Penyelesaian *N-Puzzle* Menggunakan *A** dan *Iterative Deepening A**

Makalah IF2211 Strategi Algoritma

Marvin Jerremy Budiman (13515076)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515076@std.stei.itb.ac.id

***Abstrak*—Permainan *n-puzzle* merupakan permainan yang telah ada sejak abad ke-19. Pemain dituntut untuk memperbaiki susunan *puzzle* menjadi susunan yang benar. Di zaman sekarang, permainan *n-puzzle* ini serta permainan yang sejenis, seperti catur, sudoku, dan semacamnya, sudah dapat diselesaikan oleh komputer dengan menggunakan beberapa pilihan algoritma. Salah satu algoritma yang dapat digunakan untuk mencari solusi permainan ini adalah algoritma A*. Algoritma A* dapat dimodifikasi lagi menjadi algoritma *iterative-deepening A** yang lebih efisien dalam penggunaan memori, dibanding dengan algoritma A* biasa.**

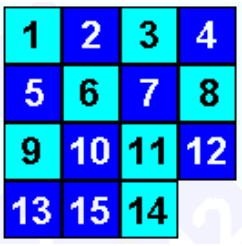
***Kata kunci*—algoritma, heuristik, pencarian, puzzle**

I. PENDAHULUAN

N-puzzle merupakan *puzzle* yang terdiri dari petak-petak yang dapat berupa angka-angka maupun potongan gambar dan sebuah petak kosong. Ide permainan ini adalah menggeser petak yang bersebelahan dengan petak kosong, sehingga posisi kedua petak bertukar, sampai susunan petak menjadi susunan yang benar. Untuk petak yang berupa angka, petak-petak tersebut harus terurut menaik dari kiri ke kanan dan dari atas ke bawah. Pada umumnya *n-puzzle* berukuran m petak × m petak, dan terdiri dari n = m^2 - 1 petak (tidak termasuk petak kosong). Terdapat berbagai versi untuk *n-puzzle* sesuai dengan jumlah petaknya, misalnya 8-*puzzle* (3 × 3), 15-*puzzle* (4 × 4), dst.

Versi awal dari *n-puzzle* ini merupakan 15-*puzzle* (4 × 4), dibuat sekitar tahun 1870-an oleh Noyes Chapman. Namun puzzle ini dikenal dunia karena Sam Loyd membuat variasi dari 15-*puzzle* ini, yang pada susunannya hanya petak nomor 14 dan 15 yang tertukar, sedangkan petak yang lain sudah dalam posisi yang benar. Beliau memberi puzzle tersebut nama *14-15 puzzle*. Sam Loyd pun menawarkan hadiah 1000 dollar bagi yang mampu menyelesaikan *puzzle* tersebut. Namun kenyataannya *puzzle* yang dibuat beliau mustahil untuk diselesaikan, karena tidak memenuhi syarat *solvability* dari sebuah *n-puzzle*.

Menyelesaikan *n-puzzle* bukanlah persoalan yang dapat diselesaikan dengan waktu yang singkat. Semakin ukuran *puzzle* bertambah, maka semakin besar tingkat kesulitannya. Zaman sekarang ini, *puzzle* ini dapat diselesaikan dengan bantuan komputer. Banyak algoritma dan teknik yang dapat diterapkan untuk menyelesaikan *puzzle* ini. Walau demikian, setiap algoritma dan teknik memiliki kelebihan dan kekurangan masing-masing. Salah satu algoritma yang dapat digunakan untuk menyelesaikan *sliding puzzle* ini adalah algoritma A* yang termasuk ke dalam kategori algoritma *informed search*.



Gambar 1. Susunan 14-15 puzzle buatan Sam Loyd.
Sumber: https://hc11.home.xs4all.nl/15puzzle/15puzzen.htm

II. DASAR TEORI

A. Algoritma Pencarian

Dalam sains komputer, masalah seperti mencari solusi *n-puzzle* dapat diselesaikan dengan metode *searching*. Metode *searching* tersebut membentuk pohon pencarian, yang terdiri dari simpul dan cabang. Pada permasalahan *n-puzzle* ini, simpul pada pohon melambangkan status (*state*) *puzzle*, yaitu susunan petak pada *puzzle*, sedangkan cabang melambangkan langkah-langkah apa saja yang dapat dilakukan dari suatu simpul (gerakan petak kosong ke kiri, kanan, atas, atau bawah). Setiap cabang memiliki bobot tertentu, yaitu ”ongkos” yang diperlukan untuk mencapai simpul anak dari simpul sebelumnya, dan pada persoalan *n-puzzle* ini “ongkos” tiap cabang besarnya sama, yaitu 1.

Proses pencarian solusi dilakukan dengan menentukan simpul akar, yaitu susunan *puzzle* yang belum benar dan ingin dicari solusinya. Kemudian simpul akar tersebut diproses, yaitu dicek apakah sudah sama dengan *goal state* yang diinginkan. Jika belum sama maka dari simpul akar tersebut dibentuk simpul-simpul anak berdasarkan langkah yang dapat dilakukan dari simpul akar. Lalu satu per satu simpul yang telah terbentuk diproses, dan simpul-simpul baru terus dibangkitkan sampai didapati simpul yang sama dengan *goal state*. Namun pencarian

bukan hanya bertujuan untuk menemukan solusi menuju *goal state*, tapi juga untuk menemukan solusi paling optimal untuk mencapai *goal state*; pada permasalahan *n-puzzle* ini, solusi yang diinginkan adalah solusi dengan langkah minimal.

Kualitas algoritma pencarian dapat diukur dari beberapa aspek, yaitu aspek *completeness*, aspek optimalitas, kompleksitas waktu, dan kompleksitas ruang. Aspek *completeness* diukur dari apakah algoritma tersebut mampu menemukan solusi ketika solusi seharusnya didapati (jika ya, maka algoritma tersebut sifatnya *complete*). Aspek optimalitas diukur dari apakah solusi yang didapati merupakan solusi optimal (membutuhkan "ongkos" minimal, tapi "profit" maksimal). Kompleksitas waktu diukur dari waktu yang dibutuhkan untuk mencari solusi hingga ditemukan. Kompleksitas ruang diukur dari jumlah memori yang diperlukan selama proses pencarian dilakukan.

Algoritma pencarian dapat dikelompokkan menjadi 2 golongan, yaitu:

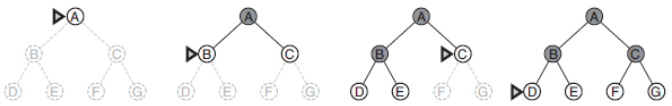
1) Uninformed Search

Dalam *uninformed search*, tidak ada informasi tambahan pada *state* setiap simpul selain bobot simpul ("ongkos" untuk mencapai simpul tersebut dari simpul akar). Karena tidak adanya informasi tersebut, maka tidak dapat ditentukan manakah simpul yang lebih menjanjikan akan menghasilkan simpul solusi jika diproses, sehingga umumnya proses pencarian dengan teknik ini kurang cepat.

Beberapa jenis algoritma yang termasuk *uninformed search* adalah:

a) Breadth-First Search (BFS)

Algoritma BFS memprioritaskan simpul dengan level paling dangkal. Misal, simpul-simpul pada level 1 diproses dan membangkitkan simpul-simpul level 2, kemudian semua simpul pada level 2 diproses dahulu dan menghasilkan simpul-simpul level 3, kemudian dilanjutkan dengan memproses semua simpul level 3, dan seterusnya. Karena itu pencarian menggunakan BFS menghasilkan pohon pencarian yang melebar. BFS dibantu dengan menggunakan struktur data *queue* yang sifatnya FIFO (*First In First Out*). Setiap simpul yang dibangkitkan, dimasukkan ke dalam *queue*, lalu simpul yang akan diproses diambil dari kepala *queue*. BFS bersifat *complete* dan optimal (jika bobot tiap cabang sama). Namun jumlah memori yang diperlukan untuk BFS jauh lebih besar dari waktu eksekusi yang diperlukan, sehingga komputer akan cepat kehabisan memori.



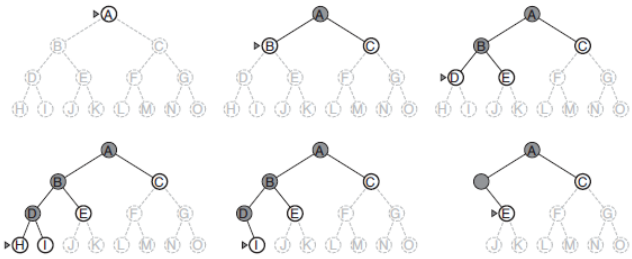
Gambar 2. Contoh penelusuran dengan BFS
Sumber: Artificial Intelligence, A Modern Approach 3rd Edition

b) Uniform-Cost Search (UCS)

Ketika hendak mencari solusi optimal dari persoalan dan bobot untuk tiap langkah sama, maka BFS dapat menyelesaikannya dengan optimal. UCS dapat digunakan untuk mencari solusi optimal dari persoalan yang bobot tiap langkahnya berbeda-beda. UCS dibedakan dengan BFS dari urutan pemrosesan simpul. Dari seluruh simpul yang telah dibangkitkan dan belum diproses, diambil satu simpul dengan bobot paling kecil, kemudian diproses. *Priority queue* digunakan sebagai pengganti *queue* biasa pada BFS. Pada *priority queue* tersebut, simpul-simpul diurutkan menaik berdasarkan bobot simpul. Karena simpul dengan bobot paling kecil diproses terlebih dulu, maka UCS dijamin optimal.

c) Depth-First Search (DFS)

Algoritma DFS memprioritaskan simpul dengan level paling dalam. Misal, pada level 2 terdapat 2 simpul, lalu salah satu dari simpul tersebut diproses dan menghasilkan 2 simpul level 3, maka yang akan diproses selanjutnya adalah 1 simpul pada level 3, bukan simpul pada level 2. Karena itu pencarian menggunakan DFS menghasilkan pohon pencarian yang mendalam. Dalam implementasi DFS dapat digunakan struktur data *stack* yang sifatnya LIFO (*Last In First Out*). Selain menggunakan *stack*, DFS dapat diimplementasikan menggunakan fungsi rekursif. DFS bersifat *complete* ketika setiap simpul yang sudah diproses ditandai untuk tidak diproses lagi. Ketika simpul yang sudah diproses tidak ditandai maka pada pembentukan pohon pencarian ada kemungkinan terjadinya *infinite loop*, yang menyebabkan pencarian tidak *complete*. DFS tidak optimal karena DFS memproses simpul dengan kedalaman paling besar, yang berarti bobot dari simpul akarnya juga besar. Penggunaan memori untuk DFS jauh lebih kecil dibandingkan BFS, namun waktu eksekusinya dapat menjadi sangat lama.



Gambar 3. Contoh penelusuran dengan DFS
Sumber: Artificial Intelligence, A Modern Approach 3rd Edition

d) Depth-Limited Search (DLS)

DLS dapat menjadi alternatif untuk DFS, untuk mencegah pembentukan pohon pencarian yang terlalu dalam. Ide DLS adalah dengan membatasi kedalaman pencarian hingga mencapai suatu nilai tertentu, yang biasa disebut *cutoff*. Konsekuensi dari pembatasan kedalaman ini adalah DLS menjadi tidak *complete* ketika kedalaman solusi lebih besar dari *cutoff* dan tidak optimal ketika kedalaman solusi lebih kecil dari *cutoff*.

e) Iterative-Deepening Search (IDS)

IDS merupakan pengembangan dari DLS. Pada IDS, DLS dilakukan secara iteratif dengan nilai *cutoff* yang bertambah setiap kali iterasi. Akibat *cutoff* kedalaman yang bertambah setiap kali iterasi, IDS bersifat *complete* dan menjamin solusi optimal, namun di setiap iterasi, IDS membangkitkan simpul-simpul kembali dari awal, sehingga terkesan boros. Namun demikian kompleksitas waktu dari IDS, secara asimptotik sama dengan BFS.

2) Informed Search

Golongan lain dari algoritma pencarian adalah *informed search*. Pada *informed search* terdapat informasi tambahan pada setiap simpul, yaitu fungsi evaluasi,(*f(n)*, n adalah simpul) yang merupakan estimasi "ongkos" dari simpul akar menuju *goal state* dengan melalui simpul n. Salah satu komponen yang dapat dimasukkan ke dalam *f(n)* adalah fungsi heuristik (*h(n)*), yaitu estimasi "ongkos" dari simpul n menuju *goal state*. Pada *informed search*, simpul-simpul yang sudah dibangkitkan, dimasukkan ke dalam *priority queue*, terurut menaik berdasarkan nilai *f(n)*, lalu simpul pada kepala *queue* diambil untuk diproses.

Beberapa algoritma yang termasuk *informed search* adalah:

a) Greedy Best-First Search

Greedy best-first search hanya menggunakan *h(n)* sebagai komponen *f(n)*, atau dapat dituliskan

$$f(n) = h(n)$$

Contoh penggunaan *greedy best-first search* adalah dalam persoalan penentuan rute perjalanan dari kota awal ke kota tujuan. Setiap kota menjadi suatu simpul, simpul akar merupakan kota awal, *goal state* merupakan kota tujuan. Fungsi heuristik (*h(n)*) yang akan dipakai adalah jarak garis lurus antara suatu kota ke kota tujuan. Simpul akar diproses dan membangkitkan simpul-simpul yang merupakan kota tetangga dari kota awal. Simpul-simpul yang dibangkitkan tersebut, dimasukkan ke dalam *queue*. Kota yang jarak garis lurusnya paling kecil akan selalu diproses terlebih dahulu. Algoritma pencarian ini memiliki kelemahan di *completeness* (algoritma ini tidak *complete*) dan optimalitasnya (algoritma ini tidak optimal).

b) A Search*

Pada *A* search*, komponen dari *f(n)* tidak hanya terdiri dari *h(n)*, tetapi juga disertai dengan *g(n)*, yaitu "ongkos" dari simpul akar ke simpul n. Jadi f(n) merupakan penjumlahan dari keduanya, atau dapat dituliskan

$$f(n) = g(n) + h(n)$$

Pada persoalan yang sama, yaitu penentuan rute perjalanan, *g(n)* yang akan dipakai adalah jarak dari kota awal ke kota n menurut peta. *A* search* dijamin optimal dan *complete* jika fungsi heuristik bersifat *admissible*, yang berarti nilai fungsi heuristik tidak pernah melebihi "ongkos" sesungguhnya dari simpul akar ke *goal state*. Masalah utama dari penggunaan *A* search* yang umumnya sama dengan algoritma pencarian lainnya adalah, kebutuhan memori yang besar untuk melakukan pencarian.

c) Iterative-Deepening A (IDA*)*

Solusi untuk memangkas penggunaan memori pada *A* search* adalah menggunakan IDA*. Cara kerja IDA* menggabungkan prinsip A* dengan IDS, hanya saja nilai *cutoff* yang dipakai bukanlah kedalaman pohon, melainkan nilai *f(n)*. Untuk setiap simpul yang nilai *f(n)* nya lebih kecil dari nilai *cutoff*, maka simpul tersebut diproses. Jika sudah tidak ada simpul yang memenuhi syarat tersebut, maka diambil nilai minimal *f(n)* dari simpul-simpul yang sudah dibangkitkan namun belum diproses, dan nilai tersebut dijadikan nilai *cutoff* yang baru. IDA* umumnya diimplementasikan dengan fungsi rekursif. Sama seperti A*, *completeness* dan optimalitas dari IDA* bergantung pada fungsi heuristik yang dipakai.

B. Teori N-Puzzle

1) Solvability dari N-Puzzle

Dari sekian banyak susunan untuk suatu *puzzle*, tidak semuanya dapat mencapai *goal state*. Dapat atau tidaknya suatu *puzzle* diselesaikan, bergantung pada jumlah *inversion* dari *puzzle* tersebut. *Inversion(i)* didefinisikan sebagai jumlah petak j yang nilainya lebih kecil dari i, namun berada setelah i (di sebelah kanan atau bawah dari i). Untuk setiap *n-puzzle* berukuran m × m, untuk m merupakan bilangan ganjil berlaku

$$\sum_{i=1}^{n} inversion(i) \ is \ even \Rightarrow solvable$$

Untuk m merupakan bilangan genap berlaku

$$(\sum_{i=1}^{n} inversion(i) + x) \ is \ odd \Rightarrow solvable$$

dimana x merupakan nomor baris dimana terdapat petak kosong (penomoran baris dimulai dari 0).

```
      1  3                    1  2  3  4
  4  2  5                    5  6     8
  7  8  6                    9 10  7 11
                            13 14 15 12
1 3 4 2 5 7 8 6
                          blank row = 1
 inversions = 4           inversions = 6
(3-2 4-2 7-6 8-6)         --------------
                                sum = 7
```

Gambar 4.
Di sebelah kiri merupakan *puzzle* dengan ukuran ganjil yang *solvable*.
Di sebelah kanan merupakan *puzzle* dengan ukuran genap yang *solvable*
Sumber: https://www.cs.princeton.edu/courses/archive/fall12/cos226/assignments/8puzzle.html

2) Fungsi Heuristik untuk N-Puzzle

Untuk menyelesaikan persoalan *n-puzzle* menggunakan *informed search*, contohnya dengan menggunakan A*, diperlukan fungsi heuristik tertentu. Terdapat beberapa pilihan fungsi heuristik untuk persoalan *n-puzzle* ini, diantaranya adalah:

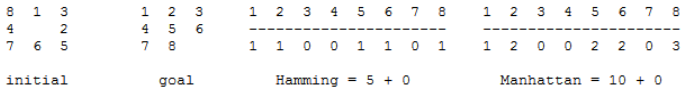
a) Fungsi jarak Manhattan

Untuk suatu simpul n yang merupakan status dari *puzzle*, nilai fungsi jarak Manhattan untuk simpul n tersebut, atau *manhattan(n)* merupakan jumlah jarak yang harus ditempuh oleh setiap petak secara horizontal dan vertikal untuk mencapai posisi seharusnya pada *goal state*. Pada perhitungan fungsi ini, petak kosong tidak dianggap.

b) Fungsi Hamming

Untuk suatu simpul n yang merupakan status dari *puzzle*, nilai fungsi Hamming untuk simpul n tersebut, atau *hamming(n)* merupakan jumlah petak yang tidak berada pada tempat yang seharusnya pada *goal state*. Pada perhitungan fungsi ini, petak kosong tidak dianggap.

Jika terdapat 2 buah fungsi heuristik h_1 dan h_2 dan untuk semua simpul n berlaku $h_2(n) > h_1(n)$, maka h_2 mendominasi h_1. Ketika h_2 lebih dominan dari h_1, maka proses pencarian yang menggunakan heuristik h_2 akan membangkitkan lebih sedikit simpul dibanding pencarian yang menggunakan heuristik h_1. Penggunaan fungsi heuristik yang lebih dominan, namun tetap *admissible* dapat meningkatkan efisiensi pencarian.



Gambar 5. Contoh perhitungan fungsi jarak Manhattan dan fungsi Hamming. Sumber: https://www.cs.princeton.edu/courses/archive/fall12/cos226/assignments/8puzzle.html

III. Implementasi & Perbandingan A* dan IDA*

A. Penyelesaian N-Puzzle dengan A*

Pendekatan pertama untuk menyelesaikan mencari solusi *n-puzzle* ini adalah dengan menggunakan *A* search*. Pada eksperimen ini *A* search* diimplementasikan dalam program Java.

1) Struktur Data

Struktur data utama yang digunakan adalah kelas Puzzle dan kelas PuzzleNode. Puzzle digunakan untuk merepresentasikan susunan/*state* dari *n-puzzle*, sedangkan PuzzleNode merepresentasikan simpul yang terdiri dari *state* (Puzzle) dan bobot dari simpul tersebut (*f(n)*). Untuk mencegah agar simpul yang sudah diproses tidak diproses lagi, maka pada PuzzleNode ditambahkan *state* sebelumnya dari suatu simpul.

2) Algoritma

Program diawali dengan menciptakan sebuah objek Puzzle dengan suatu ukuran tertentu. Program sudah dirancang agar penciptaan setiap objek Puzzle akan menghasilkan susunan yang acak, namun setiap kali susunan tersebut tidak *solvable*, maka akan dilakukan randomisasi ulang.

Objek Puzzle yang sudah diciptakan, kemudian akan diproses oleh objek QueueSolver. Di dalam objek QueueSolver ini terdapat *priority queue*, tempat memasukkan PuzzleNode yang akan diproses. Setiap kali penciptaan objek QueueSolver, akan dibuat PuzzleNode dari Puzzle awal, lalu PuzzleNode tersebut menjadi simpul pertama yang akan diproses. P*seudocode* dari fungsi getSolution() pada objek QueueSolver ditunjukkan pada Gambar 6.

```
function getSolution()
    while (current != goal state)
        neighbors = getNeighbors(current)
        for (node in neighbors)
            addToQueue(node)
        current = removeHeadFromQueue()
    return current
```

Gambar 6. *Pseudocode* untuk fungsi getSolution() pada objek QueueSolver

Fungsi heuristik yang digunakan dalam program ini adalah fungsi jarak Manhattan.

3) Hasil

Dalam 5 kali uji coba eksekusi program, hasil yang didapat untuk *puzzle* yang berukuran 3 × 3 adalah sebagai berikut:

No.	Bobot simpul awal	Solusi		
		Jumlah langkah	Waktu eksekusi (ms)	Jumlah simpul yang terbentuk
1	13	23	15	3167
2	11	19	22	680
3	18	20	4	113
4	14	24	16	3141
5	11	17	16	198

Tabel 1. Hasil eksekusi program yang menggunakan A*, pada *puzzle* dengan ukuran 3 × 3

Dalam 5 kali uji coba eksekusi program, hasil yang didapat untuk *puzzle* yang berukuran 4 × 4 adalah sebagai berikut:

No.	Bobot simpul awal	Solusi		
		Jumlah langkah	Waktu eksekusi (ms)	Jumlah simpul yang terbentuk
1	20	36	932	508820
2	20	32	163	94987
3	20	38	5671	2550942
4	19	29	85	38262
5	25	N/A	N/A	N/A

Tabel 2. Hasil eksekusi program yang menggunakan A*, pada *puzzle* dengan ukuran 4 × 4

Untuk ukuran *puzzle* 3 × 3, tidak ada masalah terjadi. Hal yang menarik adalah semakin dekat nilai bobot simpul awal (*h(n)* simpul awal) dengan bobot sesungguhnya (*g(n)* pada simpul solusi), maka jumlah simpul yang perlu dibangkitkan menjadi semakin sedikit.

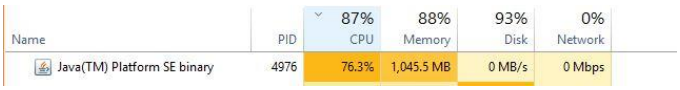
```
Pendekatan A*
Puzzle awal:
 9  2  3  4
 5 16  7  8
12  1 10 15
11  6 13 14
Bobot: 20

11919 simpul dibangkitkan
Solusi:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
Bobot: 30
30 langkah: 3 1 4 2 2 3 2 4 4 1 3 3 2 4 4 4 1 3 3 3 1 4 1 3 2 4 2 2 4 4
78 milisekon
```

Gambar 7. Contoh hasil eksekusi program dengan pendekatan A*

Masalah memori terjadi saat pengujian program dilakukan pada *puzzle* berukuran 4 × 4. Pada uji coba kelima terjadi *exception OutOfMemoryError*, karena banyaknya simpul yang dibangkitkan namun jarang digunakan. Tidak ada *progress* pada

program dan porsi CPU sebagian besar dialokasikan untuk *garbage collector* untuk mengelola objek-objek yang tidak terpakai tersebut. Karena itu JVM (*Java Virtual Machine*) melakukan *throw exception*. Pada pendekatan dengan menggunakan A*, eksekusi program dapat terhenti karena terlalu banyak simpul yang dibangkitkan. Banyaknya jumlah simpul yang dibangkitkan, menyebabkan ukuran *queue* dan jumlah memori yang digunakan semakin besar. Pada uji coba kelima, jumlah memori yang tercatat untuk eksekusi program mencapai 1045 MB dengan penggunaan CPU sebesar 89%.



Gambar 8. Penggunaan memori untuk program A*

B. *Meningkatkan Efisiensi Memori dengan IDA**

Masalah penggunaan memori memang menjadi masalah utama pada algoritma pembentukan pohon pencarian. Solusi dari masalah ini adalah dengan menggunakan pendekatan berikutnya, yaitu menggunakan IDA*, yang juga diimplementasikan dalam program Java. Strukur data untuk *state puzzle* dan simpul tetap sama seperti pada program A*.

```
function getSolution()
  bound = heuristic(current)
  result = null
  while (result == null)
    newbound = ∞
    result = search(current, bound)
    bound = newbound
  return result
function search(node, bound)
  neigbors = getNeighbors(current)
  for (node in neighbors)
    if (node != goal state)
      return node
    currentCost = heuristic(current) + g(current)
    if (currentCost <= bound)
      tempSolution = search(node, bound)
      if (tempSolution = null and (solution = null or
      g(tempSolution) < g(solution)))
        solution = tempSolution
    else
      if (currentCost < newbound)
        newbound = currentCost
  return solution
```

Gambar 9. *Pseudocode* untuk fungsi getSolution() pada objek RecursiveSolver

1) *Algoritma*

Sama seperti pada program A*, awalnya dibentuk objek Puzzle yang *solvable* dengan ukuran tertentu. Kemudian objek Puzzle tersebut diproses oleh objek RecursiveSolver. Tidak ada struktur data khusus yang dipakai pada RecursiveSolver. Fungsi getSolution() pada objek RecursiveSolver menggunakan pendekatan rekursif. *Pseudocode* untuk fungsi getSolution() tersebut ditunjukkan pada Gambar 9.

2) *Hasil*

Dalam 5 kali uji coba eksekusi program, hasil yang didapat untuk *puzzle* yang berukuran 3 × 3 adalah sebagai berikut:

No.	Bobot simpul awal	Solusi		
		Jumlah langkah	Waktu eksekusi (ms)	Jumlah simpul yang terbentuk
1	14	22	16	2047
2	7	17	17	791
3	14	24	16	4837
4	15	25	22	8003
5	16	24	18	3140

Tabel 3. Hasil eksekusi program yang menggunakan IDA*, pada *puzzle* dengan ukuran 3 × 3

Dalam 5 kali uji coba eksekusi program, hasil yang didapat untuk *puzzle* yang berukuran 4 × 4 adalah sebagai berikut:

No.	Bobot simpul awal	Solusi		
		Jumlah langkah	Waktu eksekusi (ms)	Jumlah simpul yang terbentuk
1	20	34	597	299529
2	19	43	66589	41251586
3	20	42	68975	40530407
4	20	42	22383	13184742
5	19	39	24412	14703250

Tabel 4. Hasil eksekusi program yang menggunakan IDA*, pada *puzzle* dengan ukuran 4 × 4

Dari hasil uji coba, didapati bahwa walaupun jumlah simpul yang dibangkitkan banyak, namun jumlah tersebut tidak terlalu berpengaruh pada jumlah memori yang terpakai untuk program. Jumlah simpul tercatat pada tabel merupakan total jumlah simpul yang dibangkitkan pada seluruh iterasi di IDA*, sedangkan pada setiap iterasi, simpul yang dibangkitkan pada iterasi sebelumnya tidak "diingat" lagi. Karena itu penggunaan memori pada setiap uji coba pada *puzzle* 4 × 4 tidak pernah melebihi 70 MB dan penggunaan CPU tidak pernah melebihi 40%. Walau demikian memang waktu eksekusi program menjadi lebih lama dibandingkan dengan waktu eksekusi

program A*, karena setiap kali iterasi pemrosesan simpul selalu dimulai dari simpul akar.

```
Pendekatan IDA*
Puzzle awal:
11  2  1  5
 6 16  3  4
 9 14  8  7
13 10 15 12
Bobot: 20

197563 simpul dibangkitkan
Solusi:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
Bobot: 34
34 langkah: 3 1 4 4 4 2 2 3 3 1 3 2 4 1 1 4 2 4 2 2 3 1 3 3 1 4 1 3 2 4 2 2 4 4
417 miliseconds elapsed
```

Gambar 10. Contoh hasil eksekusi program dengan pendekatan IDA*

IV. Simpulan

Dari hasil uji coba dari kedua implementasi tersebut, terdapat beberapa hal yang dapat disimpulkan:

1. Untuk kedua implementasi, semakin besar ukuran puzzle, maka waktu eksekusi menjadi semakin lama, dan jumlah simpul yang dibangkitkan menjadi semakin banyak.
2. Pada implementasi menggunakan A*, jumlah simpul yang dibangkitkan memang lebih sedikit dibanding dengan implementasi menggunakan IDA*, namun laju pertambahan penggunaan memori pada A* besar. Sedangkan pada IDA*, laju pertambahan penggunaan memori tidak besar, dikarenakan pada setiap iterasi, hanya simpul yang lebih kecil dari nilai cutoff yang diproses, dan simpul yang dibangkitkan pada iterasi sebelumnya tidak "diingat" lagi pada iterasi berikutnya.
3. Waktu eksekusi pada IDA* lebih lama dibandingkan A*, karena adanya iterasi pada IDA* dan setiap kali iterasi, pemrosesan selalu dimulai dari simpul akar. Karena itu jika terdapat batasan memori dan waktu pada eksekusi program, maka program yang menggunakan A* berpeluang terhenti karena melebihi batas memori, sedangkan program yang menggunakan IDA* berpeluang terhenti karena melebihi batas waktu. Namun karena keterbatasan memori merupakan masalah yang lebih sering ditemui (keterbatasan memori fisik pada komputer/gawai), maka algoritma IDA* lebih baik untuk digunakan

Dengan menggunakan algoritma IDA*, *puzzle* dengan ukuran 3 × 3 serta 4 × 4 dapat diselesaikan dalam waktu yang wajar dengan kebutuhan memori yang tidak besar. Walau begitu, untuk menyelesaikan *puzzle* dengan ukuran lebih dari 4 diperlukan optimasi yang lain. Dari hasil uji coba, didapati bahwa semakin dekat nilai fungsi heuristik pada simpul awal dengan bobot sesungguhnya untuk mencapai *goal state*, maka jumlah simpul yang dibangkitkan semakin kecil dan mengakibatkan waktu eksekusi pun berkurang. Karena itu jawaban untuk optimasi pada penyelesaian *n-puzzle* adalah dengan memperbaiki fungsi heuristik, sehingga nilai fungsi heuristik semakin dekat ke nilai bobot sesungguhnya.

Salah satu heuristik yang mungkin diterapkan adalah dengan menggunakan *pattern database heuristic*. *Database* ini dibuat dengan cara membentuk pohon pencarian menggunakan BFS, yang simpul akarnya merupakan *goal state*. Setiap simpul pada pohon pencarian ini memiliki bobot sama dengan jumlah gerakan yang dilakukan untuk mencapai simpul tersebut dari *goal state*. Nilai bobot simpul pada pohon pencarian ini yang akan menjadi nilai fungsi heuristik dalam A* maupun IDA*. Karena bobot simpul pada *database* merupakan jumlah langkah dari *goal state*, maka heuristik ini jauh lebih akurat dibandingkan dengan fungsi jarak Manhattan maupun fungsi Hamming.

Ucapan Terima Kasih

Saya mengucapkan terima kasih kepada Tuhan Yesus Kristus, karena oleh campur tangan-Nya, saya bisa menyelesaikan perkuliahan, tugas dan ujian yang harus saya lalui di semester 4 ini. Saya mengucapkan terima kasih kepada Ibu Masayu, Ibu Ulfa dan Bapak Rinaldi, sebagai dosen-dosen dari mata kuliah IF2211 Strategi Algoritma, atas bimbingan dan ilmu yang sudah diberikan selama 1 semester ini pada mata kuliah tersebut.

References

[1] Levitin, Anany. 2012. *Introduction to Design and Analysis of Algorithm 3rd Edition*. Addison-Wesley.

[2] Russel, Stuart & Norvig, Peter. 2010. *Artificial Intelligence, A Modern Approach, 3rd Edition*. Pearson Education.

[3] Slide kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung.

[4] https://hc11.home.xs4all.nl/15puzzle/15puzzen.htm terakhir diakses tanggal 18 Mei 2017

[5] https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/ terakhir diakses tanggal 18 Mei 2017

[6] https://www.cs.princeton.edu/courses/archive/fall12/cos226/assignments/8puzzle.html terakhir diakses tanggal 18 Mei 2017

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, 19 Mei 2017

Marvin Jerremy Budiman (13515076)