Queen Mary
University of London

Parallel Search in KL1

Huntbach, Matthew

For additional information about this publication click this link.
http://qmro.qmul.ac.uk/jspui/handle/123456789/4560

Information about this research object was correct at the time of download; we occasionally make corrections to records, please therefore check the published record when citing. For more information contact scholarlycommunications@qmul.ac.uk

ISSN 1369-1961

Department of
Computer Science

Technical Report No. 737

Parallel Search in KL1

Matthew Huntbach

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

June 1997

737

Parallel Search in KL1

Matthew Huntbach
Dept. of Computer Science
Queen Mary and Westfield College
London E1 4NS

10 June 1997

Abstract

KL1 is a language in which parallelism is inherent, and abstract: it is not tied down to any particular architectural model. In this paper we show how it may be used with ease to implement parallel versions of standard search algorithms, including best-first search, branch-and-bound search, and alpha-beta search. We investigate some of the problems and opportunities which parallel implementations of these algorithms provide.

The language KL1

KL1 originates from the Japanese Fifth Generation project [Sh & Wa 93]. While this project is often said to have chosen the Prolog as its basis, in fact it chose the more abstract concept of logic programming . The relationship of Prolog to logic programming is like that of Lisp to lambda-calculus: an early attempt to base a programming language on an abstract concept of computation, adding several features which did not correspond to anything in the abstract model to aid efficiency and turn it into a practical programming language. Both prospered through having been first on the scene and having some reasonably efficient early implementations.

KL1 has a syntax very similar to Prolog, but an operational behaviour very different, of the form characterised by Kowalski [Kowa 79] as "don't care" non-determinism, where Prolog has "don't know" non-determinism. In KL1 attempts to match goals against all clauses for a predicate are made in parallel (OR-parallelism), rather than in textual order with backtracking as in Prolog. Once a goal has matched against a clause it is said to have "committed" and cannot backtrack to another. This is accomplished by, in effect having a cut in every clause, with only system prmitives allowed to the left of the cut. The cut is written | when the set of system primitives to its left is non-empty, but is unwritten otherwise. There is no concept of goal ordering, rather any goal may be expanded with any matching clause at any time (AND-parallelism). Some control limitation on the AND-parallelism is imposed by making unification one-way: a goal only matches against a clause if it can match against its head without requiring any binding of its variables, and commits to a matching clause only when any variables are also sufficiently bound for any system primitives before the cut to execute and succeed. Any goal insufficiently bound to

commit to a clause stays suspended. Variables passed into a clause may be unified only by means of the system primitives =, which binds a variable on its left-hand side to its right-hand side, and := which binds a variable to the evaluation of an expression on its right-hand side (evaluation being suspended if the expression contains unbound variables). The usual logic programming convention that once given a value a variable may never be reassigned another holds.

In fact KL1 is one of a family of concurrent logic programming languages (surveyed in [Shap 89]) which differ only in fairly minor details. More details on the derivation of these languages from Prolog, and some consideration of programming techniques in them is given in [Hu & Ri 95]. An efficient implementation of KL1 on multiprocessor systems is available [RNC 96]; nearly all the programs in this paper are transferrable to the similar language Strand [Fo&Ta 89], which has been marketed as a practical language for parallel systems. In this paper we shall concentrate on using them to implement simple search algorithms of the sort used in Artificial Intelligence applications, with particular attention given to the problems and opportunities provided by the parallelisation of these algorithms given by implementing them in a concurrent logic language.

Search

The idea of search had a starring role in early work on Artificial Intelligence. Early textbooks on the subject (e.g. [Nils 71]) gave over a great deal of space to it, improved search methods had a major place in Artificial Intelligence conferences and journals. More recently, consideration of search in the abstract has tended to move from the field of Artificial Intelligence to mainstream Computer Science and Operational Research. Search continues to be an important tool for artificial intelligence workers, less likely to be discussed because now taken for granted. Parallelising search continues to be an active field of research in computer science, much of the interest coming from the fact that search algorithms designed for sequential computers rarely map easily onto parallel architectures. The challenge for the programming language designer offered by research in parallel search is to come up with languages which free the programmer from having to consider low level details of the architecture in order to concentrate on the more abstract aspects of parallel algorithms, while retaining sufficient control to efficiently exploit the parallelism available in the architecture.

In the field of logic programming, Prolog was notable for incorporating a search mechanism as a built-in part of the language. The inderminate logic languages like KL1 have been criticised for abandoning this built-in search, resulting in various attempts to create hybrid languages which can revert from indeterminism to nondeterminism [Cl & Gr 87], [Hari 90]. However, the lesson that can be picked up from any good Prolog textbook (e.g. [Brat 86]) is

that Prolog's search must be subverted and search explicitly programmed in if search is to be at any more than a trivial level. Thus it is the symbolic nature of Prolog rather than its built-in search which makes it suitable for artificial intelligence programming, and this symbolic nature is inherited by the stream parallel logic languages.

A Naïve Prolog Solution to the 8-Puzzle

The 8-puzzle is a familiar example used to illustrate state space search problems [Nils 71]. The problem consists of a 3×3 arrangement of eight cells marked with the numbers "1" to "8" and one blank space. A state represents a particular arrangement of the cells, its successors are generated by the four different ways in which a cell may be moved onto the blank space: from the left, right, and from up and down, although only if the blank space is in the centre will all four ways generate a valid successor. The aim of the puzzle is to find a list of moves which will change the arrangement from the given initial state to the desired goal state.

The goal state conventionally used is:

1	2	3
8		4
7	6	5

As an example, there are three possible moves from the following state :

2	8	3
1	4	5
7		6

The three successor states are:

2	8	3
1	4	5
7	6	

`Left`

2	8	3
1	4	5
	7	6

`Right`

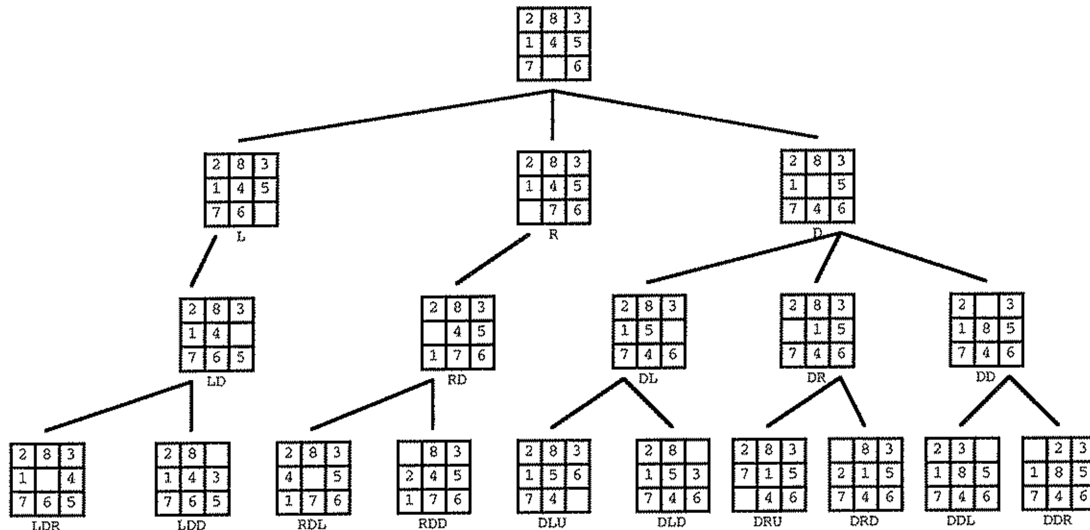
2	8	3
1		5
7	4	6

`Down`

A complete solution to the problem is the sequence of moves `[Left, Down, Right, Down, Right, Up, Left]`.

It is usually the case that the state information includes the last move made, so that the move which reverses this is not counted as a legitimate successor, thus further reducing the size of the search tree, but not to the extent that it is trivial. In fact in our case, we will assume that the state information includes all the moves made to reach the state from the initial state, so that if the state is found to be a solution, an answer can be returned straight away. The alternative would be to add the moves as the solution is passed up the search tree, leading to less memory usage, but a more complex program; this leads to the "layered streams" [Ok & Ma 87] approach which we discuss later on.

Part of the search tree for the problem above, including the paths taken to reach the states is given below:



Let us consider a naïve Prolog program to solve this puzzle:

```
search(State,State)  :- solution(State).
search(State,Soln)  :- left(State,Left), search(Left,Soln).
search(State,Soln)  :- right(State,Right), search(Right,Soln).
search(State,Soln)  :- up(State,Up), search(Up,Soln).
search(State,Soln)  :- down(State,Down), search(Down,Soln).
```

The predicates **`left`**, **`right`**, **`up`** and **`down`** give the successor states of the state given as their first argument, or fail if there is no such successor state.

This program, though entirely correct declaratively, will most likely fail to terminate since it can in effect gives a depth-first search of an infinite tree. We could extend the information stored at each state to record the intermediate positions passed through on the way to that state, and make the movement predicates fail if they return to a previously encountered position. This would limit search to a finite tree, but would still be grossly inefficient. In fact there are well known heuristics to direct the search for this problem [Nils 71]. The naïve Prolog program works badly because it transfers the left-to-right depth-first scheduling of Prolog directly to the search of the problem which actually requires a more sophisticated search scheduling.

A Naïve KL1 Solution to the 8-puzzle: OR-parallelism simulated by AND-parallelism

Notwithstanding the naïve nature of the program above, let us consider a similar solution in KL1:

```
search(nostate,Soln) :- Soln=none.
search(State,Soln) :- State≠none |
   issolution(State,Flag), expand(Flag,State,Soln).

expand(true,State,Soln) :- Soln=State.
expand(false,State,Soln) :-
   left(State,Left), search(Left,SolnL),
   right(State,Right), search(Right,SolnR),
   up(State,Up), search(Up,SolnU),
   down(State,Down), search(Down,SolnD),
   choose(SolnL,SolnR,SolnU,SolnD,Soln).

choose(SolnL,_,_,_,Soln) :- SolnL≠none | Soln=SolnL.
choose(_,SolnR,_,_,Soln) :- SolnR≠none | Soln=SolnR.
choose(_,_,SolnU,_,Soln) :- SolnU≠none | Soln=SolnU.
choose(_,_,_,SolnD,Soln) :- SolnD≠none | Soln=SolnD.
choose(none,none,none,none,Soln) :- Soln=none.
```

A number of points may be noted here. Firstly, the OR-parallelism which was implicit in the Prolog program is converted to KL1 AND-parallelism with the failure of an OR-parallel branch indicated by the return of the dummy solution `none`. Secondly, the predicate **choose** is introduced to make a non-determinate choice between solutions in OR-parallel branches, returning `none` itself if none of the branches returns a solution. This technique of converting conceptually OR-parallelism to AND-parallelism is a cliché in concurrent logic programming, introduced by Gregory [Greg 87] and Codish and Shapiro [Co & Sh 86]. In fact, the same technique is used in Prolog if we want to return all solutions to some problem, naturally, since returning all solutions from a situation where a solution is obtainable by method A *or* method B is equivalent to combining the solutions obtained from method A *and* method B. The Prolog all-solutions program is:

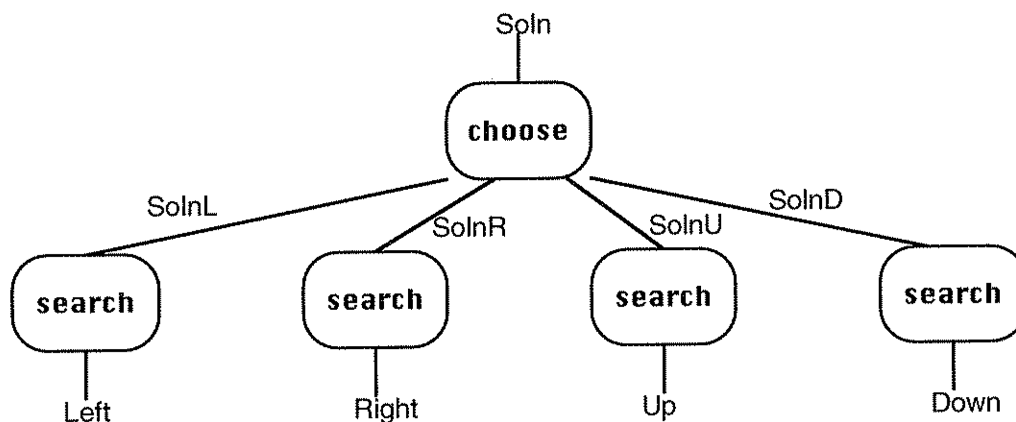
```
search(nostate,[]) :- !.
search(State,[State]) :- solution(State),!.
search(State,Solns) :-
   left(State,Left), search(Left,SolnsL),
   right(State,Right), search(Right,SolnsR),
   up(State,Up), search(Up,SolnsU),
   down(State,Down), search(Down,SolnsD),
   append4(SolnsL,SolnsR,SolnsU,SolnsD,Solns).
```

where **append4** simply appends its first four arguments together to give its fifth. It is assumed that rather than fail, **left**(State,Left) will bind Left to nostate if there is no left successor state, and likewise with the other moves. The same will happen in the KL1 program.

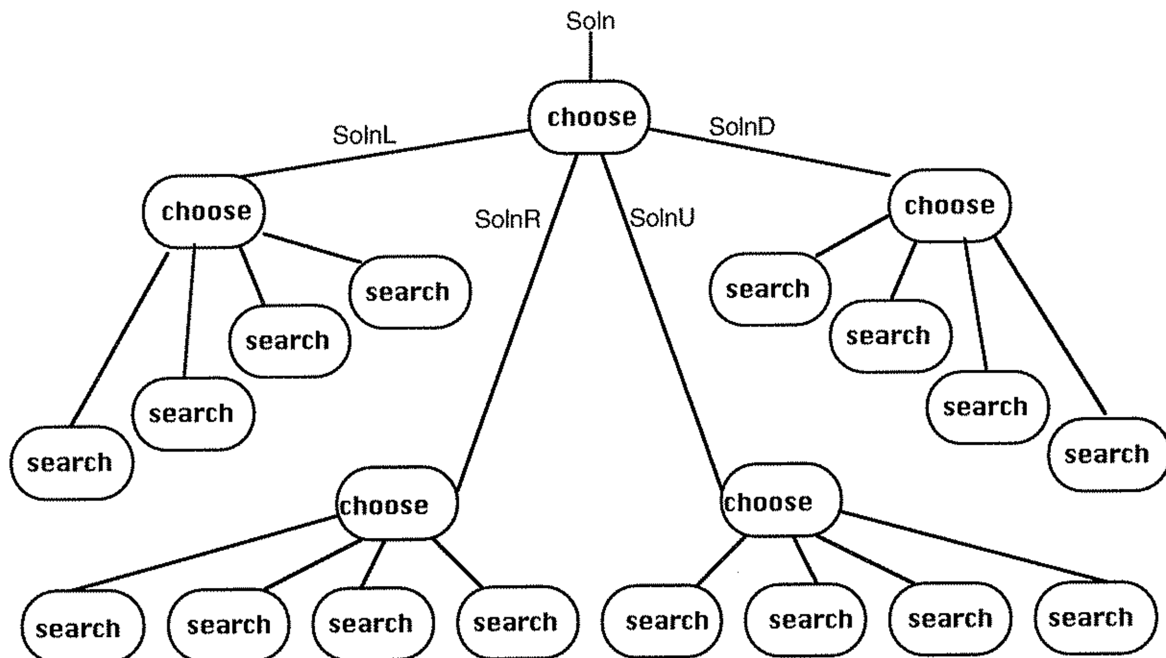
A third point is, given the flat nature of KL1, the conversion of the Prolog test predicate **solution** to the KL1 predicate **issolution**, which outputs true or false in a second argument, combined with the introduction of a new predicate to cover the search beyond the test, which takes the boolean value from **issolution** and reacts accordingly. For simplicity here, we have assumed that goal states in the search tree are leaves, hence the cut in the Prolog program is a green cut. A more complex Prolog program would be required to deal with cases where goal states in the search tree could have descendants which are also goal states.

The **choose** predicate is introduced in order to represent explicitly the independent binding frames for each non-determinately choosable clause, which are implicit in Prolog. The indeterminism of KL1 maps directly onto an indeterministic choice of solutions. The nondeterminism of Prolog is managed by wiping out information in the binding frame on backtracking. It may be assumed that **choose**(SolnL,SolnR,SolnU,SolnD,Soln) will rewrite as soon as any of its first four arguments have been bound to anything but none, or when all its first four arguments have been bound to none.

The result of executing the KL1 program will be to create a tree of **choose** processes, replicating the search tree of the problem itself. The goal **search**(State,Soln), will first rewrite to **issolution**(State,Flag),**expand**(Flag,State,Soln), and if **issolution**(State,Flag) caused Flag to be bound to true, following the rewrite of **expand**(Flag,State,Soln) the situation will be:



using a diagrammatic representation where the nodes are processes and the arcs variables. Unless either a solution is found, or no further state is generated by the move, each of the descendants of the root **choose** process will become a further **choose** process, and so on:



It can be seen that a major difficulty with the program is that the number of processes will expand exponentially, until any practical parallel system is overwhelmed. Although we know that in problems like this there are usually heuristics which tell us that a solution is more likely to be found in one subtree than another, since we are assuming unlimited parallelism, we have not built in a way of deciding which leaves in the search tree to expand given a limited number of processors to do the expansion. One solution to this, which we shall consider later, makes use of the KL1 priority operator.

Speculative Parallelism

Another problem in our naïve search is that we have omitted to describe a mechanism for halting processes which are not going to contribute to a solution. When we have a goal `choose(SolnL,SolnR,SolnU,SolnD,Soln)`, as soon as any of the first four arguments is bound to something other than `none`, it rewrites, and the continuing execution which is determining the values of the other arguments is redundant. This sort of parallelism is known as "speculative parallelism", since in effect any computation of the solution returned from a subtree is a speculation on that solution being necessary for the overall solution. In an ideal parallel processing situation, we make that speculation because we have nothing to lose – the processors engaged on the speculative computation would otherwise be left idle, so they might as well be used on computations which might be found to be needed. Of course, in practice the overhead of managing speculative parallelism means that it is too glib to

say there is "nothing to lose", so speculative parallelism should only be engaged in if the likelihood of it being needed outweighs the overhead. In fact, as shown by Lai and Sahni [La & Sa 84], in a search on a multiprocessor system where there are more possible node expansions than processors and node expansion choice is guided by heuristics, even in the absence of communication overheads it is possible for anomalous results to occur, ranging from slowdowns to speedups greater than the number of processors. We shall discuss this issue later in this paper.

Grit and Page [Gr & Pa 81] discuss the possibilities of incorporating a mechanism for automatically cutting off speculative computations which have been found unnecessary in a functional programming language. Without such a facility though, such a mechanism must be programmed in explicitly by the programmer (the problem is analogous to the consideration of whether garbage collection should be automatic or programmer-defined). It is essential that any mechanism which passes on the information that a solution has been found and further computation on alternative solutions is unnecessary should have priority over those further computations, otherwise we have the possibility of a node in the search tree being expanded and taking over the computational resources that would pass on the message that the expansion is unnecessary, a situation that might aptly be described as "undeadlock".

Speculative parallelism is associated particularly with OR-parallelism [Burt 88], indeed, it could be argued that the two are the same thing, the former term being used in a functional language context, the latter in a logic language context. However, as our search example shows, speculative parallelism is a real issue in AND-parallel languages, since we can write predicates like **choose** which make OR choices between their arguments. These might be considered generalisations of functional programming's **cond** operator [Turn 79], the operator which with three arguments, a boolean `V` and `A` and `B`, evaluates to `A` if `V` evaluates to `true`, and to `B` if `V` evaluates to `false`.

In functional languages, the operation **cond** is usually evaluated lazily, that is its second and third arguments are not evaluated before the operation is applied. A similar suspension of evaluation can be programmed in KL1 using its suspension mechanism:

```
eval_cond_lazily(V,A,B,Val) :- eval(V,VVal), cond(VVal,A,B,Val).

cond(true,A,_,Val) :- eval(A,Val).
cond(false,_,B,Val) :- eval(B,Val).
```

This is not fully lazy evaluation, since either `A` or `B` is fully evaluated rather than returned as an unevaluated reference, but for now serves to illustrate the point. Speculative

parallelism in functional languages requires a rejection of the evaluate **cond** lazily convention, generally involving special language constructs. No special mechanisms are required in KL1:

```
eval_cond_speculatively(V,A,B,Val) :-
      eval(V,VVal), eval(A,AVal), eval(B,BVal),
      cond(VVal,AVal,BVal,Val).

cond(true,AVal,_,Val) :- Val=AVal.
cond(false,_,BVal,Val) :- Val=BVal.
```

It is clear here that **eval_cond_lazily** and **eval_cond_speculatively** represent two extremes: in the former no potential parallelism is exploited, in the latter parallelism is exploited perhaps inappropriately. In fact, although it is picked out and made clear here, these two opposite approaches to potential speculative parallelism may occur less obviously in KL1 programs in general, and in fact fairly subtle changes to a program may cause a switch from overabundant parallelism to no parallelism at all. This is one of the reasons why, as noted in [Ti & Ic 90], a straightforward translation of a sequential logic program to a concurrent logic program does not always result in a good parallel program. We shall discuss the issue of over abundant against. no parallelism in further detail later on, and suggest that a simple middle-way solution is to make use of the priorities of KL1 in a similar way to Burton's use of priorities ro control speculative parallelism in functional languages [Burt 85].

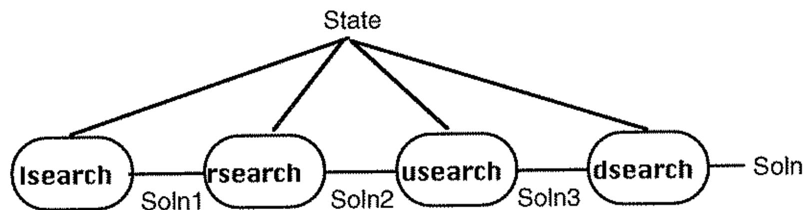
Non-speculative non-parallel linear search

Note that a simple attempt to remove the speculative element from our search program will result in us losing the parallelism of the search altogether. For example the following KL program for the 8-puzzle starts search of one branch of the tree only if no solution has been found in the previous branch:

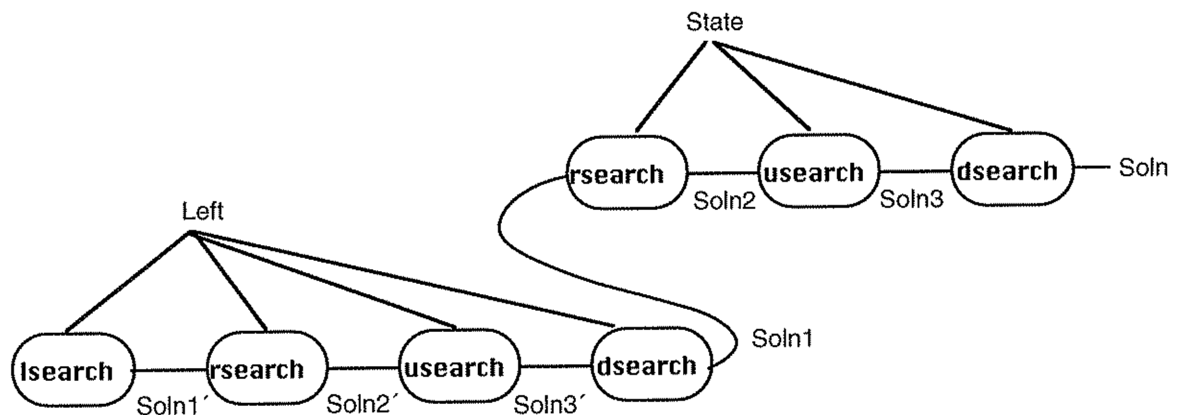
```
search(nostate,Soln) :- Soln=none.
search(State,Soln) :- State≠nostate |
   issolution(State,Flag), expand(State,Flag,Solution).

expand(true,State,Soln) :- Soln=State.
expand(false,State,Soln) :-
   lsearch(State,Soln1),
   rsearch(Soln1,State,Soln2),
   usearch(Soln2,State,Soln3),
   dsearch(Soln3,State,Soln).

lsearch(State,Soln) :-
   left(State,Left), search(Left,Soln).
```

```
rsearch(none,State,Soln) :-
   right(State,Right), search(Right,Soln).
rsearch(InSoln,State,Soln) :- InSoln≠none | InSoln=Soln.

usearch(none,State,Soln) :-
   up(State,Up), search(Up,Soln).
usearch(InSoln,State,Soln) :- InSoln≠none | InSoln=Soln.

dsearch(none,State,Soln) :-
   down(State,Down), search(Down,Soln)
dsearch(InSoln,State,Soln) :- InSoln≠none, InSoln=Soln.
```

It will result in a strict depth-first left-to-right search of the search tree, since consideration of any node in the tree is suspended on the result from consideration of the previous one in this search order in the tree. The process structure created by execution will be a chain of processes, all except for the first waiting for either a solution or a `none` to be passed in, in the former case the solution is passed straight down the chain and its length is reduced by one, in the latter the front of the chain is expanded. The process structure after the first expansion is:



After the next expansion it is:



We describe this form of search program as "linear search" from the linear form the processes link into during execution.

A Practical Prolog Solution to the 8-puzzle

In practice, Prolog solutions to search problems where left-to-right depth-first search is not the optimal search method abandon attempts to represent the search of the problem directly by Prolog's built-in search mechanisms. Instead, a meta-programming approach is used, in which some explicit representation of the global search space is manipulated. This gives us the following Prolog program for the 8-puzzle, in which search is specifically programmed in:

```
search([State|States],State) :- solution(State).
search([State|States],Soln) :-
    left(State,Left),right(State,Right),up(State,Up),down(State,Down),
    insert(Left,States,States1), insert(Right,States1,States2),
    insert(Up,States2,States3), insert(Down,States3,States4),
    search(States4,Soln).
```

Note that like the programs discussed above, this one simulates disjunctions in the problem by conjunctions in the program. The first argument to the **search** predicate is a list of states representing the current global search space (initially the list containing just the initial state). We can assume that **insert** inserts a state into its appropriate place in the list. We must also assume that **left**, **right**, **up** and **down** return the dummy value none for those states where they do not lead to a successor or where the successor to which they lead has already occurred as an intermediate state; attempting to insert none into the list of states leaves the list unchanged. The program is initially called with the list of states consisting of just the initial state as its single element.

The reason such a program is preferred is that it is generally possible to find "heuristics" [Pear 84] which are estimates of the likely cost of a solution through any state. The list of states is kept in order of the heuristic calculated for each state, so that **insert** in our above program keeps the active list of states in heuristic order. A simple heuristic in the 8-puzzle is to count the number of spaces each cell is away from its final position (the "Manhattan disatance"). Anyone familiar with this puzzle will know that this heuristic, while useful, is not always accurate as it is sometimes necessary to move a cell away from its correct position in order to make corrections elsewhere, and then move it back again. Still, we can see that using even this simple heuristic is an improvement on the fixed search order of Prolog, and in general for any program which involves a high degree of search we would not use a non-intelligent built-in search.

The Prolog meta-programming program is already very close to a KL1 solution. The only backtracking involved is in the testing for goal states, this is also the only case of output by unification. Removing this gives the KL1 version:

A Practical Prolog Solution to the 8-puzzle

In practice, Prolog solutions to search problems where left-to-right depth-first search is not the optimal search method abandon attempts to represent the search of the problem directly by Prolog's built-in search mechanisms. Instead, a meta-programming approach is used, in which some explicit representation of the global search space is manipulated. This gives us the following Prolog program for the 8-puzzle, in which search is specifically programmed in:

```
search([State|States],State) :- solution(State).
search([State|States],Soln) :-
   left(State,Left),right(State,Right),up(State,Up),down(State,Down),
   insert(Left,States,States1), insert(Right,States1,States2),
   insert(Up,States2,States3), insert(Down,States3,States4),
   search(States4,Soln).
```

Note that like the programs discussed above, this one simulates disjunctions in the problem by conjunctions in the program. The first argument to the **search** predicate is a list of states representing the current global search space (initially the list containing just the initial state). We can assume that **insert** inserts a state into its appropriate place in the list. We must also assume that **left**, **right**, **up** and **down** return the dummy value none for those states where they do not lead to a successor or where the successor to which they lead has already occurred as an intermediate state; attempting to insert none into the list of states leaves the list unchanged. The program is initially called with the list of states consisting of just the initial state as its single element.

The reason such a program is preferred is that it is generally possible to find "heuristics" [Pear 84] which are estimates of the likely cost of a solution through any state. The list of states is kept in order of the heuristic calculated for each state, so that **insert** in our above program keeps the active list of states in heuristic order. A simple heuristic in the 8-puzzle is to count the number of spaces each cell is away from its final position (the "Manhattan disatance"). Anyone familiar with this puzzle will know that this heuristic, while useful, is not always accurate as it is sometimes necessary to move a cell away from its correct position in order to make corrections elsewhere, and then move it back again. Still, we can see that using even this simple heuristic is an improvement on the fixed search order of Prolog, and in general for any program which involves a high degree of search we would not use a non-intelligent built-in search.

The Prolog meta-programming program is already very close to a KL1 solution. The only backtracking involved is in the testing for goal states, this is also the only case of output by unification. Removing this gives the KL1 version:

```
search([State|States],Soln) :-
   issolution(State,Flag), expand(Flag,State,States,Soln).

expand(true,State,States,Soln) :- Soln=State.
expand(false,State,States,Soln) :-
   left(State,Left),right(State,Right),up(State,Up),down(State,Down),
   insert(Left,States,States1), insert(Right,States1,States2),
   insert(Up,States2,States3), insert(Down,States3,States4),
   search(States4,Soln).
```

This program is deceptive. It appears to give an example of parallel heuristic search. In fact it is a good example of the problem of parallelism disappearing if speculative as mentioned above. A goal **search**(States1,Soln) will only rewrite to **isgoal**(State,Flag),**search1**(Flag,State,States,Soln) when States1 has become bound to [State|States]. This means that although the four descendant states of state can all be calculated in parallel, any further state expansion is left until the head of the subsequent list of goals is known, that is, after all the insertions, since **search**(States4,Soln) will not rewrite until that is the case. Any search before this would have to speculate on which goals would be at the head of the list.

However, we can pursue parallelism a little further with the above approach. Suppose we we have N processors available, and we have process mapping notations, so **pred**(Args)@p, $1 \leq p \leq N$, indicates that **pred**(Args) should be executed on processor p. Then we could arrange to expand the first N states in the list simultaneously, distributing each state to a separate processor, where above we only expand the first state. This gives the following program:

```
search(N,States,Soln) :-
   distribute(N,States,OutStates,Soln1),
   search_if_no_soln(N,Soln1,OutStates,Soln).

search_if_no_soln(N,none,States,Soln) :- search(N,States,Soln).
search_if_no_soln(N,Soln1,States,Soln) :- Soln1≠none | Soln=Soln1.

distribute(N,[State|States],OutStates,Soln) :- N>0 |
   try_state(State,OutStates1,OutStates,Soln1)@N, N1:=N-1,
   distribute(N1,States,OutStates1,Soln2), choose(Soln1,Soln2,Soln).
distribute(0,States,OutStates,Soln) :- OutStates=States, Soln=none.
distribute(N,[],OutStates,Soln) :- OutStates=[], Soln=none.

choose(none,none,Soln) :- Soln=none.
choose(Soln1,_,Soln) :- Soln1≠none | Soln=Soln1.
choose(_,Soln1,Soln) :- Soln1≠none | Soln=Soln1.
```

```
try_state(State,InStates,OutStates,Soln) :-
   issolution(State,Flag),expand(Flag,State,InStates,OutStates,Soln).

expand(true,State,InStates,OutStates,Soln) :-
   Soln=State, OutStates=InStates.
expand(false,State,InStates,OutStates,Soln) :-
   left(State,Left),right(State,Right),up(State,Up),down(State,Down),
   insert(Left,InStates,States1), insert(Right,States1,States2),
   insert(Up,States2,States3), insert(Down,States3,OutStates),
   Soln=none.
```

There are several problems with this program. Firstly it is dependent on a specific architecture: one in which all processors may be accessed from a central controller (although it does allow a flexibility in the number of processors). Secondly it involves high communication costs: at each cycle states are sent out to each processor and their descendants then collected together again on one processor. Thirdly it does not give optimal use of the multiprocessors. If a processor finishes work on the state it has been sent while the others are still busy, it cannot go on to search further states; rather it must wait until all processors have finished and work is distributed again. The main problem, however, is that it does not separate algorithm from control, as in Kowalski's famous dictum [Kowa 79]. The abstract consideration of expanding the search tree is mixed up with the lower level control considerations. The resulting program is a long way removed from the abstract control-free program we started with. We would prefer to start with a simple declarative program and add annotations where necessary to obtain a practical parallel program.

This latest program may be regarded as a meta-programming solution to the problem, since the actual tree expansion of the task can be considered a meta-program running on top of the main program which covers the control aspect. The meta-programming aspect would become even more dominant on a more complex architecture such as a distributed network of processors. Taking this meta-programming approach further leads to the replicated worker approach to parallel search [Bal 91].

A Generic Search Program

Before continuing, let us switch from discussing specifically the 8-puzzle to a more general pattern of search program into which procedures as necessary could be plugged in to make it an 8-puzzle or any other state-space search. We will need a predicate **issol**`(State,Flag)` which takes the representation of a state in the search, and binds `Flag` to `true` if it is a solution state, and `false` otherwise, and **successors**`(State,Succs)` which binds `Succs` to the list of successor states of `State`. This will give us the following general declarative search program:

```
search(State,Sols) :- issol(State,Flag), expand(Flag,State,Sols).

expand(true,State,Sols) :- Sols=[State].
expand(false,State,Sols) :-
   successors(State, Succs), branch(Succs, Sols).

branch([], Sols) :- Sols=[].
branch([State|Siblings], Sols) :-
   search(State,Sols1), branch(Siblings,Sols2),
   combine(Sols1,Sols2,Sols).
```

The underlined italic ***combine*** is meant to indicate that a number of different predicates could be put here, resulting in a variety of search programs with different properties.

To get a similar behaviour to our initial search program, for ***combine*** we use a simple indeterminate `choose`, similar to before except taking only two inputs:

```
choose([Sol], _, OutSol) :- OutSol=[Sol].
choose(_, [Sol], OutSols) :- OutSols=[Sol].
choose([], Sols, OutSols) :- OutSols=Sols.
choose(Sols, [], OutSols) :- OutSols=Sols.
```

Note that in the place of the previous `none`, the empty list `[]` is used, and the solution is returned inside a single element list. This enables us to obtain a greater variety of programs, including multiple solution programs just by varying ***combine***. The result of using this `choose` for ***combine*** is that program execution will result in a binary tree of `choose` processes which represents the search tree of the problem using the standard n-ary to binary tree mapping that can be found in any good data structures text book, where the left branch represents the relationship "first child" and the right branch the relationship "next sibling". Declaratively, the solution obtained is as before, though pragmatically the change would mean that the bias between indeterminate choices would change the likelihood of any particular solution being returned in the case of multiple solutions. For example, suppose a solution is obtainable through all four branches of the 8-puzzle, and the choice is made when all four solutions are available so we can discount time-related factors, and when there is a choice there is an equal chance of any of the clauses being used for rewriting. In our initial program there is an equal chance of any of the four solutions being returned, while in this version the first solution has only to get through one `choose`, while the second has to get through two, and the third and fourth through three, so there is a 50% chance of the first solution being returned, a 25% chance of the second and a 12.5% chance for each of the third and fourth.

If ***combine*** is standard list append, the output will be all solutions in left-to-right order from the search tree. If it is stream merger, the output will be all-solutions in an indeterminate order; in fact ignoring communications factors, in the order in which they are found temporally.

Other possibilities are:

```
leftmost([Sol], _, OutSols) :- OutSols=[Sol].
leftmost([], Sols, OutSols) :- OutSols=Sols.
```

which will result in the leftmost solution in the search tree being returned, and

```
indeterminate(Sols, _, OutSols) :- OutSols=Sols.
indeterminate(_, Sols, OutSols) :- OutSols=Sols.
```

which maps the indeterminacy of KL1 straight into an indeterminate return of any solution (obviously, not an option to be considered unless every leaf in the search tree is a solution!). If there are costs associated with solutions, and **cost**(Sol,Cost) gives the associated cost, then using **lowestcost** as given below for ***combine*** will result in the lowest cost solution being returned:

```
lowestcost([], Sols, OutSols) :- OutSols=Sols.
lowestcost(Sols, [], OutSols) :- OutSols=Sols.
lowestcost([Sol1], [Sol2], OutSols) :-
   cost(Sol1,Cost1), cost(Sol2,Cost2),
   cheapest(Sol1, Cost1, Sol2, Cost2, Sol), OutSols=[Sol].

cheapest(Sol1,Cost1,_,Cost2,Sol) :- Cost1<Cost2 | Sol=Sol1.
cheapest(_,Cost1,Sol2,Cost2,Sol) :- Cost1>=Cost2 | Sol=Sol2.
```

A generic version of the linear search program may also be written, using the principle that a branch in the tree is searched only if search of the branch to its left has completed and no solution was found in it. Unlike the linear search version of the 8-puzzle, however, this program generates the successors of a node in advance rather than if and when needed.

```
search(State,Sols) :- issol(State,Flag), expand(Flag,State,Sols).

expand(true,State,Sols) :- Sols=[State].
expand(false,State,Sols) :-
   successors(State, Succs), branch(none, Succs, Sols).

branch(Sol1, [], Sol) :- Sol=Sol1.
branch(Sol1, _, Sol) :- Sol1≠none, Sol=Sol1.
branch(none, State:Siblings, Sol) :-
   search(State, Sol1), branch(Sol1, Siblings,Sol).
```

Another form of search which is also linear since rather than have a separate ***combine*** predicate the solutions from one branch are passed as input to the process working on its sibling, is the all-solutions program which instead of appending lists of solutions adds them to an accumulator. In concurrent logic programming, this is known as a short-circuit technique, since it amounts to the various solutions eventually being linked together like an electric circuit. In this case, the program is fully parallel. The generic version is:

```
search(Acc,State,Sols) :-
   issol(State,Flag), expand(Flag,Acc,State,Sols).

expand(true,Acc,State,Sols) :- Sols=State:Acc.
expand(false,Acc,State,Sols) :-
   successors(State, Succs), branch(Acc,Succs,Sols).

branch(Acc,[],Sols) :- Sols=Acc.
branch(Acc,[State|Siblings],Sols) :-
   search(Acc,State,Sols1),
   branch(Sols1,Siblings,Sols).
```

In Prolog this technique is known as the difference-list technique [Cl & Tä 77], with the convention being that the pair `Acc` and `Sols` is written `Sols-Acc`, indicating that a conceptual way of viewing it is as the solutions being returned being those in `Sols` less those in `Acc`.

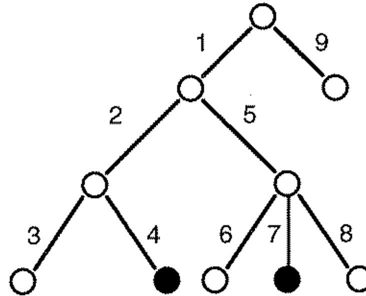
The Layered Stream Approach to Returning Solutions

As mentioned previously, the layered stream approach is an alternative to the method we have assumed that complete solutions are delivered at the leaf of the search tree. The idea is that if the solutions to some search problem consist of a list of moves, the head of the list being derived from the first possible move independent of subsequent moves, and the tail of the list being the remaining moves, then the set of solutions can be returned in a tree form matching the tree structure of the search space. Using the notation convention introduced by Okumura and Matsumoto, a set of solutions with a common first move M is stored in the form $M*[T_1, ..., T_n]$ where T_i is a tree in similar form representing a set of possible tails to the list of moves. A layered stream representing the set of solution moves from a given state takes takes the form of either a list of these sets of solutions or two special atomic values, either begin indicating that the state is itself a solution state and so a solution is found with the empty list of moves, or [] indicating a state which is not a solution and there are no possible moves from it that will lead to a solution.

So, for example, the layered stream

[1*[2*[3*begin,4*[]],5*[6*begin,7*[],8*begin]],9*begin]

is a representation of the set of solutions {[1,2,3],[1,5,6],[1,5,8],[9]}, which is more intuitively viewed as the tree:



where the arcs are labelled with the moves they represent, and the black circles represent nodes that are not solutions but have no descendants.

The generic search program which produces a layered tree representing the set of possible solutions is:

```
search(State,Sols) :- issol(State,Flag), expand(Flag,State,Sols).

expand(true,State,Sols) :- Sols=begin.
expand(false,State,Sols) :-
   successors(State, Succs), branch(Succs, Sols).

branch([], Sols) :- Sols=[].
branch([State/Move|Siblings], Sols) :-
   search(State,Sols1),
   branch(Siblings,Sols2),
   join(Move,Sols1,Sols2,Sols).

join(Move,Sols1,Sols2,Sols) :- Sols=Move*Sols1:Sols2.
```

where it is assumed that states are returned paired with the move that generated them from their parent in **successors**.

A slightly more complex form of **join** removes dead-end branches from the tree as they are constructed:

```
join(Move,Sols1,Sols2,Sols) :- Sols1≠[] | Sols=Move*Sols1:Sols2.
join(Move,[],Sols2,Sols) :- Sols=Sols2.
```

A version of **join** that returns a single solution indeterminately is easy to define:

```
join(Move,Sols1,_,Sols) :- Sols1≠[] | Sols=Move:Sols.
join(Move,_,Sols2,Sols) :- Sols2≠[] | Sols=Move:Sols2.
join(Move,[],[],Sols) :- Sols=[].
```

Further variants of **join** may be considered in a similar way to the variants of ***combine*** we have considered previously, and we will consider later. From this it may be seen that though layered streams may seem complex when combined in an undisciplined way with other elements of search (for example in some of the programs in [Tick 91]), when analysed in this generic way it can be seen to be a natural extension to the techniques already introduced. The complexity comes about because in the interests of efficiency elements of the search which we have split up here into separate predicates may in practice be interlinked in single multi-purpose predicates. Our preferred mode of programming would be to encourage the use of generic programming patterns or clichés [Wate 85] in program development, but to use program transformation methods to produce more efficient but less understandable programs.

Deleting Irrelevant Search

A simple way of deleting unnecessary speculative search is through the use of a "termination variable". The idea is that search may only continue while this variable remains unbound. When a solution is found, the variable is bound to some constant `found`. This variable is checked each time an expansion of the search tree takes place. Of course, objections may be raised that this technique requires the use of an extra-logical test for whether a variable is bound, but we shall override those objections and pursue it for now. A generic search program using the termination variable technique is given below:

```
search(Term, State, Sols) :-
   issolution(State, Flag),
   expand(Flag, Term, State, Sols).

expand(Flag, found, State, Sol) :- Sol=none.
expand(true, Term, State, Sol) :- unknown(Term) | Sol=State.
expand(false, Term, State, Sol) :- unknown(Term) |
   successors(State, States),
   branch(Term, States, Sol).

branch(Term, [], Sol) :- Sol=none.
branch(Term, [State|Siblings], Sol) :-
   search(Term, State, Sol1),
   branch(Term, Siblings, Sol2),
   choose(Sol1, Sol2, Sol).
```

The exact nature of the test for a variable being unbound in a concurrent and potentially distributed program is an issue for discussion. We have used **unknown** to indicate an inexpensive test which simply checks whether the variable is known to be bound at the processor on which the unknown test takes place. It may be that it has been found

elsewhere, but news of that binding has not yet reached the processor. The distinction between this test and a **var** test which locks the variable and makes a full check for lack of a binding throughout the system is noted in [Yard 90]. This reference also suggests a "test-and-set" operator, which as an atomic operation checks the whole system for whether a variable is unbound, and if it is unbound sets it to a value, otherwise it has no effect. The advantage of this operator is that it can overcome the binding conflict problem [Bu & Ri 88], and allow us to have a single shared variable to return a solution, in effect this variable has the dual role of returning a solution and acting as a termination variable. This version is given below:

```
search(State, Sols) :-
   issolution(State, Flag),
   expand(Flag, State, Sols)

expand(Flag, State, Sol) :- nonvar(Sol) | true.
expand(true, State, Sol) :- test_and_set(Sol,State).
expand(false, State, Sol) :- var(Term) |
   successors(State, States),
   branch(States, Sol).

branch([], Sol) :- true.
branch([State|Siblings], Sol) :-
   search(State, Sol),
   branch(Siblings, Sol).
```

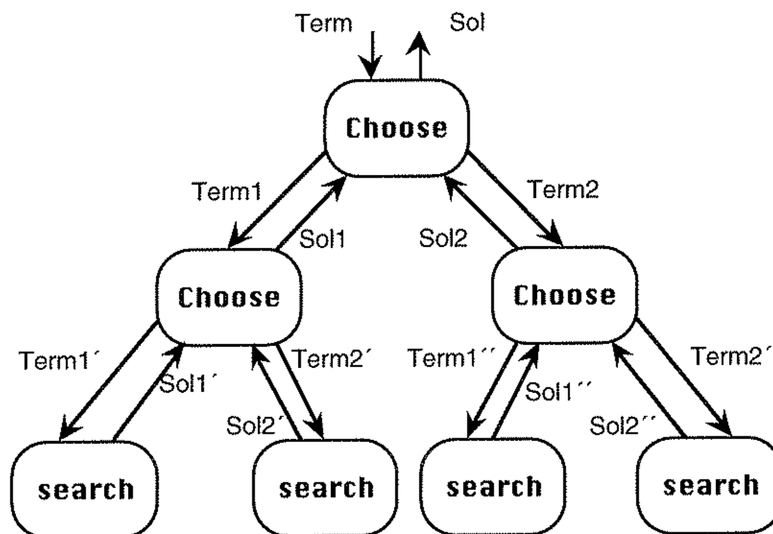
The version using **test_and_set** enbables us to dispense with the hierarchy of **choose** goals which are necessary in the version using **unknown**. This is because without a **test_and_set** we cannot tell whether a variable we are binding has been bound elsewhere; even with **var** we cannot guarantee that in between `var(X)` succeeding and the execution of `X=value`, `X` has not been bound elsewhere. This problem does not occur with the termination variable so long as = is unification rather than assignment since then `Term=found` will succeed even if `Term` has been bound elsewhere to `found`. Nevertheless, if multiple writers to the termination variable is a problem, or if in practice we wish to avoid the bottleneck of every process over a distributed system needing access to a single global termination variable, we can use a version in which each process has its own termination variable, and then **choose** has a multiple role – as well as passing solutions upwards in the tree, it passes termination signals downwards. The program below does this:

```
search(Term, State, Sols) :-
   issolution(State, Flag),
   expand(Flag, Term, State, Sols).

expand(Flag, found, State, Sol) :- Sol=none.
expand(true, Term, State, Sol) :- unknown(Term) | Sol=State.
expand(false, Term, State, Sol) :- unknown(Term) |
   successors(State, States),
   branch(Term, States, Sol).

branch(Term, [], Sol) :- Sol=none.
branch(Term, [State|Siblings], Sol) :-
   search(Term1, State, Sol1),
   branch(Term2, Siblings, Sol2),
   choose(Term, Term1, Term2, Sol1, Sol2, Sol).
```

The six argument version of **choose**, including termination variables is:

```
choose(found, Term1, Term2, _, _, Sol) :-
   Term1=found, Term2=found, Sol=none.
choose(_, _, Term2, Sol1, _, Sol) :- Sol1≠none |
   Term2=found, Sol=Sol1
choose(_, Term1, _, _, Sol2, Sol) :- Sol2≠none |
   Term1=found | Sol=Sol2
choose(_, _, _, none, none, Sol) :-
   Sol=none.
```

The effect of executing this program is to set up a binary tree of **choose** goals in which each arc is represented by two variables. The arrows show the direction of the flow of data in these variables.



Note that some versions of KL1 do not have an **unknown** test as a primitive, but provide a way of expressing a preference between clauses when expanding a goal, using the keyword **alternatively**. A goal will only commit to a clause textually below an **alternatively** if at the time of attempting to commit it cannot commit to one above, possibly because of insufficient variable binding. It will not necessarily wait, however, for a variable to becomes bound if this would hold up its expansion, rather in the absence of the information in will move on to using clauses below the **alternatively**. In this way it differs from the keyword **otherwise**, also found in KL1; a goal will not commit to a clause following an **otherwise** unless its arguments are sufficiently bound to show that it cannot commit to any clauses before it. The version of the above **expand** which uses **alternatively** rather than **unknown** is:

```
expand(Flag, found, State, Sol) :- Sol=none.
alternatively.
expand(true, Term, State, Sol) :- Sol=State.
expand(false, Term, State, Sol) :-
   successors(State, States),
   branch(Term, States, Sol).
```

The short-circuit all-solutions search program may also be used as the basis for a speculative search program with a cutoff when a solution is found. The following program will accomplish this:

```
search(Acc,_,Sols) :- data(Acc) | Sols=Acc.
search(Acc,State,Sols) :- unknown(Acc) |
   issol(State,Flag), expand(Flag,Acc,State,Sols).

expand(true,Acc,State,Sols) :- Sols=[State|Acc].
expand(false,Acc,State,Sols) :-
   successors(State, Succs), branch(Acc,Succs,Sols).

branch(Acc,[],Sols) :- Sols=Acc.
branch(Acc,[State|Siblings],Sols) :-
   search(Acc,State,Sols1), branch(Sols1,Siblings,Sols).
```

where `data(X)` cannot fail, but if `X` is unbound suspends until it becomes bound. The initial call to this program should be `search(Acc,State,Sols)`, `complete(Sols,Acc)`, where **complete** is defined by:

```
complete(Sols,Acc) :- data(Sols) | Acc=[].
complete(Sols,Acc) :- Sols=Acc | Acc=[].
```

The result is that when a solution is found, any search which remains to be done is cut off and the circuit is linked up to the right of the solution in the search tree, with **complete**

completing the circuit and halting any remainining search to the solution's left. It is assumed the = test in the second clause for `complete` will succeed if its two arguments are the same unbound variable, and suspend if they are two different unbound variables. This will prevent deadlock in the case where there are no solutions. Since it is possible for a solution to be found in one place and linked in to the list of solutions before news of the finding of a solution elsewhere is received, the output of this search will be a short list containing at least one but possibly more solutions unless there are no solutions.

A Direct KL1 Solution Using Priorities

We still have not dealt with the problem of unrestricted parallelism. At any point in the execution of a search any of the goal which is sufficiently bound to rewrite could do so. In practice, `choose` goals will remain suspended waiting for solutions or termination variables, so it will be the `search` goals at the leaves of the tree that will be rewriting. We know that at any time some of the `search` goals are more likely to lead soon to a solution state than others. In the absence of architectural limitations on the amount of parallelism available, we would prefer that if a processor has a choice of `search` goals which it can expand, it will pick the one most likely to lead to a solution. In many problems we will have a heuristic which tell us which one it is.

Once we have found a solution, we want the `choose` goals which pass it upwards and termination values downwards to the rest of the tree to have priority over the `search` goals which are expanding the tree. Otherwise we have the undeadlock situation where expanding `search` goals take over the resources that would otherwise be used by the `choose` goals which would pass on the message that a solution has been found and no further expansion is necessary.

Our preferred solution is to control the parallelism through the priority pragma of KL1. The idea of introducing priorities into concurrent logic languages was arrived at independently by the author [Hunt 88], [Hunt 91] and the ICOT group developing the parallel language KL1 [Ue & Ch 90]. Our work was influenced by Burton's use of a similar priority pragma in functional languages [Burt 85]. The priority pragma may be considered a practical implementation of the precedence relationship between subproblems in a General Problem Solving Algorithm [RSMB 88]: the precedence relationship establishes a partial ordering among subproblems, which in practice may be modelled by assigning them a numerical priority. The precedence gives an ordering of the likelihood of a subproblem contributing to the overall solution.

The idea is that should two goals be located on a single processor with its own memory, and both goals have variables sufficiently bound so that they can rewrite, the one with the highest priority will always be rewritten first. If N processors share a memory, and there are more than N rewriteable goals in that memory, then the N highest priority goals will be rewritten. It is assumed that an underlying load-balancing mechanism will give a reasonable distribution of high priority goals across processors. The alternative would be a global priority list shared out across the processors, but this would be an expensive bottleneck on a non-shared memory system. Otherwise the programmer need not be concerned with process-mapping. This means that a program employing priorities may be ported with ease between a variety of architectures.

Load-balancing between processors in a parallel system may be compared to such things as register allocation in a standard sequential system. In both cases, although a program could be made more efficient by giving the programmer direct control over allocations, we believe it is best to free the programmer from such low-level considerations, or at least to confine them to a separate program layer so that the abstract structure of the algorithm is not buried within code dealing with the allocations.

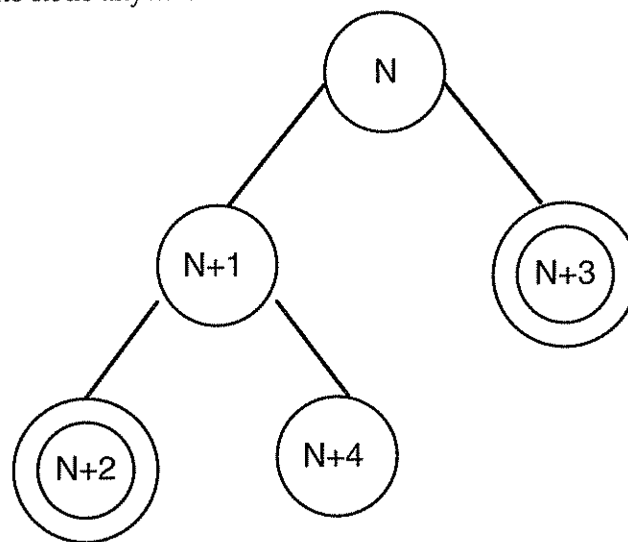
The syntax for priorities writes goals as `Goal@priority(P)` where `Goal` is any KL1 goal, and `P` is a real number, or a KL1 variable that will be bound to a real (in the latter case the goal will be suspended until P is bound). When `Goal` is sufficiently bound to rewrite, `Goal@priority(P)` will rewrite in just the same way as `Goal` would, providing there are no other rewriteable goals with higher priorities sharing the same processor. The priority call is an unobtrusive addition to a program which enables the programmer to write the program as if the maximum parallelism is available, and add annotations at a later stage to cover the fact that in practice parallelism is limited. To add priorities to our existing search program, we need simply change the clause in branch which sets up a search goal. If we have a predicate **heuristic** which takes as its first argument a state in the search space, and returns in its second argument a heuristic value associated with that state, the following new clause for **branch** suffices:

```
branch(Term, [State|Siblings], Sol) :-
   heuristic(State,H), P:=0-H,
   search(Term1,State,Sol1)@priority(P),
   branch(Term2,Siblings,Sol2),
   choose(Term,Term1,Term2,Sol1,Sol2,Sol).
```

By convention, the lower the heuristic, the closer the state is to a solution, hence the priority is the negation of the heuristic.

Search anomalies

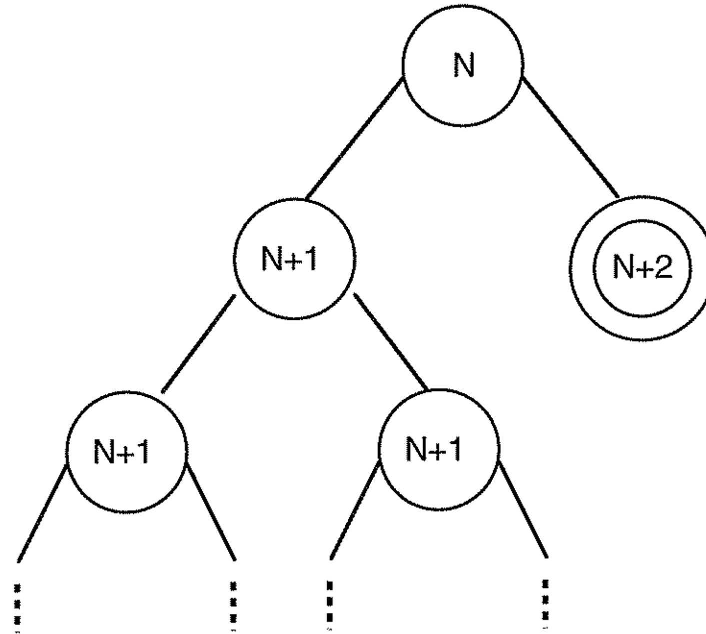
On a single processor system, our search program with priorities will degenerate to a situation where the implicit priority queue of **search** processes matches exactly the explicit priority queue of states in the meta-programming approach above. It is well known that if the heuristic associated with any state is a lower bound on the cost of all solutions reachable through that state (a heuristic with this property is termed *admissible*), then the first solution found by expanding the search tree in the heuristic order will be the optimal solution. We cannot guarantee this on a multi-processor system however, even if the priority list is shared. Consider the situation where the following subtree is part of the search tree, where double circled nodes indicate solution nodes, the values in the nodes are the heuristics and no node anywhere else in the tree has a heuristic less than N+5.



On a single processor system, the node with heuristic N will be expanded first, leading to the two nodes with heuristic N+1 and N+3 being at the head of the priority queue. The node with heuristic N+1 will then be expanded leading to the three nodes with heuristic N+2, N+3 and N+4 being at the head of the priority queue. The node with heuristic N+2 will then be expanded and found to be a solution.

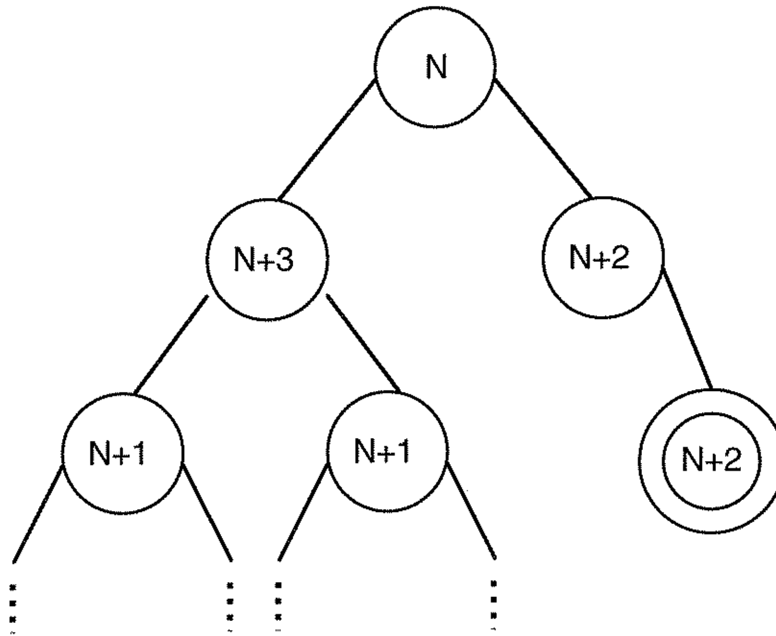
On a two-processor system, however, the node with heuristic N and some other node (with, as said, heuristic N+5 or more) will both be expanded. This will lead to the nodes with heuristics N+1 and N+3 being at the head of the priority queue. Both will be expanded and the node with heuristic N+3 found to be a solution. Clearly there is more scope for this sort of anomaly if there is a separate priority queue for each processor, since this leads to the possibility for example, that where we have three nodes with heuristics N, N+1 and N+2 on a two processor system, it may be the case that the nodes with heuristics N and N+1 are on the same processor, and so the nodes with heuristics N and N+2 are expanded, so we lose the strict order of priority.

To see how a multiprocessor search can lead to arbitarily large speedup or slowdown, consider first the case where the descendants of a state with heuristic N are a solution state with heuristic N+2 and a non-solution state with heuristic N+1 which has an arbitarily large number of non-solution descendants all of heuristic N+1 with no other state having a heuristic less than N+3:



On a single processor, the state with heuristic N will be expanded, putting the top state with heuristic N+1 and the goal state with heuristic N+2 at the head of the priority queue. The state with heuristic N+1 will then be expanded putting two more states with heuristics N+1 at the head of the priority queue, and so on for an arbitary length of time, and only when the N+1 heuristic states are exhausted is the N+2 heuristic state tried and found to be a solution. On a two processor system the top N+1 and the N+2 heuristic state will be expanded, and the N+2 heuristic state found to be a solution and returned, without any further search needed. Since the number of N+1 nodes that are no longer considered is arbitarily high, the speedup with two processors is arbitarily high, not restricted to 2.

Deceleration anomalies or slowdowns occur when a multiprocessing system causes a node to be expanded that would not be considered in a single processor system, but on expansion it reveals promising-looking descendants which do not though have any solutions in them. For example, suppose in the following tree, the N+1 heuristic states have an arbitary number of descendants all with heuristic N+1 but with no solutions among them. On a single processor they would never be reached since on expansion of the N heuristic node, the N+2 and N+3 heuristic nodes would be produced, and the N+2 expanded first. This would reveal its successor N+2 heuristic node, which again would be expanded before the N+3 node and found to be a solution.



On a two processor system, however, after the expansion of the initial N heuristic node, both its successors would be expanded, resulting in the top two N+1 heuristic nodes being at the head of the priority queue, followed by the N+2 heuristic goal state. Since there are an arbitary number of N+1 heuristic desendants of the top N+1 heuristic nodes, the two processor system would spend an arbitarily long amount of time searching them, before find ing the N+2 heuristic solution.

As Lai and Sahni point out [La & Sa 84], although the conditions which produce these anomalies look unusual, in practice it is quite common for nodes in a search tree surrounding a solution to have heuristic values which are very close to each other. Therefore they are a factor which needs to be taken into account. Fortunately, the conditions which lead to superlinear speedup are more common than those that lead to slowdown, so, as McBurney and Sleep show experimentally [McB & S 87], it is not unusual for state space search with N processors to result in a speedup of more than N. Lie and Wah [Li & Wa 84] describe conditions on heuristics which assure that the detrimental slowdowns that are possible with parallelism cannot occur, and also conditions which are necessary for superlinear speedup. The possibility of superlinear speedups led Kornfeld to suggest that the use of a parallel language and pseudo-parallel execution could improve algorithm efficiency, even when actually executed on a single processor [Korn 82]. The improvement which Stockman notes his SSS* algorithm gives over the standard alpha-beta search of game trees is due to the pseudo-parallel search of the game tree which SSS* uses in the place of alpha-beta search's strict depth-first left-to-right approach [Stoc 79].

The more usual situation, however, is that search on a multiprocessor system will yield a speedup, but one less than the number of processors. In addition to dealing with the

overhead of the parallelism, if some of the parallelism is speculative the total amount of work done in a parallel search will be greater than that done with the same problem on a single processor, due to parts of the tree being searched that would not be considered in a single processor search. If there is no speculative computation, for example in a search for all solutions, any failure of the speedup to reach the number of processors will be due just to the overhead of parallelism, since the total amount of work done on the problem itself will not change.

Branch-and-Bound Search

As we have shown, on a multiprocessor system, even if we have admissible heuristics, we cannot guarantee that the first solution found is the lowest cost one. Its cost, however, is obviously an upper bound on the lowest cost solution. If we have a way of determining a lower bound on the cost of any solution reached through a particular state, then if that lower bound is greater than or equal to the cost of the best solution so far we needn't expand the state since we know it will not give us a better solution. The point at which search of a subtree is abandoned because its lower bound is greater than a known upper bound is known as a "cut-off". This method of searching with bounds is known as "branch-and-bound" search [La & Wo 66]. It is often the case that a heuristic used for search is also a lower bound, that is so with the previously described heuristic for the 8-puzzle, since to move all cells into position requires at the minimum the number of squares each cell is out of position.

A branch-and-bound search program will be based on an all-solutions program, returning a stream of solutions, though the cutoffs will stop many of the solutions being reached and joining the stream. A simple version of a branch-and-bound program has a manager process to which all solutions are sent and which keeps an account of the lowest cost solution found so far. Since the cost of this solution is an upper bound, the manager also receives requests from the search processes for the current upper bound. So the manager will deal with `solution` messages which pass on new solutions and their associated costs, and `request` messages which request the current upper bound and return it in a reply variable. We replace the **choose** goals of single-solution search with stream merges. There is no need for any other termination mechanisms, so we do not need to use the extra-logical **unknown**.

The following program implements this:

```
search(State,Messgs) :-
   Messgs=[request(UpBound)|Messg1],
   lowerbound(State,LoBound), cutoff(State,LoBound,UpBound,Messgs1).

cutoff(State,LoBound,UpBound,Messgs) :- LoBound>=UpBound | Messgs=[].
cutoff(State,LoBound,UpBound,Messgs) :- LoBound<UpBound |
   issolution(State,Flag), expand(Flag,State,Messgs).

expand(true,State,Messgs) :-
   cost(State,Cost), Messgs=solution(State,Cost):[].
expand(false,State,Messgs) :-
   successors(State, Succs), branch(Succs, Messgs).

branch([], Messgs) :- Messgs=[].
branch([State|Siblings], Messgs) :-
   heuristic(State,H), search(State,Messgs1)@priority(H),
   branch(Siblings,Messgs2),
   merge(Messgs1, Messgs2, Messgs).
```

It is assumed that **cost** gives the cost associated with a solution state, and **lowerbound** gives the lower bound on solutions reachable through a given state. Note that since **cutoff**, the predicate that either cuts off search or continues as appropriate, contains an arithmetic test using `LoBound` in its guards it will suspend until the lower bound has been returned in the reply variable from the manager. The **manager** clauses are:

```
manager([],ASolution,ACost,BestSol,LowestCost) :-
   BestSol=Solution, LowestCost=Cost.
manager([request(UpBound)|Messgs],ASol,ACost,BestSol,LowestCost) :-
   UpBound=ACost,
   manager(Messgs,ASol,ACost,BestSol,LowestCost).
manager([solution(Sol1,Cost1)|Messgs],Sol2,Cost2,BestSol,LowCost) :-
   Cost1<Cost2 |
   manager(Messgs,Sol1,Cost1,BestSol,LowestCost).
manager([solution(Sol1,Cost1)|Messgs],Sol2,Cost2,BestSol,LowCost) :-
   Cost1>=Cost2 |
   manager(Messgs,Sol2,Cost2,BestSol,LowestCost).
```

The system is set up with initial calls:

```
search(InitState,Messgs), manager(Messgs,Dummy,_,BestSol,LowestCost)
```

where `Dummy` is a value which is known to be above any solution cost that might be found.

Here we could use normal stream merger, but some efficiency gains could be made with a more sophisticated merger. If two `request` messages are being sent, they can be merged into one by unifying their reply variables, thus reducing the number of messages that need to be

dealt with. If two solution messages are being sent, only the one with lowest cost needs to be sent further. Making these considerations gives the following **merge**:

```
merge([request(R1)|S1],request(R2):S2,S) :-
   R1=R2, merge([request(R1)|S1],S2,S).
merge([request(R1),request(R2)|S1],S2,S) :-
   R1=R2, merge([request(R1)|S1],S2,S).
merge(S1,[request(R1),request(R2)|S2],S) :-
   R1=R2, merge([request(R1)|S1],S2,S).
merge([request(R)|S1],S2,S) :-
   S=[request(R)|S3], merge(S1,S2,S3).
merge(S1,[request(R)|S2],S) :-
   S=[request(R)|S3], merge(S1,S2,S3).
merge([solution(Sol1,Cost1)|S1],[solution(Sol2,Cost2)|S2],S) :-
   Cost1<Cost2 |
   merge(solution(Sol1,Cost1):S1,S2,S).
merge([solution(Sol1,Cost1),solution(Sol2,Cost2)|S1],S2,S) :-
   Cost1<Cost2 |
   merge(solution(Sol1,Cost1):S1,S2,S).
merge(S1,[solution(Sol1,Cost1),solution(Sol2,Cost2)|S2],S) :-
   Cost1<Cost2 |
   merge([solution(Sol1,Cost1)|S1],S2,S).
merge([solution(Sol1,Cost1)|S1],[solution(Sol2,Cost2)|S2],S) :-
   Cost1>=Cost2 |
   merge([solution(Sol2,Cost2)|S1],S2,S).
merge([solution(Sol1,Cost1),solution(Sol2,Cost2)|S1],S2,S) :-
   Cost1>=Cost2 |
   merge([solution(Sol2,Cost2)|S1],S2,S).
merge(S1,[solution(Sol1,Cost1),solution(Sol2,Cost2)|S2],S) :-
   Cost1>=Cost2 |
   merge(solution(Sol2,Cost2)S1,S2,S).
merge([solution(Sol,Cost)|S1],S2,S) :-
   S=[solution(Sol,Cost)|S3], merge(S1,S2,S3).
merge(S1,[solution(Sol,Cost)|S2],S) :-
   S=[solution(Sol,Cost)|S3], merge(S1,S2,S3).
```

An alternative way of dealing with parallel branch-and-bound search takes a similar approach to that we used to avoid the bottleneck of a global termination variable on the search for a single solution. In this case, each node in the tree of **merge** processes created by the search will store its own upperbound limit. When a lower cost solution is received from one branch, the limit is updated, the new solution sent upwards in the tree, and the new lower bound sent downwards on the other branch. When a solution reaches a **merge** process with a cost higher than the current limit, it is not sent any further. It can thus be seen that a lower bound will gradually diffuse through the tree when it is found. The complete program for this branch-and-bound search with distributed limits is:

```
search(InCosts,State,Sols) :-
   lowerbound(State,LoBound), cutoff(LoBound,InCosts,State,Sols).

cutoff(LoBound,[UpBound1,UpBound2|UpBounds],State,Sols) :-
   LoBound<UpBound1 |
   cutoff(LoBound,[UpBound2|UpBounds],State,Sols).
cutoff(LoBound,[UpBound|UpBounds],State,Sols) :-
   LoBound >= UpBound | Sols=[].
cutoff(LoBound,[UpBound|UpBounds],State,Sols) :-
   unknown(UpBounds), LoBound<UpBound |
   issol(State,TV), expand(TV,[UpBound|UpBounds],State,Sols).

expand(true,Bounds,State,Sols) :-
   cost(State,Cost), Sols=[State/Cost]
expand(false,Bounds,State,Sols) :-
   successors(State,States), branch(Bounds,States,Sols).

branch(Bounds,[],Sols) :- Sols=[].
branch([Bound|Bounds],[State|Siblings],Sols) :-
   heuristic(State,H),
   search([Bound|Bounds1],State,Sols1)@priority(H),
   branch([Bound|Bounds2],Siblings,Sols2),
   merge(Dummy,[Bound|Bounds],Bounds1,Bounds2,Sols1,Sols2,Sols).

merge(Lim,[Bnd1,Bnd2|InBnds],Bnds1,Bnds2,Sols1,Sols2,Sols) :-
   merge(Lim,[Bnd2|InBnds],Bnds1,Bnds2,Sols1,Sols2,Sols).
merge(Limit,[Bnd|InBnds],Bnds1,Bnds2,Sols1,Sols2,Sols) :-
   unknown(InBnds), Bnd<Limit |
   Bnds1=[Bnd|NewBnds1], Bnds2=[Bnd|NewBnds2],
   merge(Bnd,InBnds,NewBnds1,NewBnds2,Sols1,Sols2,Sols).
merge(Limit,[Bnd|InBnds],Bnds1,Bnds2,Sols1,Sols2,Sols) :-
   unknown(InBnds), Bnd>=Limit |
   merge(Limit,InBnds,Bnds1,Bnds2,Sols1,Sols2,Sols).
merge(Limit,InBnds,Bnds1,Bnds2,[Sol/Cost|Sols1],Sols2,Sols) :-
   unknown(InBnds), Cost<Limit |
   Bnds2=[Cost|NewBnds2], Sols=[Sol/Cost|NewSols],
   merge(Cost,InBnds,Bnds1,NewBnds2,Sols1,Sols2,NewSols).
merge(Limit,InBnds,Bnds1,Bnds2,Sols1,[Sol/Cost|Sols2],Sols) :-
   unknown(InBnds), Cost<Limit |
   Bnds1=[Cost|NewBnds1], Sols=[Sol/Cost|NewSols],
   merge(Cost,InBnds,NewBnds1,Bnds2,Sols1,Sols2,NewSols).
merge(Limit,InBnds,Bnds1,Bnds2,[Sol/Cost|Sols1],Sols2,Sols) :-
   Cost>=Limit |
   merge(Limit,InBnds,Bnds1,Bnds2,Sols1,Sols2,Sols).
merge(Limit,InBnds,Bnds1,Bnds2,Sols1,[Sol/Cost|Sols2],Sols) :-
   Cost>=Limit |
   merge(Limit,InBnds,Bnds1,Bnds2,Sols1,Sols2,Sols).
```

```
merge(Limit,InBounds,Bounds1,Bounds2,[],Sols2,Sols) :-
    Sols=Sol2, Bounds2=InBounds.
merge(Limit,InBounds,Bounds1,Bounds2,Sols1,[],Sols) :-
    Sols=Sols1, Bounds1=InBounds.
```

To avoid repeated calculations of the cost associated with a state, here costs are passed around in pairs with solutions. The result of executing the program will be to create a tree of **merge** processes, each of which will have one incoming stream of lower bounds, two outgoing streams of lower bounds, two incoming streams of solution/cost pairs, and one outgoing stream of solution/cost pairs, all streams being in decreasing order. Note the use of **unknown** to check that the latest available bound from the stream of incoming bounds is being used – when this is the case, **unknown** will succeed when it takes as argument the variable storing the rest of the stream.

Game Tree Search

Games-playing programs (at least with games like chess) are another form of state-space search in which we have a tree of states with the root being the initial state, its descendants generated by all possible moves from this state, and the descendants of its descendants generated from them likewise. In a complete game tree, the leaves represent board conditions where no further moves may be made thus (ignoring the possibility of games with draw positions) the leaves may be labelled "win" or "lose". The root state in the search tree, if it is not a leaf, is labelled "win" if at least one of its descendant states is labelled "win" (since by making the move which leads to that state a winning position is reached), and is labelled "lose" if none of the descendants is labelled "win" (since whatever move is made, it is not possible to reach a winning position). The labelling of the descendant subtrees from the root is done differently since they represent positions in which it is the opponent trying to force a "lose" position, so they are labelled "lose" if they have themselves any subtree labelled "lose" and "win" if they have no subtree labelled "lose". The next level down is labelled as the root level, and so on alternately down to the leaves. For this reason, game trees are a form of AND/OR tree, since if we treat the win/lose values as booleans we have trees of logic terms where the odd level branches are conjunctions and the even level disjunctions.

In the search of a complete game tree, it can be seen that when labelling nodes at an even level in the tree as soon as search of one subtree returns a win, it is unnecessary to search the rest since we have already found a path that leads to a certain win. Similarly, at even level as soon as one subtree returns lose it is unnecessary to search the rest. So at any point if we decide to search more than one branch of the tree in parallel we are engaging in

speculative computation since we do not know whether the search of the second and subsequent branches will be necesary or not to return a solution.

We are therefore in a similar situation to that we were in with standard search for a solution in a tree – either we take a cautious approach and lose all opportunity for parallelism, since all the parallelism in the problem is speculative, or we ignore the fact that the parallelism is speculative and thus run into the problem of overwhelming the system with speculative computations. The simple solution, with all speculative search possible, but no cutting off of speculative computations found unnecessary is:

```
player(State, Val) :-
   isleaf(State, L), playermove(L, State, Val).

playermove(true, State, Val) :- eval(State, Val).
playermove(false, State, Val) :-
   successors(State, Succs), or(Succs, Val).

or([], Val) :- Val=lose.
or([State|Siblings], Val) :-
   opponent(State,Val1), or(Siblings,Val2), either(Val1,Val2,Val).

either(win, _, Val) :- Val=win.
either(_, win, Val) :- Val=win.
either(lose, lose, Val) :- Val=lose.

opponent(State, Val) :-
   leaf(State, L), opponentmove(L, State, Val).

opponentmove(true, State, Val) :- eval(State, Val).
opponentmove(false, State, Val) :-
   successors(State, Succs), and(Succs, Val).

and([], Val) :- Val=win.
and([State|Siblings], Val) :-
   player(State,Val1), and(Siblings,Val2), both(Val1,Val2,Val).

both(lose, _, Val) :- Val=lose.
both(_, lose, Val) :- Val=lose.
both(win, win, Val) :- Val=win.
```

An alternative formulation of game tree search labels nodes "win" or "lose" depending on whether they represent a win or lose state for the player who is next to move at the node itself, rather than always for the player at the root of the tree. In this case a non-leaf node is labelled "win" if any of its successors are labelled "lose" and is labelled "lose" if none of its successors are labelled "lose". This leads to a simpler algorithm and program, as it is no

longer necessary to consider two different sorts of nodes, though it is perhaps not so intuitive:

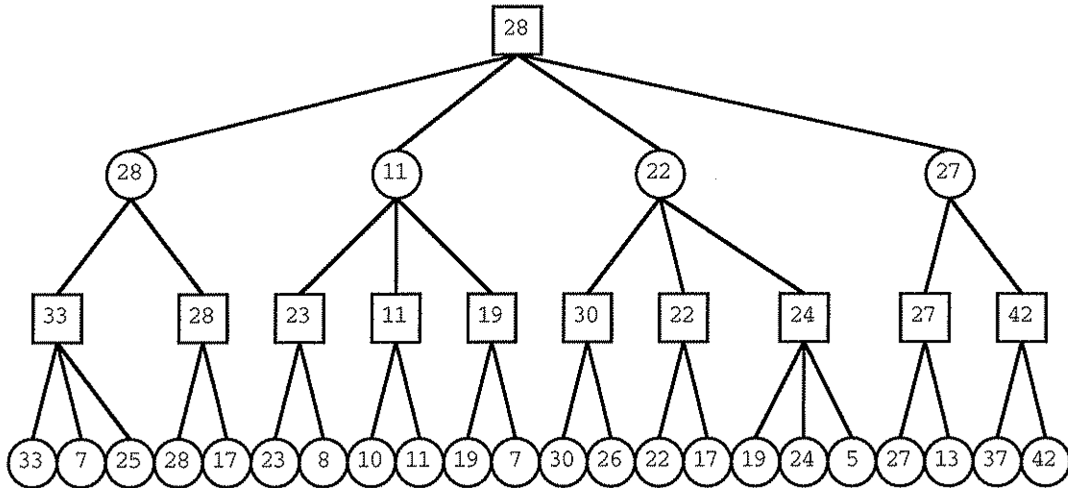
```
search(State, Val) :-
   isleaf(State, L), expand(L, State, Val).

expand(true, State, Val) :- eval(State, Val).
expand(false, State, Val) :-
   successors(State, Succs), branch(Succs, Val).

branch([], Val) :- Val=win.
branch([State|Siblings], Val) :-
   game(State,Val1), branch(Siblings,Val2), combine(Val1,Val2,Val).

combine(lose, _, Val) :- Val=win.
combine(_, lose, Val) :- Val=win.
combine(win, win, Val) :- Val=lose.
```

It can be seen that this program is a version of our generic search program, and thus we may use termination variables as previously to halt speculative search which is found to be unnecessary.

Minimax and Alpha-Beta Search

In practice, apart from that subfield of games-playing concerned with end-game situations, game trees are usually too large for a complete analysis. Rather a state which is not an end position in the game may be treated as a leaf and assigned a numerical value which represents an estimated measure of the likelihood of that state being a win position in a full analysis. Typically, all states at a given depth in the tree will be taken as leaves and assigned a numerical value. Then, on similar principles to full game trees above, a value for the root node and every non-leaf node at an even depth in the tree is determined by selecting the maximum value of its descendants, while the value for every non-leaf node at odd depth is determined by selecting the minimum value of its descendants. Hence the name "minimax" search. At the root level, the move corresponding to the subtree with highest value is given as the next move to make since this value is the maximum value which the player can force the opponent to concede to, and making this move forces this concession.

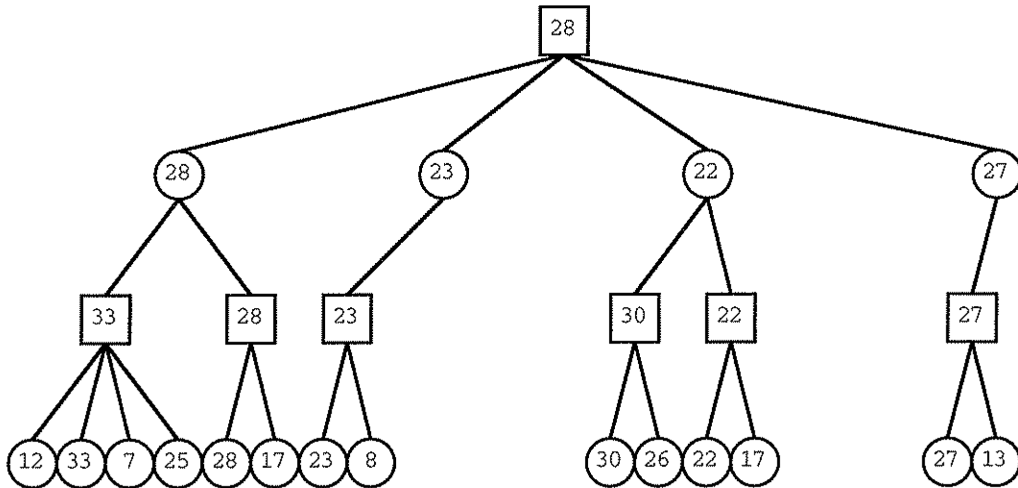
For example, the following is a complete tree, assuming nodes at depth 3 are treated as leaves and evaluated according to the evaluation function, with the values of the internal nodes worked out using the minimax procedure. The convention is that square boxes represent maximising nodes, and round boxes minimising nodes.



In this case, as the highest value at the root is supplied by its leftmost branch, the move represented by the leftmost branch is the estimated best move to make since, at least estimating the position three plays ahead, this is the move that can guarantee leaving the player in the best position whatever moves the opponent makes.

It is, however, the case that it is not necessary to search the whole tree. In the above example, consider the case where a sequential depth-first left-to right evaluation of the tree is taking place. When the leftmost descendant tree of the root has been found to return a value of 28, since the root node is maximising, and its descendants are minimising, if any of the descendants of the descendants should evaluate to less than 28 as its parent is minimising it cannot return a value greater than 28, and so its value cannot be picked as the maximum. So when the first subbranch of the second branch returns a value of 23, it is not necessary to search the rest of the branch since at most it will return a value of 23, which will not exceed the 28 already established. With the third branch, its third subbranch's return of 30 indicates a possibility of a better choice, but when the second subbranch of the third branch returns 22 the possibility that the moves represented by the third branch is a better choice is lost.

So the tree only needs to be searched to the extent shown below:



A formalisation of these considerations of cutoffs in game-tree search, together with a change to the formulation similar to that we had with complete game-trees to avoid having to distinguish between maximising and minimising nodes, gives us the well-known alpha-beta algorithm [Kn & Mo 75] for estimated game-tree search. This may be described in pseudo-code form by:

```
function alphabeta(p: position; α,β: valuation):valuation=
{if leaf(p) then return(value(p)) else
  {for each successor position of p, p1 to pw do
    {α:=max(α,–alphabeta(pi,–β,–α));
    if α≥β then return(α)
   };
  return(α)
 }
}
```

The evaluation is set off with α and β set to $-\infty$ and ∞ respectively. α may be increased by recursive calls, but if it is increased to the value of β or beyond, a cutoff position has been reached and no further evaluation is done as it is not necessary. When α is increased, a reduced value for β is passed to recursive calls, which again may cause cutoffs. The difference between α and β is described as the "alpha-beta window".

We can give a KL1 version of this alphabeta algorithm:

```
alphabeta(Position, Alpha, Beta) :-
   isleaf(Position,Flag), expand(Flag,Position,Alpha,Beta,Eval).

expand(true, Position, _, _, Eval) :-
   evaluate(Position, Eval).
expand(false, Position, Alpha, Beta, Eval) :-
   successors(Position, Succs),
   branch(Alpha, Beta, Succs, Eval).

branch(Alpha, Beta, [], Eval) :- Eval=Alpha.
branch(Alpha, Beta, _, Eval) :- Alpha>=Beta | Eval=Alpha.
branch(Alpha, Beta, Pos:Siblings, Eval) -
   NAlpha:=0-Alpha, NBeta:=0-Beta,
   alphabeta(Pos, NBeta, NAlpha, PosEval),
   max(PosEval, Alpha, Alpha1),
   branch(Alpha1, Beta, Siblings, Eval).
```

but like the pseudo-code algorithm, it will be sequential since the evaluation of any branch is not started until the evaluation of its left sibling has completed and returned an alpha value. In fact, it can be seen that the program follows the general pattern of the previous linear search programs.

Parallel Game Tree Search

The sequential nature of alpha-beta search comes about from the desire to avoid speculative computation. Since search of any one branch is likely to narrow the alpha-beta window, it makes sense to wait until it has completed before searching the next branch. This is particularly so because in most cases the branches in a game tree search can be ordered so that there is a strong likelihood that search of the initial few branches will so reduce the alpha-beta window as to make search of the rest unnecessary [Ma & Ca 82]. It is thus particularly important that the parallelism in parallel game tree search is ordered so that that most likely to contribute to a solution is given priority.

The usual approach to parallel alpha-beta search is described as "tree-splitting" [Fi & Fi 83], which works in a similar way to the parallel tree search we have considered previously, setting up processes to search every subtree in parallel. A different approach to parallel alpha-beta search which we shall not consider here divides the alpha-beta window up among processors and sets up multiple searches of the complete tree with the different subwindows [Baud 78]. In tree-splitting alpha-beta search when one subtree terminates and returns alpha value, if this results in an updating of alpha, the new alpha-beta window is sent to those branch searches which have not yet terminated. This means that search processes also have to take account of the possibility that they may receive updated alpha-beta windows from their parents, if so they have to be passed down to their

children, and search is halted if they cause alpha to exceed beta. Huntbach and Burton [Hu & Bu 88] develop the tree-splitting algorithm of Fishburn and Finkel by moving away from an approach which depends on a particular tree structured architecture towards a more abstract virtual tree approach [Bu & Hu 84] in which a tree of potentially parallel processes matching the search tree is constructed, but it is left to the underlying system to map it onto a physical architecture. The algorithm is described in a pseudo-code style using asynchronous remote procedure calls:

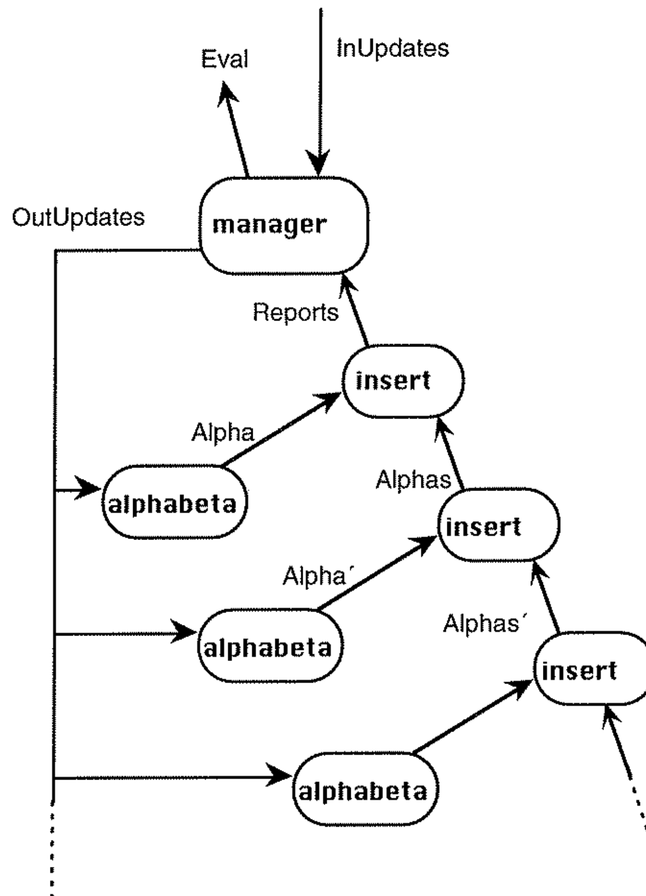
```
process alphabeta(p:position; α,β:valuation):valuation=
{
    asynchronous procedure report(α1:valuation)=
    {if α1≥α then
        {α:=α1;
        for all remaining children do child.update(−β,−α);
        if α≥β then terminate
        }
    }

    asynchronous procedure update(α1,β1:valuation)=
    {α:=max(α,α1);
    β:=min(β,β1);
    for all remaining children do child.update(−β,−α);
    if α≥β then terminate
    }

 for each successor position of p, p1 to pw do
    setup new process alphabeta(pi,−β,−α);
 wait until no remaining children;
 parent.report(−α);
 terminate
}
```

Here report is the procedure which send new alpha values upwards, note only when all children have terminated, while update is the procedure which sends new alpha and beta values downwards and is executed whenever new alpha or alpha and beta values are received. The following gives a direct translation of this algorithm into KL1:

```
alphabeta(Pos,_,Alpha,Beta,Eval) :- Alpha>=Beta | Eval=Alpha.
alphabeta(Pos,[Alpha1/Beta1|Updates],Alpha,Beta,Eval) :-
    max(Alpha,Alpha1,Alpha2), min(Beta,Beta1,Beta2),
    alphabeta(Pos,Updates,Alpha2,Beta2,Eval).
alphabeta(Pos,Updates,Alpha,Beta,Eval) :-
    unknown(Updates), Alpha<Beta |
    isleaf(Position,Flag),
    expand(Flag,Position,Updates,Alpha,Beta,Eval).

expand(true,Position,_,_,_,Eval) :-
    evaluate(Position,Eval).
expand(false,Position,Updates1,Alpha,Beta,Eval) :-
    successors(Position,Succs),
    branch(Updates1,Alpha,Beta,Succs,Reports),
    manager(Updates,Alpha,Beta,Reports,Updates1,Eval).

branch(UpDates,Alpha,Beta,[],Reports) :- Reports=[].
branch(UpDates,Alpha,Beta,[Pos|Siblings],Reports) :-
    heuristic(Pos,H),
    alphabeta(Pos,UpDates,Alpha,Beta,Report)@priority(H),
    branch(UpDates,Alpha,Beta,Siblings,Reports1),
    insert(Report,Reports1,Reports).

insert(Alpha,Alphas,Alphas1) :- data(Alpha) |
    Alphas1=[Alpha|Alphas].
insert(Alpha,[Alpha1|Alphas],Alphas1) :-
    Alphas1=[Alpha1|Alphas2],
    insert(Alpha,Alphas,Alphas2).
insert(Alpha,[],Alphas) :- Alphas=[Alpha].

manager(InUpdates,Alpha,Beta,[Alpha1|Reports],OutUpdates,Eval) :-
    Alpha1>Alpha |
    NAlpha:=0-Alpha1, NBeta:=0-Beta,
    OutUpdates = [NBeta/NAlpha|OutUpdates1],
    manager(InUpdates,Alpha1,Beta,Reports,OutUpDates1,Eval).
manager(InUpdates,Alpha,Beta,[Alpha1|Reports],OutUpdates,Eval) :-
    Alpha1<=Alpha |
    reports(InUpdates,Alpha,Beta,Reports,OutUpdates,Eval).
manager([Alpha1/Beta1|InUpdates],Alpha,Beta,Reports,OutUpdates,Eval):-
    max(Alpha,Alpha1,Alpha2),
    min(Beta,Beta1,Beta2),
    NBeta:=0-Beta2,
    NAlpha:=0-Alpha2,
    OutUpdates = [NBeta/NAlpha|OutUpdates1],
    manager(InUpdates,Alpha2,Beta2,Reports,OutUpDates1,Eval).
manager(InUpdates,Alpha,Beta,[],OutUpdates,Eval) :-
    Eval=Alpha, OutUpdates=[].
```

Note again the use of **unknown** and **data** to ensure the latest alpha and beta values are used. Some improvements could be made at the cost of a rather more complex program (and one which is not such a direct translation of the algorithm in [Hu & Bu 88]), for example checks on update values would ensure that update messages are not forwarded when they do not change the alpha-beta window.

The process structure is complicated by the use of a single one-to-many stream used to pass updates down the tree from the **manager** process, and a merger of results using **insert** to insert individual reports into the stream of reports passed upwards into the manager. The situation after the expansion of an individual **alphabeta** process and before the expansion of any of its descendants is:



If the heuristic used is the depth in the tree, the search will revert to standard alpha-beta search on a single processor. As suggested in [Hu & Bu 88], ordering of the tree as in [Ma & Ca 82] can be obtained if the priority is a real number, composed of the sum of an integer representing the depth and a number between 0 and 1 representing the ordering evaluation of the position. Alternatively, if the integer part of the priority is a heuristic evaluation, and the mantissa based on the depth to distinguish between positions of the same heuristic at differing depths, a parallel version of the SSS* algorithm [Stoc 79] is obtained. In fact, Kumar and Kanal [Ku & Ka 83] give a general formulation which indicates that both alpha-beta search and SSS* can be fitted into the branch-and-bound model. Wilson [Wils

87] also notes that in a concurrent logic program apha-beta search and SSS* are simply variants of a common game-tree search program which depend on scheduling, though his program does not make use of priorities, instead relying on an explicit oracle process as part of the program which interacts with the search.

Conclusion

The inherently parallel nature of KL1 enables us to move attention away from some of the systems details of parallel search, so that it may be viewed in a more abstract form. We show that parallel search algorithms may easily be implemented in KL1, but also show some of the pitfalls. The simple nature of KL1 makes it a very suitable language to discuss abstract parallel algorithms without the distraction of having to be concerned with low-level issues of implementation. We gave a suite of programs that illustrate many of the issues arising from parallelising search algorithms. This suggests KL1 may be of particular use in education or as a prototyping language for parallel systems.

References

[Bal 91] H.E.Bal. Heuristic search in Parlog using replicated worker style parallelism. *Future Generation Computer Systems 6*, 4 pp.303-315.

[Baud 78] G.Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, PhD Dissertation, Carnegie Mellon University.

[Brat 86]. I.Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley.

[Bu & Hu 84]. F.W.Burton and M.M.Huntbach. Virtual Tree Machines. *IEEE Trans. Computers, C-33*, 3 pp.278-281.

[Bu & Ri 88] A.D.Burt and G.A.Ringwood. *The Binding Conflict Problem in Concurrent Logic.* Tech. Rep., Dept. of Computing, Imperial College, London.

[Burt 85]. F.W.Burton. Speculative computation, parallelism and functional programming. *IEEE Trans. on Computers C-34*, 12 pp.1190-1193.

[Burt 88]. F.W.Burton. Nondeterminism with referential transparency in functional programming languages. *Computer Journal 31*, 3 pp.243-247.

[Cl & Gr 87]. K.L.Clark and S.Gregory. Parlog and Prolog united. *4th Int. Conf. on Log. Prog.*, pp.243-247.

[Cl & Tä 77]. K.L.Clark and S-Å.Tärnlund. A first-order theory of data and programs. In *Information Processing 77: Proc. IFIP Congress*, pp.939-944.

[Co & Sh 87] M.Codish and E.Shapiro. Compiling OR-parallelism into AND-parallelism. *New Generation Computing 5*, 1 pp.45-61.

[Fi & Fi 83] R.Finkel and J.Fishburn. Parallelism in alpha-beta search. *Art. Int.19*, pp.89-106.

[Fo & Ta 89] I.Foster and S.Taylor. Strand: a practical parallel programming tool. *Proc. North Am. Conf. on Log. Prog.*, pp.497-512.

[Gr & Pa 81]. D.H.Grit and R.L.Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Trans. Prog. Lang. Sys. 3*, 1 pp.49-59.

[Greg 87]. S.Gregory *Parallel Logic Programming in PARLOG* Addison Wesley.

[Hari 90]. S.Haridi. A logic programming language based in the Andorra model. *New Generation Computing 7*, pp.109-125.

[Hu & Bu 88]. M.M.Huntbach and F.W.Burton. Alpha-beta search on virtual tree machines. *Information Sciences 44*, pp.3-17.

[Hu & Ri 95] M.M.Huntbach and G.A.Ringwood. Programming in concurrent logic languages. IEEE Software 12, 6 pp.71-82.

[Hunt 88] M.Huntbach. *Parlog as a Language for Artificial Intelligence.* Tech. Rep., Parlog Group, Dept. of Computing, Imperial College, London, UK.

[Hunt 91]. M.Huntbach. Speculative computation and priorities in concurrent logic languages. In *3rd UK Annual Conference on Logic Programming.* Published in Springer-Verlag Workshops in Computing series, eds. G.A.Wiggins, C.Mellish and T.Duncan, pp.23-35.

[Kn & Mo 75] D.Knuth and R.Moore. An analysis of alpha-beta pruning. *Art. Int. 6*, pp.293-326.

[Korn 82]. W.A.Kornfeld. Combinatorially implosive algorithms. *Comm. ACM 22*, 10 pp.734-738.

[Kowa 79]. R.A.Kowalski. Algorithm = logic + control. *Comm. ACM 22*, 7 pp.424-436.

[Ku & Ka 83] V.Kumar and L.N.Kanal. A general branch-and-bound formulation for understanding and synthesizing and/or search procedures. *Art. Int. 21*, pp.179-198.

[La & Sa 84]. T-H.Lai and S.Sahni. Anomalies in parallel branch-and-bound algorithms. *Comm. ACM 27*, 6 pp.594-602.

[La & Wo 66] E.Lawler and D.Wood. Branch-and-bound methods: a survey. *Oper. Res. 14*, pp.699-719.

[Li & Wa 84]. G-J.Lie and B.W.Wah. How to cope with anomalies in parallel approximate branch-and-bound search. *AAAI-84 conference*, pp.212-215.

[Ma & Ca 82]. T.A.Marsland and M.Campbell. Parallel search of strongly ordered game trees. *Computing Surveys 14*, 4 p.533-551.

[McB & S 87]. D.L.McBurney and M.R.Sleep. Transputer-Based experiments with the ZAPP architecture. In *Parallel Architectures and Languages Europe '87 (PARLE 87).* Published as Springer LNCS 258, pp.242-259.

[Nils 71]. N.J.Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw Hill.

[Ok & Ma 87] A.Okumura and Y.Matsumoto. Parallel Programming with Layered Streams. *IEEE Symp. on Logic Programming*, San Francisco, pp.224-233.

[Pear 84] J.Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

[RNC 96] K.Rokusawa, A.Nakase and T.Chikayama. Distributed memory implementation of KL1. *New. Gen. Comp. 14*, pp261-280.

[RSMB 88] V.J.Rayward-Smith, G.P.McKeown and F.W.Burton. The General Problem Solving Algorithm and its implementation. *New Generation Computing 6*, 1 pp.41-66.

[Shap 84] E.Shapiro. Systolic programming: a paradigm for parallel processing. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, pp.458-471.

[Shap 89] E.Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys 21*, 3 pp.413-510.

[Sh & Ta 83] E.Shapiro and A.Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing 1,* 1 pp.25-48.

[Sh & Wa 93] E.Shapiro and D.H.D.Warren (eds). The fifth generation project: persoanl perspectives. *Comm. ACM 36,* 3.

[Stoc 79] G.C.Stockman. A minimax algorithm better than alpha-beta? *Art. Int. 12,* pp.179-196.

[Tick 91] E.Tick *Parallel Logic Programming,* MIT Press.

[Ti & Ic 90] E.Tick and N.Ichiyoshi. Programming techniques for efficiently exploiting parallelism in logic programming languages. *2nd ACM Symp. on Princ. and Pract. of Parallel Programming. SIGPLAN Notices 25,* 3 pp.31-39.

[Turn 79] D.A.Turner. A new implementation technique for applicative languages. *Soft. Pract. Exp. 9,* pp.31-49.

[Ue & Ch 90] K.Ueda and T.Chikayama. Design of the kernel language for the parallel inference machine. *Computer Journal 33,* 6 pp.494-500.

[Wate 85] R.C.Waters. The Programmer's Apprentice: a session with KBEmacs. *IEEE Trans. Soft. Eng. SE-5,* 3 pp.237-247.

[Wils 87] W.G.Wilson. Concurrent alpha-beta, a study in concurrent logic programming. *IEEE Symp. on Logic Programming,* San Francisco, pp.360-367.

[Yard 90]. E.Yardeni, S.Kliger, and E.Shapiro. The Languages FCP(:) and FCP(:,?). *New Generation Computing 7,* pp.89-107.