TUGAS

ARTIFICIAL INTELLIGENCE

PROBLEM SOLVING 8 PUZZLE WITH A* AND GREEDY ALGORITHM IN PYTHON

Nama
Jarot Achid Alvian
433780

Fakultas/Sekolah
S2 ILMU KOMPUTER - MII
UNIVERSITAS GADJAH MADA

2019

I. PENDAHULUAN

Permainan puzzle dengan 8 kotak merupakan salah satu permainan yang populer, dimana kita harus merubah dari kondisi awal atau *initial state* ke kondisi yang yang ditentukan atau *final state*. Cara memainkan permainan ini cukup mudah yaitu dengan cara menggeser kotak satu per-satu sehingga dapat memenuhi kondisi yang diinginkan.

Permainan ini dapat digunakan untuk penerapan algoritma untuk dapat menyelesaikan permasalahan ini. Sehingga secara otomatis dapat mencari jalur sendiri atau *pathfinding* sampai kondisi yang diinginkan.

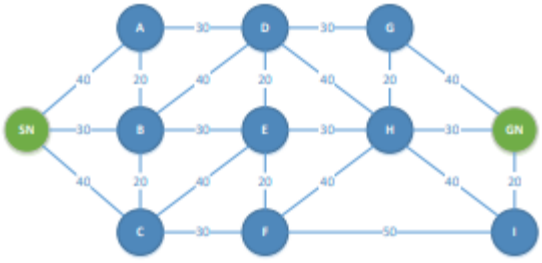
Terdapat beberapa jenis algoritma yang bisa digunakan, tetapi untuk menyelesaikan permainan puzzle dengan 8 kotak ini akan menggunakan algoritma A*. Karena algoritma A* dinilai dapat menemukan solusi terbaik.

II. LANDASAN TEORI

A. Path Finding

Path Finding merupakan salah satu materi yang sangat penting didalam *Artificial Intelligence*. *Path Finding* biasanya digunakan untuk menyelesaikan masalah pada sebuah *graph*. Dalam matematika *graph* merupakan himpunan titik-titik atau biasa disebut dengan *node* yang tehubung oleh *edge*. *Edge* yang menghubungkan setiap *node* merupakan suatu vektor yang memiliki arah dan besaran tertentu.

Untuk dapat menemukan jalan dari *Node* Awal menuju *Node* Tujuan, dilakukan penelusuran terhadap graph tersebut. Penelusuran biasanya dilakukan dengan mengikuti arah *edge* yang menghubungkan antar *node*.



Gambar 1. Contoh *Graph*

Pada gambar 1, suatu *graph* dengan *Node* Awal SN dan *Node* Tujuan GN terhubung dengan *node-node* lain oleh *edge-edge* yang memiliki besaran yang berbeda. Jika ditelusuri, terdapat banyak kombinasi rute yang dapat dilalui untuk menuju node tujuan. Bisa dikatakan dari *graph* tersebut, setiap *node* akan memberikan solusi arah menuju *node* tujuan.

B. Shortest Path

Shortest Path atau rute terpendek merupakan suatu upaya optimalisasi dari *path finding*. Setiap rute yang ditemukan pada *path finding* akan dicari rute terpendeknya. Pencarian rute

terpendeknya di tentukan dengan akumulasi besaran vektor yang dilalui untuk mencapai node tujuan. Akumulasi *cost* yang memiliki nilai minimum merupakan rute terpendek dari *graph* tersebut.

Ada beberapa macam persoalan lintasan terpendek, antara lain:

1. Lintasan terpendek antara dua buah simpul tertentu (*a pair shortest path*).
2. Lintasan terpendek antara semua pasangan simpul (*all pair shortest path*).
3. Lintasan terpendek dari simpul tertentu ke semua simpul yang lain (*single-source shortest path*).
4. Lintasan terpendek antara dua buah simpul yang melalui beberapa simpul tertentu (*intermediate shortest path*). (Ulva, 2014)

C. Algoritma Greedy

"Algoritma adalah urutan logis langkahlangkah penyelesaian masalah yang disusun secara sistematis" [1]. Awalnya kata algoritma merupakan istilah yang merujuk kepada aturan-aturan aritmetis untuk menyelesaikan persoalan dengan menggunakan bilangan numerik Arab, namun pada abad ke-18 istilah ini telah berkembang sehingga makna algoritma menjadi lebih luas lagi menjadi suatu urutan langkah atau prosedur yang jelas dan diperlukan untuk menyelesaikan suatu permasalahan. Kata algoritma berasal dari latinisasi nama seorang ahli matematika dari Uzbekistan Al Khawārizmi (hidup sekitar abad ke-9), sebagaimana tercantum pada terjemahan karyanya lam bahasa latin dari abad ke-12 "Algorithmi de numero Indorum".

Algoritma Greedy merupakan metode yang paling populer dalam memecahkan persoalan optimasi [2]. Hanya ada dua macam persoalan optimasi, yaitu maksimasi dan minimasi. Pada penelitian ini, algoritma greedy yang digunakan menerapkan pendekatan maksimasi. Algoritma Greedy adalah algoritma yang memecahkan masalah langkah per langkah. Pada setiap langkah terdapat banyak pilihan yang perlu dieksplorasi. Persoalan optimalisasi dalam konteks algoritma greedy disusun oleh elemen-elemen sebagai berikut:

1. Himpunan Kandidat, C.

   Himpunan ini berisi elemen-elemen pembentuk solusi. Pada setiap langkah, satu buah kandidat diambil dari himpunannya.

2. Himpunan Solusi, S.

   Himpunan ini berisi kandidat-kandidat yang terpilih sebagai solusi persoalan. Dengan kata lain, himpunan solusi adalah himpunan bagian dari himpunan kandidat.

3. Fungsi Seleksi (Selection Function)

   Fungsi ini dinyatakan dengan predikat seleksi. Merupakan fungsi yang pada setiap langkah memilih kandidat yang paling memungkinkan mencapai solusi optimal. Kandidat

yang sudah dipilih pada suatu langkah tidak pernah dipertimbangkan lagi pada langkah selanjutnya.

4. Fungsi Kelayakan (Feasible)

   Fungsi ini dinyatakan dengan predikat layak. Fungsi kelayakan ini merupakan fungsi yang memeriksa apakah suatu kandidat yang telah dipilih dapat memberikan solusi yang layak, yakni kandidat tersebut bersama-sama dengan himpunan solusi yang sudah terbentuk tidak melanggar batasan/aturan (constraints) yang ada. Kandidat yang layak dimasukkan ke dalam himpunan solusi, sedangkan yang tidak layak dibuang dan tidak pernah dipertimbangkan lagi.

5. Fungsi Obyektif

   Fungsi obyektif ini merupakan sebuah fungsi yang memaksimumkan atau meminimumkan nilai solusi. Dengan kata lain, algoritma greedy melibatkan pencarian sebuah himpunan bagian, S, dari himpunan kandidat, C yang dalam hal ini, S harus memenuhi beberapa kriteria yang ditentukan, yaitu menyatakan suatu solusi dan S dioptimasi oleh fungsi obyektif. Ada kalanya solusi optimum global yang diperoleh dari algoritma greedy yang diharapkan sebagai solusi optimum dari persoalan, belum tentu merupakan solusi optimum (terbaik), tetapi solusi sub-optimum atau pseudo-optimum. Hal ini dikarenakan algoritma greedy tidak beroperasi secara menyeluruh terhadap semua alternatif solusi yang ada dan terdapat beberapa fungsi seleksi yang berbeda, yaitu jika fungsi seleksi tidak identik dengan fungsi obyektif karena fungsi seleksi biasanya didasarkan pada fungsi obyektif. Sehingga harus dipilih fungsi yang tepat jika menginginkan algoritma menghasilkan solusi optimal atau nilai yang optimum. Jadi, pada sebagian masalah algoritma greedy tidak selalu berhasil memberikan solusi yang benar-benar optimum, tetapi algoritma greedy pasti memberikan solusi yang mendekati (approximation) nilai optimum.

Menurut Angga & Munir [1] jika jawaban terbaik mutlak tidak diperlukan, maka algoritma greedy sering berguna untuk menghasilkan solusi cukup baik (approximation), daripada menggunakan algoritma yang lebih rumit untuk menghasilkan solusi yang terbaik. Bila algoritma greedy optimum, maka keoptimalannya itu dapat dibuktikan secara matematis.

D. Algoritma A*

Algoritma A Star atau A* adalah salah satu algoritma pencarian yang menganalisa *input*, mengevaluasi sejumlah jalur yang mungkin dilewati dan menghasilkan solusi. Algoritma A* adalah algoritma komputer yang digunakan secara luas dalam *graph* traversal dan penemuan jalur serta proses perencanaan jalur yang bisa dilewati secara efisien di sekitar titik-titik yang disebut *node* (Reddy, 2013).

Karakteristik yang menjelaskan algoritma A* adalah pengembangan dari "daftar tertutup" untuk merekam area yang dievaluasi. Daftar tertutup ini adalah sebuah daftar untuk merekam area berdekatan yang sudah dievaluasi, kemudian melakukan perhitungan jarak yang dikunjungi dari "titik awal" dengan jarak diperkirakan ke "titik tujuan" (Reddy, 2013).

Algoritma A* menggunakan path dengan *cost* paling rendah ke node yang membuatnya sebagai algoritma pencarian nilai pertama yang terbaik atau *best first search*. Menggunakan rumus

$(x) = (x) + h(x)$ .....(1)

dimana:

- (x) adalah jarak total dari posisi asal ke lokasi sekarang.
- $h(x)$ adalah fungsi heuristik yang digunakan untuk memperkirakan jarak dari lokasi sekarang ke lokasi tujuan. Fungsi ini jelas berbeda karena ini adalah perkiraan semata dibandingkan dengan nilai aslinya. Semakin tinggi keakuratan heuristik, semakin cepat dan bagus lokasi tujuan ditemukan dan dengan tingkat keakuratan yang lebih baik. Fungi $(x) = (x) + h(x)$ ini adalah perkiraan saat ini dari jarak terdekat ke tujuan (Lubis, 2016).

Algoritm A* juga menggunakan 2 (dua) senarai *Open List* dan *Closed List* sama seperti algoritma dasar *Best First Search*. Terdapat 3 (tiga) kondisi bagi setiap suksesor yang dibangkitkan, yaitu sudah berada pada di *Open*, sudah berada di *Closed*, dan tidak berada di *Open* maupun *Closed*. Pada ketiga kondisi tersebut diberikan penanganan yang berbeda-beda (Suyanto, 2014).

Jika suksesor sudah pernah berada di *Open*, maka dilakukan pengecekan apakah perlu pengubahan *parent* atau tidak tergantung pada nilai g-nya melalui *parent* lama atau *parent* baru. Jika melalui parent baru memberikan nilai g yang lebih kecil, maka dilakukan pengubahan *parent*. Jika pengubahan *parent* dilakukan, maka dilakukan pula *update* nilai g dan f pada suksesor tersebut. Dengan perbaharuan ini, suksesor tersebut memiliki kesempatan yang lebih besar untuk terpilih sebagai simpul terbaik (*best node*) (Suyanto, 2014).

Jika suksesor sudah pernah berada di *Closed*, maka dilakukan pengecekan apakah perlu pengubahan parent atau tidak, jika ya, maka dilakukan perbaharuan nilai g dan f pada suksesor tersebut serta pada semua "anak cucunya" yang sudah pernah berada di *Open*. Dengan perbaharuan ini, maka semua anak cucunya tersebut memiliki kesempatan lebih besar untuk terpilih sebagai simpul terbaik (*best node*) (Suyanto, 2014).

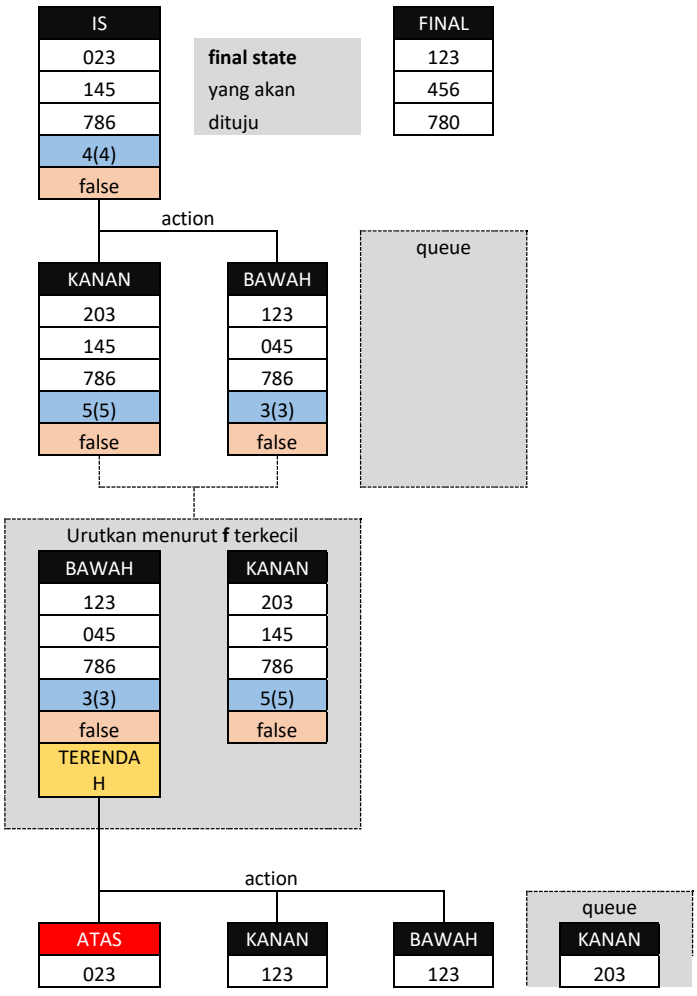
Jika suksesor tidak berada di *Open* maupun *Closed*, maka suksesor tersebut dimasukkan kedalam *Open*. Tambahkan suksesor tersebut sebagai suksesornya *best node*. Hitung *cost* suksesor tersebut dengan menggunakan persamaan 1 (Suyanto, 2014).

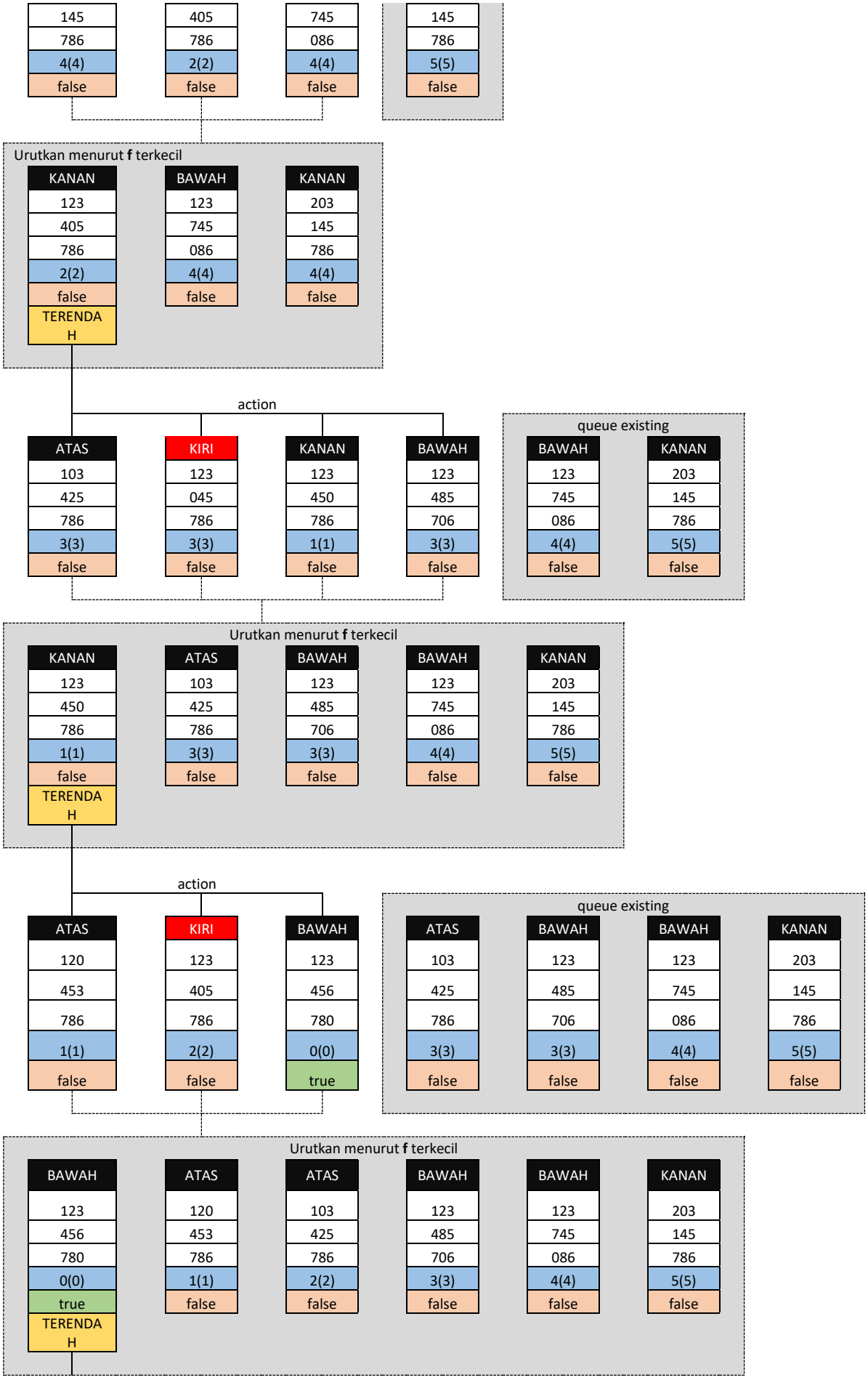
III. HEURISTIC

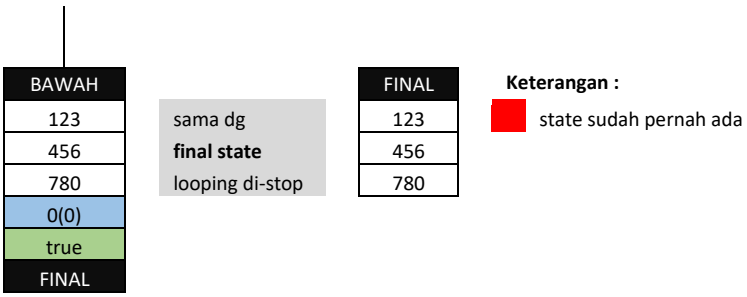
Secara umum permainan adalah merubah dari *initial state* kotak(12345678) digerakkan ke kotak kosong(0) sehingga dapat sama dengan *final state*. Langkah awal adalah pembuatan variable untuk menampung semua *state* kecuali *parent state* dan *state* yang pernah muncul. Kemudian dari variable tampungan state tersebut di urutkan menurut *cost* yang dibutuhkan, dan state dengan *cost* terkecil akan menjadi *parent state* yang baru.

Kemudian parent state tersebut akan dicek apakah sudah sama dengan final state atau belum jika sudah maka solusi dari 8 puzzle sudah ditemukan. Tapi apabila belum sama maka *parent state* tersebut akan berikan aksi sehingga akan memunculkan *state* baru yang akan disebut dengan *child state*, karena maximal aksi yang akan dilakukan hanyalah 4 aksi yaitu : atas, bawah, kiri, dan kanan maka jumlah maximal dari *child state* tersebut adalah 4, kemudian setiap *child state* tersebut dicari *cost* menggunakan fungsi *manhattan*, dan status ke-samaan dengan *final state*. Dan kemudian *child state* tersebut di masukkan ke variable tampungan apabila *state* belum pernah muncul. Setelah itu proses akan kembali seperti proses awal, sampai ditemukan state yang memiliki status yang sama dengan final state. Berikut penjelasan menggunakan gambar :

A. Algoritma Greedy

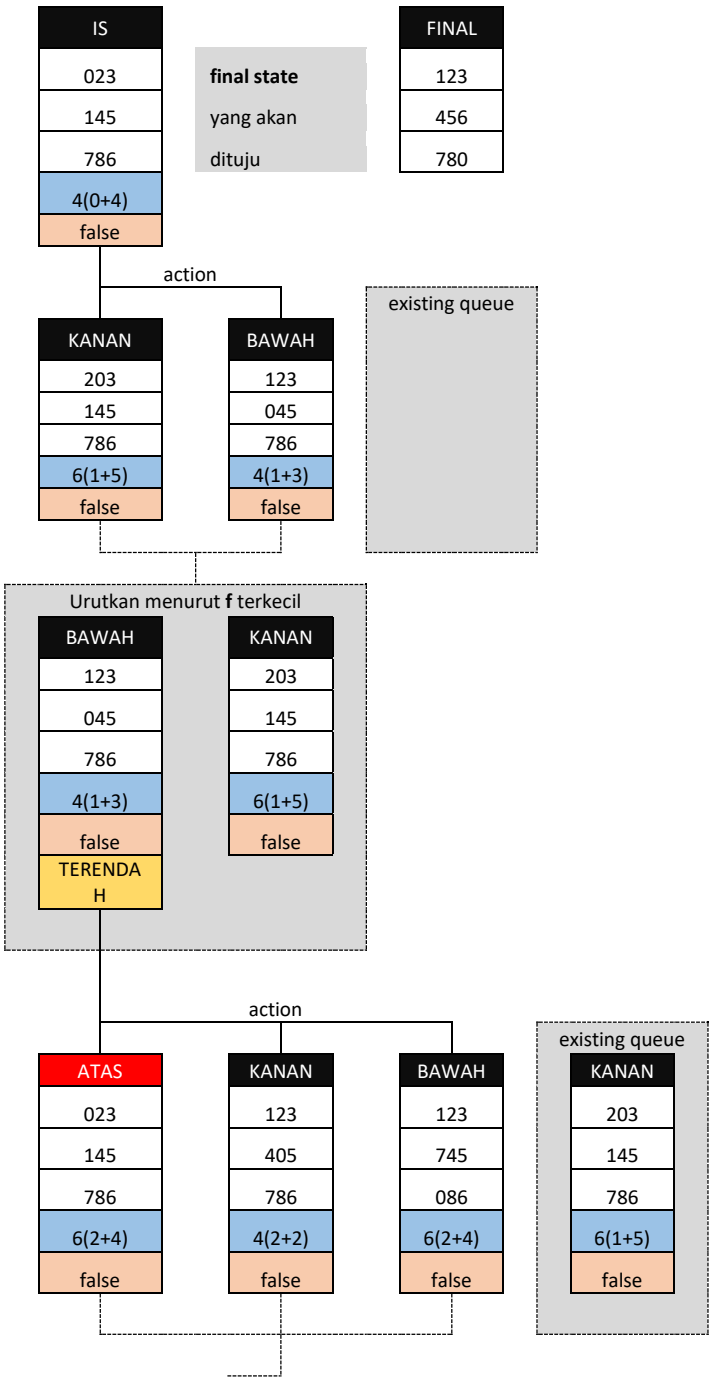


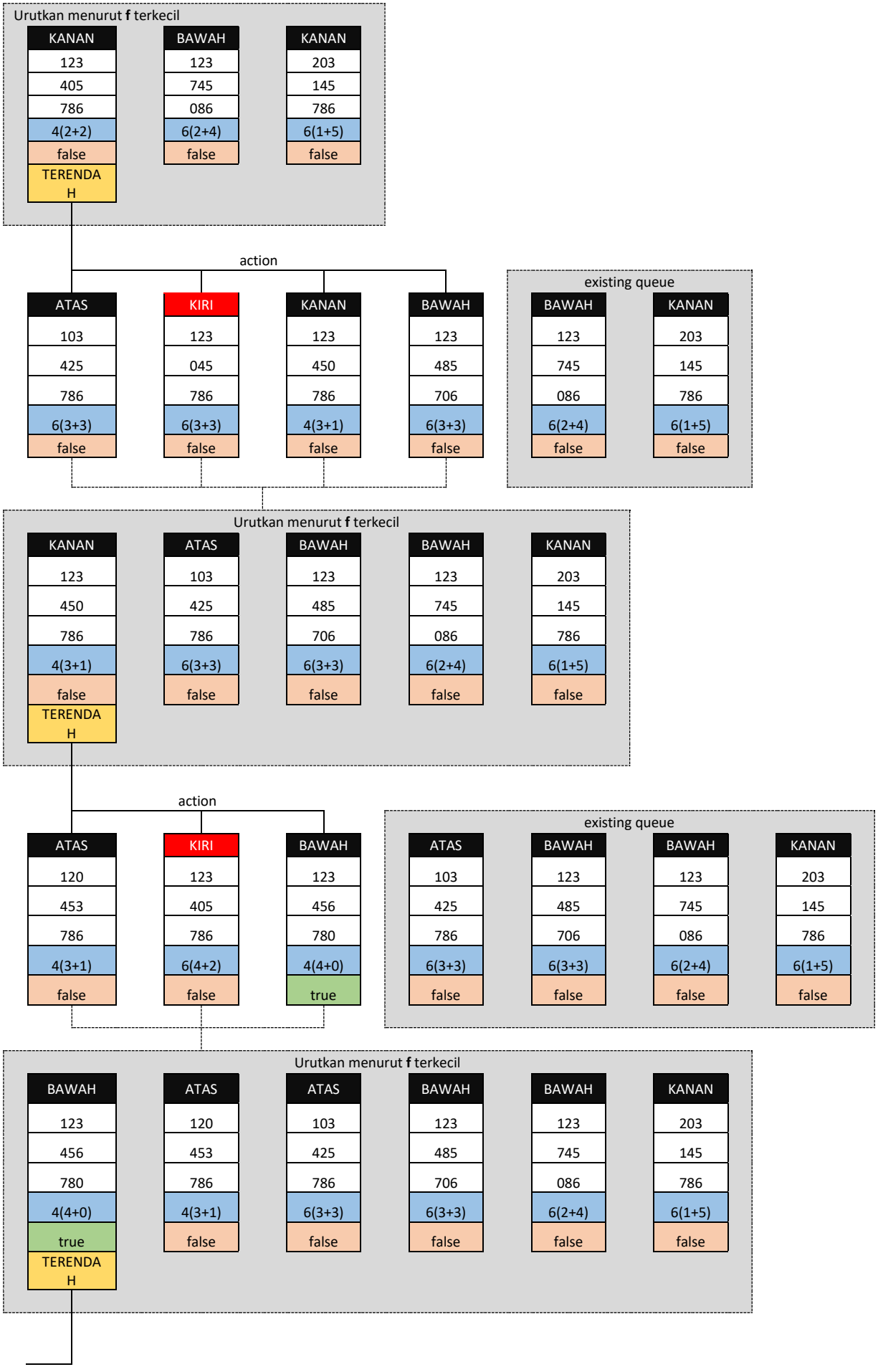

145
786
4(4)
false
405
786
2(2)
false
745
086
4(4)
false
145
786
5(5)
false
Urutkan menurut f terkecil
KANAN
123
405
786
2(2)
false
TERENDAH
BAWAH
123
745
086
4(4)
false
KANAN
203
145
786
4(4)
false
action
ATAS
103
425
786
3(3)
false
KIRI
123
045
786
3(3)
false
KANAN
123
450
786
1(1)
false
BAWAH
123
485
706
3(3)
false
queue existing
BAWAH
123
745
086
4(4)
false
KANAN
203
145
786
5(5)
false
Urutkan menurut f terkecil
KANAN
123
450
786
1(1)
false
TERENDAH
ATAS
103
425
786
3(3)
false
BAWAH
123
485
706
3(3)
false
BAWAH
123
745
086
4(4)
false
KANAN
203
145
786
5(5)
false
action
ATAS
120
453
786
1(1)
false
KIRI
123
405
786
2(2)
false
BAWAH
123
456
780
0(0)
true
queue existing
ATAS
103
425
786
3(3)
false
BAWAH
123
485
706
3(3)
false
BAWAH
123
745
086
4(4)
false
KANAN
203
145
786
5(5)
false
Urutkan menurut f terkecil
BAWAH
123
456
780
0(0)
true
TERENDAH
ATAS
120
453
786
1(1)
false
ATAS
103
425
786
2(2)
false
BAWAH
123
485
706
3(3)
false
BAWAH
123
745
086
4(4)
false
KANAN
203
145
786
5(5)
false

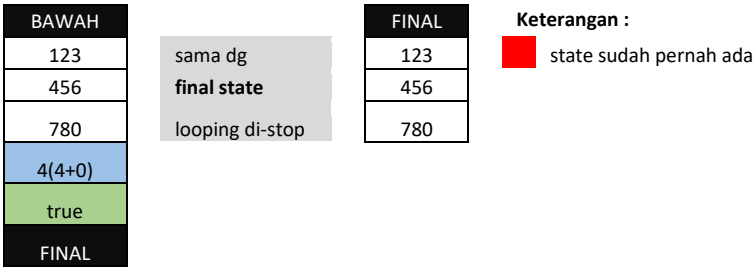


Gambar 2. Contoh Alur Proses dengan Greedy

B. Algoritma A*




Urutkan menurut f terkecil
KANAN
123
405
786
4(2+2)
false
TERENDAH
BAWAH
123
745
086
6(2+4)
false
KANAN
203
145
786
6(1+5)
false
action
ATAS
103
425
786
6(3+3)
false
KIRI
123
045
786
6(3+3)
false
KANAN
123
450
786
4(3+1)
false
BAWAH
123
485
706
6(3+3)
false
existing queue
BAWAH
123
745
086
6(2+4)
false
KANAN
203
145
786
6(1+5)
false
Urutkan menurut f terkecil
KANAN
123
450
786
4(3+1)
false
TERENDAH
ATAS
103
425
786
6(3+3)
false
BAWAH
123
485
706
6(3+3)
false
BAWAH
123
745
086
6(2+4)
false
KANAN
203
145
786
6(1+5)
false
action
ATAS
120
453
786
4(3+1)
false
KIRI
123
405
786
6(4+2)
false
BAWAH
123
456
780
4(4+0)
true
existing queue
ATAS
103
425
786
6(3+3)
false
BAWAH
123
485
706
6(3+3)
false
BAWAH
123
745
086
6(2+4)
false
KANAN
203
145
786
6(1+5)
false
Urutkan menurut f terkecil
BAWAH
123
456
780
4(4+0)
true
TERENDAH
ATAS
120
453
786
4(3+1)
false
ATAS
103
425
786
6(3+3)
false
BAWAH
123
485
706
6(3+3)
false
BAWAH
123
745
086
6(2+4)
false
KANAN
203
145
786
6(1+5)
false



Gambar 3. Contoh Alur Proses dengan A*

IV. PENJELASAN CODE

Langkah pertama adalah membuat kondisi awal puzzle dan menjadikkan nya objek dengan kode :

```
""" deklarasi state awal pada board """
board = [[0,2,3],[1,4,5],[7,8,6]]

""" me-masukkan board ke dalam class puzzle """
puzzle = Puzzle(board)
```

Kemudian pencarian solusi dengan memanggil kelas *Solver* dengan parameter object kondisi awal yang disimpan dengan nama varibel *puzzle*

```
""" Mencari solusi solusi dengan kelas solver """
solver = Solver(puzzle)
```

Dari proses tersebut urutan *initial state* sampai *final state* akan menjadi kembalian nilai dari kelas *Solver* dan akan kita simpan dalam varibel *solver_datas*

```
""" Node solusi disimpan dalam variable solver_datas """
solver_datas = solver.solve()
```

Dalam kelas Solver ini objek *initial state* akan dijadikan objek dari kelas *Node* dan dimasukkan kedalam *collections* dan objek node *initial state* tersebut dimasukkan ke dalam *collections* dengan nama variable *queue*.

```
      """ Data awal dimasukkan ke collections """
        queue = collections.deque([Node(self.start)])
```

Kemudian membuat set dengan variable *seen* dan data pertama dari *queue* dimasukkan ke set tersebut

```
        """ Buat variable seen untuk memasukkan data node yang sudah di-loop """
        seen = set()
        """ Masukkan root ke variable seen """
        seen.add(queue[0].state)
```

Kemudian data dalam variable *queue* tadi dilakukan proses *BFS* dengan menggunkanan *while*

```
        """ BFS"""
        while queue:
```

Kemudian urutkan isi dari dari *queue* dengan urutan *cost* terendah ke tertinggi, kemudian masukkan data terendah ke dalam variable *node* dan hapus data tersebut pada *queue*, proses ini bisa disebut **proses ekstrak** untuk memudahkan dalam penyebutan selanjutnya

```
            """Sorting child berdasarkan nilai F yang paling kecil, jadi nilai terkecil akan
di ekstrak paling awal"""
            queue = collections.deque(sorted(list(queue), key=lambda node: node.f))

            print((queue[1]));

            """ masukkan nilai terkecil dari queue ke variable node """
            node = queue.popleft()
```

Kemudian lakukan cek apakah *node* tersebut apakah sudah sama kondisi statenya dengan *final state*, jika sudah maka kembalikan *path node* nya

```
            """ mengembalikan path node apabila sudah ketemu solusi """
            if node.solved:
                return node.path
```

Apabila kondisi belum sama maka lakukan *looping* terhadap node tersebut menurut *action* yang bisa digunakan atas, bawah, kanan kiri. Proses ini bisa disebut **proses aksi** untuk memudahkan dalam penyebutan selanjutanya

```
for move, action in node.actions:
```

kemudian buatlah varibel *child* untuk menyimpan *state* hasil dari *action* dari node awal tersebut sehingga dapat mendapatkan nilai *status* dan *cost* menggunakan *manhattan* untuk perhitungan selanjutnya

```
  """ Buat variable child untuk menyimpan node child """
                child = Node(move(), node, action)
```

kemudian lalukan cek dalam variable seen apakah state dari child tersebut sudah ada atau belum jika belum ada makan masukkan node child ke dalam seen dan masukkan paling kiri ke *queue*

```
  """
                Check apakah state TIDAK ADA dalam variable seen?
                - masukkan child state ke variable queue paling kiri
                - masukkan child state ke variable seen
                """
                if child.state not in seen:
                    queue.appendleft(child)
                    seen.add(child.state)
```

Kemudian tunggu sampai proses **proses aksi** selesai, kemudian program akan kembali ke **proses ekstrak** dan akan berputar terus sampai status solve di temukan, jika tidak ditemukan maka akan terjadi *infinity loop*.

Yang terakhir kita tinggal menampilkan proses akhir step per-step state yang berjalan dengan melakukan *looping* dari kembalian data yang disimpan dalam variable *solver_datas.* Dalam objek tersebut banyak data yang tersimpan dan bisa kita tampilkan seperti : *action* atau pergerakan puzzle, *cost* yang dibutuhkan, status *state* apakah sama dengan *final state*, dan urutan *state* tersebut.

```
""" Menampilkan hasil jadi dari puzzle """
print()
print("=====================================================")
print("RANGKAIAN URUTAN PUZZLE")
print("-----------------------------------------------------")
steps = 0
for data in solver_datas:
    print("STEP " + str(steps+1) + " ")
    print("- Geser      : " + str(data.action))
    print("  F(G+H)     : " + str(data.score) +  " (" + str(data.gg) +  "+" + str(data.h) +
")")
    print("  solved     : "+ str(data.solved))
    print_puzzle(data, "               ")
    steps += 1
print()
print("-----------------------------------------------------")
print("TOTAL STEP   : " + str(steps) + " step")
print("=====================================================")
```

Khusus untuk mencetak bentuk state dibutuhkan fungsi khusus untuk mencetak dari *raw* ke bentuk seperti kotak puzzle pada umumnya dengan menggunakan fungsi *print_puzzle,* berikut cara penggunaan dan kode fungsi tersebut :

```
    print_puzzle(data, "               ")
```

```
def print_puzzle(datas, separator):
    """
    melakukan cetakan puzzle
    dengan masukan data raw puzzle dan separator di depannya
    """
    x=1
    boards = ""
    for board in str(datas):
        boards += board + ""
        if(x%3==0) :
            print(separator + "|" + boards + "|")
            boards = ""
        else :
            boards += " "
        x+=1
```

Secara umum kode dari algoritma C* dan Greedy sama, hanya saja dalam Greedy nilai f = heuristik, berbeda dengan A*.

V. GITHUB

Berikut ini adalah link github yang telah di-upload :

https://github.com/djatoyz/8-Puzzle-Problem

VI. INPUT DAN OUTPUT

Berikut salah satu contoh bentuk puzzle awal dan puzzle tujuan dan dan urusan perpindahan puzzle sehingga kondisi yang diinginkan terpenuhi :

Initial state :

```
|0 2 3|
|1 4 5|
|7 8 6|
```

Final state :

```
|1 2 3|
|4 5 6|
|7 8 0|
```

A. Algoritma Greedy

```
=====================================================
RANGKAIAN PROSES BFS
=====================================================
Jns_Node Num. Action - F(H) -> Result_Solving
-----------------------------------------------------

PARENT 1. None - 4(4) -> False
|0 2 3|
|1 4 5|
|7 8 6|

LIST CHILDS   :
Child 1.1. Kanan - 6(5) -> False
|2 0 3|
|1 4 5|
|7 8 6|
Child 1.2. Bawah - 4(3) -> False
|1 2 3|
|0 4 5|
|7 8 6|


 PARENT 2. Bawah - 4(3) -> False
 |1 2 3|
 |0 4 5|
 |7 8 6|

 LIST CHILDS   :
 Child 2.1. Atas - 6(4) -> False
 |0 2 3|
 |1 4 5|
 |7 8 6|
 Child 2.2. Kanan - 4(2) -> False
 |1 2 3|
 |4 0 5|
 |7 8 6|
 Child 2.3. Bawah - 6(4) -> False
 |1 2 3|
 |7 4 5|
```

```
 |0 8 6|


  PARENT 3. Kanan - 4(2) -> False
  |1 2 3|
  |4 0 5|
  |7 8 6|

  LIST CHILDS   :
  Child 3.1. Atas - 6(3) -> False
  |1 0 3|
  |4 2 5|
  |7 8 6|
  Child 3.2. Kiri - 6(3) -> False
  |1 2 3|
  |0 4 5|
  |7 8 6|
  Child 3.3. Kanan - 4(1) -> False
  |1 2 3|
  |4 5 0|
  |7 8 6|
  Child 3.4. Bawah - 6(3) -> False
  |1 2 3|
  |4 8 5|
  |7 0 6|


   PARENT 4. Kanan - 4(1) -> False
   |1 2 3|
   |4 5 0|
   |7 8 6|

   LIST CHILDS   :
   Child 4.1. Atas - 6(2) -> False
   |1 2 0|
   |4 5 3|
   |7 8 6|
   Child 4.2. Kiri - 6(2) -> False
   |1 2 3|
   |4 0 5|
   |7 8 6|
   Child 4.3. Bawah - 4(0) -> True
   |1 2 3|
   |4 5 6|
   |7 8 0|


    PARENT 5. Bawah - 4(0) -> True
    |1 2 3|
    |4 5 6|
    |7 8 0|

==========================================================
RANGKAIAN URUTAN PUZZLE
----------------------------------------------------------
STEP 1
- Geser      : None
  F(G+H)     : 4 (4)
  solved     : False
               |0 2 3|
               |1 4 5|
               |7 8 6|
```

```
STEP 2
- Geser       : Bawah
  F(G+H)      : 4 (3)
  solved      : False
                |1 2 3|
                |0 4 5|
                |7 8 6|
STEP 3
- Geser       : Kanan
  F(G+H)      : 4 (2)
  solved      : False
                |1 2 3|
                |4 0 5|
                |7 8 6|
STEP 4
- Geser       : Kanan
  F(G+H)      : 4 (1)
  solved      : False
                |1 2 3|
                |4 5 0|
                |7 8 6|
STEP 5
- Geser       : Bawah
  F(G+H)      : 4 (0)
  solved      : True
                |1 2 3|
                |4 5 6|
                |7 8 0|

------------------------------------------------------
TOTAL STEP    : 5 step
======================================================
```

B. Algoritma A*

Proses :

```
======================================================
RANGKAIAN PROSES BFS
======================================================
Jns_Node Num. Action - F(G+H) -> Result_Solving
------------------------------------------------------

PARENT 1. None - 4(0+4) -> False
|0 2 3|
|1 4 5|
|7 8 6|

LIST CHILDS   :
Child 1.1. Kanan - 6(1+5) -> False
|2 0 3|
|1 4 5|
|7 8 6|
Child 1.2. Bawah - 4(1+3) -> False
|1 2 3|
|0 4 5|
|7 8 6|


 PARENT 2. Bawah - 4(1+3) -> False
```

|1 2 3|
|0 4 5|
|7 8 6|

LIST CHILDS :
Child 2.1. Atas - 6(2+4) -> False
|0 2 3|
|1 4 5|
|7 8 6|
Child 2.2. Kanan - 4(2+2) -> False
|1 2 3|
|4 0 5|
|7 8 6|
Child 2.3. Bawah - 6(2+4) -> False
|1 2 3|
|7 4 5|
|0 8 6|

PARENT 3. Kanan - 4(2+2) -> False
|1 2 3|
|4 0 5|
|7 8 6|

LIST CHILDS :
Child 3.1. Atas - 6(3+3) -> False
|1 0 3|
|4 2 5|
|7 8 6|
Child 3.2. Kiri - 6(3+3) -> False
|1 2 3|
|0 4 5|
|7 8 6|
Child 3.3. Kanan - 4(3+1) -> False
|1 2 3|
|4 5 0|
|7 8 6|
Child 3.4. Bawah - 6(3+3) -> False
|1 2 3|
|4 8 5|
|7 0 6|

PARENT 4. Kanan - 4(3+1) -> False
|1 2 3|
|4 5 0|
|7 8 6|

LIST CHILDS :
Child 4.1. Atas - 6(4+2) -> False
|1 2 0|
|4 5 3|
|7 8 6|
Child 4.2. Kiri - 6(4+2) -> False
|1 2 3|
|4 0 5|
|7 8 6|
Child 4.3. Bawah - 4(4+0) -> True
|1 2 3|
|4 5 6|
|7 8 0|

```
    PARENT 5. Bawah - 4(4+0) -> True
    |1 2 3|
    |4 5 6|
    |7 8 0|

=====================================================
RANGKAIAN URUTAN PUZZLE
-----------------------------------------------------
STEP 1
- Geser      : None
  F(G+H)     : 4 (0+4)
  solved     : False
               |0 2 3|
               |1 4 5|
               |7 8 6|
STEP 2
- Geser      : Bawah
  F(G+H)     : 4 (1+3)
  solved     : False
               |1 2 3|
               |0 4 5|
               |7 8 6|
STEP 3
- Geser      : Kanan
  F(G+H)     : 4 (2+2)
  solved     : False
               |1 2 3|
               |4 0 5|
               |7 8 6|
STEP 4
- Geser      : Kanan
  F(G+H)     : 4 (3+1)
  solved     : False
               |1 2 3|
               |4 5 0|
               |7 8 6|
STEP 5
- Geser      : Bawah
  F(G+H)     : 4 (4+0)
  solved     : True
               |1 2 3|
               |4 5 6|
               |7 8 0|

-----------------------------------------------------
TOTAL STEP   : 5 step
=====================================================
```

VII. PERBANDINGAN

Perbandingan antara penggunaan algoritma Greedy dan A* dalam menyelesaikan 8 puzzle, antara lain algoritma Greedy 5 step dan A* 5 step. Dan dari hasil perbandingan tersebut membuktikan bahwa algoritma A* sama dengan algoritma Greedy.

VIII. DAFTAR PUSTAKA

[1] Angga, C., & Munir, R. 2012. Pengembangan Algoritma Greedy untuk Optimalisasi Penataan Peti Kemas Pada Kapal Pengangkut. Jurnal Sarjana Institut Teknologi Bandung bidang Teknik Elektro dan Informatika.

[2] Juniar, Ahmad. 2015. Penerapan Algoritma Greedy pada Penjadwalan Produksi Single-Stage dengan Parallel Machine di Industri Konveksi.Vol. 16, No. 2. Jakarta.