Pencarian Jalan untuk Menghasilkan Skor Tertinggi pada Permainan "Voracity"

Okaswara Perkasa (13510051)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
okaswara@students.itb.ac.id

***Abstrak* — Makalah ini membahas mengenai algoritma-algoritma yang dapat digunakan untuk mencari jalan yang menghasilkan skor tertinggi pada permainan Voracity. Algoritma-algoritma ini menggunakan teknik DFS dan dynamic programming. Dasar teori mengenai permainan Voracity, serta DFS dan dynamic programming dijabarkan terlebih dahulu. Setelah itu terdapat penjabaran ide dasar serta pseudocode kedua algoritma. Terakhir, terdapat hasil uji coba dari kedua algoritma beserta analisis dari hasil yang didapatkan.**

***Kata kunci* — voracity, depth-first search, dynamic programming**

I. PENDAHULUAN

Dewasa ini, utilitas komputer sebagai alat bantu bagi manusia sudah tidak dipertanyakan lagi. Dari berbagai bidang ilmu, baik teoritis maupun terapan, memakai bantuan komputer untuk melakukan perhitungan dalam jumlah yang besar dengan cepat. Pemahaman mengenai desain algoritma yang baik menjadi sangat penting dalam pembuatan algoritma yang efisien. Salah satu cara untuk lebih memahami mengenai algoritma adalah dengan mencoba menyelesaikan persoalan-persoalan pada *video games.*

Walaupun sekilas terlihat sepele, namun sejumlah *video games* yang sederhana ternyata memiliki hubungan yang dalam dan erat dengan teori komputasi, bahkan yang masih belum terpecahkan sampai saat ini. Salah satu contohnya adalah permainan *Minesweeper*[6].

Oleh karena itu, makalah ini dibuat bertujuan untuk lebih memahami mengenai perancangan algoritma dan sifat-sifat komputasi. Waluapun cenderung tidak praktikal, pemahaman yang diberikan dapat dijadikan pondasi pada saat merancang algoritma lain nantinya.

II. DASAR TEORI

A. Voracity

Voracity adalah salah satu permainan yang terdapat dalam paket permainan *24 Games for Windows*. Paket permainan ini sendiri dibuat pada tahun 1995, oleh *Expert Software*[1]. Sesuai nama serta tahun pembuatannya, 24 Games ditujukan untuk sistem operasi *Microsoft Windows 95*.

Gambar 1 – Tampilan permainan *Voracity*

Permainan *Voracity* sebenarnya cukup sederhana. Secara garis besar, pemain menggerakan sebuah target[2] (pada Gambar 1 yang berupa lingkaran berwarna coklat-kuning) pada suatu tabel, yang masing-masing selnya memiliki suatu nilai.

Pergerakan target harus mengikuti suatu aturan: target harus bergerak pada arah tertentu sejauh nilai pada sel yang *adjacent* pada arah yang dituju.

Misal, jika target digerakkan ke bawah, maka target harus bergerak sebesar 4; nilai yang berada di bagian bawah target saat itu. Setelah itu, tabel akan menjadi seperti pada Gambar 2.

Gambar 2 – Contoh pergerakan target

Perhatikan pada title bahwa skor pemain sudah menjadi empat, yang sebelumnya nol. Penambahan skor seusai dengan jarak yang ditempuh oleh target.

Aturan lainnya, target tidak dapat digerakkan jika pergerakan tersebut akan membuat target keluar dari board, ataupun jika pergerakan akan melewati jalan yang sebelumnya sudah ditempuh.

Contoh, pada Gambar 3, target tidak bisa digerakkan ke kiri karena jika digerakkan akan melewati jalan yang pernah dilalui.

Gambar 3 – Contoh pergerakan target

Permainan berakhir ketika target sudah tidak bisa digerakkan (Gambar 4, perhatikan sel-sel yang *adjacent* dengan target). Jika skor total mencukupi, maka skor akan ditambahkan pada *high score*.

Gambar 4 – Akhir dari permainan

Terdapat ukuran tabel yang berbeda-beda yang disediakan pada permainan. Untuk setiap ukuran, jarak dari nilai acak yang berada pada setiap sel juga berbeda. Berikut daftar pasangan ukuran dan jarak dari nilai acak yang sudah didefinisikan dalam permainan:

1. 15x15, jarak nilai 1 s.d. 5
2. 19x19, jarak nilai 1 s.d. 7
3. 22x22, jarak nilai 1 s.d. 8
4. 25x25, jarak nilai 1 s.d. 9

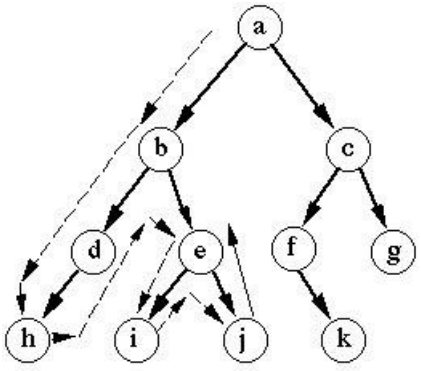
Sebagai tambahan, terdapat satu pasangan lagi, yang berada diluar standar permainan:

5. 10x10, jarak nilai 1 s.d. 4

B. Depth-First Search (DFS)[3]

DFS adalah salah satu bentuk pencarian pada pohon, atau permasalahan yang dimodelkan dalam pohon, dimana langkah pencarian dilakukan dengan memilih salah satu simpul yang mungkin yang berhubungan langsung dengan simpul saat ini.

Jika pada suatu saat tidak ada lagi simpul yang dapat dipilih, maka akan dilakukan *backtracking*, yakni membatalkan pilihan simpul yang paling terakhir, dan melanjutkan pencarian ke simpul lainnya.



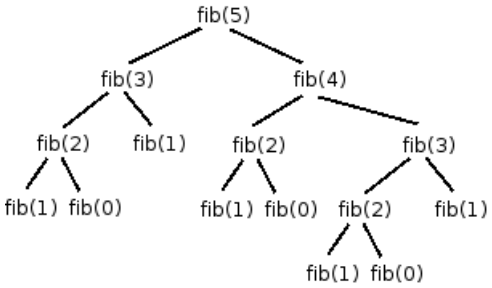
Gambar 5 – Urutan pencarian pada DFS[4]

C. Dynamic Programming[3]

Dynamic programming adalah salah satu jenis algoritma yang mencari dan menyimpan solusi dari *overlapping subproblems*. *Overlapping subproblems* sendiri adalah sub-permasalahan yang muncul beberapa kali ketika melakukan penyelesaian masalah.

Dengan menggunakan *dynamic programming*, *subproblem-subproblem* ini cukup diselesaikan sekali saja, dan disimpan hasilnya. Jika salah satu *subproblem* muncul kembali, komputasi ulang tidak perlu dilakukan, dan solusi untuk *subproblem* ini dapat langsung diambil dari hasil yang telah didapat sebelumnya.

Salah satu contoh permasalahan klasik yang dapat diselesaikan dengan dynamic programming adalah menghitung deret *fibonacci*. Jika diselesaikan dengan tidak menyimpan nilai deret yang telah didapatkan sebelumnya, kompleksitas dapat tumbuh secara eksponensial, karena sejumlah *subproblem* diselesaikan beberapa kali[5]. Dengan menggunakan *dynamic programming*, nilai ke-n dan n+1 dari deret fibonacci akan disimpan, dan dapat digunakan ulang untuk menghitung nilai ke-n+2.



Gambar 6 – Pohon *subproblem* pada perhitungan deret fibonacci[5]

III. DESKRIPSI ALGORITMA PENYELESAIAN

Sesuai dengan yang telah disebutkan, algoritma penyelesaian yang dipakai adalah mencari urutan langkah yang membuat skor yang didapatkan maksimum, pada suatu tabel tertentu. Algoritma yang dibuat diharapkan tidak hanya memberikan solusi yang paling optimal, namun juga efisien.

Pada bab ini akan dijabarkan dua algoritma dengan teknik yang berbeda, yakni dengan DFS dan *dynamic programming*. Masing-masing akan dijabarkan ide dasar serta pseudocode dari algoritma tersebut.

A.Penyelesaian dengan DFS

Pada dasarnya, penyelesaian DFS cukup *straightforward*, seperti pada permainan *Voracity* itu sendiri. Target digerakkan ke sejumlah urutan arah, sampai tidak dapat digerakkan lagi.

Namun perbedaannya, algoritma DFS akan melakukan *backtracking*, sehingga dapat melakukan langkah "mundur". Melangkah mundur ini dapat dimanfaatkan untuk mencoba seluruh kemungkinan gerakan yang mungkin dilakukan.

Nantinya setiap kali target tidak dapat bergerak lagi, nilai skor saat itu akan dibandingkan dengan nilai skor maksimum yang ada. Jika nilai skor yang baru lebih besar, nilai skor maksimum akan diganti dengan yang baru, begitu juga urutan langkah optimal saat itu.

Untuk menandakan sel yang pernah dilewati, dapat digunakan suatu tabel yang menyimpan informasi apakah suatu sel pernah dilalui sebelumnya. Sel akan di-*disable* jika hal tersebut sudah terjadi.

Berikut *pseudocode* penyelesaian dengan algoritma DFS:

```
1. Untuk setiap arah yang dapat dituju oleh target
2.     Gerakkan target ke arah tersebut
3.     Disable sel-sel yang dilalui target pada tabel
4.     Lakukan secara rekursif langkah 1, dengan
5.     memberikan data skor dan urutan gerakan,
       setelah dilakukan langkah 2
6.     Gerakkan target ke tempat semula (backtrack)
7.     Enable sel-sel yang dilalui target pada tabel
8. Jika tidak ada arah yang dapat dituju dan skor > skor
   maksimal
9.     Jadikan skor dan urutan langkah maksimal
       menjadi skor dan urutan langkah saat ini.
```

Kompleksitas waktu untuk algoritma ini dapat dengan mudah dihitung secara kasar.

Untuk setiap sel, target dapat bergerak dalam empat arah. Misalkan ukuran tabel adalah NxN, maka pada kasus terburuk setiap sel akan dilewati pada seluruh urutan langkah yang mungkin. Oleh karena itu, kompleksitas waktunya adalah $T(n) = 4^{n^2}$.

B. Penyelesaian dengan Dynamic Programming

Jika dilihat dari aturan permainan pada *Voracity*, secara intuitif dapat diperkirakan bahwa permainan ini dapat didefinisikan sebagai sejumlah *subproblem*. Perkiraan ini diperkuat dengan fakta bahwa suatu sel dapat dikunjugi beberapa kali dengan jalan yang berbeda-beda. Jika benar demikian, pencarian langkah dapat juga diselesaikan dengan *dynamic programming*.

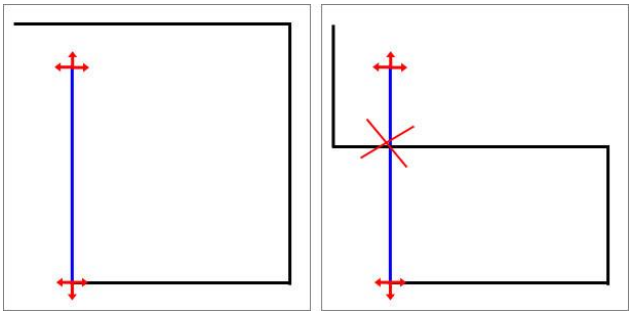
Namun sebelumnya harus terdapat pendefinisan yang tepat mengenai domain *subproblem* yang digunakan. *Subproblem-subproblem* ini secara kasar dapat dibedakan dari "*state* pada tabel", serta posisi target saat ini. Solusi yang disimpan untuk setiap *subproblem* tentu saja skor maksimal dan urutan langkah untuk mencapai nilai maksimal tersebut.

Posisi target, skor maksimal dan urutan langkah mudah untuk didefinisikan. Namun tidak untuk "*state* pada tabel". Parameter ini tidak dapat langsung terdefinisi dengan jelas. Cukup sulit untuk menentukan apa yang harus didefinisikan untuk menjamin bahwa solusi yang berkoresponden memang sudah solusi yang sesuai.

Salah satu cara pendefinisian "state pada tabel" adalah arah-arah yang tidak dapat dilalui pada sekumpulan koordinat yang menjadi jalan pada solusi, lebih tepatnya koordinat-koordinat dimana target berhenti untuk memilih arah.

Hal ini ditunjukkan pada Gambar 7 sebelah kiri. Jalan yang berwarna hitam menunjukkan jalan yang sudah ditempuh. Sedangkan yang berwarna biru merupakan solusi dari *subproblem*, yang menunjukkan jalan untuk mencapai skor yang optimal. Tanda panah merah pada sejumlah koordinat menunjukkan arah yang tidak dapat dituju oleh target pada koordinat tersebut.

Perlu diperhatikan sebelumnya bahwa arah yang digunakan untuk *backtrack* pada suatu koordinat tidak perlu diperhitungkan (e.g. koordinat dengan tanda panah yang ada di sebelah atas; arah ke bawah tidak diperhitungkan).

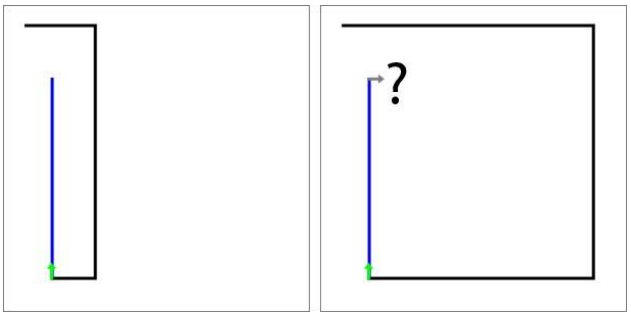


Gambar 7 – Pendefinisian domain *subproblem* (yang kurang tepat)

Namun informasi ini ternyata belum cukup. Pada Gambar 7 di sebelah kanan, bisa ditunjukkan bahwa terdapat jalan lain yang memenuhi kondisi dari *subproblem*, padahal jalan tersebut *overlap* dengan jalan pada solusi optimal. Oleh karena itu, terbukti bahwa menyimpan informasi arah-yang-tidak-dapat-dituju untuk sejumlah koordinat belum dapat memilah *subproblem* dengan tepat.

Cara lain adalah dengan mendefinisikannya dengan arah yang dapat dituju oleh sejumlah koordinat (dimana target berhenti) yang menjadi jalan optimal. Gambar 8 sebelah kiri menunjukkan hal tersebut.

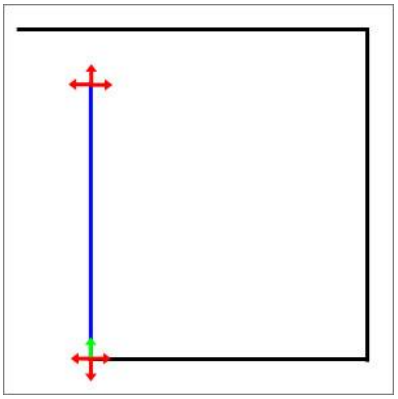
Pada Gambar 8, tanda panah hijau berarti arah-yang-dapat-ditempuh oleh target pada koordinat tersebut. Terminologi untuk segmen garis yang lain masih sama seperti sebelumnya.



Gambar 8 – Pendefinisian domain *subproblem* (yang masih kurang tepat)

Sayangnya, informasi ini juga tidak mencukupi. Hampir sama dengan sebelumnya, terdapat suatu jalan lain yang memenuhi *subproblem*, namun jalan yang ditempuh memiliki kemungkinan untuk membuka jalan baru, yang dapat dilalui untuk mencapai solusi yang lebih optimal. Gambar 8 sebelah kanan menunjukkan hal tersebut.

Ternyata, untuk menjamin pemilahan *subproblem* dengan tepat, diperlukan kedua informasi yang telah dijabarkan sebelumnya, yakni arah-yang-tidak-dapat-dituju serta arah-yang-dapat-ditempuh, pada masing-masing koordinat saat target berhenti untuk memilih arah. Dengan menggunakan informasi ini, setiap *subproblem*, jika ada, akan dipilih dengan tepat, tanpa ada kemungkinan jalan yang overlap ataupun adanya jalan yang belum ditelusuri.



Gambar 9 – Pendefinisian domain *subproblem*

Oleh karena itu, domain *subproblem* didefinisikan sebagai koordinat target, dan informasi arah (baik dapat dituju maupun tidak) untuk setiap koordinat menuju jalan yang optimal. Solusi yang disimpan adalah skor terbesar beserta urutan arah yang harus dilalui.

Secara garis besar, berikut *pseudocode* dari algoritma *dynamic programming* ini:

```
1.  Cari subproblem yang sesuai dengan koordinat target dan state board saat ini
2.  Jika ditemukan, kembalikan solusi dari subproblem tersebut
3.  Selain itu, jika tidak ditemukan
4.      Untuk setiap arah yang mungkin
5.          Gerakkan target ke arah tersebut
6.          Cari solusi dengan kembali ke langkah 1 secara rekursif
7.          Jadikan solusi maksimum jika skornya lebih besar
8.          Lakukan backtrack pada target
9.      Jika tidak ada arah yang mungkin
10.         Tambahkan subproblem dasar, hanya terdiri dari satu koordinat, yakni koordinat target, yang tidak dapat digerakkan kemanapun
11.     Selain itu,
12.         Tambah subproblem menggunakan koordinat target dan state board saat ini, serta solusi optimal yang didapatkan
```

Sekilas algoritma yang digunakan hampir sama dengan DFS. Namun perbedannya terdapat pencarian *subproblem* yang sudah pernah diselesaikan, serta penambahan solusi untuk *subproblem* yang baru.

Kompleksitas algoritma untuk algoritma ini cukup sulit untuk dihitung secara eksak. Ini dikarenakan adanya ketidakpastian pada jumlah *subproblem*. Kemungkinan besar algortima bersifat eksponensial, karena solusi dari setiap *subproblem* masih tumbuh secara eksponensial.

Walaupun demikian, diharapkan algoritma ini menghasilkan banyak *subproblem* yang overlap, sehingga waktu pencarian solusi dapat lebih dipersingkat.

IV. HASIL UJI COBA DAN ANALISIS

Bab ini akan membahas mengenai hasil uji coba yang dilakukan pada kedua algoritma. Program uji coba dibuat dengan menggunakan C++. Hasil yang diuji merupakan waktu pencarian solusi optimal, diasumsikan solusi yang didapatkan sudah optimal. Hasil uji akan dilengkapi dengan analisis.

Sampel yang digunakan pada pengujian adalah tabel acak yang ukuran serta jarak nilainya sesuai dengan yang sudah dituliskan pada dasar teori.

A. Algoritma DFS

Berikut durasi algoritma DFS, untuk setiap ukuran tabel:

Ukuran	Jarak nilai	Durasi
10x10	1 s.d. 3	12 detik
15x15	1 s.d. 5	23 detik
19x19	1 s.d. 7	16 detik
22x22	1 s.d. 8	21 menit
25x25	1 s.d. 9	> 3 jam

Tabel 1 – Durasi algoritma DFS

Durasi pengerjaan dari hasil uji sesuai dengan sifat algoritma eksponensial pada umumnya; waktu pengerjaan bertambah secara signifikan ketika ukuran masukan ditambah.

Namun terdapat anomali pada board berukuran 19x19, waktu pengerjaan lebih singkat daripada board yang berukuran lebih kecil.

Fenomena ini dapat dijelaskan dengan ikut memperhitungkan jarak nilai maksimum pada penentuan kompleksitas waktu. Sebenarnya jarak nilai maksimum mengatur seberapa jauh target dapat bergerak. Jika pergerakan lebih jauh, maka pohon pencarian yang dihasilkan menjadi lebih rendah. Jika diperkirakan melalui rasio ukuran-jarak, rasio pada tabel 15x15 (15/5 = 3) lebih besar daripada rasio pada tabel 19x19 (19/7 = 2.7).

Walaupun algoritma dapat digunakan untuk mencari langkah optimum, namun masih kurang efisien untuk ukuran board yang lebih besar.

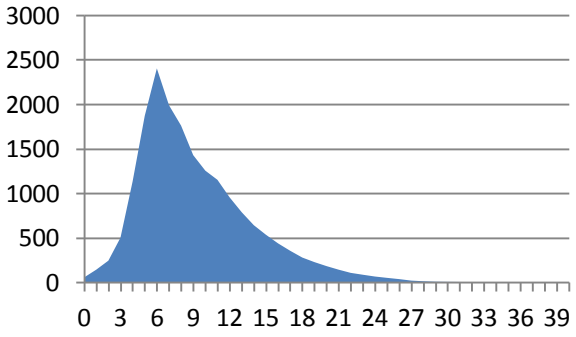
B. Algoritma Dynamic Programming

Tidak sesuai ekspektasi awal, algoritma *dynamic programming* yang diimplementasikan memiliki performa yang jauh lebih buruk dibandingkan DFS.

Untuk kasus dengan ukuran tabel terkecil, 10x10, algoritma tidak mampu melakukannya dengan efisien. Waktunya jauh lebih lambat, hingga melebihi 30 menit. Pada saat itu, jumlah memori yang digunakan juga sudah membengkak, menjadi sekitar 100 MB. Maka kemungkinan besar pencarian solusi tidak dapat dilanjutkan karena nantinya komputer akan kehabisan memori.

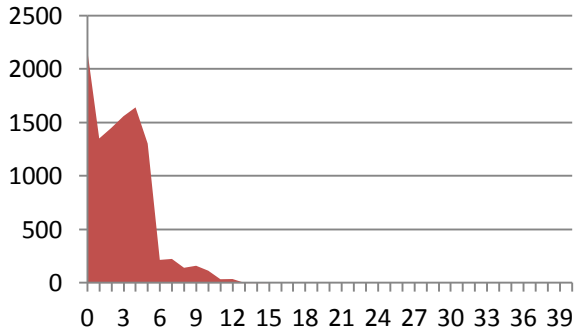
Untuk mengatasinya, ukuran tabel diperkecil sedikit, menjadi 9x9 dengan jarak nilai 1 s.d. 3. Setelah ukuran masukan diperkecil, algoritma dapat menemukan solusi paling optimal dalam waktu 8.3 detik.

Untuk lebih memahami apa yang sebenarnya terjadi, berikut distribusi jumlah *subproblem* yang dibuat, pada sumbu x dibagi berdasarkan jumlah langkah optimal dari solusi *subproblem* tersebut:



Gambar 10 – Distribusi subproblem yang dibuat

Berikut distribusi berapa kali suatu *subproblem* dipakai ulang solusinya, pada sumbu x dibagi berdasarkan jumlah langkah optimal pada solusi *subproblem* tersebut:



Gambar 11 – Distribusi subproblem yang dipakai

Jika kedua distribusi dibandingkan, dapat disimpulkan bahwa pemakaian *subproblem* masih tidak optimal. *Subproblem* yang benar-benar digunakan mayoritas hanya yang memiliki jumlah langkah yang rendah. Padahal jumlah *subproblem* yang menyimpan langkah yang jauh cukup banyak. Karena itu, dapat disimpulkan bahwa algoritma masih cukup sering melakukan *deepening*.

Selain itu, utilitas *subproblem* dengan langkah-jauh tidak besar, sedangkan jumlahnya lebih banyak daripada langkah-dekat. Ini menyebabkan algoritma pencarian linear *subproblem* yang digunakan menjadi tidak efisien, karena algoritma cenderung membutuhkan waktu yang lebih lama untuk mendapatkan suatu *subproblem* yang sesuai. Waktu yang diperlukan cenderung mendekati waktu terburuk.

KESIMPULAN

Setelah diujicobakan, pada kasus ini, algoritma DFS memiliki efisiensi yang lebih baik dibandingkan *dynamic programming*.

Penyelesaian dengan teknik *dynamic programming* ternyata tidak efisien jika jumlah *subproblem* tumbuh secara eksponensial, walaupun cukup banyak *subproblem* yang *overlap*.

Performa algoritma *dynamic programming* mungkin dapat ditingkatkan dengan menggunakan cara pencarian yang lebih baik (e.g. *hashing*). Definisi ulang pada domain *subproblem* mungkin juga dapat dilakukan, sehingga dapat meningkatkan utilitas setiap solusi pada *subproblem* secara keseluruhan, terutama *subproblem* dengan langkah yang lebih jauh.

REFERENSI

[1] 24 Games for Windows Help – About, 20 Desember 2012

[2] Voracity Help Contents, 20 Desember 2012

[3] Introduction to The Design and Analysis of Algorithm, 3rd Edition, 21 Desember 2012

[4] http://www.cse.unsw.edu.au/~billw/Justsearch.html, 21 Desember 2012

[5] http://goose.ycp.edu/~dhovemey/fall2011/cs201/notes/dynamicProgramming.html, 21 Desember 2012

[6] http://www.claymath.org/Popular_Lectures/Minesweeper, 21 Desember 2012

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, 21 Desember 2012

Okaswara Perkasa (13510051)