Utilização de DCGs

- **DCG**: Gramática de Cláusulas Definidas (Definite Clause Grammar) é uma linguagem formal para definicção de outras linguagens. Baseada em cláusulas de Horn.
- Exemplo:

```
programa --> regra; fato.
```

- Muito utilizada para definir gramáticas e para processamento de linguagem natural.
- Programa escrito com sintaxe DCG normalmente avalia se uma sentença está ou não sintaticamente correta.

Utilização de DCGs

- Representação interna de um símbolo não terminal em Prolog: acrescentar dois argumentos a cada literal da cláusula, de forma transitiva. Ex:

```
DCG: sent --> frase_nominal, frase_verbal, complemento.
Prolog: sent(S0,S) :- frase_nominal(S0,S1),
                      frase_verbal(S1,S2),
                      complemento(S2,S).
```

Utilização de DCGs

- Representação interna de um símbolo terminal em Prolog: fato. Classifica o termo do dicionário de palavras e devolve o resto para ser classificado pelo restante da análise sintática. Ex:

  ```
  DCG: artigo --> [a].
  Prolog: artigo([a|R],R).
  ```

- Ações semânticas podem ser adicionadas às cláusulas DCGs. Desta forma podemos ter uma mistura de código DCG com código Prolog. Por exemplo, ao fazer análise sintática de uma linguagem de programação, podemos querer gerar tabelas de símbolos, tabelas de variáveis, gerar código, etc. Estas ações não fazem parte da gramática. São ações que devem ser tomadas cada vez que um símbolo recebe uma determinada classificação.

Utilização de DCGs

- Exemplo

```
DCG: constante(Dic,I) --> [segunda],
                          {lookup(segunda,Dic,I)}.
Prolog: constante(Dic,I,[segunda|R],R) :-
                          lookup(segunda,Dic,I).
```

- Neste trecho de programa, após o símbolo 'segunda' ser classificado como 'constante', deseja-se inserir esta constante numa tabela (Dic) e retornar um índice para esta constante na tabela (I). Esta ação é representada em DCG com o predicado entre chaves.

Utilizando DCGs: Análise sintática de sentenças

```
sentence(sentence(NP,VP)) -->
        noun_phrase(NP),
        verb_phrase(VP).

noun_phrase(np(D,N,C)) -->
        determiner(D),
        noun(N),
        rel_clause(C).
noun_phrase(np(PN)) -->
        proper_noun(PN).
```

Utilizando DCGs: Análise sintática de sentenças

```
verb_phrase(vp(TV,NP)) -->
        trans_verb(TV),
        noun_phrase(NP).
verb_phrase(vp(IT)) -->
        intrans_verb(IT).

rel_clause(rc(that,VP)) -->
        [that],
        verb_phrase(VP).
rel_clause(rc([])) --> [].
```

Utilizando DCGs: Análise sintática de sentenças

```
determiner(det(every)) --> [every].
determiner(det(a)) --> [a].

noun(noun(man)) --> [man].
noun(noun(woman)) --> [woman].

proper_noun(pn(john)) --> [john].

trans_verb(tv(loves)) --> [loves].

intrans_verb(iv(lives)) --> [lives].
```

Utilizando DCGs: Transformação de sentenças

```
:-op(500,xfy,&).
:-op(600,xfy,'->').

sentence(P) -->
        noun_phrase(X,P1,P),
        verb_phrase(X,P1).

noun_phrase(X,P1,P) -->
        determiner(X,P2,P1,P),
        noun(X,P3),
        rel_clause(X,P3,P2).
noun_phrase(X,P,P) -->
        proper_noun(X).
```

Utilizando DCGs: Transformação de sentenças

```
verb_phrase(X,P) -->
        trans_verb(X,Y,P1),
        noun_phrase(Y,P1,P).
verb_phrase(X,P) -->
        intrans_verb(X,P).

rel_clause(X,P1,(P1&P2)) -->
        [that],
        verb_phrase(X,P2).
rel_clause(_,P,P) --> [].
```

Utilizando DCGs: Transformação de sentenças

```
determiner(X,P1,P2,all(X,(P1->P2))) --> [every].
determiner(X,P1,P2,exists(X,(P1&P2))) --> [a].

noun(X,man(X)) --> [man].
noun(X,woman(X)) --> [woman].

proper_noun(john) --> [john].

trans_verb(X,Y,loves(X,Y)) --> [loves].

intrans_verb(X,lives(X)) --> [lives].
```

Problemas de Busca

- Problema geral: encontrar um caminho de uma situação inicial (Si) para uma situação final (Sf), de forma que todos os caminhos intermediários sejam 'legais' (ou válidos).
- Exemplo: problema das pilhas de blocos:
  - Podemos mover um único bloco de cada vez.
  - Um bloco só pode se mover se não houver nenhum bloco acima dele.
- Para ecnontrar um esquema, precisamos encontrar uma seqüência de movimentos que resulte na transformação requerida.

Problemas de Busca

- Exemplo:

```
+---+          +---+
| C |          | A |
+---+          +---+
| A |  ===>    | B |
+---+   ??     +---+
| B |          | C |
+---+          +---+
/////          /////
```

- **qual** caminho seguir para chegar da config 1 para config 2?
- **como** seguir este caminho?

Problemas de Busca

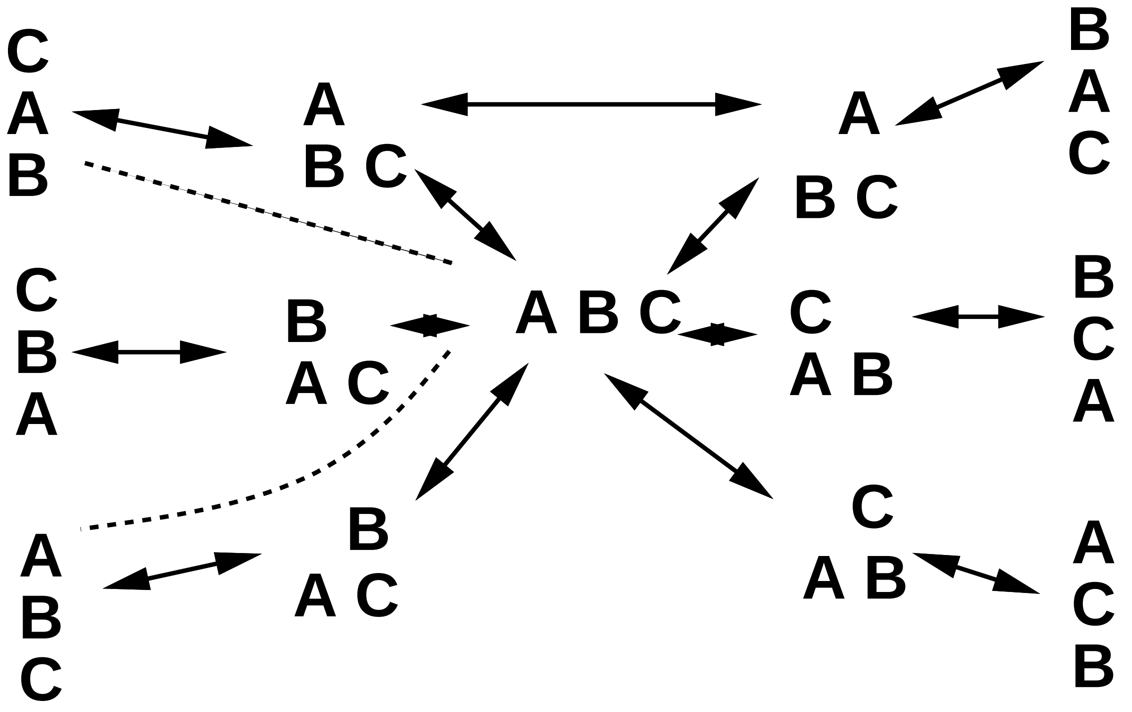
- Problema se resume a explora vários caminhos alternativos até encontrar a solução. Da figura, podemos ter as seguintes alternativas, após colocar C na superfície:
  - colocar A na superfície, **ou**
  - colocar A em cima de C, **ou**
  - colocar C em cima de A.
- Dois tipos de conceito:
  - situações.
  - movimentos possíveis, ou ações que transformam as situações em outras situações.

Problemas de Busca

- Dois tipos de conceito formam um grafo direcionado: *espaço de estados* ou *espaço de busca.*
- Os nós do grafo correspondem a situações e os arcos correspondem a transições legais entre estados.
- equivalente a encontrar um caminho entre a situação inicial (ou nó inicial do grafo, vide busca de caminho num grafo) e uma situação final específica, também chamado de nó objetivo do grafo.

Problemas de Busca

Ex:



Problemas de Busca

- Resumindo:
  - espaço de estados.
  - nó inicial.
  - objetivo a ser alcançado (nó final).
- Problema de otimização: encontrar o caminho de menor custo.

Problemas de Busca

- Representação de um espaço de estados em prolog: `s(X,Y)` ou `s(X,Y,C)`, com C sendo o custo para passar do estado X para o estado Y.
- `s(X,Y)` é verdadeiro se há um movimento possível no espaço de estados, de X para Y (dentro das características do problema).
- Uma situação no problema dos blocos, pode ser representada por uma lista de pilhas. Cada pilha pode ser representada por uma lista de blocos na qual a cabeça da lista é o bloco que está no topo da lista.

Problemas de Busca

- Estado inicial: `[[c,a,b],[],[]]`
- Situação final: qualquer arranjo que contenha uma pilha com todos os blocos ordenados:
  - `[[a,b,c],[],[]]`
  - `[[],[a,b,c],[]]`
  - `[[],[],[a,b,c]]`
- Dado um estado, para encontrarmos o próximo estado, utilizamos a seguinte regra: Sit2 é o próximo estado após Sit1 se existem duas pilhas, St1 e St2, em Sit1, e o bloco no topo de St1 pode ser movido para St2.

Problemas de Busca

```
% mv Top1 to St2 em Sit2
s(Stacks,[St1,[Top1|St2]|Otherstacks]) :-
% [Top1|St1] e' uma pilha em Sit1
   del([Top1|St1],Stacks,Stacks1),
% St2 e' uma pilha em Sit1
   del(St2,Stacks1,Otherstacks).
```

Problemas de Busca

- Condição objetivo:

  `goal(Estado) :- pertence([a,b,c],Estado).`

- Predicado de busca (pode ser implementado com qualquer algoritmo de busca:

  `solve(Inicio,Fim).`

- Consulta: `?- solve([[c,a,b],[],[]],Solucao).`

Problemas de Busca

- Busca em profundidade: (similar ao programa de busca por um caminho no grafo)

  ```
  solve(N,[N]) :- goal(N).
  solve(N,[N|Sol1]) :-
      s(N,N1),        % implem. disto depende do prob.
      solve(N1,Sol1).
  ```

- Este programa difere do programa que implementa a busca no grafo, porque não evita ciclos.

Problemas de Busca

- Busca iterativa em profundidade: (precisa de argumento extra que é o limite de profundidade)

```
solve(N,[N],_) :- goal(N).
solve(N,[N|Sol1],ProfMax) :-
    ProfMax > 0,
    s(N,N1),
    NewMax is ProfMax - 1,
    solve(N1,Sol1,NewMax).
```

- Este programa também não evita ciclos até o limite de profundidade.

Problemas de Busca

- Busca em largura:

```
bfs(Inicio,Fim) :- solve([[Inicio]],Fim).
solve([[N|Path]|_],[N|Path]) :- goal(N).
solve([Path|Paths],Solucao) :-
  extend(Path,NewPaths),
  conc(Paths,NewPaths,Paths1),
  solve(Paths1,Solucao).
extend([Node|Path],NewPaths) :-
  bagof([NewNode,Node|Path],
   (s(Node,NewNode), \+ pertence(NewNode,[Node|Path])),
    NewPaths), !.
extend(Path,_). % no nao tem sucessor.
```

- Este programa também não evita ciclos até o limite de profundidade.