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 (digamos, igual a 5) de cores, precisamos colorir um mapa (digamos, o mapa do Cazaquistão) de modo que nenhuma das regiões do mapa receba a mesma cor que outra região 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 de restrição `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[2].

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

Essa classe de problemas 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.

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)[3]. 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.

Essa forma de resolução é dita ser baseada em consistência, porque ele funciona propagando informações dos domínios de cada variável para os demais,

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

[2] No geral, restrições CSPs em restrições n-árias podem ser representados como um multigrafo.

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

tornando-os “consistentes” entre si.

Definição 1.1. *Uma restrição r/n é dita* ***consistente por nó*** *com um domínio D 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 `apply/2` usado foi definido no Capítulo 6.

Código 1: Consistência por nó

```
% consistente_por_no([Dominios-Restricoes|DsRs],
%                    [NovosDominios-Restricoes|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).
```

1.2 Consistência por limites[4]

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:

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 $X_i = x_i, X_j = x_j \ \forall j : 1 \geq j \leq n$, $j \neq i$. Uma restrição composta é dita consistente por hiper-arco se cada uma de suas restrições o é.*

Consistente por hiper-arco

Infelizmente, manter a 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 (mencionado acima), 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:

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

Definição 1.4. *Uma restrição r/n nas variáveis X_1, ..., X_n é dita consistente por limites se*

Consistente por limites

- *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.

[4] Mais conhecido como *bounds consistency*

Um método eficiente, que é 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

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

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 custoso, onde Z e Y podem assumir valores positivos e negativos, mas ainda pode ser feito.

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 é que um sistema que ofereça esse tipo de consistência 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 um domínio que poderia originalmente ser grandemente simplificado, 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 pode

Leituras adicionais

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