CONSTRAINT SATISFACTION PROBLEMS

4.1 Tujuan Instruksional

Mahasiswa dapat memformulasikan sebuah problem dalam tipe CSP

Mahasiswa dapat menggambarkan graf sebagai jalur solusi sebuah CSP

Mahasiswa dapat memecahkan berbagai CSP dengan menggunakan algoritma yang tepat

Algoritma yang menjadi kandidat dalam mencari sebuah solusi CSP, antara lain:

- Backtracking
- Forward checking
- Constraint propagation
- Arc and path consistency
- Variable and value ordering
- Hill climbing

Mahasiswa dapat menganalisis property dari algoritma di atas, dalam hal:

- Kompleksitas waktu
- Kompleksitas ruang
- Terminasi / Kelengkapan
- Optimasi

4.2 CSP

CSP merupakan sebuah pendekatan dari problem yang bersifat matematis dengan tujuan menemukan keadaan atau obyek yang memenuhi sejumlah persyaratan atau criteria. Sebuah *constraint* diartikan sebagai sebuah batasan dari solusi memungkinkan dalam sebuah problem optimasi. Banyak problem dapat dikategorikan sebagai CSP, diantaranya:

Contoh 1: n-queen problem

Dalam problem ini diusahakan bahwa sejumlah n ratu dalam sebuah nxn papan catur berada dalam keadaan “aman” (tidak dapat saling menyerang), dan dalam hal ini warna dari biji catur tidak menjadi batasan. Solusi yang dapat ditawarkan adalah keadaan bahwa tidak ada dua ratu yang berada pada barus, kolom atau diagonal yang sama. Problem ini mula-mula disampaikan pada tahun 1848 oleh seorang pemain catur bernama Max Bazzel, dan dalam perjalanan masa, banyak ahli matematik, termasuk Gauss, mencoba mencari solusi yang optimal untuk problem ini. Pada tahun 1874, S. Gunther

mengajukan sebuah metode untuk mencari solusi dengan menggunakan determinan, dan W.L. Glaisher mencoba mengerjakan pendekatan ini lebih lanjut.

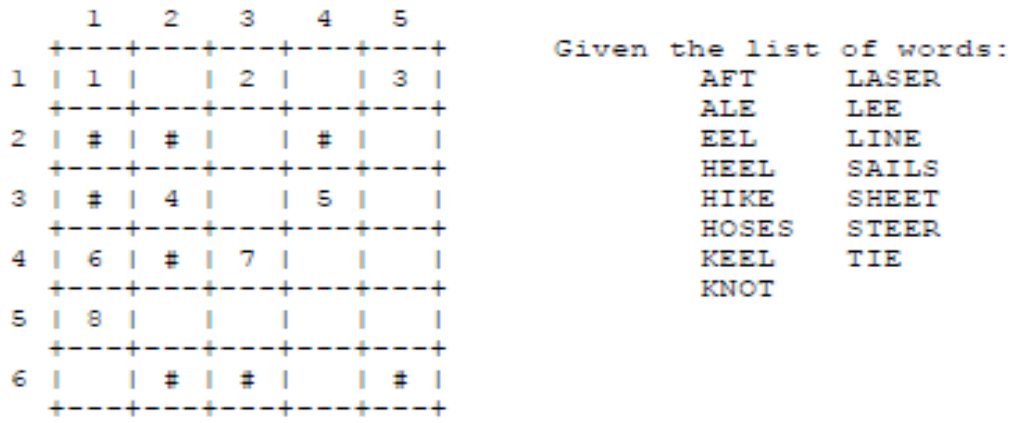
Problem 8-queens memiliki 92 solusi yang berbeda. Jika sebuah solusi berbeda hanya pada operasi yang simetris (rotasi atau refleksi) pada papan catur, dan dianggap sebagai satu solusi saja, maka 8-queens memiliki 12 solusi yang unik. Tabel di bawah ini memberikan jumlah solusi untuk problem n-queens, baik untuk solusi yang berbeda dan unik.

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	285,053
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

Ada yang menarik bahwa 6-queens memiliki solusi yang lebih sedikit dibanding 5-queens.

Contoh 2: Crossword (teka-teki silang)

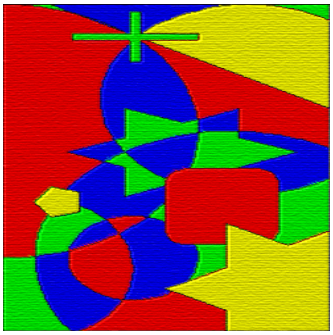
Kita berusaha untuk mengisi kotak dengan kata-kata yang disediakan. Misalnya:



Nomor 1 s.d. 8 menunjukkan kata-kata yang akan dimulai pada lokasi tersebut.

Contoh 3: Mewarnai peta (map coloring)

Diberikan sebuah peta planar (dalam satu bidang), dan diberikan asumsi bahwa kita hanya dapat mewarnai dengan sejumlah k warna, warnailah peta sehingga tidak ada 2 bidang bertetangga yang memiliki warna sama. Contoh pewarnaan dengan 4 warna untuk peta sebagai berikut:



Dua bidang disebutkan sebagai bertetangga jika ada bagian dari batas wilayah yang bersinggungan, dan saling menyambung.

Untuk peta di atas, jelas bahwa tiga warna tidak akan cukup, karena ada satu wilayah yang dikelilingi dengan lebih dari 3 wilayah lainnya. Problem di atas dapat ditangani oleh program computer, namun demikian pembuktian secara matematisnya tidak dapat dilakukan secara langsung.

Contoh 4: Boolean Satisfiability Problem (SAT)

SAT adalah sebuah problem untuk pengambilan keputusan. Setiap instans dari problem SAT akan menyertakan operator Boolean (AND, OR, NOT). Pertanyaan yang harus dijawab adalah: diberikan sebuah ekspresi logika, adakah sebuah nilai TRUE atau FALSE dari variabel dalam ekspresi tersebut yang akan memberikan jawaban benar untuk ekspresi secara keseluruhan (membentuk tautologi).

Dalam matematika, sebuah formula logika proporsional disebut sebagai satisfiable jika nilai benar dapat diberikan pada variable-variabelnya yang menyebabkan nilai formula menjadi benar. Problem ini termasuk jenis problem NP-complete, artinya tidak dapat dipecahkan dalam waktu yang polinomial, dan selalu membutuhkan ruang dan waktu yang eksponensial. Problem SAT, merupakan salah satu permasalahan yang menjadi dasar dalam teknik komputasi, termasuk di dalamnya: algoritma, kecerdasan buatan, perancangan perangkat keras dan verifikasi.

Problem ini akan dapat direduksi menjadi problem yang lebih sederhana, meskipun masih memerlukan waktu pemrosesan yang eksponensial. Misalnya dengan menerapkan Hukum de Morgan, dapat diasumsikan bahwa operator NOT hanya berlaku untuk variabel langsung, bukan ke berlaku untuk ekspresi. Sebuah variable dan negasinya disebut sebagai literal. Contoh jika kita memiliki dua literal x1 dan not(x2), dan jika kita berikan operator OR sehingga membentuk sebuah ekspresi / klausa (x1 OR not(X2)), maka melalui Hukum de Morgan, akan diperoleh not(x1 AND x2), sebagai bentuk Conjunctive Normal Form (CNF). Menentukan SAT dari bentuk formula normal inipun masih merupakan problem yang NP-complete, meskipun setiap dibatasi dengan paling banyak 3 literal. Problem terakhir ini sering disebut dengan 3-SAT, 3 CNFSAT atau 3-satisfiability.

Pada kasus lainnya, jika kita membatasi jumlah klausa dengan paling banyak 2 literal, maka problem, 2 SAT berada dalam waktu polinomial. Hal yang sama pun akan berlaku jigak setiap klausa berupa Horn-clause, yaitu hanya memiliki paling banyak 1 literal bernilai benar.

Contoh 5: Cryptarithmatic Problem

Diberikan sebuah pola sebagai berikut:

```
  S E N D
  M O R E
=========
M O N E Y
```

Maka harus diberikan nilai aritmatik untuk setiap alfabet dalam pola penjumlahan tersebut.

Contoh-contoh yang telah dituliskan di atas, dan beberapa contoh lain dalam kehidupan nyata, seperti: penjawalan kelas, penjadwalan mesin di pabrik, dsb, adalah contoh-contoh problem CSP, dengan pola problem sebagai berikut:

- Sebuah himpunan variable (x1, x2, …, xn);
- Untuk setiap variabel dalam sebuah domain Di, sebagai nilai yang mungkin untuk variable terkait;
- Sebuah himpunan constraint, yaitu relasi antara nilai variabel. Relasi ini dapat diberikan dalam bentuk formula, himpunan, ataupun prosedur.

Maka CSP yang terbentuk adalah problem untuk menemukan solusi untuk setiap variable dalam domain Di, sehingga semua constraint terpenuhi. Atau dalam bentuk

$$(\text{exist } x1)..(\text{exist } xn)\ (D1(x1)\ \&\ ..\ Dn(xn) \rightarrow C1..Cm)$$

4.3 Representasi CSP

CSP biasanya direpresentasikan dengan sebuah graf, tanpa arah, disebut sebagai Contraint Graph, dengan node-nya adalah variable dan jalurnya adalah batasan yang dimiliki oleh node. Untuk batasan tunggal, dapat dilengkapi dengan mendefinisikan ulang domain yang ada sehingga mengisi variabel tersebut. Constraint dengan batasan yang lebih tinggi dapat dinyatakan dalam arc (jalur berarah).

Sebuah constraint akan dapat mempengaruhi satu atau lebih variabel (1 .. n) dalam definisi permasalahan. Jika semua constraint dalam CSP adalah biner (ada minimal 2 variabel kemungkinan untuk solusi berikutnya), maka semua variabel dan constraint dapat direpresentasikan dalam sebuah graf, dan algoritma CSP dapat diberlakukan untuk mengeksploitasi graf.

Konversi dari sebuah problem CSP ke dalam graf binernya, dilandasi ide untuk memperkenalkan sebuah variabel baru yang mengenkapsulasi himpunan variabel yang memiliki constraint. Variabel baru ini merupakan hasil "perkalian" Cartesian antara domain dengan setiap variabel. Perkalian ini akan membentuk kombinasi antara domain dan variabel, yang merupakan himpunan terbatas (meskipun bisa sangat banyak kombinasinya).

Sekarang, setiap n-ary constraint dapat dikonversi ke dalam constraint tunggalnya, yang mengenkapsulasi variabel awalnya. Dari constraint tunggal ini, sebuah batasan terhadap variabel akan dapat dicari solusinya. Dengan demikian setiap n-ary constraint dapat disubtitusikan dengan variabel yang sesuai di dalam domainnya. Hal ini tentu menarik sebab semua CSP tentu akan dapat direpresentasikan ke dalam binary constraint-nya.

Mari kita lihat kembali **contoh nomor 2, tentang crossword**.

Kita memiliki variabel sebagai berikut:

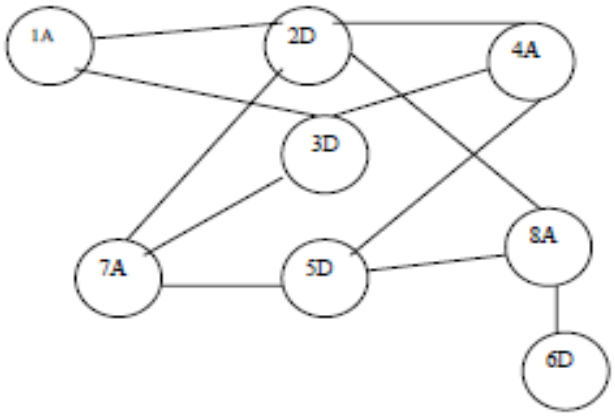
```
VARIABLE | STARTING CELL | DOMAIN
================================================
1ACROSS  | 1             | {HOSES, LASER, SAILS, SHEET, STEER}
4ACROSS  | 4             | {HEEL, HIKE, KEEL, KNOT, LINE}
7ACROSS  | 7             | {AFT, ALE, EEL, LEE, TIE}
8ACROSS  | 8             | {HOSES, LASER, SAILS, SHEET, STEER}
2DOWN    | 2             | {HOSES, LASER, SAILS, SHEET, STEER}
3DOWN    | 3             | {HOSES, LASER, SAILS, SHEET, STEER}
5DOWN    | 5             | {HEEL, HIKE, KEEL, KNOT, LINE}
6DOWN    | 6             | {AFT, ALE, EEL, LEE, TIE}
```

Domai dari setiap variabel adalah daftar kata yang "kemungkinan" merupakan isi dari variabel tersebut. Sehingga diketahui bahwa untuk variabel 1ACROSS, memerlukan 5 huruf, 2DOWN lima huruf, 5DOWN memerlukan 4 huruf, dst. Perhatikan bahwa di dalam setiap domain terdapat 5 kemungkinan solusi, dan ada 8 variabel, dengan demikian jumlah keadaan yang memungkinkan adalah 5^8 = 390.625.

Semua constraint dapat direduksi ke dalam bentuk biner:

```
        1ACROSS[3] = 2DOWN[1]  i.e. the third letter of 1ACROSS must be
equal to the first letter of 2DOWN
        1ACROSS[5] = 3DOWN[1]
        4ACROSS[2] = 2DOWN[3]
        4ACROSS[3] = 5DOWN[1]
        4ACROSS[4] = 3DOWN[3]
        7ACROSS[1] = 2DOWN[4]
        7ACROSS[2] = 5DOWN[2]
        7ACROSS[3] = 3DOWN[4]
        8ACROSS[1] = 6DOWN[2]
        8ACROSS[3] = 2DOWN[5]
        8ACROSS[4] = 5DOWN[3]
        8ACROSS[5] = 3DOWN[5]
```

Dari hasil reduksi permasalahan, dapat dibentuk graf constraint, sebagai berikut:



4.4 Pencarian Solusi CSP

Dalam bagian berikut akan dibahas empat metode umum dalam pencarian solusi CSP, yaitu: generate and test, backtracking, consistency driven dan forward checking.

4.4.1 Generate and Test

Melalui cara ini, kita harus membangkitkan satu per satu alokasi variabel sehingga memenuhi semua constraint-nya. Struktur program untuk cara ini sangat sederhana, hanya berupa konstruksi loop, satu untuk setiap variabel, dan setiap constraint. Metode ini tidak efektif karena memiliki kompleksitas waktu yang sangat besar.

4.4.2 Backtracking

Dalam metode ini, diperlukan penyusunan ulang dalam urutan pengisian variabel. Cara yang paling efektif adalah dengan mencari solusi untuk variabel dengan constraint terbanyak, atau dengan domain yang paling sedikit. Urutan akan sangat menentukan cepat atau lambatnya solusi ditemukan. Algoritma dimulai dengan mengisikan variabel dalam constraint-nya, kemudian melakukan evaluasi terhadap constraint, apakah terpenuhi atau tidak. Lakukan hal yan g sama, sampai semua variabel terisi. Jika variabel tidak dapat diisikan, maka harus dilakukan penelaahan ulang (backtracking), ke node di atasnya, atau variabel sebelumnya.

Kita lihat contoh nomor 2 kembali, tentang crossword:

Andaikata urutan pengisian variabel adalah: 1ACROSS, 2DOWN, 3DOWN, 4ACROSS, 7ACROSS, 5DOWN, 8ACROSS, 6DOWN. Maka urutan alokasi dapat dilihat sebagai berikut:

```
   1ACROSS      <- HOSES
    2DOWN       <- HOSES      => failure, 1ACROSS[3] not equal to
2DOWN[1]
                <- LASER      => failure
                <- SAILS
     3DOWN      <- HOSES      => failure
                <- LASER      => failure
                <- SAILS
      4ACROSS   <- HEEL       => failure
                <- HIKE       => failure
                <- KEEL       => failure
                <- KNOT       => failure
                <- LINE       => failure, backtrack
     3DOWN      <- SHEET
      4ACROSS   <- HEEL
       7ACROSS  <- AFT        => failure
        ..................................
```

Yang kita lihat di atas adalah Chronological Backtracking, yaitu variabel dilepas dalam urutan yang terbalik dari variabel yang diisikan, misalnya dari 4ACROSS kembali ke 3DOWN. Sebaliknya Dependency Directed Backtracking akan kembali ke node, tempat terjadinya kegagalan, tanpa memperhatikan urutan pengisian variabel, misalnya dari 4ACROSS, kembali ke 2DOWN.

Perhatikan bahwa dengan Dependency Directed Backtracking, setiap variabel akan melakukan backtracking sebanyak jumlah tetangga yang mendahuluinya. Jumlah ini disebut sebagai “lebar” (width) dari variabel. Kompleksitas waktu akan besar, seiring dengan jumlah backtracking yang terjadi. Konsekuensinya, urutan pengisian variabel akan sangat berpengaruh terhadap jalannya algoritma.

4.4.3 Consistency Driven Techniques

Teknik konsistensi secara efektif menyingkirkan banyak alokasi variabel yang inkonsisten di awal pencarian, sehingga akan mengurangi ruang pencarian. Teknik ini telah diujicobakan dan terbukti efektif untuk problem-problem yang bersifat hard (NP-complete). Teknik ini deterministic, yang berlawanan dengan sifat pencarian CSP yang non-deterministic. Dengan demikian, perhitungan yang deterministic dilakukan secepat mungkin, dan hanya melakukan perhitungan non-deterministik pada saat tidak ada lagi propagasi yang dapat dilakukan. Namun demikian, teknik konsistensi jarang diterapkan secara mandiri dalam pemecahan CSP.

Dalam CSP biner, berbagai teknik konsistensi untuk graf constraint akan diperkenalkan, yang dapat memotong ruang pencarian. Algoritma konsistensi akan dapat digunakan untuk mencari solusi parsial, dalam sebuah sub-pohon pencarian, yang dapat diperluas ke sub-pencarian berikutnya. Dengan cara seperti ini, jika ada potensi ketidakcocokan akan dapat dideteksi sedini mungkin.

4.4.3.1 Konsistensi Node

Teknik konsistensi yang paling sederhana adalah dengan merujuk pada node (variabel) pencarian biner. Setiap node akan merepresentasikan sebuah variabel V, dalam graf constraint, disebut konsisten, jika untuk setiap nilai x dalam domain saat ini dalam V, setiap constraint tunggal dapat dipenuhi. Jika domain D untuk variabel V, memiliki sebuah nilai “a”, yang tidak memenuhi constraint tunggal di dalam V, maka instansiasi untuk V, dengan “a”, akan selalu gagal. Sehingga, muncullah ketidak-konsisten-an. Cara yang terbaik untuk mencapai solusi adalah dengan mengesampingkan nilai “a” dari domain V, untuk setiap variabel V, yang tidak memenuhi constraint tunggal di dalam V.

4.4.3.2 Konsistensi Arc

Jika constraint graf adalah sebuah node yang konsisten, maka semua constraint tunggal akan dapat dikesampingkan, karena semuanya pasti akan dapat dicari solusinya. Selama kita berupaya menghasilkan biner CSP, masih ada yang tertinggal, yaitu untuk menjamin konsistensi dari constraint binernya. Dalam constraint graf, constraint biner, akan berkorespondensi dengan arc (jalur kemungkinan solusi).

Arc(Vi, Vj) disebut konsisten, jika untuk setiap nilai x dalam domain saat ini Vi, ada nilai y dalam domain Vj, sehingga Vi=x dan Vj=y, didapatkan dari constraint antara Vi dan Vj. Perhatikan bahwa konteks konsistensi arc, bersifat satu arah, yaitu jika (Vi, Vj) konsisten, belum tentu harus selalu bahwa (Vj, Vi) juga konsisten.

Jelaslah bahwa sebuah arc (Vi, Vj), dapat dibuat konsisten dengan menghapus nilai-nilai dalam domain Vi yang tidak memiliki nilai korespondensi dalam domain Dj, sehingga memenuhi constraint biner antara Vi dan Vj. Perhatikan bahwa menghapus nilai variabel tidak mengeleminasi kemungkinan solusi dari problem CSP awalnya.

Hal di atas dapat dilihat dalam algoritma:

Algorithm REVISE

```
procedure REVISE(Vi,Vj)
  DELETE <- false;
  for each X in Di do
    if there is no such Y in Dj such that (X,Y) is consistent,
    then
       delete X from Di;
       DELETE <- true;
    endif;
  endfor;
  return DELETE;
end REVISE
```

Untuk membuat setiap arc dalam constraint graf konsisten, tidak cukup untuk mengeksekusi REVISE untuk setiap arc hanya satu kali. Pada saat REVISE dilakukan untuk sebuah domain variabel Vi, maka setiap arc yang sebelumnya juga di-REVISE arc (Vj, Vi) perlu di-REVISE kembali, karena bias saja beberapa anggota dalam domain Vi tidak lagi kompatibel dengan anggota lainnya dalam domain Vi yang baru saja di-REVISE. Algoritma di bawah ini menggambarkan tingkah laku REVISE yang telah dilengkapi kemampuan untuk melakukan REVISE bagi arc lainnya (algoritma AC-1):

Algorithm AC-1

```
procedure AC-1
  Q <- {(Vi,Vj) in arcs(G),i#j};
  repeat
    CHANGE <- false;
    for each (Vi,Vj) in Q do
      CHANGE <- REVISE(Vi,Vj) or CHANGE;
    endfor
  until not(CHANGE)
end AC-1
```

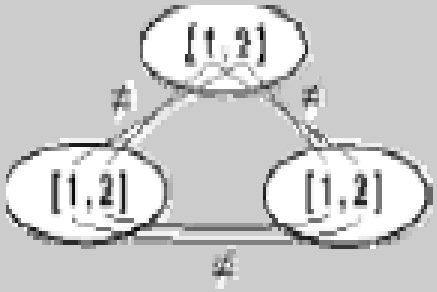
Algoritma ini tidak terlalu efisien karena revisi yang berhasil dari satu arc, akan membuat terjadinya iterasi dari semua arc pada iterasi berikutnya, meskipun hanya sedikit variabel saja yang terpengaruh revisi ini. Sesungguhnya, hanya arc dalam domain Vk, yang terpengaruh, yaitu arc (Vi, Vk). Juga jika arc (Vk, Vm) direvisi, dan domain Vk direduksi, tidak perlu harus selalu merevisi arc (Vm, Vk), karena tidak ada satupun elemen yang dihapus dari domain Vk memberikan kemungkinan solusi pada domain Vm. Variasi algoritma di bawah ini, AC-3, memberikan kemungkinan dilakukannya revisi hanya pada arc yang mungkin terpengaruh revisi sebelumnya.

Algorithm AC-3

```
procedure AC-3
  Q <- {(Vi,Vj) in arcs(G),i#j};
  while not Q empty
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      Q <- Q union {(Vi,Vk) such that (Vi,Vk) in
arcs(G),i#k,i#m}
    endif
  endwhile
end AC-3
```

Ketika algoritma AC-3, mengunjungi kembali jalur pada kesempatan kedua, algoritma ini akan melakukan evaluasi ulang terhadap nilai-nilai variabel yang telah diketahui sebelumnya (dari iterasi sebelumnya) untuk mencapai kekonsistenan atau ketidak-konsistenan, dan tidak dipengaruhi oleh reduksi domain. Karena adanya pengulangan evaluasi, dan dinilai tidak efisien, maka diperkenalkanlah algoritma AC-4 untuk menangani constraint pada jalur (edge). Algoritma ini bekerja dengan nilai individual yang berpasangan, seperti digambarkan berikut ini:

Example:



Pada awalnya, algoritma AC-4, akan menginisialisasi struktur internal, yang digunakan untuk mengingat pasangan nilai variabel yang konsisten (tidak konsisten), misalnya struktur Si,a. Ini akan menghitung pula nilai “pendukung” dari domain untuk nilai variabel – stucture counter(i,j), a – dan memindahkan isi variabel yang tidak memiliki nilai pendukung. Pada saat nilai telah dipindahkan dari dalam domain, algoritma akan menambahkan pasangan <Variabel, Nilai> ke dalam list Q untuk di-revisi efek dari nilai terhadap variabel, apakah akan konsisten atau tidak konsisten.

Algorithm INITIALIZE

```
procedure INITIALIZE
  Q <- {};
  S <- {};   % initialize each element of structure S
  for each (Vi,Vj) in arcs(G) do   % (Vi,Vj) and (Vj,Vi) are
same elements
    for each a in Di do
      total <- 0;
      for each b in Dj do
        if (a,b) is consistent according to the constraint
(Vi,Vj) then
          total <- total+1;
          Sj,b <- Sj,b union {<i,a>};
        endif
      endfor;
      counter[(i,j),a] <- total;
      if counter[(i,j),a]=0 then
        delete a from Di;
        Q <- Q union {<i,a>};
      endif;
    endfor;
  endfor;
  return Q;
end INITIALIZE
```

Setelah inisialisasi, algoritma AC-4 akan me-revisi hanya pasangan nilai yang dipengaruhi oleh revisi sebelumnya.

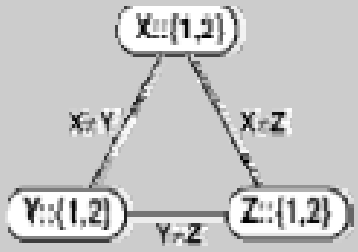
Algorithm AC-4

```
procedure AC-4
  Q <- INITIALIZE;
  while not Q empty
    select and delete any pair <j,b> from Q;
    for each <i,a> from S[j,b] do
      counter[(i,j),a] <- counter[(i,j),a] - 1;
      if counter[(i,j),a]=0 & a is still in Di then
        delete a from Di;
        Q <- Q union {<i,a>};
      endif
    endfor
  endwhile
end AC-4
```

Algoritma AC-3 dan AC-4, merupakan algoritma yang paling banyak digunakan untuk membentuk konsistensi. Ada pula algoritma AC-5, AC-6, AC-7, tapi tidak terlalu banyak digunakan sebagaimana AC3 atau AC-4.

Membentuk konsistensi arc akan menyingkirkan banyak ketidakkonsistenan dari sebuah constraint graf, tetapi akankah terbentuk solusi yang lengkap dari reduksi domain terhadap CSP? Jika ukuran domain dari setiap variabel bernilai satu, maka CSP hanya memiliki tepat satu solusi, yang dialikasikan dari setiap nilai di dalam domainnya. Jika tidak, maka tidak akan terbentuk solusi. Gambar di bawah ini menunjukkan kasus yang konsistensi sebuah arc (jalur), domain tidak kosong, namun masih saja tidak terbentuk solusi yang memenuhi semua constraint.

Example:



This constraint graph is arc consistent but there is no solution that satisfies all the constraints.

4.4.3.3 Konsistensi Path

Dengan kenyataan bahwa konsistensi arc, tidak mencukupi untuk menghindari backtracking, apakah ada cara lain untuk menghindarinya? Contoh di atas menunjukkan bahwa jika satu atau dua arc dibuka, maka semakin banyak ketidak-konsistenan dapat dibuang.

Sebuah graf disebut K-consistent, jika: terdapat sebuah variabel diantara K-1 nilai variabel, yang memenuhi semua constraint dalam variabel-variabel tersebut. Jika dipilih sembarang K^{th} variabel, maka akan terdapat sebuah nilai untuk variabel K^{th} yang memenuhi semua kondisi K variabel tersebut. Sebuah graf disebut strongly K-consistent, jika graf tersebut J-consisten, untuk semua J <= K.

Konsistensi node yang dibicarakan sebelumnya adalah 1-consistent, dan konsistensi arc dapat diekuivalensikan dengan 2-consistency (dalam hal ini biasanya dianggap bahwa jika tercipta konsistensi arc, maka tercipta pula konsistensi node. Ada berbagai algoritma untuk membuat sebuah constraint graf K-consistent untuk K>2, namun biasanya hal ini jarang dilakukan terkait efisiensi algoritma. Pengecualian untuk algoritma 3-consistent, yang sering dikenal dengan konsistensi path.

Sebuah node yang merepresentasikan konsistensi path, jika juga merupakan arc-consistent, yaitu: semua arc dari node yang ada, merupakan arc-consisten, dan hal berikut juga akan bernilai benar: untuk setiap nilai a di dalam domain Di untuk variabel Vi yang hanya memiliki satu nilai pendukung b, dari domain variabel Vj, maka akan ada nilai c dalam domain lain untuk variabel Vk, yang mengijinkan constraint biner vi dan vk, dan (c,b) adalah nilai yang diijinkan constraint biner antara Vk dan Vj.

Algoritma untuk membuat graf memiliki konsistensi patg, dapat didasarkan pada algoritma AC-4, yang menghitung jumlah nilai pendukung. Meskipun demikian algoritma ini membuang lebih banyak nilai yang inkonsisten. Dari sini dapat disebutkan pula bahwa jika sebuah constraint graf berisi n node yang sangat n-konsisten, maka sebuah solusi CSP dapat ditemukan tanpa pencarian. Namun demikian untuk kasus terburuk, nilai kompleksitas waktu, untuk mendapatkan n-konsistensi di dalam sebuah n-node constraint graf, tetap eksponensial. Jika sebuah graf sangat K-konsisten untuk K<n, maka pada umumnya, backtracking dapat dihindari, yaitu jika masih ada nilai yang inkonsisten.

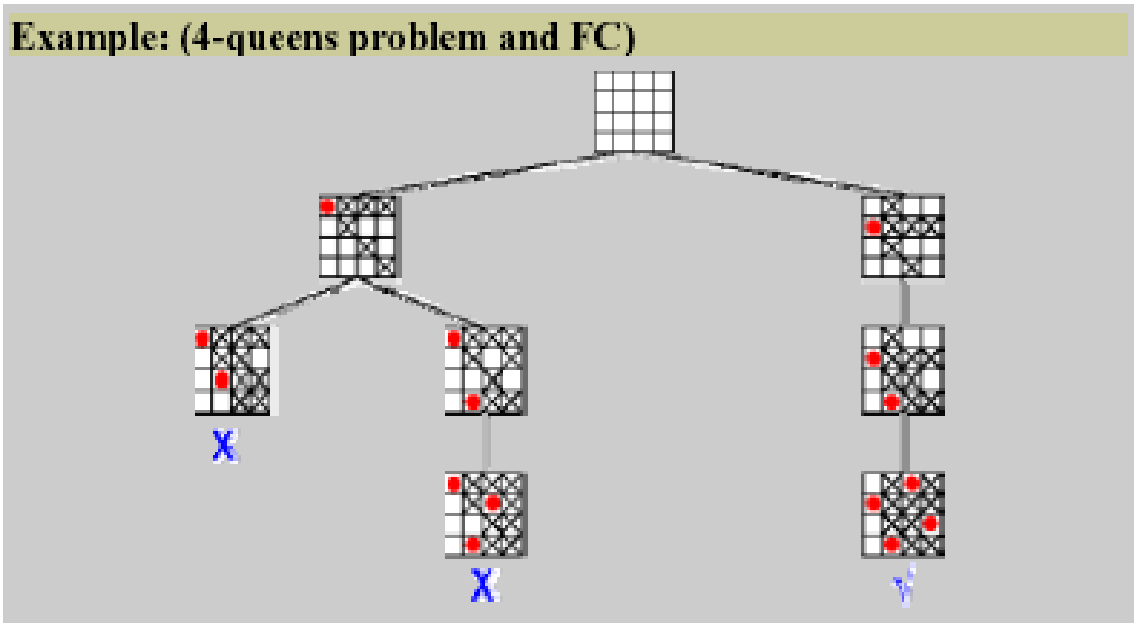
4.4.4 Forward Checking

Forward checking adalah cara termudah untuk menghindari konflik isi variabel. Sebagai pengganti konsistensi arc, untuk menginisialisasi nilai variabel, forward checking akan membatasi konsistensi arc ke dalam variabel yang belum diinisialisasi. Kita akan membicarakan tentang konsistensi arc yang dibatasi, karena melalui forward checking hanya akan dievaluasi constraint antara variabel saat ini dan variabel di depannya (berikutnya). Jika sebuah nilai dialokasikan untuk variabel saat ini, maka nilai apapun dari variabel berikutnya, yang dapat membawa konflik pada variabel saat ini, akan disingkirkan (temporer) dari domain. Keuntungan dari hal ini adalah, jika domain dari variabel berikutnya kosong, maka akan segera diketahui bahwa solusi yang terbentuk sampai saat ini adalah inkonsisten. Dengan forward checking, maka percabangan dari pohon pencarian yang akan membawa kegagalan akan dipangkas lebih awal, dibandingkan dengan backtracking. Perhatikan bahwa jika sebuah variabel baru dibuka, maka semua nilai dipastikan konsisten untuk variabel sebelumnya, dan dengan demikian evaluasi terhadap alokasi nilai yang telah dilakukan tidak perlu lagi.

Algorithm AC-3 for Forward Checking

```
procedure AC3-FC(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      consistent <- not Dk empty
    endif
  endwhile
  return consistent
end AC3-FC
```

Forward checking akan mendeteksi ketidakkonsistenan lebih awal dari backtracking, dan dengan demikian akan memangkas semua percabangan dalam pohon yang tidak konsisten. Ini akan mempersingkat waktu pencarian keseluruhan, namun harus dicatat bahwa forward checking memerlukan lebih banyak waktu pencarian, ketika setiap alokasi ditambahkan untuk solusi parsial saat ini.



Pilihan untuk melakukan forward checking, biasanya akan selalu lebih baik ketimbang melakukan backtracking.

4.4.5 Look Ahead

Forward checking hanya mengevaluasi isi constraint saat ini dan variabel yang setelahnya, bagaimana jika dilakukan evaluasi konsistensi arc yang lengkap, yang akan memangkas nilai domain, dan menghilangkan konflik? Cara ini disebut dengan (full) look ahead atau maintainig arc consistency (MAC).

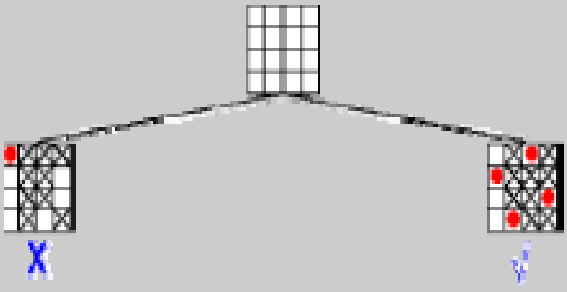
Keuntungan dari look ahead adalah bahwa dapat dideteksi konflik antara variabel berikutnya dan sebelumnya, sehingga mengijinkan dipangkasnya cabang pohon pencarian yang tidak berpotensi lebih awal dibandingkan forward checking. Sama seperti halnya forward checking, jika sebuah variabel dibuka, maka semua nilai yang telah dibuka dijamin konsisten dengan variabel sebelumnya, sehingga evaluasi terhadap alokasi variabel sebelumnya tidak diperlukan lagi.

Algorithm AC-3 for Look Ahead

```
procedure AC3-LA(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      Q <- Q union {(Vi,Vk) such that (Vi,Vk) in
arcs(G),i#k,i#m,i>cv}
      consistent <- not Dk empty
    endif
  endwhile
  return consistent
end AC3-LA
```

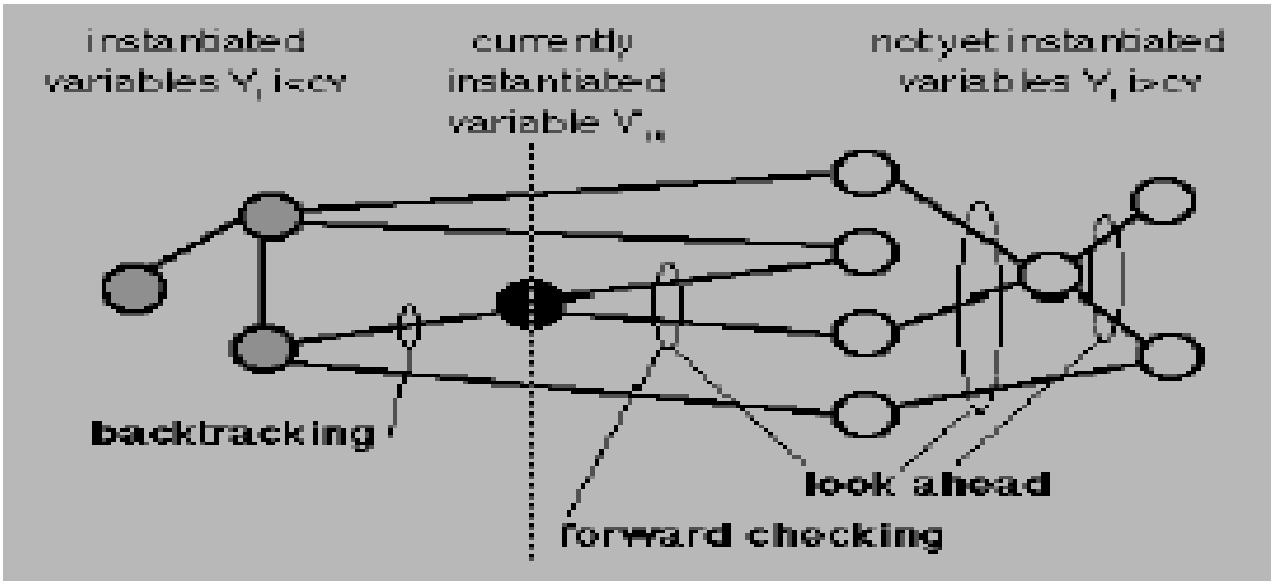
Perlu diperhatikan pula bahwa dengan look ahead, perlu dilakukan lebih banyak pekerjaan ketika sebuah alokasi variabel ditambahkan pada solusi saat ini dibandingkan dengan forward checking.

Example: (4-queens problem and LA)



4.4.6 Perbandingan Teknik Propagasi

Gambar di bawah ini memberikan perbandingan evaluasi constraints yang terjadi dengan teknik-teknik yang telah dijelaskan sebelumnya.



Lebih banyak propagasi pada setiap node, akan menghasilkan pohon pencarian dengan lebih sedikit node, tapi memiliki biaya (kerja) yang lebih banyak, yang diperoleh dari biaya pemrosesan setiap node yang lebih mahal.

Dalam satu hal, memang terbukti bahwa membentuk jalur dengan n-konsistensi dari problem awal akan mereduksi kebutuhan untuk melakukan pencarian, namun akan lebih memerlukan biaya tinggi dibanding backtracking. Dari sinilah mengapa biasanya forward checking dan backtracking lebih banyak digunakan dibandingkan dengan look ahead.