\relax

AI2 CLASS NOTES: PLANNING

BY MIKE USHCOLD

These notes are very sketchy, consisting primarily of transcriptions of the actual slides which I used in the lectures. Where these are particularly cryptic, I include additional explanation and references which treat the subjects usually from a somewhat different point of view.

The main references which treat planning are these:

- Charniak and McDermott;
  Means Ends Analysis: pp 300-306
  Planning in General, including STRIPS: Ch 9 485-527

- Nillson 80; Principles of AI;
  Chapters 7&8 pp 275-360

- Cohen & Feigenbaum; The Handbook of AI, Volume 3;
  Chapter XV 515-562

- Rich; Artificial Intelligence;
  Chapter 8 pp 247-277

SLIDE 1

REVIEW

Representation - Logic

Solving Problems
* state space
* problem reduction
* search DFS : BFS : A, A*

GAMES

PLANNING

==========

SLIDE 2

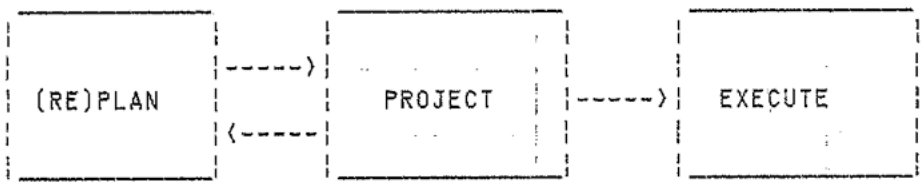
PLANNING vs EXECUTION

* PUZZLES & GAMES

  No real difference

* "REAL WORLD"

  Plans often fail



PROJECTION: Predicting What A System Will Do.

[See C&D 485-489 for further clarification of this diagram]

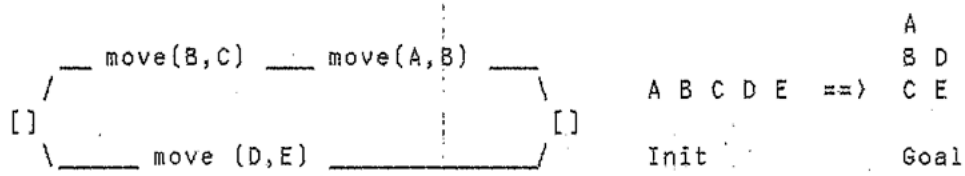
SLIDE 4

PLAN REPRESENTATION

Serial: Linear sequence of actions (operators)
[State space representations have serial plans]

Parallel: Some actions may co - occur

Partially ordered: Some order constraint



It doesn't matter in which order I make the two stacks. But, the manner in which I make the ABC stack does matter. We say this is partially ordered. Serial plans are fully ordered.

PLAN CONSTRUCTION

- SEARCH : State space, other methods
- PROJECTION : Interacting subgoals

When plans are being constructed, we must have a way to test whether they will work. One thing to test is for problematic interactions of suboals. The achieving of one goal may cause another to be undone. We will see examples of this later.

PLAN EXECUTION : Failure & Recovery

This phase of planning is largely unexplored, although people are now starting to tackle these problems in earnest.

==========

SLIDE 5

STRIPS

This was the first really significant planning project. [The best thing to read on Strips is to be found in the AI1 Problem Solving Notes which can be found in the library. Since it is so good, I neglect to elaborate on these slides.]

KEY POINTS

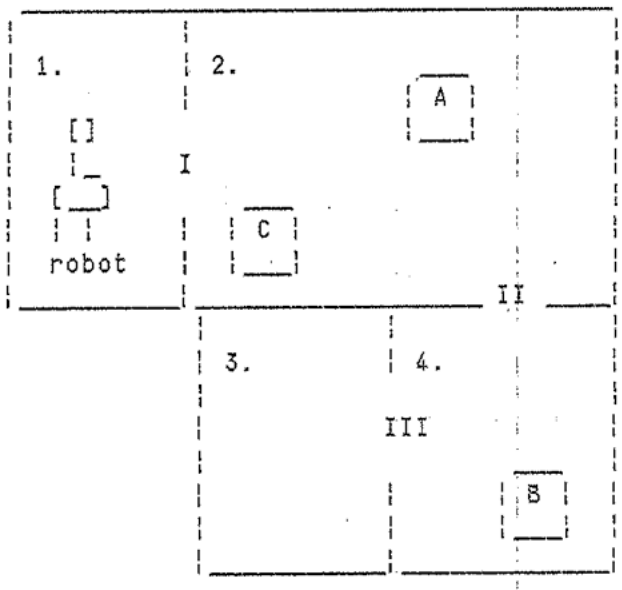
* State space search
* Means - End Analysis: A new search heuristic
* Operators
* Rich plan representation
* Controlled a real robot ("Shakey")
* Lasting impact: Many concepts originating with this project have and still are influencing much planning work. In particular, the notion of a STRIPS operator is fundamentally important.

==========

SLIDE 6

SHAKEY(the robot'S WORLD

)



connects(room1,room2,door I)
inroom(boxa,room2)
inroom(robot,room1)

connects(Ra,Rb,Da) is identical to connects(Rb,Ra,Da)

TYPICAL TASKS

* Go to room4

* Push Box B to room 1

* Block doorway II

ACTIONS

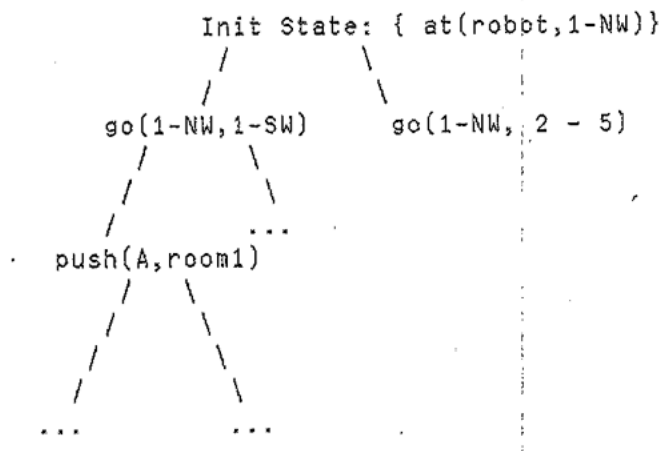
* go(X,Y)

* push(B,X,Y)

==========

SLIDE 7

STATE SPACE SEARCH



Problem -- Explosive search space, No obvious evaluation function.
Solution -- New search strategy: Means ends analysis

==========

SLIDE 7.5

MEANS - ENDS ANALYSIS

Means : operators
(actions)

Ends : problems, goals

Analysis : compare current state with goal state to determine a set of Differences.

Find the MEANS to reduce these differences

SLIDE 8

STRIPS OPERATORS

operator : name(variables)
(schema)

preconditions : What must hold for the operator to be applicable

delete list: what no longer holds after operator is applied

add list: what becomes true after operator is applied

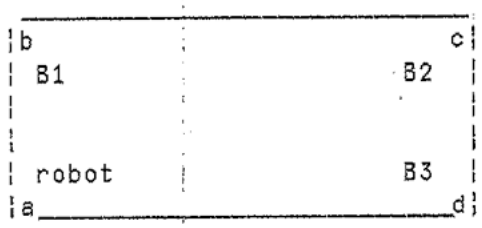
effects

SLIDE 9

EXAMPLES

operator	preconditions	delete	add
go(X,Y)	at(robot,X)	at(robot,X)	at(robot,Y)
push(B,X,Y)	at(robot,X) at(B,X)	at(robot,X) at(B,X)	at(robot,Y) at(B,Y)

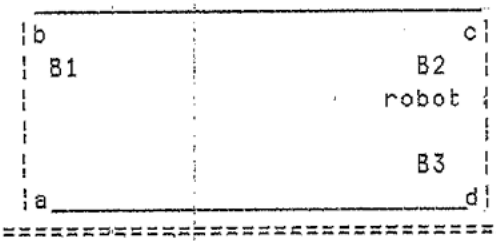
Initial state



at(robot,a)
at(B1,b)
at(B2,c)
at(B3,d)

apply operator : go(a,c)

Next state



at(robot,c)
at(B1,b)
at(B2,c)
at(B3,d)

SLIDE 10

MEANS END ANALYSIS: Control Strategy

0. Goal Stack (--- Go [Main Goal]

1. Select Subgoal From Stack

   IF Subgoal Holds In Current State

      Then IF Goal is main Goal STOP

           Else Apply operator whose
                preconditions was
                just satisfied,
                GOTO 1.

      ELSE Note differences ; GOTO 2.

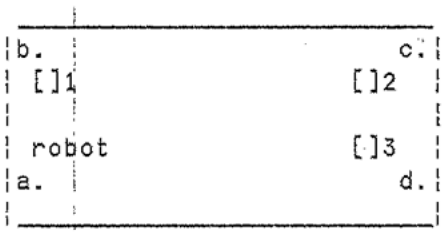
2. Select An operator which will reduce the difference.

3. Place operator preconditions on Goal Stack ; GOTO 1

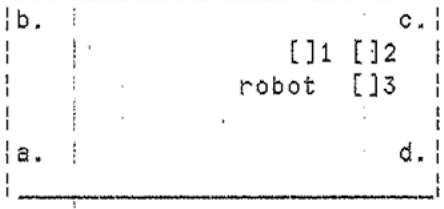
SLIDE 11

EXAMPLE

INITIAL STATE:
at(robot,a)
at(Box1,b)
at(Box2,c)
at(Box3,d)

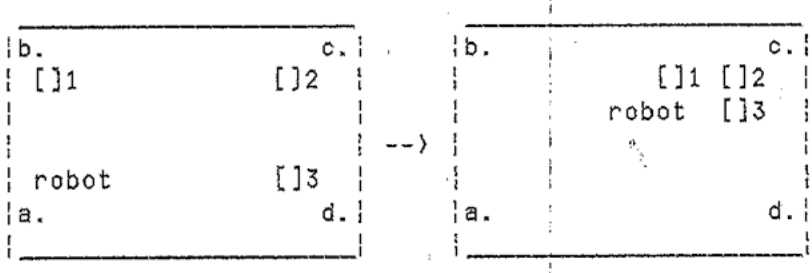


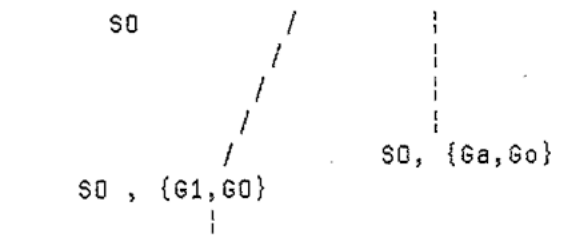
GOAL STATE :
at(robot,c)
at(Box1,c)
at(Box2,c)
at(Box3,c)

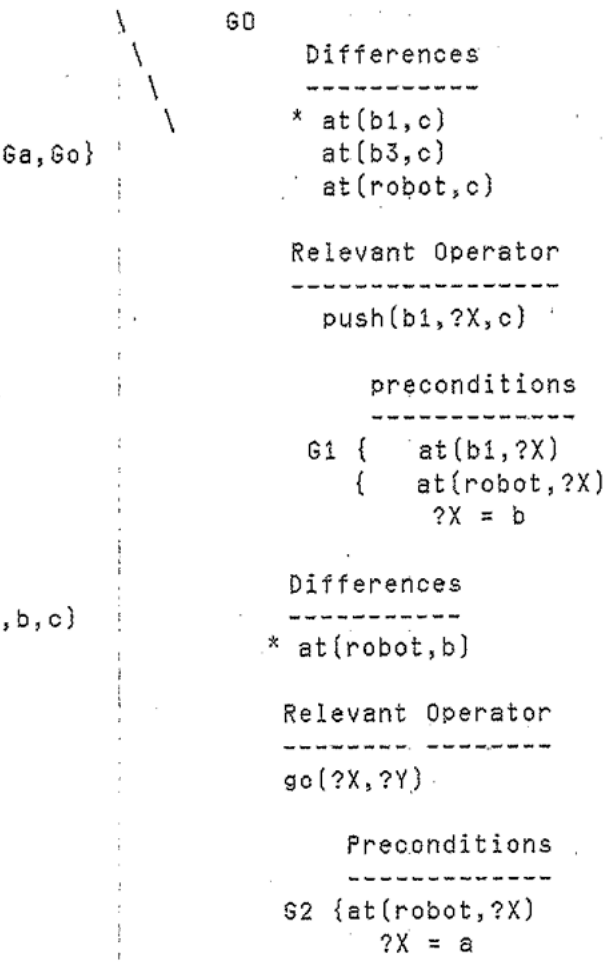


OPERATORS :
 go(X,Y) : Go from X to Y
 push(B,X,Y): Push box "B" from X to Y

SLIDE 12







SLIDE 13

MEANS ENDS ANALYSIS

- Forward State Space search (with backtracking) [You find out what operator to apply by trying to make something in the goal state true

(ie by reducing a difference). The operator which can directly be used to reduce the differenc may not be applicable, so you set up the preconditions of the operator which you would like to apply as new differences (subgoals) to be reduced. This is essentially backward chaining. BUT, when you finally find an operator which can be applied in the current state, you apply it forwards from the current (perhaps initial) state. You then start over and look for more differences to reduce. This is different from starting with the goal state and working backwards till you get to the initial state. The difference is that using means-ends analysis, you are looking to reduce a single difference and back chain till you find an operator which is applicable in the current state. Backwards search means apply the operators in reverse from a complete goal "state" wait till the the initial state is reached. At any time, MEA considers only one of many possible conditions which must hold in the goal state (a difference). IN both cases, if you reach a dead end which is not the state you are trying to reach and no more operators apply, you backtrack to the last state where there was a choice of which operator to apply and continue searching]

- Goal directed Heuristic [NB: It is NOT a heuristic evaluation function] Can plan ahead to use an operator before its actually applicable.

- Other Heuristics

  * Which Difference to reduce?
    Do hardest things first

  * [The intuition: You eventually have to reduce all the differences. You could waste a lot of time reducing all the easy ones (which by definition are more likely to be reducible, only to have to backtrack when it turns out to be impossible to reduce the hard one. So, it is better to try to reduce the harder ones first so that if it fails less time is spent chasing down an unsuccessful solution path].

  * Which operator to apply?
    - The one which reduces most differences. -or-
    - The "easiest one"

%EXERCISE 1: How do these heuristics relate to best first search using a %heuristic evaluation function? What exactly are the functions?

%ANSWER 1 This is not that easy, but have fun thrashing around with the %issues.

This helps to:

- Find better plans [eg. Selecting the op which reduces the most differences is likely to result in shorter plans. If there's more than one way to achieve something, and you pick the easier way, then your plan will involve less work overall.]

- Find plans quickly [Doing hardest things first often results in less backtracking]

==========

SLIDE 14

PRACTICAL ISSUES

World Model : Database of Predicates calculus assertions.

[Eg. at(robot,c), at(box2,d) etc.]

Test Subgoals : Automatic Theorem Prover "QA3"

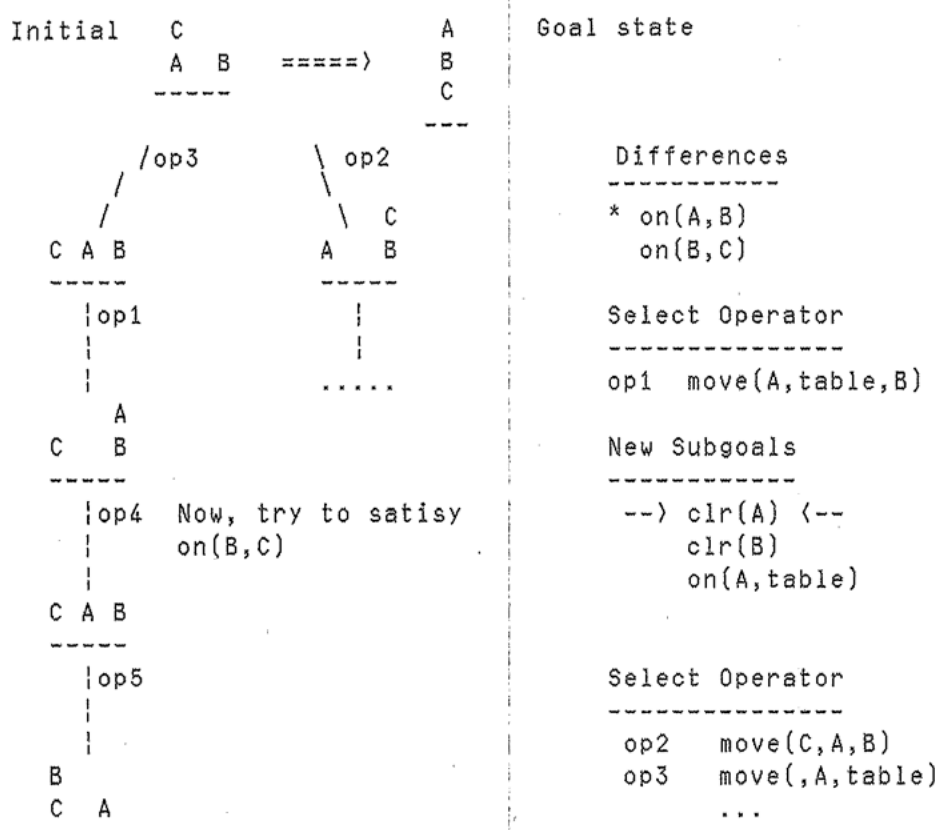
Finding Differences : Side effect of QA3.
What is needed to complete
or continue proof.

Computation Time : Tens of Minutes

==========

SLIDE 1

PROBLEMS with MEANS ENDS ANALYSIS



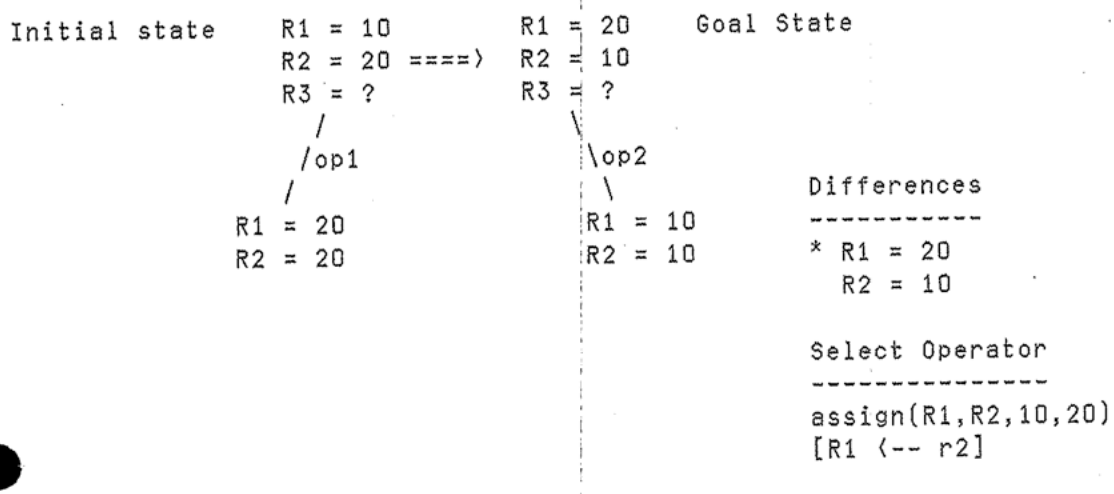
NB : This is a condensed search space. I only include world states. The actual space which is searched has several nodes which correspond to the same world state, but differ in the goal stack. Ie, from the initial world state, there would be a different node for each different subgoal The different subgoals correspond to different choices of which differences to reduce.

This is producing an obviously silly plan, which nevertheless does work.

==========

SLIDE 2

REGISTER SWAPPING



In this case, MEA cannot pruduce a working plan. By blindly attempting to reduce differences and choosing operators which will reduce them, it has ruined all possibilities of being able to solve the problem.

==========

SLIDE 3

BUT .....

* Simple depth first search works fine!

* MEA Works best when operators do not reintroduce differences already reduced.

* Interferance Among operators.
  One operator's delete list is part of another operators precondition or add list.

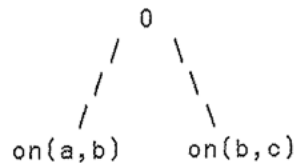
* MEA is "too eager", doesn't plan far enough ahead.

  * Finds suboptimal solutions.
  * Misses solution entirely! (shrunken search space)

==========

SLIDE 4

Interacting Subgoals



"Linearity Assumption": Subgoals are independant, and thus can be sequentially achieved in some order.

Embodied In MEA: To reduce a set of differences {A,B,...}, remove them one by one. MEA makes this assumption, and that's why it can get into trouble as we have seen.

==========

SLIDE 5

Approaches To Coping With Interacting Subgoals

Protection Violation

Once a goal is achieved, it should be protected. We say that there is a protection violation if such a goal is undone by a later action. Planners must both detect and correct such violations. Detection involves some form of bookkeeping. We will consider two approaches to correcting protection violations.

* Try Goal Reordering. [Only 10 goals gives 3 million possibilities]
  Even for small problems, trying all orderings is infeasible.

* Try Interleaving Subgoals.

```
   c
   a b     Begin work on     on(a,b)   clr(a)
   ---

   c a b   Next : achieve :  on(b,c)
   -----

   b
   c  a    Finish work on: on(a,b)
   ----
```

==========

SLIDE 6

SubGoal Promotion

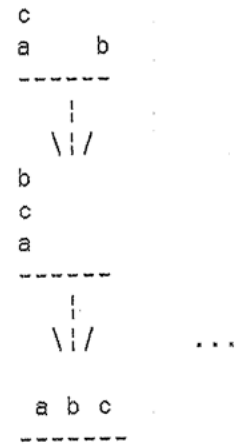
* Use MEA to start.
* Protect achieved subgoals.
* Detect Violations - Note offending subgoal
* Promote subgoal to beginning of goal stack
  (If reordering fails)
* Start over again

Problem : Throws away too much work already done

SLIDE 7

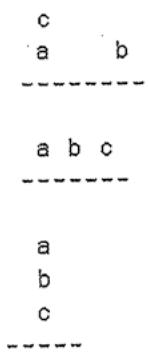
EXAMPLE : Subgoal Promotion

Goals	Opertaors
on(b,c)	move(b,table,c)
on(a,b)	



Violation! The offending subgoal which caused on(b,c) to be undone is "clr(a)" which is a precondition for moving "a" on to "b". We now promote this subgoal and start all over:

clr(a)	move(c,a,b)
on(b,c)	move(b,table,c)
on(a,b)	move(a,table,b)



This method works fine for some examples. However, if much work was done in solving the problem before the first violation was detected, then it is all for nought. So this method is too inefficient in general.

==========

SLIDE 8

GOAL REGRESSION

Move the offending subgoal back through the plan built so far to find an acceptable place for it.

We regress a goal through an operator by answering the question :
What must be true before the operator is performed to ensure that the GOAL is true after the operator is applied?

If we can answer this, then we can ensure that achieved goals are not later violated.

==========

SLIDE 9

EXAMPLES: Goal Regression.

GOAL	OPERATOR	REGRESSION
on(a,b)	move(c,a,table)	on(a,b)
on(a,b)	move(a,c,b)	True
on(a,b)	move(c,b,a)	False
have($100)	buy(radio)	have($100 + price(radio))
V = a	assign(X,Y,a,b)	V = a & V \= X

B: X = a, Y = b
X (- Y

A: X = b , Y = b

The answers true and false may seem odd. What condition must be true before we move a from c to b in order to ensure that on(a,b) is true after we make the move? Since this operator itself adds on(a,b) to the world model, there are no requirements for what must be true beforehand to guarantee the desired condition holds afterwords. Literally, we would say that the requirement is

that "true must be true. But this is always the case, thus a vacuous condition . Similarly, the literal interpretation of the third example is that false must be true in order for on(a,b) to be true after we move c from b to a. Clearly, there are no circumstances where on(a,b) can hold immediately after moving c onto a; so it is impossible. Likewise, it is impossible for false to be true!

==========

SLIDE 10

EXAMPLE: Goal Regression

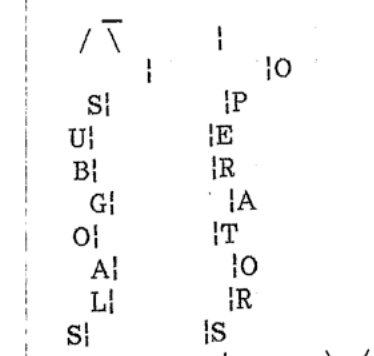
Goal Stack (w/ ops)

* move(c,a,table)
clr(a)

* move(a,table,b)

on(a,b) (protected)

on(b,c)



****

* move(a,b,?Z)
clr(b)

* move(b,table,c)

* on(b,c) * VIOLATION ==> REGRESS

What must be true before applying move(a,table,b) to ensure on(b,c) is true after its applied.

==========

SLIDE 12

EXAMPLE: Register Swap

X = 1
Y = 2

X = 2
Y = 1

```
assign(U,V,a,b) :
        assign value of variable V (which is b)
        to variable U (whose value is a)
```

If you find the goal regression heavy going, take heart. There are detailed explanations with examples to be found both in Nillson's "Principles of AI" (pp288-292; 321-333), and in Rich's "Artificial Intelligence" introductory text (pp 267-271). The slides above are not meant to be understood without accompanying explanation.

==========

SLIDE 14

SEARCH SPACES

States	Operators
Of the world	Transform the world state
Partial Plans (Goal Stack)	Reduce Goals to Subgoals : Rearrange goal Ordering

****

How to find out about world states?

A: Look it up

B: Analyse Plan

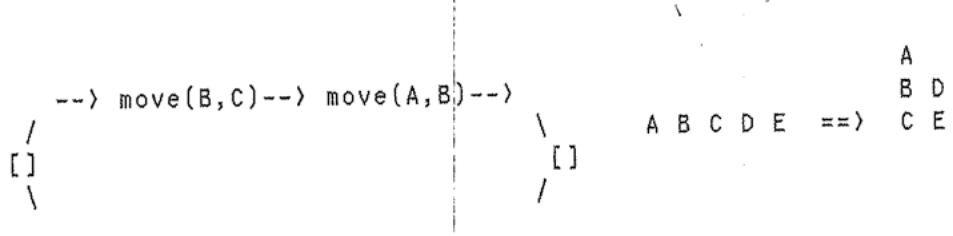
For an extended discussion of this distinction, see the second planning and search practical. This was the one which did the block stacking.

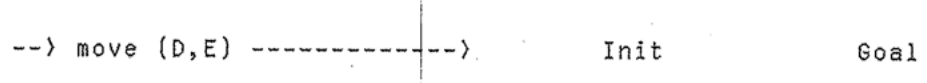
==========

PLAN REPRESENTATION

Until now, we have been doing state space search to get our plans. The plans themselves are represented as paths through a space of nodes each representing world states. Explicit information about the states of the world are kept around. The state of the world was recorded in an explicit data structure. Recall the missionaries and cannibals program.

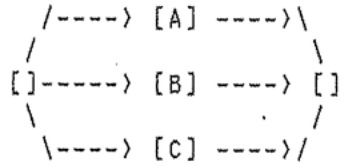
An alternate representation for plans is to store a list of actions, and the order in which they must occur. So far, we have only considered serial plans, where each action must occur in sequence, one after the other. None may occur in parallel. We say that serial plans are <u>fully</u> ordered. Alternatively, we can have <u>partially</u> ordered plans where the ordering of some actions is not prespecified. Consider the following example:





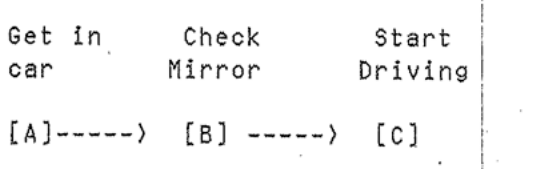
For state space plans, our nodes were world states and arcs were actions. In our new formalism, the nodes are actions and the arcs specify ordering constraints. This type of representaion is called a procedural net and was first used by Sacerdoti in a famous planner called NOAH.

In the example, we must put B on C before putting A on B, but it doesn't matter when we put D on E. This plan is non deterministic. This means that there are possibly many ways to execute it. We therefore have a more concise representaion formalism which can represent many possible plan instances. For example, consider the following simple procedural net containing three actions:

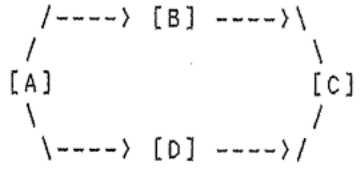


There are 6 possible serial plans represented in this concise format. They are ABC, CBA, CAB, etc.

Constructiong procedural nets is somewhat different than using state space search for creating plans. Consider the following simple example of a plan partially constructed.



We wish to add another action to the plan, namely that of fastening the seatbelt [D]. The constraint is that it must come after [A] but before [C]. Using state space search to construct serial plans would be force a choice between either ADBC or ABDC. If we use the procedural net, we can avoid making this choice and represent the plans as:



This is advantageous because later in the planning process, one or the other of the choices may be forced due to other considerations not being taken into account in the early stages of plan construction. Postponing commitments as long as possible can reduce the amount of backtracking necessary. This approach to plan construction is known as least commitment planning. Detailed examples may be found in [Cohen and Feigenbaum, 541-550] and [Rich, p271-276].

Note the difference in how we create plans. We are searching through a space of alternatives, however the nodes in this space are different sorts of animals. Till now, the nodes have been world states, and the operators have been primitive actions which can transform world states to other world states. Search stops when a state is reached which satisfies the criteria for a goal state.

When constructing procedural nets, the nodes in our search space are actually partial plans. We no longer keep track of explicit world states. The operators in this new search space come in three flavors:

- Reduce goals to subgoals
- Impose orderings on previously unordered goals
- Add new actions

The advantages to this approach are:

- Saves space, world states can be very large.
- Can modify the plan anywhere, not just by adding actions at the end
- Least commitment planning possible (for parallel plans)

The disadvantages are:

- It is difficult to see what is true in the world. When we have a copy of the world state, we just look at it. If we do not have it, we must inspect the plan to make sure it will work. This entails
    * Detect interactiing subgoals (ie protection violations)
    * Detect solutions (ie is the plan guaranteed to work?)
    * Determine where new actions can be inserted. This involves seeing what conditions hold in the world at some point in the

plan.

Projection is the term used for finding out about these things. More discussion may be found in the blocks world practical, and in [Charniak and McDermott] (pp485-489; 514).

SUMMARY FOR PLANNING

- State Space Search: Finds plans possibly using heuristic evaluation functions to aid search. [Eg missionaries and cannibals]

- Means Ends Analysis: STRIPS, Shakey the robot. This is a goal directed heuristic used in the context of state space search. It allows a limited form of planning ahead.

- Interacting Subgoals: STRIPS was stumped. Two approaches to handling this are:

    * Subgoal Promotion

    * Goal Regression

- Plan Representation: Serial/Parallel; Procedural Nets; Least commitment planning

- Plan Construction: Alternate search spaces

    * World States (State Space)

    * Partial Plans (Problem Reduction)

PROBLEM SOLVING AND SEARCH

****

N.B. Sections 1-3 have been rewritten and incorporated into chapters 1 and 2 of the notes which are available from Margaret Pithie. This section pertains primarily with the issue of search.

Mike Uschold 23 Feb 1987

****

4. Search Strategies

4.1. Introduction

We have seen a number of problems, and discussed various issues involved in finding good approaches for solving them. Even after the appropriate representation is found, the actual problem still needs to be solved. With the framework in place, we know how to recognise solutions and we have the machinery needed to go about looking for them. The final question remains: how to organise this search for a solution? In this section, we will formalise some of the discussion in section 1.

Using search for solving problems is a two step process:

1. Find a search space (ie, choose a representation)
2. Search it

We will present the basic search strategies in the context of searching the pure or-trees of the state space representation. In section 5 we will discuss their application to searching the goal (ie. and/or) trees of problem reduction.

We will discuss three basic approaches to search:

1. Generate & test
2. Uninformed
   - Depth First Search
   - Breadth First Search

3. Informed (heuristic)

   - Irrevokable
   - Revokable

4.2. Generate & Test

This is a very general method which subsumes the others. The basic algorithm is outlined below:

1. Generate a possible solution
2. Test to see if it is a solution
   Yes : Stop
   No : Go To 1

The key to this algorithm is the generator. There are three types:

- Random - No solution guaranted
- Systematic - Solution guaranted to be found if one exists
- Heuristic - Will find solution quickly. (We hope)

We have seen examples of each of these in our discussion of the 8-queens problem in section 2.2. We informally referred to these approaches as trial and error, brute force and intelligent. The so called intelligent generators can be a bit too clever and miss solutions. As such, they may or may not be systematic.

4.3. Uninformed Systematic Search Strategies

There are two of these:

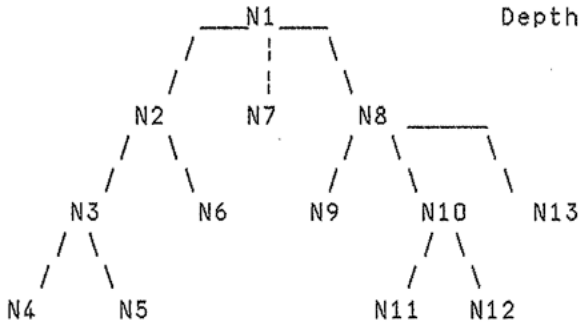
- Depth First Search - Explore a path as far as possible before considering alternatives
- Breadth First Search - Explore all paths of length N before considering any of lengh N + 1.

The algorithm for depth first search is found in figure 4-1.

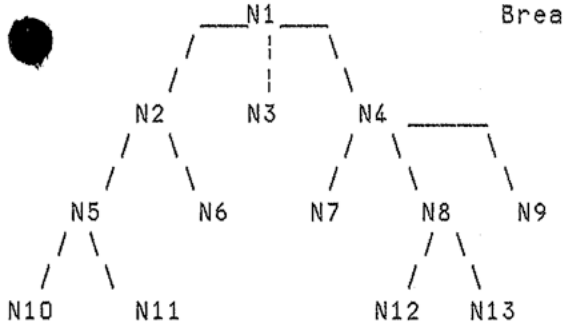
```
1. Put root node on LIST

2. REPEAT: IF LIST empty
              then : FAILED
              ELSE : X (- 1'st Item on LIST
              IF X = GOAL STATE
                     then :  SUCCEED
                     ELSE : REMOVE X
                            ADD SUCCESSORS to Front of LIST
```

Figure 4-1: Depth First Search

The algorithm for breadth first search is exactly the same except for the last line. Instead of adding the successors to the front of the list, they are added to the end. For those who are familiar with common data structures in computer science, the LIST in breadth first search is a queue (first in first out) and for depth first search, it is a stack (last in first out). Figure 4-2 indicates the order in which nodes are visited for these two strategies.



Depth First



Breadth First

Figure 4-2: Systematic Search Strategies

It is useful to compare these strategies and discuss any criteria which one

might use in deciding which is more appropriate for a particualar problem. Depth first search is more efficient on both time and space. It turns out that a stack requires less computational effort than a queue. The storage requirements for breadth first search are considerably greater. I leave this as an excercise to show.

Depth first search, however, may never find a solution. This can happen if the depth of the search tree is infinite. For these cases, a depth cutoff is required. This simply says that if the algorithm gets to a certain depth in the tree, then stop exploring that node and continue as if there were no successors. When a cutoff is used, is is clearly possible to miss the solution. Breadth first search does not suffer from this problem. It will always find a solution if one exists at some finite depth. If the solution path is long, however,it can get considerably bogged down by testing every possible node which is of that length or less. Suppose the solution in figure 4-2 is the second from the left on the bottom row. It is the 5th node examined by depth first search, but the 15th one tested using breadth first. On the other hand for short solution paths, depth first can get bogged down. Consider the node one the far right in the second row. Bfs finds it in four tries, where dfs takes eight.

Overall, the benefits of depth first search tend to outweigh the disadvantages. A depth cutoff can be avoided only if the search space is finite, or if there is a fairly high density of solutions. Otherwise, you will have to risk missing a solution or use breadth first search.

EXERCISE - Manually execute the algorithms for depth first and breadth first search on the example in figure 4-2. What is the most number of elements on **LIST** for each strategy? What are the exact contents of the lists in each case. (In the case of a tie, use the first occurrence.) What is the average number of elements in **LIST** for each strategy.

4.6. Informed Heuristic Search

For interesting problems, search spaces are too big. There are simply too many possibilities to check all of them, rendering the uninformed brute force techniques inadequate. We need a more informed search strategy. The study of heuristics is the study of finding ways to make intelligent choices during problem solving thereby reducing the amount of search. The word heuristicis somewhat loosely used in AI circles. A good synopsis is found in [--- 84] (p vii):

> ... heuristics [are] popularly known as rules of thumb, educated guesses, intuitive judgements, or simply common sense. In more precise terms, heuristics stand for strategies using readily accessible though loosely applicable information to control problem-solving processes in human beings and machine[s].

This is the general use of the word. Alternatively, a heuristic is often used to refer to an explicit evaluation function which is used to assess the goodness of a choice (to be made during problem solving) by assigning a number. I will first introduce three very general heuristics, two of which have seen already. After that, we will see how to use specific heuristic evaluation functions for solving state-space search problems. It is important to be aware of the different uses of the word. It will generally be clear from context which use applies.

Three general heuristics:

- Consider the 8-queens problem. I said it was a good idea to keep your options open. If there is choice between several operators to apply, each taking you to a different state, choose the one which will offer the most possibilities for future moves. There are a number of ways to use this general heuristic for different problems or even for the same problem. We used it as a guideline for creating the specific evaluation function: The number of free spaces on the board (See 7).

- Consider the process of testing to see if an operator applies, it may be the case that many conditions must hold. We must decide which preconditions to test first? A general heuristic to use here is to test the condition most unlikely to hold first. Since all the conditions must hold, you can stop as soon as any one fails. It would be a waste of effort testing many conditions which were found to hold only to get to the last one and discover that it does not.

- A final example of a general heuristic is to reason backwards from the goal state instead of the other way around [4]. This was discussed in the context of the monkey and bananas problem. The rationale behind this heuristic is that there are often many possible actions from the initial state, and choosing which one is likely to lead to a solution is difficult. When going backwards, everything you do is relevant to the goal state, since you are esentially undoing actions which can get you there. An example of where this works particularly well is for most maze problems that one finds printed in books or magazines etc. There are many false paths and dead ends when going forwards. But, in going backwards, there are usually fewer. Note that this is not inherent in the problem of mazes. They could as easily be constructed whth many false paths going backwards. In my own personal experience, they just usually aren't.

---

[4] Some people be unhappy with this use of the word heuristic, but in my opinion, it seems to fall under the definition above.

It is crucial to understand the tentative nature of heuristics. They are only guidelines and can't be assumed to work in every situation. There are times when keeping your options open in the short term may a poor strategy. There may be no way to a priori determine which conditions are "harder" to satisfy. There are times when reasoning forwards is the better idea. The frustrating thing is that it is difficult to know beforehand just how good a heuristic is.

Heuristic Evaluation Functions

A heuristic evaluation function is a procedure which assign numbers to choices for the purpose of determining what the best choice is. Typically, the choice is among operators to apply, or equivalently, which state to visit next. In the 8-queens problem, an operator is placing a queen in a square, the state is the board position. In most cases, heuristic evaluation functions assess the state directly. There are two common interpretations for the number associated with a state which are in common use:

- How good is a state?
- Cost of attaining the goal state from the current state (eg the distance from nearest goal state)

In one case, high numbers are desireable, and in the other, low numbers. For the sake of consistency, I shall always assume that the number represents a cost. If the measure is a measure of goodness, you can simply negate the numbers and retain the same point of view. This consistency is desireable since one algorithm can then be used for both cases.

Consider the following examples:

- N-Queens:
  * Number of choices for placing next queen
  * Total number of 'free' spaces left
- Noughts and Crosses:
  * 50 points for sure win
  * 30 points for a direct threat (ie. the opponent is forced to block)
- 8-Puzzle: See figure 4-3. There are 8 tiles and one blank space. Any tile may freely move into the blank space so long as it is adjacent to it. The object is to reconfigure the positions of the tiles to match the goal state. The scores are in reference to the initial state in the figure.
  * F1: Number of tiles out of place
    Score: (3)

* F2: Sum of horozontal & vertical distances to goal state of tiles out of place.
  Score: (5)
* F3: 1 point for each tile out of sequence
  Score: (3)
* F4 = F2 + 3*F3 Several features of a problem may be combined into a single function which more accurately asseses the overall situation.
  Score: (12)

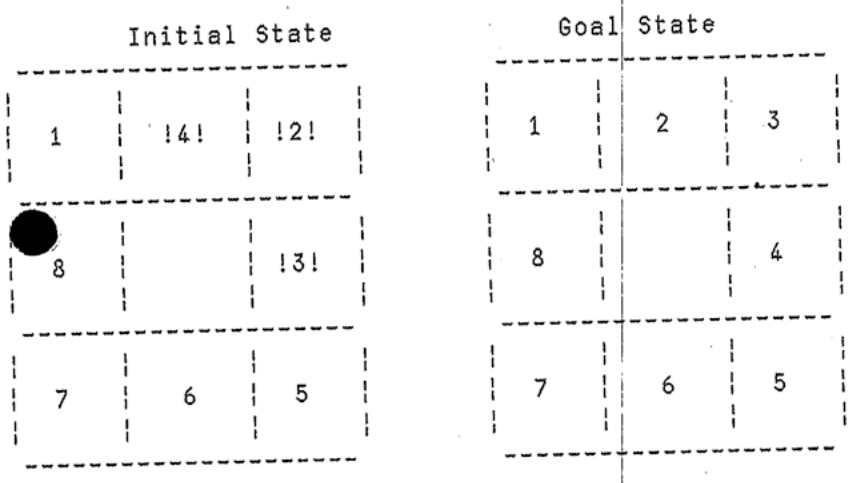


Figure 4-3: 8-Puzzle

These heuristics are used to guide the search, selecting the most promising of many choices. It is important that they be fairly simple, or else the extra cost involved in computing the value will nullify any other savings. The basic idea is that when you are in a state and there are several choices of operators to new states, evaluate each state by assigning a number to it. The state which seems most promising is the one to explore next. This contrasts with the uninformed methods of pure breadth first and depth first search which visits states in a prespecified order independent of any information about the states themselves. There are a number of algorithms for implementing this basic strategy. We shall discuss three of these.

- Hill Climbing
- Best-First Search
- A, A*

EXERCISE - Evaluate the states in figure 4-4 using the heuristics suggested on page 29.

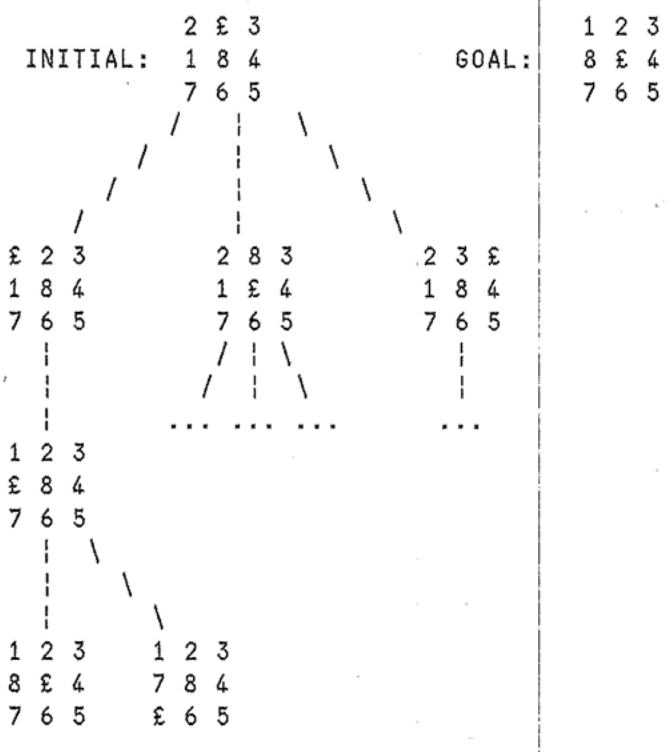


Figure 4-4: Partial Search Space for an 8-puzzle Problem

Hill Climbing

This techniques uses the metaphor of climbing hills to guide the search. You wish to attain the summit, and reason that the quickest way to get there is to go in the direction of steepest ascent from wherever you happen to be. In terms of the using the heuristic, this means select the successor state which has the lowest estimated distance to the summit. The algorithm is found in figure 4-5.

1. Current Node (== Initial Node

2. Repeat:
   If Current Node = Target Node
   Then STOP: you have a solution!

   Else: Expand Current Node
   If all successors have worse rating, STOP: failed!

   Else Next Current Node (==
   Most favorable successor

Figure 4-5: Hill-Climbing Algorithm

Note that this method is a local one. You only have access to information about immediate successors. Other choices which were not as good at the time may well have been better in the long run. In particular, you could be pursueing a false summit(a state all of whose successors are further from the goal than the current state), or may get stuck on a plateau a state from which all choices are equally good, but none of which improve the situation).

The disadvantages are:

- Irrevokable - Once a dead end is reached, there is no going back. Thus there is no guarantee of finding a solution.
- Very sensitive to the evaluation function - Whether it works or not depends on the existance of such features as false summits and plateaus and also on where you happen to start.

The advantages of this method are:

- Space efficient
- Can move to a solution very quickly
- Can be used in conjunction with other methods by getting off to a quick start.

Best-First Search

The algorithm is found in figure 4-6. The basic intuition here is to explore the best nodes first. It is similar to hill-climbing in this respect, except that it is a global method. Instead of forgetting the cost estimates for states not chosen in the first instance, they are remembered for possible future use. Thus, where hill climbing would sputter and die if it got to a node from where things only got worse (false summit) the best first strategy checks all the nodes that have been expanded so far and chooses the best one (even if it happens to be no better than the current node). We can see this method as sort of a combination of depth-first and breadth-first search. It starts off by going depth first in the same way as hill-climbing. When things stop improving, the breadth-first component comes in by jumping back up to previous nodes at higher levels in the tree. There is one possible disadvantage of this method, as it stands: it will not find optimal solutions. For some problems, it is important to get the best solution; an arbitrary one may not suffice. This method stops when it finds the first solution. The next algorithms we will discuss (A & A*) addresse this issue.

. Put initial node on search list

. Repeat :
    If list empty
    Then: STOP: You have Failed!!
    Else CURNODE (== First node on search list

    If CURNODE = Target
        Then: STOP: Solution found

    Else i. Remove it from list
         ii. Add successors to list
         iii. Sort list, best nodes first

Figure 4-6: Best-First Search Algorithm

The Algorithms: A & A*

These algorithms were developed to help find optimal solutions. In fact, they are identical to best-first search, except for some new twists in the evaluation function.

An optimal solution is one which is cheapest. For instance in the 8-puzzle, it's the one with the least number of moves, for the monkey and bananas problem, it might be the one in which the monkey expended the least amount of energy, etc. For some problems, such as the 8-queens, there is no sense of a cheapest solution. All solutions are equivalent since we are only interested the having a configuration satisfying certain properties. Once we know what the configuration is, the problem is done. The manner in which it was found is of no interest. There may be other solutions, but there is no reason for preferring one over the other.

The straight best-first search algorithm finds only one solution which is often of little use in getting the best one. What is needed is a method which can focus its attention on getting to the best solution first without wasting time with sub-optimal solutions. The insight which allows us to do this is to encode additional information in the heuristic function. Instead of only estimating the cost of getting to a solution from the current state, include as well the accumulated actual cost of having gotten to that state in the first place. There are thus two components of evaluation function:

- g(node) : Actual cost of getting to current node
- h(node) : Estimate of cost of getting to nearest goal state from node

The evaluation function used is: F(n) = g(n) + h(n). F(n) estimates of the cost of going from start to finish passing through node "n" (see figure 4-7). Note that the h is the heuristic component (ie, an estimate), g is known with certainty.

F(n) = g(n) + h(n)

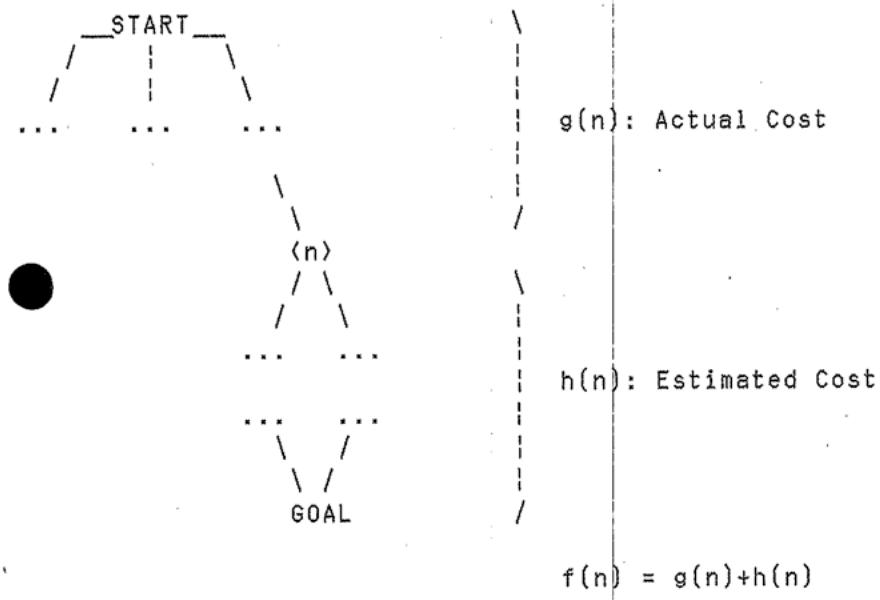


Figure 4-7: New Heuristic for Finding Optimal Solutions

To see how defining the evaluation function this way helps, let us understand it's behaviour. The function evaluated for goal states returns the actual cost of getting to that goal state. **EXERCISE 122** - Prove the last statement. Thus if a goal state is ever reached, and there are other better goal states (ie. ones which will ultimately be cheaper to attain) yet to be found they should have lower numbers on them and will thus be pursued. The should is the key. All will be well if the heuristic evaluation function is accurate. The component g(n) is accurate by definition. That leaves the heuristic component: (h(n)) to consider. There are two possible problems:

1. Suppose h is too low (ie, the algorithm thinks that the goal state being pursued is better than it actually is). In this situation, the algorithm will be chasing down non-optimal solutions unnecessarily. Eventually, when the solution is reached, f(n) is accurate thus revealing the poorness of the solution which should have been noticed before it ever got there. The algorithm will continue looking for the optimal solution and may eventually find it, but will have done a

lot of unnecessary work. In fact, if we let $h(n) = 0$ and $g(n) =$ depth(n) then the algorithm reduces to straight breadth first search which is guaranteed to find the optimal solution given enough time. **EXERCISE 123** Show this by way of a simple example.

2. Suppose h is too high (ie. the goal state being pursued is considered to be worse than it really is). In this case, the algorithm will incorrectly stop pursuing a goal state because something else seems better. This is clearly bad, since it could well be the optimal one that it misses.

If $h(n)$ can be shown to never be too high, then best-first search using the function $f(n) = g(n) + h(n)$ is guaranteed to find the optimal solution. This has to be true for the following reasons. We said that if the estimate for a any node on the path to the optimal solution ever *is* too high, then the algorithm might stop pursuing this node and miss the optimal solution. If, on the other hand, the cost is never overestimated, then the algorithm will continue to pursue the nodes on the path to the optimal solution until it is found. If the algorithm ever does stumble on a non-optimal solution, it's f value is the actual cost of that solution which, because it is non-optimal, is greater than the cost of the optimal solution node. The costs of the currently active node(s) on the path to the optimal solution node are guaranteed not to be overestimated (as are all the rest). Thus these costs are certain to be less than cost of the non-optimal solution node already found. Therefore, when that non-optimal solution node is found, the algorithm simply switches its attention to one of the nodes on a path to a better solution until the optimal solution node is eventually found.

A best-first search algorithm which has a heuristic evaluation function of the form: $F(n) = g(n) + h(n)$ is called algorithm "A". If, furthermore the heuristic has the optimality property described above the algorithm is called "A*". It seems somewhat of a misnomer to call these different algorithms just because of the nature of the heuristic used, but it wasn't my idea. Maybe it's like renaming addition to subtraction if the second thing being added is negative. Then again, maybe it's not. Hmmmm....

This is all illustrated in the example in figure 4-8. The numbers on the arcs are actual costs of traversing the arc. Thus, g values for a node are obtained by adding up the numbers on the arcs of the path back to the initial state (I). The h values are indicated inside the nodes. The f values are not shown. The node names are indicated in boldface. In step I, node B gets expanded since it has the lowest value for f. The node which gets expanded at each step is indicated by double square brackets.

$f(A) = 3+1 = 4$
$f(B) = 1+1 = 2$
$f(C) = 1+2 = 3$

The next node gets to get expanded is D. Note that the value for $h(D)$ being 0 doesn't mean it is necessarily a goal state. It's just an estimate which is too low. At step III, we have found a solution whose total cost is 6. There are

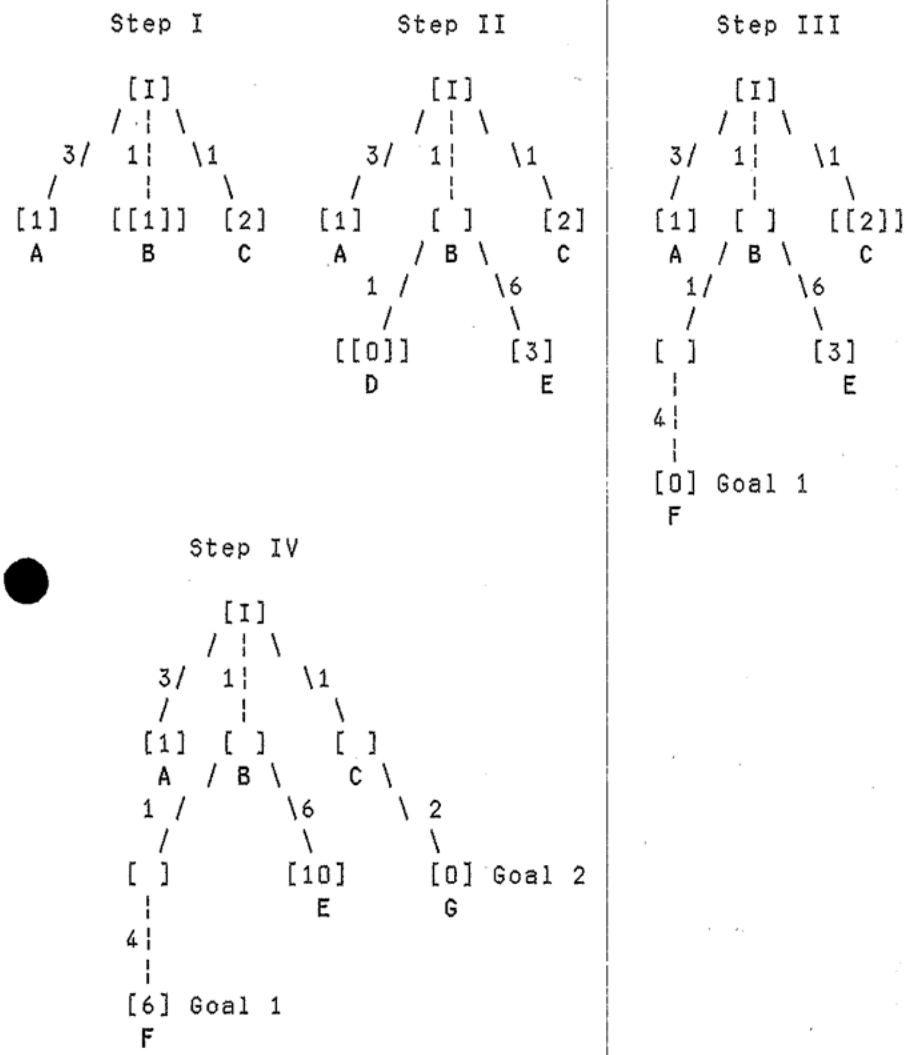


Figure 4-8: The Algorithms A & A*

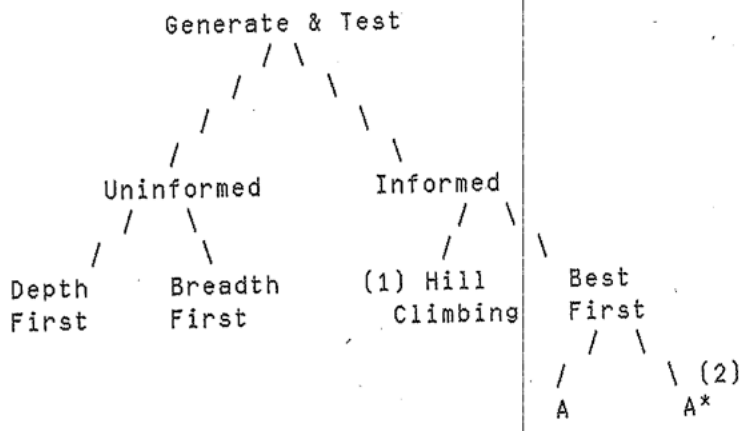
still better nodes to explore, in particular, nodes A and C of which C is the preferred choice which leads us to the optimal goal state. It is important to see what would have happened if the estimate for the cost of getting from C to it's nearest goal state, [Goal 1] had been too high. In particular, suppose $\underline{h}(\underline{C})=7$ instead of 2. In this case, the algorithm would have stopped when it found the first suboptimal goal state. Note the effect of the $\underline{h}(\underline{D})$ being too low. The algorithm wasted some effort chasing down a suboptimal solution. The perfect heuristic which is neither too high or too low will cause this algorithm to zoom in on the optimal solution immediately. It is guaranteed to find it by never overestimating the cost to a nearest goal state. Similarly, it avoids chasing sub-optimal solutions by failing to be too low.

In practice, it is extremely difficult to hit the right balance. In order to guarantee optimality, the heuristic has to be kept low. But, we have seen that

f it is too low, its usefulness as a heuristic (heuristic power) is severely imited. The whole reason for heuristics in the first place is to avoid chasing down unlikely paths. If the heuristic function is regularly too low, then bad paths are going to look good and they will be checked out anyway so there's no point in having the heuristic at all. It is often worthwhile to sacrifice optimality for heuristic power by letting the heuristic occasionally overestimate the cost to a goal state, but keeping it sufficiently high to be a powerful aid in reducing the amount of search. A nice example of this is the heuristic F4 on page 29. See [Nillson 80] (page 86) for an impressive reduction in search using this heuristic.

4.5. Summary - Search Techniques

We have discussed a number of basic search strategies and algorithms for problem solving (see figure 4-9). So far, we have only addressed the problem of searching the pure OR-trees of the state space paradigm. In the next section we explore some of the issues involved in searching the more general And/Or (Goal) trees which naturally represent problems cast into the problem reduction framework. Furthermore, we will discuss the close relationship shared between problem reduction and game playing. Most of the ideas, techniques, and algorithms presented in this section are highly relevant and applicable to searching goal trees. However, there are some important differences which must be addressed.



(1) Irrevokable
(2) Finds optimal solutions

Figure 4-9: Search Taxonomy

5. Applying Search Algorithms to Problem Reduction Representation

In this section, we explore the issues involved in applying the search techniques we have studied to problems cast into the problem reduction framework. After a brief review of problem reduction, we discuss some of the important differences between this and state space representations with an aim to discovering what modifications or new methods are required. Following this, I show how game playing can be viewed as problem reduction by describing an alternate interpretation of goal trees. Finally, I present some search techniques which are special to game trees.

Review Problem Reduction

Problem reduction is an alternate way to view or represent problems. It is useful when a problem can be decomposed into subproblems which can be further decomposed etc. We have seen some examples of problems which are naturally cast into this framework. These are the Towers of Hanoi, the counterfeit coin, and the problem of pleasing your date. Instead of speaking of problems and subproblems, we can speak of "goals", which decompose into suboals. The goal, is to solve the problem. Problems are represented by "goal" trees. The root node of a goal tree is the top level goal, the branches correspond to subgoals. There are "and" nodes and "or" nodes. The former are appropriate when the goal splits into subgoals, all of which must be achieved in order for the parent goal to be achieved. "Or" nodes, on the other hand, correspond to situations when a goal may be achieved by some number of alternate methods, any one of which is sufficient. In general, a goal tree will have some sort of constraint(s) associated with it. Examples of constraints include the 25 pound limit on pleasing your date, or for the counterfeit coin problem, the number of weighings must be no more than three. Finally, a goal tree has leaf nodes which are primitive subgoals which decompose no further. They correspond to primitive operations in the problem domain which are immediately achievable. For the towers of Hanoi, the leaves are the subgoals which correspond to moving individual discs. For the counterfeit coin problem, the leaves are the subgoals which involve no more weighings, because the counterfeit coin has been identified as heavy or light. There is no work which needs to be done to determine how to achieve a primitive subgoal. You can just go ahead and move the disc, or in the counterfeit coin problem, the coin is already identified.

Goal trees, are similar to and/or trees, in fact they are a special sort of and/or tree. The only difference is the existance of some constraints. If a goal tree has no constraints, then we call it a pure and/or tree. For the sake of simplicity in the discussion which follows, I will use the term goal tree even if there are no constraints.

.1. Searching Goal Trees

his is a rather more more complicated procedure than the searching we did in tate space representations. In state space search, we start with an initial tate (of the world), and try to transform it into some goal state. Each node n the search space corresponds to some world state. From each state, there are number of successor states possible. A solution will contain only one of hese choices. Thus, these nodes can be viewed as "or" nodes. In fact the whole state space can be viewed as a pure or-tree. There are no "and" nodes. it is exactly these "and" nodes which make life difficult for us when searching goal trees. There are two possible approaches one might consider:

1. Convert to state space representation, and then use the or-tree search algorithms (eg. best first). (See C&D)

2. Search goal trees directly.

Since, as I mentioned above the conversion from problem reduction to state space is rather complicated, we take the latter approach. We must make one thing very clear before this discussion gets underway. That is, what precisely is the nature of a solution? In state space search of pure or-trees, a solution was simply a path to a goal state, or the goal state itself. In goal trees, the solution is no longer a simple path. Rather, it is a more complex subtree with "and" nodes and all their successors etc. See figure below. Note that in a case where only "or" nodes are in a solution, they correspond to simple solution paths in state space representations.

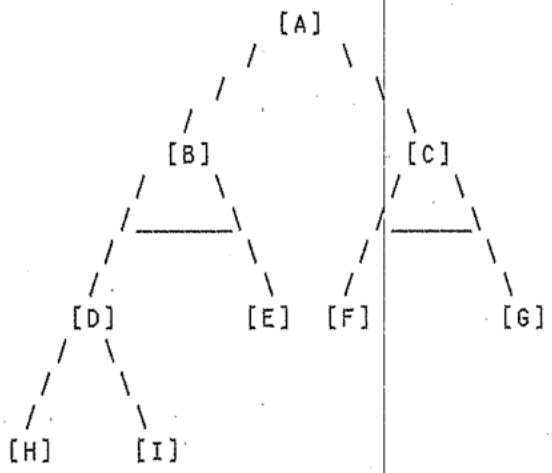
How might we characterise a solution for a goal tree? We define this recursively by introducing the notion of a "solved" goal tree node. We have three types of node, which may be solved in different ways.

1. And nodes are solved when all of its successors are solved

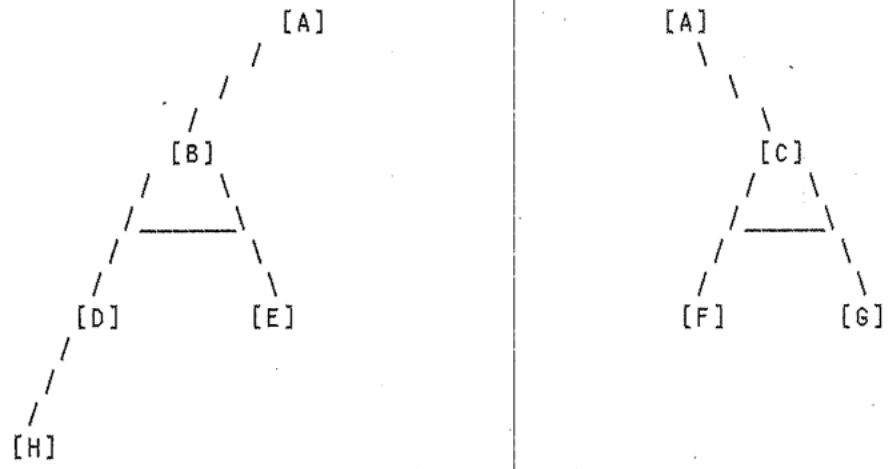
2. Or nodes are solved when (at least) one of its successors is solved.

3. Leaf nodes are solved if they obey all the constraints

We say the goal tree is solved when its root node is solved. Note that a root node may be an any type of the above nodes. Of course if the root node is a leaf node, we have a trivial goal tree (ie. there is no problem). If there are no constraints to worry about, we notice that leaf nodes are automatically solved. This is reasonable, since we are interpreting them as primitive problems which do not decompose. Thus, they are not really "problems" in the normal sense, since there is nothing to solve. Consider goal trees which define grammars and may be used to build sentences. The non-leaf nodes represent rules for putting types of objects together. For example, a sentence may be composed of a noun phrase *and* a verb phrase (an "and" node). A noun phrase may be



Complete Tree



Two possible solution subtrees.

Figure 5-1: Example Goal Tree

composed of a simple noun, *or* a determiner followed by a noun (an "or" node). These may be further decomposed as defined by the grammar. Finally, at the leaf nodes, we have the strings themselves. The work stops there, with no more grammar rules to worry about. We just take the strings. These are the primitive "problems".

Now that we've characterised the nature of the solution objects that we are looking for (subtrees satisfying the conditions listed above), how might we systematically go about searching for them to solve the problem?. Let us attempt to apply a best first search, as we did for or-trees and see what happens. In the figures below, I use squiggly brackets to indicate a score which reflects combined scores of children of an "and" node, and round brackets otherwise. The scores reflect cost estimates, and we are therefore trying to

achieve low scores. NB: I have used a shorthand way to present goal trees here which does not explicitly separate out "and" nodes and "or" nodes. The meaning should be clear, however for purposes of this illustration.

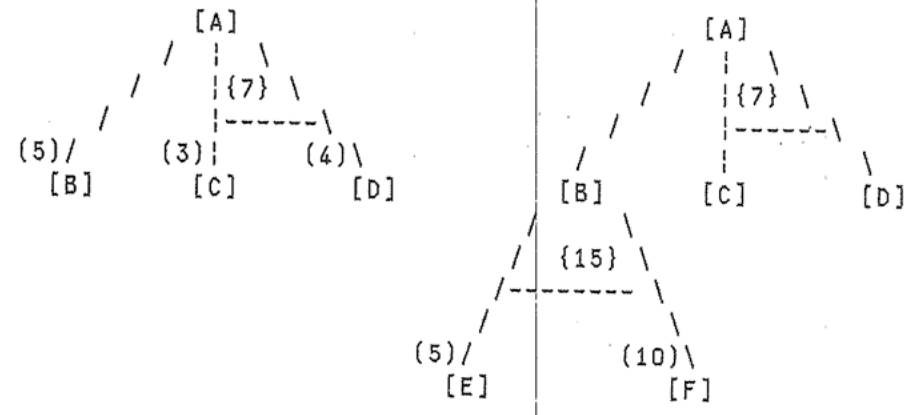


Figure 5-2: Heuristic Search of Goal Trees

In the first figure, we see that the best individual node is [C]. However, we note that any solution path which includes [C] also includes [D] because they are linked together indicating "and". Thus, it would be better to expand node [B] since it has a cost estimate of only 5 where any solution containing node [C] is estimated to cost 7 units. It is clear that this best first approach will not work as it did for state space search. Our solution objects do not correspond to simple paths to nodes. Furthermore, there is no explicit node which represents a goal state. (Remember, nodes in a goal tree represent (sub)goals, or (sub)problems). Instead of choosing the best node, we choose a node on the best potential solution *path*. Note that in a pure or-tree, these are the same thing. The method of searching goal trees differs considerably from what we have done until now. It will however, still retain the flavor of best first search. I note below, five important differences between searching goal trees and pure or-trees from state space representation.

1. For goal trees, choose a node on the best potential solution path rather than the best node to expand next.

2. When trying to expand new nodes in a goal tree, you must always check to see if any constraints are violated. Eg. cost. In state space search, this sort of thing was done when determining whether operators were applicable to states.

3. The "and" nodes of goal trees require some extra bookeeping. In particular, one must make sure that all successors of any "and" nodes are eventually solved.

4. For goal trees, one must update values on nodes when new information

from further down the potential solution path becomes available. For example, when node [B] was expanded, we had to update the cost of the solution path containing [B] from 5 to 15. Furthermore, it may need updating again when(if) nodes [E] and/or [F] get expanded later on. The pure or-trees we saw in state space search did not require this bookkeeping. This is because, once we are at a node (state), then what is behind us is no longer of concern. This is related to the first point about the nature of the solution. The reason previously expanded nodes are of no concern is that the solution is a simple path to a goal state, or the goal state itself. Each time we expand a node in an or-tree, we simply apply the evaluation function to the new nodes and carry on. The "and" nodes in goal trees cause all the grief. A solution is not a simple path, but rather a subtree which will generally contain some "and" nodes. Consequently, when we expand a node in an goal tree, we may well have to worry about previously expanded "and" nodes some of whose successor nodes may be expanded further.

5. When searching goal trees, we must have some scheme for combining heuristic functions at "and" nodes so that we can properly assess how good the solution path which contains them is. In the example above, we viewed the nodes "anded" together as being separate subproblems all of which had to be solved. If we view the heuristic function as a cost, then it seems reasonable to add the costs. There are, however other possibilities. Consider, for example, the counterfeit coin problem. The "and" nodes arise when we perform a weighing. All three possible outcomes of the weighing must be checked out to get a strategy which is guaranteed to work. In this case, we only really have to worry about the worst outcome, not the others. We might use a heuristic for this problem which is the size of the subproblem remaining. If we start with 12 coins, and weigh 4, then the worst that could happen is we are left with an 8 coin problem. If we are lucky, we only have a four coin problem. In any case, it doesn't make much sense to add these values to get the value for the and node over all. A more appropriate way to combine these is to take the maximum. You have to assume that sod's law is operating. The distinction here is:

   a. Multiple actions, all of which must be performed

   b. Multiple possibilities all of which must be checked out, but only one of which will happen in fact.

Another example of type "1" would be the problem of pleasing your date which involved food and entertainment ("and" node). The cost of the "and" node is (most reasonably), the sum of the costs of each successor node. Another example of type "2" is the whole area of finding strategies for playing games. This will developed in the next section. Note that the counterfeit coin problem is much like a game, where the opponent is chance, and the solution is a strategy

for guaranteed success.

)

5.2. Game Trees

See [Charniak & McDermott 85] (pp 281-286) for a nice introduction to games trees interpreted as goal trees.

Note the similarity to state space representation. We have explicit states which correspond to board positions and explicit operators which transform states to other states. (For convenience, I will assume we are talking about some sort of board game, although these techniques apply to other games equally well. When we refer to a "board position" we really are referring to some complete description of the state of the game whether or not there is any board.) In spite of this similarity, we can not use the state space approach for solving the problem of how to play games. The reason is that the nature of a solution is entirely different. In state space search, you are only interested in some sequence of operators which will get you to a goal state. In games, this is not adequate. Many goal states (winning positions) could be achieved if your opponent made lots of stupid moves. Getting to a goal state is not enough. You have to guarantee that you can get to a winning position no matter what your opponent may do. In game trees, the nodes do in fact correspond to explicit board positions. However, the "problem" from any board position is not simply to find a path to a winning position (goal state). The goal at a node is to find a winning strategy from that position. The subgoals consist of finding winning strategies for each of the board positions which result from possible moves from some position. The goal tree interpretation to this problem fits quite naturally.

Complete Game Trees: All possible moves and board positions are represented. Leaf nodes are one of {WIN, LOSE, DRAW}. The goal is to find a winning strategy, ie one that guarantees you can win whatever your opponent may do.

Incomplete Game Trees: All possible moves for a limited number of "turns" are represented. The leaf nodes are simply board positions whose "goodness" must be measured. The goal for incomplete game trees is different. You cannot hope to find a winning strategy. Rather, you settle for finding the best move.

Searching Game Trees:

- Two players: MAX - Strives for high scores (me)
  MIN - Strives for low scores (you)

- Static game state evaluation function Estimates how good the state of

the game is from MAX's point of view. Evaluates to some number.

- Lookahead Depth The number of levels represented in the tree. Ie, how many moves ahead are being considered.
- Mini-Max[5] MAX nodes: Take maximum of the values of successor nodes
  MIN nodes: Take minimum of the values of successor nodes The values percolate up the tree starting from the leaves

Searching complete trees:

- Apply mini-max. (You could let Win = +100, Lose = -100, Draw = 0)
- What does root node evaluate to?
  * 100 for MAX (-100 for MIN) ==> Winning strategy exists!
  * 0 for MAX/MIN ==> There is not a strategy which guarantees win, but you can guarantee at least a draw -100 for MAX, (100 for MIN) ==> You can't even guarantee a draw. If your opponent plays his best, your guaranteed to lose.

Searching incomplete game trees:

- Grow tree to some depth (lookahead) Apply static board evaluation function to each leaf node. This is similar to the heuristic evaluation functions we saw for 8-puzzle, and other problems which we used state space search to solve.
- Apply mini-max to assign values to non-leaf nodes, bottom up eventually leading to the root.
- The value of the root is the value of the best possible move that can be made from that node.
- In real game playing programs, at this point, the tree is extended furthur to take into account one more level, and mini-maxing is applied again. Many of the dynamic values applied in the previous iteration will be inaccurate and must be updated.

NB: The heuristic function used plays the same role as other evaluation functions we have seen for some other problems (eg. 8-puzzle) in that it

---

[5] See notes for lecture 1, on "What is an AI Technique?"

measures the "goodness" of a state. However, it is not used in the same way. In best first search of or-trees, we chose which node to next expand on the basis of its heuristic evaluation. When searching game trees, we must generate the tree first to some level, apply the evaluation function to the leaves of the tree and then use mini-max to compute the values of nodes higher up the tree.

Problems:

- Game trees are too big, you always have to settle for incomplete trees for interesting games.
- It may be very difficult to find good board evaluation functions Eg. For a draughts program, 25-30 separate features of the board were taken into account to achieve expert performance.
- Horizon Effect: Consequences of moves beyond the lookahead depth are unaccounted for. Errors may result from this.

Controlling Search: Alpha-Beta Pruning

One technique for helping to trim the search space is known as alpha-beta pruning. The reasons for this name will become clear later. This is described in the references above. Briefly, it is a technique for avoiding unnecessary search. Consider the complete (admittedly trivial) game tree(1) in figure 5-3. The leaf nodes all evaluate to Win, Lose, or Draw. We apply mini-max to evaluate the non-leaf nodes. Suppose we start by looking at node [B]. There is no difference between alternatives [D] and [E], so [B] evaluates to "Win". At this point, if we consider node [A] which is a MAX node we note that there is a possible move which guarantees that MAX can win (namely, to node [B]). If that is the case, then why bother considering any other moves (nodes)? You will never do better than getting a winning strategy, so you might as well stop (unless, of course you have some kind of superiority complex and desperately need to find more and perhaps better ways of defeating your opponent).

Suppose we change the values of nodes [E] & [H] to "Draw" (see Tree 2 figure 5-3). Now, node [B] evaluates to "Draw", since MIN will choose his/her best move (ie, the one which is worst for MAX). Remember, the Win, Lose, and Draw valuations are from MAX's perspective. Now, we still have reason to keep searching. MAX now knows that s/he can at least get a Draw, but why not go for a Win? We start to evaluate node [C] next and begin by looking at node [F]. This is a "Lose" for MAX which is bad news. What is even worse news is that it's MIN's turn at that point and MAX must assume that MIN would make that move if given the chance. But MAX already knows that s/he is guaranteed at least a draw, (by moving to node [B]). If MAX were to move to node [C], s/he would risk losing. Surely, there is no reason to do that. Note that we have come to the conclusion that MAX will not move to [C] without ever having checked the values for [G] or [H]. We say that they have been pruned from the game search tree.

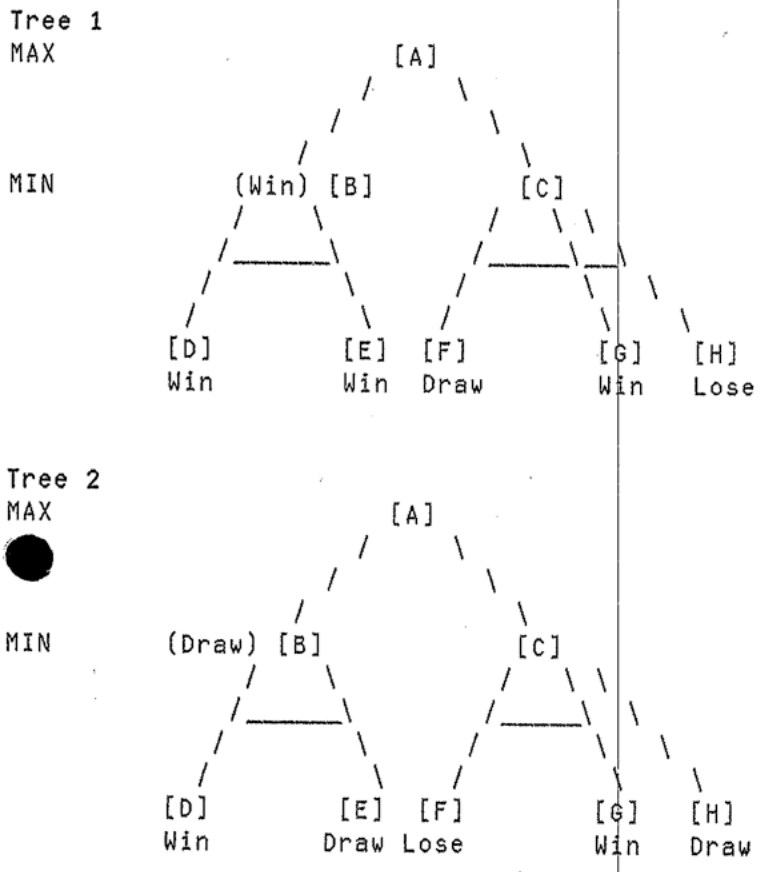


Figure 5-3: Complete Game Trees

This procedure generalises for incomplete game trees and may be applied in the same way. The only difference is that you will have numbers instead of Win, Lose, and Draw. These numbers are estimates for how good the board position is for MAX. This is described in detail with examples in the references listed above. The primary intuition is to skip nodes which you know a priori can not improve the current sutuation. The way to do this is to keep a tally of the best you can do for a particular node (Eg. When considering node [A] above, the first successor evaluates to "Draw". This becomes the tally value corresponding to the current best move considered so far). If at any point when considering alternate moves, (Eg. move to node [C]) it becomes clear that the alternative can do no better than what you already know you can do (from the tally value) then abandon searching that alternative.

The literature distinguishes two types of situations for abandoning search before an alternitive move has been fully explored, one for MAX nodes and one for MIN nodes. In the former case, the tally which keeps track of the best move will be a high value and search stops when considering alternatives which are lower. For MIN, this is reversed. The best move corresponds to a low value and search stops when considering alternatives which are higher. This is important if you are going to be writing a computer program to do this, but for the

purposes of understanding why the method works and remembering how to apply it on examples, I think it's easier to forget about this distinction. Simply keep track of the "best" move and remember that best for MIN nodes is different than best for MAX nodes. The terms alpha cutoff and beta cutoff are used in the literature. This is because you are cutting off part of the search tree. They refer to the two different situations for abandoning search described above. This why the method is known as alpha-beta pruning.

SO WHAT?

These techniques are in fact used for game playing programs that are used in competitions and/or reside in computers that you can buy at the local shop. In particular, chess playing programs are already playing master level chess. There are programs for draughts and backgammon which are nearly world class competitors.

THAT'S WHAT!

What are you expected to know?

1. What are state space and problem reduction representations? What are they good for? How can you use them. Characterise the nature of a solution for each representation and how they differ.
2. How to search a simple goal tree.
3. How to view a game tree as a goal tree.
4. How to create a game tree given a game and rules for playing.
5. The difference between complete and incomplete game trees and what the goal for each case is.
6. How to apply mini-max (using alpha-beta pruning) to example game trees.

Further Reading:

Problem Reduction (Goal Trees):
Rich, pp 87-94
Barr & Feigenbaum, AI Handbook, Vol I, pp 74-83, pp 36-42
Pearl, "Heuristics", pp 14-31
Charniak & McDermott, 270-281 (C&D)

Game Trees:
Rich, pp 113-131
Barr & Feigenboum, AI Handbook, Vol I, pp 84-108
Nillson, Principles of AI, pp 112-126
C&D, 281-291

6. Answers to Selected Exercises

8-Queens: Count the number of free spaces that are removed by the selection of a particular square. So, the choices in the diagram evaluate to 6, 5, and 2 respectively. From this point of view, low scores are desirable. This is really equivalent to measuring the number of free spaces. Only the point of view is different. But, note that it is a different function returning different values which must be compared in different ways.

122: This is true because the estimated cost of getting to the nearest goal state from a goal state is 0 (you are already there!)

REFERENCES

[Barr & Feigenbaum 81]
Aaron Barr & Edward Feigenbaum.
The Handbook of AI.
HeurisTech Press, 1981.

[Charniak & McDermott 85]
E. Charniak and D. McDermott.
Introduction to Artificial Intelligence.
Addison Wesley, 1985.

[--- 84] Judea Pearl.
HEURISTICS - Intelligent Search Strategies for Computer Problem Solving.
Addison-Wesley, 1984.

[Nillson 80] Nils J. Nillson.
Principles of AI.
Tioga Publishing Company, 1980.

Table of Contents

List of Figures