Penerapan Algoritma *Depth-First Search* dan *Exhaustive Search* untuk Menyelesaikan Teka-teki *Skyscraper*

Jonathan Yudi Gunawan - 13518084
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13518084@std.stei.itb.ac.id

***Abstract*—Pada makalah ini akan dibahas mengenai penerapan algoritma *depth-first search* dan *exhaustive search* untuk menyelesaikan teka-teki *Skyscraper*. Depth-first search adalah algoritma pencarian berdasarkan penelusuran graph sedangkan exhaustive search adalah algoritma pencarian berdasarkan *brute force*. Pada bagian pertama akan membahas mengenai algoritma yang digunakan, sedangkan pada bagian selanjutnya akan ditunjukkan implementasi penyelesaian teka-teki *Skyscraper*.**

Keywords—Depth-First Search, Skyscraper, Puzzle

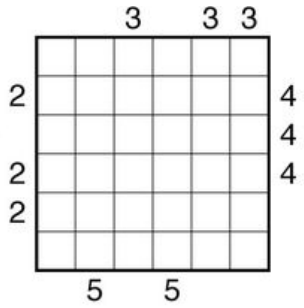
I. Pendahuluan

Ada banyak jenis puzzle di dunia, salah satu jenisnya adalah *pencil and paper* puzzle. Ada banyak jalur yang dapat ditempuh dalam menyelesaikan *pencil and paper* puzzle. Contohnya dalam menyelesaikan *sudoku*, dapat digunakan metode eliminasi angka, deduksi angka, metode mencoba lalu mencari kontradiksi, dll. Pada makalah ini akan dibahas salah satu metode untuk memecahkan *Skyscraper* puzzle.

II. Mengenal Skyscraper Puzzle

A. Komponen Skyscraper Puzzle

Skyscraper puzzle memiliki nuansa seperti sudoku. Skyscraper puzzle terdiri dari sebuah papan berukuran NxN (biasanya 4x4 hingga 7x7) yang harus diisi dengan angka mulai dari 1 hingga N. Sama seperti sudoku, tidak boleh ada angka yang berulang pada setiap baris dan kolom.



Gambar 1. Contoh skyscraper puzzle yang dibuat oleh Thomas Synder untuk Turkish Puzzle Championship

Masing-masing petak akan diisi oleh sebuah angka yang merepresentasikan tinggi sebuah bangunan pencakar langit. Semakin besar angkanya maka semakin tinggi bangunannya.

Perbedaannya dengan sudoku, terdapat aturan tambahan yaitu terdapat angka (selanjutnya akan disebut sebagai clue) di sekeliling papan berupa yang menunjukkan berapa banyak bangunan yang dapat dilihat jika seseorang melihat dari arah tersebut. Bangunan yang lebih tinggi akan menghalangi bangunan yang lebih rendah.

Gambar 2. Ilustrasi skyscraper puzzle menggunakan blok mainan, diambil dari https://buildingmathematicians.wordpress.com/2018/12/20/skyscraper-templates-for-relational-rods/

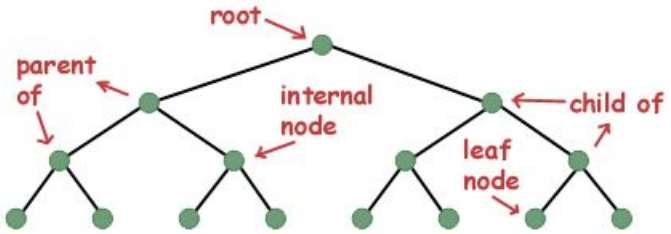
B. Sejarah Singkat

Skyscraper puzzle merupakah salah satu turunan dari jenis teka-teki Latin Square yang berasal dari Jepang. Skyscraper dikenal luas sejak puzzle ini digunakan pada lomba "*1st World Puzzle Championship*" di New York City pada tahun 1992. Namun sayangnya orang pertama yang mencetuskan ide Skyscraper tidak tercatat pada sejarah.

III. Dasar Teori

A. Komponen Graph

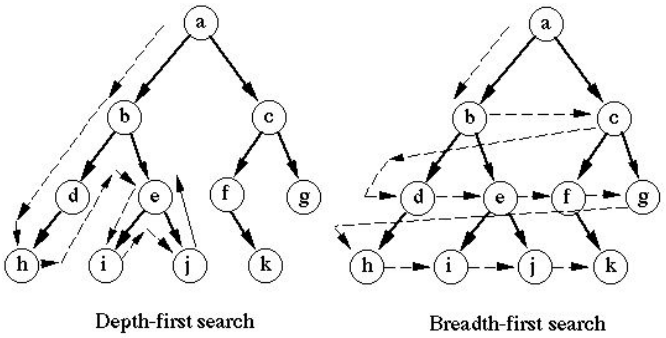
Sebuah graph terdiri dari kumpulan simpul (*node*) dan sisi (*edge / vertex*). Sebuah sisi menghubungkan dua buah simpul. Sisi bisa berarah dan bisa tak berarah. Simpul pertama disebut sebagai simpul akar (*root node*), sedangkan simpul pada level terbawah / simpul yang hanya memiliki satu sisi (selain root), biasa disebut simpul daun (*leaf node*). Pohon (*tree*) merupakan salah satu jenis graph yang tidak memiliki gelang (*loop*) di dalamnya.



Gambar 3. Ilustrasi graph dan komponennya, diambil dari http://bloggermcbloggerman.blogspot.com/2015/03/slog-week-9.html

B. Depth-First Search

Pencarian mendalam / *depth-first search (DFS)* merupakan salah satu algoritma penelusuran graph tanpa informasi prior (*uninformed search*) berdasarkan kedalaman. Berbeda dengan *breadth-first search (BFS),* ketika sudah dipilih satu simpul untuk ditelusuri, maka DFS akan melanjutkan penelusuran hingga ke ujung graph / level terdalam / simpul daun. Setelah mencapai simpul daun, penelusuran akan dilakukan kembali dari simpul yang berada pada satu level di atasnya, dilanjutkan dengan simpul anak



Gambar 4. Ilustrasi DFS dan BFS, diambil dari http://www.cse.unsw.edu.au/~billw/Justsearch.html

C. Algoritma Depth-First Search

DFS dimulai dari simpul v. Penelusuran dilakukan secara traversal. Berikut adalah algoritma DFS:

1. Kunjungi simpul v
2. Kunjungi simpul w yang bertetangga dengan simpul v.
3. Ulangi DFS mulai dari simpul w.
4. Ketika mencapai simpul sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

(Munir, Rinaldi (2020). *BFS dan DFS 2020*. Diktat Perkuliahan ITB.)

Pada implementasinya, biasa digunakan antrian last in first out (LIFO) / *stack* dalam menyimpan simpul yang akan dikunjungi selanjutnya, berbeda dengan BFS yang menggunakan antrian First In First Out (FIFO) / *queue*. Berikut adalah pseudocode untuk mengimplementasikan DFS:

DFS-iterative (G, s):

```
let S be stack
S.push(s)
mark s as visited.
while (S is not empty):
  v  =  S.top()
  S.pop()
  // Push all the neighbours of v in stack
that are not visited
  for all neighbours w of v in Graph G:
       if w is not visited :
       S.push( w )
       mark w as visited
```

DFS-recursive(G, s):

```
mark s as visited
for all neighbours w of s in Graph G:
  if w is not visited:
       DFS-recursive(G, w)
```

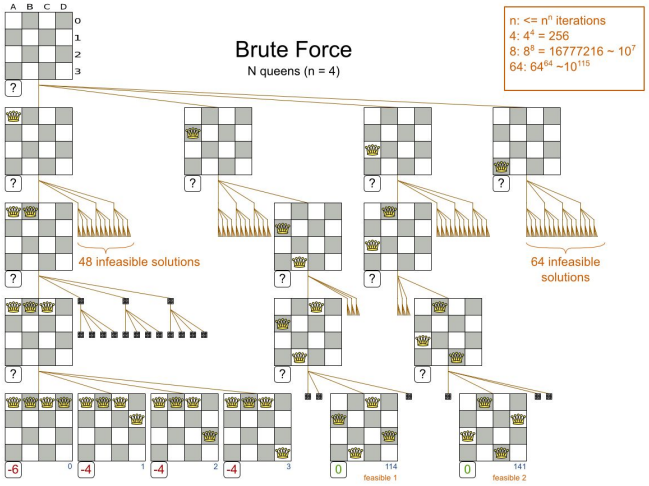
(Hackerrank. *Depth First Search Tutorial*. Hackerrank)

D. Brute Force

Brute force merupakan suatu metode pemecahan masalah secara *straightforward* / langsung. Biasanya algoritmanya sederhana dan tidak membutuhkan pengetahuan lebih secara khusus. Karakteristik brute force adalah algoritmanya tidak "cerdas" atau tidak efisien / mangkus. Algoritma brute force lebih cocok untuk persoalan yang kecil, yang tidak membutuhkan implementasi yang rumit.

D. Exhaustive Search

Exhaustive search adalah teknik pencarian solusi secara solusi brute force untuk persoalan-persoalan-masalah kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan.



Gambar 4. Ilustrasi Exhaustive Search pada permasalahan N-queen, diambil dari https://docs.jboss.org/drools/release/6.2.0.CR3/optaplanner-docs/html/exhaustiveSearch.html

E. Algoritma Exhaustive Search

Berikut adalah langkah-langkah untuk menerapkan metode exhaustive search:

1. Enumerasi(list) setiap solusi yang mungkin dengan cara yang sistematis.
2. Evaluasi setiap kemungkinan solusi satu per satu, simpan solusi terbaik yang ditemukan sampai sejauh ini (the best solution found so far).
3. Bila pencarian berakhir, umumkan solusi terbaik (the winner)

(Munir, Rinaldi (2016). Algoritma Brute Force 2016. Diktat Perkuliahan ITB.)

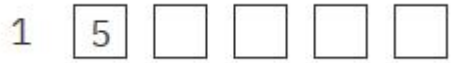
IV. Metode Penyelesaian Puzzle

A. Inferensi Aturan Skyscraper

Sebelum dilakukan algoritma-algoritma di atas, ada baiknya kita mencermati aturan permainan terlebih dahulu. Dari aturan permainan, dapat di-deduksi beberapa aturan sederhana untuk memulai memecahkan puzzle Skyscraper.

"*...terdapat clue di sekeliling papan berupa sebuah angka yang menunjukkan berapa banyak bangunan yang dapat dilihat jika seseorang melihat dari arah tersebut.*"

Bila clue merupakan angka 1, maka gedung pertama yang terlihat pasti merupakan gedung tertinggi (gedung dengan angka / ketinggian N), jika tidak, maka gedung tertinggi tersebut akan terlihat sebagai gedung kedua sehingga menyebabkan kontradiksi.



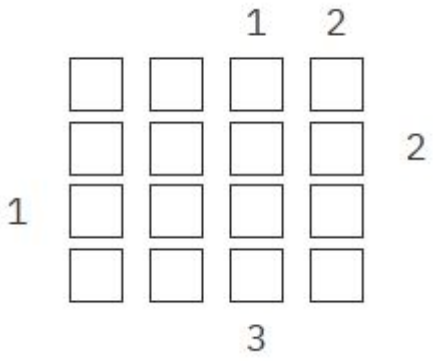
Gambar 5. Inferensi clue 1

Bila clue merupakan angka N, maka semua gedung terlihat dari arah tersebut, sehingga urutan gedung pasti urut dari 1 hingga N dari arah tersebut. Clue ini jarang ditemui karena dapat menyelesaikan satu baris/kolom secara langsung sehingga sangat memudahkan untuk menyelesaikan puzzle.

Gambar 6. Inferensi clue N

Bila clue bernilai antara 1 < clue < N, maka tidak dapat dilakukan inferensi secara langsung, sehingga diperlukan analisa lebih lanjut.

Untuk memperjelas, selanjutnya akan digunakan contoh papan berukuran 4x4 sebagai berikut:



Gambar 7. Contoh papan skyscraper berukuran 4x4

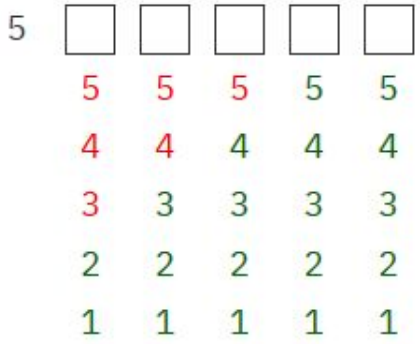
B. Restriksi dari Clue

Misalkan terdapat papan berukuran 5x5, maka dapat di deduksi bahwa:

- untuk clue 2, bangunan 5 tidak dapat diletakkan pada posisi pertama,
- untuk clue 3, bangunan 5 tidak dapat diletakkan pada posisi pertama dan kedua,
- untuk clue 4, bangunan 5 tidak dapat diletakkan pada posisi pertama, kedua, dan ketiga

Terlebih lagi, aturan ini juga berlaku untuk bangunan 4 dan seterusnya, yaitu:

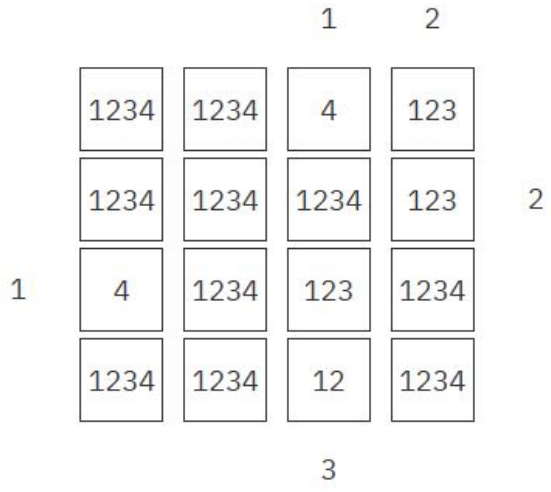
- untuk clue 2, bangunan 4 tidak dapat dilakukan deduksi apapun,
- untuk clue 3, bangunan 4 tidak dapat diletakkan pada posisi pertama,
- untuk clue 4, bangunan 4 tidak dapat diletakkan pada posisi pertama dan kedua



Gambar 8. Inferensi clue 1 < clue < N

Secara general, aturan ini dapat dinyatakan sebagai berikut:
"Pada papan `N * N`, untuk clue `c` di mana `1 < c < N`, di mana `d` adalah jarak gedung dari pinggir papan (dihitung dari nol), kita dapat merestriksi / meng-*exclude* semua nilai dari `N - c + 2 + d` hingga `N`, inklusif."
(Kurinsky, Jonathan. 2019. *Solving Every Skyscraper Puzzle: Part One*. Web)

Setelah dilakukan restriksi maka kemungkinan solusi pada papan akan menjadi lebih sedikit, sehingga mempercepat pencarian solusi. Berikut kondisi papan setelah dilakukan restriksi clue:



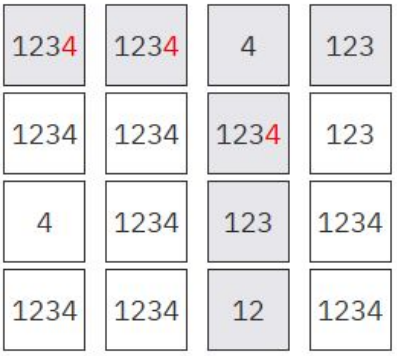
Gambar 9. Kondisi papan setelah dilakukan restriksi clue

C. Perambatan Restriksi

"*...tidak boleh ada angka yang berulang pada setiap baris dan kolom.*"

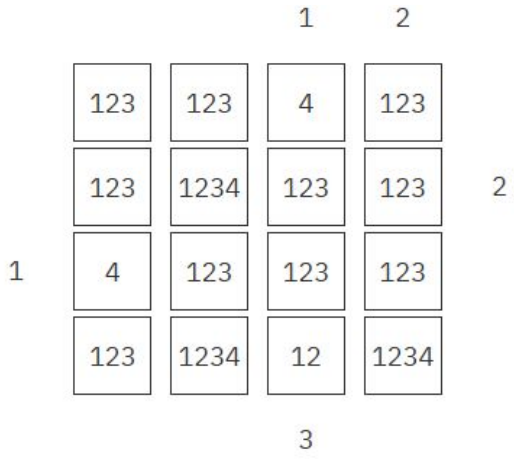
Pada gambar 9 terlihat bahwa ada angka berulang pada satu kolom / baris. Tahap ini akan mengeliminasi angka-angka tersebut.

Idenya adalah untuk masing-masing petak yang berhasil ditentukan nilainya / di-*resolve,* dilakukan ekspansi eliminasi angka tersebut ke keempat sisinya.



Gambar 10. Ekspansi eliminasi angka 4 dari petak (1,3)

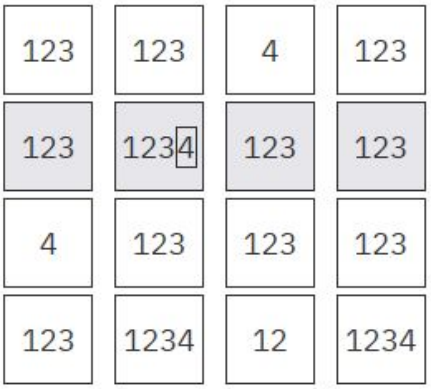
Berikut adalah kondisi papan setelah dilakukan perambatan restriksi:



Gambar 11. Kondisi papan setelah dilakukan perambatan restriksi

D. Resolve Angka Unik

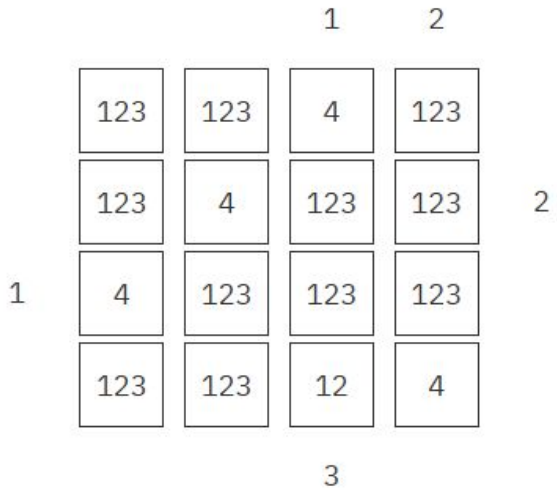
Perhatikan baris kedua dari atas pada gambar 11. Angka 4 hanya terdapat pada kolom kedua dari kiri, sehingga dapat dilakukan *resolve* pada petak tersebut menjadi angka 4. Hal ini diulangi pada setiap baris dan kolom pada papan.



Gambar 12. Angka 4 pada baris kedua hanya mungkin diletakkan pada kolom kedua

Setelah dilakukan resolve pada petak (2,2), dapat dilihat bahwa petak (4,2) seharusnya tidak mungkin berisi angka 4. Sehingga perlu dilakukan kembali perambatan restriksi. Secara general, setiap dilakukan resolve angka / pencoretan angka perlu dilakukan kembali tahap perambatan restriksi.

Setelah dilakukan keempat tahapan di atas, maka kondisi papan menjadi:



Gambar 13. Kondisi papan setelah dilakukan keempat tahapan pertama hingga tidak terjadi perubahan pada papan

E. Pencarian Solusi

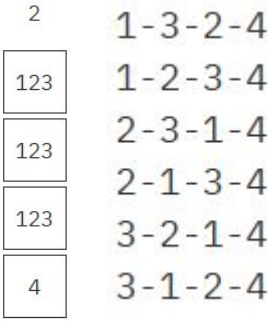
Pada setiap petak di papan, ada himpunan solusi yang mungkin bagi petak tersebut. Salah satu angka dari himpunan solusi tersebut merupakan solusi dari puzzle ini. Karakteristik puzzle ini cocok untuk diterapkan algoritma exhaustive search.

Namun pencarian tidak langsung diterapkan pada satu papan, namun per baris / kolom, agar program dapat berjalan dengan lebih cepat.

Masing-masing solusi kemudian dicek kebenarannya terhadap clue yang bersisian / bersangkutan. Bila cocok jumlah bangunan yang terlihat dengan clue yang ada, maka dapat disimpulkan solusi sudah ditemukan.

Namun bisa saja solusi tidak tunggal, maka harus disimpan himpunan solusi yang mungkin tersebut, untuk selanjutnya dapat dicek lagi dengan clue yang lain.

Misalkan untuk kolom paling kanan, terdapat 6 kemungkinan solusi, yaitu:

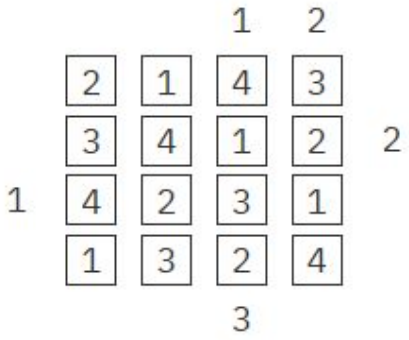


Gambar 14. Kolom 4 pada papan (kiri) - 6 kemungkinan solusi (kanan)

Namun hanya 2 dari 6 solusi yang memenuhi / *satistisfies* clue yaitu (3-2-1-4) dan (3-1-2-4). Dari kemungkinan solusi ini dapat disimpulkan bahwa petak pertama berisi angka 3, petak kedua dan ketiga berisi angka 1 atau 2), dan petak keempat berisi angka 4.

Secara general, untuk masing-masing solusi yang mungkin, dicari untuk setiap petaknya himpunan angka terkecil yang mungkin menjadi solusi, bila jumlahnya hanya 1 angka, maka dapat dilakukan resolve petak tersebut.

Ketika dilakukan resolve, maka dilakukan pula kembali 4 tahap di awal. Bila solusi papan belum ditemukan, lakukan kembali pencarian solusi untuk setiap baris dan kolom yang belum di-*resolve* secara utuh.



Gambar 15. Kondisi papan contoh setelah 3x iterasi

V. Implementasi Solver pada Bahasa Python

A. Persiapan Struktur Data

```
class Board:
    def __init__(self, size):
        self.board = [{i + 1 for i in range(size)} for j in range(size * size)]
        self.size = size
```

```
class Skyscraper:
    def __init__(self, size: int, clues=None):
        self.board = Board(size)
        self.clues = clues if clues else [0 for i in range(size * size)]
        self.size = size
```

Gambar 16. Kelas Skyscraper berisi sebuah Papan dan sebuah array yang menyimpan clue

Array yang digunakan untuk menyimpan clue adalah array 1 dimensi. Hal ini dilakukan untuk meningkatkan kecepatan program walaupun perlu dilakukan manipulasi untuk mendapatkan indeks yang tepat.

B. Inferensi dari Clue

```
for i in cell_indices:
    if clue == 1 or clue == self.size:
        self.resolve(i, last)
    else:
        self.exclude(i, exclude)
        exclude.add(last)
    last -= 1
```

Gambar 17. Implementasi inferensi dari clue

Clue 1 atau N dapat langsung me-resolve baris / kolom, sedangkan sisanya perlu dilakukan perhitungan seperti yang sudah dibahas di atas.

```
def resolve(self, i, last: int):
    self.board[i] = {last}
    self.propagate_constraint(i)
```

```
def exclude(self, idx, exclude):
    for num in iter(exclude):
        if num in self.board[idx]:
            self.board[idx] -= {num}
            self.eliminate(self.get_row_from_cell_index(idx), num)
            self.eliminate(self.get_col_from_cell_index(idx), num)
```

Gambar 18. Implementasi resolve dan exclude / pencoretan angka

C. Perambatan Restriksi

```
def propagate_constraint(self, idx) -> None:
    num = self.board[idx]
    for direction in Direction:
        for i in self.get_indices(idx, direction):
            if i != idx:
                self.exclude(i, num)
```

Gambar 19. Implementasi perambatan restriksi

Untuk menerapkan aturan ini, dilakukan penelusuran papan secara DFS dengan simpul akar adalah petak pertama di sebelah clue (petak dengan jarak 0 dari clue) dengan arah sesuai arah clue.

D. Resolve Angka Unik

```
def try_resolve_board(self):
    for i in range(self.size):
        self.try_resolve_row(i)
        self.try_resolve_col(i)

def try_resolve_row(self, row: int):
    clue_idx = row + self.size
    self.resolve_helper(clue_idx, self.get_indices_from_clue_all(clue_idx))

def try_resolve_col(self, col: int):
    self.resolve_helper(col, self.get_indices_from_clue_all(col))
```

Gambar 20. Implementasi resolve angka unik

Digunakan fungsi bantuan resolve helper untuk membuat program lebih modular. Resolve helper berfungsi untuk mengecek semua kemungkinan apakah cocok dengan clue pinggirnya.

```
def resolve_helper(self, clue_idx, sequences):
    clue1 = self.clues[clue_idx]
    clue2 = self.clues[self.get_opposite_clue_index(clue_idx)]
    possibles = [sequence for sequence in self.get_all_possible_sequences(sequences)
                 if len(set(sequence)) == 4 and self.check_clue(sequence, clue1)
                 and self.check_clue(reversed(sequence), clue2)]
    ans = [set() for x in range(self.size)]
    for possible in possibles:
        for i, x in enumerate(possible):
            ans[i].add(x)
    if all(ans):
        for i, idx in enumerate(self.get_indices_from_clue_all(clue_idx)):
            self.board[idx] = ans[i]
            if len(ans[i]) == 1:
                self.propagate_constraint(idx)
```

```
def get_all_possible_sequences(self, indices):
    return product(*map(lambda x: list(self.board[x]), indices))
```

Gambar 21. Implementasi resolve helper

Digunakan fungsi get all possible sequences untuk mendapatkan semua kemungkinan pada suatu baris/kolom.

E Fungsi Bantuan

```
def get_indices(self, start: int, direction: Direction) -> iter:
    if not self.valid_index(start):
        return range(-1)
    idx_range = {
        Direction.up: range(start, -1, -self.size),
        Direction.right: range(start, (start // self.size + 1) * self.size),
        Direction.down: range(start, self.size * self.size, self.size),
        Direction.left: range(start, start // self.size * self.size - 1, -1),
    }[direction]
    return iter(idx_range)
```

Gambar 22. Implementasi fungsi get_indices

Digunakan fungsi get indices untuk mempermudah algoritma DFS dalam mencari simpul / petak selanjutnya yang akan dievaluasi. Fungsi ini menerima simpul / petak awal dan arah penelusuran dan mengembalikan sequence indeks petak yang akan ditelusuri.

```
def get_row_from_cell_index(self, cell_idx):
    return self.get_iter_from_cell_index(cell_idx, [Direction.left, Direction.right])

def get_col_from_cell_index(self, cell_idx):
    return self.get_iter_from_cell_index(cell_idx, [Direction.up, Direction.down])
```

Gambar 23. Implementasi fungsi bantuan get row dan get col from cell index

Selain itu juga digunakan fungsi bantuan untuk mendapatkan baris dan kolom dari suatu petak (untuk mengeliminasi angka pada tahap perambatan restriksi).

Ada pula fungsi bantuan seperti is_valid_number dan board_is_solved yang cukup trivial.

VI. Implementasi Testing Solver

Berikut merupakan papan yang digunakan pada contoh sebelumnya (lihat gambar 7)

```
board1 = Skyscraper(4, clues=[0, 0, 1, 2, 0, 2, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0])
t1 = time.time()
board1.solve()
print((time.time() - t1) / 1000, "ms")
print(board1)
```

Gambar 24. Contoh Main Program untuk testing solver

```
{1, 2, 3}{1, 2, 3}{4}{1, 2, 3}
{1, 2, 3}{4}{1, 2, 3}{1, 2, 3}
{4}{1, 2, 3}{1, 2, 3}{1, 2, 3}
{1, 2, 3}{1, 2, 3}{1, 2}{4}

{1, 2}{1, 2}{4}{3}
{1, 2}{4}{3}{1, 2}
{4}{3}{1}{1, 2}
{3}{1, 2}{1, 2}{4}

{1, 2}{2}{4}{3}
{2}{4}{3}{1}
{4}{3}{1}{2}
{3}{1}{2}{4}

1.2404918670654297e-05 ms
{1}{2}{4}{3}
{2}{4}{3}{1}
{4}{3}{1}{2}
{3}{1}{2}{4}
```

Gambar 25. Contoh output main program

Pada output ditampilkan kondisi papan masing-masing iterasi, setelah 4 iterasi dapat ditemukan hasil akhir / solusi papan.

VII. Pranala Video di Youtube

Saya juga menyediakan penjelasan berupa video yang dapat diakses melalui pranala https://youtu.be/pxm2NKQdh0Y

VIII. Ucapan Terima Kasih

Saya ingin mengucapkan terima kasih kepada Tuhan Yang Maha Esa karena rahmat-Nya sehingga makalah ini dapat diselesaikan tepat waktu. Saya juga ingin mengucapkan terima kasih kepada para dosen mata kuliah IF2211 Strategi Algoritma terutama Pak Rinaldi Munir selaku dosen kelas K03 yang telah membimbing saya dan teman-teman sehingga kami memiliki pengetahuan yang cukup untuk dapat menyelesaikan makalah ini.

IX. Referensi

[1] Munir, Rinaldi. "BFS-dan-DFS(2020)." Program Studi Informatika, 2020. Web, http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/BFS-dan-DFS-(2020).pdf. Accessed on 1 May. 2020.

[2] Munir, Rinaldi. "Algoritma Brute Force." Program Studi Informatika, 2020. Web, http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/stima19-20.htm. Accessed on 1 May. 2020.

[3] "Lecture 14: Depth-First Search (DFS), Topological Sort," MIT OpenCourseWare, MIT. Web, https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-14-depth-first-search-dfs-topological-sort/. Accessed on 2 May. 2020.

[4] "Depth First Search" , Hacker Rank, Web. https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/. Accessed on 3 May. 2020.

[5] "Solving Every Skyscraper Puzzle: Part One," K. Jonathan, Web. https://www.krnsk0.dev/writing/skyscraper-puzzle-1. Accessed on 30 April. 2020

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, 3 Mei 2020

Jonathan Yudi Gunawan
13518084