Chapter 4

One person games & problem solving

We shall look at several examples of one-person games and puzzles and introduce several techniques for finding strategies or solutions. These are based on constructing a tree or graph of possible moves and finding efficient ways of searching these structures. The structures are referred to as the search space. In order to tackle large instances it usually is necessary to reduce the size of the search space. If this still gives the best solution then all is well. However sometimes it is better finding a reasonable solution in a short time rather than spend an exorbitant amount of time trying to find the optimum solution. Often heuristics are applied in order to cut down the branching or suggest the more likely branch to try first. These are often found empirically and it may not be possible to show that they are in fact always the best approach to take but hopefully in most cases they turn out to be satisfactory.

If this type of pruning is not done then we are often lead to what has been called the combinatorial explosion. For example consider the well known Rubik's cube or instant insanity (the game where you have to get the colours of four cubes all matching when placed on top of each other), the search spaces for these are enormous.

4.1 Some small examples

We shall begin our study by considering some simple examples which can be solved easily by constructing the tree of all possible moves. From the tree we can discover, whether it is impossible, whether there is a solution, and also obtain all solutions if this is the case.

We can also exhibit the moves made to reach the goal and in fact in many puzzles this is what we actually require as the solution i.e. how do you go from one state to another (the goal state).

4.1.1 Example 1: (3-puzzle)

This is a simplified version of the 15-puzzle but has a very small search space for us to comprehend. We start with an initial configuration and are to transform this into the goal

configuration. Note the possible theoretical configurations split into two groups and one can only transform between configurations in the same group (unless of course you remove the pieces from the board!).

See slide showing two possible solutions to the problem one being shorter than the other. We of course are disregarding solutions which involve cycles of moves. Clearly these can not find an essentially new solution and must be longer than necessary.

4.1.2 Example 2: (Cryptarithmetic)

Here we have an addition sum where letters denote distinct digits. In the example on the slide we assume we do not want solutions with D=0. The goal is to find the possible solutions to the problem. We are not so much interested in showing how to get there although we do wish to know that all solutions have been obtained.

4.1.3 Example 3: (The fox,hen & corn problem)

This is typical of many recreational puzzles. We have the following situation:

There is a boat rowed by one man which can only take one article across the river at a time. On one side of the river where the man is there are a fox,a hen and a bag of corn. The problem is that if the fox is left alone with the hen he will of course devour it and similarly if the hen is left unattended with the corn it will eat it up. How is the man to transport all three to the other side of the river without having any mishap.

We require the shortest solution. In fact there are two such as we see on the slide.

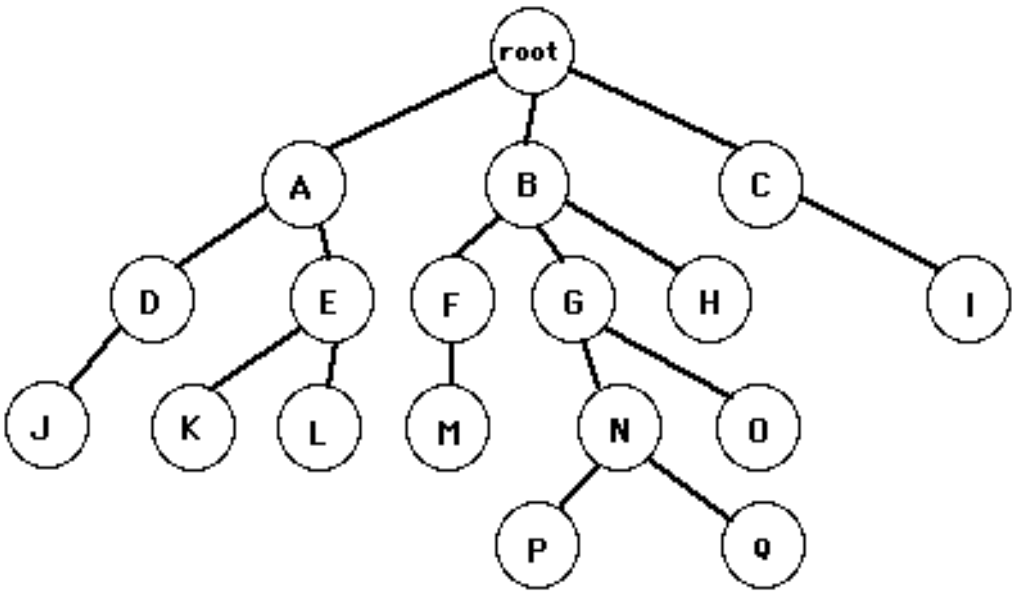
4.2 Tree searching algorithms

Once a tree or graph has been constructed for solving a problem, we require techniques for searching the structure. There are essentially three such algorithms.

1. Depth first: Go down the leftmost branch until a leaf is reached. If not a goal then back up branch until find a branching node and go down leftmost branch not yet visited.

2. Breadth first: Scan level 1 for goal first in a left to right manner. If no goal found then scan level 2 and continue in this manner.

3. Progressive deepening: This is a combination of [1] and [2]. A level and an increment is specified. A depth first search is performed but only up to a depth of level. If no goal is found then the depth of search is increased by increment. This is repeated until a goal is found. If level = 1 and increment = 1 then this reduces to (1).



Sample Tree

The order of visitation of the nodes in the above tree are:

1. depth first: `ADJEKLBFMGNPQOHCI`

2. breadth first: `ABCDEFGHIJKLMNOPQ`

3. progressive deepening: (level =2 inc = 1) `ADEBFGHCIJKLMNOPQ`
   (level = 2 inc = 2) `ADEBFGHCIJKLMNPQO`

4.3 Tree searching algorithms & Heuristics

We need some general algorithms to find the optimum path in a tree. So we assume that each arc in the tree has some cost associated with it. It is then required that we find a branch or path from the root to a goal state which has minimum cost if all the arc-costs are summed. If there is no explicit cost on the arc we take it as unit and then we are finding the solution branch which takes the fewest steps. In applications to games it is this cost which is usually taken.

We shall see that for many games we have some extra information or knowledge which can guide our search and so in effect reduce the size of the search space . This extra data is supplied by means of a heuristic function. This we be discussed in more detail later.

Here we give the basic algorithm which makes no use of heuristics.

4.3.1 Algorithm A^T (the tree algorithm)

```
{ let the cost of the edge between nodes n and m be c(n,m)
   set live  will hold the nodes to be considered next.
   g will hold the cost of the minimal path up to node n       }

      live <- {root};
      g(root) <- 0;
      while live not empty do
       [ n <- select({m in live | g(m)  minimal});
        live <- live -{n};
        if n is goal then [ output(n);halt]
        else [ live <- live + Suc(n);
              forall m in Suc(n) (g(m) <-g(n) + c(m,n))]
        ];
      output('no goal node')
```

g could of course be implemented by means of an array or list structure storing $graph(g)$.

The above is the most efficient algorithm if we have no further knowledge about the problem. If the cost is unit then it reduces to the breadth first search. We don't have to generate the tree before hand — and in fact in general would not do so — we only are required to generate the successors of the node which we are dealing with so only a small part of the full tree may be needed. Of course in some cases it might any way be impossible to construct the whole search tree.

4.3.2 Algorithm A^*

If we have a heuristic function - this is a function which can be thought of as an intelligent guess at the cost of the length of the shortest path to a goal from the node. Let h be the

heuristic function. Then A^* is just a modification of A^T where : we replace the function g by the function $f = g + h$. This is known as a *best-first search algorithm.*

This function $f(n)$ is then an estimate of the shortest path to a goal passing through node n. The heuristic function can be quite a crude guess at the distance to the goal and of course if it could be estimated accurately then we would already be able to compute the minimum path without the need of this algorithm!

There is a theoretical result which says when we can be certain that using the heuristic gives the real minimum path.

Theorem: *If $h(n) \leq H(n)$, where $H(n)$ is the real cost of the minimum path to a goal state, then the algorithm A^* does indeed determine the goal state with minimum cost path.*

The theorem is of little use in practice as determining if $h(n) \leq H(n)$ is difficult to do in general. However the reduction in search space yielded by using a heuristic is often so dramatic that it is worth the wile even if the optimum path is not actually found. Often the solution is just as good.

To give a heuristic function one just tries to give a weighting to a state so that the nodes which one feels intuitively should lead to a solution are given smaller values so that they are considered before the less likely. This can be somewhat ad hoc.

A heuristic function h which statisfies $h(n) \leq H(n)$ is called an *admissible* heuristic. We now present the proof of the above theorem that admissible heuristics yield optimum solutions.

So suppose we have

$$h(n) \leq H(n) \quad \forall n$$

where $H(n)$ is the shortest distance to a goal from n (if there is such a path). Let G be a goal found by using A^*. Then if G is not an optimal goal, then there must be a node n which has not been expanded so that

$$f(n) \geq f(G) = g(G)$$

since goal G satisfies $h(G) \leq H(G) = 0$ and there is an optimal path through n. Now $f(n) = g(n) + h(n)$ and $h(n) \leq H(n)$ and so

$$f(n) \leq g(n) + H(n) = H(root) = M$$

Assuming G is not optimal we have $f(G) = g(G) > M$ and so

$$M < g(G) = f(G) \leq f(n) \leq M$$

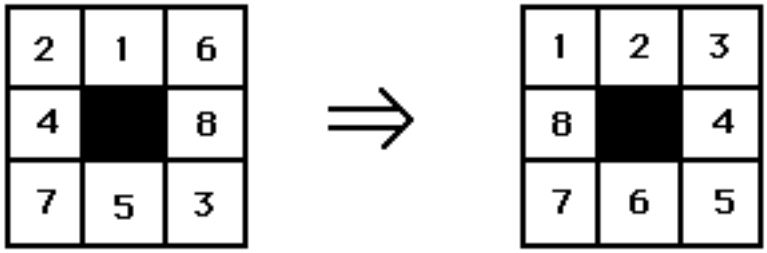
a contradiction, so G must be optimal. $\square$

4.3.3 Example

In the 8-puzzle one possible heuristic might be h(n) = number of misplaced numbers. We apply this to

Example 1

4.3.4 Example



Example 2

The heuristic function used this time is

$$h(n) = P(n) + 3 * S(n)$$

where $P(n)$ = sum of distance each tile is from correct position and $S(n)$ = sum over all non-central tiles where a 2 given for tile not followed by successor and 1 if tile in centre. This reflects the difficulty of swapping pieces around. See handout for complete solution using this heuristic.

4.3.5 Exercise

Use the heuristic function $h(n)$ = sum of distances from correct position to find solution to

Example 3

4.4 Generate & Filter

Suppose we are required to find a particular goal state of a problem. In many cases we can generate a possible solution and then test if this is a goal. Of course this is what AT does in a particular order. Now this approach is very inefficient in general - especially if the generation is just some standard enumeration of states and does not take into account the nature of the puzzle. One improvement is to generate possible states and then pass them through a filter before we try to see if it is a goal. In this way very many fewer states are tested. We expect testing for a goal takes very much longer than applying the filter.

Examples where this can be successful is when the test for goal is actually supplied by another player. e.g. When one player thinks of some entity and the other player tries to discover its identity by a sequence of questions. This is essentially a one player game as the one to supply the entity only has the role of answering the questions. To play a decent game one does not want to ask too many questions and this is the purpose of the filter.

The use of the filter is in a sense a heuristic - helping cut down the large search space.

4.4.1 Example

The game of *bulls and cows* is a simpler version of mastermind. A hidden 4-digit number is supplied and the player is asked to guess what the number is. He is told after each guess how many digits are in the correct position — the number of bulls and how many are the correct digit but in the wrong position — the number of cows. It is usual for the digits to be all distinct.

Here is a fairly decent strategy for a machine to play:

- *Generate* the sequence of 4-digit numbers in any uniform way.
- *Filter* these numbers as possible solutions by seeing that if it were the correct code it would give the same answers as we have already received. Once a few replies have been obtained this filters out very many impossible tries.

Eventually the correct code will be discovered and in practice takes usually less than 6 moves to find it. It would be fairly difficult for a human to use this strategy as the amount of book-keeping as quite large. Many improvements can be made to this strategy. e.g. If we eventually find that the sum of the bulls and cows is 4 then we need only generate 4-digit numbers from this set. — so we improve the generation process. If we have made 2 choices which are 3 bulls and they agree on three digits then we need only consider the digit in the remaining place. If the don't agree on three places then we know the answer is one of 2 possibilities. Many more refinements like this can be made.

4.4.2 Abstract View of Generate and Filter Technique

This applies to games of the following form:

- *Player II is really a passive player and is only there to supply answers and choose the initial goal — it is thus really a one-player game.*
- *Player II chooses a goal state*
- *Player I then tries to find the goal by making informed guesses. Use is made of the information received from player II for all previous moves. Play ceases when Player I has discovered the goal state.*
- *We assume that player II always gives correct information on any move made!*

4.4.3 Examples

Mastermind, Moo, One-way Battleships.

```
Prolog Skeleton
play :-
  generate_move(M),
  filter(M),
  get_response(R),
  (R == goal
   -> write('congratulations');
      (assert(response(M-R)),
      fail) ).
% see that M gives same responses as on
% previous guesses if it were the correct guess

filter(M) :-
  forall(response(S-R),val(M,S,R)).
```

In the above: `generate_move` — generates all possible moves `val(M,S,R)` calculates the response for state `S` if `M` were the correct guess. We could alternatively have stored the moves in an accumulator list and then have no need to assert the replies into the database. We would also have the moves and replies stored in a list.

It would be essentially:-

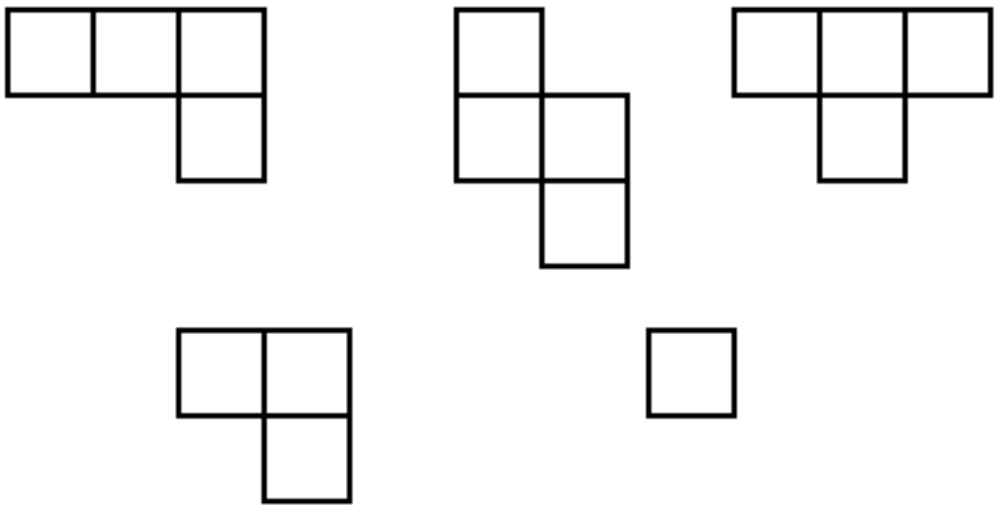
```
play([F-FR|T],X) :-
  generate_move(F,M),
  filter(M,[F-FR|T]),
  get_rep(R),
  (R == goal
   -> X = [M-R,F-FR|T];
      play([M-R,F-FR|T],X)).

play(Moves) :- play([start],Moves).
```

Moo played with 4 digits can be solved on average in about 6 moves. We can sometimes improve the generate predicate as we often know that a move cannot contain a particular piece so shouldn't appear in guess — e.g. when have 4 c's know code consists of these digits only.

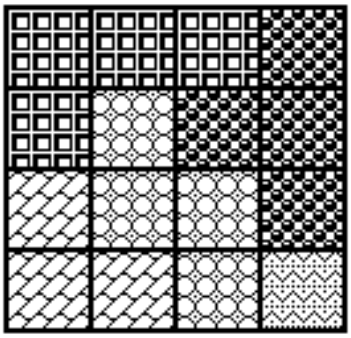
4.4.4 Exercises

1] Find a solution to the missionaries and cannibals problem. The initial state is $< MMMCCCB, \_ >$. The boat can carry at most two people. If the cannibals outnumber the missionaries at either bank then they will devour the missionaries.
2] Cover a 4x4 square with the following five shapes:

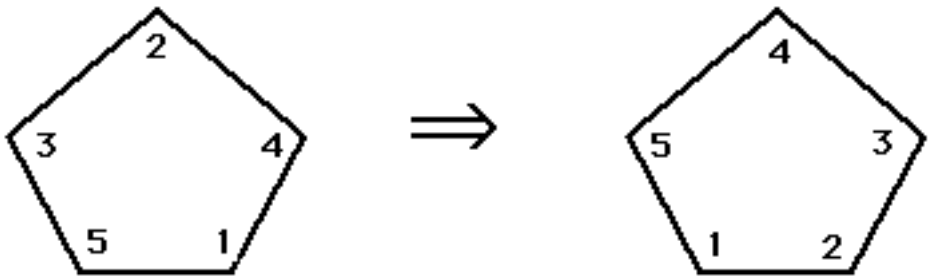


You should start by considering the most irregular shape first. Also take into account the symmetry of the square.

Possible Solution



3] Solve the following puzzle using A^* . Transform the first numbered pentagon into the second by a series of allowed moves.



The allowed moves consist of swapping non-adjacent numbers. e.g. we can swap 5 and 4 but we can not swap 5 and 1 or 5 and 3 as these are connected by an edge.

Use the following heuristic function: v(n) is first defined on numbers given a particular state.

$$v(n) \simeq \begin{cases} 0 & \text{if n is in its final position} \\ 1 & \text{if n opposite final position} \\ 2 & \text{if n is adjacent to its correct position} \end{cases}$$

$$h(state) = \Sigma_{n=1}^{5} v(n)$$

4.5 Two person Games

In order to analyse two person games we need to construct an *and/or* tree. The typical type of games we shall consider are between two players P1 and P2. They alternate moves and continue until one of them reaches a winning state. Often this is when the other player can't make a move.

In order to find a winning strategy for P1 we begin by constructing the tree with an *or* node. Below this are the possible moves P1 can make. These are all then marked as and nodes and we expand below each node with the possible moves P2 can make and continue in this way.

A winning strategy for P1 is then a subtree starting at the root such that every leaf is a winning state for P1 and at every *or* node there is one branch emanating and at every *and* node all edges in the original tree must also be in this subtree. Thus P1 begins by making the indicated move from root and then whatever move P2 makes P1 will take the branch indicated in the subtree. Clearly P1 can just follow this mode of play and must eventually reach a leaf which is a winning state for him.

A tree for finding a strategy for P2 is constructed in a dual manner. We begin with the root being an *and* node and then nodes when P2 moves will be *or* nodes all the others for P1's moves are *and*. A winning strategy for P2 is then represented by a subtree whose leaves are winning states for P2 and as before all *and* nodes have every edge emanating from them, *or* nodes just one edge.

4.5.1 Nim

This is typical of simple two person games. It also has the advantage that it has been thoroughly investigated and strategies for players are well known.

It consists of several piles of coins. A valid move is to take any number of coins from one pile — it must be a positive number. A winning position is when that player takes the last coin. See following figure for an and/or tree giving a winning strategy for P1.

4.5.2 Mathematical Treatment

Define the nim-sum $+_2$ of integers as follows :

Expand the two numbers in binary, then perform addition in bit-wise manner without a carry. Put another way it is bitwise exclusive or.

$$5 +_2 7 = 101b +_2 111b = 10b = 2$$

If the nim-sum of the numbers in the piles is positive then that player has a winning strategy. What he must do is reduce the nim-sum of new piles to zero. But clearly we have for any number x, $x +_2 x = 0$, so just add the nim-sum to each pile until we get one which is reduced (always will) and then reduce this pile to that sum, we then have the resulting nim-sum is 0.

$$x_1 +_2 x_2 +_2 x_3 +_2 x_4 .... +_2 x_n = s$$

and so

$$x_1 +_2 x_2 +_2 (x_3 +_2 s) +_2 x_4 \ldots +_2 x_n = s +_2 s$$

hence the new sum is 0 meaning the second player loses.

Proof of claims:

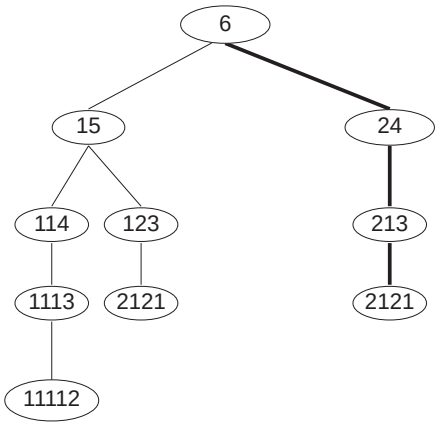
i) Adding s to piles must reduce one of them. For example look at the leftmost bit of s ($s > 0$). One x_i must have this bit set to 1 otherwise s would have this bit as 0. Now adding s to x_i to give x_i' would make this bit 0 and so reduce x_i.

ii) When the nim sum $s = 0$, then reducing any x_i to x_i' will make new nim sum $s' > 0$. For taking from pile size x_i must alter at least one 1 in its bit pattern. Hence this column sum must change from 0 to 1 and hence $s' > 0$.

4.6 Grundy's Game

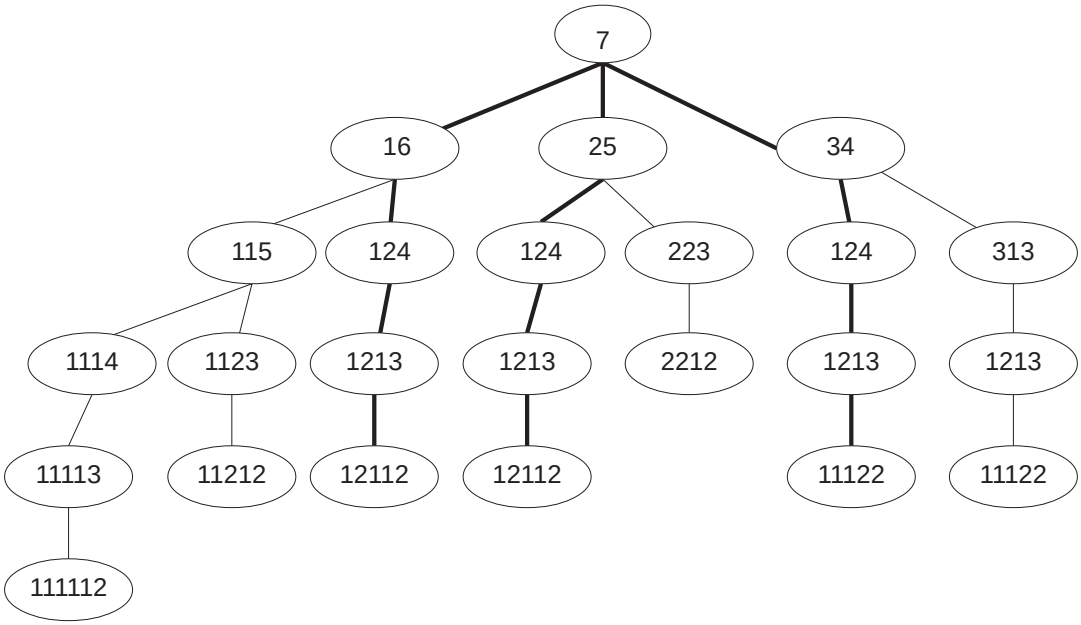
Rules: Play starts with one pile of counters. A valid move is to split one pile only into two new unequal sized piles. Play continues until no more piles to split (and so only 1s and 2s remain).

4.6.1 Example



Winning Strategy for I

4.6.2 Example



Winning Strategy for II

There are many variations of both NIM and Grundy's game. For example, misère games have the goal of the *opponent* taking the last piece.

4.7 Mini-Max Method

To deal with realistic games such as Chess or Backgammon, we have to use heuristics to reduce the search space. A heuristic function is a mapping h assigning values to positions of the game (node in the and/or tree). The more positive the value of h the better the position for player I and the more negative the better for player II. These functions are often obtained empirically by observation of expert players and analysis of many games. The mini-max method then uses h to choose the next move as follows:

- Construct the tree up to a certain depth (n move look-ahead, or n-ply)
- Calculate the h values at the leaves of this subtree.
- Work out the backed up value h by

$$h(s) = max\{h(s')|s' \text{ is immediate successor of } s\}$$

for *or* node and

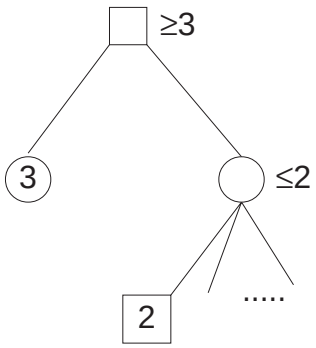
$$h(s) = min\{h(s')|s' \text{ is immediate successor of } s\}$$

for *and* node.

This is done all the way up the tree until root is reached. This is an *or* node for player I so the node s with maximum $h(s)$ value is chosen. Example For 0's and X's we can take $h(s)$ = number of lines which I could fill completely — number of lines II could fill.

4.8 Alpha-beta Pruning

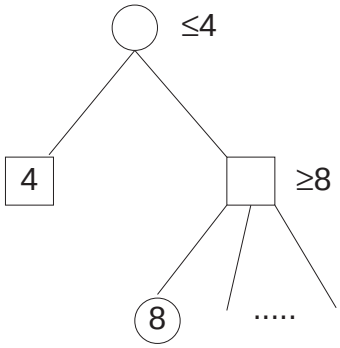
In calculating the backed up value h, it is often possible to avoid some computation by using $\alpha - \beta$ pruning. The following is an instance of α pruning for the values to right of 2 can only decrease the min node's value.



α-cut off

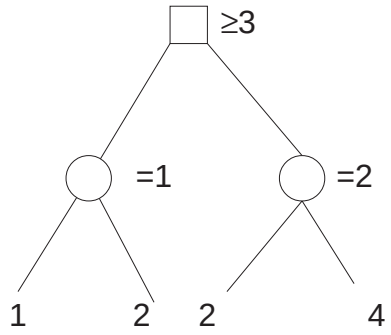
Don't need to evaluate these branches — can only *decrease* value of 2

The corresponding technique for min nodes:



β-cut off

These paths can only increase the value of 8
It does not always help as can be seen from the next example where all leaves need to be evaluated.



No saving

All branches have to be investigated — no saving in evaluation.
Remark: If the branching factor of the tree is b, then $\alpha - \beta$ pruning on average reduces this to $\sim \sqrt{(b)}$. So the effective search tree for chess could be reduced from a tree with branching factor ~ 30 to ~ 6. This would then make it possible to increase the depth of the search.

4.9 Prolog Code for Mini-max

Here is a program which will evaluate the backed-up values using a mini-max evaluation.

```
% mini max function We assume root is a max node.
% L is the level of look-ahead

min_max(L,Nd,Val) :- maxx(L,Nd,Val).
```

```
% for max node
maxx(_,Nd,Val) :-
      leaf(Nd),!,
      h(Nd,Val).

maxx(0,Nd,Val) :-
      h(Nd,Val),!.

maxx(L,Nd,Val) :-
      L1 is L-1,% now form set of all solns to goal
      findall(V1,(succ(Nd,Nd1),minn(L1,Nd1,V1)),Vals),
      max_lst(Vals,Val).

% for min node
minn(_,Nd,Val) :-
      leaf(Nd),!,       % no successors
      h(Nd,Val).

minn(0,Nd,Val) :-
      h(Nd,Val),!.

minn(L,Nd,Val) :-
      L1 is L-1,
      findall(V1,(succ(Nd,Nd1),maxx(L1,Nd1,V1)),Vals),
      min_lst(Vals,Val).

% find max of a list
max_lst([X],X) :- !.
max_lst([X|Xs],Mx) :-
     max_lst(Xs,Mxs),
     maximum(Mxs,X,Mx).

% find min of a list
min_lst([X],X) :- !.
min_lst([X|Xs],Mx) :-
     min_lst(Xs,Mxs),
     minimum(Mxs,X,Mx).

maximum(X,Y,X) :- X >= Y,!.
maximum(X,Y,Y) :- Y > X.

minimum(X,Y,X) :- X =< Y,!.
minimum(X,Y,Y) :- Y < X.
```

4.10 Logic Puzzles

Here we want to consider a backtracking (or another variant of generate and test) to solve the kind of problems often set as *brain teasers*. In particular a *logic puzzle* consists of:

- some facts concerning a number of objects;
- they have various attributes;
- a minimum number of facts is given about the objects and their attributes to produce a unique solution.

Here is a typical example:
Three students came first, second and third in an AI exam. All have different names; like different types of music and own different cars.

1. *Kevin likes rave and did better than the Porsche owner;*
2. *David, the Skoda owner, did better than the student who likes folk music;*
3. *the student who likes jazz came first.*

The questions to be answered are:

a *Who owns the Ford?*

b *What type of music does Andy like?*

This type of problem is solved by essentially a generate and test technique. Notice even the form of the question determines some information not known from the facts. (e.g. in the above, we find the third type of car is a Ford and the other students name is Andy).

The first part of the puzzle tells us the sort of objects we are looking for in the solution. So we set up a predicate `structure` which gives the form/data structure for the set of objects to be found.

In the above example:

```
structure([stdnt(Nm1,Ps1,M1,Cr1),
           stdnt(Nm2,Ps2,M2,Cr2),
           stdnt(Nm3,Ps3,M3,Cr3) ] ).
```

i.e. the *structure* of the solution is a list of three terms (`stdnt` where the four parameters represent the `name, position, music and car`. The aim of solving the puzzle is then to fully instantiate all these variables from the facts and queries.

In order to represent the facts easily we need some predicates to extract certain components from the terms (which are really records). In the case we are considering:

```
fname(stdnt(Nm,_,_,_),Nm).      % pick out name
position(stdnt(_,Pos,_,_),Pos). % pick out position
music(stdnt(_,_,Mus,_),Mus).    % pick out music
car(stdnt(_,_,_,Car),Car).      % pick out car
```

We need ordering of `first, sceond, third`

```
better(first,second).
better(first,third).
better(second,third).
```

(or of course we could use numbers here instead and the ordinary $<$ relation.)

The information regarding the clues is contained in assertion `clues` and queries in assertion `queries`.

We set up the puzzle with `our_puzzle` and then solve using `solve_puzzle` and `solve` which just goes down a list of goals and tries to satisfy each one in turn.

Here is the complete program:

```
% Solve the Logic puzzle

test:- our_puzzle(P), % set up and solve
solve_puzzle(P,S),
write(S),nl.

solve_puzzle(puzzle(Cls,Qrs,Soln),Soln) :-
         solve(Cls), % solve all clues
          solve(Qrs). % solve the queries

solve([]):-!.
solve([Cl|Cls]) :-
         Cl,solve(Cls).

fname(stdnt(Nm,_,_,_),Nm).
position(stdnt(_,P,_,_),P).
music(stdnt(_,_,M,_),M).
car(stdnt(_,_,_,C),C).

better(first,second).
better(first,third).
better(second,third).

% get the clues and queries from database
our_puzzle(puzzle(Cls,Qs,Soln)) :-
        structure(Struct),
```

```
        clues(Struct,Cls),
        queries(Struct,Qs,Soln).

% Here is the structure
structure([stdnt(Nm1,Ps1,G1,Cr1),
                stdnt(Nm2,Ps2,G2,Cr2),
                stdnt(Nm3,Ps3,G3,Cr3)]) .

%three objketc distinct
distinct(A,B,C) :- A\==B,A\==C,B\==C.

% three object distinct
dist_structure([stdnt(Nm1,Ps1,M1,Cr1),
                stdnt(Nm2,Ps2,M2,Cr2),
                stdnt(Nm3,Ps3,M3,Cr3)])  :-
                     distinct(Nm1,Nm2,Nm3),
                     distinct(Ps1,Ps2,Ps3),
                     distinct(M1,M2,M3),
                     distinct(Cr1,Cr2,Cr3).

% HERE ARE THE CLUES
clues(Stdnts,[(did_better(S1Cl1,S2Cl1,Stdnts),fname(S1Cl1,kevin),
                       music(S1Cl1,rave),car(S2Cl1,porsche)),
                      (did_better(S1Cl2,S2Cl2,Stdnts),fname(S1Cl2,david),
                        car(S1Cl2,skoda),girl_typ(S2Cl2,folk)),
                      (member(S1Cl3,Stdnts),position(S1Cl3,first),
                           music(S1Cl3,jazz)) ]).

queries(Stdnts, [(member(Q1,Stdnts),fname(Q1,Name),car(Q1,ford)),
                     (member(Q2,Stdnts),fname(Q2,Andy),music(Q2,M)),
                     (member(Q3,Stdnts),
                     fname(Q3,Nm3),position(Q3,second)),
                      dist_structure(Stdnts)],
                       [ ['the ford owner is ',Name],['Andy likes ',M],
                       [Nm3,'came second']]).

did_better(S1,S2,Stdnts) :-
         member(S1,Stdnts),member(S2,Stdnts),
         position(S1,Pos1),position(S2,Pos2),
         better(Pos1,Pos2).
```

4.10.1 The Zebra Puzzle

This is a well known example from recreational puzzle books.

There are five houses, each of a different colour and inhabited by a man of a different nationality, with a different pet, drink and brand of cigarettes.

The following information is given:

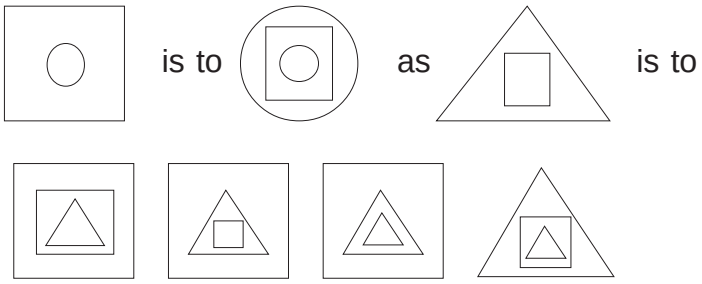
1. The Englishman lives in the red house.
2. The Spaniard owns a dog.
3. Coffee is drunk in the green house.
4. The Ukranian drinks tea.
5. The green house is immediately to the right (your right) of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The man who smokes Chesterfields lives in the house next to the man with the fox.
11. Kools are smoked in the house next to where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The japanese smokes Parliaments.
14. the Norwegian lives next to the blue house.

Who owns the zebra? Who drinks water.

4.10.2 Analogical Reasoning

Here we present a simple program which can solve problems by using geometric analogy. The type of problems to be tackled are typical of IQ tests.

4.10.3 Example



Geometric Analogy

The problem is to find the shape which corresponds to the third figure. This can be solved abstractly by:

1. Find a rule which relates `A` to `B`.
2. Apply the rule to `C` to obtain `Z`.
3. Find answer nearest to `Z` in the possible alternatives.

So to write a program to solve such puzzles, we have to incorporate a number of common rules. In the above example, the rule would be *copy inside shape to outside* (with some scaling).

```
% filename: analogy.pl
% geometric analogy program

analogy(A=B,C=X,Answers) :-
   match(A,B,Rule),
   match(C,X,Rule),
   member(X,Answers).

match(inside(Fig1,Fig2),inside(Fig2,Fig1),invert).
match(above(Fig1,Fig2),above(Fig2,Fig1),swap).
match(inside(Fig1,Fig2),inside(Fig1,Fig2,Fig1),copy_outside).
```

```
% representing tests figures(Name,A,B,C) - A is to B as C is to ?

figures(test1,inside(circ,sq),inside(circ,sq,circ),
             inside(sq,tri)).
answers(test1,[inside(tri,sq,sq),inside(sq,tri,sq),
             inside(tri,tri,sq),inside(tri,sq,tri)]).

figures(test2,inside(sq,tri),inside(tri,sq),inside(circ,sq)).
answers(test2,[inside(circ,tri),inside(sq,circ),inside(tri,sq)]).


test_analogy(Name,X) :-
    figures(Name,A,B,C),
    answers(Name,Ans),
    analogy(A=B,C=X,Ans).
```

`analogy` is called as `analogy(A=B,C=X,Answers)` — `A` is to `B` as `C` is to `X`. The initial geometric shapes are given by the assertion `figures` — so for the above given by by assertion with first argument test1. The possible solutions are given by assertions `answers`. Finally `match` encodes the rules. So the rule for the above will be *copy_outside*.

We see that `analogy` formalises the informal description given above.

The drawback with this simple program, is that very many rules would have to be incorporated in it to deal with the variety of such problems. There is no way to generalise the rules. Probably a good approach would be to incorporate some meta-rules — rules about rules. Or perhaps have a system which could learn the rules from being presented many examples. We are then leading onto the topic of *Machine Learning*.