

# Artificial Intelligence: Logic Programming IV

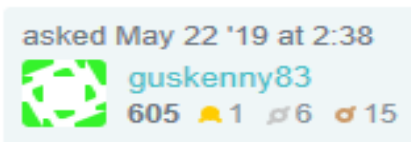
Oliver Ray

[bristol.ac.uk](http://bristol.ac.uk)



# From Prolog to meta-logical programming

- The last week introduced the **syntax** of **Prolog** along with its **declarative** and **procedural** semantics - culminating in the visualization of Prolog computations via **proof trees** and **search trees**
- This lecture shows how exploration of Prolog search space can be controlled with two key mechanisms: the **cut** operator “!” and **meta-interpreters**
- We will motivate some prototypical use cases of cut and look at some higher-level operators (such as \+) which are defined in terms of it (**EXAMINABLE**)
- And we will see how meta-interpreters can be used to customize the search strategy and command shell (**NON-EXAMINABLE**, but nice link to next point)
- This will lead naturally on to the final Prolog lecture which will provide a more formal introduction to blind and heuristic search methods in general



<https://cs.stackexchange.com/questions/109673/how-can-i-compute-the-most-general-unifier-for-these-two-expressions>

I have the following first order logic expressions:

$$f(g(a, h(b)), g(x, y)), f(g(z, y), g(y, y))$$

1

and I want to compute the most general unifier for them. If I follow the algorithm found on [these slides](#) then I get the following.



$$S_0 = \{f(g(a, h(b)), g(x, y)), f(g(z, y), g(y, y))\}$$

$$D_0 = \{g(a, h(b)), g(z, y)\}$$

[first disagreement]



(need to unify these two functions)

$$D'_0 = \{a, z\}$$

[first sub-disagreement]

$$\sigma = \{z = a\}$$

$$D''_0 = \{h(b), y\}$$

[second sub-disagreement]

$$\sigma = \{z = a, y = h(b)\}$$

$$S_1 = \{f(g(a, h(b)), g(x, h(b))), f(g(a, h(b)), g(h(b), h(b)))\}$$

$$D_0 = \{x, h(b)\}$$

[second disagreement]

$$\sigma = \{z = a, y = h(b), x = h(b)\}$$

$$S_2 = \{f(g(a, h(b)), g(h(b), h(b))), f(g(a, h(b)), g(h(b), h(b)))\}$$

No disagreement, unifier found!  $\sigma = \{z = a, y = h(b), x = h(b)\}$ .

However, this is not the most general unifier as we have  $\{y = h(b), x = h(b)\} \in \sigma$ , so i think the MGU should actually be  $\sigma' = \{z = a, y = h(b), x = y\}$ .

Is this correct? When I use Prolog to unify them, it gives  $\sigma'$ . What did I do wrong in my following of the algorithm, so that it gave me a unifier that wasn't the most general?

How would you answer this?

Given these two expressions

$$\begin{aligned} t_1 &= f(g(a, h(b)), g(X, Y)) \\ t_2 &= f(g(Z, Y), g(Y, Y)) \end{aligned}$$

a student computes their mgu as

Yes,  $\sigma$  **is** indeed an mgu (and is also returned by our mgu algorithm)

$$\sigma = \{ Z/a, Y/h(b), X/h(b) \}$$

Yes,  $\sigma'$  **is** indeed more general than  $\sigma$  (strictly) as  $\sigma' \{Y/h(b)\} = \sigma$

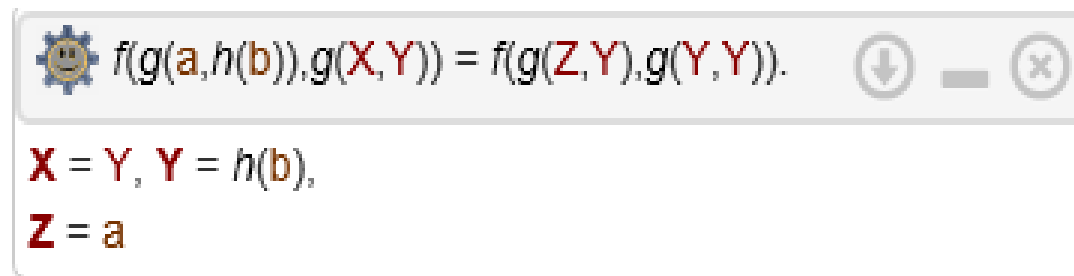
but then suggests the following would be more general

$$\sigma' = \{ Z/a, Y/h(b), X/Y \}$$


But  $\sigma'$  is **not** a unifier of  $t_1$  and  $t_2$  as  
 $t_1\sigma' = f(g(a, h(b)), g(Y, h(b))) \neq$   
 $t_2\sigma' = f(g(a, h(b)), g(h(b), h(b)))$

and believes this is actually what Prolog returns

And, Prolog returns  $\sigma$  not  $\sigma'$  as its output should be read as binding both  $X$  **and**  $Y$  to the same term  $h(b)$ !



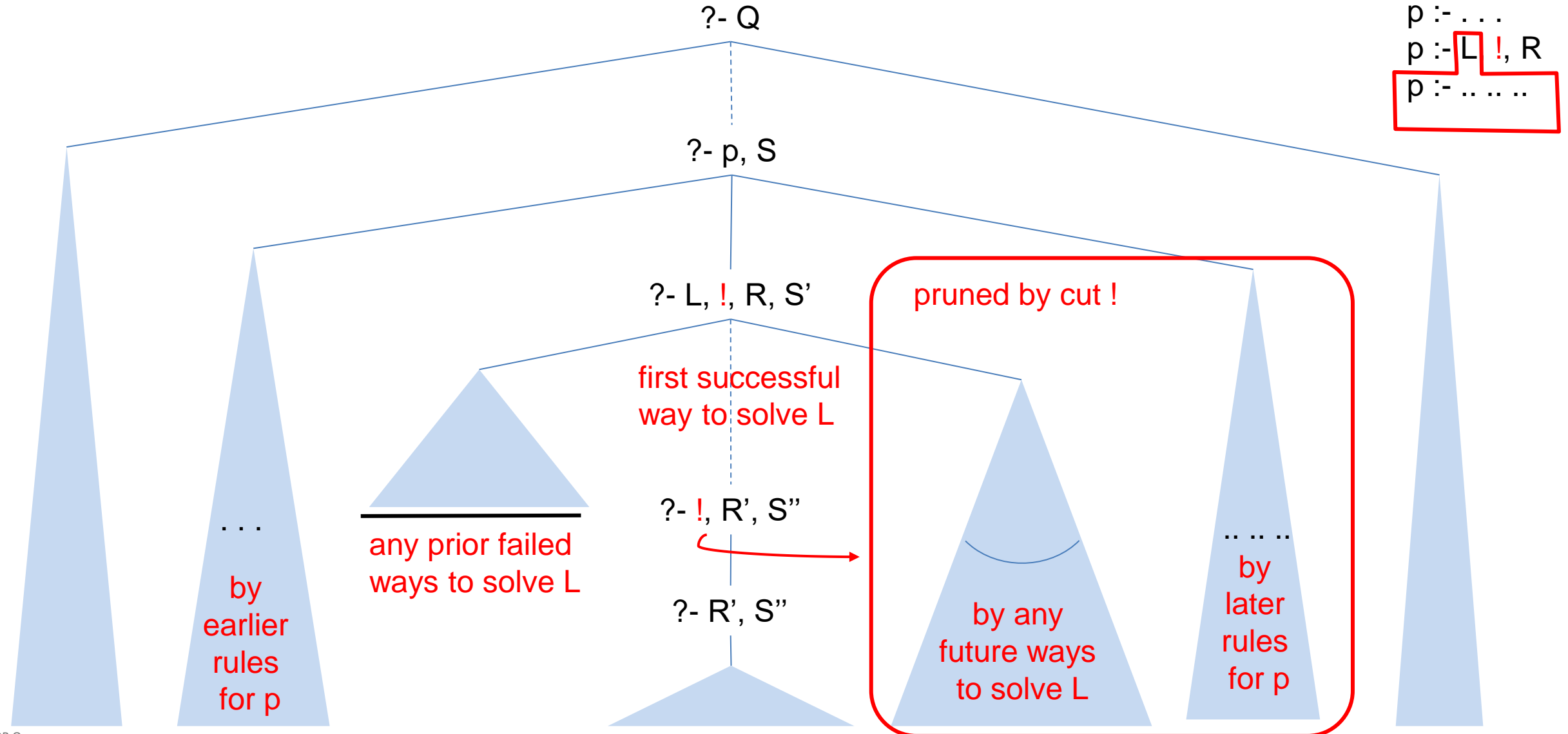
# Pruning the search with cut (!)

- Sometimes we want to prune (i.e. cut) away unwanted backtracking points in the SLD tree
  - For efficiency (**green cut**): avoid branches that contain no additional answers
  - For correctness (**red cut**): avoid branches that contain “incorrect”/“unwanted” answers 
- For this, Prolog offers the “cut” operator “!” which is declaratively read as “T”, but which has a procedural side-effect of pruning choice points made until now by the current and parent clause
- C can be very useful and convenient (when used thoughtfully); but they also have the potential to make programs hard to understand or behave in unexpected ways (when used carelessly)!
- Several higher-level constructs are defined using cut, which may be safer to use:
  - \+ naf
  - if -> then ; else
  - if \*-> then ; else
  - once
  - ignore

<https://apps.nms.kcl.ac.uk/reactome-pengine/documentation/man?section=control>

<https://apps.nms.kcl.ac.uk/reactome-pengine/documentation/man?section=metacall>

# The effect of cut



# ASCII notation for SLD trees with cuts

Expected in exams: “?-” precedes each goal, “+” is a choice point, “X” marks a subtree pruned by cut, “#” is a failure branch, “:” are infinite branches, “[ ]” is a success branch, and {..} is a computed answer

```
?- goal.  
|  
?- resolvent.  
|  
+-----+-----X-----.  
|               |               |  
?- branch_1     ?- branch_2     ?- branch_n  
|               |               |  
...             .!.             ...  
|               |               |  
?- success      ?- failure      ?- infinite  
|               |               |  
[ ]             #               :  
{..}
```



# Example: max(+Int, +Int, ?Int)

```
?- max(3,5,Z).
|
+-----X-----
|
?- 3=<5,!.      ?- 3>5.
|              |
|.             #
|
[]
{Z=5}
```

```
?- max(3,5,5).
|
?- 3=<5,!.
|.
|
[]
```

```
?- max(3,5,3).
|
?- 3>5.
|
#
```

```
?- max(3,5,4).
|
#
```

max(X,Y,Y) :- X=<Y, !.  
max(X,Y,X) :- X>Y.

Green cuts prune redundant failure branches (to avoid unnecessary tests when guards are mutually exclusive)

<http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse44>

```
?- max(5,3,Z).
|
+-----
|
?- 5=<3,!.      ?- 5>3.
|              |
#              []
               {Z=5}
```

```
?- max(5,3,5).
|
?- 5>3
|
[]
```

```
?- max(5,3,3).
|
?- 5=<3,!.
|
#
```

```
?- max(3,5,4).
|
#
```



## Example: size(+Int, -Size)

```
size(X,small) :- X<5, !.  
size(X,large) :- X>9, !. % X>=5  
size(X,medium).         % X>=5, X=<9
```

Red cuts can further increase efficiency (and also program compactness) by letting us make some guards completely implicit!

```
min(X,Y,Y) :- X>=Y, !.  
min(X,Y,X).      % X>Y
```

But, be very careful when combining guards with pattern matching in the predicate head: e.g. try running `?- min(5,3,5).`

# The “cut-fail” definition of negation

---

Negation-as-Failure can be defined in terms of cut as shown by the following definitions of the operators `\\+` and `\\\+` which are both logically equivalent to Prolog’s built-in `\+` operator:

```
:- op(900, fy, \\+).
```

```
\\+ X :- X, !, fail.
```

```
\\+ _.
```

```
:- op(900, fy, \\\+).
```

```
\\\+ X :- X, !, fail ; true.
```

# Example: negation as failure

`P :- \+q, r.`

`P :- q.`

`q.`

`r.`

```
?- p.
|
+-----+
|                                     |
?- \+q, r.                             ?- q.
|                                     |
+-----X-----+                     []
|                                     |
?- q, !, fail, r.                   ?- r.
|                                     |
?- !, fail, r.                       []
|
?- fail, r.
|
#
```

# The meta-predicate clause/2

- Prolog provides a special meta-predicate of the form `clause(+Goal,-Resolvent)` that succeeds once for every clause in the knowledge base that resolves with the given Goal in order to leave a resolvent, which is returned
- Note that an empty clause body is denoted by the constant “true”; a singleton body is denoted by its literal; a longer body is denoted by a tuple (A,B) where A is a literal and B is a literal or another tuple; and there is no empty tuple in Prolog!
- For example, given a program consisting of the following two clauses

`brother(X,Y) :- brother(Y,X) .`      `brother(paul,peter) .`

- Then query `?-clause(brother(peter,B),R)` succeeds with one answer:

`{R/brother(B,peter) }`

- But query `?-clause(brother(B,peter),R)` succeeds with two answers:

`{R/brother(peter,B) }`

`{B/paul,R/true}`

# The “vanilla” meta-interpreter

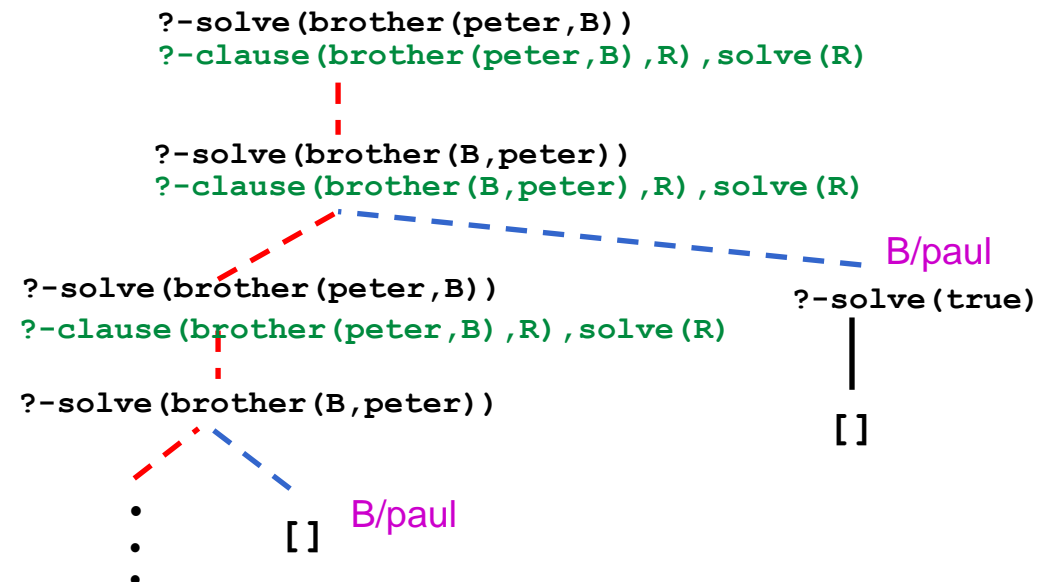
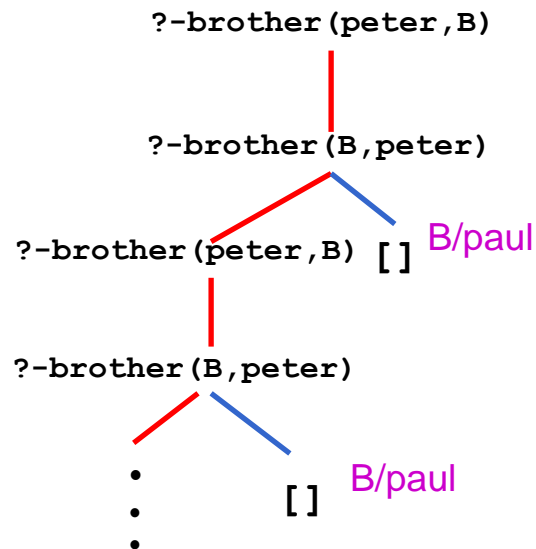
- The vanilla meta-interpreter is a Prolog predicate that simulates the evaluation of Prolog queries with respect to (pure) Prolog programs (cf. Simply Logical, p.70)

```

solve(true)    :- !.
solve((A,B))   :- !, solve(A), solve(B).
solve(A)       :- clause(A,B), solve(B).
  
```

- e.g. **brother(X,Y) :- brother(Y,X).**

**brother(paul,peter).**

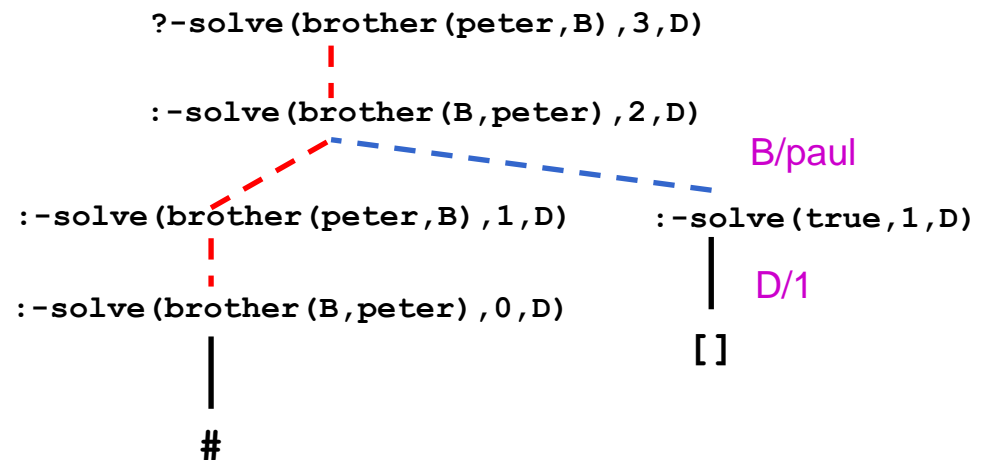
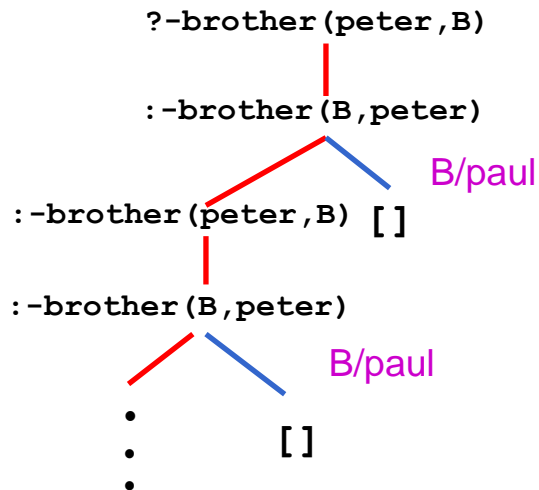


# Depth-bounded meta-interpreter

- The vanilla meta-interpreter can easily be adapted to customise the search strategy e.g. depth-bound

```
% solve (+Goal, +AllowedDepth, -RemainingDepth)
solve(true,Y,Y)      :- !.
solve((A,B),X,Z)     :- !, solve(A,X,Y), solve(B,Y,Z).
solve(A,X,Z)         :- X>0, clause(A,B), Y is X-1, solve(B,Y,Z).
```

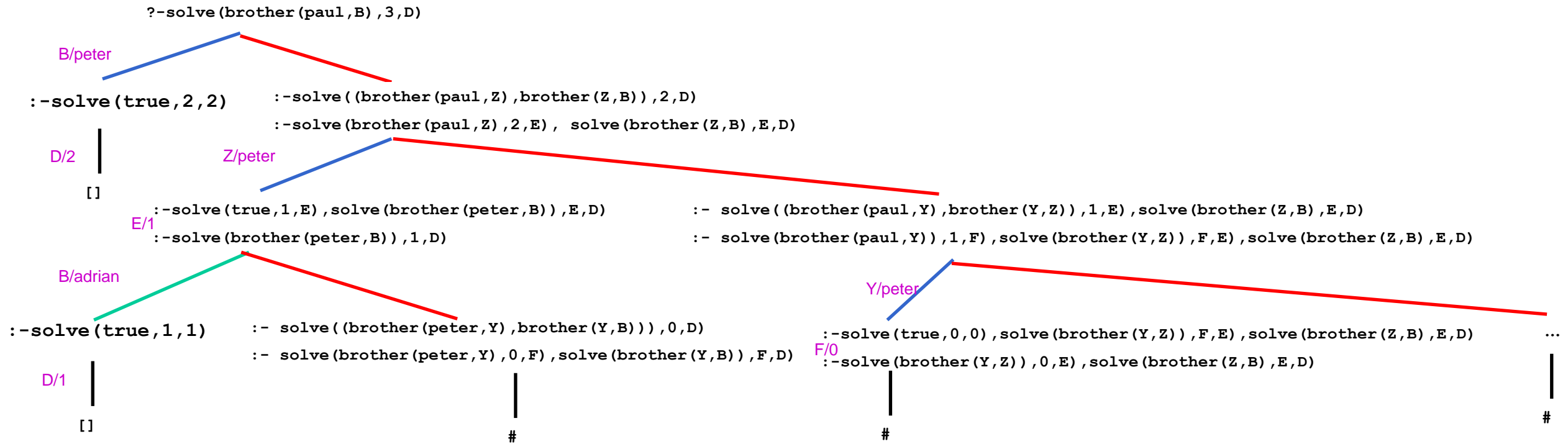
- e.g. **brother (X, Y) :- brother (Y, X) .**      **brother (paul, peter) .**



# Depth-Bounded SLD Tree

```
brother(paul,peter) .
brother(peter,adrian) .
brother(X,Y) :- brother(X,Z) , brother(Z,Y) .
```

```
solve(true,Y,Y) :- !.
solve((A,B),X,Z) :- !, solve(A,X,Y), solve(B,Y,Z).
solve(A,X,Z) :- X>0, clause(A,B), Y is X-1, solve(B,Y,Z).
```





# Incorporating into Interactive User Shell

---

```
shell :-  
    writeln("Hello!"),  
    repeat,  
        write("Q> "),  
        read(X),  
        (  
            X=quit, write("Good Bye!"),!  
        );  
        solve(X,5,_), format("A> ~w\n",[a(X)]) ; format("Done!\n\n"),fail  
    ).
```

NOTE THAT (M)ANY OTHER SEARCH STRATEGIES CAN BE IMPLEMENTED IN THIS WAY

- to achieve different trade-offs between time and memory
- e.g. Iterative Deepening – between(0, Max, Depth), solve(Query, Depth, \_)
- e.g. Breadth First Search, Best First Search, A\* search (see later!)

# An extended meta-interpreter

- The vanilla meta-interpreter

```
solve(true)    :- !.  
solve((A,B))   :- !, solve(A), solve(B).  
solve(A)       :- clause(A,B), solve(B).
```

- can easily be extended to handle disjunction, negation, and other built-ins (which are all trivial except cut – which would require a bit more work to simulate properly)

```
solve(true)           :- !.  
solve((A,B))          :- !, (solve(A) , solve(B)).  
solve((A;B))          :- !, (solve(A) ; solve(B)).  
solve(not(A))         :- !, not solve(A).  
solve(clause(A,B))    :- !, clause(A,B).  
solve(A=B)            :- !, A=B.  
solve(A\=B)           :- !, A\=B.  
solve(A)              :- clause(A,B), solve(B).
```

# A meta-circular interpreter

- Rewriting the extended meta-interpreter to replace the cuts by explicit negations results in a meta-interpreter that can interpret its own code (hence called circular)

```
solve(true) .  
solve( (A,B) )      :- (solve(A) , solve(B)) .  
solve( (A;B) )      :- (solve(A) ; solve(B)) .  
solve(not(A) )      :- not solve(A) .  
solve( clause(A,B) ) :- clause(A,B) .  
solve(A=B)          :- A=B .  
solve(A\=B)         :- A\=B .  
solve(A)            :- A\=true, A\=(_,_) , A\=(_;_) , A\=not(_) ,  
                      A\=clause(_,_), A\=(_=_), A\=(_\=_),  
                      clause(A,B) , solve(B) .
```

```
?-solve(solve(brother(peter,B))) .
```

# Prolog proofs (& other tasks) as Graph search

---

- Most AI tasks reduce to searching for goals in a graph structure (some tasks may initially be expressed as constraint satisfaction or function optimization, but these are ultimately solved by graph search techniques!)
- Search space is a **graph**, with partial solutions or **states** as nodes and possible transitions as arcs
  - search space may contain cycles
  - often approximated by a tree: easier algorithm, requires less memory, but is less efficient because of repetitions
  - or can add loop detection by keeping a list of visited nodes
  - In general transitions may have costs associated with them (not on this unit)
- Blind search methods look for goals in a systematic way, but do not take the quality of partial solutions into account (e.g. using heuristics)
- Different search strategies can be modelled with the notion of an **agenda** (which is simply a list of the nodes we plan to search next)

# Agenda-based search

```
search (Agenda, Goal) :-  
    next (Agenda, Goal, Rest) ,  
    goal (Goal) .  
search (Agenda, Goal) :-  
    next (Agenda, Current, Rest) ,  
    children (Current, Children) ,  
    add (Children, Rest, NewAgenda) ,  
    search (NewAgenda, Goal) .
```

Note that, in an agenda-based meta-interpreter, nodes on the agenda would simply be queries that represent choice points in the SLD tree; and arcs are determined by the resolution operator.

But nodes and arcs can be defined so as to give a direct representation of some hypothesis space and the technique of agenda-based search can be abstracted beyond meta-interpretation.

- This code makes an abstraction of the order in which search nodes are added to and removed from the agenda
  - Given an +Agenda, which represents a set of nodes at the **frontier** of the search process (initially set to start node)
  - it will return a -Goal node (as identified by a goal function) that is accessible from the nodes on the agenda

# Agenda-based search (using lists)

```
search ([Goal | Rest] , Goal) :-  
    next (Agenda, Goal, Rest),  
    goal (Goal) .  
search ([Current | Rest] , Goal) :-  
    next (Agenda, Current, Rest),  
    children (Current, Children) ,  
    add (Children, Rest, NewAgenda) ,  
    search (NewAgenda, Goal) .
```

Note that “next” has been implicitly implemented here by simply taking the first goal from head of a list representing the agenda to leave rest in the tail

- This code makes an abstraction of the order in which search nodes are added to and removed from the agenda
  - Given an +Agenda, which represents a **\*list\*** of nodes at the **frontier** of the search process (initially set to start node)
  - it will return a -Goal node (as identified by a goal function) that is accessible from the nodes **\*in\*** the agenda

# Depth-first vs Breadth-first

```
search_df([Goal|Rest],Goal):-  
    goal(Goal).  
search_df([Current|Rest],Goal):-  
    children(Current,Children),  
    append(Children,Rest,NewAgenda),  
    search_df(NewAgenda,Goal).
```

in depth-first, children are  
added to *front* of agenda

```
search_bf([Goal|Rest],Goal):-  
    goal(Goal).  
search_bf([Current|Rest],Goal):-  
    children(Current,Children),  
    append(Rest,Children,NewAgenda),  
    search_bf(NewAgenda,Goal).
```

in breadth-first, children are  
added to *back* of agenda

```
children(Node,Children):-  
    findall(C,arc(Node,C),Children).
```

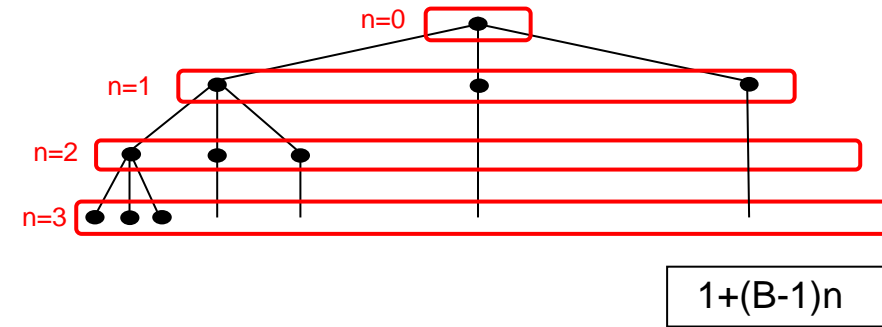
Note that in case of cyclic graphs,  
we may want to only take children  
we've not previously visited!



# Memory Requirements

- Depth-first search

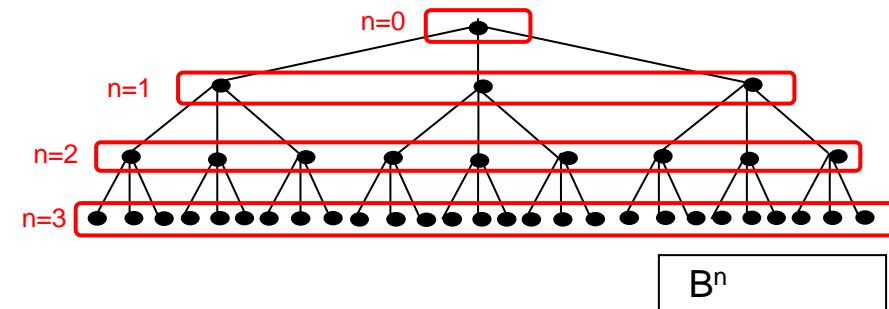
- agenda = stack (last-in first-out)
- incomplete: may get trapped in infinite branch
- no shortest-path property
- requires  $O(B \times n)$  memory



B is branching factor (av. number of children)  
n is the depth

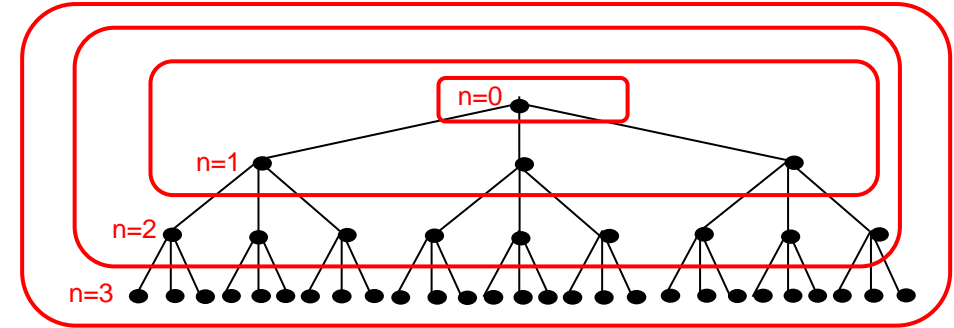
- Breadth-first search

- agenda = queue (first-in first-out)
- complete: will find all solutions
- first solution found along shortest path
- requires  $O(B^n)$  memory



# Iterative Deepening

- Iterative Deepening search
  - Combines good features of bfs and dfs
  - Optimality of bfs + Memory footprint of dfs
  - Repeats work, but works well in practice
  - (in a binary tree, half the nodes are in the leaves!)

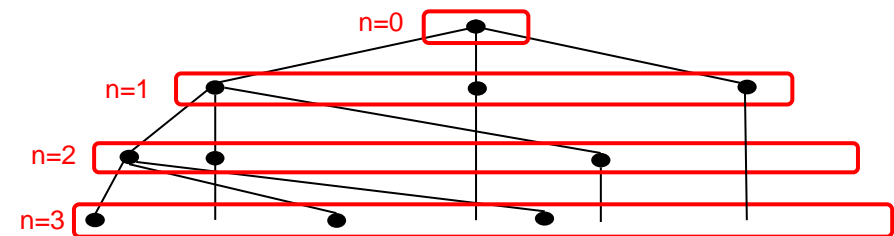


# Best-First Search

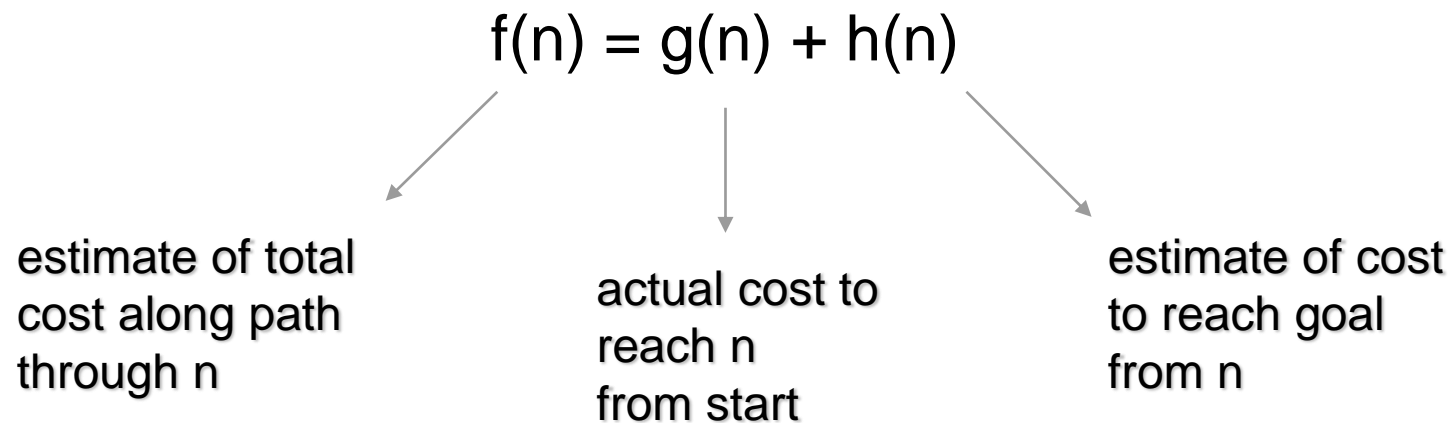
```
search_bstf([Goal|Rest], Goal) :-  
    goal(Goal).  
search_bstf([Current|Rest], Goal) :-  
    children(Current, Children),  
    add_bstf(Children, Rest, NewAgenda),  
    search_bstf(NewAgenda, Goal).
```

When adding children into the agenda, we can order them with respect to some (easily computed) metric (function) that tries to assess the quality of each node (by estimating the cost of the best solution path through that node). Thus we should put nodes with low scores at the front of the agenda (a.k.a priority queue)

- Best-first search
  - agenda = priority queue (preferentially ordered)
  - Behaviour depends on heuristic employed
  - Certain guarantees may be obtained if the metric satisfies certain properties (e.g. as in A\* search)

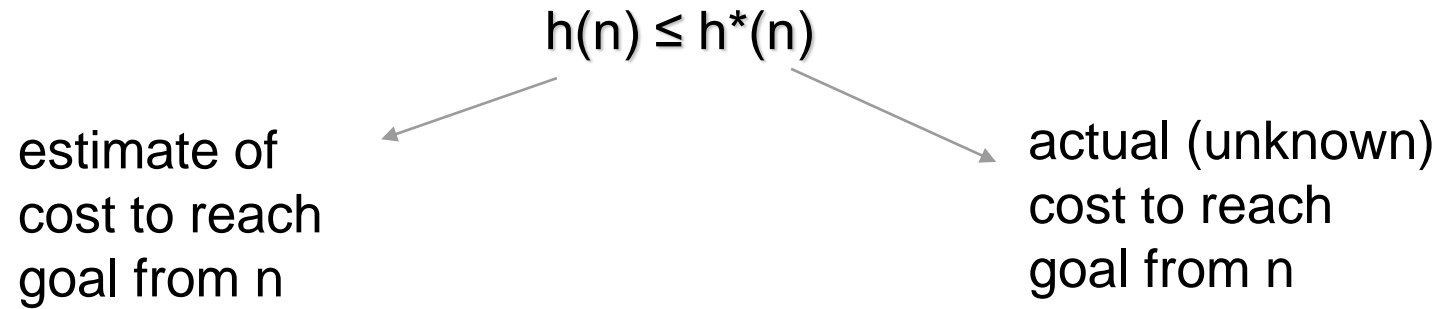


An “**A**” *algorithm* is a best-first search algorithm that aims at minimising the **total cost** along a path from start to goal.



Note:  $h(n)=0$  results in Breadth-first search  
 $g(n)=0$  results in Greedy Best First Search

A heuristic is (**globally**) *optimistic* or *admissible* if the estimated cost of **reaching a goal** is always less than the actual cost (from any node  $n$ ):



A heuristic is *monotonic* (or **locally optimistic** or *consistent*) if it never decreases by more than the cost  $c$  of an edge from a node  $n_1$  to a child  $n_2$ :

$$h(n_1) - h(n_2) \leq c$$

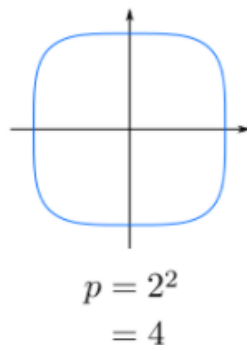
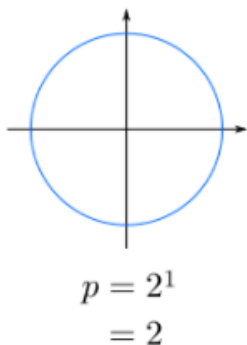
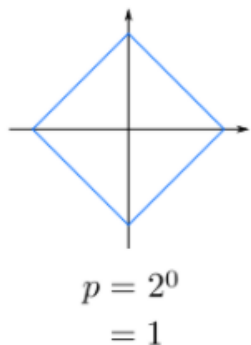
e.g. Manhattan, Euclidean,  
(or Minkowski/Chebyshev)  
distances in grid world!

# Distance metrics GridWorld settings

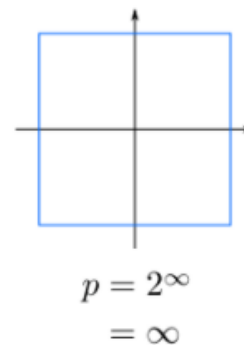
- In GridWorld pathfinding tasks the most common way to estimate the cost of a path to the goal is to simply compute the straight-line distance from the current coordinates to the goal (without trying to work out if any walls might turn out to block that path).
- The most common metrics for determining the distance between two points on n-dimensional grids are the Manhattan (or TaxiCab) distance and the Euclidean distance – which are both special cases of a metric called the Minkowski distance [Minkowski distance - Wikipedia](#)

The Minkowski distance of order  $p$  (where  $p$  is an integer) between two points  $X = (x_1, x_2, \dots, x_n)$  and  $Y = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n$

is defined as:

$$D(X, Y) = \left( \sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}.$$


...



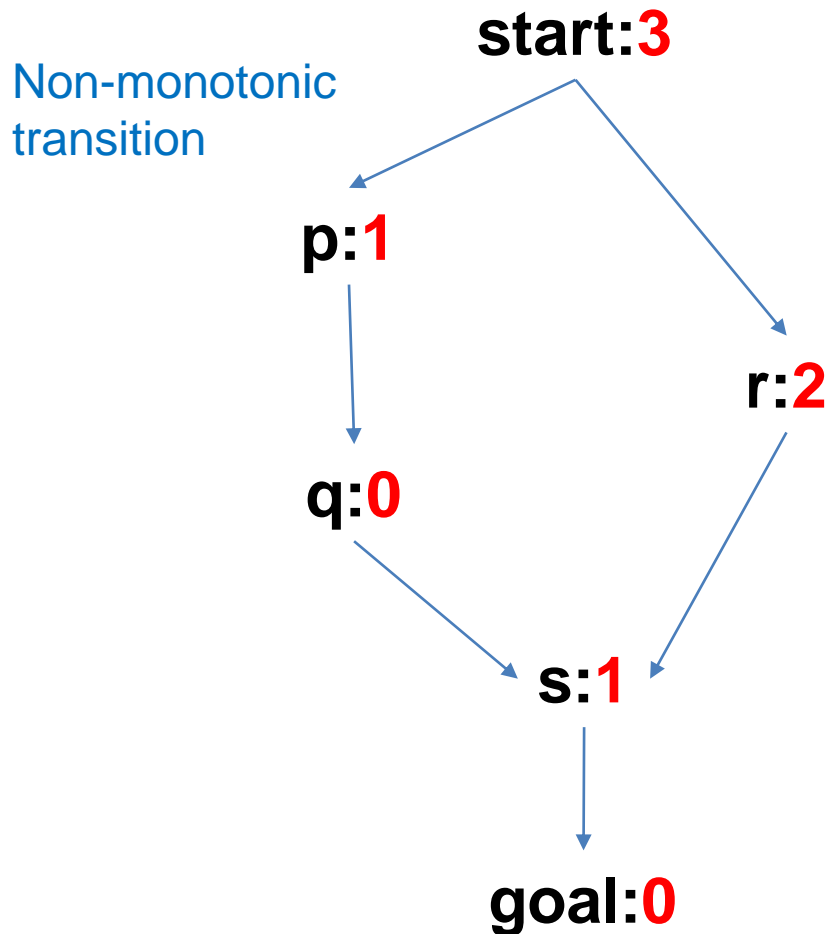
visualisation of 2D unit balls for increasing Minkowski order  
 $p=1$  (Manhattan)  
 $p=2$  (Euclidean)  
 ...

- A\* (A-star): an A algorithm with an admissible heuristic<sup>#</sup>
    - always reaches a goal along the cheapest path first
    - breadth-first is a special case
  - Monotonicity makes search more efficient<sup>#</sup>
    - first time a node is put on the agenda it is reached along the cheapest path
    - called monotonicity because f-values are never decreasing along a path
    - in the absence of monotonicity, agenda may contain multiple copies of the same node reached along different paths
- <sup>#</sup> assuming tied goals are processed in the order added to the agenda!



# A Non-Monotonic (but admissible) Heuristic

search graph (node-**hScore**)  
where all transitions have cost=1



A\* agenda (node-**fScore**)

[start-3]

[p-2, r-3]

[q-2, r-3]

[r-3, s-4]

[s-3, s-4]

[goal-3, s-4]

Non-optimal  
initial path  
to node s

Common Misconceptions Concerning Heuristic Search

Robert Holte, Proc. 3rd Annual Symposium on Combinatorial Search (SOCS-10)

<https://aaai.org/ocs/index.php/SOCS/SOCS10/paper/viewFile/2073/2500>

A Result on the Computational Complexity of Heuristic Estimates for the A\* Algorithm

Marco Valtorta, Information Sciences 34, 41-59 (1984) 47

<https://www.sciencedirect.com/science/article/pii/0020025584900094>

Analyzing Tie-Breaking Strategies for the A\* Algorithm – IJCAI'18

<https://www.ijcai.org/Proceedings/2018/0655.pdf>

[Inconsistent heuristics in theory and practice - ScienceDirect ...](#)

Artificial Intelligence 175(9-10):1570-1603, 2011

Ariel Felner, Robert Holte, Jonathan Schaeffer, Nathan Sturtevant, Zhifu Zhang

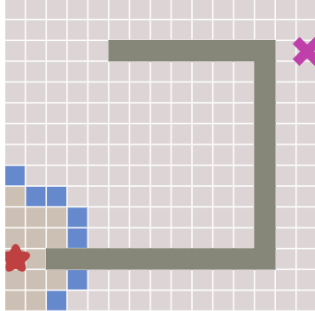
[Heuristics \(stanford.edu\)](#)

[A\\* Search: What's in a Name? | January 2020 | Communications of the ACM](#)

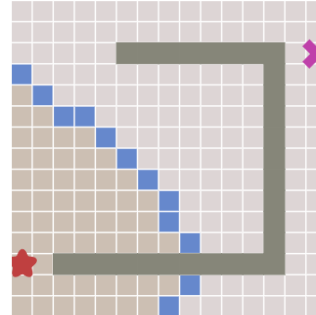
# Path Finding (in grid world)

Breadth-First

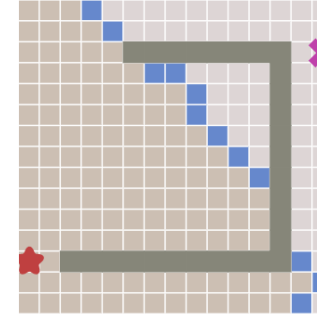
Breadth First Search



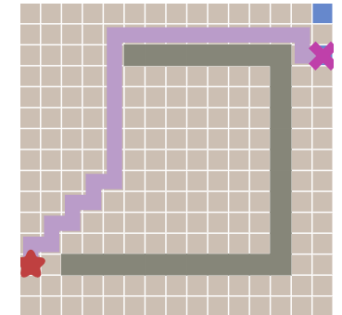
Breadth First Search



Breadth First Search

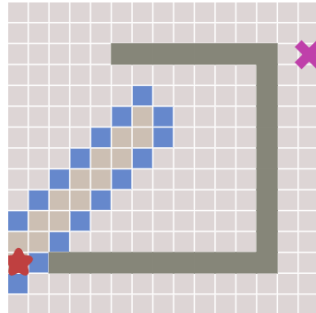


Breadth First Search

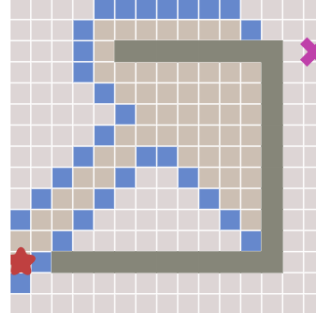


Greedy Best-First

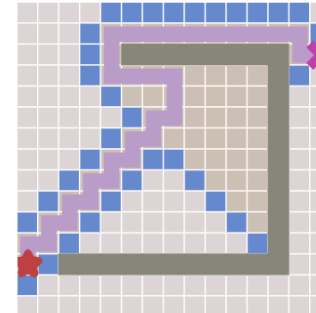
Greedy Best-First Search



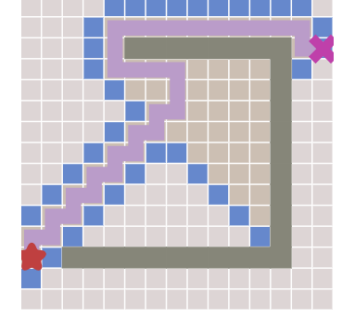
Greedy Best-First Search



Greedy Best-First Search

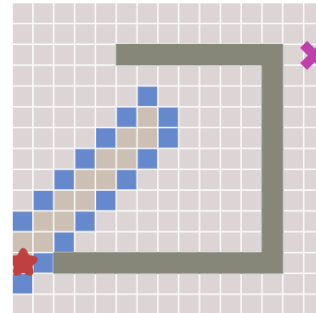


Greedy Best-First Search

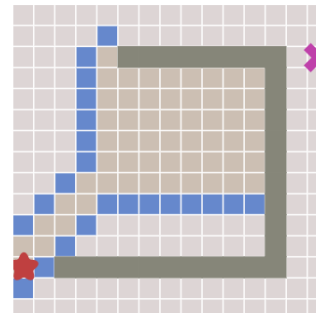


$A^*$   
(Manhattan)

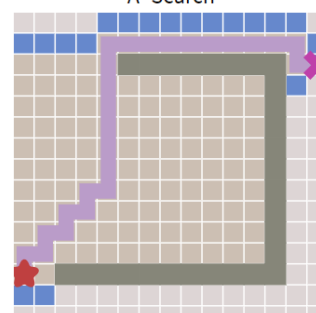
$A^*$  Search



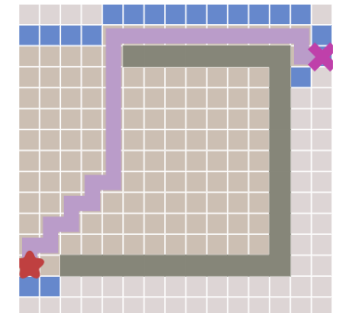
$A^*$  Search



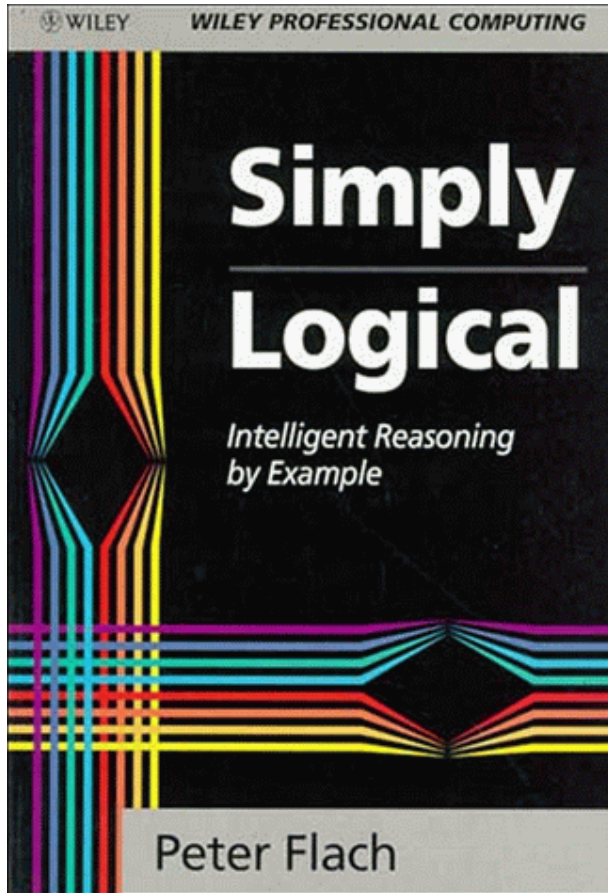
$A^*$  Search



$A^*$  Search



# More information on search

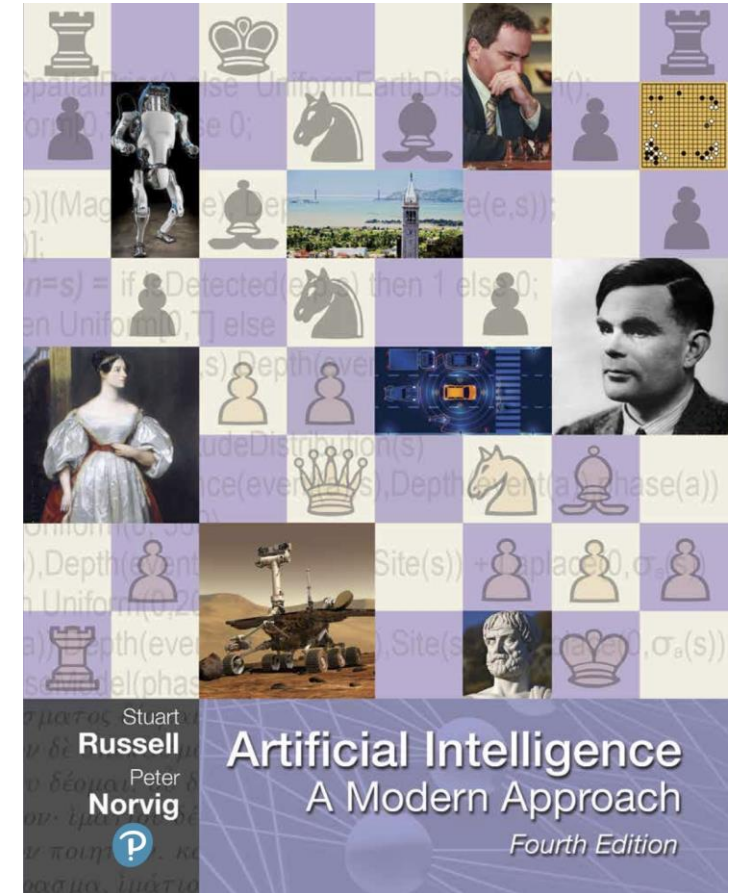


free, online, see chaps 5 & 6

Both books explain A\* search in some detail.

They also both contain very useful information on classical logic and Prolog (with the former also having good sections on SLD-tree/meta-interpretation and the latter also having a good overview of agent-based systems).

So feel free to look up any concepts you may find difficult to see if they can provide a fresh perspective or useful examples!



In library, see chaps 3 & 4