagian ini, masalah akan dianalisis lebih lanjut. Awalnya, masalah akan dimodelkan terlebih dahulu sebelum diselesaikan. Hasil dari pemodelan masalah akan digunakan untuk membentuk model masalah *binary integer programming* (2.2) yang akan diselesaikan menggunakan algoritma *Balas's additive* (2.2.1). Dengan model masalah *binary integer programming*, solusi optimal masalah dapat ditemukan, yaitu solusi berupa penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan. Selanjutnya, pemodelan dan penyelesaian tersebut akan digunakan untuk merancang perangkat lunak yang dapat menerima masalah ini dan menyelesaikannya. Setiap kebutuhan perangkat lunak yang terdiri dari masukan, keluaran, dan kelas-kelas akan dibahas lebih lanjut pada bagian analisis kebutuhan perangkat lunak.

3.1 Pemodelan Masalah

Masalah yang dibahas pada penelitian ini perlu dimodelkan terlebih dahulu agar menjadi konkret. Pemodelan masalah terdiri dari pemodelan ruangan, kamera CCTV, dan cakupan kamera CCTV. Pemodelan yang dilakukan pada penelitian sebelumnya (2.3) akan diterapkan kembali pada penelitian ini dengan adanya modifikasi. Modifikasi dilakukan pada pemodelan ruangan dan pemodelan cakupan kamera CCTV. Sebelumnya, ruangan dimodelkan menggunakan *grid point* sehingga suatu bagian daerah dalam ruangan dinyatakan dalam bentuk titik. Sedangkan pada penelitian ini, suatu bagian dalam ruangan dinyatakan dalam bentuk *cell*. Modifikasi pemodelan ruangan turut disertai dengan modifikasi pemodelan daerah cakupan kamera CCTV. Sebelumnya, cakupan kamera CCTV terdiri dari kumpulan titik. Namun, karena suatu daerah dalam ruangan dinyatakan dalam bentuk *cell*, maka cakupan kamera CCTV pada penelitian ini berubah sehingga terdiri dari kumpulan *cell*.

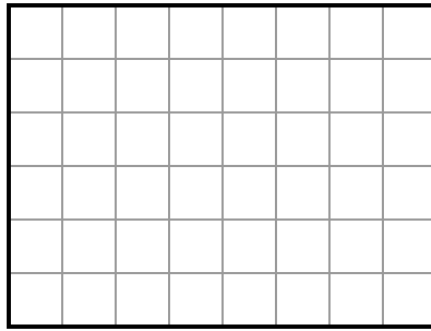
3.1.1 Ruangan

Bentuk ruangan pada masalah ini dibatasi sehingga berbentuk persegi panjang dalam bidang 2 dimensi. Karena berbentuk persegi panjang, maka ruangan terdiri dari ukuran panjang dan lebar seperti pada gambar 3.1. Kedua ukuran ini menggunakan satuan ukuran sentimeter (cm).



Gambar 3.1: Pemodelan ruangan

Ruangan akan dipecah menjadi matriks *cell* yang berukuran lebih kecil seperti pada gambar 3.2. Hal ini bertujuan agar daerah dalam ruangan dinyatakan dalam kumpulan daerah yang lebih kecil. Setiap bagian daerah yang lebih kecil tersebut disebut sebagai *cell*. *Cell* juga menyatakan bagian daerah terkecil yang tidak dapat dibagi lagi.

Gambar 3.2: Pemecahan ruangan menjadi matriks *cell*

Ukuran *cell* dapat ditentukan berdasarkan ukuran daerah terkecil. Namun, apabila *cell* diharuskan berbentuk persegi, maka terdapat kemungkinan di mana susunan *cell* tidak dapat membentuk ukuran ruangan yang utuh. Untuk menyiasati hal tersebut, *cell* dibuat dapat berbentuk persegi panjang dan ukurannya didapatkan berdasarkan suatu ukuran. Ukuran tersebut adalah ukuran terbesar *cell* yang menyatakan ukuran terbesar yang dapat dimiliki *cell* sehingga ukuran panjang dan lebarnya tidak melebihi ukuran ini. Terdapat perhitungan yang dilakukan untuk mendapatkan ukuran *cell* berdasarkan ukuran terbesar *cell*. Didefinisikan variabel sebagai berikut:

$$\begin{aligned}
 l &: \text{panjang ruangan} \\
 w &: \text{lebar ruangan} \\
 l_c &: \text{panjang cell} \\
 w_c &: \text{lebar cell} \\
 m &: \text{ukuran terbesar cell}
 \end{aligned} \tag{3.1}$$

Selanjutnya, akan dicari ukuran matriks berdasarkan ukuran ruangan dan ukuran terbesar *cell*:

$$\begin{aligned}
 &cols : \text{jumlah kolom pada matriks } cell \\
 &rows : \text{jumlah baris pada matriks } cell \\
 &cols = \text{ceil} \left(\frac{l}{m} \right) \\
 &rows = \text{ceil} \left(\frac{w}{m} \right)
 \end{aligned} \tag{3.2}$$

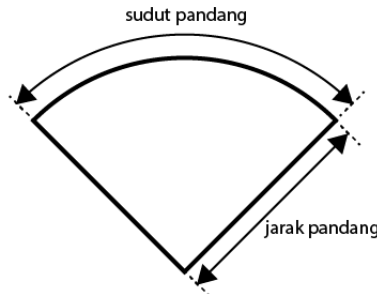
Setelah mendapatkan ukuran matriks, ukuran ruangan akan dibagi berdasarkan ukuran matriks:

$$\begin{aligned}
 l_c &= \frac{l}{cols} \\
 w_c &= \frac{w}{rows}
 \end{aligned} \tag{3.3}$$

Dengan demikian, didapatkan ukuran *cell* berdasarkan ukuran terbesar *cell*. Ukuran ini merupakan ukuran terbesar yang tidak melebihi ukuran terbesar *cell*.

3.1.2 Kamera CCTV

Kamera CCTV pada masalah ini dimodelkan dalam bentuk sebagian lingkaran pada bidang 2 dimensi. Kamera CCTV memiliki parameter spesifikasi yang beragam. Namun, pada masalah ini, hanya digunakan 2 parameter spesifikasi, yaitu jarak pandang dan besar sudut pandang seperti pada gambar 3.3. Kedua parameter ini menentukan daerah yang dapat dicakup oleh kamera CCTV. Jarak pandang dinyatakan dalam satuan ukuran sentimeter (cm) dan besar sudut pandang dinyatakan dalam satuan ukuran derajat ($^{\circ}$).



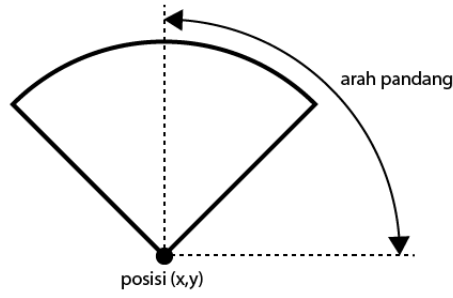
Gambar 3.3: Pemodelan kamera CCTV

Penempatan kamera CCTV dalam ruangan terdiri dari 2 bagian, yaitu posisi penempatan dan sudut arah pandang seperti pada gambar 3.4. Posisi penempatan kamera CCTV dinyatakan dalam titik koordinat kartesius 2 dimensi (x, y) . Sudut arah pandang menunjukkan arah yang dituju kamera CCTV. Arah pandang kamera CCTV dinyatakan dalam jarak sudut derajat ($^{\circ}$) antara arah yang dituju dengan garis 0° .

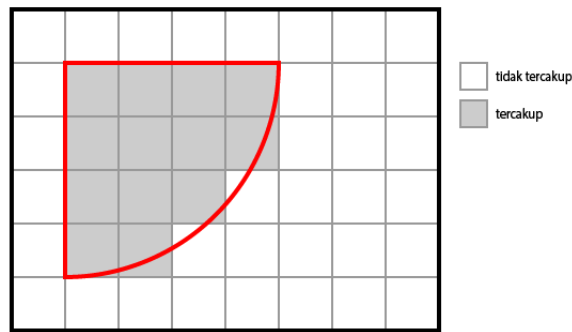
3.1.3 Cakupan Kamera CCTV

Dengan pemecahan ruangan ke dalam matriks *cell*, maka cakupan kamera CCTV terdiri dari kumpulan *cell*. Contoh cakupan kamera CCTV dapat dilihat pada gambar 3.5.

Setiap *cell* memiliki titik tengah yang berada di tengah *cell*. Titik tengah digunakan untuk menentukan ketercakupan *cell* oleh suatu penempatan kamera CCTV. Untuk menentukan apakah suatu *cell* dapat dicakup oleh suatu penempatan kamera CCTV, maka perlu dilakukan 2 jenis pengecekan. Pengecekan pertama dilakukan untuk memastikan bahwa jarak titik tengah *cell*



Gambar 3.4: Penempatan kamera CCTV dalam ruangan

Gambar 3.5: Cakupan kamera CCTV yang terdiri dari kumpulan *cell*

terhadap titik penempatan kamera CCTV lebih kecil atau sama dengan jarak pandang kamera CCTV. Untuk melakukannya, terlebih dahulu didefinisikan variabel sebagai berikut:

$$\begin{aligned}
 (x_{cam}, y_{cam}) &: \text{titik penempatan kamera CCTV} \\
 (x_{cell}, y_{cell}) &: \text{titik tengah } cell \\
 r &: \text{jarak pandang kamera CCTV}
 \end{aligned}
 \tag{3.4}$$

Pengecekan pertama dinyatakan berhasil apabila memenuhi pernyataan berikut:

$$\sqrt{(x_{cam} - x_{cell})^2 + (y_{cam} - y_{cell})^2} \leq r
 \tag{3.5}$$

Pengecekan kedua dilakukan untuk memastikan bahwa titik tengah *cell* berada di antara sudut

pandang kamera CCTV. Untuk melakukannya, didefinisikan variabel dan fungsi sebagai berikut:

$$\begin{aligned}
 &\alpha : \text{besar sudut pandang kamera CCTV} \\
 &\beta : \text{sudut arah pandang kamera CCTV} \\
 &\text{norm}(\theta) : \text{fungsi untuk menormalkan sudut sehingga } \theta \text{ berada dalam rentang } [0, 2\pi) \\
 &\text{atan2}(x, y) : \text{fungsi untuk mendapatkan sudut rotasi titik } (x, y) \text{ terhadap titik O} \\
 &\text{norm}(\theta) = \begin{cases} \text{norm}(\theta + 2\pi) & \text{jika } \theta < 0 \\ \text{norm}(\theta - 2\pi) & \text{jika } \theta \geq 2\pi \\ \theta & \text{jika } \theta \geq 0 \text{ dan } \theta < 2\pi \end{cases} \\
 &\text{atan2}(x, y) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{jika } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{jika } x < 0 \text{ dan } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{jika } x < 0 \text{ dan } y < 0 \\ +\frac{\pi}{2} & \text{jika } x = 0 \text{ dan } y > 0 \\ -\frac{\pi}{2} & \text{jika } x = 0 \text{ dan } y < 0 \\ \text{tak terdefinisi} & \text{jika } x = 0 \text{ dan } y = 0 \end{cases}
 \end{aligned} \tag{3.6}$$

Selanjutnya, akan dicari sudut pandang awal, sudut pandang akhir, dan sudut rotasi titik tengah *cell* terhadap titik penempatan kamera CCTV:

$$\begin{aligned}
 &\alpha_{half} : \text{setengah dari besar sudut pandang kamera CCTV} \\
 &\beta_{start} : \text{sudut pandang awal kamera CCTV} \\
 &\beta_{end} : \text{sudut pandang akhir kamera CCTV} \\
 &\beta_{cell} : \text{sudut rotasi titik tengah } cell \text{ terhadap titik penempatan kamera CCTV} \\
 &\alpha_{half} = \frac{\alpha}{2} \\
 &\beta_{start} = \text{norm}(\beta - \alpha_{half}) \\
 &\beta_{end} = \text{norm}(\beta + \alpha_{half}) \\
 &\beta_{cell} = \text{atan2}((y_{cell} - y_{cam}), (x_{cell} - x_{cam}))
 \end{aligned} \tag{3.7}$$

Pengecekan kedua dibedakan berdasarkan sudut pandang awal dan sudut pandang akhir. Apabila sudut pandang awal lebih kecil daripada sudut pandang akhir ($\beta_{start} < \beta_{end}$), maka kedua pernyataan berikut harus dipenuhi agar pengecekan kedua dinyatakan berhasil:

$$\begin{aligned}
 &\beta_{start} < \beta_{cell} \\
 &\beta_{cell} < \beta_{end}
 \end{aligned} \tag{3.8}$$

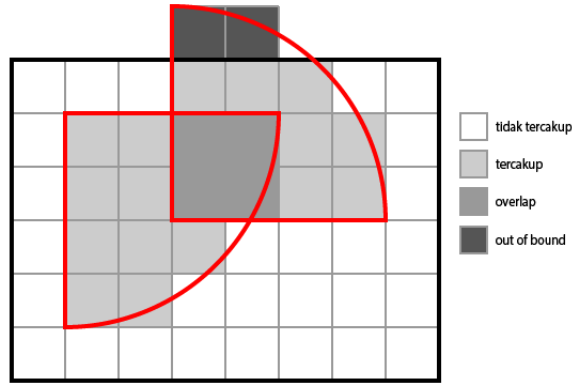
Apabila sudut pandang awal lebih besar atau sama dengan sudut pandang akhir ($\beta_{start} \geq \beta_{end}$), maka minimal satu dari kedua pernyataan berikut harus dipenuhi agar pengecekan kedua dinyatakan berhasil:

$$\begin{aligned}
 &\beta_{start} < \beta_{cell} \\
 &\beta_{cell} < \beta_{end}
 \end{aligned} \tag{3.9}$$

Dengan memenuhi pengecekan pertama dan kedua, maka *cell* dinyatakan tercakup oleh penempatan kamera CCTV.

Perhitungan tingkat *overlap* dan *out of bound* dapat dilakukan dengan membandingkan jumlah *cell*. *Overlap cell* adalah *cell* yang dicakup oleh lebih dari 1 kamera CCTV, sedangkan *out of bound cell* adalah *cell* di luar ruangan yang tercakup oleh kamera CCTV. Gambar 3.6 menggambarkan penempatan kamera CCTV yang menghasilkan *overlap cell* dan *out of bound cell*. Tingkat *overlap* dan *out of bound* dapat dihitung dengan membandingkan jumlah *overlap cell* dan *out of bound cell* dengan total *cell* dalam ruangan. Perhitungan tingkat *overlap* dan *out of bound* hanya dilakukan

apabila seluruh *cell* dalam ruangan telah tercakup oleh kamera CCTV.



Gambar 3.6: *Overlap cell* dan *out of bound cell*

3.2 Penyelesaian Masalah

Setelah memodelkan masalah, masalah akan dimodelkan ke dalam bentuk masalah *binary integer programming*. Solusi yang diharapkan dari masalah ini adalah penempatan-penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan. Berdasarkan pemodelan masalah, ruangan telah dibagi ke dalam matriks *cell* sehingga solusi penempatan kamera CCTV harus mencakup seluruh *cell* dalam ruangan. Untuk mendapatkan solusi tersebut, akan dibangun seluruh kemungkinan penempatan kamera CCTV. Dari setiap kemungkinan tersebut, akan dipilih penempatan-penempatan kamera CCTV berjumlah minimum yang dapat mencakup seluruh *cell* dalam ruangan. Pemilihan dapat dilakukan dengan memeriksa seluruh kombinasi kemungkinan penempatan hingga didapatkan kombinasi yang sesuai dengan kriteria solusi. Namun, metode ini bukan merupakan metode efisien karena bersifat *exhaustive search* sehingga dibutuhkan metode lainnya yang dapat mencari kombinasi penempatan kamera CCTV tanpa melibatkan *exhaustive search*. Salah satu metodenya adalah memodelkan masalah ke dalam bentuk masalah *binary integer programming* dan menyelesaikannya menggunakan algoritma *Balas's additive*. Pada 2.2.1 telah dibahas bagaimana algoritma ini dapat menyelesaikan masalah *binary integer programming* secara efisien tanpa melibatkan *exhaustive search*. Dengan demikian, solusi dari masalah ini dapat ditemukan dengan memodelkan masalah ke dalam bentuk masalah *binary integer programming* dan menyelesaikannya menggunakan algoritma Balas's additive.

3.2.1 Variabel

Variabel dalam model masalah *binary integer programming* ini terdiri dari seluruh kemungkinan penempatan kamera CCTV. Penempatan kamera CCTV dapat dilakukan di setiap titik sudut pada setiap *cell* yang dinyatakan dalam himpunan s_x dan himpunan s_y berdasarkan sumbu x dan sumbu y . Seluruh kemungkinan sudut arah pandang penempatan kamera CCTV akan dibangun berdasarkan jumlah kemungkinan arah pandang (n) dan dinyatakan dalam himpunan s_φ sehingga $s_\varphi = \{\frac{m}{n} \times 2\pi | m = 1, 2, \dots, n\}$. Setelah membangun seluruh kemungkinan penempatan kamera CCTV, didefinisikan variabel biner sebagai berikut:

$$x_{ij\varphi} = \begin{cases} 1 & \text{jika kamera CCTV ditempatkan pada titik } (i, j) \text{ dengan} \\ & \text{arah pandang } \varphi \\ 0 & \text{jika sebaliknya} \end{cases} \quad (3.10)$$

3.2.2 Fungsi Tujuan

Fungsi tujuan dalam model masalah *binary integer programming* ini dinyatakan dalam bentuk minimasi dari seluruh kemungkinan penempatan kamera CCTV. Dengan fungsi tujuan ini, maka solusi dari model masalah *linear program* ini merepresentasikan penempatan kamera CCTV yang berjumlah minimum. Berikut ini merupakan fungsi tujuan dalam model masalah *binary integer programming* ini:

$$\min z = \sum_{\varphi=0}^{s_{\varphi}-1} \sum_{i=0}^{s_i-1} \sum_{j=0}^{s_j-1} x_{ij\varphi} \quad (3.11)$$

3.2.3 Batasan

Selain mendapatkan penempatan kamera CCTV yang berjumlah minimum, setiap *cell* dalam ruangan harus tercakup oleh kamera CCTV. Untuk memenuhinya, pada model masalah *linear program* akan ditambahkan batasan di mana setiap *cell* harus dicakup oleh setidaknya 1 penempatan kamera CCTV. Terdapat fungsi biner yang digunakan untuk menyatakan ketercukupan *cell* oleh suatu penempatan kamera CCTV. Berikut ini merupakan fungsi biner tersebut:

$$cov(i, j, \varphi, p, q) = \begin{cases} 1 & \text{jika kamera CCTV ditempatkan pada titik } (i, j) \text{ dengan} \\ & \text{arah pandang } \varphi \text{ dapat mencakup } cell \text{ pada baris } p \text{ kolom } q \\ 0 & \text{jika sebaliknya} \end{cases} \quad (3.12)$$

Fungsi biner tersebut digunakan menyatakan ketercukupan *cell* oleh setiap kemungkinan penempatan kamera CCTV pada batasan. Berikut ini merupakan batasan pada model masalah *binary integer programming* ini:

$$\begin{aligned} \sum_{\varphi=0}^{s_{\varphi}-1} \sum_{i=0}^{s_i-1} \sum_{j=0}^{s_j-1} x_{i,j,\varphi} \times cov(i, j, \varphi, p, q) &\geq 1 \\ 0 \leq p \leq (s_p - 1), 0 \leq q \leq (s_q - 1) \end{aligned} \quad (3.13)$$

3.2.4 Model Masalah *Binary Integer Programming*

Variable pada model masalah *binary integer programming* ini terdiri dari seluruh kemungkinan penempatan kamera CCTV. Fungsi tujuan ditujukan untuk mendapatkan penempatan kamera CCTV yang berjumlah minimum. Agar seluruh isi ruangan dapat tercakup, maka ditambahkan batasan di mana setiap *cell* dalam ruangan harus dicakup oleh setidaknya 1 penempatan kamera CCTV. Dengan menggabungkan ketiganya, didapatkan model masalah *binary integer programming* sebagai berikut:

$$\begin{aligned} \min z &= \sum_{\varphi=0}^{s_{\varphi}-1} \sum_{i=0}^{s_i-1} \sum_{j=0}^{s_j-1} x_{ij\varphi} \\ s.t. \quad &\sum_{\varphi=0}^{s_{\varphi}-1} \sum_{i=0}^{s_i-1} \sum_{j=0}^{s_j-1} x_{i,j,\varphi} \times cov(i, j, \varphi, p, q) \geq 1 \\ &0 \leq p \leq (s_p - 1), 0 \leq q \leq (s_q - 1) \\ &x_{ij\varphi} \in \{0, 1\} \end{aligned} \quad (3.14)$$

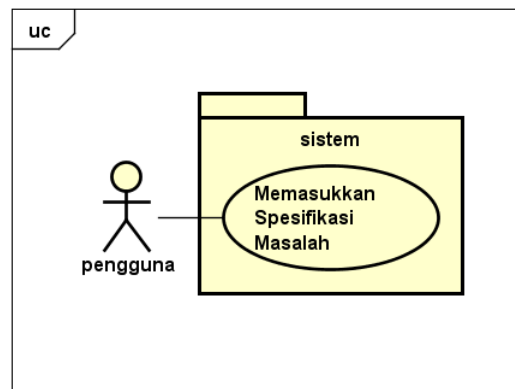
Selanjutnya, model ini akan diselesaikan menggunakan algoritma *Balas's additive* sehingga didapatkan solusi berupa kombinasi penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan.

3.3 Analisis Kebutuhan Perangkat Lunak

Pada bagian ini, akan dibahas mengenai kebutuhan dari perangkat lunak yang akan dibangun. Terdapat diagram *use case* dan skenario untuk menjelaskan aksi yang dapat dilakukan pengguna terhadap perangkat lunak. Perangkat lunak yang dibangun dapat menerima masukan-masukan masalah dan menghasilkan keluaran yang sesuai dengan masukan tersebut. Terdapat juga diagram kelas sederhana yang digunakan untuk membangun perangkat lunak. Diagram kelas sederhana ini akan dikembangkan lebih lanjut pada tahap perancangan.

3.3.1 Diagram *Use Case* dan Skenario

Pada perangkat lunak yang dibangun, hanya terdapat 1 buah aktor, yaitu pengguna. Diagram *use case* pada gambar 3.7 menunjukkan aktor beserta dengan aksi yang dapat dilakukannya.



Gambar 3.7: Diagram *use case*

Berikut ini skenario dari aksi pada diagram *use case*:

- Skenario: **Memasukkan Spesifikasi Masalah**

- Aktor: Pengguna
- Langkah:
 1. Aktor memasukkan spesifikasi masalah pada kolom masukan yang telah disediakan dan dilanjutkan dengan menekan tombol "submit".
 2. Sistem menampilkan tampilan penempatan kamera CCTV.

3.3.2 Kebutuhan Masukan Perangkat Lunak

Berdasarkan pemodelan masalah dan penyelesaian masalah menggunakan *binary integer programming*, ditetapkan masukan untuk perangkat lunak sebagai berikut:

- Panjang ruangan dalam satuan ukuran sentimeter (cm).
- Lebar ruangan dalam satuan ukuran sentimeter (cm).
- Jarak pandang kamera CCTV dalam satuan ukuran sentimeter (cm).
- Besar sudut pandang kamera CCTV dalam satuan sudut derajat (°).
- Ukuran terbesar *cell* dalam satuan ukuran sentimeter (cm).
- Jumlah kemungkinan sudut pandang penempatan kamera CCTV.

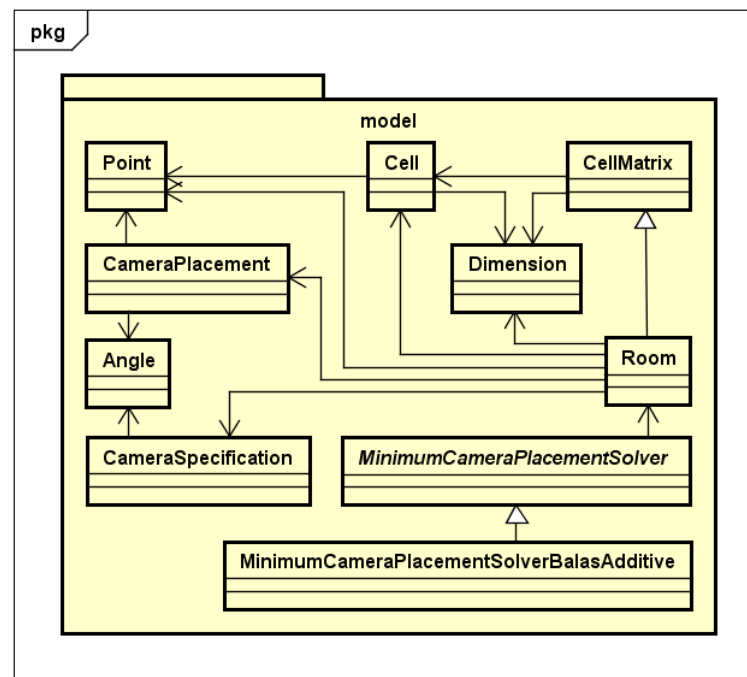
3.3.3 Kebutuhan Keluaran Perangkat Lunak

Perangkat lunak akan mencari solusi masalah berdasarkan masukan yang telah diberikan pengguna dan menghasilkan keluaran berupa:

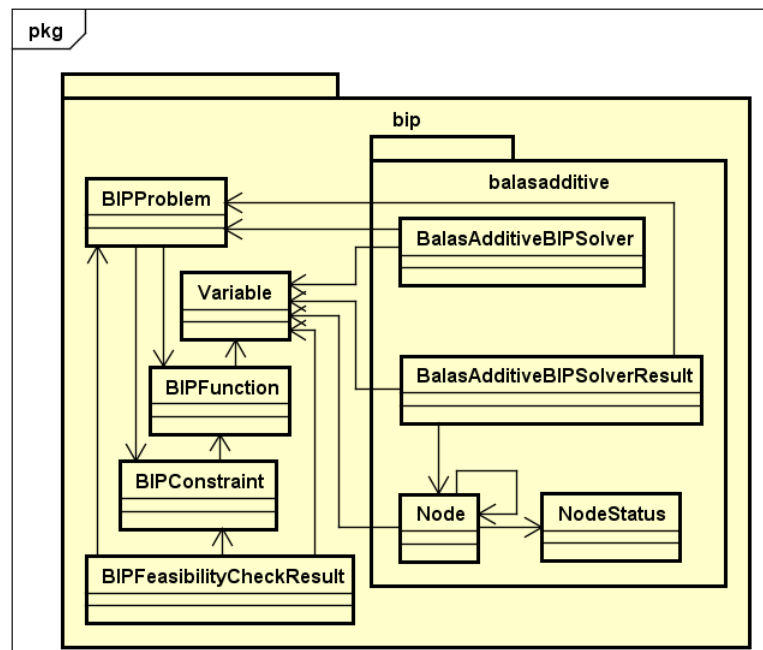
- Posisi dan sudut arah pandang dari setiap penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan.

3.3.4 Diagram Kelas Sederhana

Pada bagian ini terdapat diagram kelas sederhana yang menunjukkan rancangan awal kelas-kelas yang akan digunakan untuk membangun perangkat lunak. Kelas-kelas tersebut dikelompokkan ke dalam 2 *package*. *Package* pertama adalah *package* model yang berisi kelas-kelas untuk memodelkan masalah. *Package* kedua adalah *package* bip yang berisi kelas-kelas untuk menyelesaikan masalah *binary integer programming*. Di dalam *package* bip terdapat *subpackage* *balasadditive* yang berisi kelas-kelas untuk menyelesaikan masalah *binary integer programming* menggunakan algoritma *Balas's additive*. Diagram kelas sederhana untuk kedua *package* tersebut dapat dilihat pada gambar 3.8 dan 3.9.



Gambar 3.8: Diagram kelas sederhana untuk *package* model



Gambar 3.9: Diagram kelas sederhana untuk *package* bip dan *subpackage* balasadditive

BAB 4

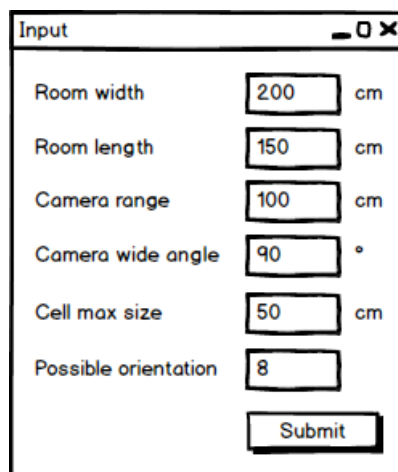
PERANCANGAN

4.1 Perancangan Antarmuka

Perangkat lunak yang dibangun akan menggunakan tampilan antarmuka grafis. Tampilan antarmuka ini digunakan agar mempermudah interaksi pengguna dengan perangkat lunak. Selain itu, jenis tampilan ini juga digunakan untuk menghasilkan visualisasi penempatan kamera CCTV sehingga pengguna dapat memahami penempatan-penempatan tersebut secara visual. Pada bagian ini akan dijelaskan bentuk dari setiap antarmuka yang terdapat dalam perangkat lunak. Berikut bentuk antarmuka tersebut:

- Antarmuka: **Penerima Masukan**

Antarmuka ini berfungsi untuk menerima masukan dari pengguna seperti pada gambar 4.1. Pada antarmuka ini terdapat kolom-kolom masukan yang dapat diisi oleh pengguna. Pengguna dapat mengisi ukuran ruangan, spesifikasi kamera CCTV, ukuran terbesar cell, dan jumlah kemungkinan arah pandang kamera CCTV pada kolom tersebut. Apabila pengguna sudah yakin dengan masukan yang diberikan, maka pengguna dapat menekan tombol "*submit*" yang akan mengarahkan pengguna pada antarmuka penempatan kamera CCTV.



Input	
Room width	<input type="text" value="200"/> cm
Room length	<input type="text" value="150"/> cm
Camera range	<input type="text" value="100"/> cm
Camera wide angle	<input type="text" value="90"/> °
Cell max size	<input type="text" value="50"/> cm
Possible orientation	<input type="text" value="8"/>
<input type="button" value="Submit"/>	

Gambar 4.1: Perancangan antarmuka penerima masukan

- Antarmuka: **Penempatan Kamera CCTV**

Antarmuka ini berfungsi untuk menampilkan solusi masalah penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan seperti pada gambar 4.2. Di dalam antarmuka ini terdapat 2 bagian, yaitu:

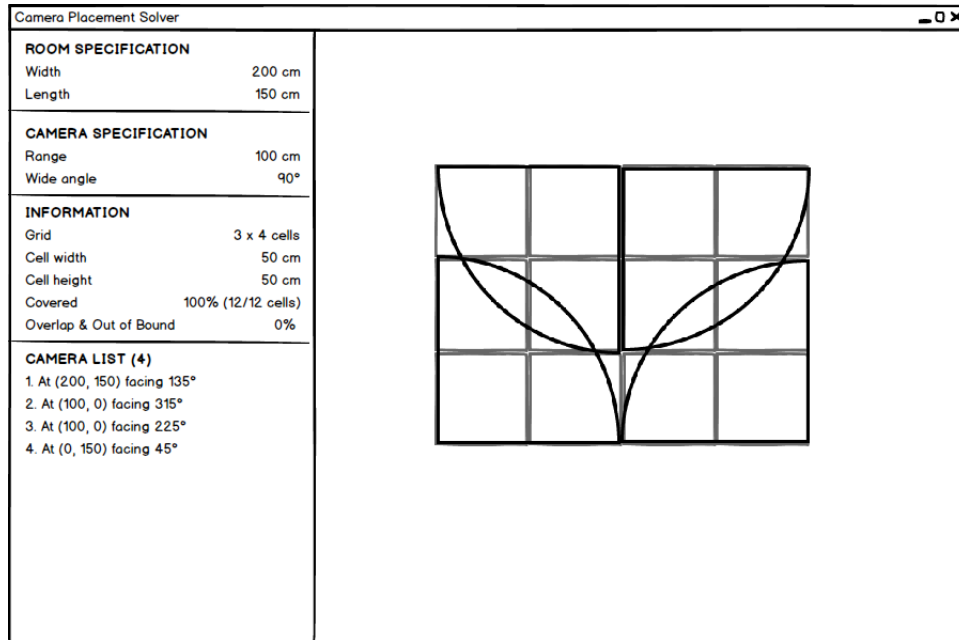
- Panel informasi

Panel ini berada di bagian kiri antarmuka yang berfungsi untuk memberikan informasi mengenai spesifikasi masalah dan solusi masalah. Solusi masalah yang terdiri dari

penempatan-penempatan kamera CCTV dapat dilihat pada bagian daftar kamera CCTV (*Camera List*). Pada setiap baris penempatan terdapat informasi titik lokasi penempatan dan juga sudut arah pandang yang dituju.

- Panel visualisasi penempatan kamera CCTV

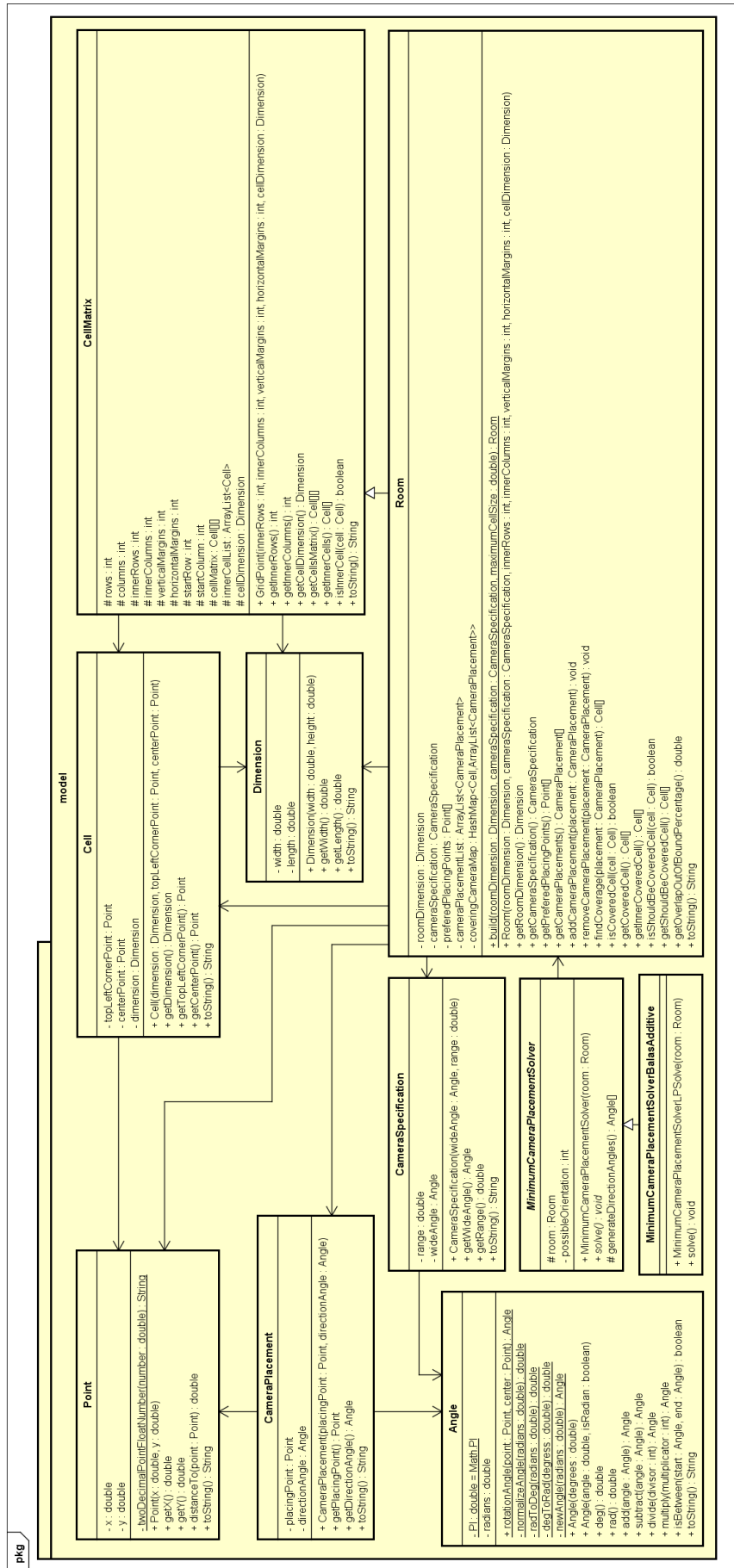
Panel ini berada di bagian kanan antarmuka yang berfungsi untuk menampilkan visualisasi matriks *cell* dan penempatan kamera CCTV.

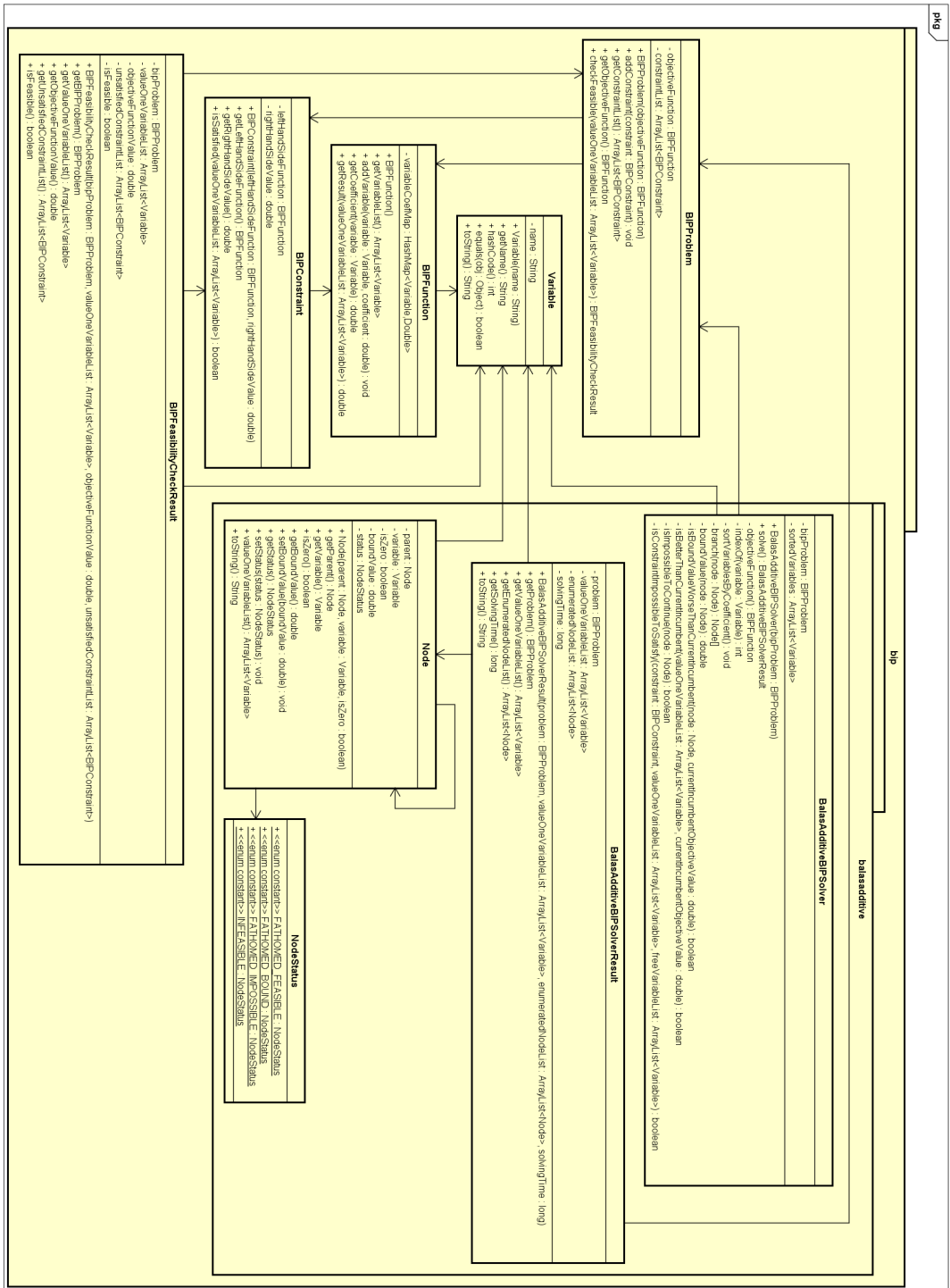


Gambar 4.2: Perancangan antarmuka penempatan kamera CCTV

4.2 Perancangan Kelas

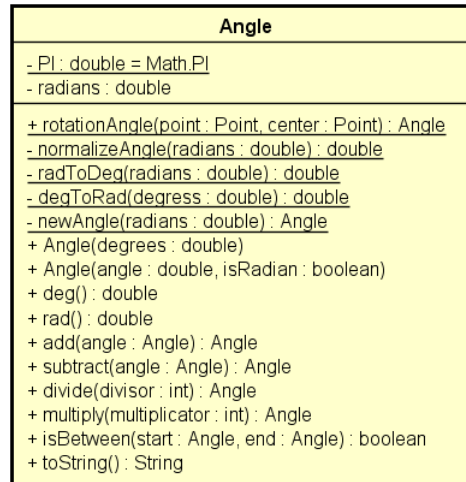
Rancangan awal kelas-kelas yang telah dipaparkan di 3.3.4 akan dikembangkan lebih lanjut pada bagian ini. Kelas-kelas yang digunakan untuk memodelkan masalah diletakkan dalam *package* model. Kelas-kelas untuk memodelkan dan menyelesaikan masalah *binary integer programming* diletakkan pada *package* bip. Diagram kelas rinci untuk kedua *package* tersebut dapat dilihat pada gambar 4.3 dan 4.4.

Gambar 4.3: Diagram kelas rinci untuk *package* model



Gambar 4.4: Diagram kelas rinci untuk *package* bip dan *subpackage* balasadditive

4.2.1 Kelas *Angle*



Gambar 4.5: Diagram kelas *Angle*

Kelas ini merepresentasikan sudut dan menangani fungsi-fungsi yang berhubungan dengan sudut. Diagram kelas *Angle* dapat dilihat pada gambar 4.5. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Angle*:

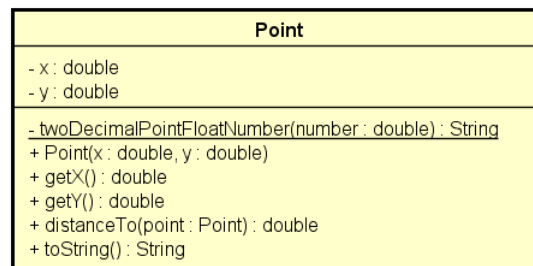
- ***PI* → *double***
Atribut ini merupakan atribut statis bernilai π yang dapat digunakan oleh setiap objek dari kelas *Angle*.
- ***radians* → *double***
Atribut ini berguna untuk menampung sudut dalam bentuk radian.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *Angle*:

- ***rotationAngle(point* → *Point*, *center* → *Point*) → *Angle***
Fungsi ini merupakan fungsi statis yang berguna untuk mendapatkan sudut rotasi dari titik *point* terhadap titik *center*.
- ***normalizeAngle(radians* → *double*) → *double***
Fungsi ini merupakan fungsi statis yang berguna untuk melakukan normalisasi sudut *radians* sehingga berada dalam rentang $0 \leq \text{radians} < 2\pi$.
- ***radToDeg(radians* → *double*) → *double***
Fungsi ini merupakan fungsi statis yang berguna untuk mengubah sudut dalam bentuk radian menjadi sudut dalam bentuk derajat.
- ***degToRad(degrees* → *double*) → *double***
Fungsi ini merupakan fungsi statis yang berguna untuk mengubah sudut dalam bentuk derajat menjadi sudut dalam bentuk radian.
- ***newAngle(radians* → *double*) → *Angle***
Fungsi ini merupakan fungsi statis yang berguna untuk membuat objek *Angle* baru.
- ***deg()* → *double***
Fungsi ini berguna untuk mendapatkan sudut dalam bentuk derajat.

- ***rad()* → *double***
Fungsi ini berguna untuk mendapatkan sudut dalam bentuk radian.
- ***add(angle → Angle) → Angle***
Fungsi ini berguna untuk menghasilkan objek sudut baru yang merupakan hasil penjumlahan antara sudut objek ini dengan sudut objek *angle*.
- ***subtract(angle → Angle) → Angle***
Fungsi ini berguna untuk menghasilkan objek sudut baru yang merupakan hasil pengurangan antara sudut objek ini dengan sudut objek *angle*.
- ***divide(divisor → int) → Angle***
Fungsi ini berguna untuk menghasilkan objek sudut baru yang merupakan hasil pembagian antara sudut objek ini dengan nilai *divisor*.
- ***multiply(multiplier → int) → Angle***
Fungsi ini berguna untuk menghasilkan objek sudut baru yang merupakan hasil pengalihan antara sudut objek ini dengan nilai *multiplier*.
- ***isBetween(start → Angle, end → Angle) → boolean***
Fungsi ini berguna untuk mengetahui apakah sudut objek ini berada di antara sudut objek *start* dan sudut objek *end*.

4.2.2 Kelas *Point*



Gambar 4.6: Diagram kelas *Point*

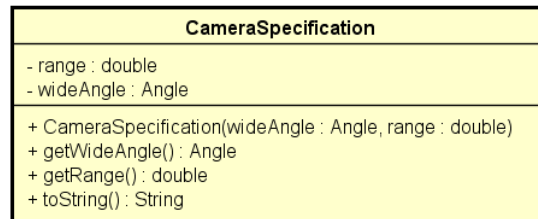
Kelas ini merepresentasikan titik koordinat 2D. Diagram kelas *Point* dapat dilihat pada gambar 4.6. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Point*:

- ***x → double***
Atribut ini berguna untuk menampung nilai titik pada sumbu x.
- ***y → double***
Atribut ini berguna untuk menampung nilai titik pada sumbu y.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *Point*:

- ***twoDecimalPointFloatNumber(number → double) → String***
Fungsi ini merupakan fungsi statis yang berguna untuk mengubah bilangan *number* ke dalam bentuk *String* dengan maksimal bilangan di belakang koma berjumlah 2 buah.
- ***distanceTo(point → Point) → double***
Fungsi ini berguna untuk mendapatkan jarak antara titik objek ini dengan titik objek *point*.

4.2.3 Kelas *CameraSpecification*

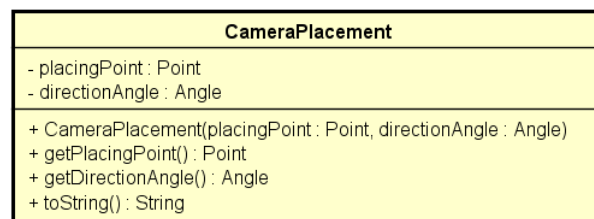


Gambar 4.7: Diagram kelas *CameraSpecification*

Kelas ini merepresentasikan spesifikasi kamera CCTV yang terdiri dari jarak pandang dan besar sudut pandang. Diagram kelas *CameraSpecification* dapat dilihat pada gambar 4.7. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *CameraSpecification*:

- ***range* → *double***
Atribut ini berguna untuk menampung jarak pandang kamera CCTV.
- ***wideAngle* → *Angle***
Atribut ini berguna untuk menampung besar sudut pandang kamera CCTV.

4.2.4 Kelas *CameraPlacement*

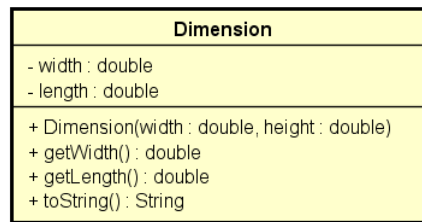


Gambar 4.8: Diagram kelas *CameraPlacement*

Kelas ini merepresentasikan penempatan kamera CCTV yang terdiri dari posisi dan sudut arah pandang. Diagram kelas *CameraPlacement* dapat dilihat pada gambar 4.8. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *CameraPlacement*:

- ***placingPoint* → *Point***
Atribut ini berguna untuk menampung posisi penempatan kamera CCTV.
- ***directionAngle* → *Angle***
Atribut ini berguna untuk menampung sudut arah pandang kamera CCTV.

4.2.5 Kelas *Dimension*

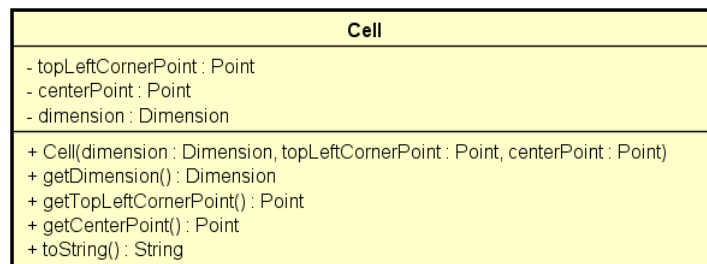


Gambar 4.9: Diagram kelas *Dimension*

Kelas ini merepresentasikan dimensi yang terdiri dari panjang dan lebar. Diagram kelas *Dimension* dapat dilihat pada gambar 4.9. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Dimension*:

- ***width* → *double***
Atribut ini berguna untuk menampung ukuran lebar.
- ***length* → *double***
Atribut ini berguna untuk menampung ukuran panjang.

4.2.6 Kelas *Cell*

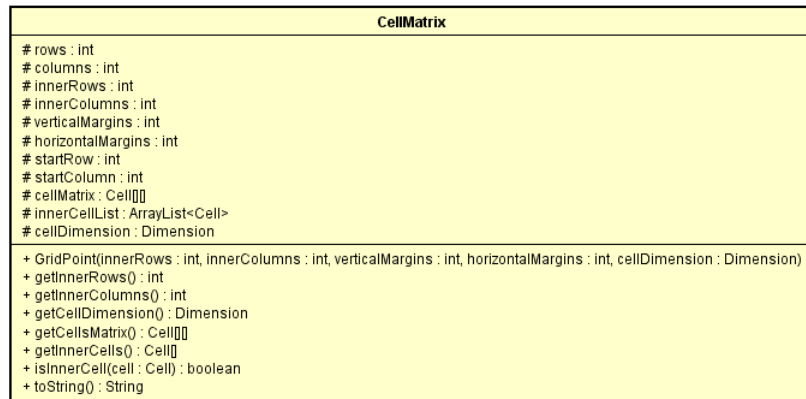


Gambar 4.10: Diagram kelas *Cell*

Kelas ini merepresentasikan cell. Diagram kelas *Cell* dapat dilihat pada gambar 4.10. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Cell*:

- ***topLeftCornerPoint* → *Point***
Atribut ini berguna untuk menampung titik ujung kiri atas cell.
- ***centerPoint* → *Point***
Atribut ini berguna untuk menampung titik tengah cell.
- ***dimension* → *Dimension***
Atribut ini berguna untuk menampung dimensi cell.

4.2.7 Kelas *CellMatrix*

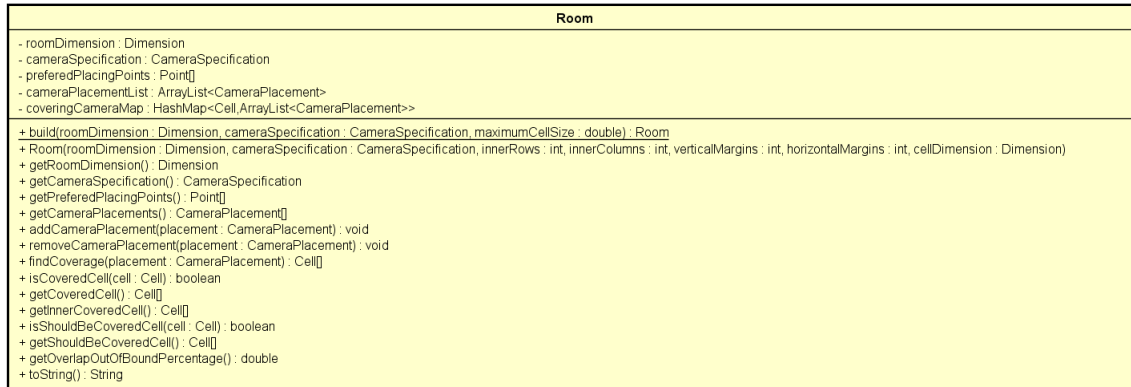


Gambar 4.11: Diagram kelas *CellMatrix*

Kelas ini merepresentasikan matriks *cell* dalam ruangan. Diagram kelas *MatrixCell* dapat dilihat pada gambar 4.11. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *CellMatrix*:

- ***rows* → *int***
Atribut ini berguna untuk menampung jumlah baris matriks *cell* secara keseluruhan.
- ***columns* → *int***
Atribut ini berguna untuk menampung jumlah kolom matriks *cell* secara keseluruhan.
- ***innerRows* → *int***
Atribut ini berguna untuk menampung jumlah baris matriks *cell* bagian dalam.
- ***innerColumns* → *int***
Atribut ini berguna untuk menampung jumlah kolom matriks *cell* bagian dalam.
- ***verticalMargins* → *int***
Atribut ini berguna untuk menampung jumlah margin vertikal matriks *cell*.
- ***horizontalMargins* → *int***
Atribut ini berguna untuk menampung jumlah margin horizontal matriks *cell*.
- ***startRow* → *int***
Atribut ini berguna untuk menampung indeks baris pertama matriks *cell* bagian dalam.
- ***startColumn* → *int***
Atribut ini berguna untuk menampung indeks kolom pertama matriks *cell* bagian dalam.
- ***cellMatrix* → *Cell*[][]**
Atribut ini berguna untuk menampung matriks *cell* 2 dimensi.
- ***innerCellList* → *ArrayList*<*Cell*>**
Atribut ini berguna untuk menampung seluruh *cell* pada matriks *cell* bagian dalam.
- ***cellDimension* → *Dimension***
Atribut ini berguna untuk menampung dimensi *cell*.

4.2.8 Kelas *Room*



Gambar 4.12: Diagram kelas *Room*

Kelas ini merepresentasikan ruangan yang dapat diisi oleh kamera-kamera CCTV. Kelas ini merupakan turunan dari kelas *CellMatrix*. Diagram kelas *Room* dapat dilihat pada gambar 4.12. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Room*:

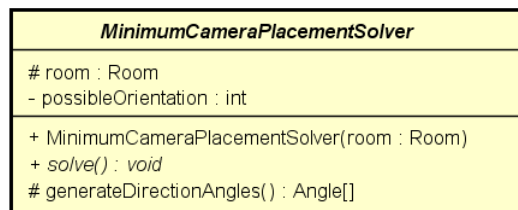
- ***roomDimension* → *Dimension***
Atribut ini berguna untuk menampung dimensi ruangan.
- ***cameraSpecification* → *CameraSpecification***
Atribut ini berguna untuk menampung spesifikasi dari kamera CCTV yang akan ditempatkan dalam ruangan.
- ***preferredPlacingPoints* → *Point*[]**
Atribut ini berguna untuk menampung posisi-posisi yang dapat ditempati oleh kamera CCTV.
- ***cameraPlacementList* → *ArrayList*<*CameraPlacement*>**
Atribut ini berguna untuk menampung daftar penempatan kamera CCTV.
- ***coveringCameraMap* → *HashMap*<*Cell*, *ArrayList*<*CameraPlacement*>>**
Atribut ini berguna untuk menampung pemetaan *cell* dengan penempatan kamera CCTV yang dapat mencakup *cell* tersebut.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *Room*:

- ***build*(*roomDimension* → *Dimension*, *cameraSpecification* → *CameraSpecification*, *maximumCellSize* → *double*) → *Room***
Fungsi ini merupakan fungsi statis yang berguna untuk membuat objek *Room* yang dimana jumlah baris, jumlah kolom, jumlah margin vertikal, jumlah margin horizontal, dan ukuran cell akan ditentukan berdasarkan ukuran ruangan *roomDimension*, spesifikasi kamera *cameraSpecification*, dan ukuran terbesar cell *maximumCellSize*.
- ***addCameraPlacement*(*placement* → *CameraPlacement*) → *void***
Fungsi ini berguna untuk menambahkan penempatan kamera CCTV ke dalam daftar penempatan kamera CCTV dan memperbaharui pemetaan cell pada atribut *coveringCameraMap*.
- ***removeCameraPlacement*(*placement* → *CameraPlacement*) → *void***
Fungsi ini berguna untuk membuang penempatan kamera CCTV dari daftar penempatan kamera CCTV dan memperbaharui pemetaan cell pada atribut *coveringCameraMap*.

- ***findCoverage(placement → CameraPlacement) → Cell[]***
Fungsi ini berguna untuk mendapatkan *cell* yang tercakup oleh penempatan kamera CCTV *placement*.
- ***isCoveredCell(cell → Cell) → boolean***
Fungsi ini berguna untuk mengetahui apakah *cell* telah tercakup oleh setidaknya 1 penempatan kamera CCTV.
- ***getCoveredCell() → Cell[]***
Fungsi ini berguna untuk mendapatkan *cell* yang telah tercakup oleh setidaknya 1 penempatan kamera CCTV.
- ***getInnerCoveredCell() → Cell[]***
Fungsi ini berguna untuk mendapatkan *cell* yang berada pada matriks *cell* bagian dalam yang telah tercakup oleh setidaknya 1 penempatan kamera CCTV.
- ***isShouldBeCoveredCell(cell → Cell) → boolean***
Fungsi ini berguna untuk mengetahui apakah *cell* berada pada matriks *cell* bagian dalam dan belum tercakup oleh setidaknya 1 penempatan kamera CCTV.
- ***getShouldBeCoveredCell() → Cell[]***
Fungsi ini berguna untuk mendapatkan *cell* yang berada pada matriks *cell* bagian dalam dan belum tercakup oleh setidaknya 1 penempatan kamera CCTV.
- ***getOverlapAndOutOfBoundPercentage() → double***
Fungsi ini berguna untuk mendapatkan persentase *overlap* dan *out of bound*.

4.2.9 Kelas *MinimumCameraPlacementSolver*



Gambar 4.13: Diagram kelas *MinimumCameraPlacementSolver*

Kelas ini merepresentasikan pemecah masalah penempatan kamera CCTV dalam ruangan yang berjumlah minimum yang dapat mencakup seluruh isi ruangan. Kelas ini merupakan kelas abstrak. Diagram kelas *MinimumCameraPlacementSolver* dapat dilihat pada gambar 4.13. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *MinimumCameraPlacementSolver*:

- ***room → Room***
Atribut ini berguna untuk menampung ruangan yang di mana masalah penempatan kamera CCTV-nya akan diselesaikan.
- ***possibleOrientation → int***
Atribut ini berguna untuk menampung jumlah kemungkinan sudut arah pandang kamera CCTV.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *Room*:

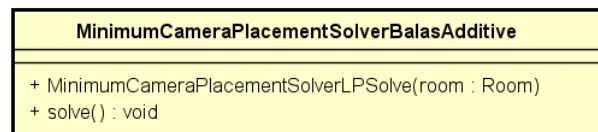
- *solve()* → *void*

Fungsi ini merupakan fungsi abstrak yang bertujuan untuk menyelesaikan masalah penempatan kamera CCTV dalam ruangan yang berjumlah minimum yang dapat mencakup seluruh isi ruangan.

- *generateDirectionAngles()* → *Angle[]*

Fungsi ini berguna untuk menghasilkan sudut-sudut arah pandang berdasarkan jumlah kemungkinan sudut arah pandang *possibleOrientation*.

4.2.10 Kelas *MinimumCameraPlacementSolverBalasAdditive*



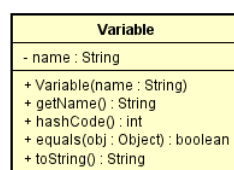
Gambar 4.14: Diagram kelas *MinimumCameraPlacementSolverBalasAdditive*

Kelas ini merepresentasikan pemecah masalah penempatan kamera CCTV dalam ruangan yang berjumlah minimum yang dapat mencakup seluruh isi ruangan dengan menggunakan algoritma *Balas's additive* (2.2.1). Kelas ini merupakan turunan dari kelas *MinimumCameraPlacementSolver*. Diagram kelas *MinimumCameraPlacementSolverBalasAdditive* dapat dilihat pada gambar 4.14. Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *MinimumCameraPlacementSolverBalasAdditive*:

- *solve()* → *void*

Fungsi ini berguna untuk menyelesaikan masalah penempatan kamera CCTV dalam ruangan yang berjumlah minimum yang dapat mencakup seluruh isi ruangan menggunakan algoritma *Balas's additive*.

4.2.11 Kelas *Variable*



Gambar 4.15: Diagram kelas *Variable*

Kelas ini merepresentasikan variabel. Diagram kelas *Variable* dapat dilihat pada gambar 4.15. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Variable*:

- *name* → *String*

Atribut ini berguna untuk menampung nama variabel.

4.2.12 Kelas *BIPFunction*

BIPFunction
- <code>variableCoefMap : HashMap<Variable, Double></code>
+ <code>BIPFunction()</code>
+ <code>getVariableList() : ArrayList<Variable></code>
+ <code>addVariable(variable : Variable, coefficient : double) : void</code>
+ <code>getCoefficient(variable : Variable) : double</code>
+ <code>getResult(valueOneVariableList : ArrayList<Variable>) : double</code>

Gambar 4.16: Diagram kelas *BIPFunction*

Kelas ini merepresentasikan persamaan yang terdiri dari variabel biner. Diagram kelas *BIPFunction* dapat dilihat pada gambar 4.16. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *BIPFunction*:

- ***variableCoefMap* → *HashMap<Variable, Double>***
Atribut ini berguna untuk menampung koefisien dari setiap variabel.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *BIPFunction*:

- ***getVariableList()* → *ArrayList<Variable>***
Fungsi ini berguna untuk mendapatkan daftar variabel yang terdapat dalam persamaan.
- ***addVariable(variable* → *Variable*, *coefficient* → *double*) → *void***
Fungsi ini berguna untuk menambahkan variabel beserta koefisiennya ke dalam persamaan.
- ***getCoefficient(variable* → *Variable*) → *double***
Fungsi ini berguna untuk mendapatkan koefisien *variable*.
- ***getResult(valueOneVariableList* → *ArrayList<Variable>*) → *double***
Fungsi ini berguna untuk mendapatkan hasil persamaan berdasarkan daftar variabel yang bernilai 1.

4.2.13 Kelas *BIPConstraint*

BIPConstraint
- <code>leftHandSideFunction : BIPFunction</code>
- <code>rightHandSideValue : double</code>
+ <code>BIPConstraint(leftHandSideFunction : BIPFunction, rightHandSideValue : double)</code>
+ <code>getLeftHandSideFunction() : BIPFunction</code>
+ <code>getRightHandSideValue() : double</code>
+ <code>isSatisfied(valueOneVariableList : ArrayList<Variable>) : boolean</code>

Gambar 4.17: Diagram kelas *BIPConstraint*

Kelas ini merepresentasikan batasan dalam masalah *binary integer programming*. Diagram kelas *BIPConstraint* dapat dilihat pada gambar 4.17. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *BIPConstraint*:

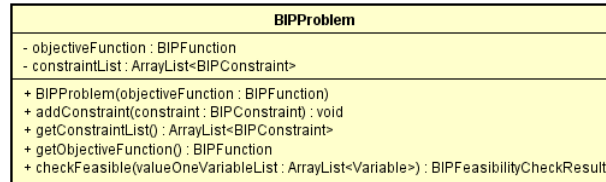
- ***leftHandSideFunction* → *BIPFunction***
Atribut ini berguna untuk menampung persamaan pada ruas kiri batasan.
- ***rightHandSideValue* → *double***
Atribut ini berguna untuk menampung nilai pada ruas kanan batasan.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *BIPConstraint*:

- ***isSatisfied(valueOneVariableList → ArrayList<Variable>) → boolean***

Fungsi ini berguna untuk mengetahui apakah batasan dapat dipenuhi berdasarkan daftar variabel yang bernilai 1.

4.2.14 Kelas *BIPProblem*



Gambar 4.18: Diagram kelas *BIPProblem*

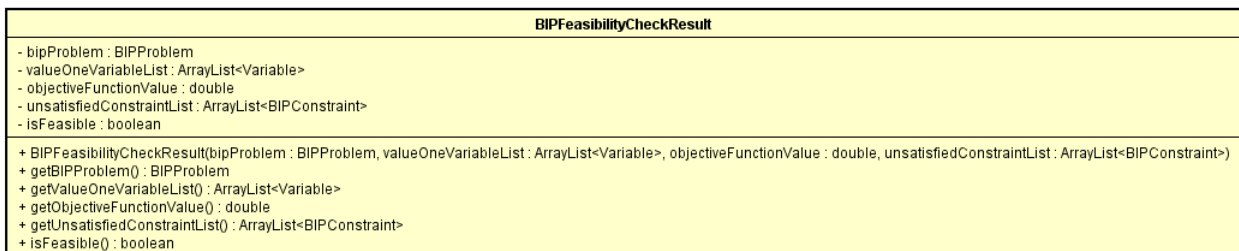
Kelas ini merepresentasikan model masalah *binary integer programming*. Diagram kelas *BIPProblem* dapat dilihat pada gambar 4.18. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *BIPProblem*:

- ***objectiveFunction → BIPFunction***
Atribut ini berguna untuk menampung persamaan fungsi tujuan.
- ***constraintList → ArrayList<BIPConstraint>***
Atribut ini berguna untuk menampung batasan-batasan.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *BIPProblem*:

- ***addConstraint(constraint → BIPConstraint) → void***
Fungsi ini berguna untuk menambahkan batasan pada model masalah *binary integer programming*.
- ***checkFeasible(valueOneVariableList → ArrayList<Variable>) → BIPFeasibilityCheckResult***
Fungsi ini berguna untuk mengembalikan hasil pengecekan solusi *feasible* berdasarkan daftar variabel bernilai 1.

4.2.15 Kelas *BIPFeasibilityCheckResult*

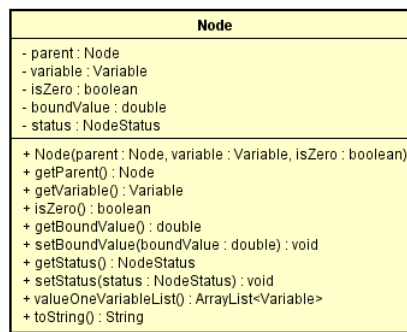


Gambar 4.19: Diagram kelas *BIPFeasibilityCheckResult*

Kelas ini merepresentasikan hasil pengecekan solusi *feasible* pada masalah *binary integer programming*. Diagram kelas *BIPFeasibilityCheckResult* dapat dilihat pada gambar 4.19. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *BIPFeasibilityCheckResult*:

- ***blpProblem* → *BLPProblem***
Atribut ini berguna untuk menampung model masalah *binary integer programming*.
- ***valueOneVariableList* → *ArrayList<Variable>***
Atribut ini berguna untuk menampung solusi berupa daftar variabel yang bernilai 1.
- ***objectiveFunctionValue* → *double***
Atribut ini berfungsi untuk menampung nilai hasil fungsi tujuan.
- ***unsatisfiedConstraintList* → *ArrayList<BIPConstraint>***
Atribut ini berfungsi untuk menampung batasan-batasan yang tidak dipenuhi.
- ***isFeasible* → *boolean***
Atribut ini berfungsi untuk menampung apakah solusi bersifat *feasible*.

4.2.16 Kelas *Node*



Gambar 4.20: Diagram kelas *Node*

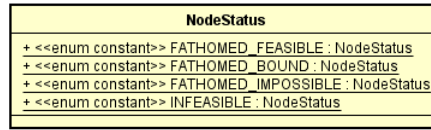
Kelas ini merepresentasikan *node* yang digunakan dalam algoritma *Balas's additive*. Diagram kelas *Node* dapat dilihat pada gambar 4.20. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *Node*:

- ***parent* → *Node***
Atribut ini berguna untuk menampung *node* orang tua dari *node* ini.
- ***variable* → *Variable***
Atribut ini berguna untuk menampung variabel yang dituju oleh *node* ini.
- ***isZero* → *boolean***
Atribut ini berguna untuk menampung apakah variabel pada *node* ini bernilai 1.
- ***boundValue* → *double***
Atribut ini berguna untuk menampung nilai *bound* dari *node* ini.
- ***status* → *NodeStatus***
Atribut ini berguna untuk menampung status dari *node* ini.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *Node*:

- ***valueOneVariableList()* → *ArrayList<Variable>***
Fungsi ini berguna untuk menghasilkan daftar variabel bernilai 1 yang dimulai dari *node* ini hingga *root node*.

4.2.17 Kelas *NodeStatus*

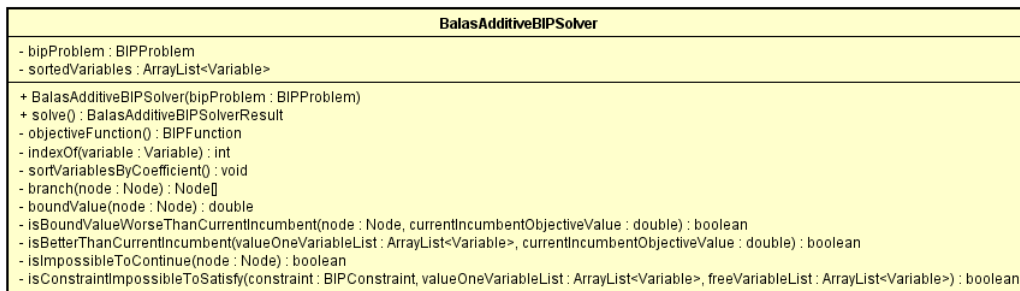


Gambar 4.21: Diagram kelas *NodeStatus*

Kelas ini merepresentasikan status yang dapat dimiliki *node*. Diagram kelas *NodeStatus* dapat dilihat pada gambar 4.21. Berikut ini merupakan pilihan-pilihan status yang terdapat pada kelas *NodeStatus*:

- ***FATHOMED_FEASIBLE* → *NodeStatus***
Pilihan ini menyatakan status *fathomed* dengan alasan solusi *feasible*.
- ***FATHOMED_BOUND* → *NodeStatus***
Pilihan ini menyatakan status *fathomed* dengan alasan nilai *bound* yang lebih buruk daripada solusi *incumbent*.
- ***FATHOMED_IMPOSSIBLE* → *NodeStatus***
Pilihan ini menyatakan status *fathomed* dengan alasan *impossible* atau tidak dapat menghasilkan *incumbent*.
- ***INFEASIBLE* → *NodeStatus***
Pilihan ini menyatakan status *infeasible*.

4.2.18 Kelas *BalasAdditiveBIPSolver*



Gambar 4.22: Diagram kelas *BalasAdditiveBIPSolver*

Kelas ini merepresentasikan metode penyelesaian masalah *binary integer programming* menggunakan algoritma *Balas's additive*. Diagram kelas *BalasAdditiveBIPSolver* dapat dilihat pada gambar 4.22. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *BalasAdditiveBIPSolver*:

- ***bipProblem* → *BIPProblem***
Atribut ini berguna untuk menampung model masalah *binary integer programming* yang akan diselesaikan.
- ***sortedVariables* → *ArrayList<Variable>***
Atribut ini berfungsi untuk menampung variabel biner yang telah diurut menaik berdasarkan koefisiennya.

Berikut ini merupakan fungsi-fungsi yang terdapat pada kelas *BalasAdditiveBIPSolver*:

- ***solve()* → *BalasAdditiveBIPSolverResult***
Fungsi ini berguna untuk mendapatkan solusi masalah *bipProblem* menggunakan algoritma *Balas's additive*.
- ***objectiveFunction()* → *BIPFunction***
Fungsi ini berguna untuk mendapatkan fungsi tujuan dari masalah *bipProblem*.
- ***indexOf(variable → Variable) → int***
Fungsi ini berguna untuk mendapatkan indeks *variable* dalam daftar variabel yang telah terurut.
- ***sortVariablesByCoefficient()* → void**
Fungsi ini berguna untuk mengurutkan variabel secara menaik berdasarkan koefisiennya.
- ***branch(node → Node) → Node []***
Fungsi ini berguna untuk membuat cabang *node-0* dan *node-1* dari *node*.
- ***boundValue(node → Node) → double***
Fungsi ini berguna untuk menghitung nilai *bound* dari *node*.
- ***isBoundValue Worse Than Current Incumbent(node → Node, currentIncumbentObjectiveValue → double) → boolean***
Fungsi ini berguna untuk mengetahui apakah nilai *bound* pada *node* lebih buruk daripada nilai solusi *incumbent*.
- ***isBetterThanCurrentIncumbent(valueOneVariableList → ArrayList<Variable>, currentIncumbentObjectiveValue → double) → boolean***
Fungsi ini berguna untuk mengetahui apakah daftar variabel bernilai 1 dapat menghasilkan solusi yang lebih baik daripada solusi *incumbent*.
- ***isImpossibleToContinue(node → Node) → boolean***
Fungsi ini berguna untuk mengetahui apakah *node* tidak mungkin untuk menghasilkan *incumbent*.
- ***isConstraintImpossibleToSatisfy(constraint → BIPConstraint, valueOneVariableList → ArrayList<Variable>, freeVariableList → ArrayList<Variable>) → boolean***
Fungsi ini berguna untuk mengetahui apakah *constraint* dapat dipenuhi berdasarkan daftar variabel bernilai 1 dan daftar variabel bebas.

4.2.19 Kelas *BalasAdditiveBIPSolverResult*

BalasAdditiveBIPSolverResult
- problem : BIPProblem - valueOneVariableList : ArrayList<Variable> - enumeratedNodeList : ArrayList<Node> - solvingTime : long
+ BalasAdditiveBIPSolverResult(problem : BIPProblem, valueOneVariableList : ArrayList<Variable>, enumeratedNodeList : ArrayList<Node>, solvingTime : long) + getProblem() : BIPProblem + getValueOneVariableList() : ArrayList<Variable> + getEnumeratedNodeList() : ArrayList<Node> + getSolvingTime() : long + toString() : String

Gambar 4.23: Diagram kelas *BalasAdditiveBIPSolverResult*

Kelas ini merepresentasikan hasil dari penyelesaian masalah *binary integer programming* menggunakan algoritma *Balas's additive*. Diagram kelas *BalasAdditiveBIPSolverResult* dapat dilihat pada gambar 4.23. Berikut ini merupakan atribut-atribut yang terdapat pada kelas *BalasAdditiveBIPSolverResult*:

- ***bipProblem*** → ***BIPProblem***
Atribut ini berguna untuk menampung model masalah *binary integer programming*.
- ***valueOneVariableList*** → ***ArrayList<Variable>***
Atribut ini berguna untuk menampung solusi penyelesaian berupa daftar variabel yang bernilai 1.
- ***enumeratedNodeList*** → ***ArrayList<Node>***
Atribut ini berfungsi untuk menampung daftar *node* yang diperiksa selama melakukan penyelesaian masalah menggunakan algoritma *Balas's additive*.
- ***solvingTime*** → ***long***
Atribut ini berfungsi untuk menampung waktu yang digunakan dalam menyelesaikan masalah dalam satuan milisekon.

BAB 5

IMPLEMENTASI DAN PENGUJIAN

Pada bab ini, akan dibahas hasil dari implementasi dan pengujian terhadap perangkat lunak yang dibangun. Terdapat spesifikasi lingkungan perangkat lunak dan perangkat keras yang digunakan dalam melakukan implementasi dan pengujian.

5.1 Lingkungan Implementasi Perangkat Keras

Berikut ini merupakan spesifikasi perangkat keras yang digunakan baik pada tahap implementasi maupun pada tahap pengujian:

- CPU: Intel® Core™ i5-7200U Processor, 3M Cache, up to 3.10 Ghz
- GPU: NVIDIA GeForce 930MX
- RAM: 8GB

5.2 Lingkungan Implementasi Perangkat Lunak

Berikut ini merupakan spesifikasi perangkat lunak yang digunakan baik pada tahap implementasi maupun pada tahap pengujian:

- OS: Windows 10 Pro, 64-bit
- Pemrograman: Java 8 Update 152 (64-bit)

5.3 Implementasi Antarmuka

Pada bagian ini, akan dibahas hasil implementasi antarmuka sesuai dengan perancangan antarmuka yang dilakukan pada 4.1. Berikut ini merupakan hasil implementasi antarmuka:

- Antarmuka: **Penerima Masukan**
Gambar 5.1 menunjukkan tampilan antarmuka penerima masukan. Pada antarmuka ini terdapat kolom-kolom masukan yang dapat diisi oleh pengguna. Apabila pengguna telah yakin dengan masukan yang diberikan, pengguna dapat menekan tombol "*submit*" untuk diarahkan ke antarmuka penempatan kamera CCTV.

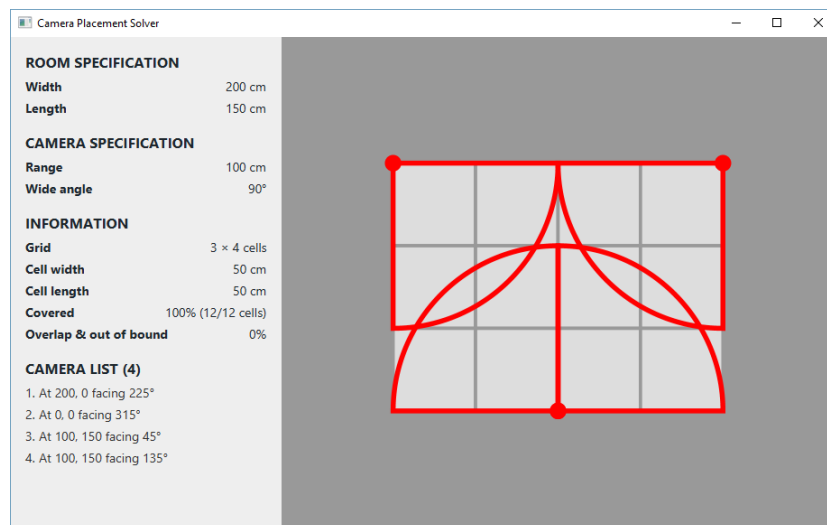
Room width	200	cm
Room length	150	cm
Camera range	100	cm
Camera wide angle	90	°
Cell max size	50	cm
Possible orientation	8	

Submit

Gambar 5.1: Antarmuka penerima masukan

- Antarmuka: **Penempatan Kamera CCTV**

Gambar 5.2 menunjukkan tampilan antarmuka penempatan kamera CCTV. Pada antarmuka ini, pengguna dapat melihat solusi penempatan-penempatan kamera CCTV berdasarkan masukan yang telah dimasukkan sebelumnya. Pada bagian kiri terdapat informasi mengenai spesifikasi masalah yang terdiri dari spesifikasi ruangan dan spesifikasi kamera CCTV. Selain itu, terdapat informasi lainnya seperti ukuran matriks *cell*, ukuran *cell*, tingkat ketercakupan, dan tingkat *overlap out of bound*. Pada bagian kanan terdapat visualisasi penempatan kamera CCTV dalam ruangan yang dipecah menjadi matriks *cell*.



Gambar 5.2: Antarmuka penempatan kamera CCTV

5.4 Pengujian

Pada perangkat lunak yang telah dibangun, akan dilakukan 2 buah pengujian. Pengujian pertama dilakukan untuk melihat pengaruh perubahan masukan terhadap ukuran masalah masalah *binary integer programming*. Pengujian kedua dilakukan untuk membandingkan jumlah iterasi pada algoritma *Balas's additive* dengan iterasi *exhaustive search* berjumlah 2^n .

5.4.1 Pengujian-1

Pada pengujian ini, akan dilihat pengaruh perubahan masukan masalah terhadap ukuran masalah *binary integer programming*. Sebanyak 4 buah eksperimen akan dilakukan dalam pengujian ini.

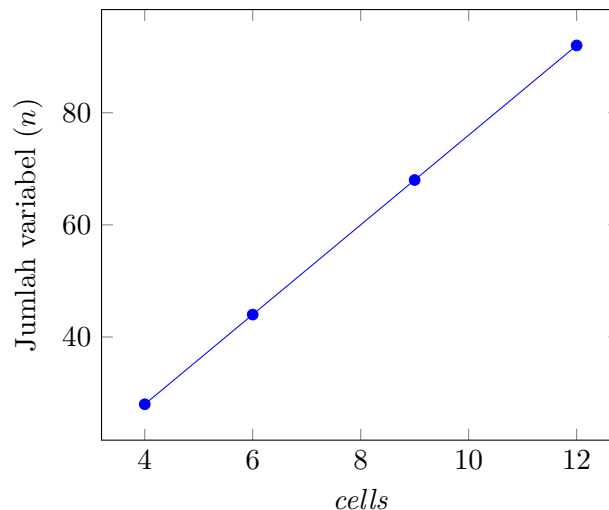
Masukan yang digunakan untuk eksperimen ini dibagi menjadi 2 bagian:

- **Masukan tetap**
 - Jarak pandang kamera CCTV : 100 cm
 - Sudut pandang kamera CCTV : 90°
 - Ukuran terbesar *cell* : 50 cm
 - Jumlah kemungkinan arah pandang kamera CCTV : 8 buah
- **Masukan tidak tetap**
 - (Eksperimen-1) Ukuran ruangan : 100 cm \times 100 cm
 - (Eksperimen-2) Ukuran ruangan : 100 cm \times 150 cm
 - (Eksperimen-3) Ukuran ruangan : 150 cm \times 150 cm
 - (Eksperimen-4) Ukuran ruangan : 200 cm \times 150 cm

Berdasarkan masukan tersebut didapatkan informasi sebagai berikut:

- **Informasi**
 - (Eksperimen-1) Matriks *cell* : $2 \times 2 = 4$ *cells*
 - (Eksperimen-2) Matriks *cell* : $3 \times 2 = 6$ *cells*
 - (Eksperimen-3) Matriks *cell* : $3 \times 3 = 9$ *cells*
 - (Eksperimen-4) Matriks *cell* : $4 \times 3 = 12$ *cells*

Matriks *cell* didapatkan berdasarkan masukan ukuran ruangan dan masukan ukuran terbesar *cell*.



Gambar 5.3: Diagram hubungan jumlah *cells* terhadap jumlah variabel

Berdasarkan eksperimen yang dilakukan, didapatkan hubungan jumlah *cells* terhadap jumlah variabel (n) yang dapat dilihat pada gambar 5.3. Jumlah variabel (n) merupakan jumlah kemungkinan penempatan kamera CCTV. Apabila ukuran ruangan semakin besar, maka ukuran matriks *cell* akan semakin besar. Apabila matriks *cell* semakin besar, maka akan titik penempatan kamera CCTV semakin banyak sehingga kemungkinan penempatan kamera CCTV juga akan semakin banyak. Dengan demikian, dapat disimpulkan bahwa apabila ukuran ruangan semakin besar, maka kemungkinan penempatan kamera CCTV juga akan semakin banyak.

5.4.2 Pengujian-2

Pada 2.2.1 diketahui bahwa pencarian solusi menggunakan algoritma *Balas's additive* dilakukan secara efisien tanpa melibatkan *exhaustive search*. Dengan menggunakan algoritma *Balas's additive*, solusi optimal masalah *binary integer programming* dapat ditemukan tanpa memeriksa seluruh 2^n kemungkinan solusi di mana n menunjukkan jumlah variabel. Untuk menguji hal tersebut, akan

dilakukan eksperimen perbandingan antara jumlah iterasi pada algoritma *Balas's additive* dengan iterasi *exhaustive search* yang berjumlah 2^n . Eksperimen ini dilakukan sebanyak 5 kali. Masukan yang digunakan untuk eksperimen ini dibagi menjadi 2 bagian:

- **Masukan tetap**

- Ukuran *cell* : 100 cm × 100 cm
- Jarak pandang kamera CCTV : 100 cm
- Sudut pandang kamera CCTV : 90°
- Ukuran terbesar *cell* : 50 cm

- **Masukan tidak tetap**

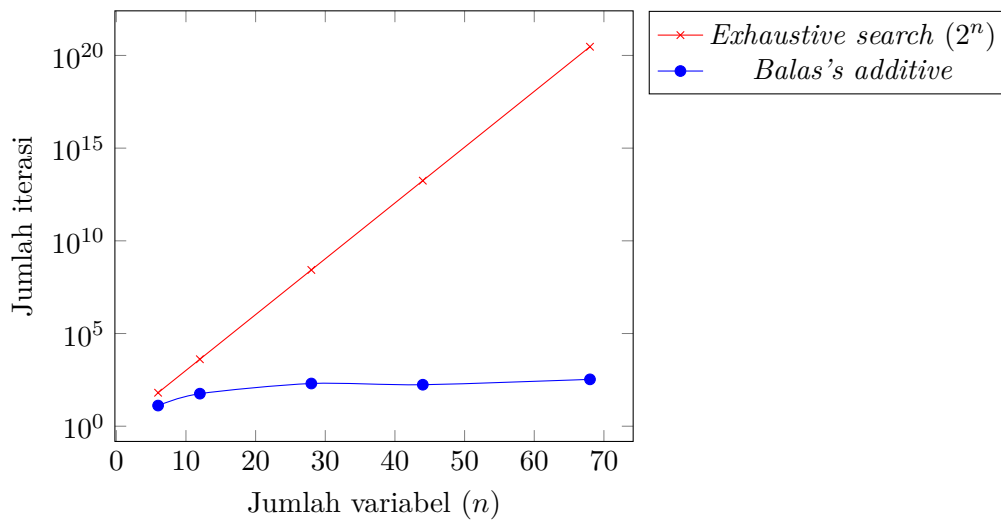
- (Eksperimen-1) Jumlah kemungkinan arah pandang kamera CCTV : 2 buah
- (Eksperimen-2) Jumlah kemungkinan arah pandang kamera CCTV : 4 buah
- (Eksperimen-3) Jumlah kemungkinan arah pandang kamera CCTV : 8 buah
- (Eksperimen-4) Jumlah kemungkinan arah pandang kamera CCTV : 12 buah
- (Eksperimen-5) Jumlah kemungkinan arah pandang kamera CCTV : 16 buah

Berdasarkan masukan tersebut didapatkan informasi sebagai berikut:

- **Informasi**

- (Eksperimen-1) Jumlah kemungkinan penempatan kamera CCTV : 6 buah
- (Eksperimen-2) Jumlah kemungkinan penempatan kamera CCTV : 12 buah
- (Eksperimen-3) Jumlah kemungkinan penempatan kamera CCTV : 28 buah
- (Eksperimen-4) Jumlah kemungkinan penempatan kamera CCTV : 44 buah
- (Eksperimen-5) Jumlah kemungkinan penempatan kamera CCTV : 68 buah

Setiap kemungkinan penempatan kamera CCTV merupakan variabel pada model masalah *binary integer programming* sehingga jumlah kemungkinan penempatan kamera CCTV menunjukkan jumlah variabel (n).



Gambar 5.4: Diagram perbandingan jumlah iterasi antara *Balas's additive* dengan *exhaustive search* (2^n)

Berdasarkan eksperimen yang dilakukan, didapatkan diagram perbandingan jumlah iterasi yang dapat dilihat pada gambar 5.4. Terlihat bahwa jumlah iterasi pada algoritma *Balas's additive* lebih kecil dibandingkan dengan iterasi pada *exhaustive search* yang berjumlah 2^n . Berdasarkan perbandingan tersebut, dapat disimpulkan bahwa algoritma *Balas's additive* dapat menemukan solusi optimal masalah *binary integer programming* secara efisien tanpa memeriksa seluruh 2^n kemungkinan solusi.

BAB 6

KESIMPULAN DAN SARAN

6.1 Kesimpulan

Berdasarkan penelitian yang telah dilakukan, terdapat beberapa hal yang dapat disimpulkan, yaitu:

1. Masalah ini dapat dimodelkan ke dalam bentuk masalah *binary integer programming*. Variabel-variabel pada bentuk masalah *binary integer programming* terdiri dari seluruh kemungkinan penempatan kamera CCTV. Kemungkinan penempatan kamera CCTV dapat dibangun dengan mengkombinasikan seluruh titik penempatan kamera CCTV dengan seluruh kemungkinan sudut arah pandang penempatan kamera CCTV. Fungsi tujuan pada model masalah *binary integer programming* ditujukan untuk meminimasi jumlah penempatan kamera CCTV. Untuk memastikan bahwa setiap bagian dalam ruangan tercakup oleh kamera CCTV, maka ditambahkan batasan-batasan yang menyatakan bahwa setiap *cell* dalam ruangan harus tercakup oleh setidaknya 1 penempatan kamera CCTV.
2. Pemodelan masalah ini ke dalam bentuk masalah *binary integer programming* dapat diselesaikan menggunakan algoritma *Balas's additive*. Karena masalah ini dapat dimodelkan ke dalam bentuk masalah *binary integer programming*, maka solusi masalah berupa penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan dapat ditemukan dengan menggunakan algoritma *Balas's additive*.
3. Perangkat lunak untuk menyelesaikan masalah ini dapat menerima masukan masalah dan menghasilkan solusinya. Perangkat lunak ini telah menerapkan hasil analisis pemodelan dan penyelesaian masalah sehingga solusi dari perangkat lunak ini merupakan solusi penempatan kamera CCTV yang berjumlah minimum yang dapat mencakup seluruh isi ruangan. Solusi ini ditampilkan perangkat lunak melalui visualisasi pada tampilan antarmuka grafis sehingga penempatan-penempatan kamera CCTV dapat dipahami dengan lebih mudah.
4. Pengujian pertama menghasilkan kesimpulan bahwa ukuran ruangan memiliki hubungan dengan ukuran model masalah *binary integer programming*. Berdasarkan eksperimen yang dilakukan, didapati bahwa apabila ukuran ruangan semakin besar, maka ukuran model masalah *binary integer programming* juga akan semakin besar.
5. Pengujian kedua menghasilkan kesimpulan bahwa solusi optimal yang didapatkan dengan menggunakan algoritma *Balas's additive* dapat ditemukan tanpa perlu memeriksa seluruh 2^n kemungkinan solusi. Hal ini menandakan bahwa penyelesaian masalah *binary integer programming* menggunakan algoritma *Balas's additive* merupakan metode yang efisien karena tidak melibatkan *exhaustive search*.

6.2 Saran

Terdapat beberapa saran untuk pengembangan penelitian ini lebih lanjut, yaitu:

1. Pemodelan masalah yang dilakukan pada penelitian ini masih dibatasi pada bidang 2 dimensi sehingga pemodelan masalah dapat dikembangkan lebih lanjut pada bidang 3 dimensi.
2. Ruang yang digunakan dalam penelitian ini berbentuk sederhana (persegi panjang) tanpa adanya penghalang di dalamnya. Ruang ini dapat dikembangkan lebih lanjut sehingga dapat berbentuk selain persegi panjang dan/atau memiliki suatu penghalang di dalamnya.
3. Kamera CCTV yang digunakan dalam penelitian ini merupakan kamera statis sehingga dapat dikembangkan dengan menggunakan kamera yang dapat bergerak.

DAFTAR REFERENSI

- [1] Winston, W. L. dan Goldberg, J. B. (2004) *Operations research: applications and algorithms*. Thomson/Brooks/Cole Belmont^ eCalif Calif.
- [2] Balas, E. (1965) An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, **13**, 517–546.
- [3] Narendra, P. M. dan Fukunaga, K. (1977) A branch and bound algorithm for feature subset selection. *IEEE Transactions on computers* , **?**, 917–922.
- [4] Horster, E. dan Lienhart, R. (2006) Approximating optimal visual sensor placement. *2006 IEEE International Conference on Multimedia and Expo*, pp. 1257–1260. IEEE.

LAMPIRAN A
KODE PROGRAM

LAMPIRAN B
HASIL EKSPERIMEN