1 Propagação de restrições em domínios finitos

Nesta seção trabalharemos primariamente com restrições em domínios finitos. Domínios finitos são importantes porque costumam ser bons para modelar decisões, o que é algo com que gostaríamos que o computador ajudasse.

Um exemplo simples e bem conhecido é o problema de coloração de um mapa: dado um conjunto finito de cores (de tamanho 15, por exemplo), precisamos colorir um mapa (digamos, o do Cazaquistão) de modo que nenhuma de suas regiões receba a mesma cor que outra com que faça fronteira[1]. Outro exemplo bem conhecido é o do "casamento a moda antiga" (menos popularmente conhecido como o "problema da correspondência bipartida"). Nesse problema, temos um conjunto de homens, um de mulheres a relação `gosta/2`, que existe quando um indivíduo i gosta de outro j. O problema é separar esses grupos de homens e mulheres em casais que se gostam.

Esses dois exemplos tem a particularidade de terem restrições primitivas binárias e, por isso, são chamados de CSPs binários. Um ponto interessante em CSPs binários é que sempre podem ser representados como um grafo não direcionado: cada variável (cada indivíduo no segundo exemplo ou cada região do Cazaquistão, no primeiro) é representada como um nó e cada restrição como um arco entre suas variáveis. Mais em geral, restrições n-árias CSPs podem ser representados como um multi-grafo (isto é, um grafo em que podem existir mais de uma aresta entre dois vértices).

Em particular, problemas como os de roteamento e criação de cronogramas costumam ser facilmente expressos como CPs em domínio finito, o que indica sua importância comercial.

Sendo de uso tão amplo, essa classe de problemas (a de CPs em domínios finitos) foi estudada por diferentes comunidades científicas. A comunidade de Inteligênica Artificial desenvolvou técnicas de consistência por arco e por nó, para CSPs, a comunidade de programação por restrições desenvolveu técnicas de propagação de limites e a comunidade de pesquisas operacionais desenvolveu técnicas de programação inteira. Daremos uma olhada em cada uma dessas abordagens a seguir.

1.1 Consistência por nó e por arco

Resolução de CSPs por consistência por nó e por arco acontece em tempo polinomial (mas, possívelmente, de forma incompleta)[2]. A ideia aqui é diminuir os domínios das variáveis, transformando o problema em outro equivalente (com as mesmas soluções). Se o domínio de alguma variável for vazio, é o fim do CSP.

[1] Incidentalmente, acontece que esse problema é essencialmente o mesmo que as companhias de aviação tem para alocar seu tráfico aéreo.

[2] Mas ela, assim como as demais formas de consistência vistas aqui pode ser usada em conjunto com *backtracking*, gerando um resolvedor completo.

Essa forma de resolução é dita baseada em consistência porque ele funciona propagando informações dos domínios de cada variável para os demais, tornando-os, em algum sentido, "consistentes" entre si.

Definição 1.1. *Uma restrição r/n é dita* ***Consistente por nó*** *se $n > 1$ ou, X sendo for uma variável de r, se para cada d no domínio de X* `X=d,r(X).` *resulta em sucesso. Uma restrição composta é dita consistente por nó se cada uma de suas restrições primitivas o é.*

Consistente por nó

Definição 1.2. *Uma restrição r/n é dita* ***consistente por arco*** *se $n \neq 2$ ou, se r é uma restrição nas variáveis X e Y e se D_x é o domínio de X e D_y o de Y, então $x \in D_x \Rightarrow \exists y \in D_y : X = x, Y = y, r(X, Y)$. resulta em sucesso. Uma restrição composta é dita consistente por arco se cada uma de suas restrições primitivas o é.*

Consistente por arco

Essas noções de consistência são noções fracas no sentido de que um CSP pode não ser satisfazível e ainda manter consistência por arco e por nó.

Não é difícil escrever um código para manter consistêncai por arco e por nós. A seguir segue um exemplo. Ele é para fins demonstrativos: para algoritmos mais eficientes veja [1]. O funtor `apply/2` é o definido no Capítulo 6.

Código 1: Consistência por nó

```
% consistente_por_no([Dominio-Restricao|DsRs],
%                     [NovosDominio-Restricao|DnsRs]) :-
%   NovosDominios sao consistentes por no com suas respectivas restricoes
%

consistente_por_no([], []).
consistente_por_no([D-Res|DsRs],[Dn-Res|DnsRs]) :-
  functor(Res, _, N),
  (
    N \== 1 ->
    Dn = D
  ;
     consistente_por_no_primitivo(D, Res, Dn),
  ),
  consistente_por_no(DsRs, DnsRs).


consistente_por_no_primitivo([], _, []).
consistente_por_no_primitivo([D1|Ds], R, [D1|Dn]) :-
  apply(R, [D1]), !,
  consistente_por_no_primitivo(Ds, R, Dn).


consistente_por_no_primitivo([D|Ds], R, Dn) :-
  consistente_por_no_primitivo(Ds, R, Dn).
```

Código 2: Consistência por arco

```
% consistente_por_arco([Dominios-Restricoes|DsRs],
%                       [NovosDominios-Restricoes|DnsRs]) :-
%   NovosDominios sao consistentes por arco com suas respectivas restricoes
%

consistente_por_arco([],[]).
consistente_por_arco([D1-D2-Res|DsRs],[D1n-D2n-Res|DnsRs]) :-
  consistente_por_arco_primitivo([D1-D2], Res, [D1n-D2n]),
  consistente_por_arco(DsRs, _).

consistente_por_arco([_|Cs],[Cns]).
  consistente_por_arco(Cs,Cns).


consistente_por_arco_primitivo([], _, []).
consistente_por_arco_primitivo(_, [], []).
consistente_por_arco_primitivo([D11|D1s]-[D22|D2s], R, [Dx|D1ns]-[Dy|D2ns]) :-
  (
    apply(R, [D11, D22]) ->
    (
      !, consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns),
      Dx = D11, Dy = D22
    )
  ;
    (
      consistente_por_arco_primitivo([D11]-D2s,R, _-_) ->
      (
        !, Dx = D11,
        consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns)
      )
    ;
      (
        consistente_por_arco_primitivo(D1s-[D22],R, _-_) ->
        (
          !,Dy = D22,
          consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns)
        )
      )
    )
  ).

consistente_por_arco_primitivo([D11|D1s]-[D22|D2s], R, D1ns-D2ns) :-
 consistente_por_arco_primitivo(D1s-D2s, R, D1ns-D2ns).
```

`consistente_por_no/2` recebe uma lista de pares "Domínio-Restrição". Se a restrição for unária, ela checa cada valor do domínio. Os valores que resultam em falha são retirados do domínio. `consistente_por_arco/2` tem alguns casos a mais, mas é essencialmente a mesma coisa.

1.2 Consistência por limites[3]

As noções de consistência desenvolvidas acima funcionam bem para restrições em uma ou duas variáveis, mas se quisermos usar algo do tipo para mais variáveis, precisaremos generalizar um pouco:

Consistente por hiper-arco

Definição 1.3. *Uma restrição r/n nas variáveis X_1, ..., X_n é dita* ***consistente por hiper-arco*** *se para cada x_i no domínio de X_i, existem x_j nos domínios de X_j tal que $X_i = x_i, X_j = x_j \ \forall j : 1 \leq j \leq n,\ j \neq i$. Uma restrição composta é dita consistente por hiper-arco se cada uma de suas restrições o é.*

Infelizmente, manter consistência por hiper-arco é algo caro demais para se fazer em um problema geral. Para encontrarmos uma nova checagem de consistência realmente útil, precisaremos restringir o domínio com que lidamos.

CSP aritmético

Dizemos que um CSP é aritmético se o domínio de cada variável é uma união finita de intervalos finitos de números inteiros e se as restrições são aritméticas. Muitos CSPs podem ser modelados como aritméticos de forma natural, e muitos outros podem ser transformados em CSPs aritméticos por uma mudança de variáveis. Por exemplo, se o problema tem a ver com escolhas, uma mudança de variáveis natural é denotar cada escolha por um número. No problema da coloração do mapa do Cazaquistão, por exemplo, ao invés de denotar as cores como "vermelho", "azul", etc., podemos denotá-las como "1", "2", etc., obtendo resultados equivalentes.

Lidando com CSPs aritméticos, podemos definir a noção de **consistência por limites**. A ideia é limitar o domínio de uma variável por limites inferiores e superiores. As sequintes convenções de notação serão convenientes:

- $min_D(X) :=$ algum $x : y \geq x \forall\ y \in D$;
- $max_D(X) :=$ algum $x : y \leq x \forall\ y \in D$.

Consistente por limites

Definição 1.4. *Uma restrição r/n nas variáveis X_1, ..., X_n é dita consistente por limites se para cada x_i variável de r/n, existem valores reais x_1, ..., x_n tal que $min_D(x_j) \geq x_j \leq max_D(x_j)$ e $X_i = min_D(X_i), X_j = x_j \ \forall j \neq i$ é uma solução de r e outros valores reais x_1, ..., x_n tal que $min_D(x_j) \geq x_j \leq max_D(x_j)$ e $X_i = max_D(X_i), X_j = x_j \ \forall j \neq i$ é uma solução de r e Uma restrição composta é dita consistente por limites se cada uma de suas restrições o é.*

Um método eficiente, dito uma **regra de propagação** pode ser eleborado a partir de uma constatação ilustrada no seguinte exemplo:

Considere a restrição `X = Z + Y`. Ela pode ser escrita nas formas

$$X = Z + Y,\ Y = X - Y,\ Z = X - Y$$

[3] Também conhecido como *bounds consistency*

Podemos ver que:

$$X \geq min_D(Y) + min_D(Z), X \leq max_D(Y) + max_D(Z) \quad (1)$$
$$Y \geq min_D(Y) + min_D(Z), Y \leq max_D(Y) + max_D(Z) \quad (2)$$
$$Z \geq min_D(Y) + min_D(Z), Z \leq max_D(Y) + max_D(Z) \quad (3)$$

Podemos usar essa observação para tentar diminuir os domínios de X, Y e Z. Com essa ideia, obtemos o seguinte programa:

Código 3: Busca

```
bounds_consistent_addition([], []).
bounds_consistent_addition([Dx,Dy,Dz], [Dnx, Dny, Dnz]) :-
  min_member(Dx, Xmin),
  min_member(Dy, Ymin),
  min_member(Dz, Zmin),

  Xm is max(Xmin, Ymin + Zmin),
  XM is min(Xmax, Ymax + Zmax),
  new_domain(Xm, XM, Dx, Dnx),

  Ym is max(Ymin, Xmin - Zmax),
  YM is min(Ymax, Xmax - Zmin),
  new_domain(Ym, YM, Dy, Dny),

  Zm is max(Zmin, Ymin - Ymax),
  ZM is min(Zmax, Xmax - Ymin),
  new_domain(Zm, ZM, Dz, Dnz).


new_domain(Vm, VM, Ds, Dn) :-
  Vm =< VM,
  (member(Vm, D) -> append([Vm], Dn) ; true),
  Vm is Vm + 1,
  new_domain(Vm, VM, D, Dn).
new_domain(_, _, _, []).
```

Observações semelhantes podem ser feitas para outros tipos de restrições aritméticas. Para restrições do tipo $X \neq Z$ e $X \neq min(Z, Y)$, isso é especialmente simples de ser feito. Para restrições não lineares do tipo $X < Z \times Y$, isso pode ser especialmente complicado, ainda mais se Z e Y puderem assumir valores positivos e negativos, mas ainda é fazível.

Como é meio chato escrever uma regra para cada caso de restrição dessas para a manutenção de consistência por limites, o que é mais usual em um sistema que ofereça esse tipo de consistência é que suporte apenas uma quantidade reduzida dessas restrições, sendo as demais transformadas em versões equivalentes às quais essas restrições se apliquem, o que não é difícil de se fazer. Isso está sujeito ao potencial incoveniente de que restrições equivalentes mas escritas

de formas diferentes podem oferecer oportunidades diferentes para a diminuição de domínio de cada restrição e a reescrita pode tornar um domínio que poderia originalmente ser grandemente simplificado, em um que sofra apenas uma pequena alteração.

Apesar disso, a aplicação de consistência por limites frequentemente é útil. Um programa que realiza essa aplicação é simples de se fazer: ele toma cada restrição primitiva e os domínios de suas respectivas variáveis e aplica um algo como o mostrado no código 1.2. Assim, temos um mecanismo de busca incompleto. Torná-lo um mecanismo completo é simples e pode ser feito com a adição do backtracking.

1.3 Consistência generalizada e complexa

Consistência por limites também podem ser aplicadas a restrições em duas ou em uma variável, mas, nesse caso, consistência por arco ou por nó costumam resultar em diminuição maior nos domínios. Um problema geral, no entanto, pode ser composto por uma combinação de restrições de diferente aridade, tornando vantajosa a aplicação de diferentes noções de consistência.

No entanto, um potencial problema com abordagens baseadas em consistência é que elas consideram apenas uma ou um pequeno número de restrições e variáveis por vez, enquanto que frequentemente muitas restrições e variáveis oferecem informações sobre as demais.

Tome, por exemplo, a restrição `X` $\neq$ `Y, Y` $\neq$ `Z, X` $\neq$ `Z`, que é equivalente a dizer que as variáveis X, Y e Z são todas diferentes. Dos métodos de consistência que vimos, o mais indicado a essa restrição é o de consistência em arco, já que $\neq$ é de aridade 2. Mas esse método é muito fraco para restrições de desigualdade, o que significa que, na prática, resolver isso por *backtraking*, que tem um crescimento assintótico exponencial.

Essa dificuldade decorre de não usarmos que o domínio de cada variável, nesse caso, oferece informações sobre os domínios das demais. Essa restrição é tão usada que recebeu o nome especial de `alldiferent/1`[4] em vários sistemas CLP e é, talvez, a restrição mais estudada. Restrições que fazem uso da informação no domínio de muitas variáveis para realizar a atualização de cada domínio são chamadas **restrições complexas**. No cado do *alldifferent/1*, podemos notar que essa restrição é equivalente ao supra-mencionado problema de correspondência bipartida e que existem algoritmos eficientes para lidar com ele.

Assim como nas restrições vistas anteriormente, os domínios das variáveis em alldifferent/1 são importantes na decisão de como resolvê-la. Em particular, se as variáveis são inteiras um algoritmo de propagação baseado em consistência

[4]Na verdade, a aridade dessa restrição pode ser arbitrária, mas, por simplicidade, assumimos que as variáveis que devem ser diferentes entre si estão organizadas em uma lista, tornando a aridade igual a um.

por limites atinge um bom desempenho. Se as variáveis não são inteiras, no entanto, uma algoritmos baseados em consistência por híper-arco podem ser usados. A seguir é apresentada os fundamentos de um tal algoritmo. Esse material é baseado em [4].

1.3.1 Alldifferent

Precisaremos das seguintes definições:

Definição 1.5. *Um **grafo** é uma tupla $G = (V, A)$, de vértices e arestas (V é um conjunto de vértices e A de arestas). Em um grafo não orientado, uma aresta é uma tupla de vértices.*

Dado um grafo $G = (V, E)$, um **pareamento**[5] M em G é um conjunto de arestas cujos vértices aparecem em apenas uma aresta de M (em outras palavras, M é um pareamento em G se os vértices em (V,M) tem no máximo grau 1).

Definição 1.6. *Um pareamento M em G é dito máximo se para todos os pareamentos P de G, $|P| \leq |M|$.*

A teoria de pareamento é relevante para nosso problema porque, para qualquer restrição r/n com variáveis X_j, de domínios D_j, a informação de que $X_j \in D_j$ pode ser expressa por um grafo bipartido $(\cup_{j=1}^{n} X_i \cup (\cup_{i=1}^{n} D_i), E)$, onde $(X_i, d) \in E \Leftrightarrow d \in D_i$. Esse grafo é conhecido como **grafo valor** e pode ser construido em tempo polinomial.

É fácil verque a restrição $alldifferent/n$ tem solução se e somente se existe um pareamento máximo de tamanho n em seu grafo valor. Incidentalmente, existe um algoritmo que, dado um grafo, encontra um pareamento máximo em $O(\sqrt[2]{|X|} \times |E|)$.

Ademais, dado um pareamento máximo, existem algoritmos eficientes que tornam $alldifferent/n$ hiper arco consistente (cada aresta em um pareamento máximo corresponde a uma atribuição de valor a uma variável da restrição).

Para desenvolvermos essas afirmações um pouco melhor, as seguintes definições serão uteis.

Definição 1.7. *Dado um pareamento P em G, um vértice e em G é dito pareado se $e \in P$, ou livre caso contrário.*

Definição 1.8. *Um pareamento P em G é dito uma **cobertura** para os vértices do G se todo v vértice de G pertence a P.*

Definição 1.9. *Dado um pareamento P em G, um **caminho (ou ciclo) alternante** de G é um caminho (ou ciclo) cujos vértices são alternadamente pareadas e livres.*

[5] Também comumente conhecido como *matching*.

Teorema 1.1. *Um vértice pertence a algum pareamento máximo P de tamanho n se, e somente se, para qualquer pareamento máximo P', ele ou pertence a P' ou a um caminho alternante em P' de comprimento par que começa em um vértice livre em P', ou a um ciclo alternante em P' de comprimento par.*[6]

Disso segue que quando uma aresta não pertencer a algum circuito ou caminho alternante, podemos extraí-la do grafo original. Para sabermos se tal ocasião acontece, podemos transformar o grafo em um grafo direcionado bipartido G' da seguinte forma: dado G=(V,A), nas arestas em A que são pareadas com P a aresta é orientada das variáveis para os valores e, nas demais, dos valores ara as variáveis.

Assim, podemos fazer uso do seguinte:

Teorema 1.2. *Todo circuito direcionado de G' tem comprimento par e corresponde a um circuito alternante par de G. Além disso, todo caminho em G' que for ímpar pode ser extendido em um caminho par, que corresponde a um caminho alternante par de G começando em um vértice livre.*

Para achar os caminhos procurados, podemos, então determinar os componentes fortemente conectados de G', assim como oscaminhos direcionados em G' começando num vértice livre.

Um resumo do algoritmo é como se segue[7]:

1. É dada a restrição `alldifferent(X)`, onde $X = [X_1, ..., X_n]$;
2. Construímos então o grafo valor $G = (\cup_{j=1}^{n} X_i \cup (\cup_{i=1}^{n} D_i), E)$;
3. Computamos algum pareamento máximo de G, P;
4. Se $|P| < |X|$, falha;
5. Caso contrário, construa G';
6. Marque as arestas de G' que corresponden a arestas de G pertencentes a P como consistentes;
7. Encontre os componentes fortemente conectados de G' e marque as arestas nesses componentes como consistentes;
8. Encontre os caminhos direcionados que começam em um vértice livre e marque suas arestas como consistentes;
9. Para cada aresta de G' não marcada como consistente, remova a aresta correspondente em G.

[6] Se esse teorema não soa intuitivo, você pode fazer uns desenhos e se convencer de sua veracidade (a prova real não vai ser muito diferente dos desenhos que fizer).

[7] Tem algumas sutilezas nele não comentadas aqui. Para saber mais, veja a bibliografia (em particular, [4] e, em [5], o capítulo *Algorithms for matching*)

1.4 Indexicals

Como deve ter dado para notar, uma operação chave em CLP(FD) é a propagação de restrições.

Em sistemas CLP(FD) iniciais, como o CHIP, as restrições eram primeiro transformadas em termos de forma canônica e, então, executados em um interpretador que, entre outras coisas, realiza a propagação de restrições (veja [2]). Devido ao sucesso na compilação de programas Prolog para a **máquina abstrata de Warren** (ou WAM, da sigla em inglês), que é o tipo de máquina abstrata mais usada na compilação de programas em Prolog, foram feitos esforços para a compilação de restrições em CLP(FD) nos mesmos moldes (sendo, na prática, uma extensão da WAM).

Nesse modelo, as restrições são traduzidas para códigos de baixo nível de modo que formas de propagação especializadas sejam usadas para cada tipo de restrição. Isso foi chamado de "modelo caixa-preta", já que o programador não sabe, a princípio, que tipo de propagação seria realizada em suas restrições. Na prática, esse modelo não se provou flexível o suficiente e foi abandonado.

Uma construção chamada ***indexicals***, mais flexível do que a anterior, foi então criada para auxiliar na compilação de restrições em CLP(FD). O modelo com *indexicals* é dito de "caixa de vidro", denotando seu caráter intermediário entre "o programador dita como as restrições são tratadas" e o seu contrário. No nosso contexto, um *indexical* é algo da forma `X in S`, onde X é uma variável de domínio finito e S é uma expressão.

Por exemplo, uma regra de propagação em consistência por limites para a restrição `X = Y + Z.` em *indexical* é:

X in $min(Y) + min(Z)..max(Y) + max(Z)$

A ideia é descrever o domínio de cada variável como uma função do domínio inicial por meio de construções como *in* e *min(Z)..max(Y)*.

Indexicals não são usados no ECL^iPS^e mas são usados em outros sistemas e algumas outras formas de propagação fazem uso de métodos que tomam *indexicals* por base, por isso o discutimos brevemente por nesta ocasião.

Incidentalmente, os *indexicals* usados aqui provém de um conceito mais geral de *indexical* proveniente da filosofia da linguagem. Para entender esse conceito um pouco melhor, considere a seguinte situação (retirada de [3]):

Digamos que existem dois irmãos gêmeos, um que só diz a verdade e outro que só mente. Além disso, o primeiro irmão, que só diz a verdade também tem uma crença perfeita do que é ou não verdade (isto é, todas as proposições que são verdadeiras ele crê serem verdade e análogamente o contrário). Em contra-partida, o outro irmão tem uma crença completamente imperfeita do que é verdade (o que é verdade ele crê não o ser e analogamente o contrário). O interessante é que, ao serem feitos a mesma pergunta, os dois irmãos dariam a mesma resposta. Considerando essa situação e tendo em mente que o irmão

mentiroso o deve dinheiro, um logicista pergunta a um outro "Você acha que é possível, fazendo perguntas de sim ou não, descobrir se um irmão é o mentiroso ou o verdadeiro?", ao que ele responde "Claramente não, já que eles dariam as mesmas respostas às mesmas questões". Você acha que o segundo logicista estava correto?
Na verdade, não estava: Considere a pergunta "Você é o que diz a verdade?". Se ele o for, dirá que é. Se não o for, crerá que o é e dirá que não o é. Apesar disso, o segundo logicista estava correto ao dizer que eles dariam as mesmas respostas às mesmas questões: o "você" em "Você é o que diz a verdade?" se refere a "coisas" diferentes se usados com pessoas diferentes, então a pergunta feita ao mentiroso seria fundamentalmente diferente da feita ao não mentiroso. Aqui, "você" é um *indexical*.

O ponto é que um mesmo *indexical* pode significar coisas diferentes em contextos diferentes (na verdade, do ponto de vista filosófico, é isso que o define como sendo um *indexical*). No exemplo de *indexical* acima, os domínios iniciais de X, Y e Z poderiam mudar significativamente o que aquele *indexical* significa.

Leituras adicionais

[1] E. Tsang (1930), "Foundations of Constraint Satisfaction", Academic Press.

[2] Neng-Fa Zhou (2006), "Programming Finite-Domain Constraint Propagator in Action Rules", Theory and Practice of Logic Programming, Vol. 6, No. 5, pp.483-508, 2006

[3] Smullyan, Raymond, "5000 B.C. and Other Philosophical Fantasies"

[4] Basileos Anastasatos "Propagation Algorithms for the Alldifferent Constraint"

[5] Christos Papadimitriou (1998), "Combinatorial Optimization, Algorithms and Complexity", Dover Publications