Penyelesaian *Game Peg Solitaire* Menggunakan Algoritma *Backtracking* dan *Dynamic Programming*

Frederic Ronaldi / 13519134
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): frederic.ronaldi@gmail.com

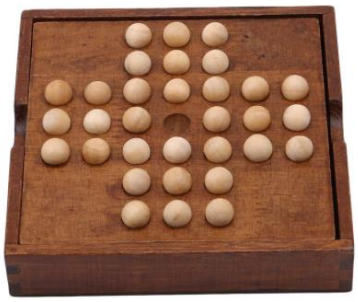
***Abstrak*—Peg Solitaire adalah sebuah board game untuk satu pemain yang mengasah otak. Permainan ini melibatkan peg/pasak yang diisi pada papan berlubang. Pemain dapat melompati satu pasak di atas pasak lainnya ke tempat kosong, lalu hilangkan pasak yang dilompati dari papan. Tujuannya adalah, terus membuat gerakan yang valid, untuk mengosongkan seluruh papan menyisakan satu pasak. Dengan menggunakan algoritma backtracking, kita dapat mencari solusi dari game peg solitaire dengan menelusuri semua kemungkinan solusi namun solusi yang sudah tidak memenuhi syarat akan dipangkas dan solusi selanjutnya tidak akan dicek. Solusi yang dihasilkan adalah step-step yang harus dilakukan oleh pemain agar dapat mengosongkan seluruh papan kecuali satu pasak. Optimisasi dengan dynamic programming dapat dilakukan juga untuk memangkas waktu yang diperlukan untuk mencari solusi.**

Kata Kunci—Algoritma Backtracking, Dynamic Programming, Optimisasi, Peg Solitaire

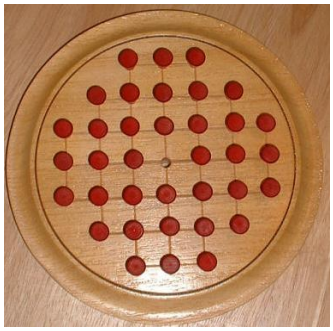
I. PENDAHULUAN

Peg Solitaire adalah sebuah game puzzle board game yang memiliki berbagai macam sebutan yang berbeda untuk setiap negara. Di Inggris, permainan ini disebut dengan Solitaire, sedangkan permainan Solitaire yang melibatkan kartu disebut dengan Patience. Permainan ini disebut Solitaire karena dipengaruhi oleh bahasa Latin, yaitu solus yang berarti sendiri, hal ini cocok dengan permainan ini karena hanya melibatkan seorang pemain. Di berbagai negara lain juga, permainan ini disebut Jumper, bahkan di India, permainan ini disebut sebagai Brainvitta.

Peg Solitaire melibatkan satu orang pemain dan satu buah papan dengan berbagai bentuk yang memiliki banyak lubang yang disusun sesuai dengan bentuk dan jenis papan. Setiap lubang dalam papan permainan akan diisi oleh pasak kecuali satu lubang yang akan menjadi starting point dari pemainan. Dalam permainan ini, pemain diharapkan dapat menghilangkan seluruh pasak yang dipasang, menyisakan satu pasak terakhir. Pemain dapat menghilangkan sebuah pasak setelah pasak tersebut dilewati/dilompati oleh pasak lain. Pasak yang ingin melompat, dapat melompat melewati satu pasak yang berada diantara pasak yang ingin melompat dan lubang, sehingga pasak yang dilompati akan dicabut, dan menjadi lubang agar pasak lain dapat melompat ke lubang tersebut.



Gambar 1.1 Peg Solitaire dari Inggris
Sumber: https://id.aliexpress.com/item/32916380136.html
(diakses pada 7 Mei 2021, 19:00)



Gambar 1.1 Peg Solitaire dari Eropa
Sumber: https://id.aliexpress.com/item/32916380136.html
(diakses pada 7 Mei 2021, 19:00)

Permainan ini dapat memiliki banyak solusi. Solusi yang dimaksud adalah step-step yang harus diambil oleh pemain untuk dapat menghilangkan semua pasak dan menyisakan satu pasak terakhir. Hal ini menyebabkan state papan dapat direpresentasikan sebagai simpul dalam graf dengan sisinya adalah step yang diambil oleh pemain, lalu dicari akar yang merupakan solusi dari permainan.

Pada makalah ini, penulis akan menggunakan algoritma backtracking untuk mencari solusi dari permainan Peg Solitaire dari Inggris. Graf akan ditelusuri, lalu memangkas graf yang tidak menuju pada solusi. Namun, apabila lubang, dan pasak semakin banyak, maka graf yang dibentuk dan harus ditelusuri akan semakin luas, sehingga penggunaan algoritma

backtracking tetap membutuhkan waktu yang lama untuk mencari solusi, sehingga algoritma pencarian solusi akan kembali dioptimisasi oleh dynamic programming.

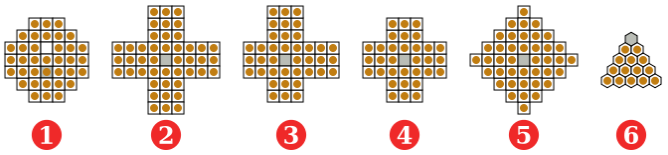
II. Teori Dasar

Pada makalah ini, ada beberapa teori-teori dasar yang diperlukan dalam pencarian solusi dalam permain Peg Solitaire yaitu permainan Peg Solitaire itu sendiri, algoritma backtracking dan dynamic programming.

1. Peg Solitaire

Peg Solitaire adalah sebuah permainan puzzle yang cukup popular dalam Inggris. Permainan ini melibatkan satu orang pemain dan satu buah papan yang memiliki banyak lubang yang diisi oleh pasak, kecuali satu buah lubang yang tidak memiliki pasak sebagai salah satu peraturan pertama agar permainan dapat dimulai. Lubang awal yang ada dalam papan permainan Peg Solitare biasanya berada di titik tengah papan, namun tidak menutup kemungkinan bahwa lubang awal dapat dipindah agar solusi yang dihasilkan berbeda dan lebih menantang pemain untuk mencari solusi.

Bentuk papan yang digunakan dalam permainan Peg Solitaire bervariasi, mulai dari pattern lubang yang dimiliki hingga jumlah lubang dan pasak yang ada dalam papan tersebut. Pattern lubang sendiri sudah memiliki banyak variasi, mulai dari pattern menyilang, belah ketupat, bahkan segitiga, lalu ada yang simetris dan juga asimetris. Setiap papan tersebut memiliki solusi yang berbeda dan peraturan pergerakan yang berbeda.

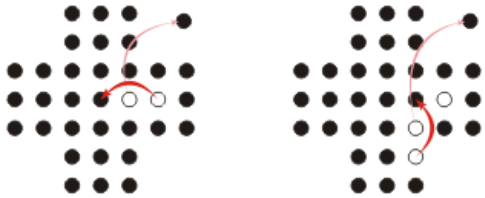


Gambar 2.1.1 Berbagai jenis variasi papan Peg Solitaire
Sumber: https://en.wikipedia.org/wiki/Peg_solitaire
(diakses pada 8 Mei 2021, 09:00)

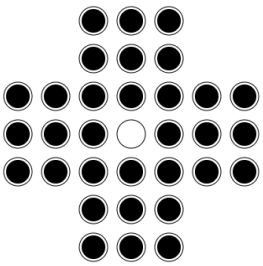
Terlihat pada gambar 2.1.1, ada berbagai macam bentuk papan Peg Solitaire. Berikut adalah penjelasan yang lebih lengkap terkait gambar 2.1.1 tersebut:

1. Tipe Eropa, berbentuk belah ketupat/wajik dan memiliki 37 lubang.
2. Tipe Jerman, berbentuk menyilang dan memiliki 45 lubang.
3. Tipe 3-3-2-2 asimetris, berbentuk menyilang yang tidak simetris dan memiliki 39 lubang.
4. Tipe Inggris (standar), berbentuk menyilang dan memiliki 33 lubang.
5. Tipe Wajik, berbentuk belah ketupat/wajik dan memiliki 41 lubang, lebih banyak daripada tipe Eropa.
6. Tipe Segitiga, berbentuk segitiga dan memiliki 15 lubang. Tipe ini memiliki pergerakan yang berbeda dari tipe-tipe lainnya karena hanya memiliki 3 arah untuk setiap lubang.

Pemain dapat menghilangkan pasak dengan cara melompati pasak yang akan dihilangkan dengan pasak lain. Pasak hanya dapat melompat ke lubang yang kosong yang diantara lubang kosong tersebut dengan pasak yang akan meloncat harus ada pasak yang akan dihilangkan. Berikut adalah gambar ilustrasi yang mendukung bagaimana langkah valid yang dapat dilakukan oleh pemain:

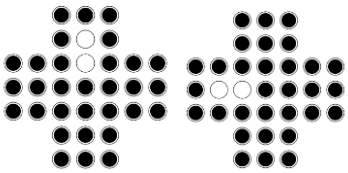


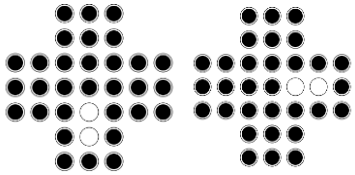
Gambar 2.1.2 Dua pergerakan, satu pergerakan loncat ke kiri, satu pergerakan loncat ke atas dalam papan Inggris



Gambar 2.1.3 State awal pada permainan Peg Solitare dari Inggris

Gerakan yang valid dan dapat dilakukan pemain adalah gerakan yang berarah ke 4 arah mata angin (kiri, kanan, atas, bawah), sehingga tidak dapat dilakukan pergerakan secara diagonal, kecuali dalam papan tipe segitiga. Dalam permainan awal pada papan Inggris, lubang yang berada ditengah dikosongkan (dapat dilihat dalam gambar 2.1.3) dan pemain diharuskan untuk mengosongkan lubang-lubang lain sehingga tersisa satu pasak dan mengisi lubang awal di tengah. Untuk papan Inggris, solusi yang diminta adalah membuat pasak terakhir berada di tengah papan, namun pada papan jenis lain hal tersebut tidak dimungkinkan dan biasanya dibebaskan asalkan ada satu pasak yang tersisa. Pada makalah ini, penulis akan membebaskan lokasi pasak terakhir, walaupun papan yang digunakan adalah papan Inggris. Sehingga pada jenis papan Inggris, ada empat langkah yang dapat dilakukan pada awal permainan, yaitu:





Gambar 2.1.4 Empat langkah awal dalam papan Inggris

Salah gerak satu langkah saja dapat membuat pemain menjadi tidak dapat mencapai solusi. Kesalahan ini umumnya membuat pasak menjadi jauh dan tidak terjangkau sehingga tidak dapat dihilangkan/diloncati. Hal ini membuat pemain diharuskan untuk memikirkan beberapa langkah kedepan sekaligus.

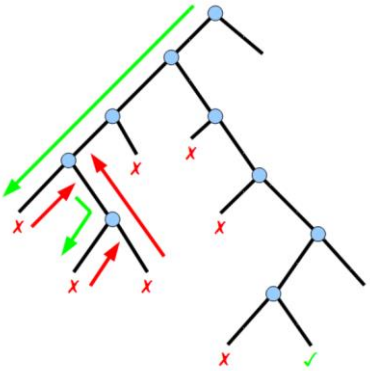
2. Algoritma Backtracking

Algoritma backtracking dapat dipandang sebagai sebuah fase dalam algoritma traversal DFS. Karena hal tersebut, algoritma traversal DFS biasanya mengimplementasikan stack atau menggunakan rekursif agar dapat kembali ke simpul sebelumnya. Selain itu, backtracking juga dipandang dapat menyelesaikan permasalahan optimisasi maupun non-optimisasi secara mangkus, sistematis, dan terstruktur. Pada algoritma exhausted search, semua kemungkinan solusi akan dieksplorasi dan dievaluasi satu per satu, sehingga akan memakan waktu yang lama. Sedangkan pada algoritma backtracking, hanya pilihan yang mengarah ke solusi yang dieksplorasi, pilihan yang tidak mengarah ke solusi tidak dipertimbangkan lagi. Hal tersebut dapat diperoleh dengan memangkas simpul-simpul yang tidak mengarah ke solusi. Ada beberapa properti umum dalam algoritma backtracking, yaitu:

1. Solusi persoalan
   Solusi persoalan dapat dinyatakan dalam vektor n-tuple: $X = (x_1, x_2, \ldots, x_n)$, $x_i \in S_i$. Umumnya $S_1 = S_2 = \ldots = S_n$.
2. Fungsi pembangkit nilai x_k
   Fungsi pembangkit dapat dinyatakan sebagai peredikat T(), dimana T(x[1], x[2], …, x[k – 1]) akan membangkitkan nilai x_k (komponen vektor solusi).
3. Fungsi pembatas
   Fungsi pembatas dapat dinyatakan sebagai predikat $B(x_1, x_2, \ldots, x_k)$, dimana B akan bernilai true jika $(x_1, x_2, \ldots, x_k)$ mengarah ke solusi dan tidak melanggar constraint. Jika true, maka pembangkitan untuk x_{k+1} akan dilanjutkan, jika false, maka $(x_1, x_2, \ldots, x_k)$ akan dibuang.

Semua kemungkinan solusi dalam suatu persoalan disebut dengan ruang solusi. Ruang solusi tersebut dapat diorganisasikan ke dalam struktur pohon berakar. Lalu tiap simpul pohon akan merepresentasikan sebagai state persoalan, sedangkan sisinya akan dilabeli dengan nilai-nilai xi. Lalu, akan dicari lintasan dari akar ke daun, yang merupakan salah satu solusi yang mungkin. Seluruh lintasan dari akar ke daun akan membentuk ruang solusi. Pengorganisasian pohon ruang solusi diacu sebagai pohon ruang status / state space tree.

Ada beberapa prinsip pencarian solusi dengan algoritma backtracking. Solusi akan dicari dengan membangkitkan simpul-simpul status sehingga menghasilkan lintasan dari akar ke daun. Selain itu, aturan traversal algoritma DFS digunakan sebagai aturan pembangkitan simpul. Simpul yang sudah dibangkitkan dinamakan simpul hidup/live node. Simpul hidup yang sedang ditraversal untuk diperluas disebut simpul-E/expand node. Simpul yang sudah tidak memenuhi fungsi pembatas (menghasilkan false), maka simpul tersebut akan dibuang dan disebut simpul mati/dead node. Ketika sebuah simpul menjadi simpul mati, maka sebenarnya secara implisit kita telah memangkas simpul-simpul anaknya karena simpul-simpul anaknya tidak akan menjadi simpul ekspan. Jika pembentukan sebuah lintasan berakhir dengan matinya sebuah simpul, maka proses pencarian lintasan akan di-backtrack ke simpul diatasnya dan dilanjutkan dengan membangkitkan simpul anak lainnya yang belum ditraversal, simpul inilah yang akan menjadi simpul ekspan yang baru. Pencarian akan berhenti apabila kita telah sampai pada simpul tujuan/goal node.



Gambar 2.2.1 Ilustrasi Algoritma Backtracking
Sumber: https://www.w3.org/2011/Talks/01-14-steven-phenotype/ (diakses pada 8 Mei 2021, 09:30)

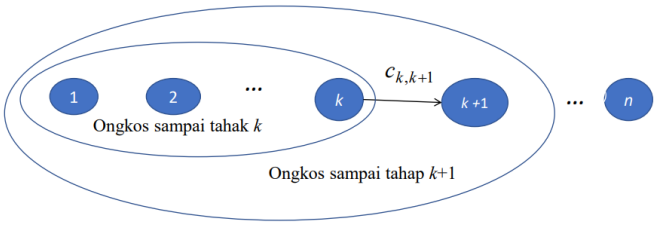
3. Dynamic Programming

Program dinamis atau dynamic programming adalah metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan tahapan, sedemikian sehingga solusi persoalan dapat dipandang sebagai serangkaian keputusan yang saling berkaitan. Kata program sendiri tidak ada kaitannya dengan pemrograman, dan istilah dinamis muncul karena pencarian solusi dihitung menggunakan tabel yang dapat berkembang seiring jalannya program. Dynamic programming digunakan untuk menyelesaikan persoalan-persoalan optimisasi, mau itu maksimasi atau minimisasi.

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan prinsip optimalitas. Prinsip optimalitas menyatakan bahwa jika solusi total sudah optimal, maka bagian solusi sampai tahap ke-k juga sudah optimal. Hal ini berarti bahwa jika kita dapat bekerja dari tahap k ke tahap k+1, kita dapat menggunakan hasil optimal dari tahap k tanpa harus kembali lagi ke tahap awal karena tahap k sudah disimpan sebelumnya. Sebagai contoh, hasil pada tahap k+1

adalah hasil pada tahap k ditambah dengan hasil dari tahap k ke tahap k+1.

Persoalan program dinamis dapat dibagi menjadi beberapa bagian, yang pada setiap tahapnya hanya akan diambil satu bagian. Masing-masing tahap ini terdiri dari sejumlah status yang saling berhubungan dengan tahap tersebut. Secara umum, status tersebut merepresentasikan bermacam kemungkinan masukan yang ada pada suatu tahap. Lalu, hasil dari keputusan yang diambil pada setiap tahap akan diubah dari status yang bersangkutan ke status berikutnya pada tahap berikutnya. Hasil dari suatu tahap biasanya meningkat secara teratur dengan bertambahnya jumlah tahapan dan bergantung pada hasil dari tahap-tahap sebelumnya yang telah berjalan. Tentu saja prinsip optimalitas berlaku pada persoalan ini.



Gambar 2.3.1 Ilustrasi Program Dinamis
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf
(diakses pada 8 Mei 2021, 10:30)

Ada dua pendekatan dalam program dinamis, yaitu program dinamis maju dan program dinamis mundur. Program dinamis maju adalah program yang melakukan perhitungan secara maju, dari tahap 1,2,3,…, hingga n. Sedangkan program dinamis mundur adalah program yang melakukan perhitungan secara mundur, dari tahap n, n-1, n-2, …, hingga 1.

Berikut adalah langkah-langkah yang harus dilakukan dalam menulis program dinamis:

1. Karakteristikkan struktur solusi optimalnya, setiap tahap yang dilalui, variabel keputusan, status, dsb.
2. Definisikan secara rekursif nilai solusi optimal (hubungan nilai optimal suatu tahap dengan tahap sebelumnya)
3. Hitung nilai solusi optimal secara maju atau mundur (simpan menggunakan tabel)
4. Rekonstruksi solusi optimal (opsional)

III. PEMBAHASAN

A. *Implementasi Permainan*

Pertama, jika kita mengingat susunan tata letak papan dari masing-masing jenis papan yang ada, kita dapat menggunakan papan sebesar 7x7 untuk dapat memuat papan jenis Inggris, Eropa dan juga segitiga. Ketiga jenis papan tersebut adalah jenis papan yang paling sering kali dipakai dalam permainan sebenarnya. Dengan menggunakan array dua dimensi/*dictionary*, kita dapat menyimpan isi dari state papan.

Untuk papan Inggris, kita dapat melihat bahwa 4 (2x2) kotak pada setiap ujung papan tidak dipakai/hilang. Untuk mempermudah kita, kita akan mendefinisikan beberapa konstanta global, yaitu panjang dan lebar papan (7), dan juga seberapa besar ujung yang tidak dipakai (2). Karena akan dibahas pencarian solusi untuk papan Inggris, maka akan konstanta tersebut akan diisi oleh panjang, lebar, dan ujung papan Inggris. Dalam papan ini, koordinat (X,Y) akan diberi tanda X apabila dalam lubang tersebut terdapat sebuah pasak, dan tanda O apabila dalam lubang tersebut kosong dan dapat menjadi tempat lompat sebuah pasak. Sehingga papan permainan akan terlihat seperti gambar berikut ini:

	0	1	2	3	4	5	6
0			X	X	X		
1			X	X	X		
2	X	X	X	X	X	X	X
3	X	X	X	O	X	X	X
4	X	X	X	X	X	X	X
5			X	X	X		
6			X	X	X		

Gambar 3.1 Ilustrasi State Awal Permainan Peg Solitaire papan Inggris

Pertama, state awal permainan peg solitaire papan Inggris akan diinisiasi agar dapat dicari solusinya. Tidak lupa, kita perlu membuat bagian-bagian ujung papan tidak dapat diakses pasak, sehingga lubang tersebut akan diisi dengan tanda spasi/kosong. Berikut adalah kode Python sederhana yang dapat digunakan untuk inisialisasi permainan:

```
PANJANG = 7
LEBAR = 7
UJUNG = 2

gameBoard = {}

def init(board):
    for y in range(LEBAR):
        for x in range(PANJANG):
            if ((x in range(0, UJUNG) or x in
range(PANJANG-UJUNG, PANJANG)) and (y in range(0, UJUNG) or
y in range(LEBAR-UJUNG, LEBAR))):
                board[x, y] = ' '
            elif x == int(PANJANG/2) and y == int(LEBAR/2):
                board[x, y] = 'O'
            else:
                board[x, y] = 'X'
    return board
```

```
gameBoard = init(gameBoard)
```

Untuk mempermudah kita untuk melihat permainan, kita perlu mengimplementasikan sebuah fungsi untuk menampilkan state papan yang sedang dimainkan, maka dari itu, akan dibuat fungsi printBoard untuk menampilkan state papan. Berikut adalah kode Python sederhana yang dapat digunakan untuk menampilkan state papan:

```
def printBoard(board):
    for y in range(LEBAR):
        for x in range(PANJANG):
            print (board[x,y], end = " ")
        print ('\n')
    print ('\n')
```

Selanjutnya, pengecekan solusi terhadap state papan yang sedang dimainkan pun perlu dilakukan. Hal ini dapat dicapai dengan mengecek apabila jumlah pasak yang ada di papan adalah satu. Hal ini dengan asumsi bahwa lokasi lubang awal tidak perlu menjadi lokasi pasak terakhir. Berikut adalah kode Python sederhana yang dapat digunakan untuk pengecekan solusi:

```
def checkResult(board):
    count = 0
    for i in board:
        if board[i] == 'X':
            count += 1
            if count > 1:
                return False
    return True
```

B. Implementasi Algoritma Backtracking untuk Menyelesaikan Permainan Peg Solitaire

Setelah menginisialisasi permainan agar dapat dicari solusinya, selanjutnya adalah mengimplementasikan algoritma backtracking untuk mencari solusi terhadap papan Inggris yang telah didefinisikan sebelumnya. Algoritma backtracking menggunakan traversal algoritma DFS, maka dari itu, berikut adalah pseudocode awal untuk mencari solusi permainan Peg Solitaire:

```
if solusi telah ditemukan
    return
else (solusi belum ditemukan)
    untuk setiap pergerakan yang valid
        ganti state papan menjadi state yang mengalami pergerakan tersebut dan panggil algoritma backtracking ini secara rekursif
                jika algoritma berhasil, simpan pergerakan tersebut dan kembalikan bahwa kita sudah menemukan solusi
                jika belum, state papan akan dikembalikan ke state papan sebelum melakukan pergerakan
Solusi tidak ditemukan
```

Kita perlu menyimpan list-list step yang sudah dilalui dan perubahannya terhadap state papan, sehingga akan dibuat sebuah variable pembantu yang menyimpan state-state papan untuk mencapai solusi tersebut. Setelah itu, pseudocode diatas akan diubah menjadi kode Python. Berikut adalah kode Python sederhana yang dapat digunakan untuk mencari solusi dari permainan Peg Solitaire:

```
def solve(board):
    if checkResult(board):
        print("=======SOLUTION========\n")
        for item in stepsBoard:
            printBoard(item)
        print("======================\n")
        return
    else:
        for item in board:
            # Cek apakah pasak ini dapat pindah ke kanan
            if item[0] < (PANJANG - 2) and board[item] == 'X' and board[item[0]+2, item[1]] == 'O' and board[item[0]+1, item[1]] == 'X':
                newBoard = copy.deepcopy(board)
                newBoard[item] = 'O'
                newBoard[item[0]+2, item[1]] = 'X'
                newBoard[item[0]+1, item[1]] = 'O'
                if newBoard not in stepsBoard:
                    stepsBoard.append(newBoard)
                    solve(newBoard)
                    stepsBoard.remove(newBoard)

            # Cek apakah pasak ini dapat pindah ke kiri
            if item[0] > (1) and board[item] == 'X' and board[item[0]-2, item[1]] == 'O' and board[item[0]-1, item[1]] == 'X':
                newBoard = copy.deepcopy(board)
                newBoard[item] = 'O'
                newBoard[item[0]-2, item[1]] = 'X'
                newBoard[item[0]-1, item[1]] = 'O'
                if newBoard not in stepsBoard:
                    stepsBoard.append(newBoard)
                    solve(newBoard)
                    stepsBoard.remove(newBoard)

            # Cek apakah pasak ini bisa pindah ke bawah
            if item[1] < (LEBAR - 2) and board[item] == 'X' and board[item[0], item[1]+2] == 'O' and board[item[0], item[1]+1] == 'X':
                newBoard = copy.deepcopy(board)
                newBoard[item] = 'O'
                newBoard[item[0], item[1]+2] = 'X'
                newBoard[item[0], item[1]+1] = 'O'
```

```
                if newBoard not in stepsBoard:
                    stepsBoard.append(newBoard)
                    solve(newBoard)
                    stepsBoard.remove(newBoard)


            # Cek apakah pasak ini bisa pindah ke atas
            if item[1] > (1) and board[item] == 'X' and
board[item[0], item[1] - 2] == 'O' and board[item[0],
item[1] - 1] == 'X':
                newBoard = copy.deepcopy(board)
                newBoard[item] = 'O'
                newBoard[item[0], item[1]-2] = 'X'
                newBoard[item[0], item[1]-1] = 'O'
                if newBoard not in stepsBoard:
                    stepsBoard.append(newBoard)
                    solve(newBoard)
                    stepsBoard.remove(newBoard)

        return
```

Untuk memudahkan pengecekan solusi, akan digunakan state papan awal yang lebih mudah, yaitu dengan cara menggunakan pasak yang lebih sedikit. Berikut adalah state papan awal yang digunakan untuk pengetesan adalah:

```
    0 0 0
    0 X 0
0 0 X X X 0 0
0 0 0 X 0 0 0
0 0 0 X 0 0 0
    0 0 0
    0 0 0
```

Setelah dijalankan, didapatkan bahwa ada dua solusi untuk menyelesaikan state papan tersebut. Berikut adalah salah satu solusi yang diberikan oleh program:

```
    0 0 0
    0 X 0
0 0 X 0 0 X 0
0 0 0 X 0 0 0
0 0 0 X 0 0 0
    0 0 0
    0 0 0
```

```
    0 0 0
    0 X 0
0 0 X X 0 X 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0
```

```
    0 0 0
    0 X 0
0 0 0 0 X X 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0
```

```
    0 0 0
    0 X 0
0 0 0 X 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0
```

```
    0 0 0
    0 0 0
0 0 0 0 0 0 0
0 0 0 X 0 0 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0
```

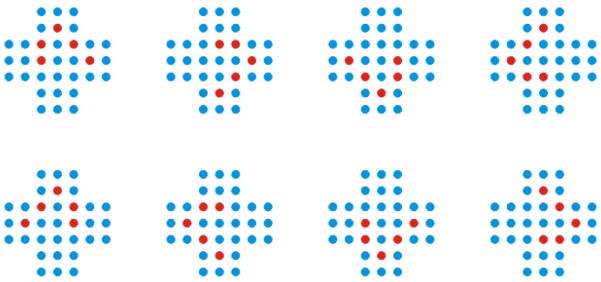
Dari hasil percobaan diatas, dapat diketahui bahwa algoritma berjalan dengan baik dan menghasilkan solusi. Semua pergerakan yang dilakukan valid dan menuju solusi yang diinginkan. Selanjutnya adalah mencari solusi dari persoalan utama yaitu papan Inggris. Maka dari itu ditemukanlah solusi-solusi yang memenuhi, namun solusi yang ditemukan sangatlah banyak sehingga menjalankan algoritma akan membutuhkan waktu yang lama. Karena solusi yang ada memiliki banyak langkah yang harus diambil, maka dari itu, solusi yang akan diberikan disini bukanlah state papannya melainkan langkah-langkah yang harus diambil pemain agar lebih menyederhanakan makalah ini. Berikut adalah langkah-langkah yang harus diambil pemain untuk mencapai salah satu solusi yang diberikan oleh program:

1. Gerak dari (3,5) ke (3,3),
2. Gerak dari (3,2) ke (3,4),
3. Gerak dari (3,0) ke (3,2),
4. Gerak dari (5,3) ke (3,3),
5. Gerak dari (3,3) ke (3,1),
6. Gerak dari (5,2) ke (3,2),
7. Gerak dari (4,0) ke (4,2),
8. Gerak dari (2,1) ke (4,1),
9. Gerak dari (2,3) ke (2,1),
10. Gerak dari (2,0) ke (2,2),
11. Gerak dari (2,5) ke (2,3),
12. Gerak dari (4,4) ke (2,4),
13. Gerak dari (2,3) ke (2,5),
14. Gerak dari (0,4) ke (2,4),
15. Gerak dari (0,2) ke (0,4),
16. Gerak dari (4,6) ke (4,4),
17. Gerak dari (2,6) ke (4,6),
18. Gerak dari (3,2) ke (5,2),
19. Gerak dari (1,2) ke (3,2),
20. Gerak dari (6,2) ke (4,2),
21. Gerak dari (3,2) ke (5,2),
22. Gerak dari (6,4) ke (6,2),
23. Gerak dari (6,2) ke (4,2),
24. Gerak dari (4,1) ke (4,3),
25. Gerak dari (4,3) ke (4,5),
26. Gerak dari (4,6) ke (4,4),
27. Gerak dari (5,4) ke (3,4),
28. Gerak dari (3,4) ke (1,4),
29. Gerak dari (0,4) ke (2,4),
30. Gerak dari (2,5) ke (2,3),
31. Gerak dari (1,3) ke (3,3)

Dari hasil percobaan diatas, dapat diketahui bahwa algoritma berjalan dengan baik dan menghasilkan salah satu solusi untuk papan Inggris.

C. Implementasi Dynamic Programing untuk Mengoptimisasi Algoritma Backtracking

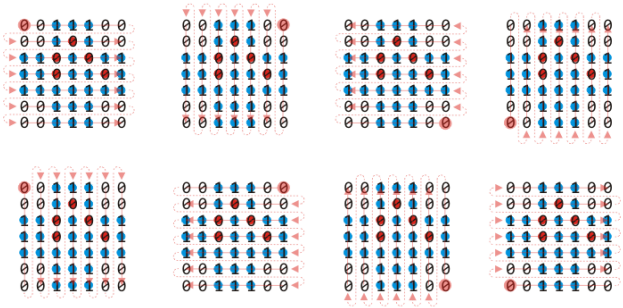
Setelah mengimplementasikan algoritma backtracking untuk permainan Peg Solitaire untuk papan Inggris. Terlihat untuk menghasilkan seluruh solusi dari permainan membutuhkan waktu yang sangat lama, dari berjam-jam bahkan bisa berhari-hari. Hal ini disebabkan karena banyaknya pilihan yang memungkinkan dalam satu langkah di satu state papan. Namun, jika diperhatikan lebih lanjut, diketahui bahwa sebenarnya kita masih bisa mengoptimisasi algoritma tersebut. Perhatikan gambar dibawah ini, gambar ini menunjukan bahwa state papan ini pada dasarnya mewakili posisi yang sama.



Gambar 3.2 Delapan state papan yang kongruen

Kedelapan papan tersebut sebenarnya hanyalah rotasi atau pembalikkan lalu dirotasi kembali dari satu sama lainnya. Kita dapat mengatakan bahwa papan-papan ini saling kongruen satu sama lain. Karena kita menggunakan algoritma backtracking, hal ini sudah menjadi risiko karena papan-papan yang kongruen tetap akan dicari solusinya berkali-kali, bahkan apabila salah satu papan yang kongruen sudah dibuktikan tidak akan mencapai solusi.

Karena hal ini, kita perlu sebuah cara untuk mengecek apabila sebuah papan kongruen dengan papan yang sudah dibuktikan tidak mencapai solusi atau tidak. Hal ini dapat diraih dengan cara mengimplementasikan bit board dan memanfaatkan dynamic programming untuk menyimpan state-state papan yang tidak menuju solusi. Mengingat bahwa papan yang kita gunakan adalah sebesar 7x7, maka satu state papan dapat direpresentasikan dengan 49 bit. Ini berarti state papan dapat disimpan dalam variabel bertipe long (64 bit) tanpa harus menyentuh dictionary yang besarannya akan jauh lebih besar daripada long. Kita dapat merepresentasikan lubang yang berisi pasak dengan 1 dan lubang serta ujung papan dapat diisi dengan angka 0. Karena itu, kita dapat dengan mudah merotasi integer long binary tersebut dan melakukan pengecekan apabila papan tersebut kongruen dengan papan yang sudah tidak menuju solusi atau tidak. Berikut adalah ilustrasi yang mendukung untuk lebih mudah memahaminya:



Gambar 3.3 Delapan state papan yang kongruen dan representasi binernya

Dari ilustrasi diatas, terlihat bahwa kedepalan papan tersebut memiliki bilangan biner yang dibalik akan saling menghasilkan hasil yang sama. Kedelapan papan tersebut akan mempunyai salah satu bilangan biner yaitu 001110000101001 10101111110111111110011100011100, atau jika dalam decimal adalah 123861450346012. Sehingga algoritma

backtracking yang sebelumnya dapat direvisi dan ditambahkan dynamic programming untuk mempercepat jalannya algoritma, berikut adalah pseudocodenya:

```
if solusi telah ditemukan

    return

if papan ini sudah dicek dan kongruen terhadap papan yang
sebelumnya sudah dicek dan tidak menuju solusi

    papan ini tidak menuju solusi juga

else (solusi belum ditemukan)

    untuk setiap pergerakan yang valid

        ganti state papan menjadi state yang mengalami
        pergerakan tersebut dan panggil algoritma
        backtracking ini secara rekursif

                jika algoritma berhasil, simpan pergerakan
                tersebut dan kembalikan bahwa kita sudah
                menemukan solusi

                jika belum, state papan akan dikembalikan
                ke state papan sebelum melakukan
                pergerakan
Solusi tidak ditemukan, tandai bahwa papan ini tidak menuju
solusi agar papan berikutnya tidak perlu dilakukan
pengecekan
```

IV. Kesimpulan

Permainan Peg Solitaire adalah salah satu permainan puzzle yang menantang dan hanya melibatkan satu pemain. Dengan menggunakan algoritma backtracking kita dapat mencari solusi-solusi terhadap permainan Peg Solitaire. Solusi terhadap permainan Peg Solitaire papan Inggris tergolong cukup banyak sehingga jumlah solusi tidak dapat ditentukan dalam waktu singkat dan membutuhkan waktu untuk mencari seluruh solusi yang ada. Solusi yang banyak tersebut disebabkan oleh banyaknya solusi yang kongruen dengan solusi yang lain, diantaranya adalah solusi yang hanyalah rotasi atau pembalikkan lalu dirotasi kembali untuk menjadi solusi lainnya. Sehingga, optimisasi terhadap algoritma backtracking tersebut dapat dilakukan dengan memberlakukan dynamic programming, yaitu dengan menyimpan setiap state papan dan kekongruenannya yang sudah dipastikan tidak menuju ke solusi.

V. Ucapan Terima Kasih

Pertama-tama penulis ingin mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas berkatNya, saya dapat menyelesaikan makalah ini dengan baik. Kemudian saya ingin berterima kasih kepada Prof. Ir. Dwi Hendratmo Widyantoro, M.Sc., Ph.D. selaku dosen mata kuliah IF2211 Strategi Algoritma yang telah mengajari saya materi ini dengan sangat baik. Terakhir saya ingin berterima kasih kepada keluarga serta teman-teman yang sudah menyemangati saya dan memberi banyak masukan dalam proses pembuatan makalah ini. Saya berharap dengan adanya makalah ini dapat menambah wawasan kita semua.

Video Link di Youtube

https://youtu.be/oTbuepYRR6U

Referensi

[1] Munir, Rinaldi, Slide Perkuliahan IF2211 Algoritma Runit-Balik (Bagian 1 dan 2) dan Program Dinamis (Bagian 1 dan 2), diakses pada tanggal 7 Mei 2021, 10.45.
[2] https://en.wikipedia.org/wiki/Peg_solitaire, diakses pada tanggal 8 Mei 2021, 09.00
[3] http://www.chessandpoker.com/peg-solitaire-solution.html, diakses pada tanggal 8 Mei 2021, 09.30
[4] https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/, diakses pada tanggal 10 Mei 2021, 11.00
[5] https://markusthill.github.io/solving-peg-solitaire/, diakses pada tanggal 10 Mei 2021, 10.00
[6] http://trevorappleton.blogspot.com/2015/09/solving-peg-solitaire-using-depth-first.html, diakses pada tanggal 10 Mei 2021, 10.00

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, 11 Mei 2021

Frederic Ronaldi 13519134