

More Clever Searching

```
techer(stephen).
techer(todd).
nerd(X) :- techer(X).
?- nerd(X).
```

"Tell me about everybody who's provably a nerd."

As before, start with query. Rule only interesting thing.

Unifying `nerd(X)` with `nerd(X)` is vacuously true, so we need to establish `techer(X)`.

More Clever Searching

```
techer(stephen).
techer(todd).
nerd(X) :- techer(X).
?- nerd(X).
```

Unifying `techer(X)` with `techer(stephen)` succeeds, setting `X = stephen`, but we're not done yet.

Unifying `techer(X)` with `techer(todd)` also succeeds, setting `X = todd`, but we're still not done.

Unifying `techer(X)` with `nerd(X) :-` fails, returning no.

More Clever Searching

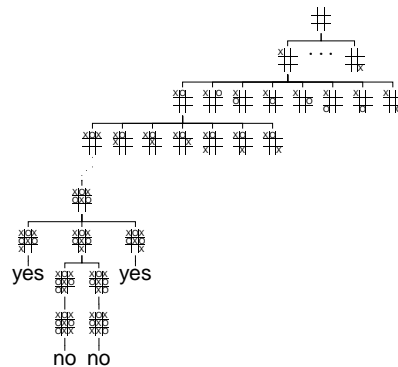
```
> ~/tmp/beta-prolog/bp
Beta-Prolog Version 1.2 (C) 1990-1994.
| ?- [user].
|:techer(stephen).
|:techer(todd).
|:nerd(X) :- techer(X).
|:^D
yes
| ?- nerd(X).
X = stephen?;
X = todd?;
no
| ?-
```

Order Matters

```
> ~/tmp/beta-prolog/bp
Beta-Prolog Version 1.2 (C) 1990-1994.
| ?- [user].
|:techer(todd).
|:techer(stephen).
|:nerd(X) :- techer(X).
|:^D
yes
| ?- nerd(X).
X = todd?;
X = stephen?;
no
| ?-
```

Todd returned first

Searching and Backtracking



The Prolog Environment

Database consists of clauses.

Each clause consists of terms, which may be constants, variables, or structures.

Constants: `foo my_Const + 1.43`

Variables: `X Y Everybody My_var`

Structures: `rainy(rochester)`
`teaches(edwards, cs4115)`

Structures and Functors

A structure consists of a functor followed by an open parenthesis, a list of comma-separated terms, and a close parenthesis:

```

"Functor"
  |
  v
bin_tree( foo, bin_tree(bar, glarch) )
          ^
          |
          paren must follow immediately

```

What's a structure? Whatever you like.

A predicate `nerd(stephen)`

A relationship `teaches(edwards, cs4115)`

A data structure `bin(+, bin(-, 1, 3), 4)`

Unification

Part of the search procedure that matches patterns.

The search attempts to match a goal with a rule in the database by unifying them.

Recursive rules:

- A constant only unifies with itself
- Two structures unify if they have the same functor, the same number of arguments, and the corresponding arguments unify
- A variable unifies with anything but forces an equivalence

Unification Examples

The = operator checks whether two structures unify:

```

| ?- a = a.
yes
| ?- a = b.
no
| ?- 5.3 = a.
no
| ?- 5.3 = x.
X = 5.3?;
no
| ?- foo(a,X) = foo(X,b).
no
| ?- foo(a,X) = foo(X,a).
X = a?;
no
| ?- foo(X,b) = foo(a,Y).
X = a
Y = b?;
no
| ?- foo(X,a,X) = foo(b,a,c).
no

```

% Constant unifies with itself
% Mismatched constants
% Mismatched constants
% Variables unify
% X=a required, but inconsistent
% X=a is consistent
% X=a, then b=Y
% X=b required, but inconsistent

The Searching Algorithm

```
search(goal g, variables e)
for each clause h :- t1, ..., tn in the database
    e = unify(g, h, e)
    if successful,
        for each term t1, ..., tn,
            e = search(tk, e)
        if all successful, return e
return no
```

Order matters

```
search(goal g, variables e)
for each clause h :- t1, ..., tn in the database
    e = unify(g, h, e)
    if successful,
        for each term t1, ..., tn,
            e = search(tk, e)
        if all successful, return e
return no
```

In the order they appear

In the order they appear

Order Affects Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :-
    edge(X, Z), path(Z, Y).
```

```
path(a,a)
|
path(a,a)=path(X,X)
|
X=a
|
yes
```

Consider the query

```
?- path(a, a).
```

Good programming practice: Put the easily-satisfied clauses first.

Order Affect Efficiency

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
    edge(X, Z), path(Z, Y).
path(X, X).
```

```
path(a,a)
|
path(a,a)=path(X,Y)
|
X=a Y=a
|
edge(a,Z)
|
edge(a,Z)=edge(a,b)
|
Z=b
|
path(b,a)
|
...
```

Consider the query

```
?- path(a, a).
```

Will eventually produce the right answer, but will spend much more time doing so.

Order can cause Infinite Recursion

```
edge(a, b). edge(b, c).
edge(c, d). edge(d, e).
edge(b, e). edge(d, f).
path(X, Y) :-
    path(X, Z), edge(Z, Y).
path(X, X).
```

```
path(a,a) ← Goal
|
path(a,a)=path(X,Y) ← Unify
|
Subgoal X=a Y=a ← implies
|
path(a,Z) edge(Z,a)
|
path(a,Z)=path(X,Y)
|
X=a Y=Z
|
path(a,Z) edge(Z,a)
|
path(a,Z)=path(X,Y)
|
X=a Y=Z
|
...
```

Consider the query

```
?- path(a, a).
```

Like LL(k) grammars.

Prolog as an Imperative Language

A declarative statement such as

P if Q and R and S

can also be interpreted procedurally as

To solve P, solve Q, then R, then S.

This is the problem with the last example.

```
path(X, Y) :- path(X, Z), edge(Z, Y).
```

"To solve P, solve P..."

Prolog as an Imperative Language

```
go :- print(hello_), print(world).
```

```
?- go.
hello_world
yes
```

Cuts

Ways to shape the behavior of the search:

- Modify clause and term order.
 - Can affect efficiency, termination.
- "Cuts"
 - Explicitly forbidding further backtracking.

Cuts

When the search reaches a cut (!), it does no more backtracking.

```
techer(stephen) :- !.
techer(todd).
nerd(X) :- techer(X).
```

```
?- nerd(X).
X= stephen?;
no
```

Controlling Search Order

Prolog's ability to control search order is crude, yet often critical for both efficiency and termination.

- Clause order
- Term order
- Cuts

Often very difficult to force the search algorithm to do what you want.

Elegant Solution Often Less Efficient

Natural definition of sorting is inefficient:

```
sort(L1, L2) :- permute(L1, L2), sorted(L2).
permute([], []).
permute(L, [H|T]) :-
    append(P, [H|S], L), append(P, S, W), permute(W, T).
```

Instead, need to make algorithm more explicit:

```
qsort([], []).
qsort([A|L1, L2] :- part(A, L1, P1, S1),
    qsort(P1, P2), qsort(S1, S2), append(P2, [A|S2], L2).
part(A, [], [], []).
part(A, [H|T], [H|P], S) :- A >= H, part(A, T, P S).
part(A, [H|T], P, [H|S]) :- A < H, part(A, T, P S).
```

Prolog's Failings

Interesting experiment, and probably perfectly-suited if your problem happens to require an AI-style search.

Problem is that if your peg is round, Prolog's square hole is difficult to shape.

No known algorithm is sufficiently clever to do smart searches in all cases.

Devising clever search algorithms is hardly automated: people get PhDs for it.