# CONSTRAINT SATISFACTION PROBLEMS (2)

### 4.5 Penyusunan Urutan Variabel dan Nilai

Sebuah algoritma pencarian untuk CSP memerlukan urutan pengisian variabel, demikian pula dengan mekanisme backtracking-nya, harus sesuai dengan urutan yang ditentukan sebelumnya. Memilih urutan yang tepat untuk variabel dan nilai dapat meningkatkan efisiensi solusi CSP.

### 4.5.1 Urutan Variabel

Berbagai percobaan dan analisis dari berbagai peneliti menunjukkan bahwa urutan variabel untuk diinisialisasi dapat memberikan efek terhadap kompleksitas backtracking dalam pencarian. Urutan tersebut dapat dilakukan dengan mekanisme:

- Urutan Statik, yaitu mengurutkan pengisian variabel sebagaimana diberikan sebelum pencarian dilakukan, dan tidak akan diganti setelahnya; atau
- Urutan Dinamik, yaitu pemilihan pengisian variabel berikutnya dengan tergantung pada keadaan pencarian saat ini.

Urutan dinamik tidak dapat diterapkan pada semua algoritma pencarian, misalnya degnan backtracking, tidak ada informasi ekstra yang tersedia terkait dengan cara pemilihan urutan selain urutan di awal. Bagaimanapun, dengan forward checking, keadaan saat ini termasuk di dalam domain pengisian variabel, sehingga pemangkasan telah terjadi sesuai dengan inisiasi yang terjadi, dan dengan demikian dimungkinkan untuk memilih variabel berikutnya berdasarkan informasi ini.

Beberapa heuristik telah dikembangkan dan dianalisis untuk menentukan urutan pengisian variabel. Yang paling sering digunakan berdasarkan pada **prinsip "first-fail"** (kegagalan pertama). Hal ini dapat dijelaskan sebagai berikut: untuk berhasil, coba pertama-tama dengan yang Anda rasa akan paling besar kegagalannya.

Dengan cara ini, variabel dengan alternatif paling sedikit akan dipilih untuk diinisiasikan. Sehingga urutan inisiasi variabel pada umumnya akan berbeda, untuk cabang yang berbeda, dan dibentuk dengan dinamik. Metode ini didasari pada asumsi, bahwa setiap nilai dapat berpartisipasi dalam sebuah solusi, sehingga akan lebih banyak nilai tersedia untuk mencapai keberhasilan.

Prinsip "kegagalan pertama", sering nampak menyesatkan, karena pada dasarnya kita tidak ingin gagal. Alasannya adalah bahwa jika solusi parsial saat ini tidak membawa pada solusi yang lengkap, maka makin cepat diketahui akan semakin baik. Dengan demikian melalui usaha untuk menggagalkan di awal, maka kegagalan akan dideteksi lebih awal, dan merupakan keuntungan di masa depan. Di sisi lain, jika solusi parsial saat ini dapat diekspansi ke dalam solusi yang lengkap, maka setiap variabel yang tersisa harus diinisiasikan, dan variabel dengan nilai domain yang paling "kecil" kemungkinannya adalah variabel yang akan paling sulit dicari isinya (karena kalau menginisiasi variabel lain, nilai domain akan berkurang, dan akan semakin mungkin mengalami kegagalan). Oleh karena itu prinsip di atas, dapat pula dikatakan dengan: "carilah solusi untuk

kasus-kasus yang sulit terlebih dahulu: kasus seperti itu akan menjadi semakin sulit jika diabaikan di awal."

Heuristik semacam ini akan mereduksi kedalaman rata-rata dari percabangan dalam pohon pencarian dengan men-trigger kegagalan di awal.

Heuristik lainnya, adalah dengan mengaplikasikan semua variabel yang memiliki nilai yang sama, untuk memilih variabel yang dapat berpartisipasi dalam constraint yang paling banyak (bukan yang paling sulit). Heuristik semacam ini sebenarnya juga berupaya untuk lebih dahulu mengisi variabel untuk kasus-kasus yang sulit.

Ada juga heuristik untuk pengurutan statik, yang cocok untuk backtracking. Heuristik ini mengatakan: pilihlah variabel dengan jumlah constraint terbesar dengan variabel sebelumnya. Sebagai contoh, pada saat memecahkan problem map coloring, akan sangat logis untuk memberi warna pada daerah yang memiliki irisan batas (bertetangga), dengan titik yang telah diwarnai sebelumnya, sehingga dengan demikian konflik warna akan dapat dideteksi sesegera mungkin.

### 4.5.2 Urutan Nilai

Setelah keputusan diambil untuk menginisiasi sebuah variabel, maka akan tersedia nilai yang dapat dialokasikan untuk variabel tersebut. Sekali lagi, urutan pengalokasian nilai akan dapat memiliki akibat pada urutan ditemukannya solusi pertama. Bagaimanapun, jika semua solusi adalah wajib untuk ditemukan, atau jika tidak ada solusi, maka urutan nilai akan berbeda.

Urutan alokasi nilai yang berbeda akan memberikan urutan yang berbeda pula pada percabangan dalam pohon pencarian. Ini adalah sebuah keuntungan yang dapat menjamin bahwa sebuah cabang yang akan membawa pada solusi akan ditemukan lebih dahulu, dibanding cabang yang berupa jalan buntu. Sebagai contoh, jika CSP memiliki sebuah solusi, dan jika sebuah nilai yang cocok dipilih untuk setiap variabel, maka sebuah solusi akan ditemukan tanpa harus melakukan backtracking.

Seandainya kita telah memilih sebuah variabel untuk diinisiasi: bagaimana kita memilih nilai untuk dicoba pertama? Ada kemungkinan bahwa tidak ada satupun nilai yang akan memenuhi kriteria, dalam kasus tersebut, setiap nilai untuk variabel saat ini harus diperhitungkan, dan urutan menjadi tidak penting. Pada sisi lainnya, jika kita dapat menemukan sebuah solusi lengkap berdasarkan pada inisiasi sebelumnya, maka perlu dipilih nilai yang paling mungkin untuk sukses, dan tidak menimbulkan konflik. Dengan demikian kita menerapkan **prinsip "succeed first"**.

Salah satu heuristik yang sering digunakan adalah untuk memaksimalkan jumlah pilihan yang tersedia. Biasanya, algoritma AC-4, cocok untuk heuristik seperti ini, dengan menghitung nilai "pendukung". Dimungkinkan pula untuk menghitung "janji" yang muncul dari setiap nilai, yaitu perkalian (produk) antara ukuran domain dengan variabel berikutnya setelah memilih isi nilai untuk variabel-variabel terkait (ini merupakan batas atas dari jumlah solusi yang memungkinkan). Nilai dengan "janji" tertinggi akan dipilih. Dimungkinkan pula untuk menghitung persentasi nilai dalam domain berikutnya yang tidak lagi dapat digunakan. Pilihan terbaik akan didapat dari nilai dengan biaya terendah.

Heuristik yang lain adalah dengan mencari nilai (dari yang tersedia), yang akan membawa pada solusi termudah. Hal ini memerlukan estimasi kesulitan terhadap problem CSP-nya. Salah satu metode yang diusulkan adalah dengan mengkonversi sebuah CSP ke dalam struktur pohon CSP dengan menghapus jumlah minimum arc dan kemudian menemukan semua solusi dari CSP yang dihasilkan lagi (solusi yang lebih tinggi dihitung, merupakan CSP yang lebih mudah).

Untuk problem yang dibuat secara acak, dan biasanya dalam bentuk umum, pencarian solusi dengan cara mengevaluasi setiap nilai dianggap tidak berharga, dibanding keuntungan untuk memilih nilai rata-rata dari problem yang dihadapi. Untuk problem yang spesifik, sebaiknya digunakan juga berbagai informasi yang memungkinkan dibuatnya urutan nilai sesuai dengan prinsip untuk memilih nilai yang kemungkinan suksesnya terbesar.

### 4.6 Heuristik dalam CSP

Dalam beberapa tahun terakhir, pencarian lokal dalam bentuk greedy search (mengambil nilai heuristik terbaik), menjadi popular kembali. Dengan algoritma yang bersifat greedy search, nilai ketidakkonsistenan akan digantikan seiring jalannya pencarian. Dengan metafora “perbaikan” perlahan (hill climbing), algoritma greedy search, akan mencapai solusi lengkap. Untuk menghindari adanya “lokal optima”, algoritma harus dilengkapi dengan heuristik yang “mengacak” pencarian. Sifat stokastik ini secara umum akan mengurangi kelengkapan dari pencarian yang dihasilkan dalam metode pencarian sistematik.

Metodologi pencarian lokal untuk CSP, menggunakan beberapa istilah berikut ini:

• **state (configuration):** sebuah alokasi yang memungkinkan untuk membentuk solusi untuk semua variabel, jumlah state adalah sama dengan hasil perkalian dari ukuran domain dan variabel.

• **evaluation value:** jumlah constraint yang tidak memenuhi syarat (biasanya diberikan bobot).

• **neighbor:** keadaan yang didapatkan dari keadaan saat ini, dengan menggantikan isi sebuah variabelnya.

• **local-minimum:** sebuah state yang bukan merupakan solusi, dan memiliki nilai evaluasi tetangga yang semuanya lebih besar (secara heuristik) atau sama dengan nilai evaluasi state sekarang.

• **strict local-minimum:** sebuah state yang bukan merupakan solusi, dan memiliki nilai evaluasi tetangga yang semuanya lebih besar (secara heuristikdibanding nilai evaluasi state sekarang.

• **non-strict local-minimum**: state yang merupakan local-minimum namun tidak mutlak.

#### 4.6.1 Hill Climbing

Hill Climbing kemungkinan merupakan algoritma yang paling terkenal dalam pencarian lokal. Ide dari algoritma ini adalah:

1. mulai pada sembarang keadaan

2. pindah ke tetangga dengan nilai evaluasi terbaik
3. jika local minimum mutlak dicapai, maka mulai pencarian kembali pada keadaan secara acak.

Prosedur ini dilakukan berulang hingga solusi dicapai. Di dalam algoritma, di bawah ini, parameter Max_Flips, digunakan untuk membatasi jumlah maksimal langkah di antara keadaan restrart, yang akan membantu jika algoritma terjebak dalam keadaan lokal minimum mutlak.

**Algorithm Hill-Climbing**

```
procedure hill-climbing(Max Flips)
  restart: s <- random valuation of variables;
  for j:=1 to Max Flips do
    if eval(s)=0 then return s endif;
    if s is a strict local minimum then
        goto restart
    else
      s <- neighborhood with smallest evaluation value
    endif
  endfor
  goto restart
end hill-climbing
```

Perhatikan bahwa algoritma Hill Climbing, harus mengevaluasi semua state tetangga, dan ini kemungkinan akan memakan waktu yang sangat lama.

**4.6.2 Min Conflicts**

Untuk menghindari bahwa semua tetangga harus dieksplorasi dari keadaan sekarang, maka beberapa heuristik diusulkan sebagai perbaikan.

Heuristik Min-Conflics akan memilih secara acak variabel apapun yang konflik, yaitu setiap variabel yang diperhitungkan dalam keadaan tidak memenuhi constraint, dan kemudian mengambil sebuah nilai yang meminimalkan jumlah constraints yang tidak terpenuhi. Jika tidak ada nilai yang memenuhi, maka akan diambil satu nilai secara acak yang tidak menaikkan jumlah constraints yang tidak dipenuhi. Nilai variabel saat ini, diambil jika dan hanya jika semua nilai lainnya tidak mengurangi jumlah constraints yang tidak dipenuhi.

**Algorithm Min-Conflicts**

```
procedure MC(Max Moves)
  s <- random valuation of variables;
  nb moves <- 0;
  while eval(s)>0 & nb moves<Max Moves do
    choose randomly a variable V in conflict;
    choose a value v' that minimizes the number of conflicts for V;
    if v' # current value of V then
      assign v' to V;
      nb moves <- nb moves+1;
    endif
  endwhile
  return s
end MC
```

Perhatikan bahwa algoritma min-conflicts yang murni sebagaimana diberikan di atas, tidak dapat meninggalkan local minimum. Tambahan lagi, jika algoritma mencapai local-minimum mutlak, maka tidak akan ada lagi pergerakan, sehingga algoritma ini tidak akan dapat dihentikan.

**4.6.3 GSAT**

GSAT merupakan sebuah greedy search yang dilakukan lokal untuk memenuhi formula logika dalam bentuk konjungsi normal (CNF). Problem seperti itu disebut dengan SAT atau k-SAT (k adalah jumlah literal dalam formula), dan dikenal memiliki kompleksitas eksponensial (NP-complete).

Prosedur dimulai dengan inisiasi secara acak dari variabel di dalam problem, dan berusaha mencapai derajat pemenuhan tertinggi melalui suksesi dari tranformasi yang kecil-kecil, disebut sebagai repairs atau flips.

## Algorithm GSAT

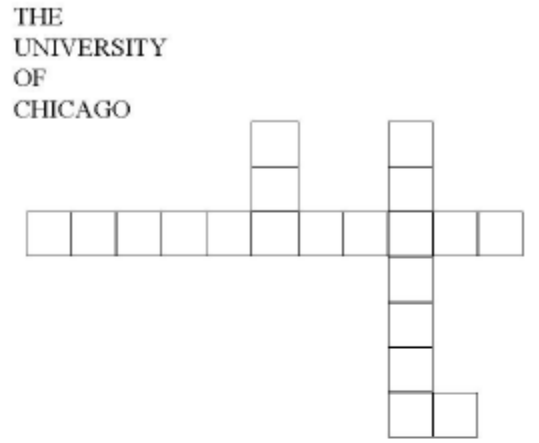
---

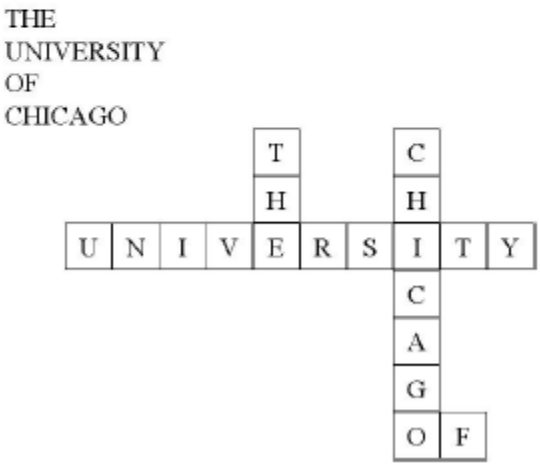
```
procedure GSAT(A,Max Tries,Max Flips)
  A: is a CNF formula
  for i:=1 to Max Tries do
    S <- instantiation of variables
    for j:=1 to Max_Iter do
      if A satisfiable by S then
        return S
      endif
      V <- the variable whose flip yield the most important raise in the
number of satisfied clauses;
      S <- S with V flipped;
    endfor
  endfor
  return the best instantiation found
end GSAT
```

**Pertanyaan:**

1. Crossword Puzzle

Asumsikan kita memiliki sejumlah kata W1, .., Wn untuk mengisi sebuah crossword. Contoh dari crossword adalah sebagai berikut, dilengkapi dengan kemungkinan isinya.




Buatlah sebuah definisi problem CSP untuk crossword di atas:

a. Berikan variabel apa saja yang harus dipenuhi isinya
b. Berikan nilai domain yang mungkin diisikan untuk setiap variabel
c. Constraints apa saja yang harus dipertahankan di antara variabel? (tuliskan dalam bentuk pseudocode)
d. Berikan contoh pengisian dengan mendemonstrasikan keterbatasan konsistensi arc
e. Jelaskan mengapa backtracking tidak cocok untuk CSP ini
f. Jelaskan secara singkat penggunaan min-conflicts, dalam mengisi puzzle ini
g. Demonstrasikan prosedur pada f. untuk crossword dengan 4 kata.

2. Penjadwalan Kelas

Diberikan sebuah pen jadwalan kelas sebagai berikut: ada 4 kelas (C1,… ,C4), dan 3 ruangan (R1, .., R3). Terdapat penjadwalan sebagai berikut:

| Class | Time |
|---|---|
| C1 | 8am-10:30am |
| C2 | 9am-11:30pm |

| | |
|---|---|
| C3 | 10am-12:30pm |
| C4 | 11am-1:30pm |

Terdapat pembatasan sebagai berikut:

- Setiap kelas harus menggunakan salah satu dari ketiga ruangan yang tersedia
- R3 terlalu kecil untuk C3
- R2 dan R3 terlalu kecil untuk C4

Salah satu cara untuk memformulasikan problem ini adalah dalam CSP dengan mengambil asumsi C1, .., C4 sebagai variabel, dan R1,.., R3 sebagai nilai yang memungkinkan untuk mengisi variabel.

a. Tunjukkan kemungkinan isi nilai untuk setiap variabel sesuai dengan constraints di atas.
b. Ekspresikan constraints problem secara formal.
c. Diberikan bahwa setiap pasang variabel dalam constraint di nomor b, tunjukkan pasangan mana saja yang memiliki konsistensi arc, untuk nilai inisial sesuai nomor a. Untuk pasangan-pasangan tersebut, yang tidak arc konsisten, tunjukkan operasi yang dapat membuatnya menjadi arc konsisten.