MENGESTIMASI KESANGKILAN DARI ALGORITMA *BACKTRACKING*

Indah Kuntum Khairina

Laboratorium Ilmu dan Rekayasa Komputasi
Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha no. 10, Bandung
E-mail: if15088@students.if.itb.ac.id

ABSTRAK

Ketika menggunakan algoritma backtracking untuk memecahkan masalah-masalah kombinatorial, salah satu kesulitan yang selalu ditemukan adalah ketidakmampuan kita untuk memprediksi kesangkilan dari algoritma tersebut, atau membandingkan kesangkilan dari algoritma-algoritma dengan pendekatan berbeda, tanpa menjalankan program dengan algoritma backtracking tersebut. Terkadang, sebuah program backtrack dapat berjalan kurang dari satu detik, namun kemudian pada percobaan kasus lainnya, program backtrack tersebut berjalan sangat lama, seolah-olah tidak akan pernah ditemukan solusi yang ingin dicapai. Ketimpangan yang mencolok dalam waktu eksekusi inilah yang merupakan karakteristik dari program yang menggunakan algoritma backtrack. Tidaklah jelas apa yang akan terjadi pada percobaan pemecahan suatu kasus, sebelum algoritma tersebut diimplementasi dan kemudian dijalankan. Makalah ini akan mencoba menjelaskan kembali (dari penjelasan yang penulis dapat dari makalah-makalah referensi) mengenai sebuah metode sederhana yang akan menghasilkan suatu pendekatan/estimasi yang masuk akal untuk kebanyakan program yang menggunakan algoritma backtrack, dan hanya membutuhkan sejumlah kalkulasi sederhana.

Kata kunci: backtrack, sudoku, estimasi, sangkil, efisien.

1. PENDAHULUAN

Contoh persoalan yang akan ditinjau di dalam makalah ini adalah persoalan Sudoku. Untuk membatasi pembahasan agar tidak terlalu luas, akan dipakai *puzzle* dengan ukuran 4x4, sehingga di sini dapat didefinisikan bahwa Sudoku merupakan permainan penempatan angka pada sebuah puzzle yang direpresentasikan sebagai sebuah matriks 4x4. Artinya, puzzle ini terdiri dari 4 baris (horisontal), 4 kolom (vertikal), dan 4 kotak (yaitu, upa-puzzle yang direpresentasikan sebagai matriks 2x2).

Baris-baris dinomori dengan nomor 1 sampai 4 (dari atas ke bawah), begitu juga dengan kolom-kolomnya (dinomori dari kanan ke kiri). Sebuah sel didefinisikan dengan cara menyebutkan nomor baris dan kolomnya, misalnya sel b1k1 untuk sel yang terletak pada baris 1 dan kolom 1. Total terdapat 16 sel. Selanjutnya, sel-sel dalam puzzle ini diisi dengan angka-angka 1 sampai 4, yang memiliki kendala, yaitu: hanya boleh terdapat tepat satu angka untuk setiap nilai pada baris, kolom, dan kotak yang sama.

Biasanya, sejumlah sel telah diisi dengan angka-angka yang konstan (tidak dapat diubah oleh pemain), angka-angka ini juga biasa disebut sebagai himpunan angka petunjuk. Tujuan pemain adalah untuk menemukan angka-angka untuk dimasukkan ke dalam sel-sel kosong yang tersisa, dengan syarat angka-angka tersebut memenuhi kendala, sampai seluruh sel terisi lengkap (dengan himpunan angka petunjuk sebagai upa-himpunan angka-angka solusi). Contoh puzzle Sudoku dengan himpunan angka petunjuk dapat dilihat pada gambar 1, sementara contoh puzzle Sudoku yang telah terisi lengkap dan valid dapat dilihat pada gambar 2.

Pada makalah ini, penulis hanya akan membahas contoh persoalan Sudoku ini sebatas kemungkinan-kemungkinan cara menempatkan angka-angka pada baris pertama, sebab pembahasan akan lebih ditekankan pada perhitungan nilai estimasi kesangkilan algoritma *backtracking*, bukan pada pengimplementasian algoritma tersebut dalam permainan Sudoku.

1			**4**
	3		
	1		

Gambar 1. Puzzle Sudoku dengan himpunan angka petunjuk

1	**2**	**3**	**4**
4	**3**	**1**	**2**
2	**1**	**4**	**3**
3	**4**	**2**	**1**

Gambar 2. Puzzle Sudoku yang telah terisi lengkap dan valid

2. CONTOH PERSOALAN

2.1 Deskripsi persoalan

Terdapat sebuah puzzle Sudoku 4x4 dengan himpunan angka petunjuk yang diberikan (dapat dilihat pada gambar 3). Susunlah urutan angka-angka pada baris pertama sedemikian sehingga angka-angka tersebut memenuhi kendala, yaitu: hanya boleh terdapat tepat satu angka untuk setiap nilai pada baris, kolom, dan kotak yang sama.

4	**3**		**2**
	1		**3**
3		**2**	**1**

Gambar 3. Puzzle Sudoku pada contoh persoalan

2.2 Menghitung Kemungkinan Susunan

Asumsikan bahwa penempatan angka dimulai dari sel paling kiri menuju sel paling kanan. Setiap sel memiliki 4 kemungkinan angka, sehingga satu baris memiliki kemungkinan sebanyak 4x4x4x4 = 256. Cara ini adalah menggunakan skema *brute force* yang benar-benar murni, yaitu benar-benar mengenumerasi semua kemungkinan susunan, kemudian baru memilih susunan yang layak.

2.2.1 Perbaikan Pertama

Jika kita mempertimbangkan keunikan angka-angka yang dapat dimasukkan dalam sel-sel pada baris pertama, kita akan mendapatkan kemungkinan penyusunan angka-angka pada baris pertama sebanyak 4x4! = 96. Cara ini masih menggunakan *brute force*, namun dengan mempertimbangkan fakta bahwa angka-angka pada baris yang sama tidak boleh bernilai sama. Dengan begitu didapat enumerasi semua kemungkinan (x_1,x_2,x_3,x_4) dimana $x_i \neq x_{i+1}$.
Perbaikan dengan cara ini mereduksi jumlah kemungkinan, namun masih tidak sangkil karena belum tentu angka yang dibangkitkan memenuhi fungsi kendala.

2.2.2. Perbaikan Kedua

Solusi dari permasalahan tersebut dapat kita pecahkan dengan menggunakan algoritma *bactracking.* Algoritma *backtracking* adalah perbaikan dari algoritma *brute force* yang mengenumerasi semua kemungkinan. *Backtracking* akan mereduksi secara drastis ruang pencarian dengan menghindari upa-himpunan kandidat solusi yang sebenarnya tidak pernah mengandung solusi. *Backtracking* merupakan algoritma yang berbasis DFS (*Depth-First Searching*) yang mentraversal pohon pencarian secara pre-order, dan lebih alami jika dinyatakan dalam bentuk rekursif. Namun, pada makalah ini, penulis akan menjabarkan algoritma *backtracking* secara iteratif, walaupun dalam salah satu referensi disebutkan bahwa terdapat perbedaan kecil (tapi tidak signifikan) antara melakukan *backtracking* dengan skema rekursif dan skema iteratif (lebih cepat rekursif). [2]

Selanjutnya, permasalahan yang ingin dipecahkan dapat secara tidak langsung digambarkan sebagai menemukan semua urutan (x_1, x_2, x_3, x_4) yang memenuhi kendala $P_4(x_1, x_2, x_3, x_4)$, yaitu batasan di mana hanya terdapat tepat satu angka untuk setiap nilai pada baris, kolom, dan kotak yang sama.

Pendekatan *backtrack* secara umum terdiri dari menemukan kendala/batasan tingkat lanjut $P_k(x_1,x_2,...,x_k)$ di mana $P_{k+1}(x_1,...,x_k,x_{k+1})$ juga menyatakan $P_k(x_1,...,x_k)$, untuk $0 \leq k < n$ (dalam contoh kasus pada makalah ini, n=4).

Hal ini berarti, jika $(x_1,...,x_k)$ tidak memenuhi batasan P_k, maka tidak akan ada perluasan $(x_1,...,x_k,x_{k+1})$ yang memenuhi P_{k+1}. Algoritma backtrack secara sistematis akan mengenumerasi semua solusi $(x_1,...,x_n)$ dengan hanya mempertimbangkan solusi parsial $(x_1,...,x_k)$ yang memenuhi P_k.

Algoritma *backtracking* dapat dijabarkan sebagai berikut.

- Langkah 1: $k \leftarrow 0$
- Langkah 2: (Saat ini $P_k(x_1,...,x_k)$ terpenuhi, dan $0 \leq k < n$.)
  Bentuk himpunan Sk dari semua xk+1 sedemikian sehingga $P_{k+1}(x_1,...,x_k,x_{k+1})$ terpenuhi.
- Langkah 3: Jika S_k kosong, lanjutkan ke langkah 6.
- Langkah 4: Pilihlah elemen yang mana pun dari S_k, sebutlah x_{k+1}, dan hapus/*delete* elemen tersebut dari Sk. Tambahkan k dengan 1.
- Langkah 5: (Saat ini $P_k(x_1,...,x_k)$ terpenuhi, dan $0 \leq k < n$.)

Jika k<n, kembali ke langkah 2. Jika tidak, keluarkan solusi (x_1,...,x_n), kemudian lanjutkan ke langkah 6.

- Langkah 6: (Semua elemen dari (x1,...,xn) kini telah dieksplorasi.)
  Kurangki k dengan 1. Jika k≥0, kembali ke langkah 3. Jika tidak, algoritma berhenti.

Setiap algoritma backtrack menghasilkan sebuah pohon pencarian. Pemilihan properti P_k akan berpengaruh terhadap ukuran dari pohon ini. Secara umum, Properti yang kuat akan membatasi pencarian, namun membutuhkan lebih banyak komputasi.

3. Mengestimasi Waktu Running

Misalkan T adalah sebuah pohon yang akan ditraversal oleh algoritma backtrack. Nilai cost(T) adalah ongkos untuk memroses semua simpul pada T.

$$(1)\ \ cost(T) = \sum c(t), \text{ untuk setiap } t \in T$$

Nilai c(t) adalah ongkos untuk memroses simpul $t \in T$. Nilai c() adalah ongkos untuk memroses simpul akar.

Tujuan kita adalah untuk menemukan cara untuk mengestimasi cost(T), yang artinya untuk menentukan waktu running dari program backtrack, tanpa mengetahui banyak hal tentang fungsi kendala.

Pendekatan yang biasa dilakukan untuk mengetahui nilai estimasi ini (yaitu, pendekatan Monte Carlo) melibatkan pengambilan sampel dari pohon yang akan ditraversal. Sampel diambil dengan mengeksplor sebuah jalur dari akar ke daun. Langkah-langkah yang dipilih adalah dengan memilih suksesor tiap simpul secara acak.

Algoritmanya dapat dijabarkan sebagai berikut,

- Langkah 1: k←0, D←1, C←c()
  Di sini C akan menjadi estimasi dari (1), sementara D adalah variabel pembantu yang digunakan dalam perhitungan C.
- Langkah 2: Bentuk himpunan S_k dari semua x_{k+1} di mana $P_{k+1}(x_1,...,x_k,x_{k+1})$ terpenuhi, dan ambil d_k yang merupakan jumlah elemen S_k (jika k=n, S_k kosong dan d_k=0).
- Langkah 3: Jika d_k=0, algoritma berhenti, dengan C adalah estimasi dari cost(T).
- Langkah 4: Pilih sebuah elemen $x_{k+1} \in S_k$ secara acak. (Dengan begitu, pemilihan setiap elemen akan memiliki probabilitas $1/d_k$).
  Bentuk himpuan D←d_kD, lalu bentuk C←C+ $c(x_1,\ldots,x_{k+1})$D. Tambahkan nilai k dengan 1, kemudian kembali ke langkah 2.

Algoritma ini akan membentuk sebuah jalur traversal yang acak pada pohon, tidak melakukan *backtracking*, dan kemudian menghitung nilai estimasi yang dirumuskan sebagai berikut,

$$C = c() + d_0c(x_1) + d_0d_1c(x_1,x_2) + \ldots$$

di mana d_k adalah sebuah fungsi (x1,…,x_k), atau dengan kata lain, x_{k+1} memenuhi $P_{k+1}(x_1,...,x_k,x_{k+1})$.

4 Kekurangan dari Algoritma Ini

Dengan menggunakan algoritma estimasi di atas, artinya kita sedang mencoba memprediksikan karakteristik dari sebuah pohon berdasarkan pengetahuan kita tentang hanya satu buah cabang! Kemungkinan himpunan solusi yang bersifat kombinatorial dari kebanyakan persoalan *backtrack* menjelaskan bahwa solusi parsial yang berbeda dapat menghasilkan kelakukan berbeda yang drastis pula. Di bawah ini akan diilustrasikan sebuah contoh yang sesuai.

Misalkan ada sebuah percobaan yang menghasilkan pilihan 1 dengan probabilitas 0.999, sementara pilihan 1000001 dihasilkan dengan probabilitas 0.001. Nilai yang diinginkan (solusi) adalah 1001, namun sampel yang terbatas hampir selalu menyakinkan kita bahwa solusi yang benar adalah 1.

Dari ilustrasi tersebut, terlihat jelas adanya bahaya bahwa pendekatan kita akan hampir selalu rendah, kecuali untuk beberapa kasus tertentu (yang sangat jarang terjadi), di mana pendekatan menjadi terlalu tinggi.

Pada akhirnya, kita dapat mengetahui bahwa nilai C yang kita hitung dari algoritma tersebut ekivalen dengan cost(T), tapi kita tidak tahu sama sekali mengenai selisih estimasi tersebut dengan nilai yang sebenarnya.

5 Perbaikan

Salah satu ide untuk mengimprovisasi algoritma di atas adalah dengan mengenalkan aturan yang lebih sistematis ke dalam langkah 4, sehingga pemilihan x_{k+1} tidak sepenuhnya acak.

- Langkah 4.1: Tentukan (dengan acak) sebuah urutan angka-angka positif sebanyak d_k, yaitu $p_k(1),p_k(2),...,p_k(d_k)$. Kemudian, pilih sebuah integer acak J_k yang memiliki rentang nilai $1 \le J_k \le d_k$ sedemikian sehingga didapat J_k=j dengan probabilitas $p_k(j)$. Ambil x_{k+1} sebagai elemen ke J_k dari S_k, dan bentuk himpunan D←D/$p_k(J_k)$, C←C+$c(x_1,...,x_{k+1})$D. Tambahkan nilai k dengan 1, lalu kembali ke langkah 2.

Langkah 4 ini adalah kasus khusus $p_k(j)=1/d_k$ untuk semua j. Lagi-lagi dapat kita buktikan bahwa nilai C yang dihasilkan merupakan cost(T), tidak peduli seberapa ‘aneh’ kemungkinan $p_k(j)$ yang diperoleh pada langkah 4.1.

Beberapa pilihan nilai $p_k(j)$ dapat jauh lebih baik daripada pilihan yang lain, dan fakta yang paling menarik

adalah bahwa salah satu dari pilihan yang mungkin adalah pilihan yang betul-betul tepat.

IV. KESIMPULAN

Kesangkilan suatu program yang menggunakan algoritma *backtracking* dapat dihitung nilai estimasinya dengan algoritma yang telah dijelaskan di atas. Algoritma tersebut diketahui telah dites pada banyak aplikasi, dan walaupun mungkin terjadi kasus yang buruk sekali (seperti yang telah dijelaskan pada bab 4), algoritma ini secara konsisten berperilaku dengan sangat baik, bahkan dalam permasalahan yang sengaja diberikan sebagai contoh kasus yang buruk.

REFERENSI

[1] Annika Poerschke, *Estimating the Efiiciency of Backtrack Program*, http://www.math.tu-berlin.de/~ziegler, diakses tanggal 22 Mei 2007 pukul 17:30.

[2] Dave Tapley, *Computational Sudoku Puzzle Solving*, http://compalg.inf.elte.hu/~tony, diakses tanggal 23 Mei 2007 pukul 9:00.

[3] Rinaldi Munir, *Diktat Kuliah IF2251 Strategi Algoritmik*, Program Studi Teknik Informatika ITB, 2007.