University of Sheffield

Department of Computer Science

COM1080: AI Techniques

This module covers a number of computer-based problem solving methods developed for Artificial Intelligence.

We are looking at techniques for **symbolic** rather than **connectionist** AI.

We will take a practical approach, showing how these techniques can be programmed using Java.

1 REFERENCES

There are many AI textbooks covering this material, for example:

- **Winston**, Artificial Intelligence, Addison Wesley.
- **Luger & Stubblefield**, Artificial Intelligence and the design of Expert Systems, Benjamin Cummings.
- **Nillson**, Artificial Intelligence: a new Synthesis, Morgan Kaufmann.
- **Cawsey**, The Essence of Artificial Intelligence, Prentice Hall

There is no **requirement** to buy any of these to follow the course, but you might find it useful to look at them.

The COM1080 web site is

www.dcs.shef.ac.uk/~pdg/com1080

Java code used in the course is in the **java** directory on that site and also in

/share/public/com1080/java (unix)

\\holly\public\com1080\java (windows)

2 SEARCH

We look first at **problem-solving by search**. In the early days of AI it was thought that this paradigm might prove **sufficient** to produce generally intelligent behaviour. While the modern view is that there is more to it than that, it is still **necessary** to be able to conduct a search sensibly.

A search-based problem-solver makes decisions based on the **likely consequences of potential actions.** The only knowledge used is what actions are available in a given problem state, how these actions will change the state and, in the general case, costs associated with these actions.

From an initial problem state, the search engine generates the states which can be reached by a single action. It then selects one of these states and (if the new state is not a goal), generates all its successors and so on.

2.1 Examples:

We'll use three simple search problems as exemplars:

2.1.1 Jugs Problems

A typical jugs problem is:

Suppose you have a 7-pint jug, a 4-pint jug, a water tap and a sink.

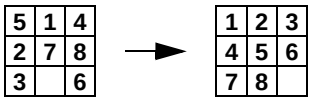
You are allowed to fill either jug from the tap, empty either jug into the sink, or pour from one jug into the other until either the first jug is full or the second jug is empty.

Show that by a sequence of such moves it is possible to measure any number of pints between 0 and 7.

We always want to **generalise** our solutions, so we will consider that our jugs can be any capacity (not just 7 & 4 pints), and the target amount can be anything we like.

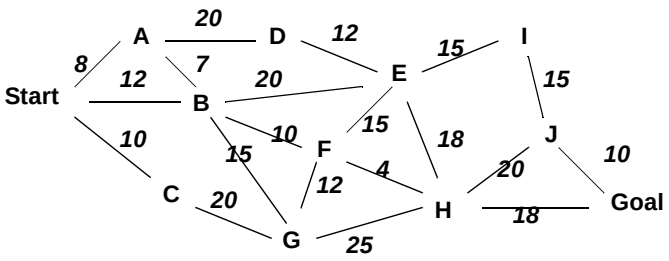
Note that **there may be no solution** to some given problem.

2.1.2 8-Puzzle



Rearrange the tiles to the desired pattern by sliding adjacent tiles into the hole.

2.1.3 Map Traversal



- Think of **A,B** etc. as cities

- The numbers are distances between cities.
- Find the shortest route from Start to Goal.

Later on we will look at how search techniques are adapted for game-playing and how they are used in speech recognition.

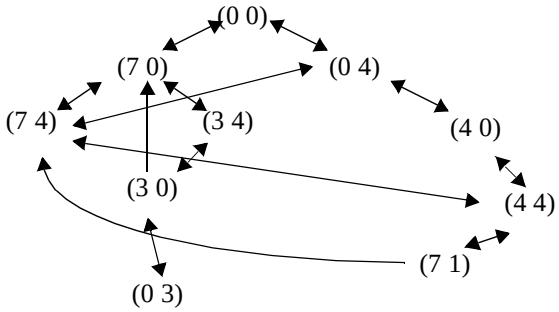
All problem-solvers search to some extent. Effective use of knowledge can make the search trivial or at least manageable.

2.2 Definitions

We consider that the problem involves moving between a number of problem **states.**

State Space contains all allowed problem states. Consists of a **graph** in which states are connected by **arcs** (legal moves from one state to another).

- *Jugs*: all possible configurations that can be reached from the initial state where both jugs are empty:



Incomplete state space for jugs with capacities 7 & 4

- *8-puzzle:* all possible arrangements of tiles.
- *Map Traversal:* the map.

The **expand** function takes a given state and returns a list of all the states that can be reached by a single move from that state.

- *Jugs* - all possible configurations by one move from a given state
- *8-puzzle* - all possible tile configurations from a given one
- *Map-Traversal*: All cities that can be reached from a given city.

Sometimes it is natural to think of the **expand** function as invoking **operators.** An operator is a function which embodies a 'legal move' from one state to another.

- Jugs: empty j1 to sink, empty j2 to sink, fill j1 from tap, fill j2 from tap, pour from j1 to j2, pour from j2 to j1.
- *8-puzzle:* Move-Up, Move-Down, Move-Left, Move-Right

Not all operators will be applicable to a given state.

The operators/expand-function represent the 'rules of the game'. The process of considering all the successors of a given state is called **developing** the state.

A **problem** is seen as finding a **path** through state space from a given **initial state** to some **goal state.**

The **solution** can be expressed either as the sequence of states traversed, the sequence of operators used or the goal state, as appropriate.

There may be **one or many goal states.** In general, we need a **goal test** that, given a state, tells us whether it is a goal or not.

It may be sufficient to find any solution, or we may be required to find the '**best**' solution.

- *Jugs/8-puzzle:* minimum number of moves.
- *Map traversal:* least costly path.

2.3 Search Trees and the Dynamic Programming Principle

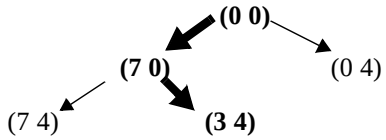
We need to distinguish between the **state space** and the **search tree** that is grown when we explore it.

State-space is generally a graph (i.e. there may be >1 way of reaching a state), but in the search tree each **search_node** will have 1 parent.

This is because **we only need to remember one way (the best way so far found) of reaching any state.**

This is called the '**dynamic programming principle'** and is fundamental to search algorithms. In simple cases, it means that if, in the course of a search, we come across a node we have already seen, we need do nothing.

Each node in the search tree will be associated with a different state, but may also contain other information (for instance, a pointer to the parent node):

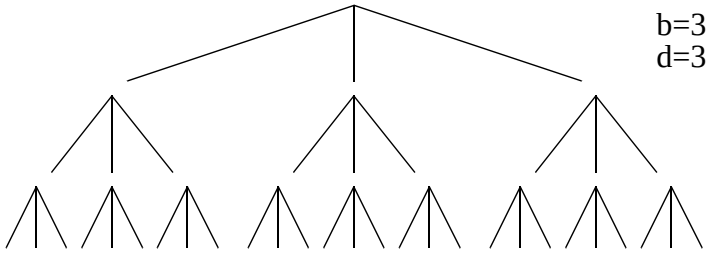


Jugs search for target 3. Solution path in bold

2.4 AI and the Combinatorial Explosion

The problem of relying on search is the **combinatorial explosion.** The potential search tree grows explosively with increasing depth:

If there is a uniform **branching factor b** and we search the tree to **depth n**, there will be $\mathbf{b^n}$ tip nodes.



Chess:

- b averages around 30
- for a complete game, n will be around 100
- So exploring the whole tree will produce 30^{100} or around $\mathbf{10^{145}}$ nodes.
- This is considerably more than the number of atoms in the universe

We therefore have to arrange to stop the search before resources run out, and to search selectively.

If the state space is large, we may have to trade-off the quality of the solution against the amount of effort required to find it.

It is important to choose our problem representation so that the search space is not too large.

We can try to do this by deploying more knowledge (e.g. not all legal chess moves are useful). We return to this later.

'The more knowledge, the less search'.

Many problem-solving programs can be seen as conducting a state-space search. Later on we will relate *forward-chaining and backward-chaining (rule-based systems) and structure-matching (frame-based systems)* to search.

2.5 Programming a State Space Search

We will need

- A way of **representing the states**,
- An **expand** method**,** perhaps calling **operators.**
- A method acting as a **goal predicate,**
- A method implementing a **strategy** for navigating the State Space.

2.6 Assessing Search Strategies: Criteria

- Is the strategy **Guaranteed** to find a solution if one exists?

- Is the strategy **Admissible**, i.e. guaranteed to find the best solution?, or a solution of sufficient quality?
- How **Efficient** is the strategy: i.e. how much effort does it expend in finding a solution?

The **optimal strategy** would visit only those nodes which turn out to be on the solution path.

For an individual search, **Efficiency = (Number of nodes on solution path)/ (Number of nodes visited)**

Note that *admissibility is a property of the algorithm*, but *efficiency is different for each different search we perform.* To assess the efficiency of a search algorithm in a particular problem domain we would run the algorithm for a representative set of problems from that domain & average the efficiency over these trials.

2.7 A Skeleton Algorithm for State Space Searching

Expressed imperatively:

We maintain

- a list of nodes awaiting development. This is called the **open** list or **agenda.**
- a list of nodes already developed. This is called the **closed** list**.**

Initially,

- we put the start node on **open.**
- **closed** is empty

We repeatedly

1. Check if **open** is empty. If so, exit with failure
2. Use the search strategy to **select** some node (the **current_node**) from **open** for development.
3. Remove **current_node** from **open.**
4. Check if **current_node** is a goal - if so, exit with success.
5. use **expand** to find **current_node'**s successors,
6. add all successors which are not already on **open** or **closed** to **open.**

2.8 Java Implementation

We will develop a generalised search engine in Java, which we can then use in any search problem domain. The user has to set up what is necessary for her domain: we specify below how to do this.

This policy of abstracting the problem-solver is typical of AI (and CS in general). It saves repeated work & forces us to understand what we're doing.

The baseline version of the search engine is in

www.dcs.shef.ac.uk/~pdg/com1080/java/search1

2.8.1 classes and methods

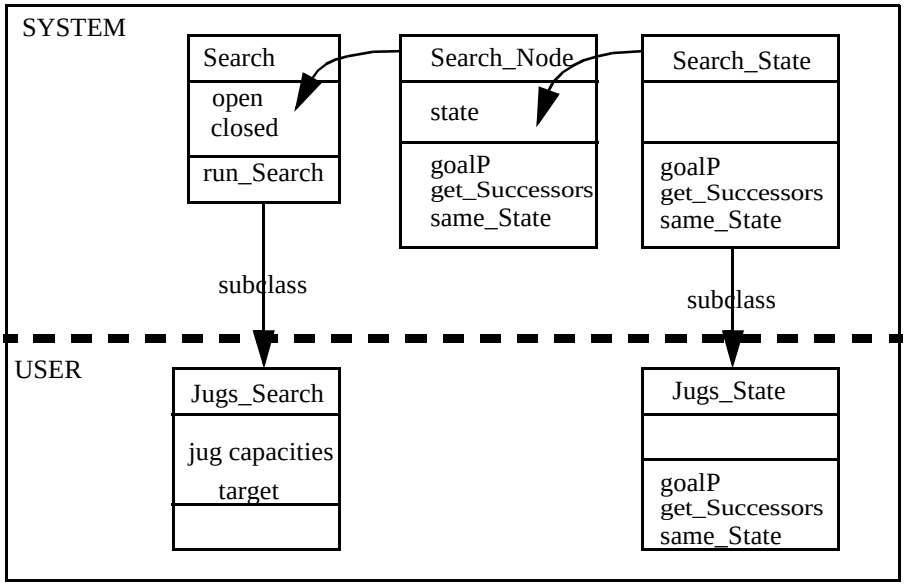
- We need a class which contains a method **run_Search** implementing the algorithm above. We'll call this class **Search**.
- **run_Search** uses **open** and **closed**, so these will be variables in **Search**.
- **run_Search** deals with search nodes, so we'll have another class **Search_Node**.
- We'll need to keep track of the **current_node**, so this will be another variable in Search.
- To programme a search in a particular problem domain (e.g. Jugs problems), our 'user' needs to create a subclass of Search for this domain – e.g. **Jugs_Search**.
- *This subclass will have variables carrying the information needed to define a search problem in this domain* – for jugs, the jug capacities and the target amount.
- To run a search, we create an instance of the subclass, giving it the defining information, and call **run_Search**.
- So we will never make an instance of Search itself – it will be an *abstract class.*
- The class **Search_Node** makes the distinction between a node in a search and a problem state clear: the user is concerned with states. The search engine sees only nodes.
- **Search_Node** will have a variable **state** which contains the state associated with this node. Later on, we'll add more variables, e.g. the parent node and the cost of getting to this node.
- **Search_Node** will have methods which express what **run_Search** needs to know:

  is a node a goal (**goalP**)?,

  what are its successor nodes (**get_Successors**)?,

  is its state the same as that of another node (**same_State**)?.. this is needed for dynamic programming.
- **state** will be of type **Search_State**. This will be another abstract class which we specialise for a particular problem domain (e.g. **Jugs_State**).
- **Search_State** specifies what the user needs to implement for her domain – **goalP**, **get_Successors** and **same_State**. The corresponding methods in **Search_Node** will in turn use these user-defined methods

So the (partial) design looks like this:



2.8.2 Search_State.java

Here's the code for Search_State: an abstract class which specifies what sub-classes must provide:

```
/**
* Search_State.java
* State in a state-space search
* abstract class
*  concrete sub-classes must implement
*    goalP, get_Successors, same_State, toString
*/
import java.util.*; //needed in order to use  ArrayLists
public abstract class Search_State {
   /**
   * goalP takes a Search_Node & returns true if it's a goal for the current search
   */
   abstract boolean goalP(Search searcher);
   /** get_Successors returns an ArrayList of states which are successors to
   * the current state in a given search
   */
   abstract ArrayList get_Successors(Search searcher);
   /**
   * same_State: is this state identical to a given one?
   */
   abstract boolean same_State(Search_State n2);
}
```

Note that

- To decide whether a given state is a goal, we will generally need information which is defined for the current search problem - e.g. the target amount for Jugs. This is why **goalP** is given the current instance of **Search.**
- **get_Successors** also needs the **Search** instance, for instance in Jugs we need to know what the capacities are for the current problem.

2.8.3 Jugs_State.java

Now we can specialise **Search_State** for the Jugs domain: We have to choose the internal representation of a state (2 **ints j1** & **j2**) and provide, a constructor, accessors and the methods required by **Search_State**:

```
/** Jugs_State.java
* State in a jugs problem
*/
import java.util.*;
public class Jugs_State extends Search_State{
 private int j1; //content of jug1
 private int j2; //content of jug2
```

The constructor is given the contents of the jugs..

```
  public Jugs_State(int j1c, int j2c){
  j1=j1c;
  j2=j2c;
 }
```

We provide accessors..

```
  public int get_j1() {return j1;};
  public int get_j2() {return j2;};
```

goalP picks up the target amount for the current search & checks against the jug contents. Note that we have to cast the **Search** passed in to its concrete class **Jugs_Search**:

```
 public boolean goalP(Search searcher) {
  Jugs_Search jsearcher = (Jugs_Search) searcher; //cast as Jugs_Search
  int tar=jsearcher.get_Target(); // get target amount
  return ((j1 == tar) || (j2 == tar));
 }
```

Get_Successors is a bit complicated for Jugs...

```
 public ArrayList get_Successors (Search searcher) {
  Jugs_Search jsearcher = (Jugs_Search) searcher;
  int cap1=jsearcher.get_cap1(); //get jug capacities
  int cap2=jsearcher.get_cap2();
  int j1_space=cap1-j1; // work out space in each jug
```

```
  int j2_space=cap2-j2;
  ArrayList slis=new ArrayList(); //the list of successor states

  if (j1_space > 0) slis.add(new Jugs_State(cap1,j2)); //fill jug1
  if (j2_space > 0) slis.add(new Jugs_State(j1,cap2)); //fill jug2
  if (j1 != 0) slis.add(new Jugs_State(0,j2));        //empty j1
  if (j2 != 0) slis.add(new Jugs_State(j1,0));        //empty j2
  if ((j1 != 0) && (j2_space != 0)) {                 //pour from j1 into j2
   if (j1 > j2_space)
   {slis.add(new Jugs_State(j1-j2_space, cap2));} //until j2 is full
    else
   {slis.add(new Jugs_State(0, j1+j2));}; //or j1 is empty
    };
  if ((j2 != 0) && (j1_space != 0)) {                 //similar for pouring  from j2 into j1
   if (j2 > j1_space)
   {slis.add(new Jugs_State(cap1, j2-j1_space));}
    else
   {slis.add(new Jugs_State(j1+j2,0));};
    };
  return slis; //now contains the list of successor states
 }
```

Same_State compares the jug contents for this state with those of another..

```
 public boolean same_State (Search_State s2) {
  Jugs_State js2= (Jugs_State) s2;
  return ((j1==js2.get_j1())&& (j2==js2.get_j2()));
  }
```

Finally we provide a **toString** method..

```
 public String toString () {
  return "Jug State: Jug1-> "+j1+" Jug2-> "+j2;
  }
```

2.8.4 Search_Node.java

Next we look at the **Search_Node** class.

```
import java.util.*;
public  class Search_Node {
```

Any instance of a Search_Node will have a corresponding **Search_State**:

```
 private Search_State state;
 public Search_State get_State(){ //accessor for state
  return state;
 }
```

The **Search_State** will be provided when the node is created:

```
public Search_Node(Search_State s){
  state= (Search_State) s;
}
```

Search_Node has methods **goalP, get_Successors** & **same_State** which are used by **Search**. **goalP** just calls the corresponding method for the node's state:

```
public  boolean goalP(Search searcher){
  return state.goalP(searcher);
}
```

get_Successors gets the list of successor states, then makes a corresponding list of nodes:

```
public ArrayList get_Successors(Search searcher){
  ArrayList slis = state.get_Successors(searcher);
  ArrayList nlis= new ArrayList();

  for (Iterator it = slis.iterator();it.hasNext();){
    Search_State suc_state = (Search_State) it.next();
    Search_Node n = new Search_Node(suc_state);
    nlis.add(n);
  }
  return nlis;
}
```

same_State compares the states in two nodes:

```
public boolean same_State(Search_Node n2){
  return state.same_State(n2.get_State());
}
```

Finally, toString adds more text to toString for the state:

```
public  String toString(){
  return "Node with state " + state.toString();
}
```

2.8.5 Search.java

Now for the search engine…

```
/**
* Search.java - abstract class specialising to Jugs_Search etc
*/
import java.util.*;
import Search_Node;
import Search_State;
import sheffield.*; //for i/o
```

```
public abstract class Search {
```

We need to set up variables for open, closed, the current node etc. These will be protected - available only in instances of Search and its sub-classes:

```
   protected Search_Node init_node; //initial node
   protected Search_Node current_node; // current node
   protected ArrayList open;    //open - list of Search_Nodes
   protected ArrayList closed; //closed - .....
   protected ArrayList successor_nodes; //used in expand & vet_successors
   protected EasyWriter scr; //output to console window
```

The search algorithm is implemented by the **run_Search** method. This takes an initial **search_State** from the user and (for the moment) returns a string indicating success or failure. The first thing to do is create an initial **search_Node** for the initial state:

```
  public  String run_Search (Search_State init_state) {
    init_node= new Search_Node(init_state); // create initial node
```

run_Search will generate a commentary as the search proceeds:

```
    scr=new EasyWriter();
    scr.println("Starting Search");
```

The initial conditions are that **open** contains the initial node and **closed** is empty:

```
open=new ArrayList(); // initial open, closed
open.add(init_node);
closed=new ArrayList();
```

We'll also count how many nodes have been expanded:

```
int cnum = 1;// counts the iterations
```

Now we're ready to start the search. Each iteration of the while loop expands 1 node. We stop either when the search succeeds (within the while) or **open** is empty.

```
while (!open.isEmpty()) {
   // print contents of open
   scr.println("-------------------------");
   scr.println("iteration no "+cnum);
   scr.println("open is");
   for (Iterator it = open.iterator();it.hasNext(); ){
       Search_Node nn = (Search_Node) it.next();
       String nodestr=nn.toString();
       scr.println(nodestr);
   }

   select_Node(); // select_Node selects next node,
                  // makes it current_node & removes it from open
   scr.println("Current node "+current_node.toString());
```

```
   //- is current node a goal? - must pass current search to goalP
   if (current_node.goalP(this)) return "Search Succeeds";  //success, exit
   //current_node not a goal
   expand(); // find successors of current_node and add to open
   closed.add(current_node); // put current node on closed

   cnum=cnum+1;
};  //end of the while controlling the search
return "Search Fails";  // out of the while loop - failure
}
```

Expand has to find the current node's successors and put them on open, unless they've been seen before. A method vet_Successors checks that:

```
 private void expand () {
  // get all successor nodes - as ArrayList of Objects
  // pass search instance to get_Successor of current_node
  successor_nodes = current_node.get_Successors(this);
  vet_Successors(); //filter out unwanted - DP check
  //add surviving nodes to open
  for (Iterator i = successor_nodes.iterator(); i.hasNext();)
    open.add(i.next());
  }
```

vet_Successors removes any node if its state is the same as that of a node already on **open** or **closed**: (the DP check):

```
 private void vet_Successors() {
   ArrayList vslis = new ArrayList();

 for (Iterator i = successor_nodes.iterator(); i.hasNext();){
  Search_Node snode = (Search_Node) i.next();
    if (!(on_Closed(snode)) && !(on_Open(snode))) vslis.add(snode);
 };
 successor_nodes=vslis;
}
```

vet_Successors uses the following:

```
//on_Closed - is the state for a node the same as one on closed?
private boolean on_Closed(Search_Node new_node){
 boolean ans = false;
 for (Iterator ic = closed.iterator(); ic.hasNext();){
   Search_Node closed_node = (Search_Node) ic.next();
   if (new_node.same_State(closed_node)) ans=true;
 }
```

```
 return ans;
}
//on_Open - is the state for a node the same as one on closed?
private boolean on_Open(Search_Node new_node){
 boolean ans = false;
 for (Iterator ic = open.iterator(); ic.hasNext();){
   Search_Node open_node = (Search_Node) ic.next();
   if (new_node.same_State(open_node)) ans=true;
 }
 return ans;
}
```

select_Node embodies the search strategy. For the moment, we take the last node added to **open**. This becomes the **current_node** & is removed from open:

```
private void select_Node() {
 int osize=open.size();
 current_node= (Search_Node) open.get(osize-1); // last node added to open
 open.remove(osize-1); //remove it
}
```

2.8.6 Jugs_Search.java

Now we can specialise Search for Jugs problems. This just sets up the search parameters (jug capacities and target) and provides a constructor and accessors for the parameters.

```
/**
* Jugs_Search.java
* search for jugs problems
*/


import Search;
import Search_Node;
import java.util.*;

public class Jugs_Search extends Search {

   private int cap1; //capacity of jug1
   private int cap2; //..... jug2
   private int target; //target

   public  Jugs_Search (int c1, int c2, int tar) {
    cap1=c1;
```

```
    cap2=c2;
    target=tar;
   }
   public int get_cap1(){
    return cap1;
   }
   public int get_cap2(){
     return cap2;
   }
   public int get_Target(){
     return target;
   }
}
```

2.8.7 Run_Jugs_Search..java

Here's a top-level programme to try all this out:

```
import sheffield.*;
import java.util.*;
import Search;
import Search_Node;
import Search_State;
import Jugs_Search;
import Jugs_State;

public class Run_Jugs_Search {

public static void main(String[] arg) {
   // create an EasyWriter
   EasyWriter screen = new EasyWriter();
   // create the searcher
   Jugs_Search searcher = new Jugs_Search(7,4,2);
   // create the initial state, cast it as Search_State
   Search_State init_state = (Search_State) new Jugs_State(0,0);
   //go!
   String res = searcher.run_Search(init_state);
   screen.println(res);
}
}
```

2.8.8 Some Results

A: Capacities 7,4: Target 2

```
java Run_Jugs_Search

Starting Search
-------------------------
iteration no 1
open is
Node with State Jug State: Jug1-> 0 Jug2-> 0
Current node Node with State Jug State: Jug1-> 0 Jug2-> 0
-------------------------
iteration no 2
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 0 Jug2-> 4
Current node Node with State Jug State: Jug1-> 0 Jug2-> 4
-------------------------
iteration no 3
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 4 Jug2-> 0
Current node Node with State Jug State: Jug1-> 4 Jug2-> 0
-------------------------
iteration no 4
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 4 Jug2-> 4
Current node Node with State Jug State: Jug1-> 4 Jug2-> 4
-------------------------
iteration no 5
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 7 Jug2-> 1
Current node Node with State Jug State: Jug1-> 7 Jug2-> 1
-------------------------
```

iteration no 6
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 0 Jug2-> 1
Current node Node with State Jug State: Jug1-> 0 Jug2-> 1
---
iteration no 7
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 1 Jug2-> 0
Current node Node with State Jug State: Jug1-> 1 Jug2-> 0
---
iteration no 8
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 1 Jug2-> 4
Current node Node with State Jug State: Jug1-> 1 Jug2-> 4
---
iteration no 9
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 5 Jug2-> 0
Current node Node with State Jug State: Jug1-> 5 Jug2-> 0
---
iteration no 10
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 5 Jug2-> 4
Current node Node with State Jug State: Jug1-> 5 Jug2-> 4
---
iteration no 11
open is
Node with State Jug State: Jug1-> 7 Jug2-> 0
Node with State Jug State: Jug1-> 7 Jug2-> 4
Node with State Jug State: Jug1-> 7 Jug2-> 2
Current node Node with State Jug State: Jug1-> 7 Jug2-> 2

Search Succeeds

Process Run_Jugs_Search finished

B: Capacities 4,3: Target 2

java Run_Jugs_Search

Starting Search

---

iteration no 1

open is

Node with State Jug State: Jug1-> 0 Jug2-> 0

Current node Node with State Jug State: Jug1-> 0 Jug2-> 0

---

iteration no 2

open is

Node with State Jug State: Jug1-> 4 Jug2-> 0

Node with State Jug State: Jug1-> 0 Jug2-> 3

Current node Node with State Jug State: Jug1-> 0 Jug2-> 3

---

iteration no 3

open is

Node with State Jug State: Jug1-> 4 Jug2-> 0

Node with State Jug State: Jug1-> 4 Jug2-> 3

Node with State Jug State: Jug1-> 3 Jug2-> 0

Current node Node with State Jug State: Jug1-> 3 Jug2-> 0

---

iteration no 4

open is

Node with State Jug State: Jug1-> 4 Jug2-> 0

Node with State Jug State: Jug1-> 4 Jug2-> 3

Node with State Jug State: Jug1-> 3 Jug2-> 3

Current node Node with State Jug State: Jug1-> 3 Jug2-> 3

---

iteration no 5

open is

Node with State Jug State: Jug1-> 4 Jug2-> 0

Node with State Jug State: Jug1-> 4 Jug2-> 3

Node with State Jug State: Jug1-> 4 Jug2-> 2

Current node Node with State Jug State: Jug1-> 4 Jug2-> 2

```
Search Succeeds

Process Run_Jugs_Search finished

C: Capacities 4,3: Target 5
java Run_Jugs_Search

Starting Search
-------------------------
iteration no 1
open is
Node with State Jug State: Jug1-> 0 Jug2-> 0
Current node Node with State Jug State: Jug1-> 0 Jug2-> 0
-------------------------
iteration no 2
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 0 Jug2-> 3
Current node Node with State Jug State: Jug1-> 0 Jug2-> 3
-------------------------
iteration no 3
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 3 Jug2-> 0
Current node Node with State Jug State: Jug1-> 3 Jug2-> 0
-------------------------
iteration no 4
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 3 Jug2-> 3
Current node Node with State Jug State: Jug1-> 3 Jug2-> 3
-------------------------
iteration no 5
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 4 Jug2-> 2
Current node Node with State Jug State: Jug1-> 4 Jug2-> 2
```

```
------------------------
iteration no 6
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 0 Jug2-> 2
Current node Node with State Jug State: Jug1-> 0 Jug2-> 2
------------------------
iteration no 7
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 2 Jug2-> 0
Current node Node with State Jug State: Jug1-> 2 Jug2-> 0
------------------------
iteration no 8
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 2 Jug2-> 3
Current node Node with State Jug State: Jug1-> 2 Jug2-> 3
------------------------
iteration no 9
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 4 Jug2-> 1
Current node Node with State Jug State: Jug1-> 4 Jug2-> 1
------------------------
iteration no 10
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 0 Jug2-> 1
Current node Node with State Jug State: Jug1-> 0 Jug2-> 1
------------------------
iteration no 11
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 1 Jug2-> 0
```

```
Current node Node with State Jug State: Jug1-> 1 Jug2-> 0
----------
iteration no 12
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Node with State Jug State: Jug1-> 1 Jug2-> 3
Current node Node with State Jug State: Jug1-> 1 Jug2-> 3
----------
iteration no 13
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Node with State Jug State: Jug1-> 4 Jug2-> 3
Current node Node with State Jug State: Jug1-> 4 Jug2-> 3
----------
iteration no 14
open is
Node with State Jug State: Jug1-> 4 Jug2-> 0
Current node Node with State Jug State: Jug1-> 4 Jug2-> 0
Search Fails
Process Run_Jugs_Search finished
```

2.9 Implementing State-Space Search for a new Problem Domain

This is a summary of what we have to do for a new domain:

- Define a subclass of Search_State which

  Represents a problem state in whatever way we choose.

  Provides a constructor

  Provides accessors for the state representation

  Implements **goalP, get_Successors** and **same_State** as specified.
- Define a subclass of Search with variables for the search parameters and a constructor.

We then initiate a search by

Creating an instance of our search class

Calling run_Search for this instance, giving it the initial state.

Note that

- The algorithm terminates as soon as the ***first goal node is selected from open.***
- It doesn't (yet) **return a solution path**, just an indication of success or failure.

2.10 Reconstructing the Solution path

If it is necessary to reconstruct the solution path (as in the map-traversal problem, for instance), *we must keep a record of how we reached each node that we close: together with each node we remember its parent node.* To do this:

- A **parent** variable is added to the **Search-Node** class. For the start node, there will be no parent: in Java, its parent will have value **null** (i.e. not initialised).
- The **expand** method in **Search** is modified so that when new nodes are added to open their **parent** is set to **current_node**.

When a search succeeds we need to reconstruct the solution path. Starting with the **current_node** and working with **closed,** we follow the chain of **parents** until we come to a node with parent **null.** We need to print out or return the states on the solution path in order from the start node. Here is a method of Search to do this: it is called when the search succeeds and returns the solution path as a string. It also reports the efficiency of the search:

```
private String report_Success(){
 Search_Node n = current_node;
 StringBuffer buf = new StringBuffer(n.toString());
 int plen=1;
 while (n.get_Parent() != null){
   buf.insert(0,"\n");
   n=n.get_Parent();
   buf.insert(0,n.toString());
   plen=plen+1;
  }

 scr.println("========================== \n");
 scr.println("Search Succeeds");

scr.println("Efficiency "+ ((float) plen/(closed.size()+1)));
scr.println("Solution Path\n");
return buf.toString();
 }
```

For example, with capacities 7 & 4 and target 2 we get:

```
Search Succeeds
Efficiency 1.0
Solution Path
node with state Jug State: Jug1-> 0 Jug2-> 0
node with state Jug State: Jug1-> 0 Jug2-> 4
node with state Jug State: Jug1-> 4 Jug2-> 0
node with state Jug State: Jug1-> 4 Jug2-> 4
node with state Jug State: Jug1-> 7 Jug2-> 1
```

```
node with state Jug State: Jug1-> 0 Jug2-> 1
node with state Jug State: Jug1-> 1 Jug2-> 0
node with state Jug State: Jug1-> 1 Jug2-> 4
node with state Jug State: Jug1-> 5 Jug2-> 0
node with state Jug State: Jug1-> 5 Jug2-> 4
node with state Jug State: Jug1-> 7 Jug2-> 2
```

2.11 Elementary Search Strategies

How do we choose which node on open to expand next?

2.11.1 Depth-First Search

The **select_node** method of **Search** which we've been using up to now:

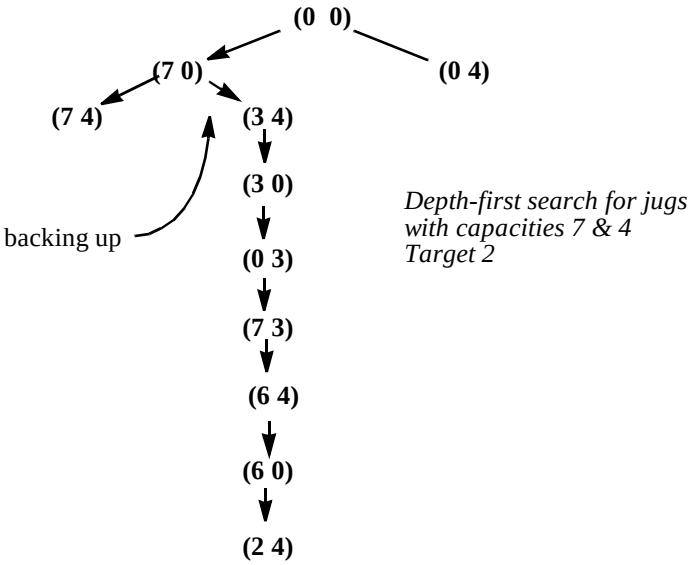
```
private void select_Node() {
 int osize=open.size();
 current_node= (Search_Node) open.get(osize-1); // last node added to open
 open.remove(osize-1); //remove it
}
```

Implements **depth-first search**:

Each iteration expands one of the successors of the node expanded on the previous iteration, until a goal is reached or a dead end (node with no successors) is found. In this case, search will resume at the last level in the search tree where there is an alternative. This is sometimes called *Backing Up*.

open behaves as a *stack*.. last in, first out.

Depth-first is guaranteed to find a solution, but not admissible.



2.11.2 Breadth-First Search

Expands all nodes at level **l** before moving to level **l+1**.

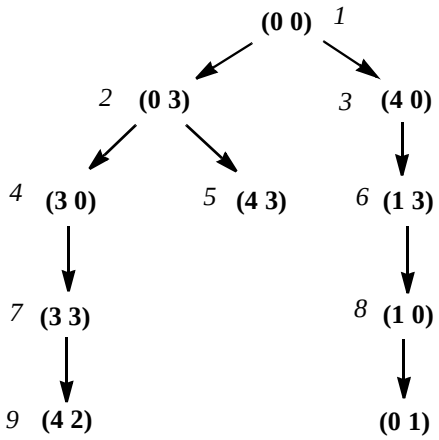
select_node selects that node which has been on open the longest.

If nodes are added to the head of **open**, then select_node should take the last node.

open behaves as a *Queue*..first in, first out.

Guaranteed to find a solution.

Admissible in case of uniform costs.



Breadth-first jugs search
Capacities 3 and 4
Target 2

Nodes closed in numerical order

The following implements breadth-first search:

```
private void breadth_first(){
 current_node= (Search_Node) open.get(0); //first node on open
 open.remove(0);
}
```

2.11.3 Compromises

- **Depth-Bounds**: Search depth-first to some depth-limit **N**. Enforce backup from this level, but keep a list **l** of the nodes found at level **N**. If **open** becomes empty (i.e. searched completely to depth **N** without finding a solution), increase **N**, set **open** to **l**, and resume.
- **Beam Search:** Like breadth-first, but only expand from the best few nodes at each level. Assumes we have some way of deciding which they are.

Generally, we want to allow our user to *choose which search strategy* to use, by naming the **select_node** function to use in the call of **run_Search**: this is the next part of the project.

2.11.4 Adding these improvements

The code in

www.dcs.shef.ac.uk/~pdg/com1080/java/search2

reconstructs the solution path, returning it as a string, and allows the user to choose either a breadth-first or a depth-first search. The strategy is specified as an additional argument in the call to **run_Search**:

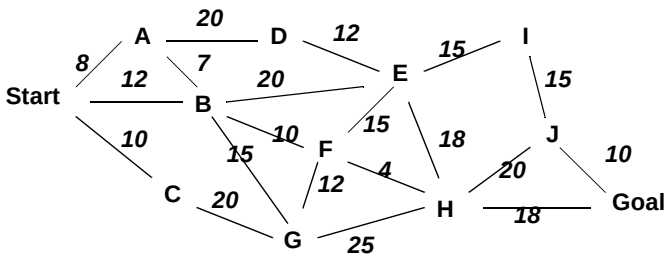
```
Jugs_Search searcher = new Jugs_Search(7,4,2);
Search_State init_state = (Search_State) new Jugs_State(0,0);
```

```
String resb = searcher.run_Search(init_state, "breadth_first");
screen.println(resb);
String resd = searcher.run_Search(init_state, "depth_first");
screen.println(resd);
```

Now **select_node** chooses which of **depth_first** and **breadth_first** to call by comparing with the given string. The default is **breath_first**. The old **select_node** is renamed **depth_first**.

2.12 Branch-and-Bound search

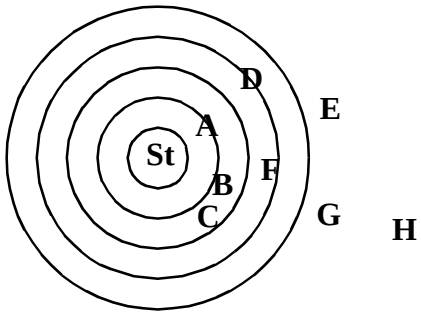
We now turn to the more general case of search, where each move has a *variable cost* associated with it. Our exemplar is map-traversal:



Branch-and-Bound is a generalisation of breadth-first.

Whereas breadth-fist chooses the node at *minimum depth* into the search tree from Start, branch-and-bound chooses the node with *minimum accumulated cost* from Start.

Branch-and-Bound explores on contours of increasing cost from Start:



Circles represent cost contours at 5, 10,15..(approx)

select-node chooses that node from **open** with *minimum cost from start.*

Reduces to Breadth-First when costs are uniform.

Branch-and-Bound is *admissible*. Since it explores on contours of increasing cost, the goal node chosen from **open** must be that one at minimum cost from Start.

This is why we don't stop the search as soon as a goal node appears during expand.

2.12.1 Implementing Branch-and-Bound Search

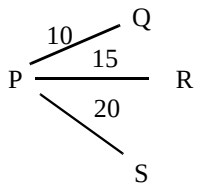
Code implementing map search is in

www.dcs.shef.ac.uk/~pdg/com1080/java/search3

The user can choose breadth-first, depth-first or branch-and-bound search.

In search3,

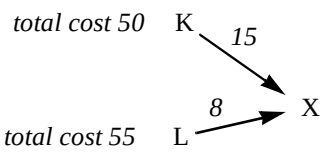
- The class **Carta** implements a map[1]. A map is represented by a **HashSet** of city names and an **ArrayList** of **MapLink**s. An instance of the class MapLink represents an individual link between 2 cities, and its cost.
- Carta has methods to read a map from a file (**mapFromFile**[2]), find all the cities in a given map (**findcities**), get the cost between 2 cities connected by a link (**costbetween**) and find all the links to a given city (**get_Links**).
- For variable-cost problems, **Search_State** needs a variable **localCost** for the cost of reaching a state from its parent in the search tree (the state in the **current_Node**). The subclass of **Search_State** for map-traversal problems is **Map_State.** The **get_Successors** method of **Map_State** now has to fill in these **localCost**s for each successor.



get_successors P

should return **Map_State**s with
city Q localCost 10
city R localCost 15
city S localCost 20

- **Search_Node** needs variables **localCost** (copied from its **state**) and **globalCost** (the total cost from the start - 0 for the node representing the initial state)
- **globalCost** can be set either in **get_Successors** for **Search_Node** or in **expand** in **Search**[3]. It is the sum of the node's **localCost** and **current_node**'s **globalCost.**
- It is necessary to implement the **dynamic programming principle**: that we only need to remember the best (i.e. minimum cost) route to a given state, e.g.



Here there are two routes to X
The route through K will be opened first
When the route through L is found,
we should forget about the one through K

- When we find a successor which has the same **state** as one we've already seen, but not yet closed (i.e. there is a **search_node N** with this **state** on **open**) , we need to compare the **globalCost** of this **search_node** (cost of the old route, **c**) with **c'**.
- This can be done within **vet_successors:** For any successor **Search_Node N** with **globalCost**t **c**,

1. It's not called **Map** to avoid confusion with the **Map** interface
2. The map we're using as an example is in **map1.txt**
3. The latter is coded in **search3**.

If **x** is not on **open** or **closed**, add a **search_node** for it to **open.**

If **x** is on **closed**, ignore it (*exercise: why is this safe?*).

If **x** is on **open,** then

if **c <= c'**, ignore **x** (*old route cheaper*)

if **c > c'**, (*new route cheaper)*

change the **globalCost** of **N** to **c'**

change the **parent** of **N** to the **current_node**

- The **branch_and_bound select-method** then chooses that node on open with minimum **cost.**
- The class **Run_Map_Search** tries all this out on the example map.

2.12.2 Performance of Branch-and-Bound

A branch-and-bound search for the best path from **St** to **Gl** in map1 should give something like:

```
OPEN-LIST
state parent cost
st      nil     0
expanding st
-----------------------------
OPEN-LIST
state parent cost
c       st      10
b       st      12
a       st      8
expanding a
-----------------------------
OPEN-LIST
state parent cost
d       a       28
c       st      10
b       st      12
expanding c
-----------------------------
OPEN-LIST
state parent cost
g       c       30
b       st      12
d       a       28
expanding b
-----------------------------
found better route to g
```

OPEN-LIST

state parent cost

f b 22

e b 32

d a 28

g b 27

expanding f

OPEN-LIST

state parent cost

h f 26

g b 27

d a 28

e b 32

expanding h

OPEN-LIST

state parent cost

gl h 44

j h 46

g b 27

d a 28

e b 32

expanding g

OPEN-LIST

state parent cost

d a 28

e b 32

gl h 44

j h 46

expanding d

OPEN-LIST

state parent cost

e b 32

gl h 44

j h 46

expanding e

OPEN-LIST

```
state parent cost
i       e       47
gl      h       44
j       h       46
expanding gl
----------------------------
Solution is (st b f h gl)
Cost 44
Effficiency 0.5
```

The efficiency of 0.5 exemplifies the problem with branch-and-bound: it is **undirected** and so will waste effort on paths which are getting no nearer to the goal. Here half the **states** expanded aren't on the eventual solution path...and we're using an unrealistic map which only has links going roughly in the St-->Gl direction.

If we have no more knowledge to deploy, Branch-and Bound is the best algorithm to use if we want to preserve admissibility.

However, there may be other knowledge to use...

2.13 Best-First Search

To improve on the efficiency of Branch-and-Bound, we need some knowledge about the *topology of the state space...are we going in the right direction?*

Suppose we have available **estimates of the remaining cost from any node n to a goal node,** i.e. a **Search_State** has a variable **est_rem_cost** which the user fills in in her **get_Successors**.

This is reasonable in many search problems e.g. in map traversal, if the map is a road map, the estimates could be 'as the crow flies' distances between cities.

Exercise: It's difficult to think of a way of getting estimates for jugs problems, but what about the 8-puzzle? or equaliser?

Best-first search selects from **open** that node with minimum **estRemCost**, i.e. the node that seems closest to the goal.

This is great when the estimates are good, but dangerous when they are not: it is **not admissible.**

2.14 The A* Algorithm

A combination of Branch-and-Bound and Best-First.

Suppose we have estimates of remaining costs as above.

A* selects that node **n** from **open** with **minimum estimated total cost of a path from start to goal through n.**

This estimate is obtained by adding the **globalCost** (branch-and-bound) and the **estRemCost** (best-first).

A* is admissible provided all the **estRemCosts** are guaranteed to be underestimates.

Informal Proof: The known distance along a complete, non-optimal path cannot be less than an underestimate of the distance along the incomplete, optimal path. Hence the wrong path cannot be selected from **open**.

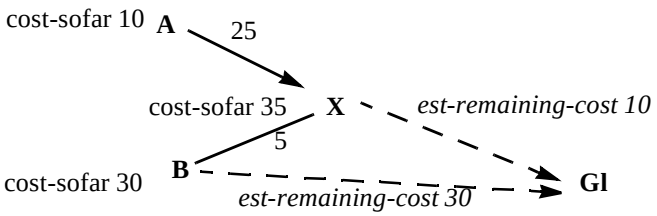
Limiting cases:

- If all the estimates are accurate, **A*** is optimal.
- If all the estimates are 0 (or identical), **A*** reduces to Branch-and-Bound.
- The better the estimates are, the more efficient **A*** is.

2.14.1 Converting branch-and-bound to A*

Changes to the search engine:

- The user supplies **estRemCost**s for each **Search_State**
- **Search_Node** also has an **estRemCost** variable, copied from the state, and an **estTotalCost** which is **globalCost+estRemCost.**
- The **A* select_node** function chooses from **open** using this field.
- **vet_successors** has to change because it's now possible (if we have a serious underestimate of remaining cost) to find a better route to a node which is already on **closed.**



X will be placed on open with parent A and estimated total cost 45.
This will be preferred to B, (estimated total cost 60).
X will therefore be closed.
Later on, the better route through to X through B will emerge.

- When this happens (which should be rarely if the estimates are good), the node (**X** in the example above) should have its **parent, globalCost** and **estRemCost** changed, then it should be taken off **closed** and put back on **open**.

Code for the A* algorithm is in

/share/public/com1080/java/search4

2.15 Things to try

In

/share/public/com1080/java/search_usa

is a map of the USA you can use to get a better feel for more realistic search problems. Using the search4 code, explore the behaviour of the different search strategies in this map. How much gain in efficiency for A* versus branch-and-bound?

2.16 Search in Automatic Speech Recognition

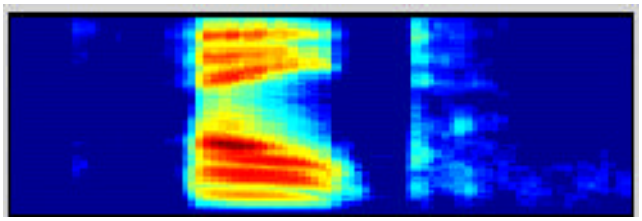
An example of state-space searching in a larger scale application.

Automatic Speech Recognition (ASR) aims to provide voice input to computers.

Consider the simplified ASR problem of **isolated word recognition**, with a small vocabulary (e.g. voice telephone dialling). Given a new word, which of the words we know about is it most likely to be? We outline the **Dynamic Time Warping (DTW)** method.

- Information in speech is represented by the distribution of energy through time and frequency. Conventionally, this is portrayed in a **spectrogram:**

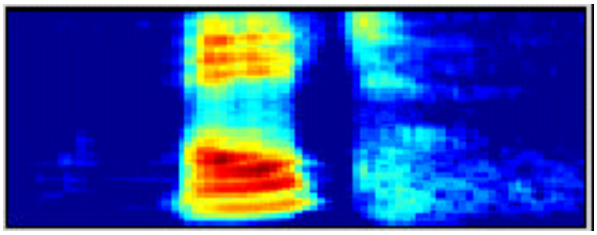
Frequency



Time

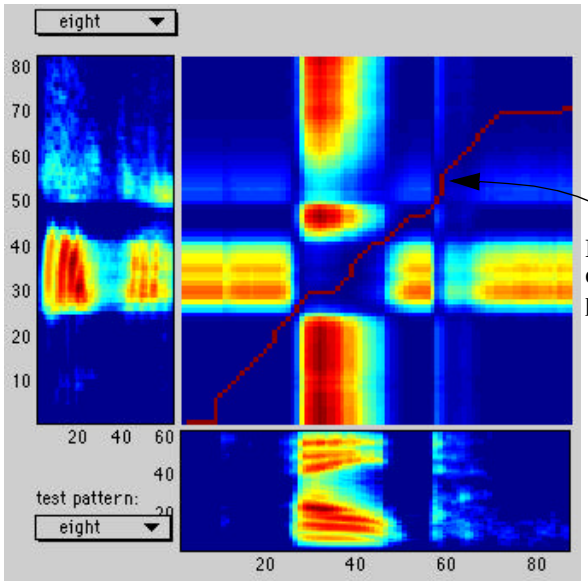
Spectrogram for a spoken 'eight'

- Think of the spectrogram as an image in which each pixel codes the energy at that frequency at that time. We might typically have around 100 time frames & 64 frequency channels.
- In DTW, we store an example of the spectrogram for each word in the vocabulary. These are called **templates**. We thus have a **template library.**
- Along comes a new word - the **pattern**. Here's another 'eight':

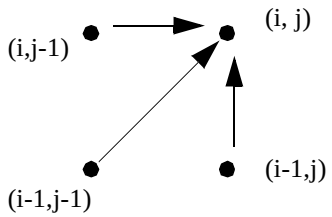


- We need a way of comparing the pattern with each template and picking the closest match.
- The problem is that two different utterances of the same word will vary in length, and the time-scale distortions will not be linear.
- We can compute a **local distance measure** for the similarity of a single time-frame of data in the pattern with a frame in the template.

- These distance measures are plotted in a ‘time-time’ plane below:



- What we need is the **minimum-cost path** from bottom-left to top-right in the time-time plane - the total cost is the sum of the local costs of the cells on the path.
- We can impose constraints on path movements:



- If we have the total cost to each of the predecessors, we can select the minimum, to find the best route to (i,j) and add the local cost at (i,j) to get its total cost.
- This is just the **Dynamic Programming Principle** again.
- We do this for each template and select that one with minimum total cost.
- DTW only works well only for small vocabularies of isolated words by a single speaker.
- For more difficult recognition problems we need to compare the incoming data with statistical models rather than templates, but the search principle is the same.

3 Game Playing

3.1 Why Program Computers to Play Games?

Because

- Skilled game-playing seems to involve important problem-solving abilities: **look-ahead, planning, strategy, judgement**.
- Games are **well-defined**, providing a tractable micro-world.
- Games provide an **opportunity to test** programs against each other, and against humans.
- There's money in it (in recent years).
- It's fun.

3.2 What kind of Games?

We will only consider

- **2-opponent** games
- with **no chance** element
- where **both players have complete-knowledge** of the state of the game

e.g. tic-tac-toe (noughts & crosses), drafts, chess, go, reversi....

There is also a body of work on other types of game, e.g. bridge, poker and back-gammon. In the latter, the world champion was beaten by a machine in about 1980.

3.3 Formulating Game Playing as State-Space Search

As before, our problem will be completely expressed by an internal structure called a **state**, e.g. the positions of the pieces on the board & a note of who is to move. A state completely defines the game position.

From a given state $\mathbf{S_1}$, the **rules of the game** will allow us to move to one of a number of successor states $\mathbf{S_{11}}$, $\mathbf{S_{12}}$**...**

We begin in some initial state **Si**. We have to find a **route through state space that leads us to a goal state**. In game-playing a **goal state represents a win** (e.g. row/ column of 3 Xs, checkmate...).

3.4 Game Trees

In game playing by state-space search, we consider our moves from a given state, the opponent's possible replies to each of these moves, our possible replies to his/ her replies etc. The resulting exploration of state-space is called a **game tree**. The exploration process is called **look-ahead**.

Terminal nodes (leaves) in the game tree represent board positions where the **result of the game is known for certain, i.e. win, lose or (if allowed) draw**.

The fundamental components of a game-playing program will be

- A way of **representing board positions** (states),

- A '**legal move generator**' which embodies the laws of the game; given a board, return a list of new boards - like **get_successors** in the search code.
- A means of **detecting terminal board positions**; given a board, return win/lose/ draw/unknown.
- A program which **explores the game tree**.

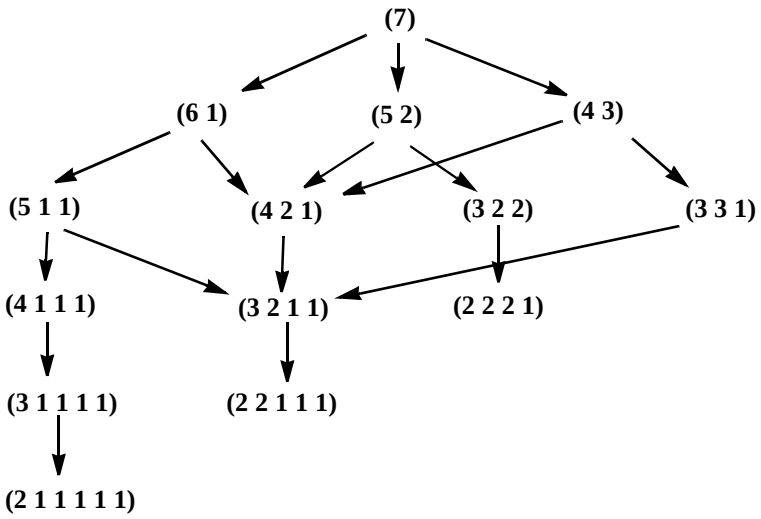
3.5 MiniMaxing

Suppose we take a game which is simple enough for us to grow the **complete** game tree (complete look-ahead), for instance

The Game of Piles (ako Grundy's game).

Start with 1 pile of tokens. A move consists of taking a pile and dividing it into 2 unequal piles. The last player who is able to move wins

Here is the state space if we start with 7 tokens:



We can decide on the move to make as follows:

- Label the terminal nodes as ***win*** if the player to move next will win, ***loss*** if the player to move next will lose. In piles they will all be labelled ***loss***.

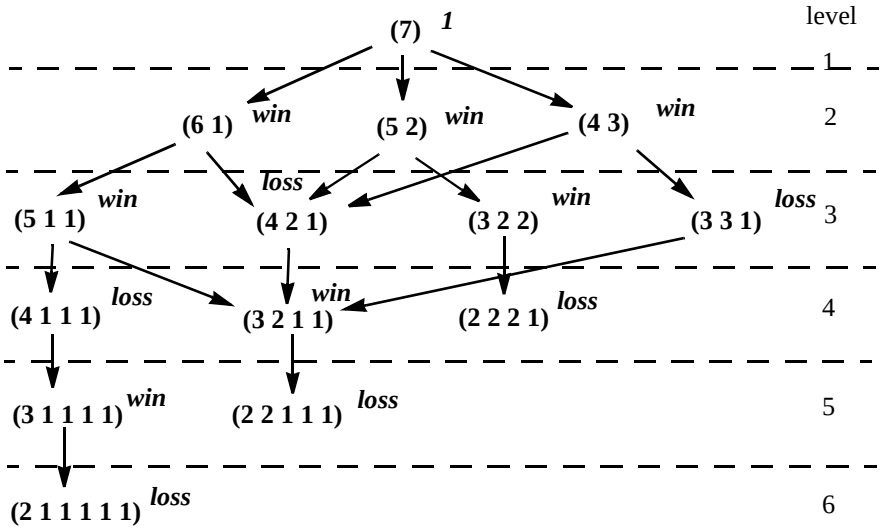
Backtrack up the tree starting from the level above the terminals:

- If all the successors of a node are labelled ***win*** then label this node ***loss.***
- Otherwise label it ***win***.

Terminate when the root has been labelled.

If it has been labelled ***win*** then find a successor of the root labelled ***loss*** and play that.

So in this case (see over) the original board position is a losing position - the second player can force a win.



3.6 Java Implementation

Growing a game tree is similar to the search processes we have already seen. At the end we have to do the backtracking rather than report success.

Again we can write a domain-independent engine which can be specialised for particular games. implements this. This can be based on the **search2** code - we don't need variable costs. The major changes will be

Abstract class **GameSearch** can be adapted from the **Search** class:

> method **run_GameSearch**, derived from **run_Search,** grows a tree of **GameSearch_Nodes**, either breadth-first or depth-first.
>
> Instead of **goalP** we have **resultP,** which tells us whether a node is a terminal. If it is, we don't look to expand it, but put it on **closed** and select the next node.
>
> The search terminates when **open** is empty - we search the whole tree.
>
> When this happens **run_GameSearch** calls a method **minimax** - see below.

Class **GameSearch_Node** is adapted from **Search_Node:**

> It now has additional variables **children** (the successor nodes)**, level** (the depth of this node in the tree) and a String **outcome** (whether this node represents a **"win"** or **"loss"** for the player to move, or whether the outcome is **"unknown".**
>
> It has a **resultP** method rather than a **goalP** method, which calls the **result** method of the subclass of **GameSearch_State**. This returns **"win"**, **"loss"** or **"unknown"**. **resultP** returns **true** for **"win"** or **"loss"**.
>
> Its **same_state** method includes a check that the 2 nodes being compared are at the same level (the same board at different levels implies different outcomes).

Abstract class **GameSearch_State** is adapted from **Search_State** & just specifies that its subclasses must have methods **result**, **get_Successors** and **same_state.**

The **minimax** method of **GameSearch** works as follows:

1. Work through the tree -i.e. through closed - filling up the children slots, i.e. for each node *n* (except the root) with parent *p*, add *n* to *p*'s children (alternatively you can do this within **expand**).
2. Work back up the tree through the levels, starting at the maximum depth - 1.
3. For each node at the current level that hasn't already been labelled, label it **"win"** unless all its children are **"win"**.
4. When the root (the single node at level 1) has been labelled, return its label and, if it's a **"win"**, the first node in its children labelled **"loss"** - this is the move to make.

3.7 Evaluation Functions.

Because of the combinatorial explosion, in practice we can only afford to explore a small part of the complete game tree, and we can't expect to reach the terminal nodes.

If we can't search to terminal positions, we have to have a way of **assessing the positions we can reach**.

If we call the 2 players **MAX** and **MIN.**

An **evaluation function E(b)** takes a board position **b** and returns a number which expresses the merit of the position: the larger **E(b**), the more the position is judged as favouring the player **MAX**, the smaller it is the better for **MIN**.

A popular form for E(b) is a weighted polynomial:

$$E(b) = w_1 f_1(b) + w_2 f_2(b) + ...$$

Where

$\mathbf{f_1}$, $\mathbf{f_2}$ ... are functions which measure **features of the board position** which seem to be important, e.g. piece advantage, control of centre, mobility...

$\mathbf{w_1}$, $\mathbf{w_2}$... are 'weights' which express the **relative importance** of the $\mathbf{f_i}$.

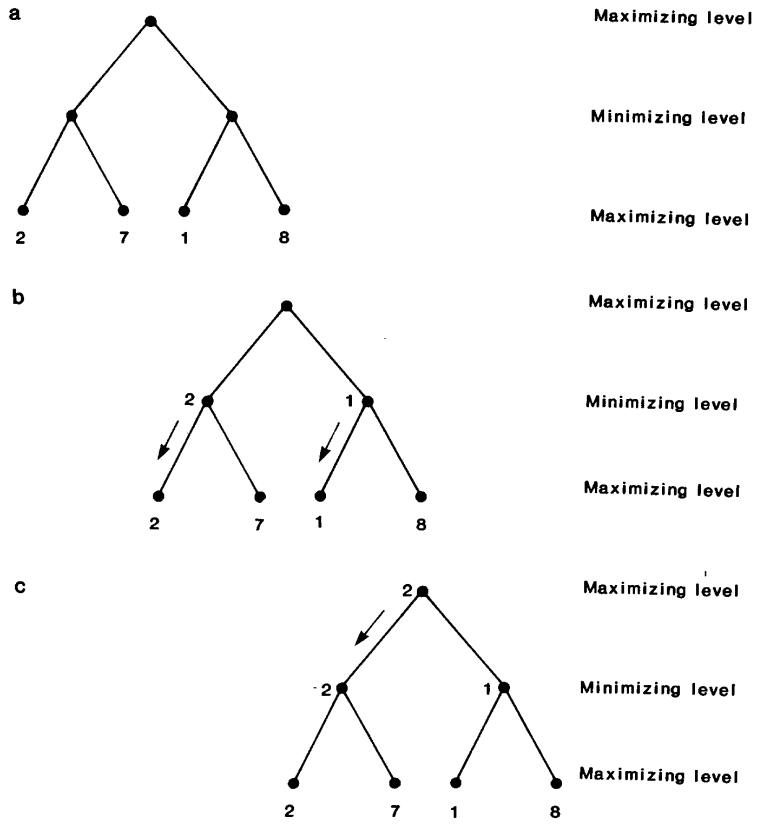
NOTE:

1. It is arguable whether it is really legitimate to reduce a board position to a single number.
2. **E(b)** is a **static** evaluation function. It does not try to take account of the dynamics of a game, e.g. a threat building up.
3. **E(b)** is likely to be **more accurate for simpler positions**, e.g. when there are few pieces around. Its unlikely that **E(b)** will say much about the first move in chess, for instance.
4. A polynomial form assumes that $\mathbf{f_1}$, $\mathbf{f_2}$ ... are essentially **independent** pieces of evidence. This may be unjustified.
5. There will be a trade-off between how far we can search and how complex **E(b)** is, because it must be computed for each leaf node.

3.8 Partial Look-ahead

Given an evaluation function we can proceed as illustrated below:

- Grow the tree as far as we can (typically to a fixed depth or '**ply**').
- Compute **v = E(b)** for each leaf node.



- '**Traceback' or 'back-up' or 'backtrack**' through the tree as follows:
- A node at a **maximising level** is given a backed-up value **v'** which is the **maximum value of its children.**
- A node at a **minimising level** is given a backed-up value **v'** which is the **minimum value of its children.**
- Eventually the root node will have backed-up values for all its children. If we are **MAX**, we make the move with maximum **v'**, vice-versa if we are **MIN**.

NOTE:

1. **We could have applied E(b) to the original children of the given board position** and used these immediate estimates to decide our move. The argument for look-ahead and backtrack is that E(b) should be able to produce **more accurate estimates later on**, when board positions are 'simpler'.
2. There is an implicit assumption that **the opponent is 'thinking in a similar way to us**', i.e. has a similar evaluation function, and always chooses on this basis.

3.9 ALPHA-BETA Pruning.

There is a way of **conducting a minimax backtrack without backing up all the nodes**.

For each **MAX** node we keep a '**provisional-backed-up-value**', ? , whose value may rise but will never decrease.

i.e. we may find another child which is better for max, but any worse case can be discarded.

For each **MIN** node we keep a **provisional-backed-up-value,** ?**,** whose value may fall but will never increase.

The **pbvs** become **vs** when all children have been explored.

This is another version of the dynamic programming principle: forget everything except the best (or worst) path so far found.

The procedure is illustrated below:

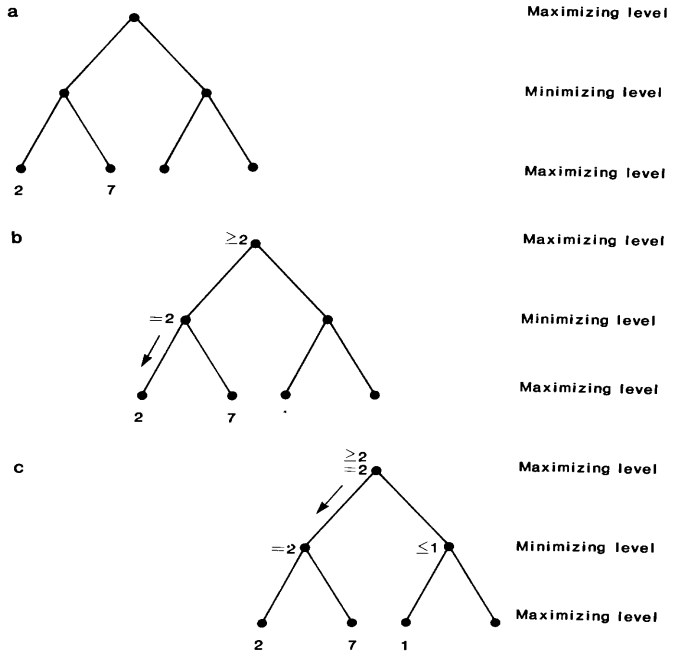


Figure 4-17. Work can be avoided by augmenting the MINIMAX procedure with the ALPHA-BETA procedure. In this illustration, there is no need to explore the right side of the tree fully, because there is no way the result could alter the move decision. Once movement to the right is shown to be worse than movement to the left, there is no need to see how much worse.

- Work from **left to right**, starting by diving down to the **depth limit**.
- Discontinue work below any node which won't be chosen by its parent.

Here is a larger example: only the circled leaf nodes get evaluated:

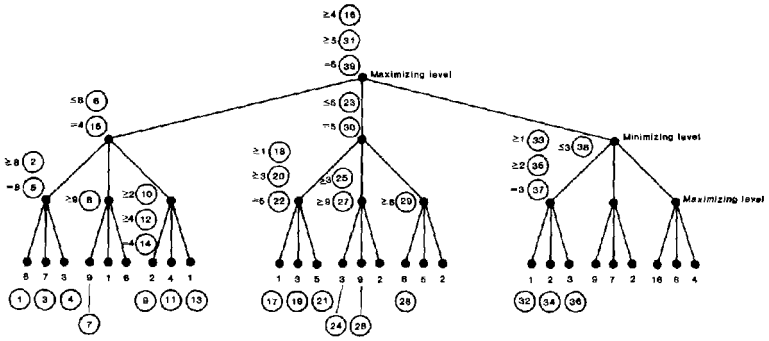


Figure 4-18. A game tree of depth 3 and branching factor 3. The circled numbers show the order in which conclusions are drawn. Note that only 16 static evaluations are made, not the 27 required without alpha-beta pruning. Evidently the best play for the maximizer is down the middle branch.

with circled event numbers placed beside each conclusion showing the order in which they are determined. These are shown in the example of figure 4-18, in which we look at another stylized tree with a depth of 3 and a uniform branching factor of 3:

1–2: Moving down the left branch at every decision point, the search penetrates to the bottom where a static value of 8 is unearthed. This 8 clearly means that the maximizer is guaranteed a score at least as good as 8 with the three choices available. A note to this effect is placed by step 2.

3–5: To be sure nothing better than 8 can be found, the maximizer examines the two other moves available to it. Since 7 and 3 both indicate inferior moves, the maximizer concludes that the score achievable is exactly 8 and that the correct move is the first one examined.

6: Nailing down the maximizer's score at the lowest node enables a conclusion about what the minimizer can hope for at the next level up. Since one move is now known to lead to a situation that gives the maximizer a score of 8, then the minimizer can do no worse than 8 here.

7–8: To see if the minimizer can do better at the second level, its two remaining moves must be examined. The first leads to a situation from which the maximizer can score at least a 9. Here cutoff occurs. By taking the left branch, the minimizer forces a score of 8, but by taking the middle branch, the minimizer will do no better than 9 and could do worse if the other maximizer choices are even bigger. Hence the middle branch is bad for the minimizer, there is no need to go on to find out how bad it is, and there is consequently no need for two static evaluations. There is no change in the minimizer's worst-case expectation; it is still 8.

9–14: The minimizer must still investigate its last option, the one to the right. This in turn means seeing what the maximizer can do there. The next series of steps bounces between static evaluations and conclusions about the maximizer's situation immediately above them. The conclusion is that the maximizer's score is 4.

15: Discovering that the right branch leads to a forced score of 4, the minimizer would take the right branch, since 4 compares favorably with 8, the previous low score.

16: Now a bound can be placed at the top level. The maximizer, surveying the situation there, sees that its left branch leads to a score of 4. It now knows it will get at least that. To see if it can do better, it must look at its middle and right branches.

17–22: Deciding how the minimizer will react at the end of the middle branch requires knowing what happens along the left branch descending from there. Here the maximizer is in action discovering that the best play is to a position with a score of 5.

23: Until something definite was known about what the maximizer could do, no bounds could be placed on the minimizer's potential. Knowing that the maximizer gets 5 along the left branch, however, is knowing something definite. The conclusion is that the minimizer can get a score at least as good as 5.

24–27: In working out what the maximizer can do below the minimizer's middle branch, it is discovered partway through the analysis that the maximizer can reach a score of 9. But 9 is poor relative to the known fact that the minimizer has one option that ensures a 5. Cutoff occurs again. There is no point in investigating the other maximizer option, thus avoiding one static evaluation.

28–29: Looking at the minimizer's right branch quickly shows that it, too, gives the maximizer a chance to force the play to a worse score than the minimizer can achieve along the left branch. Cutoff saves two static evaluations here.

30: Since there are no more branches to investigate, the minimizer's score of 5 is no longer a bound; 5 is the actual value achievable.

31: The maximizer at the top, seeing a better deal through the middle branch, chooses it tentatively and knows now that it can do at least as good as 5.

32–37: Now the maximizer's right-branch choice at the top must be explored. Diving into the tree, bouncing about a bit, leads to the conclusion that the minimizer sees a left-branch choice ensuring a score of 3.

38: The minimizer can conclude that the left-branch score is a bound on how well it can do.

39: Knowing the minimizer can force play to a situation with a score of 3, the maximizer at the top level concludes there is no point in exploring the right branch further. After all, a score of 5 follows a middle-branch move. Note that this saves not only six static evaluations but also two

NOTE:

1. Alpha-beta is **guaranteed to return the same value** as a full minimax.
2. Alpha-beta is an example of the '**dynamic programming principle**' - in a state-space search you only need to remember the best route to a given node so far found.
3. Alpha-beta may **postpone the combinatorial explosion**, **but it doesn't remove it**: in the best case the number of static evaluations, s, needed to find the best move is given by

$s = 2b^{(d/2)} - 1$ if d is even

$= b^{(d+1)/2} + b^{(d-1)/2} - 1$ if d is odd.

Where **d** is the search depth and the tree has uniform branching factor **b**.

This is illustrated below for **d=3** and **b=3.**

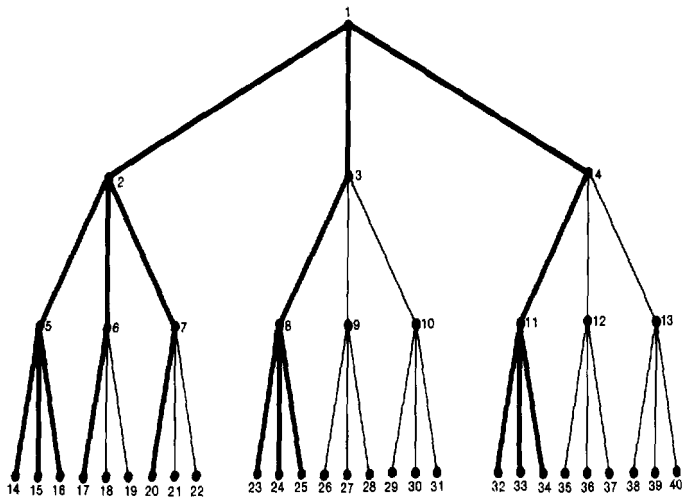


Figure 4-19. In a perfectly ordered tree, ALPHA-BETA cuts the exponent of combinatorial explosion in half. This is because all of the adversary's options need not be considered in verifying the left-branch choices. With depth 3 and branching factor 3, ALPHA-BETA can reduce the number of required static evaluations from 27 to 11.

The following compares full minimax with best-case alpha-beta:

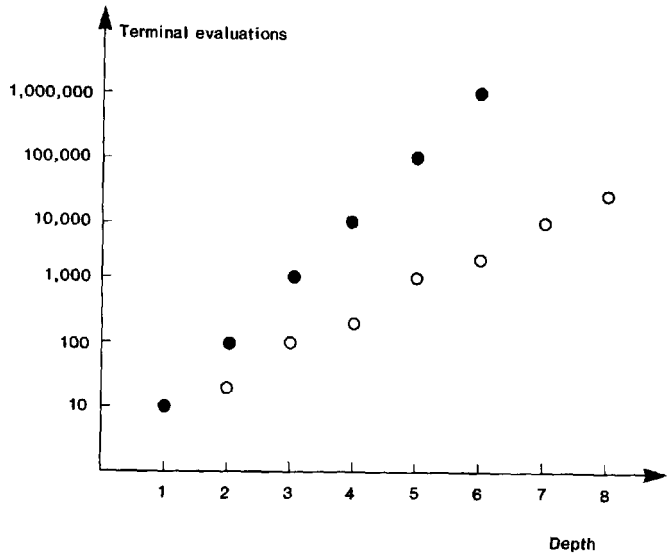


Figure 4-20. ALPHA-BETA reduces the rate of combinatorial explosion, but does not prevent it. In this illustration, the larger number of terminal evaluations is for no alpha-beta pruning; the smaller number is for the maximum possible alpha-beta pruning. In both cases, the branching factor is assumed to be 10.

3.10 Adaptations of Brute-Force Search

3.10.1 Heuristic Pruning

Compared to skilled human play, a brute-force search spends most of its time considering 'rubbish' - positions a good player would never bother with. In **heuristic pruning**, moves from each node are ordered by their **plausibility**, and search effort is expended according to that ranking. Low plausibility moves are never considered. The tree takes on a '**tapered**' look (see below).

Note that **we may make a mistake** - this is a heuristic method, not guaranteed.

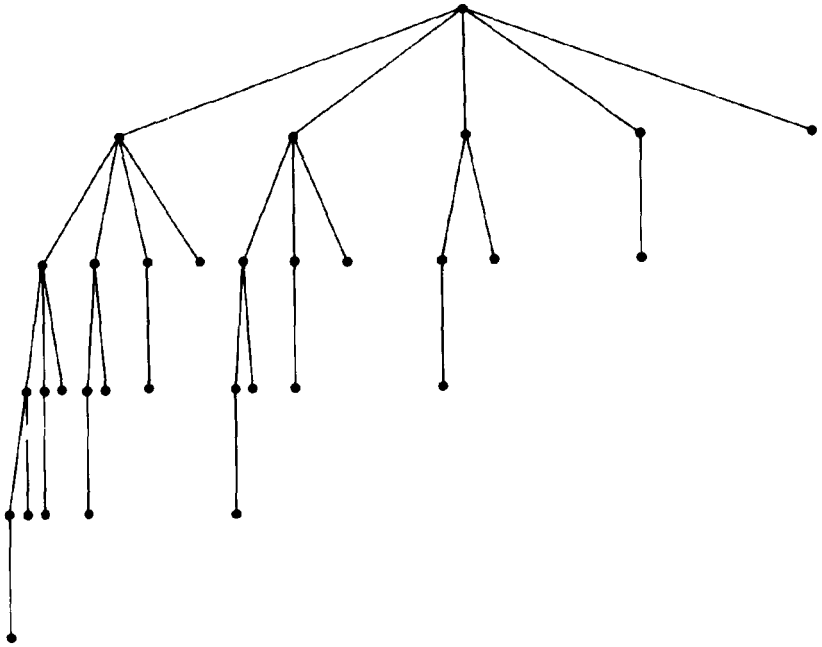


Figure 4-21. Tapering search gives more attention to the more plausible moves. Here the tapering procedure reduces the search with increases in depth and decreases in plausibility.

The general name for this sort of technique is '**beam search**'.

3.10.2 Book Moves

Serious computer game-playing programs make much use of **libraries of standard positions**, with sometimes millions of entries. If a node represents a board position which is in the library, we have effectively **already done the search below that node.**

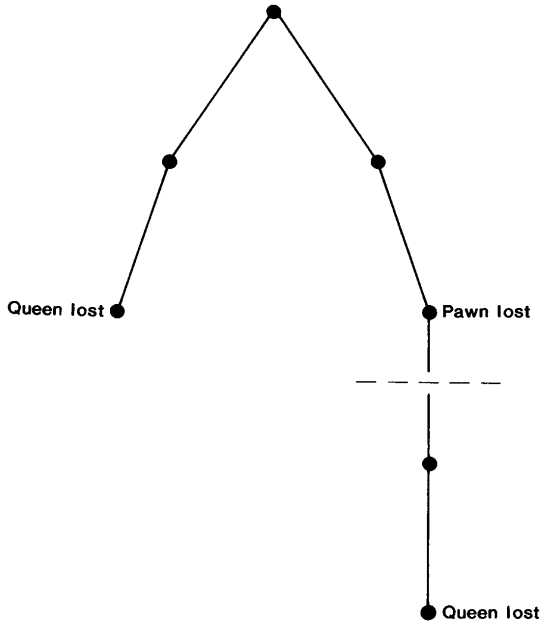
A more powerful version of this would suggest **whole sequences of moves to follow** until play diverts from a standard pattern - e.g. classic chess openings.

The difficult problem is to **recognise not when a situation is identical to one previously met, but which has a' similar pattern**', differing only in unimportant respects. Skilled human players are very good at this **intelligent pattern matching**. This ability is (in my opinion) closely-linked to human powers of perception. It has

been shown that expert chess players can memorise and recall board positions more readily than novices.

3.10.3 Combating the Horizon Effect.

If we search to a fixed depth, we can wind up choosing moves which **delay but do not prevent disasters**, e.g.



The disaster is '**beyond the horizon**'. The program can seem to be throwing material away, only postponing the inevitable.

One way of reducing the horizon effect is to **continue search past situations which are considered to be dynamic**, e.g. check, imminent loss of a piece, pawn about to queen.

Another idea is to back-up and chose a prospective move **m**, then grow a small, secondary tree below the position **p** which **m** is expected to lead too. **If the backed-up value for p is much worse than E(p), think again.**

3.10.4 Learning

In his classic work on **Checkers** or draughts (1959-67), **Samuels** attempted to get a program to learn to play better by adjusting both the features $\mathbf{f_i(b)}$ and their weights $\mathbf{w_i}$ in **E(b)**.

The idea was to **reward a win by increasing the w_i for those f_i which contributed to the good moves, and punish a defeat in a similar way**. If some $\mathbf{w_i}$ became very low, the corresponding $\mathbf{f_i}$ was removed from **E(b)** and replaced by a new feature chosen from a pool.

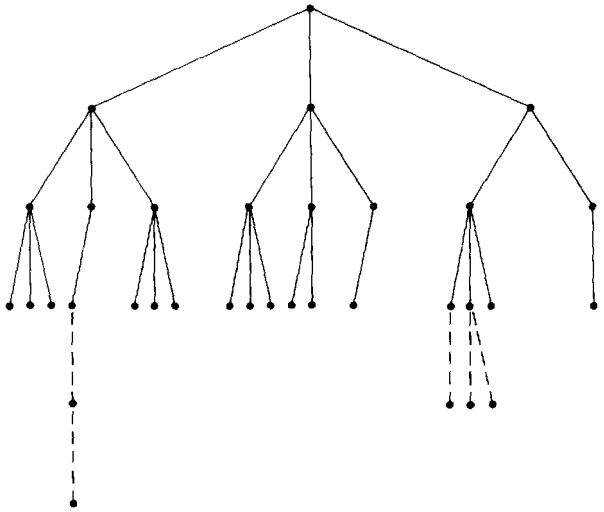


Figure 4-23. Heuristic continuation, or feedover, extends search beyond normal limits. Dotted lines are continuations of paths that ordinary search had terminated at particularly dynamic situations such as *imminent piece loss*.

The learning regime was to have **2 computer players A and B play each other**. **A** used a **fixed E(b)**, the best found so far. **B** starts with this polynomial, perhaps with some random adjustments. **B**'s polynomial is **modified by punishment-and-reward**, as above. If **B** consistently beats **A**, **B**'s polynomial is moved to **A**.

Samuels achieved some success with this learning technique. His program eventually reached world-class performance (but most games are drawn).

In general the problem is to devise a learning technique which will converge - which is guaranteed to find an optimum solution. I don't know of any such 'training theorems' for game-playing.

3.11 Comparison of Human and Machine Play

3.11.1 Strategy

A program using search techniques as described above would **consider the situation afresh on each move**. It has no long-term plan or strategy. This is initially disconcerting to a human opponent - the machine 'jumps around' rather than pursuing some logical line of play.

3.11.2 Adaptability

Good game players vary their play according to who (or what) the opponent is. They know the opponents strengths and weaknesses. The dumb machine would make exactly the same response every time it was in a particular position.

Commonly the human struggles when first playing a particular program, but soon learns to beat it. David Levy (who issued a challenge in 1968 that no computer would beat him for 10 years) advocates making unusual opening moves so that the program can't 'use its book'.

3.11.3 Search in Human Play

Overt search does play a part in expert-level play, but it often has the **secondary role** of checking out a promising move that has been identified in some other way. The search is very selective - very 'depth-first'.

3.11.4 Expert Systems for Game Playing

Search-based techniques don't 'understand' the problem in any human sense. They cannot, for instance, provide explanations. The only 'knowledge' is in the legal move generator and the evaluation function, where it's implicit rather than explicit.

Can **human knowledge about how to play a game** be captured in a program? **Michie** (1980) reported work on an 'expert system' for chess end-games in which he was able to demonstrate theoretically-correct play for some classic problems. Knowledge-based systems are coming later...

3.11.5 Really Brute Force

Computer Chess has developed into a world of its own, and is no longer seen as central to AI. Modern programs, still search-based but with specialist hardware and massive libraries, now play at international master level. Programmes are widely available which can beat all but the best few hundred humans. Kasparov was beaten by IBM's 'deep blue' in 1996. The following (from the San Francisco Chronicle) summarises the match..

Speed, Not Artificial Intelligence, Is How IBM Won

By DAVID EINSTEIN

1997 San Francisco Chronicle

For all its historic significance, Deep Blue's victory over world chess champion Garry Kasparov wasn't really much of a scientific breakthrough, computer experts said Monday. In fact, the consensus was that the 6-foot-5, 1.4-ton IBM computer depended less on artificial intelligence than on raw computing power to dispatch Kasparov in their six-game series that finished Sunday.

``The Deep Blue team used heavy computing to make up for a lack of cleverness in the algorithms themselves,'' said John McCarthy, professor of computer science at Stanford University.

It may not have been quite that simple. IBM worked hard to improve Deep Blue's chances this year, giving it a better grasp of chess openings and end games than it had in 1996, when Kasparov won the first meeting.

IBM also hired chess grandmaster Joel Benjamin as a consultant this year. ``They sat down with the grandmaster and fine-tuned the knowledge of how the computer judged some of the positions,'' said David Wilkins, a senior computer scientist (and chess expert) at SRI in Menlo Park. ``So it did have more and better chess knowledge this time.''

In the end, however, he said the real difference probably was Deep Blue's calculating speed, which was doubled this year. It could analyze 200 million possible moves a second -- 50 billion board positions in the three minutes given for each move. That kind of power, said Wilkins, ``will get you to human champion performance.'' Deep Blue employed searching and logic techniques based in early artificial intelligence theory. But research into A.I. has now gone far beyond chess-playing computers. Today, to have true artificial intelligence, a machine must be able to respond to outside stimulus such as light, sound and movement.

If Deep Blue were able to actually ``look'' at the chessboard and decide how to play, it might qualify. Right now, though, it's just a powerful computer whose real talent is to wear out its opponent. Kasparov proved to be a case in point. After winning the first game of the match, the champ surrendered in game two, even though experts said he could have gained a draw. The next three games ended in draws, and in the finale on Sunday, Kasparov was practically a basket case, botching his opening and losing in just 19 moves.

History will record this as the first time man was beaten by a machine, but that's not quite right. ``I don't think this is a story about mano a silicio,'' James Bailey, author of ``After Thought: The Computer Challenge to Human Intelligence,'' wrote in Harper's magazine recently. ``It's about a bunch of guys at IBM who by themselves had no chance of ever getting into an international chess tournament, and therefore chose to collaborate with a computer.

``The computer by itself also didn't have a chance of making it into an international chess game. But together they were able to go where neither of them could go alone.''

The real winner in the match, of course, was IBM itself, which owns and operates Deep Blue the way George C. Scott managed Paul Newman in ``The Hustler.''

IBM gave the rematch a lot of hype, which entailed a risk, because if Deep Blue had lost badly -- worse than last year -- all the buildup would have been for nothing. With Deep Blue's triumph, however, IBM has polished its image as a maker of fabulously fast computers.

4 Pattern Matching and Knowledge-based Systems

Problem solving techniques in symbolic AI which go beyond blind searching depend on the use of **knowledge** about the **problem domain**.

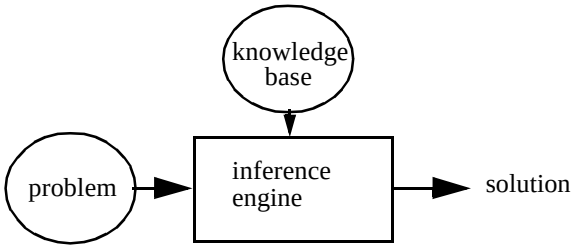
This knowledge must somehow be **matched** with the current problem-state: *What do we know which applies to the situation we are in?*

Internally, we will be comparing data structures representing the problem state with data structures which represent the knowledge. We need a **knowledge representation scheme** (sometimes called an **ontology**).

There are various knowledge representation schemes but they all involve **patterns** and their use is based on **pattern matching.**

This is the methodology of **expert systems.** The knowledge is confined to some restricted **problem domain.**

In an **expert system shell**, the idea is to provide a general mechanism for this kind of problem-solving, which the user can then adapt to her domain (like our search engine):



4.1 Programming a pattern matcher

Fundamental to pattern matching is the idea of 'wild cards'. You may be familiar with this from the operating systems MS-DOS and/or Unix:

```
chortle{pdg}41: cd /home/pdg/public/com1080/java/search1/simplejava
chortle{pdg}42: ls
Coordinates.class      SimpleReader2.class      SimpleCanvas.class
SimpleReaderException.class      SimpleGraphicsWindow.class
SimpleWriter.class                SimpleReader.class
chortle{pdg}43: ls Simp?eCan?as.class
SimpleCanvas.class
chortle{pdg}44: ls Simple*.class
SimpleCanvas.class                SimpleReader2.class
SimpleGraphicsWindow.class    SimpleReaderException.class
SimpleReader.class                SimpleWriter.class
chortle{pdg}45: ls C*.*
Coordinates.class
```

So the wild cards are **?**, which is allowed to match with any single character and *, which matches against one or more characters. They can be used alone or in combination. We only need to think about the equivalent of **?.**

In general, we can think of the wild cards as **matching variables**, and pattern matching means *'find a suitable set of substitutions for matching variables which makes one pattern identical to another'.* Another name for pattern matching is **unification**.

The Java API doesn't supply pattern matching as part of the language, so we need to provide our own facilities. The languages you'll learn at level 2, Haskell and Prolog, have pattern matching as a central feature. In Lisp, pattern matching is not part of the language itself, but there are packages for it. Lisp data structures (lists) are well-suited to pattern matching.

Code for the matching classes we need is available in

/share/public/com1080/java/pmatch

Which can be imported as a package **pmatch**.

We'll represent our patterns by **Strings**, using single spaces as delimiters between elements in the pattern, so for instance if we are matching against **"colour apple red"** we should get

Pattern to match	**Result**
"colour apple red"	true
"colour apple green"	false
"colour apple red size small"	false
"colour apple ?"	true
"colour orange ?"	false
"? apple ?"	true

4.1.1 Bindings for pattern variables

When a match is successful, it is useful to remember **what the wild cards matched against**. We'll do this by giving the wild cards names that start with a **?**, e.g. **?col**. A successful match establishes **bindings** for the pattern variables. The set of bindings thus created is called a **context:**

Pattern to match	**Result**	**Context**
"colour apple red"	true	null
"colour apple green"	false	null
"colour apple ?col"	true	?col <= red
"colour orange ?col"	false	null
"?prop apple ?val"	true	?prop <= colour ?val <=red

Where <= means 'is bound to'.

4.1.2 The MString class

We need a class **MString** to implement pattern matching between **Strings.** We can't implement this in the obvious way, as a subclass of **String**, because **String** is a 'final' class - it can't be subclassed. Instead we make it a subclass of **Object** whose constructor takes a **String** as argument.

We use the **StringTokeniser** class to split this string, and the string we are matching against, into words separated by delimiters e.g. spaces, returns. You can then iterate over the tokens. For instance

```
StringTokenizer st = new StringTokenizer("this is a test");
  while (st.hasMoreTokens()) {
    println(st.nextToken());
  }
```

Prints

```
this
is
a
test
```

We also need to decide on how we are going to represent **contexts**. A context is a **table** whose **items** associate variables (**keys**) with values (**data**). Java provides a **HashMap** class which allows us to create tables, add items to them (method **put**), retrieve items (**get**) etc.

We'll allow pattern variables to appear in either (or both) patterns **p1** and **p2** we are matching. To simplify things,

- Each pattern variable can only appear once,
- We can't have variables in the same position in **p1** and **p2** (e.g. **p1 = "size apple ?s" , p2 = "size apple ?t"**).

We should really check that these conditions hold, but to save time we won't.

Here's the start of **MString.java**:

```
public class MString extends Object {
 private String str; //the string to be matched against
 public String getStr() {return str;}; //accessor

 private HashMap context; //the matching context
 public HashMap getContext() {return context;} //accessor

//constructor
  public MString(String s){ str=s;}

 // match against given string
 public boolean match(String d){
```

```
   StringTokenizer ptok = new StringTokenizer(str); //tokenise the MString
   StringTokenizer dtok=new StringTokenizer(d); //& the string it will match
against
   boolean result; // the final result - success or failure

   if (ptok.countTokens() != dtok.countTokens())
    result=false; //number of tokens must be equal
   else {
    result=true;
    context=new HashMap();
    while (ptok.hasMoreTokens()&& result){
     String nextp=ptok.nextToken();
     String nextd=dtok.nextToken();

     if (!nextp.equals(nextd)){
      if (pvar(nextp))
       context.put(nextp,nextd);
      else {
       if (pvar(nextd))
        context.put(nextd,nextp);
       else result=false;
      }
     }
    }
   }

   return result;
 }
```

pvar here is a predicate telling us whether a string is a pattern variable:

```
 private boolean pvar(String v){return(v.startsWith("?"));}
```

We can now use **MString** like so:

```
MString n1=new MString("one two three four");
boolean n1res=n1.match("one ?too ?free" four);
```

will return **true** and

```
  screen.println(n1.getContext());
```

Will print

```
{?too=two, ?free=three}
```

4.1.3 Additions to MString

Matching with an existing context

Often we want to do matches in sequence, using contexts from earlier matches:

```
MString m1 = new MString("I want to go from ?start to ?dest");
m1res=m1.match("I want to go from Sheffield to London");
MString m2=new MString("The train from Sheffield to London leaves at 0930");
m2res=m2.match("The train from ?start to ?dest leaves at ?time", m1.getCon-
text())
```

... will match & set the context of **m2** to

```
{?start=Sheffield, ?dest=London, ?time =0930}
```

This is done by

- Defining a second constructor for MString which takes an additional argument, the initial context.
- Whenever a pattern variable appears in the match, we look in the context to see if it has a binding. If so, we use this binding in the match.

Substituting back

The method **msubst** is the inverse of matching: given an **MString** and a context, return the **MString** with pattern variables in the context replaced by their bindings:

```
MString m3=new MString("Return from ?dest to ?start");
String sres=m3.msubst(m1.getContext());
```

returns

```
Return from London to Sheffield
```

Forward and Backward Contexts

It is sometimes important to be able to distinguish between matches for variables in the given MString instance and matches for variables in the String we are matching against. The method **match_2_way** performs a match as before but returns two contexts, one for the mvars in the mString instance which is called (**pcon**) and one for the mvars in the String it's matched against (**dcon)**. e.g.

```
MString m4=new MString("?x weight heavy")
m4.match_2_way("feather weight ?y")
m4.getPcon ==>{"?x","feather"}
m4.getDcon ==>{"?y","heavy"}
```

We'll use this in backward-chaining later on. In this case we are allowed to have mvars in the same place, and **pcon** takes precedence, e.g.

```
MString m5=new MString("?x parent of Fred")
m5.match_2_way("?z parent of  ?y")
m5.getPcon ==>{"?x","?z"}
m5.getDcon ==>{"?y","Fred"}
```

4.1.4 MStringVector.java: matching against a Vector of Strings

In the 'train' example above we'd want to try matching our pattern

"The train from ?start to ?dest leaves at ?time",

in the context

{?start=Sheffield, ?dest=London}

against a whole train timetable, searching through the timetable until we find something that matches. Rather than a single pattern we need to match against each pattern in a **Vector** of patterns, until we find one that matches (we'll stop at the first match found) or the **Vector** runs out . This facility is provided by the **MStringVector** class.

MStringVector extends **Vector**. Like **MString** it has

- a **match** method (two versions, with and without an initial context),
- an **msubst** method,
- a variable **context** which is updated by a successful match.

Their behaviour is illustrated below:

```
MStringVector v = new MStringVector("PDG phone 21828| MPC phone 21822
|GJB phone 21817"); //in the constructor, | separates MStrings
boolean res=v.match("MPC phone ?m");
```

returns **true** & **v.getContext()** returns **{m=21822}**

```
MStringVector w = new MStringVector("Robbie at ?x|Suzie at ?y");
boolean res = w.match("Suzie at home")
```

returns **true** & **w.getContext()** returns **{?y home}**

```
HashMap con=new HashMap();
con.put("?x work");
con.put("?y play");
Vector res=w.msubst(con);
```

returns a vector with elements **"Robbie at work"**, **"Suzie at play".**

4.1.5 Additions to MStringVector

Sometimes we want to find all the matches for a given String against a given MStringVector, not just one. The **matchall** method does this. As usual, there are 2 versions, with & without an initial context. If it succeeds, **matchall** sets up an ArrayList of the details of individual matches, which is accessed with **getMatchDetails**. Each item in this list is an instance of the **MatchDetails** class, and contains the pattern which matched, the rest of the MStringVector and the matching context.

4.2 Other Matching Facilities

We won't need the following in our programming but it's worth discussing them

4.2.1 Restricting the match

We may wish to put restrictions on what is allowed to match against a variable, for instance by specifying a predicate that must return **true** if the binding is allowed.

e.g. we may want **"arriving at ?x"** to match against **"arriving at Sheffield"** but not against **"arriving at 0930"**. We need to define a predicate which must hold if the match is to succeed. This is natural in a functional language because you can pass functions as arguments. In Haskell you'd say something like

```
match ["arriving", "at", ("?x", townP)]
      ["arriving", "at", "Sheffield"]
```

where **townP** is a predicate returning **True** if its argument is a member of a list of town names.

4.2.2 Matching sequences

We may want variables which are allowed to match against sequences of forms, rather than a single form. These are conventionally introduced with a ***:

e.g. matching **"*pre arrives at *post"** against **"the train arrives at Sheffield at 1100"** would succed & bind

{*pre=the train,*post=Sheffield at 1100}

4.2.3 Duplicating matching variables

We may want variables which occur more than once in the same pattern. In that case, the first appearance establishes the binding, the remainder have to bind to the same:

e.g. matching **"go from ?x to ?y and back to ?x"**

would succeed against **"go from Sheffield to London and back to Sheffield"**

but not against **"go from Sheffield to London and back to York".**

5 Rule-based systems

Here the knowledge is expressed in rules. This is the most common type of expert system.

A rule looks if

If a_1 and a_2 and a_3.... then c_1 and c_2 and....

- The $\mathbf{a_i}$ are the **antecedents**.
- The $\mathbf{c_j}$ are the **consequents**.
- If all the $\mathbf{a_i}$ are met we can **deduce** all the $\mathbf{c_j}$.
- We can restrict ourselves to rules with a single consequent, without loss of generality (if there are n consequents we can have n rules, each with the same antecedents).

Rule-based systems create new knowledge from old by performing **deduction** rather than **induction** or **abduction** (the other 'modes of inference').

Deductive systems are related to formal logic, and generally implement parts of first-order predicate calculus. They usually do not have all of PC's expressive power. More about this in year 2.

The antecedents and consequents are **patterns** containing **pattern variables**, for instance we might have a rule

Grandfather rule

Antecedents: **(?x is a grandparent of ?y)**

(?x is male)

Consequent: **(?x is a grandfather of ?y)**

In LISP the lists would be written as above. In Java we are going to use strings, but for simplicity we can drop that for now.

- Note that a **fact** is a special case of a rule - one with no antecedents.
- The pattern variables play much the same role as **quantifiers** (*for all, there exists*) in logic.
- This kind of formalism is the basis of the programming language **Prolog**.

5.1 Forward and Backward Chaining

The inference engine can do its deduction in two ways. Suppose in addition to the grandfather rule we have

Father-is-parent rule

Antecedent: (?x is the father of ?y)

Consequent: (?x is a parent of ?y)

Father-is-male rule

Antecedent: (?x is the father of ?y)

Consequent (?x is male)

Grandparent rule

Antecedents: (?x is the parent of ?y)

(?y is the parent of ?z)

Consequent: (?x is a grandparent of ?z)

5.1.1 Forward Chaining

Moves from antecedents to consequents:

If we are given the fact

(henry7 is the father of henry8)

The antecedents of the **father-is-parent rule** are triggered and the rule 'fires', deducing

(henry7 is a parent of henry8)

and the **father-is-male rule** adds the deduction

(henry7 is male)

now if we are given

(henry8 is the father of Elizabeth)

The parent rule adds

(henry8 is a parent of Elizabeth)

and the antecedents of the grandparent rule are matched giving

(henry7 is a grandparent of Elizabeth)

finally the grandfather rule fires giving

(henry7 is a grandfather of Elizabeth)

- Forward chaining is **data-driven.**
- Rules used in this way are sometimes called **demons.**

5.1.2 Backward Chaining

Moves from consequents to antecedents. Look for a rule that will allow us to make the deduction we want, then try to prove its antecedents.

Suppose we want to find a grandfather of Elizabeth & we are given the same facts to begin with.

We look for a rule whose consequent matches the pattern (our 'hypothesis').

(?x is a grandfather of Elizabeth)

& come up with the grandfather rule. We therefore try to prove its antecedents, given what we know already:

(?x is a grandparent of Elizabeth)

(?x is male)

We check to see if any antecedents match directly with a fact. If not, we look for rules whose consequents match. The grandparent rule leads us to look for

(?x is the parent of ?y)

(?y is the parent of Elizabeth)

Taking the second of these first, and using the father-is-parent rule we find that we can satisfy this using the fact

(henry8 is the father of Elizabeth)

This gives us a value for **?y.** We now look for

(?x is the parent of henry8)

and find a match through **?x -->henry7.**

We're now back in the grandfather rule having deduced

(henry7 is a grandparent of Elizabeth)

Finally we prove, using the father-is-male rule that

(henry7 is male)

NOTE

- Backward chaining is **hypothesis-driven.**
- We are performing a *blind search* of the *space of possible deductions*. We may get a *combinatorial explosion*. We may have to *backtrack.*
- The deduction process is inherently *recursive*: forward chaining stops when no more deductions can be made. Backward chaining stops when we can match against a known fact, or when there are no rules whose consequents match what we are trying to prove.
- Substitutions must be correctly retained through the stages of a proof.
- Both forward and backward chaining are forms of *monatonic reasoning.* A fact always remains true: once asserted, it is never withdrawn.

5.2 Rule-based systems in Java

see

www.dcs.shef.ac.uk/~pdg/java/FChain

www.dcs.shef.ac.uk/~pdg/java/BChain

For both forward and backward chaining we need a class **Rule** to represent individual rules and **RuleSet** to represent a collection of Rules. **Rule** has variables

```
private Vector antes;
private String conseq;
```

and corresponding accessors. **RuleSet** has

```
private ArrayList rules;
```

the following (in **TestFChain**) creates a small **RuleSet**:

```
Vector gfantes= new Vector();
gfantes.add("?gf father of ?p");
gfantes.add("?p parent of ?c");
Rule gfrule = new Rule (gfantes, "?gf grandfather of ?c");

Vector fpantes = new Vector();
fpantes.add("?f father of ?c");
Rule fprule = new Rule (fpantes, "?f parent of ?c");
```

```
ArrayList rset = new ArrayList();
rset.add(fprule);
rset.add(gfrule);

RuleSet rs = new RuleSet(rset);
```

5.2.1 Forward Chaining in Java

The class **FChain** implements forward chaining through a method **run_FC.** When we create an instance of **FChain** (in **TestFChain**) we give it the **RuleSet** to use.

```
FChain fc=new FChain(rset);
```

We call **run_FC** giving it the initial facts:

```
Vector facts = new Vector();
facts.add("H7 father of H8");
facts.add("H8 father of E");

Vector res=fc.run_FC(facts);
```

run_FC makes all the deductions it can & returns a Vector of the given facts and the deduced facts.

run_FC works as follows:

Keep a record of all the known_facts and the ones which have not yet been 'pursued' i.e. we haven't yet tried to make further deductions on the basis of them. Initially **known_facts** and **to_pursue** are both set to the given facts.

Iterate the following until **to_pursue** is empty:

Remove the first fact **f** from **to_pursue** and make all possible deductions from it:

Iterate the following over all the rules in the **RuleSet**

For each rule **r,** find all the matches for **f** against its antecedants (using **matchall**).

Each success constitues a **partial match,** in which there will be a matching context and (in general) more antecedants to be satisfied. **matchall** supplies this information by returning a list of **MatchDetails** instances.

Iterate the following until the list of partial matches is empty:

Remove the first partial match **p** and develop it:

If **p** has no more antecedants, we have a deduction:

Substitute **p**'s context in **r**'s consequent (using **msubst**) and add this new fact to **known_facts** and **to_pursue.**

otherwise,

Take the first of **p**'s antecedants and use **matchall** to find all the matches for it against **known_facts**, given **p**'s context.

Each successful match forms a new partial.

Results

kf= [H7 father of H8, H8 father of E]

tp= [H7 father of H8, H8 father of E]

pursuing H7 father of H8

match for rule with antes [?f father of ?c]

Deduced H7 parent of H8

match for rule with antes [?gf father of ?p, ?p parent of ?c]

matching ?p parent of ?c

in context {?p=H8, ?gf=H7}

result = false

---

kf= [H7 father of H8, H8 father of E, H7 parent of H8]

tp= [H8 father of E, H7 parent of H8]

pursuing H8 father of E

match for rule with antes [?f father of ?c]

Deduced H8 parent of E

match for rule with antes [?gf father of ?p, ?p parent of ?c]

matching ?p parent of ?c

in context {?p=E, ?gf=H8}

result = false

---

kf= [H7 father of H8, H8 father of E, H7 parent of H8, H8 parent of E]

tp= [H7 parent of H8, H8 parent of E]

pursuing H7 parent of H8

match for rule with antes [?gf father of ?p, ?p parent of ?c]

matching ?gf father of ?p

in context {?p=H7, ?c=H8}

result = false

---

kf= [H7 father of H8, H8 father of E, H7 parent of H8, H8 parent of E]

tp= [H8 parent of E]

pursuing H8 parent of E

match for rule with antes [?gf father of ?p, ?p parent of ?c]

matching ?gf father of ?p

in context {?p=H8, ?c=E}

result = true

Deduced H7 grandfather of E

---

kf= [H7 father of H8, H8 father of E, H7 parent of H8, H8 parent of E, H7 grandfather of E]

tp= [H7 grandfather of E]

```
pursuing H7 grandfather of E
Result: [H7 father of H8, H8 father of E, H7 parent of H8, H8 parent of E, H7 grandfather of E]
```

5.2.2 Backward Chaining in Java

We'll implement backward chaining as a search technique: we are searching the space defined by all the links between the antecedants of one rule and the consequents of another rule.

We could have done this for forward chaining too, rather than implementing it directly.

Since we have uniform costs we'll make use of the **Search2** code.

We need a subclass of Search for backward chaining - **BChain _Search**. This class has a variable **rules** containing the ruleset (using the same classes as forward chaining). TestBChain sets this up for the usual problem...

```
BChain_Search bc=new BChain_Search(rs);
```

We need to define a subclass of **Search_State** representing a state in the backward chaining search:

BChain_State

A state will contain a **Vector** of goals **gl.** A goal will be a String which may have. matching variables.

In addition we will keep an answer-variable-list (**avl**) in each state. This is used to record values found for the variables in the original goal. The search succeeds when a state is found with values for all the answer-variables, and no remaining goals.

For instance, this code (in **TestBChain**) creates the initial state in the search for Elizabeth's grandfather:

```
Vector goals = new Vector(); //initial goals
goals.add("?X grandfather of E");
HashMap avl = new HashMap(); //initial ans var list
avl.put("?X",null);

BChain_State bstate = new BChain_State(goals, avl);
```

BChain_State must implement **goalP**, **getSuccessors** and **same_State**

goalP returns true if there are no more goals to satisfy & we have values for all the answer variables:

```
public boolean goalP(Search searcher) {
  if (gl.isEmpty()&&!avl.containsValue(null)) {return true;}
    else {return false;}
   }
```

same_State returns true if the two states have the same **gl** & **avl.** The action is in **get_Successors**:

get_Successors

Create an empty successsor list

Iterate the following for goal **g** in **gl**:

Iterate the following for each rule **r:**

Match the conseq of **r** against **g**, using **match_2_way**

If there is a match, we are going to create a successor **s**..

Create the goal-list for **s** from the antecedants of **r** plus the other goals in **gl**

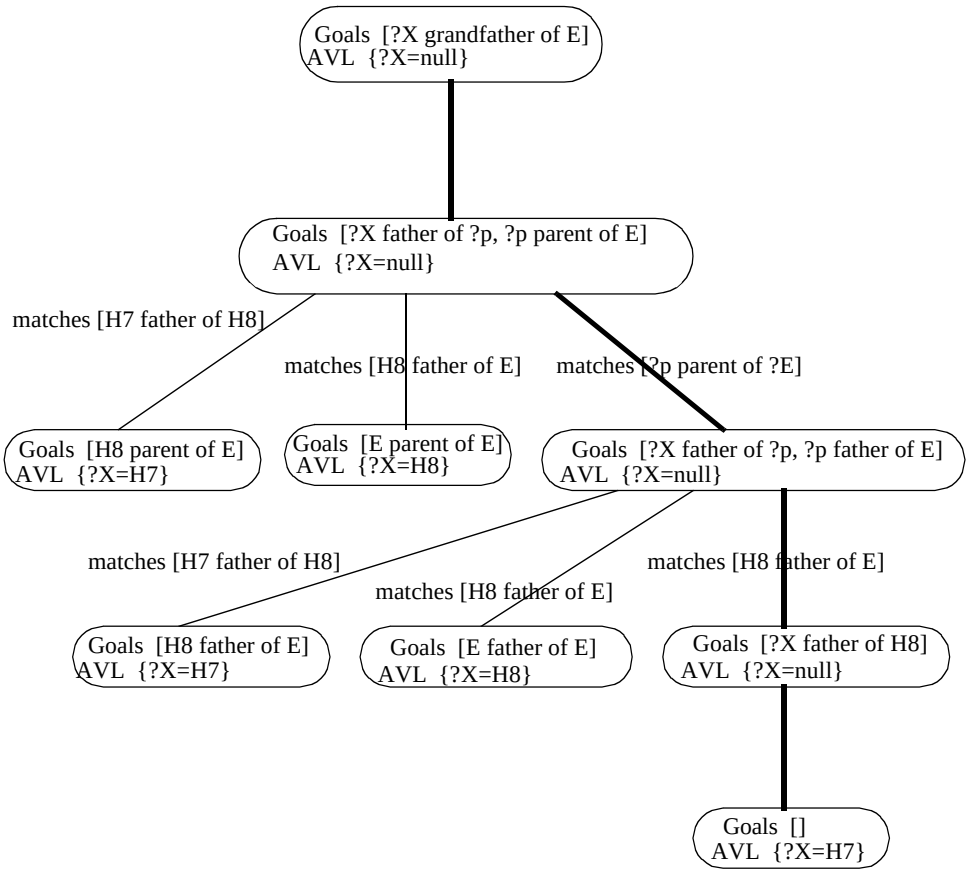
Use **msubst** to substitute in this goal-list for mvars in the **pcon** and **dcon** returned by **match_2_way**.

Create the avl for **s** from the avl for this state by substituting any values for answer vars that have emerged in **dcon.**

add s to the successor list

Results

This is the depth-first search tree for the 'Elizabeth's grandfather' problem:



5.2.3 Things to try

Add some more 'family tree' rules (there are some in 5.4 below). Compare forward and backward chaining for the same problem. How much repeated work is there?

5.3 Production Systems

Unrestricted forward and backward chaining are prone to the combinatorial explosion: they thrash around in the search space without progressing towards a solution.

Production Systems implement forward chaining but provide a way in which the behaviour of the deduction process can be controlled. Many expert system shells resemble production systems. They have also been used to model human learning.

We keep a data structure called the **short-term memory (STM).** This consists of a list of facts, like the initial facts and deductions in forward chaining. The difference is that we are allowed to delete items from the STM, to keep things under control[1]

Instead of rules we have **productions**. A production contains

- a **name**
- a number of **antecedants**, which will contain pattern variables, as before
- a **predicate** on the pattern variable bindings, i.e. a function which is given the context resulting from a successful match of the antecedants and returns **true** or **false**. *The production can only fire if the predicate returns true.*
- a **modify_context** function which takes the context after a successful match and modifies it, for instance to compute the value of a new matching variable. Examples below.

In some productions the predicate &/or modify_context may not be needed.

- a number of **actions** which are carried out if the production fires, in the context returned by **modify_context**. There are three kinds of actions:

  additions are patterns to be added to the STM (after substituting from the context).

  deletions are patterns to be deleted from the STM (after substituting from the context).

  remarks are patterns to be printed out (after substituting from the context).They are used to supply commentary and results.

A **production system interpreter** repeatedly

- Scans through the productions in order until one is found whose antecedants match the **STM**,
- Checks the **predicate** for this prodn in the resulting context and, if this returns true,
- Runs the production's **modify_context** fn,
- '**Fires**' the production, i.e. performs its actions in the resulting context.

This continues until no production fires.

1. In work modelling human learning, the maximum size of the STM is limited, to reflect the characteristics of human memory.

5.3.1 Example: bagging

Robbie the robot has a job in a supermarket. As shopping items are passed through the checkout, Robbie must put them into bags. A good bagging system will make sensible decisions about what goes where, based on factors such as

- How much space there is left in each unfilled bag,
- The size of the items,
- The weight of the items,
- How fragile each item is.

Bagging is an everyday example of the kind of problem solved by one of the most successful expert systems, **R1**, which configured DEC mainframe computers.

Here is a very simple production system for bagging, called **bagger1**. It makes the following simplifications:

- all bags are 100 units high
- items are stacked one on top of the other
- we don't care about item fragility, weight etc.

Bagger1 simply takes each item in turn & tries to fit it in the current bag. If it won't go, a new bag is started.

The **initial STM** will tell us what is in the trolley, how much space each item needs and a note that will trigger the production which starts things off:

step is start bagging

trolley contains bread space 30

trolley contains spuds space 50

trolley contains cornflakes space 40

The 'step' trick is commonly used to restrict the productions which are 'active' at a given time.

THE BAGGER1 PRODUCTIONS

Prodn No 1 **Name**: START_BAGGING

Antecedants: step is start bagging

Predicate: none **Modify_Context**: none

Actions:

deletions: step is start bagging

additions: step is get next item

current bag no 1 space 100

remarks: starting to bag

Prodn No 2 **Name**: GET_NEXT_ITEM

Antecedants: step is get next item

trolley contains ?I space ?S"

Predicate: none **Modify_Context:** none

Actions:

deletions: step is get next item
trolley contains ?I space ?S

additions: step is bag item
item to bag ?I space ?S

remarks: bagging ?I

Prodn No 3 **Name:** BAG_IN_CURRENT

Antecedants: step is bag item

item to bag ?I space ?S

current bag no ?N space ?BS

Predicate: ?BS>=?S

Modify_Context: add ?RS <== ?BS-?S

Actions:

deletions: step is bag item
item to bag ?I space ?S
current bag no ?N space ?BS

additions: step is get next item
bag ?N contains ?I
current bag no ?N space ?RS

remarks: ?I in bag no ?N

Prodn No 4

Name: START_NEW_BAG

Antecedants: step is bag item

current bag no ?N space ?BS

Predicate: none

Modify_Context: add ?NB <== ?N+1

Actions:

deletions: current bag no ?N space ?BS

additions: current bag no ?NB space 100

remarks: Starting bag ?NB

Notice that the order of the productions is important: START_NEW_BAG only fires if BAG_IN_CURRENT doesn't: i.e there isn't enough space in the current bag.

In general this isn't good enough, and attention has to be paid to the problem of **conflict resolution**: if more than one rule has its antecadents and predicate satisfied, which should fire?

5.3.2 Java Implementation

see

/share/com1080/java/Prodn_Systems

Java makes life difficult in this case, for reasons which expose its limitations for AI programming:

- The natural thing is to have each production as an instance of a **Prodn** class, as we did with rules.
- But we need to specify different **predicates** and **modify_contexts** for different productions, i.e. different code.
- The only way we can do this is to write them as methods (as far as I know).
- But we can't have different methods for different instances of the same class.
- Therefore *each production has to be coded as a different class*. **Prodn** becomes an abstract class which individual productions are subclasses of.

Here's the code for the **Prodn** class:

```
public abstract class  Prodn {

String name;//production name
String[] antes;//antecedants
String[] adds;//additions to stm
String[] dels;//deletions from stm
String[] remarks; //printouts on firing

//can't give the accessor code here, otherwise don't get val from concrete
class
 abstract String getName();
 abstract String[] getAntes();
 abstract String[] getAdds();
 abstract String[] getDels();
 abstract String[] getRemarks();

 abstract boolean pred(HashMap context);
 abstract HashMap modify_context(HashMap context);
}
```

and here's the code for BAG_IN_CURRENT

```
public class b1_bag_in_current extends Prodn {
final static String name = "BAG-IN-CURRENT";
final static String[] antes = {"step is bag item",
                               "item to bag ?I space ?S",
                               "current bag no ?N space ?BS"};
final static String[] adds = {"step is get next item",
    "bag ?N contains ?I",
    "current bag no ?N space ?RS"};
final static String[] dels = {"step is bag item",
                              "item to bag  ?I space ?S",
                              "current bag no ?N space ?BS"};
final static String[] remarks = {"?I in bag no ?N"};

//must define accessors here
public String getName(){return name;}
public String[] getAntes() {return antes;}
public String[] getAdds() {return adds;}
public String[] getDels() {return dels;}
public String[] getRemarks(){return remarks;}

public boolean pred(HashMap c){
Integer space_left = Integer.valueOf((String) c.get("?BS"));
Integer space_needed = Integer.valueOf((String) c.get("?S"));
return (space_left.intValue()>=
      space_needed.intValue());}
public HashMap modify_context(HashMap c){
Integer space_left = Integer.valueOf((String) c.get("?BS"));
Integer space_needed = Integer.valueOf((String) c.get("?S"));
c.put("?RS", String.valueOf(space_left.intValue()-space_needed.intValue()));
return c;}
}
```

The problem is that part of the knowledge that we wish to put into a production - simply by writing it down - is in fact code that we want to execute at the appropriate time. Java isn't set up for this.

In functional languages like Haskell (which you'll learn in COM2010) life is easier because we can create objects which represent functions and use them just like other objects. For the pred fn in BAG_IN_CURRENT you'd write something like

```
pred :: HashMap -> Boolean // pred is a fn that takes a HashMap
                           // & returns a Boolean
pred c = (get c ?BS)>=(get c ?S) // the code
```

In lisp it's even easily because there is no distinction between code and data & you don't have to mess around with type declarations. In your data structure for BAG_IN_CURRENT you'd say that the pred is

(>= ?BS ?S)[1]

and the code for checking any predicate would be

(eval (msubst pred context))

... where the world's most useful function **eval** takes its textual argument as code and evaluates it.

ProdSys.java is the class for running production systems. It contains a method **run_PS** which implements a production system interpreter as described above.

TestPS.java runs **bagger1** for the trolley contents above**,** producing

```
RUNNING PRODUCTION SYSTEM
----------------------------------------------------------------------
STM= [step is start bagging, trolley contains bread space 30, trolley contains
spuds space 50, trolley contains cornflakes space 40]
Firing START-BAGGING
 in context {}
START-BAGGING remarks:
  starting to bag
----------------------------------------------------------------------
STM= [trolley contains bread space 30, trolley contains spuds space 50, trolley
contains cornflakes space 40, step is get next item, current bag no 1 space
100]
Firing GET-NEXT-ITEM
 in context {?I=bread, ?S=30}
GET-NEXT-ITEM remarks:
  bagging bread
----------------------------------------------------------------------
STM= [trolley contains spuds space 50, trolley contains cornflakes space 40,
current bag no 1 space 100, step is bag item, item to bag bread space 30]

Firing BAG-IN-CURRENT
 in context {?N=1, ?RS=70, ?BS=100, ?I=bread, ?S=30}
BAG-IN-CURRENT remarks:
  bread in bag no 1
----------------------------------------------------------------------
STM= [trolley contains spuds space 50, trolley contains cornflakes space 40,
step is get next item, bag 1 contains bread, current bag no 1 space 70]
```

1. To perform a computation in lisp you form up a list of the function name followed by its arguments, & then evaluated it, e.g. (+ 1 2) evaluates to 3.

```
Firing GET-NEXT-ITEM
 in context {?I=spuds, ?S=50}
GET-NEXT-ITEM remarks:
  bagging spuds
---------------------------------------------------------------------
STM= [trolley contains cornflakes space 40, bag 1 contains bread, current bag
no 1 space 70, step is bag item, item to bag spuds space 50]
Firing BAG-IN-CURRENT
 in context {?N=1, ?RS=20, ?BS=70, ?I=spuds, ?S=50}
BAG-IN-CURRENT remarks:
  spuds in bag no 1
---------------------------------------------------------------------
STM= [trolley contains cornflakes space 40, bag 1 contains bread, step is get
next item, bag 1 contains spuds, current bag no 1 space 20]
Firing GET-NEXT-ITEM
 in context {?I=cornflakes, ?S=40}
GET-NEXT-ITEM remarks:
  bagging cornflakes
---------------------------------------------------------------------
STM= [bag 1 contains bread, bag 1 contains spuds, current bag no 1 space 20,
step is bag item, item to bag cornflakes space 40]
Firing START-NEW-BAG
 in context {?NB=2, ?N=1, ?BS=20}
START-NEW-BAG remarks:
  Starting bag 2
---------------------------------------------------------------------
STM= [bag 1 contains bread, bag 1 contains spuds, step is bag item, item to
bag cornflakes space 40, current bag no 2 space 100]
Firing BAG-IN-CURRENT
 in context {?N=2, ?RS=60, ?BS=100, ?I=cornflakes, ?S=40}
BAG-IN-CURRENT remarks:
  cornflakes in bag no 2
---------------------------------------------------------------------
STM= [bag 1 contains bread, bag 1 contains spuds, step is get next item, bag 2
contains cornflakes, current bag no 2 space 60]
RUN TERMINATED
final STM
[bag 1 contains bread, bag 1 contains spuds, step is get next item, bag 2 con-
tains cornflakes, current bag no 2 space 60]
```

5.3.3 Things to try

Write a better bagger

5.4 Rule Networks and Token-Passing

Implementing forward (and backward) chaining as above is hopelessly inefficient because it **duplicates work**.There are two reasons for this:

1. Within cycle repetition

In large rule sets it is common to find rules which share some antecedents, e.g.

Rule 1: if A and B and C and D then X

Rule 2: if A and B and C and E then Y

If we are matching antecedants independently for these 2 rules we repeat the work of checking A,B and C.

We need to make use of the relationships between the rules by creating a **rule network**. Then we can (automatically) create an intermediate node which captures what these rules have in common: **A and B and C**.

2. Between cycle iteration

Consider these 2 rules again. At a given time we might be able to match A,B and C but go no further. Later on D is added to our facts. We don't want to go back through the work of matching A,B & C again.

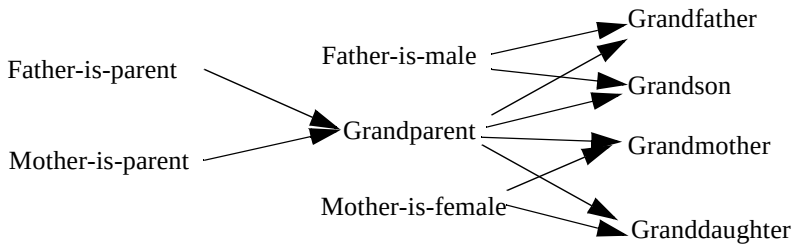
What we need to do is remember **partial matches**. These are called **tokens**. A token holds the remaining antecedants and the context formed from those that have matched.

5.4.1 Rule Networks

Let's add a few more 'family rules' to our rule-set:

Rule Name	Antecedents	Consequent
father-is-parent	(?f is the father of ?c)	(?f is a parent of ?c)
father-is-male	(?f is the father of ?c)	(?f is male)
mother-is-parent	(?m is the mother of ?c)	(?m is a parent of ?c)
mother-is-female	(?m is the mother of ?c)	(?m is female)
grandparent	(?gp is a parent of ?p) (?p is a parent of ?c)	(?gp is a grandparent of ?c)
grandfather	(?gp is a grandparent of ?c) (?gp is male)	(?gp is a grandfather of ?c)
grandson	(?gp is a grandparent of ?c) (?c is male)	(?c is a grandson of ?gp)
grandmother	(?gp is a grandparent of ?c) (?gp is female)	(?gp is a grandmother of ?c)
granddaughter	(?gp is a grandparent of ?c) (?c is female)	(?c is a granddaughter of ?gp)

The rules are related like so: a link means that the consequent of one rule might match with an antecedent of another rule (a **successor** rule).



We can create such a net automatically, given the rules. Each rule will have a successor slot containing a list of successor rules

5.4.2 Tokens and forward-chaining

We are going to implement forward chaining: we supply facts, one at a time, to the net. Every time a fact is supplied we make all the deductions we can.

Associated with each rule we maintain a list of **tokens.** Each token represents a *partial match* for the antecedents of the rule we can make on the basis of what we already know. Each token is a pair : the remaining antecedents and the context formed by those antecedents that have already matched. Initially, each rule has a single token, with all the antecedents and an empty context.

Whenever a fact becomes known that completes the match for a token, i.e. there are no remaining antes, we can make a deduction. We report it (and maybe add it to a list of known facts) and **propagate** it to all the **successors** of the rule (which may in turn lead to other deductions....). This is called **token-passing.** Note that it ensures that a deduction is only passed on to rules which might be able to make use of it.

Initially, the token-list for each node is set to contain a single token, with all the rule's antecedents and an empty context. So for the **father-is-parent** rule we have a token

token

remaining-antecedents ((?p is the father of ?c))

context: ()

Whenever a new fact is supplied from the outside world, we propagate it to **all** the rules. The fact is checked against all the tokens for each rule.

For instance, if we supply

(henry7 is the father of henry8)

This will match against the token in the **father-is-parent** token-list, producing a new token

token

remaining-antecedents ()

context: (?p henry7 ?c henry8)

The token which matched remains on the token list - we want to pick up later facts or deductions which match it, e.g. (henry8 is the father of Elizabeth).

Whenever a token has no remaining antecedents, we

- remove it from the token-list
- use the context to substitute in the consequent and make a deduction (e.g print it out or add it to a list of deduced facts)
- propagate the deduction to all the rule's successors.

So in this case **(henry7 is a parent of henry8)** is propagated to the **grandparent rule.** There it creates 2 new tokens, because it will match with either antecedent

token

remaining-antecedents (?p is a parent of ?c)

context (?gp henry7 ?p henry8)

token

remaining-antecedents (?gp is a parent of ?p)

context (?p henry7 ?c henry8)

Similarly, the **father-is-male** rule will fire, deducing **(henry7 is male)** and propagating to the **grandfather-rule** and the **grandson** rule.

Now if we add the fact **(henry8 is the father of elizabeth)**

- The **father-is-parent** rule will deduce **(henry8 is a parent of elizabeth)** and propagate this to the **grandparent-rule.**
- This will match with the first of the 2 tokens created above for that rule, deducing **(henry7 is a grandparent of elizabeth)**
- and so on: this printout is from my lisp implementation:

  propagating to grandfather-rule

  deduction: (henry7 is a grandfather of elizabeth)

  propagating to grandson-rule

  propagating to grandmother-rule

  propagating to granddaughter-rule

  deduction: (henry8 is male)

  propagating to grandfather-rule

  propagating to grandson-rule

- finally, if we add **(Elizabeth is female),** the **granddaughter-rule** will deduce **(elizabeth is a granddaughter of henry7)**

This kind of algorithm is common in applications which involve deductive rule-sets, for instance **parsing** in natural language processing and speech recognition. It was originally developed in the context of the expert system R1, where it was called the RETE match algorithm.

5.4.3 Object-Oriented implementation of Token-Passing

How would we implement this in and object-oriented language like Java or Lisp? (Lisp is basically a functional language but has an object system CLOS).

- Create a class **rule-node** for rules which are part of a net.
- **rule-node** has slots

  rule-name

  antecedents

consequent

successors - a list of **rule-nodes**

tokens

- Create a class **rule-net** which has a slot **rules** - a list of the **rule-nodes** in the net.
- The **rule-net** class has two methods

  initialise-net sets up the initial tokens for each rule

  add-fact takes a fact from the outside world and **propagate**s it to every rule.
- **propagate** is a method of a **rule-node** which is given a **fact.** It implements forward chaining by

  checking each token in the rule-node's token slot for possible matches against remaining antecedents.

  If any token completes, the deduction is reported and **propagate** recurses for the node's successors.

  Any new, but still incomplete, tokens are added to the token-list.

So the state of the net is maintained by the contents of the token slot for each rule.

6 PLANNING

6.1 Planning v. Search

In **state-space search** we find a sequence of actions which will transform an initial state into a goal state. We do this by exploring state space starting from the initial state and moving gradually away from it. If the information is available, we control and guide the search using costs from the initial state and estimates of remaining costs.

In **planning**, the problem is the same but the approach to finding the solution path is different. Planning systems make use of knowledge about the potential actions (or **operators**) that might be used, how each operator will change the problem state and what **preconditions** each operator has, i.e. what is requires before the operator can be used.

For instance, suppose I'm planning a journey from home to a workshop in Patras, Greece.

- I know about various operators to do with transport: **walk**, **drive**, **take-taxi**, **take-train**, **take-plane**, ...
- I know that **walk** is good for short distances, **take-plane** for long distance and so on.
- I know that **take-plane** requires that I'm at an airport and have a ticket, **take-taxi** requires that I'm at a taxi rank and I have local currency etc.

To plan the solution to a problem

- Compare the current state with the goal state, to find what it is that must be changed. *e.g. a difference in location of around 2000 miles.*
- Examine the potential actions, looking for something which is capable of making the change. ***take-plane*** *fits the bill*
- Having found a promising action, try to apply it. This means meeting its preconditions: *I need to be at the airport with a ticket.*
- Each unfulfilled precondition leads to a new problem: plan a way of meeting it. *How am I going to get to the airport?*
- These new problems are handled (**recursively**) in the same way.

Notice that we may find that some plan doesn't work out - *I might try* ***fly-to Thessaloniki*** *but then discover that there's no train or coach service from there to Patras. I have to backtrack and consider* ***fly-to Athens*** *instead.*

So planning involves **looking ahead** to consider what we might be able to do in some future state. In search, we only look at what is possible from the current state.

6.2 Planning with STRIPS

The classic work on planning was done by Newell et al. and led to a scheme called **GPS** (General Problem Solver). GPS introduced the planning scheme above, which the authors called **means-end analysis.**

STRIPS (Fikes, Hart & Nilsson 72) is essentially an implementation of GPS with a particular representation scheme for knowledge about operators. This is what we'll look at.

As with search, we separate the planning mechanism from problem-specific knowledge, so that we can solve planning problems in different problem domains. A fallible Java implementation is available in

/home/pdg/public/com1080/java/strips

This directory also contains the **pmatch** package.

6.2.1 Representing Operators

The example we'll take is

I'm in my living room and I want my friendly robot to get me a beer. The beer is in the kitchen. To go from living room to kitchen the door has to be open.

Code for this problem is in **beerStrips.java** in the **strips** directory.

The operators involved are **carry**, **move**, **open**-door and **close**-door**.** In STRIPS each operator has

- an **act_list**, specifying what will be added to the plan if the operator is used (or **applied**) - the action to take.
- an **add_list**, specifying what will be added to the current state if the operator is **applied**.
- a **del_list**, specifying what will be removed from the current-state on operator application
- a list **preconds** of preconditions which must be satisfied (by changing the current state) before the operator can be applied.

We'll have a class **Strips_op** to represent operators. It will have private variables **act_list, add_list, del_list** and **preconds,** together with their accessors. For the beer problem the operators are

Operator	act_list	add_list	del_list	preconds
open	open door from ?r1 to ?r2	door open ?r1 ?r2	door closed ?r1 ?r2	Robbie in ?r1 door closed ?r1 ?r2
close	close door from ?r1 to ?r2	door closed ?r1 ?r2	door open ?r1 ?r2	Robbie in ?r1 door open ?r1 ?r2
move	move from ?r1 to ?r2	Robbie in ?r2	Robbie in ?r1	Robbie in ?r1 door open ?r1 ?r2
carry	carry ?obj from ?r1 to ?r2	Robbie at ?r2 ?obj in ?r2	?obj in ?r1 Robbie in ?r1	?obj in ?r1 Robbie in ?r1 door open ?r2 ?r1

Each of these variables will be an **MStringVector.** There is a constructor which takes them as **Strings**, using the character **'|'** to separate the list items.

The **act_list** is the pattern for the action which will (with the appropriate substitutions) go into the plan which is our final result. A plan is a list of actions. So for the beer problem strips will return the plan

open door from living room to kitchen

move from living room to kitchen

```
carry beer from kitchen to living room
```

For the beer example the initial state is another **MStringVector**

```
"Robbie in living_room|beer in kitchen|door closed living_room kitchen"
```

and the goal list is an **MStringVector** with one element

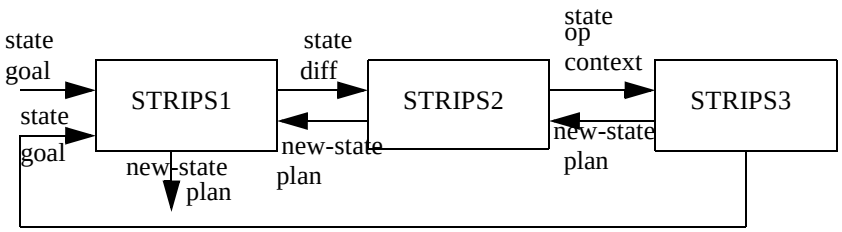
```
"beer in living_room"
```

6.2.2 A Functional Design for STRIPS

In a functional language, STRIPS can be coded in three mutually-recursive functions **Strips1**, **Strips2** and **Strips3**. We start the system by calling **Strips1**, which takes a list of goals, the current state and a list of operators e.g.

```
Strips1 goal_list init_state beer_oplis
```

The STRIPS functions are related roughly as follows:



All the functions take the **current-state** and the list of operators **oplis** as arguments. They have additional arguments as described below.

- **Strips1** is given a **goal_lis** and has the job of finding a **plan** to reach a state in which all the goals in **goal_lis** are met. It does this by finding differences between the **goal_lis** and the **current_state** and calling **Strips2** to deal with them.
- **Strips2** is given a difference and looks for an operator which can do something about it (‘reduce’ it). Having found an operator, it calls **Strips3** to try to apply it.
- **Strips3** is given an operator and a **context** in which to apply it (see below). It tries to meet the operator’s **preconditions**. If a precondition is not met in the **current_state**, **Strips1** is called with the goal of finding a plan to satisfy it. When the preconditions have been met, the operator is applied, producing a plan and a **new_state** which will result if the plan is applied.

When they succeed, **Strips1**, **Strips2** and **Strips3** all return these two values - a **plan** and a **new_state** which results from applying the plan. They may fail, however..

6.2.3 An OOP design for STRIPS

see

```
/share/com1080/java/strips
```

In Java, **Strips1**, **Strips2** and **Strips3** will be separate classes. Since they have many variables in common we’ll make them subclasses of an abstract class **StripsFn:**

```
import simplejava.*;
import java.util.*;
public abstract class StripsFn{
 //variables
 protected Vector oplis;  //operators
 protected MStringVector goalstate; //goal state
 protected MStringVector initstate; //initial state
 protected MStringVector newstate;  //state after carrying out plan
 protected Vector plan;  //final plan
 protected boolean result; //success or failure
 //accessors
 public Vector getOplis(){return oplis;}
 public MStringVector getGoalstate(){return goalstate;}
 public MStringVector getInitstate(){return initstate;}
 public MStringVector getNewstate(){return newstate;}
 public Vector getPlan(){return plan;}
}
```

The Strips functions become methods called **run** in the classes **Strips1, Strips2** and **Strips3**. The **run** methods return a boolean, representing success or failure. If they succeed, they set their variables **newstate** and **plan** appropriately.

Whenever we need to call **Strips1, Strips2** or **Strips3** on a new problem, we create a new instance. This is necessary because of the way they present their results, by setting **newstate** and **plan.** In a functional implementation we wouldn't do this, we'd just call the function..which would *return* its results in a suitable data structure.

6.2.4 Strips1.java

To run Strips, we start (in beerStrips.java) by creating the operators:

```
Strips_op open = new Strips_op("open door from ?r1 to ?r2",
                               "door open ?r1 ?r2",
                               "door closed ?r1 ?r2",
                               "Robbie in ?r1|door closed ?r1 ?r2");
Strips_op closed = new Strips_op("close door from ?r1 to ?r2",
                                 "door closed ?r1 ?r2",
                                 "door open ?r1 ?r2",
                                 "Robbie in ?r1|door open ?r1 ?r2");
Strips_op move = new Strips_op("move from ?r1 to ?r2",
                               "Robbie in ?r2",
                               "Robbie in ?r1",
                               "Robbie in ?r1|door open ?r1 ?r2");
Strips_op carry = new Strips_op("carry ?obj from ?r1 to ?r2",
```

```
                "Robbie in ?r2|?obj in ?r2",
                "?obj in ?r1|Robbie in ?r1",
                "?obj in ?r1|Robbie in ?r1|door open ?r2 ?r1");
```

& then form them into a Vector:

```
Vector beerops = new Vector();
beerops.add(open);
beerops.add(closed);
beerops.add(move);
beerops.add(carry);
```

Then we create an instance of **Strips1,** giving it the operators, initial state and goal list.

```
Strips1 str=new Strips1(beerops,
  new MStringVector("Robbie in living_room|beer in kitchen|door closed
living_room kitchen"),
  new MStringVector("beer in living_room"));
```

Then we start the planning by

```
boolean res=str.run();
```

Here's Strips1:

```
import java.util.*;
import StripsFn;
public class Strips1 extends StripsFn{
//constructor given oplis, initstate, goal state
 public Strips1 (Vector opl,MStringVector is, MStringVector gs){
  oplis=opl;
  initstate=is;
  goalstate=gs;
 }

 //run problem solver
 public boolean run(){
  //commentary
  System.out.println("-----------------");
  System.out.println("Strips1");
  System.out.println(initstate);
  System.out.println(goalstate);

  //look for a difference
  boolean diffound=false; //set to true when diff found
  String g=new String(); //where the diff found will go
```

```
Iterator git=goalstate.iterator();
while(git.hasNext()&& !diffound){ //iterate over goals
  g=(String)git.next();
  //try to match goal g within state - initstate is an MStringVector
  diffound=!initstate.match(g); //becomes true if goal not matched
}
//end of the while...was a difference found?
if (!diffound){
  result=true; //no difference found - trivial success
  newstate=initstate; //end state same as initstate
  plan=new Vector(); //empty plan
  return result;
}
else{ //found a diff, use Strips2
  Strips2 strips2=new Strips2(oplis,initstate, goalstate,g); //make instance
  //Strips2 needs oplis, initstate, goalstate and difference - the goal not met
  boolean s2res=strips2.run(); //run it
  if (!s2res){  //did Strips2 succeed?
    result=false; //no, failure
    return result;
  }
  else { //strips2 succeeded - call Strips1 again with state returned
    //must be a new instance - state will have changed
    Strips1 nstrips1=new Strips1(oplis, strips2.getNewstate(),goalstate);
    boolean nstrips1res=nstrips1.run(); //run the new Strips1
    if (!nstrips1res){ //did it succeed?
      result=false; //no, failure
      return result;
    }
    else {  //success
      result=true;
      //complete plan is strips2 plan + strips1 plan
      plan=strips2.getPlan();
      plan.addAll(nstrips1.getPlan());
      newstate=nstrips1.getNewstate();//final state is that returned by nstrips1
      return result;
    }
  }
 }
}
```

6.2.5 Strips2.java

```
import java.util.*;
import StripsFn;

public class Strips2 extends StripsFn{

 String diff;  //difference to reduce

 //constructor
 public Strips2 (Vector opl,MStringVector is, MStringVector gs, String d){
  oplis=opl;
  initstate=is;
  goalstate=gs;
  diff=d;
 }

 //run problem solver

 public boolean run(){
  //commentary
  System.out.println("------------------");
  System.out.println("Strips2");
  System.out.println("working on "+diff);

  result=false; //set to true when diff dealt with
  Iterator opit=oplis.iterator();
  //search for an operator to reduce the diff, then call strips3 to apply it
  while (opit.hasNext()&& !result){ //try ops till one succeeds or none left
   Strips_op op= (Strips_op)opit.next();
   boolean matchres=op.getAdd_list().match(diff); //match diff against addlist
   if (matchres){ //op can deal with diff
    HashMap con=op.getAdd_list().getContext(); //in this context
    System.out.println("calling Strips3 to apply operator "+op.getAct_list());
    //call strips3 to attempt to apply op
    //Strips3 needs the context
    Strips3 strips3 = new Strips3(oplis, initstate, goalstate,op,con);
    boolean strips3res=strips3.run(); //run Strips3
    if (strips3res){ //strips3 succeeded
     plan=strips3.getPlan(); //Strips2 plan is Strips3 plan
     newstate=strips3.getNewstate(); //new state is Strips3 new state
```

```
        result=true;
      }
     }
    }
    return result; //will be false if no op succeeded
   }
}
```

6.2.6 Strips3

- The **run** method of **Strips3** has the task of applying an operator. **Strips3's** constructor looks like

```
public Strips3 (Vector opl,MStringVector is, MStringVector gs, Strips_op o,
                HashMap con){
 oplis=opl; //operators
 initstate=is; //initial state
 op=o; //operator to apply
 context=con; //incoming context
}
```

- The first job is to meet **op**'s preconditions. We'll have a separate method **meet_preconditions** to do this. It will return a **boolean**, **true** if the preconds have been met. If **meet_preconditions** returns **false,** Strips3 returns false
- if **meet-preconditions** succeeds it will produce a plan, **preplan** which will result in a new state, **prestate**. It will also establish a new context, **pcon.** These will be private variables of **Strips3**. For example, if we are trying to apply **move** in the context **{?r2= kitchen}** and **current_state** contains **"Robbie in living_room"** we'll satisfy the precondition **"Robbie in ?r1"** in the context **{?r1=living_room, ?r2= kitchen}**
- we can now at last apply the operator. A separate method **apply_op** is called to do this, given **op, pre_state** and **pre_con.**
- to apply an operator to a state given a context, we use the op's **del_list** to find what needs to be deleted from the state, the **add_list** to find what must be added and the **act_list** to work out the 1-action plan (**aplan**) for applying the operator. For all these, we use the **msubst** method (they are **MStringVectors**) to replace **?vars** with their matching values in the context after meeting the preconditions, **precon**. There are **addAll** and **removeAll** methods of **Vector** which will do the job of establishing the state after applying the operator, **newstate** - the state after **Strips3** has done its job.
- The final **plan** after applying the op is the concatenation of the plan returned by meet_preconditions, **preplan**, and the plan returned by apply_op, **aplan**.

6.2.7 Meet-preconditions

- **meet_preconditions** iterates through **op's preconds** trying to meet each one. Stopping conditions are when no preconds remain (success) or we fail to meet one (failure).
- Before starting this iteration, we need to set up the following:

```
prestate=initstate; //state having met some preconds
```

```
pcon=context; //context after having matched some preconds
preplan=new Vector(); //plan for preconds, initially empty
```

- For a given precond **p**, we call the **match** method of **initstate**, giving it the current context:
  `boolean matchres=initstate.match(p,pcon)`
- If **matchres** is **true**, we update **pcon** & move on to the next precondition.
- If **matchres** is **false**, we need to invoke a new instance of **Strips1** and use it to solve the new problem of meeting this precondition.
- To specify what the goal is for this new **Strips1**, we need to substitute for the context in **p** (e.g. we are trying to apply **carry** in the context **{?r2=kitchen}**. We'll need to satisfy the precondition "**Robbie in ?r2"**. For the new call of **Strips1** we need to make the goal **"Robbie in kitchen"**.
- The new **Strips1** will either succeed or fail. If it fails then **Strips3** fails. If it succeeds, its **plan** is added to **preplan** and its **newstate** becomes **prestate**.

6.3 The beer example

Here we go... printouts from my lisp version...

```
STRIPS1
**********
 goal-list ((BEER IN LIVING-ROOM))
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 --> calling STRIPS2 to deal with (BEER IN LIVING-ROOM)

STRIPS2
**********
 diff (BEER IN LIVING-ROOM)
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 --> calling strips3 to try op (CARRY ?OBJ FROM ?R1 TO ?R2)
STRIPS3
**********
 op (CARRY ?OBJ FROM ?R1 TO ?R2)
 context (?R2 LIVING-ROOM ?OBJ BEER)
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 calling STRIPS1 to meet precondition (ROBOT in KITCHEN)
STRIPS1
```

```
**********
 goal-list ((ROBOT in KITCHEN))
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 --> calling STRIPS2 to deal with (ROBOT in KITCHEN)
STRIPS2
**********
 diff (ROBOT in KITCHEN)
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 --> calling strips3 to try op (MOVE FROM ?R1 TO ?R2)
STRIPS3
**********
 op (MOVE FROM ?R1 TO ?R2)
 context (?R2 KITCHEN)
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
calling STRIPS1 to meet precondition (DOOR-OPEN LIVING-ROOM KITCHEN)

STRIPS1
**********
 goal-list ((DOOR-OPEN LIVING-ROOM KITCHEN))
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 --> calling STRIPS2 to deal with (DOOR-OPEN LIVING-ROOM KITCHEN)
STRIPS2
**********
 diff (DOOR-OPEN LIVING-ROOM KITCHEN)
 current-state ((ROBOT in LIVING-ROOM)
                (BEER IN KITCHEN)
                (DOOR-CLOSED LIVING-ROOM KITCHEN))
 --> calling strips3 to try op (OPEN DOOR FROM ?R1 TO ?R2)
STRIPS3
**********
 op (OPEN DOOR FROM ?R1 TO ?R2)
 context (?R2 KITCHEN ?R1 LIVING-ROOM)
 current-state ((ROBOT in LIVING-ROOM)
```

```
                    (BEER IN KITCHEN)
                    (DOOR-CLOSED LIVING-ROOM KITCHEN))
  op applied with plan ((OPEN DOOR FROM LIVING-ROOM TO KITCHEN))
          giving state ((DOOR-OPEN LIVING-ROOM KITCHEN)
                        (ROBOT in LIVING-ROOM)
                        (BEER IN KITCHEN))
 --> dealt with (DOOR-OPEN LIVING-ROOM KITCHEN) - try STRIPS1 again
STRIPS1
**********
  goal-list ((DOOR-OPEN LIVING-ROOM KITCHEN))
  current-state ((DOOR-OPEN LIVING-ROOM KITCHEN) (ROBOT in LIVING-
ROOM) (BEER IN KITCHEN))
   op applied with plan ((MOVE FROM LIVING-ROOM TO KITCHEN))
          giving state ((ROBOT in KITCHEN)
                        (DOOR-OPEN LIVING-ROOM KITCHEN)
                        (BEER IN KITCHEN))
 --> dealt with (ROBOT in KITCHEN) - try STRIPS1 again
STRIPS1
**********
  goal-list ((ROBOT in KITCHEN))
  current-state ((ROBOT in KITCHEN) (DOOR-OPEN LIVING-ROOM KITCHEN)
(BEER IN KITCHEN))
   op applied with plan ((CARRY BEER FROM KITCHEN TO LIVING-ROOM))
          giving state ((ROBOT in LIVING-ROOM)
                        (BEER IN LIVING-ROOM)
                        (DOOR-OPEN LIVING-ROOM KITCHEN))
 --> dealt with (BEER IN LIVING-ROOM) - try STRIPS1 again
STRIPS1
**********
  goal-list ((BEER IN LIVING-ROOM))
  current-state ((ROBOT in LIVING-ROOM) (BEER IN LIVING-ROOM) (DOOR-
OPEN LIVING-ROOM KITCHEN))
((OPEN DOOR FROM LIVING-ROOM TO KITCHEN)
 (MOVE FROM LIVING-ROOM TO KITCHEN)
 (CARRY BEER FROM KITCHEN TO LIVING-ROOM)) ;
((ROBOT in LIVING-ROOM)
 (BEER IN LIVING-ROOM)
 (DOOR-OPEN LIVING-ROOM KITCHEN))
>
```

6.4 Limitations

Strips as implemented above is pretty crude: you should try to break it.

- The operators have to come in the right order
- It's possible to satisfy goal A, then work on goal B and in doing so undo goal A - goal conflict.
- It's thus possible to get into an endless recursion.
- There's no attempt to decide which goal it's best to work on next.
- The representation scheme is pretty unwieldy.

Later work on planning has attempted to address these limitations.

6.5 Things to try

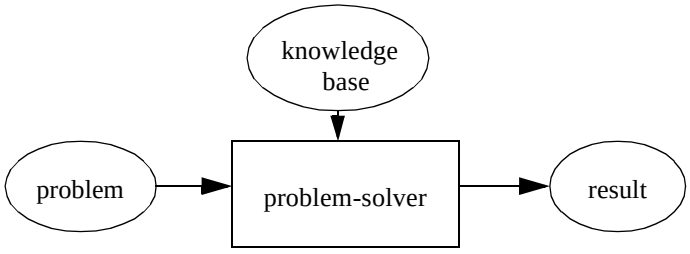
Devise stacking problems which exemplify the limitations above.

7 AI and Perception

References:

- R.L. Gregory, 'Eye and Brain'
- R.L. Gregory, 'The Intelligent Eye'
- J.P. Frisby, 'Seeing'.

In this course we've studied AI systems that look like so:



- The **knowledge base** available to the **problem-solver** is expressed in data structures that make use of some knowledge-representation scheme.
- The **problem** is expressed in a compatible data structure.
- The problem-solver applies the knowledge to the problem by doing **symbolic manipulation** using these data structures.
- The **result** is another data structure.

The data structures representing the knowledge and the problem are provided by the user.

But 'real' intelligent systems have to function without this help:

- **Perceptual** systems have to make sense of data reaching the system from the outside world,
- The system acquires knowledge by **learning** (in a variety of ways),
- **Motor** systems have to convert decisions made by the system into actions.

7.1 The problem of perception

Perception in living systems works so well that the difficulty of the task isn't obvious. We can get a feel for what is happening by driving the system to its limits:

- Here, there are few cues to indicate object boundaries and depth.
- The processing which perceives the object must be active & knowledge-based, not passive transduction.
- **Perception imposes an organisation on the sensory data**: it makes features explicit..
- This example is not so atypical of what happens all the time, and in other modalities besides vision, e.g. we 'hear' pauses between words which aren't physically there.
- It is often easier to write the 'problem-solving' part of an AI system than it is to provide the system with adequate perception.

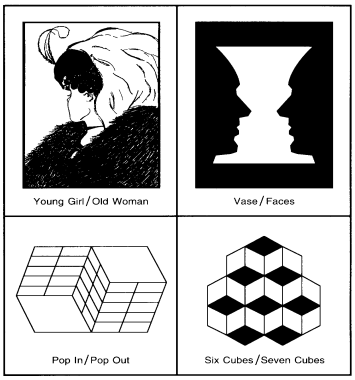
7.2 Gregory's 'Perceptions as Hypotheses' Metaphor

We can think of perceptions as being like hypotheses in science:

- perceptions **explain the data** in terms of arrangements of known objects (in vision),
- perceptions are used to **predict** - to hidden parts of objects, to the future..

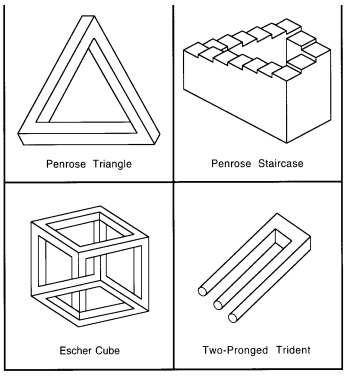
Perception works so well that it's difficult to study without devising ways of making it malfunction. Psychologists have therefore studied perception through illusions. If the system can be confused, or fooled, what does that reveal about its properties?

7.2.1 Ambiguous figures



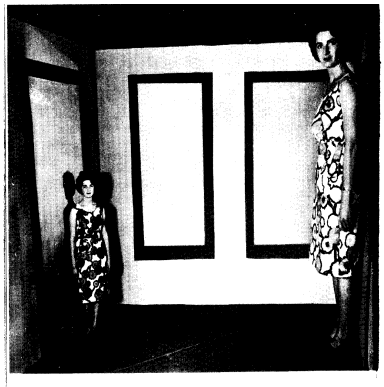
Two viable hypotheses.
Perception alternates
Many effects are due to depth perception

7.2.2 Paradoxical Pictures



The figures suggest familiar objects. Though the evidence isn't consistent, the system continues to 'believe' the hypothesis.

7.2.3 System error!



Ames Room

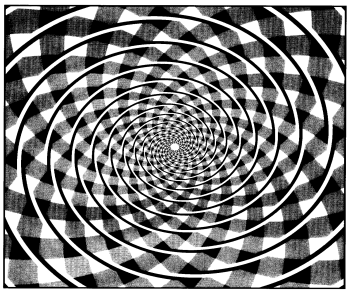
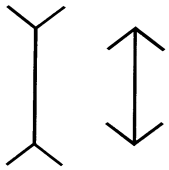


Figure 1.1 Fraser's spiral. Human vision is not quite as infallible as we tend to believe. A case in point is the illusory spiral evoked here by a collection of concentric circles, each circle comprising segments angled toward the center. To convince yourself that there is indeed no spiral, try tracing a spiral with your finger. (After [Fraser 1908].)

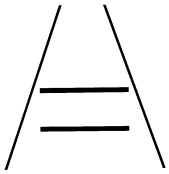
Fraser Spiral

In these cases, there is a correct but unusual hypothesis, but the system prefers th more normal one:

7.2.4 Distortion Illusions



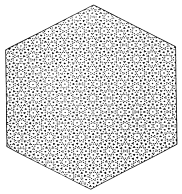
Muller-Lyer



Ponzo

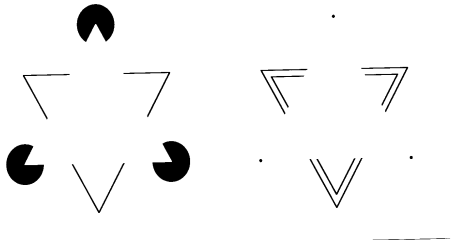
Figures are interpreted as 3D scenes. ‘Size constancy’ operates where it shouldn’t.

7.2.5 Gestalt Grouping Principles



similarity, continuity, proximity, common fate..

7.2.6 Perceptual Invention



Kanitsa’s figure: why is the illusory triangle perceived as brighter?

7.3 Conclusions

- Perception is an intelligent problem-solving activity, involving much of the cortex.
- Perception and reasoning have a rich interaction.
- There is a similar argument for motor activity.