1 Heurísticas

Nos métodos de busca vistos até agora, a ordem de variáveis adotada é a ordem apresentada na lista e a ordem de valores para cada variável é fixa. Para problemas maiores, pode ser mais conveniente considerar ordenações diferentes para cada tipo de problema. Essas ordenações são geralmente escolhidas de forma heurística e, por isso, são chamadas ordenações heurísticas. Existem, naturalmente, duas classes de ordenações heurísticas: ordenação heurística de variáveis e de valores.

Mas, antes de prosseguirmos, será útil falarmos sobre vetores, que serão utilizados nos programas a seguir.

1.1 Vetores

Listas são interessantes, mas, eventualmente, podemos querer realizar acesso em tempo constante aos seus membros, o que não é possível. Em outras linguagens, esse tipo de acesso é por costume realizado através de vetores. Vetores existem em Prolog padrão, de certa forma. Como os argumentos de funtores podem ser acessados em tempo constante por meio do funtor `arg/3`, a princípio, um funtor qualquer poderia ser usado como um vetor, o acesso aos membros do qual feitos por meio de `arg/3`. Essa abordagem, no entanto, não só não parece muito elegante, como é pouco prática. Se temos algo que chamamos de vetor, gostaríamos de poder fazer algo como `Vetor =/= 4`, por exemplo, o que com os meios do Prolog padrão não é possível.

Para esse tipo de situação, no ECL^iPS^eexiste uma implementação de vetores como um açucar sintático (semelhante ao que vimos na implementação de listas), permitindo justamente esse tipo de utilizaçãoa. Em particular, um vetor, é escrito como uma variável seguida do funtor de lista (como em `Var[4]`), ou, se anônimo, como `[](A, B, C)` (note que, assim, a representação de um vetor ou de uma lista vazia é a mesma) Para criar "vetores", possivelmente multidimensionais (na realidade, matrizes, o que justifica os ""), foi construído o predicado `dim/2`, que pode ser usado ou para criar vetores ou para extrair sua dimensão.

1.2 As n-rainhas

Analisaremos o efeito de algumas heurísticas pela resolução do *problema das n-rainhas* quando $n \geq 3$[1]. É um problema por restrições tão bem conhecido que um estudante de programação por restrições poderia (e com razão) se sentir enganado se terminasse um livro sobre o tema sem saber como resolver esse problema em especial. A discussão a seguir é baseada, em grande parte, em [1].

[1] Quando n = 2 ou n = 3, o problema não tem solução

Para quem não conhece, o *problema das n-rainhas* é o problema de alocar n rainhas em um tabuleiro como o de xadrez, mas $n \times n$, com a restrição de que as rainhas não tem permissão para se atacarem segundo as regras usuais de xadrez.

Existe um algoritmo polinomial para a resolução desse problema (que não veremos aqui), mas o que veremos muitas vezes é mais conveniente e até mais rápido. Antes de considerarmos o papel das heurística, vejamos como é uma solução mais simples. Notamos primeiro que, como temos n rainhas, precisa existir uma rainha em cada linha e em cada coluna. Ou seja, para cada coluna (ou cada linha) só precisamos indicar em qual linha (coluna) a rainha está, o que significa que a solução requer um lista[2] de n posições, não uma matriz (como alguém podria supor a princípio). O domínio de cada posição na lista é o intervalo inteiro $[1, n]$ e as restrições são que só pode existir uma rainha por linha (ou por coluna), o que se traduz na restrição `alldifferent(Cols)`, em que *Cols* é o vetor de colunas. Ademais, só pode existir uma rainha por linha, o que se traduz nas restrições X_i `-` X_j `\=\` `I - J` e X_i `-` X_j `\=\` `J - I`, em que X_i e X_j são membros distintos da lista de colunas (ou linhas).

```
%% rainhas(-Rainhas, ++Numero)
% Rainhas contém a posição de cada rainha por coluna
%

:- lib(ic)

rainhas(Rainhas, Numero) :-
  dim(RainhaStruct,[Numero]),
  restricoes(RainhaStruct, Numero),
  struct_para_lista(RainhaStruct, Rainhas),
  busca(Rainhas).

restricoes(RainhaStruct, Numero) :-
  ( for(I,1,Numero),
    param(RainhaStruct,Numero)
  do
    RainhaStruct[I] :: 1..Numero,
    (for(J,1,I-1),
      param(I,RainhaStruct)
    do
      RainhaStruct[I] #\= RainhaStruct[J],
      RainhaStruct[I]-RainhaStruct[J] #\= I-J,
      RainhaStruct[I]-RainhaStruct[J] #\= J-I
    )
  ).
```

[2] Ou um vetor.

```
struct_para_lista(Struct, Lista) :-
  ( foreacharg(Arg,Struct),
    foreach(Var,Lista)
  do
    Var = Arg
  ).
```

Esse programa primeiro declara o vetor *RainhaStruct*, impõe a ele as restrições mencionadas, transforma o vetor em lista e realiza a busca (como explicada no Capítulo 10).O programa, no computador do autor, consegue resolver o problema por esse programa para N até 29, mas, adotando um tempo limite de 50 segundos, N = 30 já está fora de mão.

Alguém poderia dizer que isso ocorre porque esse código não foi feito de maneira inteligente, já que a assimetria do problema, presente no fato de que a alocação de uma rainha em uma coluna central, por exemplo, resulta em maiores restrições nas posições das outras rainhas, não está representada no código. Esse realmente é o caso e, para fazê-lo deixar de ser, criamos uma nova versão do programa.

1.3 Ordenação de variáveis

A alteração consiste na adição de um argumento Heur (de heurística) no predicado `rainhas/2` (que se torna `rainhas/3`). Esse predicado nos diz qual a heurística usada para ordenar as variáveis. Podemos definir o `rainhas/2`, do exemplo passado, como um `rainhas/3` no qual o último argumento é "naive" (indicando que a heurística usada é uma, por assim dizer, boba).

```
busca(Rainhas, naive) :- labeling(Rainhas).
```

Uma heurística mais esperta nesta situação seria começar a testar as variáveis do meio. Chamaremos essa heurítica de *middle_out*: ela faz com que a busca comece do meio e vá alternando entre os vizinhos do lado esquerdo e direito, a partir do meio.

```
:- lib(lists).

busca(Rainhas, middle_out) :-
  middle_out(Rainhas, RainhasDeSaida),
  labeling(RainhasDeSaida).

middle_out(Lista, ListaDeSaida) :-
  halve(Lista, PrimeiraMetade, UltimaMetade),
  reverse(PrimeiraMetade, RevPrimeiraMetade),
  splice(UltimaMetade, RevPrimeiraMetade, ListaDeSaida).
```

Aqui usamos a biblioteca *lists*, que contém, entre outras coisas, `halve/3`, `reverse/2` e `splice/3`. O que o primeiro e o segundo desses predicados faz deve ser claro. O que o terceiro faz é juntar em *ListaDeSaida* as *UltimaMetade* e *RevPrimeiraMetade* colocando um membro de cada alternadamente (como uma função *merge* do *merge_sort*).

À parte dessa heurística, existe a *first_fail*, que seleciona como a próxima variável a que tem menos valores restantes no domínio. Para implementar essa heurística, usaremos o predicado da biblioteca *ic* `delete(-X, +List, -R, ++Arg, ++Select)`, que remove uma variável X da lista de variáveis *List*, deixando o resultado em R. O parâmetro *Arg* indica se a lista é uma lista de fato ou um funtor (no segundo caso, os argumentos do funtor são tratados como se fossem elementos de uma lista), e o *Select* indica o parâmetro de seleção (uma lista dos parâmetros possíveis pode ser encontrada na documentação do ECL^iPS^e). No caso, nosso parâmetro é *first_fail*. Nossa busca com o *first_fail* fica da seguinte forma:

```
busca(Lista, first_fail) :-
  ( fromto(Lista, Vars, Resto, [])
  do
    delete(Var, Vars, Resto, 0, first_fail),
    indomain(Var)
  ).
```

o parâmetro 0 indica que lidamos com uma lista de fato.

A experiência prática indica que, para instâncias pequenas (para N pequenos), a diferença entre usar o *fisrt_fail* ou não é pequena. Para N grandes, entretanto, a diferença é visível.

Não temos motivos *a priori* para não usar o *first_fail* e o *middle_out* juntos. Chamaremos essa heurística de *moff*:

```
busca(Lista, moff) :-
  middle_out(Lista, ListaDeSaida),
  ( fromto(ListaDeSaida, Vars, Resto, [])
  do
    delete(Var, Vars, Resto, 0, first_fail),
    indomain(Var)
  ).
```

1.4 Heurísticas de ordenamento de valor

A mesma observação usada para a heurística *middle_out* vale para a ordem dos valores escolhidos: a escolha de valores próximos ao centro restringem mais os valores de outras variáveis e tem a chance de falhar mais cedo. O

ECL^iPS^eoferece um predicado `indomain/2` que nos ajuda nisso: seu segundo argumento nos dá um certo controle sobre a ordem em que os valores são testados. Em particular, `indomain(Var, middle)` começa a enumeração das variáveis pelo meio do domínio de *Var*. Existem também `indomain(Vars, min)` e `indomain(Vars, max)` que começam pelos menores valores e maiores, respectivamente (usando a ordem do domínio, pela qual "menor" é o que vem antes).

Podemos assim combinar nossa heurística anterior (*moff*) com esta (que chamaremos de *moffmo*):

```
busca(Lista, moffmo) :-
  middle_out(Lista, ListaDeSaida),
  ( fromto(ListaDeSaida, Vars, Resto, [])
  do
    delete(Var, Vars, Resto, 0, first_fail),
    indomain(Var)
  ).
```

Na tabela a seguir consta a quantidade de *backtrackings* por heurística para alguns valores de N:

Figura 1: Retirado de [1]

	naive	middle_out	first_fail	moff	moffmo
8-queens	10	0	10	0	3
16-queens	542	17	3	0	3
32-queens	—	–	4	1	7
75-queens	—	–	818	719	0
120-queens	—	–	—	—	0

1.5 Outras Considerações

O problema das n-rainhas é simétrico em que se uma solução tem a enésima rainha no emésimo quadrado da coluna i, outra solução teria a enésima rainha na emésimo quadrado da linha i. Perceba que uma solução é equivalente à outra, a menos de rotaçõe s de 90º. Assim, quando a heurística de ordenação de variáveis *middle_out* sucede sem *backtrackings*, a heurística de ordenação de valores `indomain(Var, middle)` também deveria. Percebendo isso, podemos criar uma heurística de ordenação de valores que faz *backtrackings* como a *middle_out*. Depois de aplicada, uma "rotação" deve nos mostrar as mesmas soluções e na mesma ordem.

Para tanto, fazemos pequenas modificações no nosso programa. No lugar de iterar sobre a lista de variáveis, iteraremos sobre os valores do domínio. Ademais, no lugar de selecionar um valor para a variável atual (pelo `indomain/1`), selecionaremos uma variável para um valor (pelo `member/2`). Por fim, "rodamos" o resultado, para que os resultados produzidos sejam os mesmos produzidos pelo *middle_out*. Um código representando o que foi discutido é como segue:

```
rainhas(rotate, Rainhas, Numero) :-
  dim(RainhaStruct,[Numero]),
  constraints(RainhaStruct, Numero),
  struct_para_lista(RainhaStruct, QList),
  busca(QLista, rotate),
  rotate(RainhaStruct, Rainhas).

busca(QLista, rotate) :-
  middle_out_dom(QLista, MOutDom),
  ( foreach(Val, MOutDom),
    param(QLista)
  do
    member(Val, QLista)
  ).

middle_out_dom([Q | _], MOutDom) :-
  get_domain_as_lista(Q,OrigDom),
  middle_out(OrigDom, MOutDom).

rotate(RainhaStruct, Rainhas) :-
  dim(RainhaStruct,[N]),
  dim(RRainhas,[N]),
  ( foreachelem(Q,RainhaStruct,[I]),
    param(RRainhas)
  do
    subscript(RRainhas,[Q],I)
  ),
  struct_para_lista(RRainhas, Rainhas).
```

Curiosamente, os resultados para o *rotate* são muito melhores do que para *middle_out*, como pode ser visto na seguinte tabela, mostrando a quantidade de *backtrackings* por N e por heurística:

Isso ocorre porque o comportamento da propagação de restrições não é o mesmo para linhas e colunas. Se os valores para uma coluna são excluídos, exceto um, essa variável é instanciada ao velor restante. Esse é o comportamento exibido por *rotate*.

No entanto, no comportamento exibido por *middle_out*, se todos os valores

Figura 2: Retirado de [1]

	middle_out	rotate
8-queens	0	0
16-queens	17	0
24-queens	194	12
27-queens	1161	18
29-queens	25120	136

de uma linha são excluídos exceto um, a propagação de restrições não instancia a variável. Isso ocorre porque as variáveis são representadas por colunas.

Uma forma de conseguir a mesma quantidade de propagação é usar uma redundância na formulação do problema. Neste caso, poderíamos adicionar outro tabuleiro ligado ao primeiro de modo que, se uma rainha i é posta no quadrado j de um tabuleiro, no outro, uma rainha j é posta no quadrado i.

1.6 O Predicado Search

Espero que a leitora não fique triste ao saber que grande parte do trabalho desenvolvido até agora poderia ser substituído pelo uso do predicado `search/6`. Ele tem 6 argumentos: *search(+Lista, ++Arg, ++Select, +Escolha, ++Metodo, +Opcao)*. O primeiro argumento, *Lista*, é uma lista de variáveis de decisão (as variáveis do problema) quando *Arg* é 0, ou uma lista de funtores compostos quando $Arg > 0$. Assumiremos que *Arg* é 0. O argumento *Select* representa uma heurística (feita pelo usuário ou não) de ordenação de variáveis, enquanto que, o *Escolha*, uma de ordenação de valores. Pelo *Metodo*, podemos selecionar a "forma" pela qual a busca deve ser feita: algumas opções são *complete* (que realiza uma busca por todos os valores), e *sbds* (que faz uso da biblioteca SBDS de quebra de simetria para excluir partes simétricas da árvore de busca)[3]. O argumento *Opcao* é usado para passar parâmetros adicionais[4], algumas das quais são *node(daVinci)*, para criar um desenho da árvore de busca usando a ferramenta de desenho daVinci, e *backtracks(-N)*, que nos dá em N a quantidade de *backtrackings* que ocorreram durante a busca.

[3]Recomendamos a consulta ao manual do ECL^iPS^e para mais detalhes sobre esses e outros métodos de busca

[4]Novamente, é recomendado checar o manual do ECL^iPS^e para mais informações

Leituras adicionais

[1] Krzysztof R. Apt and Mark Wallace "Constraint Logic Programming using ECLiPSe" Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK