Penerapan Algoritma Backtracking dalam Permainan Mainarizumu

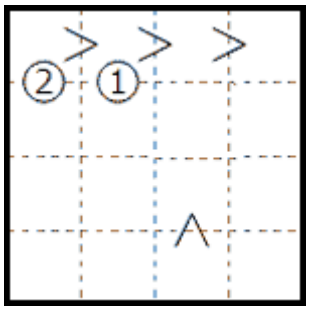
Nathaniel Jason / 13519108
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13519108@std.stei.itb.ac.id

Abstraksi—Mainarizumu adalah sebuah permainan yang berasal dari Jepang, diterbitkan oleh Nikoli, penerbit asal Jepang yang memiliki spesialisasi dalam permainan terutama *logic puzzles*. Mainarizumu dimainkan dalam kotak persegi yang berukuan N x N. Aturan permainannya cukup mudah yaitu mengisi angka 1 sampai N tetapi dengan beberapa batasan yaitu, tidak boleh mengisi angka yang sama pada baris atau kolom yang sama, jika terdapat panah dari sebuah kotak yang menunjuk ke kotak lain maka kotak yang ditunjuk harus memiliki nilai yang lebih kecil dari kotak yang menunjuk, dan jika terdapat angka dalam lingkaran yang menghubungkan dua kotak maka selisih dari angka pada kedua kotak tersebut harus sesuai dengan angka yang ada pada lingkaran tersebut. Terdapat banyak metode dalam melakukan pencarian solusi untuk persoalan ini, salah satu metode yang dipakai kali ini adalah *backtracking* atau runut balik. Secara singkat, cara kerja algoritma *backtracking* adalah dengan cara melakukan pengecekan untuk semua kemungkinan menuju solusi, jika tidak menuju solusi maka kemungkinan jawaban tersebut akan ditinggal dan akan melakukan runut balik.

Kata kunci—Mainarizumu; pencarian solusi; *backtracking*

I. PENDAHULUAN

Mainarizumu adalah sebuah teka-teki logika atau *logic puzzles* yang diterbitkan oleh Nikoli, penerbit asal Jepang yang memiliki spesialisasi dalam permainan terutama pada kategori permainan *logic puzzles*. Permainan Mainarizumu termasuk ke dalam persoalan kombinatorik. Mainarizumu dimainkan pada kotak persegi atau *square grid* yang berukuran N x N. Kotak - kotak yang ada pada persegi disebut dengan *cell*. Antara 2 *cell* yang berhimpitan ada kemungkinan terdapat tanda lebih besar / '>' atau tanda lebih kecil '<'. Antara 2 *cell* yang berhimpitan juga bisa terdapat angka dalam lingkaran.



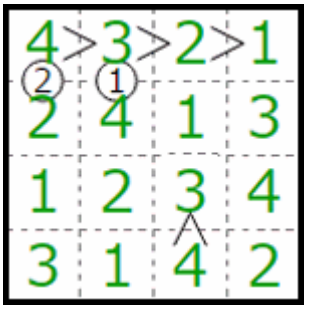
Gambar 1. Contoh permainan Mainarizumu

https://www.janko.at/Raetsel/Mainarizumu/index.htm

Mainarizumu pada kotak persegi berukuran N x N dimainkan dengan cara mengisi *cell* - *cell* dengan angka 1 sampai N inklusif. Pengisian harus dilakukan tanpa melanggar aturan permainan Mainarizumu yang ada. Aturan - aturannya adalah sebagai berikut.

1. Tidak ada angka yang berulang pada kolom atau baris yang sama
2. Jika antara dua *cell* terhubung dengan tanda lebih besar atau lebih kecil maka pengisian angka pada kedua *cell* tersebut harus mengikuti tanda tersebut.
3. Jika antara dua *cell* terhubung dengan angka dalam lingkaran, maka angka dari kedua *cell* tersebut harus memiliki selisih tepat dengan angka dalam lingkaran tersebut.

Pemain dapat dikatakan menyelesaikan permainan jika telah mengisi semua *cell* tanpa melanggar aturan. Suatu permainan Mainarizumu dapat memiliki lebih dari satu solusi. Berikut adalah salah satu contoh solusi untuk permainan Mainarizumu pada gambar 1.



Gambar 2. Contoh solusi permainan Mainarizumu

https://www.janko.at/Raetsel/Mainarizumu/index.htm

Terdapat beberapa metode komputasi atau algoritma yang dapat kita gunakan untuk menyelesaikan permainan Mainarizumu. Beberapa metode diantaranya adalah *brute force* dan *backtracking*. *Brute force* dilakukan dengan cara mengecek semua kemungkinan solusi yang ada. Metode ini dapat dikatakan memiliki kecepatan yang cukup lama dikarenakan kita mencoba semua kemungkinan solusi walaupun dalam tahap percobaan itu pastinya banyak dari

kemungkinan solusi itu tidak mengarah ke solusi sesungguhnya. Algoritma *backtracking* menyelesaikan permasalahan tersebut. Algoritma *backtracking* juga mencoba menelusuri semua kemungkinan solusi yang ada tetapi hanya pada kemungkinan solusi yang mengarah ke solusi sesungguhnya. Algoritma *backtracking* menggunakan konsep *exhaustive search* dan DFS atau *depth first search*.

II. LANDASAN TEORI

A. *Combinatorial Problems*

Combinatorial Problems atau persoalan kombinatorial adalah permasalahan yang melibatkan *grouping, ordering,* atau *assignment* dari suatu kumpulan objek yang bersifat diskrit dan *finite,* memenuhi suatu aturan atau kondisi tertentu. Solusi persoalan adalah suatu kumpulan objek yang bersifat diskrit atau *finite*. Dalam proses memecahkan *combinatorial problems* kita akan bertemu dengan *candidate solutions* atau kandidat solusi. *Candidate solutions* adalah kombinasi dari semua kemungkinan yang ada pada solusi persoalan. *Candidate solutions* tidak perlu memenuhi aturan atau kondisi yang sudah ditetapkan. Solusi akhir adalah semua kandidat solusi yang memenuhi kondisi atau kondisi tertentu.

Combinatorial problems dapat berbentuk *problem* atau *problem instance*. *Problem* adalah bentuk generalisasi dari suatu kumpulan *problem instance*

Misalnya dalam *Traveling Salesman Problem* (TSP), *problemnya* adalah “diberikan graf dengan kota - kota sebagai *node*nya, diberikan kota sembarang X, tentukan rute dengan *cost* terkecil yang mengunjungi semua kota pada graf dan mulai dan berakhir pada kota X” . Solusi dari *problem* tersebut adalah suatu algoritma yang mencari rute terpendek untuk TSP yang dimulai pada kota sembarang.

Contoh dari *problem instance* untuk kasus TSP adalah “diberikan graf dengan kota - kota sebagai *node*nya, diberikan kota sembarang X, tentukan rute dengan *cost* terkecil yang mengunjungi semua kota pada graf dan mulai dan berakhir pada kota Alfa” dengan asumsi kota Alfa ada pada graf tersebut. Solusi dari *problem instance* tersebut adalah rute terpendek untuk TSP yang dimulai pada kota Alfa.

Dapat dilihat dari contoh Traveling Salesman Problem diatas, *problem* adalah bentuk yang lebih general dari *problem instance*.

Combinatorial problems dapat dibagi jadi 2 kategori yaitu *decision problems* dan *optimisation problems*. *Optimisation problem* dapat dilihat sebagai hasil generalisasi dari *decision problems.*

Decision problems memiliki solusi yaitu semua *candidate solutions* yang memenuhi suatu kondisi tertentu. *Decision problems* dapat dibagi menjadi 2 kategori, yaitu *search variant* dan *decision variant*. *Search variant* akan mencari semua solusi yang mungkin, sedangkan pada *decision variant* akan menentukan apakah terdapat solusi yang memenuhi atau tidak. *Search variant* dan *decision variant* memiliki keterhubungan yang cukup dekat, sehingga sering kali algoritma yang digunakkan untuk *search variant* dapat kita sesuaikan untuk *decision variant* begitupun sebaliknya.

Optimisation problem dapat dilihat sebagai suatu bentuk dari hasil generalisasi *decision problems*. Pada *optimisation problems* terdapat suatu fungsi objektif yang digunakkan untuk menghitung kualitas dari suatu kandidat solusi. *Optimisation problems* akan memiliki solusi berupa kandidat solusi yang memenuhi suatu kondisi tertentu dan juga merupakan kandidat solusi dengan kualitas paling baik atau buruk tergantung jenis *optimisation problems*nya. *Optimisation problems* dapat dibagi menjadi 2 yaitu *minimisation problems* (solusi dengan kualitas paling rendah) dan *maximisation problems* (solusi dengan kualitas paling tinggi). Semua *minimisation problems* dapat direkonstruksi menjadi *maximisation problems* dan juga sebaliknya.

Permainan Mainarizumu adalah salah satu contoh dari *combinatorial problems*. Disini permainan Mainarizumu memiliki permasalahan *assignment* untuk setiap *cell* yang ada. Objek yang bisa di-*assign*-pun adalah suatu kumpulan objek yang diskrit dan *finite*, yaitu angka dari 1 sampai N pada permainan Mainarizumu berukuran N x N. Permainan Mainarizumu termasuk kedalam kategori *decision problems* dengan varian *search variant*, karena disini kita ingin mencari semua kombinasi pengisian yang mungkin dan tidak melanggar aturan - aturan yang sudah ada.

B. *Exhaustive Search*

Exhaustive search adalah suatu teknik pencarian solusi dengan menggunakan konsep *brute force*. *Exhaustive search* seringkali digunakan dalam persoalan - persoalan kombinatorik atau *combinatorial problems*. Berikut adalah langkah utama dari *exhaustive search*.

1. Enumerasi setiap kemungkinan solusi atau *candidate solution.*
2. Evaluasi setiap kemungkinan solusi lalu simpan solusi terbaik yang ditemukan sejauh ini.
3. Umumkan solusi terbaik saat pencarian berakhir.

Secara teoritis *exhaustive search* akan menghasilkan solusi, tetapi salah satu kekurangan dari metode ini adalah waktu dan sumberdaya yang digunakan untuk pencarian sangat besar, karena menggunakan sifat - sifat dari metode *brute force*.

Exhaustive search merupakan salah satu konsep yang digunakan pada algoritma *backtracking*. Algoritma *backtracking* adalah bentuk lebih ‘pintar’ dari *exhaustive search*. Algoritma *backtracking* tetap memiliki tendensi untuk mengenumerasi semua kemungkinan kandidat solusi yang ada tetapi hanya yang menuju solusi sesungguhnya.

C. *Depth First Search*

Depth First Search atau DFS adalah salah satu dari algoritma traversal graf. Algoritma traversal graf akan mengunjungi simpul dengan cara sistematik. DFS dilakukan dengan cara melakukan pencarian dengan cara mendalam. Secara umum, algoritma DFS adalah sebagai berikut.

1. Kunjungi simpul *v*
2. Kunjungi simpul *w* yaitu simpul lain yang memiliki *vertex* (bertetangga) ke simpul *v*
3. Ulangi langkah 1 dan 2 dimulai dari simpul *w*.

4. Ketika mencapai suatu simpul *u* dimana semua simpul bertetangga telah dikunjungi, lakukan runut balik ke simpul terakhir yang masih memiliki tetangga yang belum dikunjungi.
5. Pencarian akan berakhir jika tidak ada lagi simpul yang belum dikunjungi yang merupakan tetangga dari simpul yang telah dikunjungi.

Berikut adalah algoritma DFS secara umum.

```
procedure DFS(input v:integer)
{Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS

Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpul yang dikunjungi ditulis ke layar
}
Deklarasi
      w: integer

Algoritma
      write(v)
      dikunjungi(v) ← true
      for w ← 1 to n do
            if A[v, w] = 1 then {simpul v dan w bertetangga}
                  if not dikunjungi[w] then
                        DFS(w)
                  endif
            endif
      endfor
```

Gambar 3. Algoritma DFS secara umum

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

Konsep DFS digunakan dalam algoritma *backtracking*. Kita membangkitkan simpul - simpul sesuai dengan aturan dan metode pada DFS. Pencarian pada algoritma *backtracking* dilakukan secara mendalam seperti pada DFS.

D. *Backtracking*

Algoritma *backtracking* atau runut-balik adalah salah satu bentuk *improvement* dan perbaikan dari algoritma *exhaustive search*. Algoritma *backtracking* berbeda dengan *exhaustive search* yang dimana *exhaustive search* itu akan mengenumerasi semua kemungkinan solusi. Algoritma *backtracking* hanya akan mengenumerasi pilihan yang mengarah menuju solusi sesungguhnya, solusi yang tidak mengarah ke solusi sesungguh tidak akan dipertimabangkan ataupun dieksplorasi lagi. Hal ini juga biasa disebut sebagai proses *pruning*, yaitu memangkas simpul - simpul yang tidak mengarah ke solusi.

Properti umum dari algoritma *backtracking* adalah sebagai berikut.

1. Solusi persoalan

   Solusi dinyatakan sebagai vektor dengan *n-tuple*: X = (x_1, x_2, …, x_n) dimana x_i adalah bagian dari S_i. S_i adalah kemungkinan nilai pada tahap i. Umumnya $S_1 = S_2 = \ldots = S_n$. Contohnya pada persoalan *1/0 knapsack* $S_i = \{0, 1\}$, artinya x_i dapat bernilai 1 atau 0.

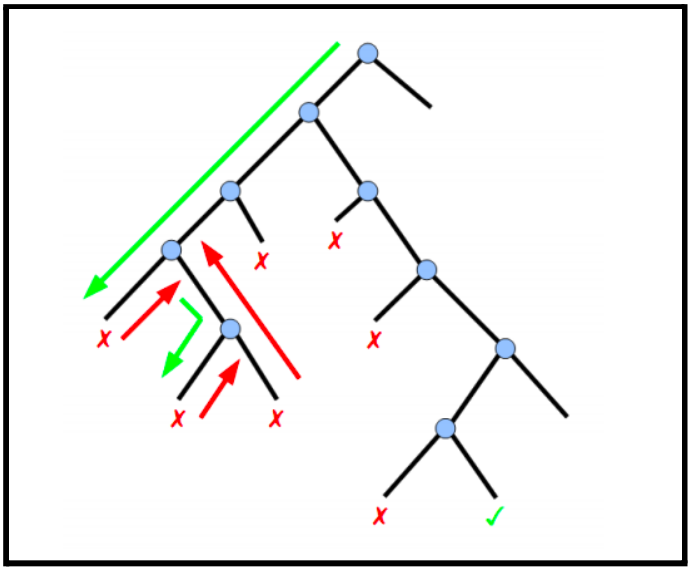
2. Fungsi pembangkit nilai x_k

   Fungsi pembangkit dinyatakan sebagai predikat(), dimana T(x[1], x[2], …, x[k - 1]) memabngkitkan nilai untuk x_k, yang merupakan komponen dari vektor solusi.

3. Fungsi pembatas

   Fungsi pembatas biasa disebut juga *bounding function*. Fungsi pembatas dinyatakan sebagai predikat B(x_1, x_2, …, x_n). B akan mengembalikan nilai *true* jika (x_1, x_2, …, x_n) mengarah ke solusi. Yang dimaksud dengan mengarah ke solusi adalah tidak melanggar kendala. Jika B mengembalikan nilai *true*, maka pembangkitan untuk nila x_{k+1} dilanjutkan. Jika B mengembalikan nilai *false* maka (x_1, x_2, …, x_n) akan dibuang.

Semua kemungkinan solusi dari suatu persoalan biasa disebut dengan *solution space* atau ruang solusi. Pada algoritma *backtracking*, ruang solusi diorganisasikan ke dalam struktur data pohon dengan akar. Tiap simpul pada pohon tersebut menyatakan *state* dari persoalan, sedangkan untuk cabang diberi label dengan nilai *assignment* untuk x_i. Lintasan dari akar menuju ke daun akan membentuk ruang solusi. Teknik pengorganisasian pohon ruang solusi disebut sebagai pohon ruang status atau *state space tree*.



Gambar 4. Contoh Algoritma *Backtracking*

http://www.w3.org/2011/Talks/01-14-steven-phenotype/

Prinsip pencarian solusi dengan algoritma *backtracking* dilakukan dengan cara membangkitkan simpul - simpul status sehingga menghasilkan lintasan dari akar ke daun. Aturan yang digunakan untuk pembangkitan simpul adalah *depth first search* atau pencarian secara mendalam. Untuk semua simpul yang berhasil dibangkitkan akan dinamakan simpul hidup atau *live node*, sedangkan simpul hidup yang sedang dilakukan proses perluasan dinamakan dengan simpul-E atau *expand-node*.

Setiap kali *expand node* dilakukan perluasan, lintasan yang dibangung akibat itu bertambah panjang. Lintasan yang tidak mengarah ke solusi akan dimatikan atau tidak diperluas lagi dan menjadi simpul mati atau *dead node*. Fungsi pembatas atau *bounding function* yang akan menentukan apakah suatu lintasan mengarah ke solusi atau tidak.

Jika suatu proses pencarian menghasilkan lintasan yang berakhir pada *dead node*, maka proses pencarian akan melakukan runut balik ke simpul orangtuanya yang masih bisa diperluas. Simpul tersebutlah yang menjadi *expand node* yang baru. Proses pencarian akan berakhir jika telah sampai *goal node*.

Berikut adalah implementasi algoritma *backtracking* metode rekursif secara umum.

```
procedure RunutBalikR(input k:integer)
{Mencari semua solusi persoalan dengan metode runut-balik
Skema rekursif

Masukan: k, yaitu indeks komponen vektor solusi, x[k]
Diasumsikan x[1], x[2], ..., x[k - 1] sudah ditentukan

Keluaran : semua solusi x = (x[1], x[2], ..., x[n])
}
Algoritma
for setiap x[k] anggota (x[1], x[2], ..., x[k - 1]) do
      if B(x[1], x[2], ..., x[k]) = true then
            if (x[1], x[2], ..., x[k] adalah lintasan dari
            Akar ke simpul solusi then
                  write(x[1], x[2], ..., x[k])
                  {cetak solusi}
            endif
            if k < n then
                  RunutBalikR(k + 1)
            endif
      endif
endfor
```

Gambar 5. Algoritma *backtracking* rekursif

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf

Prosedur RunutBalikR menerima parameter *k* yaitu indeks dari komponen vektor solusi x[k]. Pemanggilan pertama untuk menyelesaikan persoalan dengan prosedur RunutBalikR adalah dengan cara RunutBalikR(1), dengan asumsi vektor solusi dimulai dari indeks 1.

Pada proses algoritma *backtracking*, setiap simpul dalam pohon ruang status atau *state space tree*, kecuali simpul daun berasosiasi dengan pemanggilan yang bersifat rekursif. Jika jumlah simpul yang ada di pohon ruang status atau *state space tree* adalah 2^n atau n!, maka untuk kasus terburuk atau *worst case scenario*, algoritma *backtracking* memiliki kompleksitas $O(p(n)2^n)$ atau $O(p(n)n!)$, $p(n)$ adalah polinom derajat n yang merupakan kompleksitas untuk komputasi setiap simpul.

Berikut adalah implementasi algoritma *backtracking* metode iteratif secara umum.

```
procedure RunutBalikR(input k:integer)
{Mencari semua solusi persoalan dengan metode runut-balik
Skema iteratif

Masukan: k, yaitu indeks komponen vektor solusi, x[k]
Diasumsikan x[1], x[2], ..., x[k - 1] sudah ditentukan

Keluaran : semua solusi x = (x[1], x[2], ..., x[n])
}
Algoritma
while k != 0 do
      if terdapat nilai x[k] yang belum dicoba sedemikian
      sehingga x[k] anggota T(x[1], x[2], ..., x[k - 1]) and
      B(x[1], x[2], ..., x[k]) = true then
            if (x[1], x[2], ..., x[k]) adalah lintasan dari
            akar ke simpul solusi then
                  write(x[1], x[2], ..., x[k])
                  {cetak solusi}
            endif
            k ← k + 1
      else
            k ← k + 1
      endif
endwhile
```

Gambar 5. Algoritma *backtracking* iterativ

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf

III. METODE PENYELESAIAN MASALAH

Proses penyelesaian permainan Mainarizumu dimulai dari tahap menentukan properti umum algoritma *backtracking* yang terdiri dari solusi persoalan, fungsi pembangkit nilai x_k, dan fungsi pembatas atau *bounding function*.

Solusi persoalan untuk permasalahan permainan Mainarizumu dengan ukuran *n* x *n* dinyatakan sebagai matriks berukuran *n* x *n*. Berikut adalah contoh representasi fisik dari solusi persoalan permainan Mainarizumu.

$$A = \begin{pmatrix} a_{11} & a_{12} \cdots\cdots a_{1j} & \cdots\cdots a_{1n} \\ a_{21} & a_{22} \cdots\cdots a_{2j} \cdots\cdots & a_{2n} \\ \vdots & \vdots \quad \vdots & \vdots \\ a_{i1} & a_{i2} \cdots\cdots a_{ij} \cdots\cdots & a_{in} \\ \vdots & \vdots \quad \vdots & \vdots \\ a_{n1} & a_{n2} \cdots\cdots a_{nj} \cdots\cdots & a_{nn} \end{pmatrix}$$

Gambar 6. Solusi persoalan

https://www.kontensekolah.com/2017/09/menghitung-determinan-matriks-ordo-4x4.html

Matriks A pada gambar 6 adalah contoh bentuk representasi fisik dari solusi persoalan. Jika $a_{i,j}$ adalah elemen dari matriks A, maka $a_{i,j}$ merupakan anggota dari $S_{i,j}$ dimana

$S_{1,1} = S_{1,2} = S_{1,3} = \ldots = S_{2,1} = \ldots = S_{n,n}$. Untuk permasalahan permainan Mainarizumu, $S_{i,j} = \{1, 2, \ldots, n\}$.

Untuk persoalan permainan Mainaruzumu, solusi persoalan, semua elemen pada matriks A akan diinisialisasi dengan nilai yang tidak mungkin ada pada $S_{i,j}$, sebagai contoh nilai -1 dikarenakan $S_{i,j}$ hanya akan berisi angka dari satu sampai *n*.

Untuk kasus permainan Mainarizumu, fungsi pembangkit $x_{i,j}$ diimplementasi dengan sederhana, nilai yang harus dibangkitkan untuk semua *cell* adalah sama, yaitu *integer* dari 1 sampai *n* secara inklusif.

Berikut adalah implementasi dari fungsi pembatas.

```
function IsValidMainarizumu(
      input A : array of integer,
      input x : integer,
      input y : integer,
      input n : integer)
{Mengembalikan true jika array solusi A mengarah ke solusi
Mengembalikan false jika sebaliknya

Masukan:
   -  A, array of integer yang merupakan solusi persoalan
      berukuran n x n
   -  x, integer yang merupakan indeks baris iterasi sekarang
   -  y, integer yang merupakan indeks kolom iterasi sekarang
   -  n, integer yang merupakan ukuran permainan Mainarizumu

Keluaran :
   -  true jika array solusi A mengarah ke solusi
   -  false jika array solusi A tidak mengarah ke solusi
}
Deklarasi
isValid : boolean
i : integer

Algoritma
isValid ← true

{Mengecek apakah ada nilai yang sama pada baris yang sama}
for i ← 1 to n do
      if (y != i) then
            isValid ← isValid and (A[x][y] != A[x][i])
      endif
endfor

{Mengecek apakah ada nilai yang sama pada kolom yang sama}
for i ← 1 to n do
      if (x !=  i) then
            isValid ← isValid and (A[x][y] != A[i][y])
      endif
endfor

{Mengecek apakah array A valid untuk constraint
tanda lebih besar atau lebih kecil}

{Mengecek apakah array A valid untuk constraint
angka dalam lingkaran}

return isValid
```

Gambar 7. Implementasi fungsi pembatas

Fungsi pembatas atau *bounding function* dapat diimplementasi dengan fungsi yang mengembalikan nilai *true* jika *candidate solution* mengarah ke solusi sesungguhnya dan mengembalikan nilai *false* untuk sebaliknya. Terdapat 4 *constraint* yang menentukan apakah suatu *candidate solution* mengarah ke solusi sesungguhnya. Berikut adalah beberapa batasan yang harus di cek.

1. Tidak ada nilai yang sama pada baris yang sama.
2. Tidak ada nilai yang sama pada kolom yang sama.
3. Memenuhi *constraint* untuk *cell* yang dihubungkan tanda lebih besar atau lebih kecil.
4. Memenuhi *constraint* untuk *cell* yang dihubungkan angka dalam lingkaran.

Berikut adalah implementasi dari penyelesaian Mainarizumu.

```
procedure SolveMainarizumu(
      input A : array of integer,
      input x : integer,
      input y : integer,
      input n : integer)
{Menyelesaikan permainan Mainarizumu secara rekursif dan backtracking

Masukan :
   -  A, array of integer yang merupakan solusi persoalan
      berukuran n x n
   -  x, integer yang merupakan indeks baris iterasi sekarang
   -  y, integer yang merupakan indeks kolom iterasi sekarang
   -  n, integer yang merupakan ukuran permainan Mainarizumu

Keluaran : Semua solusi permainan Mainarizumu dalam bentuk matriks n x n
jika ditemukan solusi
}
Deklarasi
i : integer
xNew : integer
yNew : integer

Algoritma
for i ← 1 to n do
      A[x][y] ← i
      if (IsvalidMainarizumu(A, x, y, n)) then
      {hanya melanjutkan pencarian pada matriks solusi A yang
      mengarah ke solusi}

            if (x = n and y = n) {basis rekursi} then
            {mengecek apakah sudah mencapai daun atau belum, tampilkan
            Jawaban jika sudah sampai daun}
                  WriteSolutionMainarizumu(A, n)

            else
                  {lakukan pengecekan untuk cell berikutnya}
                  yNew ← (y + 1) mod (n + 1)
                  xNew <- x
                  {pindah ke kolom selanjutnya}

                  if (yNew = 0) then
                  {pindah ke baris selanjutnya jika sudah sampai ujung}
                        yNew ← 1
                        xNew ← x + 1
                  endif

                  SolveMainarizumu(A, xNew, yNew, n) {rekursi}
            endif
      endif
endfor
```

Gambar 8. Implementasi penyelesaian Mainarizumu

Permainan Mainarizumu memiliki penyelesaian yang diimplementasi dengan menggunakan *procedure* yang bersifat rekursif.

Secara garis besar, langkah penyelesaiannya adalah sebagai berikut.

1. Iterasi semua kemungkinan nilai yang dapat kita *assign* ke *cell.*
2. *Assign* nilai baru ke *cell* saat ini.
3. Cek apakah lintasan mengarah ke solusi.
4. Jika tidak mengarah solusi maka *backtrack*, lakukan langkah 1.
5. Jika sudah mencapai daun, tuliskan solusi ke layar, lakukan langkah 1.
6. Jika belum mencapai daun, lanjutkan ke *cell* berikutnya.

Pada implementasi penyelesaian Mainarizumu, dapat dilihat *backtrack* tidak secara eksplisit dituliskan. Hal ini dikarenakan sifat rekursif. Untuk solusi yang tidak mengarah ke solusi, tidak akan dilakukan pemanggilan rekursif. Perilaku seperti ini sama saja seperti *backtracking*. Alur program akan balik ke si pemanggil fungsi rekursif yang terakhir. Kalau diilustrasikan pada graf, ini sama saja seperti pergi ke *parent node* dari *node* saat ini.

Dalam kenyataannya, untuk menggunakan *procedure* SolveMainarizumu, harus ada beberapa inisialisasi yang perlu dilakukan. Pertama, inisialisasi semua nilai pada matriks solusi persoalan dengan suatu nilai yang tidak mungkin ada pada $S_{i,j}$, sebagai contoh kita dapat menggunakan nilai -1 dikarenakan nilai yang mungkin hanya dari 1 sampai *n*. Selanjutnya, pemanggilan pertama *procedure* SolveMainarizumu adalah sebagai berikut.

```
SolveMainarizumu(A, 1, 1, n)
```

Gambar 9. Pemanggilan pertama SolveMainarizumu

IV. STUDI KASUS

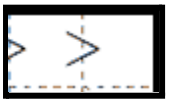
Kita akan menggunakan gambar 1 sebagai contoh persoalan yang akan kita selesaikan sebagai studi kasus.

A. Penyelesaian dengan bahasa pemrograman Python

Permainan Mainarizumu dalam implementasinya akan di-*input* dari sebuah file eksternal. File eksternal akan terbagi menjadi tiga bagian yang akan dipisahkan dengan suatu baris berisi "***".

Bagian pertama akan berisi tentang ukuran permainan Mainarizumu.

Bagian kedua akan berisi tentang *constraint inequality*. Yang dimaksud *constraint inequality* adalah seperti berikut ini.



Gambar 10. Contoh *constraint inequality*

Contoh *constraint inequality* dapat ditemukan pada gambar 1 untuk *cell* 0,2 dan 0,3, yang berarti *cell* 0,2 harus memiliki nilai yang lebih besar dari *cell* 0,3. *Constraint* ini akan diimplementasikan dengan 2 baris, dimana baris pertama adalah letak *cell* yang harus lebih besar dari *cell* pada baris kedua. Letak *cell* adalah 2 angka yang dibatasi spasi, angka pertama adalah indeks baris, angka kedua adalah indeks kolom. Pada kasus *constraint inequality* yang ada lebih dari 1, akan dibatasi dengan tanda "*" pada file eksternalnya.

Bagian ketiga adalah untuk *constraint* selisih. Contoh dari *constraint* ini adalah sebagai berikut.

Gambar 11. Contoh *constraint* selisih

Contoh *constraint* selisih dapat ditemukan pada gambar 1 untuk *cell* 0,0 dan 1,0, yang berarti selisih antara *cell* 0,0 dan *cell* 1,0 harus tepat 1. *Constraint* ini akan diimplementasikan dengan 3 baris. Baris pertama dan kedua menyatakan letak *cell* pertama dan kedua. Baris ketiga menyatakan selisihnya. Letak *cell* adalah 2 angka yang dibatasi spasi, angka pertama adalah indeks baris, angka kedua adalah indeks kolom. Pada kasus *constraint* selisih yang ada lebih dari 1, akan dibatasi dengan tanda "*" pada file eksternalnya.

Berikut adalah contoh *input* file eksternal yang akan diterima program untuk studi kasus permainan Mainarizumu pada gambar 1.

```
4
***
0 0
0 1
*
0 1
0 2
*
0 2
0 3
*
3 2
2 2
***
0 0
1 0
2
*
0 1
1 1
1

```

Gambar 12. Contoh input file eksternal

Dapat kita lihat, bagian pertama ada pada baris 1, bagian kedua ada pada baris 3 sampai 13, bagian ketiga ada pada baris 15 sampai 21.

Letak *cell* akan direpresentasikan sebagai *tuple* berukuran 2 dimana indeks pertamanya merupakan indeks baris dan indeks keduanya adalah indeks kolom. Informasi permainan Mainarizumu akan disimpan pada python dengan struktur data sebagai berikut.

1. Ukuran permainan Mainarizumu : *integer*
2. *Constraint inequality* : *array of tuple* (ukuran 2) *of* letak *cell*. Letak *cell* pada indeks pertama di *tuple* merupakan letak *cell* yang nilainya harus lebih besar. Letak *cell* pada indeks kedua di *tuple* merupakan letak *cell* yang nilainya harus lebih kecil.
3. *Constraint* selisih : *array of tuple* (ukuran 3) of letak *cell* dan selisih. Indeks pertama dan kedua *tuple* akan berisi letak *cell* dengan *constraint* terkait. Indeks ketiga *tuple* adalah nilai selisih yang harus dipenuhi.

Berikut adalah ilustrasi penyimpanan permainan Mainarizumu pada bahasa pemrograman Python.

```python
# ukuran permainan Mainarizumu
n = 4

# constraint inequality
constraintIneq = [
    (
        (0, 0),
        (0, 1)
    ),
    (
        (0, 1),
        (0, 2)
    ),
    (
        (0, 2),
        (0, 3)
    ),
    (
        (3, 2),
        (2, 2)
    )
]

# constraint selisih
constrainDiff = [
    (
        (0, 0),
        (1, 0),
        2
    ),
    (
        (0, 1),
        (1, 1),
        1
    )
]
```

Gambar 13. Ilustrasi penyimpanan permainan Mainarizumu pada bahasa pemrograman Python

Berikut adalah implementasi fungsi tambahan untuk mengecek apakah 2 posisi itu sama atau tidak. Fungsi akan mengembalikan *true* jika posisi sama.

```python
def isSamePosition(position1, position2):
    # position1 and position2 is a tuple with size of 2
    # first index represents the index of row
    # second index represents the index of column

    return position1[0] == position2[0] and position1[1] == position2[1]
```

Gambar 14. Implementasi fungsi isSamePosition dalam bahasa pemrograman Python

Berikut adalah implementasi *bounding function* pada bahasa pemrograman Python.

```python
def isValidMainarizumu(A, x, y, n, ci, cd):
    # return True if solution is moving towards the real solution
    # false otherwise

    # A : matrix of integer - temporary solution
    # x : integer - current row index
    # y : integer - current column index
    # n : integer - size of Mainarizumu board
    # ci : array of tuple - inequality constraint
    # cd : array of tuple - diffrence contraint

    isValid = True

    # check for same value on current row
    for i in range(n):
        if (y != i):
            isValid = isValid and (A[x][y] != A[x][i])

    # check for same value on current column
    for i in range(n):
        if (x != i):
            isValid = isValid and (A[x][y] != A[i][y])

    # check for inequality constraint
    for c in ci:
        if (isSamePosition(c[0], (x, y))):
            rowLess = c[1][0]
            colLess = c[1][1]
            valLess = A[rowLess][colLess]
            if valLess != UNDEFINED:
                valMore = A[x][y]
                isValidInequality = valMore > valLess
                isValid = isValid and isValidInequality

        elif (isSamePosition(c[1], (x, y))):
            rowMore = c[0][0]
            colMore = c[0][1]
            valMore = A[rowMore][colMore]
            if valMore != UNDEFINED:
                valLess = A[x][y]
                isValidInequality = valMore > valLess
                isValid = isValid and isValidInequality

    # check for diffrence constraint
    for c in cd:
        diff = c[2]
        if (isSamePosition(c[0], (x, y))):
            row2 = c[1][0]
            col2 = c[1][1]
            val2 = A[row2][col2]
            if val2 != UNDEFINED:
                val1 = A[x][y]
                isValidDiffrence = abs(val1 - val2) == diff
                isValid = isValid and isValidDiffrence

        elif (isSamePosition(c[1], (x, y))):
            row1 = c[0][0]
            col1 = c[0][1]
            val1 = A[row1][col1]
            if val1 != UNDEFINED:
                val2 = A[x][y]
                isValidDiffrence = abs(val1 - val2) == diff
                isValid = isValid and isValidDiffrence

    return isValid
```

Gambar 15. Implementasi *bounding function* (isValidMainarizumu) dalam bahasa pemrograman Python

Berikut adalah implementasi fungsi tambahan untuk menampilkan solusi Mainarizumu ke layar.

```python
def printSolutionMainarizumu(A):
    # print out the solution for a Mainarizumu puzzle

    # A : matrix of integer - Mainarizumu solution

    for row in A:
        for el in row:
            print(el, end = " ")
        print()
```

Gambar 16. Implementasi printSolutionMainarizumu dalam bahasa pemrograman Python

Berikut adalah implementasi prosedur untuk menyelesaikan permainan Mainarizumu secara rekursif dan menggunakan algoritma *backtracking*.

```python
def solveMainarizumu(A, x, y, n, ci, cd):
    # solve a Mainarizumu puzzle recursively
    # and using backtracking algorithm
    # print all solution

    # A : matrix of integer - temporary solution
    # x : integer - current row index
    # y : integer - current column index
    # n : integer - size of Mainarizumu board
    # ci : array of tuple - inequality constraint
    # cd : array of tuple - diffrence contraint

    for newVal in range(1, n + 1):
        ANew = [[el for el in row] for row in A]
        ANew[x][y] = newVal

        # check with bounding function
        if (isValidMainarizumu(ANew, x, y, n, ci, cd)):

            # check if current node is leaf
            # stop searching if current node is leaf

            # recursive base
            if ((x == (n - 1)) and (y == (n - 1))):
                printSolutionMainarizumu(ANew)

            else:

                # go to the next cell
                yNew = (y + 1) % n
                xNew = x

                # check if already on the end of the row
                if (yNew == 0):
                    # move to next row

                    xNew += 1

                # recursive call
                solveMainarizumu(ANew, xNew, yNew, n, ci, cd)
```

Gambar 17. Implementasi penyelesaian Mainarizumu (solveMainarizumu) dalam bahasa pemrograman Python

Berikut adalah implementasi program utama.

```python
A = [[UNDEFINED for j in range n] for i in range n]

solveMainarizumu(A, 0, 0, n, constraintIneq, constraintDiff)
```

Gambar 18. Implementasi program utama dalam bahasa pemrograman Python

Pada program utama kita awalnya akan menginisialisasi matriks A dengan nilai *undefined* dalam kasus ini saya menggunakan -1. Pemanggilan pertama dari prosedur penyelesaian Mainarizumu adalah seperti pada gambar 18.

Berikut adalah contoh keluaran untuk permainan Mainarizumu untuk studi kasus menggunakan gambar 1.

```
                                                    )$ python3 main.py
4 3 2 1
2 4 1 3
1 2 3 4
3 1 4 2
```

Gambar 19. Contoh keluaran untuk permainan Mainarizumu pada gambar 1

V. LINK VIDEO YOUTUBE

https://youtu.be/5JK0iMWtH1M

VI. PENUTUP

Pertama - tama saya ingin mengucapkan syukur kepada Tuhan karena telah membantu dan menguatkan saya dalam menulis makalah ini. Saya juga ingin berterima kasih kepada para dosen IF2211 Strategi Algoritma di Institut Teknologi Bandung karena telah memberikan saya kesempatan untuk menuliskan makalah ini sebagai bentuk eksplorasi dari materi yang sudah diajarkan di kelas. Saya berterimakasih kepada Dr. Nur Ulfa Maulidevi, S.T., M.Sc. yang sudah memberikan bimbingan selama proses belajar di kelas. Saya juga akan berterima kasih kepada semua yang telah mendukung saya dalam menuliskan makalah ini. Akhir kata, saya ingin meminta maaf bila ada kata - kata yang kurang tepat dalam tulisan ini.

VII. REFERENSI

[1] http://indi.s58.xrea.com/minarism/

[2] https://www.janko.at/Raetsel/Mainarizumu/index.htm

[3] https://www.owlapps.net/owlapps_apps/articles?id=52079043&lang=en

[4] http://www.cs.ubc.ca/labs/beta/Courses/CPSC532D-05/Slides/ch1-slides.pdf

[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algorima-Brute-Force-(2021)-Bag1.pdf

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

[7] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algorima-backtracking-2021-Bagian1.pdf

[8] https://www.kontensekolah.com/2017/09/menghitung-determinan-matriks-ordo-4x4.html

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2021

Ttd

Nathaniel Jason 13519108