Perbandingan Algoritma *Brute Force* dan Algoritma Runut-balik pada Pencarian Solusi di Permainan *Infinity Loop*

Muhammad Kamal Nadjieb - 13514054
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13514054@std.stei.itb.ac.id

***Abstract*—Permainan *Infinity Loop* adalah salah satu permainan *puzzle* yang terkenal di tahun 2016. Tujuan dari permainan ini adalah menyelesaikan *puzzle* yang diberikan sehingga terbentuk sebuah gambar dengan pola *infinity loop*. Dalam makalah ini akan dibahas tentang perbandingan algoritma *brute force* dan algoritma runut-balik pada pencarian solusi di permainan *Infinity Loop*.**

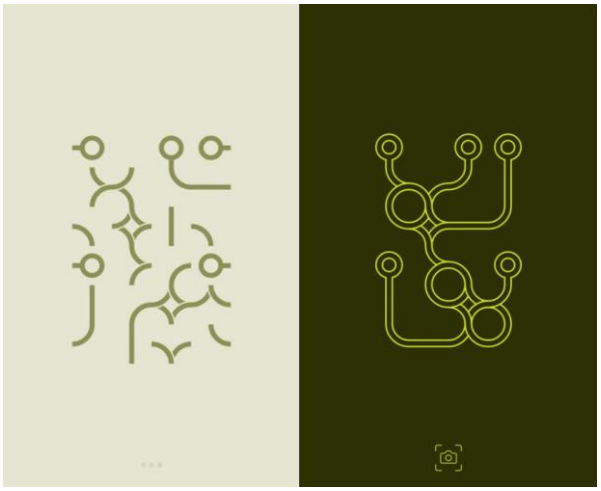
Kata kunci—infinity loop; brute force; runut-balik; bactracking

I. PENDAHULUAN

Permainan ∞ *Infinity Loop*, yang selanjutnya permainan ini akan dituliskan sebagai *Infinity Loop* untuk kemudahan penulisan, adalah salah satu permainan *puzzle* yang terkenal di tahun 2016. Permainan ini dikeluarkan oleh ∞ *Infinity Games* dan bisa dimainkan di *iOS* dengan mengunduhnya dari *App Store* dan di *Android* dengan mengunduhnya dari *Google Play*. Sampai dengan tanggal 7 Mei 2016, permainan ini telah di unduh sebanyak lima juta pengguna Android dan memiliki *rating* sebesar 4,4 dari skala tertinggi 5.

Permainan ini terdiri dari *level-level* yang jumlahnya tak hingga. Di masing-masing *level*, pemain harus menyelesaikan *puzzle* yang diberikan sehingga terbentuk sebuah gambar dengan pola *infinity loop*. Yang dimaksud dengan pola *infinity loop* di sini adalah masing-masing bagian pada *puzzle* terhubung satu sama lain tanpa terkecuali.

Algoritma *brute force* sendiri adalah sebuah pendekatan yang lempang untuk memecahkan suatu masalah yaitu mengenumerasi semua kemungkinan solusi dan mengambil solusi terbaik darinya. Algoritma runut-balik adalah perbaikan dari algoritma brute force yang berbasis pada pencarian mendalam (*Depth First Search*) untuk mencari solusi persoalan secara lebih mangkus. Dalam makalah ini, penulis akan membandingkan algoritma brute force dengan algoritma runut-balik dengan kasus yaitu mencari solusi dari suatu *level* pada permainan *Infinity Loop*.



Gambar 1 Tampilan *Level* 12 sebelum dan sesudah diselesaikan pada permainan *Infinity Loop*

II. DASAR TEORI

A. Algoritma Brute Force

1) Definisi Brute Force

Brute force adalah sebuah pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah yang biasanya didasarkan pada pernyataan masalah (*problem statement*) dan definisi konsep yang dilibatkan. Algoritma *brute force* memecahkan masalah dengan sangat sederhana, langsung, dan dengan cara yang jelas (*obvious way*).[1]

2) Karakteristik Algoritma Brute Force

[1]Algoritma *brute force* umumnya tidak "cerdas" dan tidak mangkus karena membutuhkan jumlah langkah yang besar dalam penyelesaiannya, terutama jika masalah yang dipecahkan berukuran besar. Terkadang algoritma *brute force* disebut juga algoritma naif (*naïve algorithm*). Algoritma *brute*

force seringkali merupakan pilihan yang kurang disukai karena ketidakmangkusannya. Tetapi dengan mencari pola-pola yang mendasar, keteraturan yang ada, ataupun menggunakan trik-trik khusus, biasanya hal-hal tersebut akan membantu kita dalam menemukan algoritma yang lebih cerdas dan lebih mangkus.

Untuk masalah yang ukurannya kecil, kesederhanaan *brute force* biasanya lebih diperhitungkan daripada ketidakmangkusannya. Algoritma *brute force* sendiri sering digunakan sebagai basis untuk membandingkan beberapa alternatif algoritma yang mangkus. Misalnya pada perhitungan a^n, dengan algoritma *brute force* mudah diperlihatkan bahwa kompleksitas waktunya adalah dalam $O(n)$. Bila perhitungan a^n diselesaikan dengan teknik *divide and conquer* maka kompleksitas waktunya jauh lebih baik daripada teknik *brute force* yaitu $O(\log n)$.

Meskipun *brute force* bukan merupkan teknik pemecahan masalah yang mangkus namun teknik *brute force* dapat diterapkan pada sebagian besar masalah. Agak sulit menunjukkan masalah yang tidak dapat dipecahkan dengan teknik *brute force*, bahkan untuk kasus tertentu hanya dapat diselesaikan dengan teknik *brute force*. Beberapa pekerjaan mendasar di dalam komputer dilakukan secara *brute force* seperti menghitung jumlah dari *n* buah bilangan dan mencari elemen terbesar di dalam tabel. Selain itu, algoritma *brute force* seringkali lebih mudah diimplementasikan daripada algoritma yang lebih mangkus.

B. Prinsip Pencarian Solusi dengan Algoritma Brute Force

[1]Terminologi lain yang terkait erat dengan *brute force* adalah *exhaustive search*. Baik *brute force* maupun *exhaustive search* sering dianggap sebagai dua istilah yang sama, padahal dari jenis masalah yang dipecahkan ada sedikit perbedaan diantara keduanya.

Exhaustive search adalah teknik pencarian solusi secara *brute force* untuk masalah yang melibatkan pencarian elemen dengan sifat khusus, biasanya di antara objek-objek kombinatorik seperti permutasi, kombinasi, ataupun himpunan bagian dari sebuah himpunan.

Metode *exhaustive search* dapat dirumuskan langkah-langkahnya sebagai berikut:

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

Algoritma *exhaustive search* memeriksa secara sistematis semua kemungkinan solusi satu per satu dalam pencarian solusinya. Meskipun algoritma *exhaustive search* secara teoritis menghasilkan solusi namun waktu dan sumberdaya yang dibutuhkan dalam pencarian solusinya sangat besar.

C. Algoritma Runut-balik

[1]Runut-balik (*backtracking*) adalah algoritma yang berbasis pada pencarian mendalam (*Depth First Search*) untuk mencari solusi persoalan secara lebih mangkus. Runut-balik, yang merupakan perbaikan dari algoritma *brute force*, secara sistematis mencari solusi persoalan di antara semua kemungkinan solusi yang ada. Hanya pencarian yang mengarah ke solusi saja yang selalu dipertimbangkan. Akibatnya, waktu pencarian dapat dihemat. Runut-balik lebih alami dinyatakan dalam algoritma rekursif. Kadang-kadang disebutkan pula bahwa runut-balik merupakan bentuk tipikal dari algoritma rekursif. Saat ini algoritma runut-balik banyak diterapkan untuk program *games* dan masalah-masalah pada bidang kecerdasan buatan (*artificial intelligence*).

D. Properti Umum Metode Runut-balik dan Pengorganisasian Solusi

Untuk menerapkan metode runut-balik, property berikut didefinisikan[1]:

1) Solusi persoalan

Solusi dinyatakan sebagai vektor dengan *n-tuple*: $X = (x_1, x_2, \ldots, x_3)$, $x_i \in$ himpunan berhingga S_i. Mungkin saja

$$S_1 = S_2 = \ldots = S_n.$$

2) Fungsi pembangkit nilai x_k

Fungsi pembangkit dinyatakan sebagai:

$$T(k)$$

$T(k)$ membangkitkan nilai untuk x_k, yang merupakan komponen vektor solusi.

3) Fungsi pembatas

Pada beberapa persoalan, fungsi ini dinamakan sebagai fungsi kriteria. Dinyatakan sebagai:

$$B(x_1, x_2, \ldots, x_k)$$

Fungsi pembatas menentukan apakah $(x_1, x_2, \ldots, x_k)$ mengarah ke solusi. Jika yam aka pembangkitan nilai untuk x_{k+1} dilanjutkan. Tetapi jika tidak maka $(x_1, x_2, \ldots, x_k)$ dibuang dan tidak dipertimbangkan lagi dalam pencarian solusi.

Fungsi pembatas tidak selalu dinyatakan sebagai fungsi matematis. Ia dapat dinyatakan sebagai predikat yang bernilai *true* atau *false*, atau dalam bentuk lain yang ekivalen.

Semua kemungkinan solusi dari persoalan disebut ruang solusi (solution space). Secara formal dapat dinyatakan, bahwa jika $x_i \in S_i$, maka

$$S_1 \times S_2 \times \ldots \times S_n$$

Disebut ruang solusi. Jumlah anggota di dalam ruang solusi adalah $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_n|$.

E. Prinsip Pencarian Solusi dengan Algoritma Runut-balik

Di sini kita hanya akan meninjau pencarian solusi pada pohon ruang status yang dibangun secara dinamis. Langkah-langkah pencarian solusi adalah sebagai berikut[1]:

1. Solusi dicari dengan membentuk lintasan dari akar ke daun. Aturan pembentukan yang dipakai adalah mengikuti metode pencarian mendalam (*DFS*). Simpul-simpul yang sudah dilahirkan dinamakan simpul hidup (*live node*). Simpul hidup yang sedang diperluas dinamakan simpul-E (*Expand-node*). Simpul dinomori dari atas ke bawah sesuai dengan urutan kelahirannya.

2. Tiap kali simpul-E diperluas, lintasan yang dibangun olehnya bertambah panjang. Jika lintasan yang sedang dibentuk tidak mengarah ke solusi maka simpul-E tersebut "dibunuh" sehingga menjadi simpul mati (*dead node*). Fungsi yang digunakan untuk membunuh simpul-E adalah fungsi pembatas (bounding function). Simpul yang sudah mati tidak akan pernah diperluas lagi.

3. Jika pembentukan lintasan berakhir dengan simpul mati maka proses pencarian diteruskan dengan membangkitkan simpul anak yang lain. Bila tidak ada lagi simpul anak yang dapat dibangkitkan maka pencarian solusi dilanjutkan dengan melakukan runut-balik ke simpul hidup terdekat (simpul orangtua). Selanjutnya simpul ini menjadi simpul-E yang baru. Lintasan baru dibangun kembali sampai lintasan tersebut membentuk solusi.

4. Pencarian dihentikan bila kita telah menemukan solusi atau tidak ada lagi simpul hidup untuk runut-balik.

III. Lebih Dalam Tentang Permainan *Infinity Loop*

Pada setiap *level* diberikan sebuah *puzzle* dengan dimensi sebesar $m \times n$, dimana ada kemungkinan $m = n$. Setiap *puzzle* terdiri dari beberapa keping sejumlah $m \times n$ yang masing-masing keping mempunyai jenisnya masing-masing. Pemain hanya perlu menyentuh (*touch*) sebuah keping jika ingin merubah arah menghadapnya.

Terdapat maksimal enam jenis keping yang bisa muncul pada sebuah *puzzle*. Berikut adalah jenis-jenis keping yang bisa muncul pada sebuah puzzle:

1. Keping kosong
2. Keping satu pintu
3. Keping dua pintu lurus
4. Keping dua pintu lengkung
5. Keping tiga pintu
6. Keping empat pintu

Pada permainan *Infinity Loop* juga terdapat heuristik-heuristik yang membantu dalam menyelesaikan *puzzle* yang diberikan pada setiap level.

Puzzle pada suatu *level* dikatakan terselesaikan jika semua keping yang bergambar saling terhubung satu sama lain tanpa terkecuali sehingga *puzzle* membentuk pola *infinity loop*. Hanya ada satu solusi untuk setiap *puzzle* pada sebuah *level*.

A. Keping Kosong

Keping kosong merupakan keping yang tidak bergambar. Hanya terdapat satu pola untuk keping ini.

Gambar 2 Keping Kosong

B. Keping Satu Pintu

Keping satu pintu adalah keping bergambar yang harus terhubung ke satu keping bergambar yang lain. Keping ini mempunyai empat pola yaitu keping dengan pintu menghadap ke atas, bawah, kiri, dan kanan.

Gambar 3 Keping satu pintu dan keempat polanya

C. Keping Dua Pintu Lurus

Keping dua pintu lurus adalah keping bergambar yang harus terhubung ke dua keping bergambar lainnya dan gambar dari keping tersebut membentuk garis lurus. Keping ini mempunyai dua pola yaitu pola vertikal dan pola horizontal.

Gambar 4 Keping dua pintu lurus dan kedua polanya

D. Keping Dua Pintu Lengkung

Keping dua pintu lengkunan adalah keping bergambar yang harus terhubung ke dua keping bergambar lainnya dan gambar dari keping tersebut membentuk garis lengkung. Keping ini mempunyai empat pola yang berbeda.

Gambar 5 Keping dua pintu lengkung dan keempat polanya

E. Keping Tiga Pintu

Keping tiga pintu adalah keping bergambar yang harus terhubung ke tiga keping bergambar lainnya. Keping ini mempunyai empat pola yang berbeda.

Gambar 6 Keping tiga pintu dan keempat polanya

F. Keping Empat Pintu

Keping empat pintu adalah keping bergambar yang harus terhubung ke empat keping bergambar lainnya. Keping ini hanya mempunyai satu pola karena sifatnya yang harus terhubung ke empat keping bergambar lainnya walaupun jika keping ini ditekan maka keping ini akan tetap berputar.

Gambar 7 Keping empat pintu

G. Heuristik pada Permainan Infinity Loop

Tujuan dari permainan ini adalah untuk membentuk sebuah pola *infinity loop* pada sebuah *puzzle* yang diberikan. Oleh karena itu, setiap keping bergambar pada *puzzle* harus terhubung dengan keping bergambar lainnya. Maka, tidak mungkin suatu *puzzle* dikatakan terselesaikan jika terdapat keping-keping bergambar di pinggiran *puzzle* yang pintunya menghadap ke luar *puzzle*.

Misalkan, terdapat sebuah keping satu pintu di pinggir kiri tengah *puzzle*. Maka tidak mungkin suatu *puzzle* dikatakan terselesaikan jika keping satu pintu tersebut pintunya menghadap ke kiri karena pasti keping tersebut tidak terhubung ke satu keping bergambar lainnya.

Dengan heuristik ini, kita bisa memastikan bahwa:

1. Tidak mungkin ada keping dua pintu lurus dan keping tiga pintu pada pojokan *puzzle*.
2. Tidak mungkin ada keping empat pintu pada pinggiran *puzzle*.

Heuristik ini membuat pemain bisa menyelesaikan *puzzle* pada sebuah *level* dengan lebih cepat dibandingkan dengan mencari solusi tanpa heuristik.

IV. Pencarian Solusi Dengan Algoritma *Brute Force*

Kita bisa mencari solusi dari sebuah *puzzle* dengan menggunakan algoritma *exhaustive search* yang merupakan bagian dari algoritma *brute force*. Ide dari metode ini adalah mengenumerasi semua kemungkinan solusi lalu menyeleksi satu per satu dari kumpulan kemungkinan solusi yang ada sampai menemukan solusi optimalnya.

Berikut adalah pseudocode untuk mencari solusi dengan algoritma *brute force*:

```
procedure cariSolusiBruteForce(input P1 : Puzzle,
output P2 : Puzzle)
{
  Mencari solusi dari puzzle yang diberikan dengan
algoritma brute force
  Masukan: Puzzle P1 terdefinisi
  Keluaran: Sebuah Puzzle P2 yang merupakan solusi
}

Deklarasi
  SP : SetOfPuzzle
  i : integer

Algoritma
  {Enumerasi solusi}
  SP ← enumerasiSolusi(P1)

  {Mencari solusi yang valid}
  i ← 1
  while (NOT isSolusiValid(SP[i])) do
    i ← i + 1
  P2 ← SP[i]
```

Pseudocode 1 Mencari solusi permainan *Infinity Loop* menggunakan algoritma *brute force*

Misalkan kita akan menyelesaikan *puzzle* pada *level* 12 yang terdapat pada Gambar 1. *Puzzle* tersebut berukuran 4 × 6 dan terdiri dari

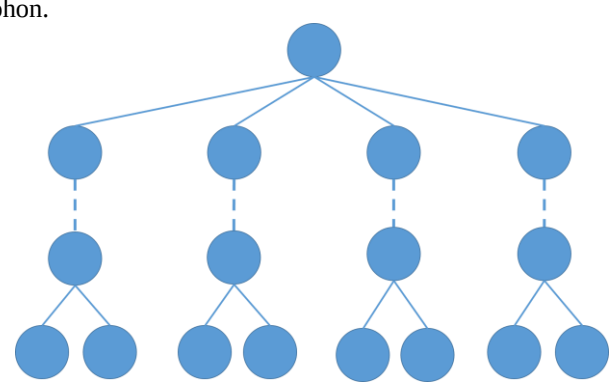
1. Keping kosong : 1 keping
2. Keping satu pintu : 5 keping
3. Keping dua pintu lurus : 4 keping
4. Keping dua pintu lengkung : 7 keping
5. Keping tiga pintu : 5 keping
6. Keping empat pintu : 2 keping

Mari kita hitung jumlah kemungkinan pola yang terbentuk berdasarkan masing-masing jenis keping.

1. Keping kosong : $1^1 = 1$
2. Keping satu pintu : $4^5 = 1024$
3. Keping dua pintu lurus : $2^4 = 16$
4. Keping dua pintu lengkung : $4^7 = 16384$
5. Keping tiga pintu : $4^5 = 1024$
6. Keping empat pintu : $1^2 = 1$

Maka, jumlah solusi yang dienumerasi adalah perkalian dari jumlah kemungkinan pola yaitu $1 \times 1024 \times 16 \times 16384 \times 1024 \times 1 = 2{,}748779069 \times 10^{11}$ kemungkinan solusi. Pasti hanya ada satu solusi valid dari sekian banyak solusi yang dikumpulkan. Pencarian solusi dengan algoritma *brute force* sangat memakan sumberdaya dan waktu yang tidak sedikit.

Berikut adalah visualisasi algoritma *brute force* dengan pohon.



Gambar 8 Visualisasi algoritma *brute force* menggunakan pohon

Terlihat dari Gambar 8 di atas bahwa kita harus menciptakan semua simpul terlebih dahulu lalu mencari satu simpul yang merupakan solusi valid.

V. Pencarian Solusi Dengan Algoritma Runut-balik

Jika kita menggunakan algoritma *brute force* untuk menyelesaikan *puzzle* maka akan sangat membutuhkan sumberdaya dan waktu yang tidak sedikit untuk menyelesaikannya. Apalagi untuk kasus *puzzle* ini hanya terdapat satu solusi valid dari sekian banyak calon solusi valid yang ada.

Algoritma runut-balik (*backtracking*) adalah perbaikan dari algoritma brute force yang berbasis pada pencarian mendalam (*Depth First Search*) untuk mencari solusi persoalan secara lebih mangkus. Di sini, kita gunakan pendekatan rekursif untuk menemukan solusi valid dari *puzzle* yang diberikan.

Berikut adalah *pseudocode* untuk mencari solusi valid dengan algoritma runut-balik:

```
procedure cariSolusiRunutBalik(input P1 : Puzzle,
input koordinat : Point, output P2 : Puzzle)
{
  Mencari solusi dari puzzle yang diberikan dengan
algoritma runut-balik
  Masukan: Puzzle P1 terdefinisi dan Point koordinat
terdefinisi yang menyatakan posisi keping acuan
  Keluaran: Sebuah Puzzle P2 yang merupakan solusi
}

Deklarasi
  SP : SetOfPuzzle
  i : integer
  koordinat2 : Point

Algoritma
  if (isSolusiValid(P1)) then
    {BASIS 1}
    P2 ← P1
  else
    SP ← isiSP(P1,koordinat)
    koordinat2 ← nextKoordinat(koordinat)
    if (NOT isKoordinatValid(koordinat2)) then
      {BASIS 0}
      P2 ← NULL
    else
      {REKURENS}
      i ← 1
      while P2 = NULL AND i <= count(SP) do
        cariSolusiRunutBalik(SP[i], koordinat2, P2)
        i ← i + 1
```

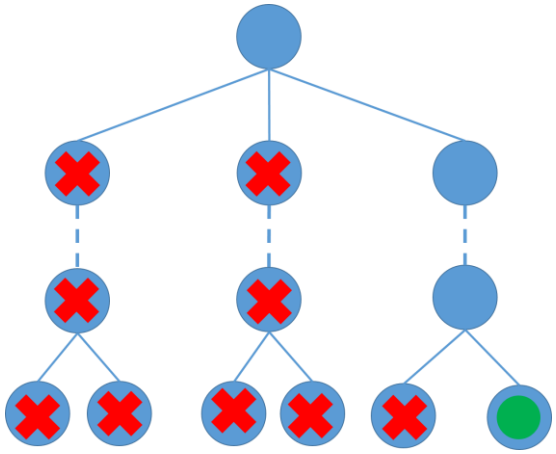
Pseudocode 2 Mencari solusi permainan *Infinity Loop* menggunakan algoritma runut-balik

Misalkan kita akan menyelesaikan kembali *puzzle* pada *level* 12 yang terdapat pada Gambar 1. Kita akan menjalankan prosedur cariSolusiRunutBalik dengan masukan soal *puzzle* pada *level* 12 dan koordinat (1,1) yang merupakan koordinat keping di pojok kiri atas.

Tentu *puzzle* pertama ini bukan solusi. Karena koordinat *puzzle* pertama ini valid maka kita akan mengisi SetOfPuzzle SP dengan kemungkinan solusi sebesar jumlah pola keping pada koordinat (1,1) yang mungkin. Karena keping pada koordinat (1,1) jenisnya adalah keping satu pintu maka terdapat empat kemungkinan solusi. Lalu dari masing-masing kemungkinan solusi akan dimasukkan ke dalam prosedur cariSolusiRunutBalik dengan koordinat selanjutnya, yaitu (1,2).

Jika kita mencapai koordinat terakhir (4,6) dan Puzzle P1 bukan merupakan solusi maka prosedur akan mengisi nilai Puzzle P2 dengan NULL lalu kembali ke prosedur sebelumnya dan menjalankan prosedur cariSolusiRunutBalik dengan kemungkinan solusi selanjutnya. Jika suatu Puzzle P1 adalah solusi yang valid maka prosedur akan mengisi Puzzle P2 dengan Puzzle P1. Pasti hanya ada satu solusi yang valid pada setiap *puzzle* yang diberikan.

Berikut adalah visualisasi algoritma runut-balik dengan pohon.



Gambar 9 Visualisasi algoritma runut-balik menggunakan pohon

Terlihat dari Gambar 9 bahwa kita tidak perlu membuat semua simpul untuk menemukan solusi yang valid. Walaupun ada kemungkinan bahwa solusi yang valid berada di simpul yang terakhir dihidupkan tetapi dengan algoritma runut balik ini kita bisa meminimalisir penciptaan simpul. Jika pada

algoritma *brute force* kita harus menciptakan semua simpul lalu mencari simpul yang merupakan solusi yang valid maka pada algoritma runut-balik kita menciptakan sebuah simpul dan langsung memeriksa kevalidannya dan jika tidak valid maka kita akan menghidupkan anak-anak dari simpul tersebut.

VI. Tambahan Fungsi Heuristik Pada Algoritma Runut-balik

Sebelumnya kita sudah membahas tentang pencarian solusi dengan algoritma *brute force* dan algoritma runut-balik. Pada bab ini, kita coba untuk menggunakan heuristik-heuristik yang ada untuk meminimalkan sumberdaya dan waktu yang dipakai. Kita coba sisipkan heuristik-heuristik tadi ke algoritma runut-balik.

Berikut adalah *pseudocode* untuk mencari solusi valid menggunakan algoritma runut balik yang telah disisipkan heuristik-heuristik yang ada.

```
procedure cariSolusiRunutBalik(input P1 : Puzzle,
input koordinat : Point,  output P2 : Puzzle)
{
  Mencari solusi dari puzzle yang diberikan dengan
algoritma runut-balik
  Masukan: Puzzle P1 terdefinisi dan Point koordinat
terdefinisi yang menyatakan posisi keping acuan
  Keluaran: Sebuah Puzzle P2 yang merupakan solusi
}

Deklarasi
  SP : SetOfPuzzle
  i : integer
  koordinat2 : Point

Algoritma
  if (isSolusiValid(P1)) then
    {BASIS 1}
    P2 ← P1
  else
    SP ← isiSP(P1,koordinat)
    koordinat2 ← nextKoordinat(koordinat)
    if (NOT isKoordinatValid(koordinat2)) then
      {BASIS 0}
      P2 ← NULL
    else
      {REKURENS}
      i ← 1
      while P2 = NULL AND i <= count(SP) do
        if    (isMemenuhiHeuristik(SP[i]),koordinat2)
then
          cariSolusiRunutBalik(SP[i],koordinat2,P2)
        i ← i + 1
```

Pseudocode 3 Mencari solusi permainan *Infinity Loop* menggunakan algoritma runut-balik dengan tambahan syarat heuristik

Fungsi isMemenuhiHeuristik memeriksa apakah keping pada koordinat sebesar Point koordinat2 pada Puzzle SP[i] memenuhi heuristik yang ada.

Mari kita tinjau keping pada koordinat (1,1) yang berada di pojok kiri atas *puzzle* pada Gambar 1. Jika kita tidak menggunakan heuristik-heuristik yang ada maka kita akan memeriksa keempat kemungkinan pola yang ada. Namun, dengan menggunakan heuristik kita hanya perlu memeriksa puzzle yang keping di koorfinat (1,1) menghadap ke arah bawah dan kanan. Kelihatannya kita hanya menghemat dua peciptaan simpul dari yang seharusnya terjadi empat penciptaan simpul. Namun, hal tersebut sangat berpengaruh pada penggunaan sumberdaya dan waktu yang ada karena itu berarti kita tidak perlu menciptakan anak-anak dari simpul yang sudah pasti tidak akan mencapai solusi yang valid. Dengan begitu, jelas bahwa tambahan fungsi heuristik pada algoritma runut-balik sangat meminimalkan penciptaan simpul sehingga menghemat penggunaan sumberdaya dan waktu yang digunakan.

VII. Kesimpulan dan Saran

Jika algoritma *brute force* dibandingkan dengan algoritma runut-balik maka jelas bahwa algoritma runut-balik lebih unggul dari segi penggunaan sumberdaya dan waktu daripada algoritma algoritma *brute force*. Algoritma runut-balik hanya menghidupkan simpul-simpul solusi yang diperlukan dan langsung memeriksa kevalidannya sedangkan bila kita menggunakan algoritma *brute force* maka kita harus menghidupkan semua simpul solusi lalu mencari simpul solusi yang valid.

Heuristik-heuristik yang berlaku sangat membantu mempercepat pencarian solusi yang valid. Sumberdaya dan waktu yang diperlukan dalam pencarian solusi yang valid jauh lebih kurang daripada kita menggunakan pencarian solusi yang valid secara naif.

Permainan *Infinity Loop* merupakan salah satu permainan yang terkenal di tahun 2016 dan tidak menutup kemungkinan bahwa akan diadakan suatu kejuaraan untuk menyelesaikan *puzzle-puzzle* pada permainan ini. Untuk itu, dibutuhkan pengetahuan tentang algoritma-algoritma yang baik untuk memenemukan solusi yang valid pada suatu *puzzle*. Pada makalah ini belum dibahas tentang penyelesaian *puzzle-puzzle* pada kasus-kasus khusus yang jika menggunakan suatu algoritma khusus maka pemain akan mendapatkan solusi lebih cepat daripada mengunakan algoritma biasa. Makalah ini bisa dikembangkan dan mengarah ke sana, yaitu terciptanya suatu algoritma-algoritma yang baik untuk berbagai macam kasus *puzzle*.

VIII. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Allah SWT atas karunia dan hidayah-Nya dalam proses pembuatan makalah ini. Penulis juga mengucapkan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T. dan Ibu Dr. Nur Ulfa Maulidevi, S.T., M.T. yang telah membimbing penulis dalam kuliah IF2211 Strategi Algoritma selama ini.

REFERENSI

[1] Munir, Rinaldi. “Diktat Kuliah IF2211 Strategi Algoritma,” Program Studi Teknik Informatika Institut Teknologi Bandung, 2009, Bandung, Indonesia.

[2] http://loopgamecheats.com/?tag=12 Tanggal akses: 7 Mei 2016

[3] Matuszek, David. “Backtracking, ” https://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html Tanggal akses: 8 Mei 2016

[4] Black, Andrew P.. “Exhaustive Search Algorithm, ” http://web.cecs.pdx.edu/~black/cs350/Lectures/lec06-Exhaustive%20Search.pdf Tanggal akses: 8 Mei 2016.

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, 9 Mei 2016

Muhammad Kamal Nadjieb - 13514054