2.1 Local search algorithm

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**. Many optimization problems do not fit the "standard" search model introduced in Chapter 3. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

To understand local search, we find it useful to consider the **state-space landscape** (as in Figure 4.1). A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**. (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

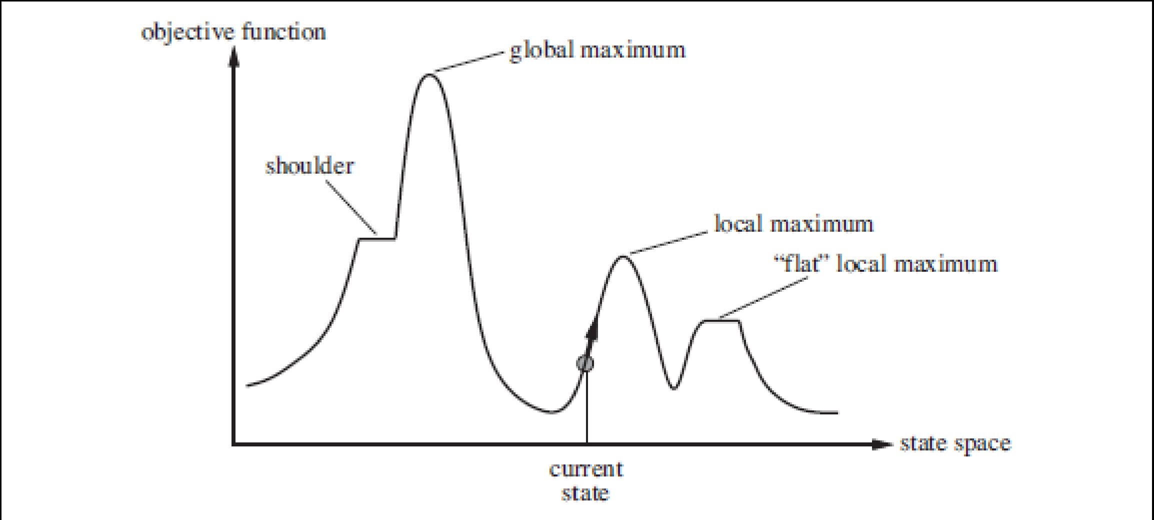


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

2.1.1 Hill Climbing Search

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

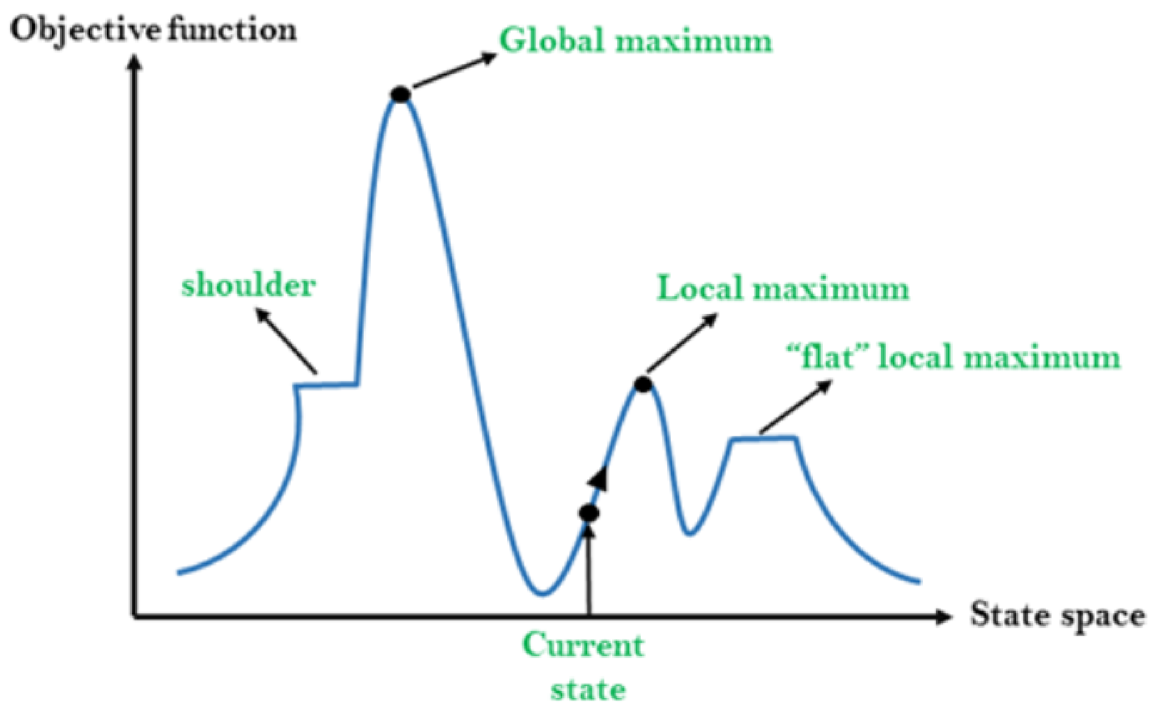
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
current ←MAKE-NODE(problem.INITIAL-STATE)
loop do
neighbor ←a highest-valued successor of current
if neighbor.VALUE ≤ current.VALUE then return current.STATE
current←neighbor
```

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

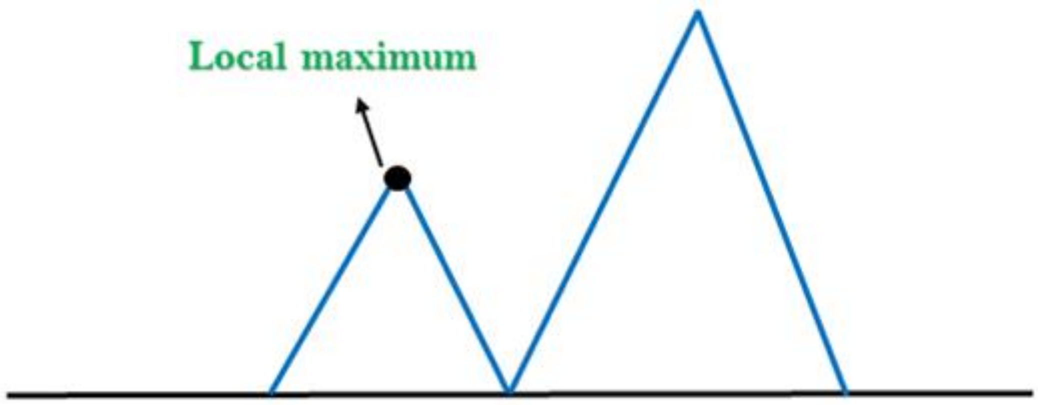
3. Stochastic hill climbing:

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

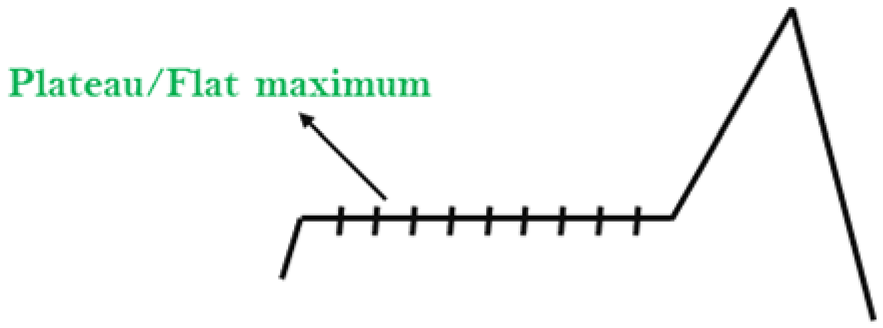
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



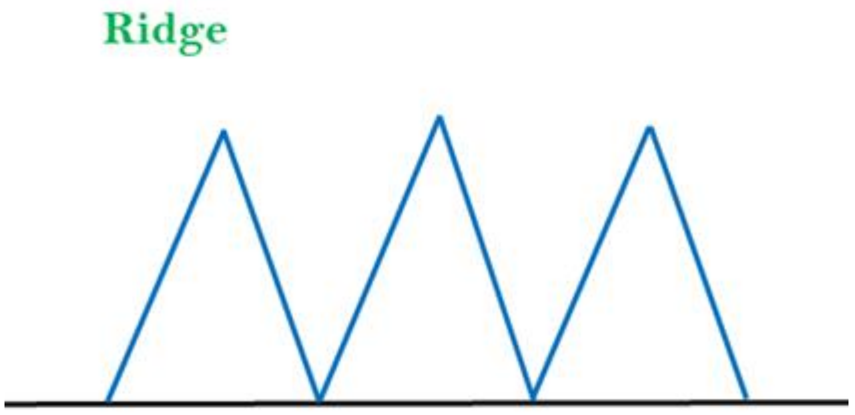
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



2.1.2 Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. **Simulated Annealing** is an algorithm which yields both efficiency and completeness.

In mechanical term **Annealing** is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

The innermost loop of the simulated-annealing algorithm (Figure 4.5) is quite similar to hill climbing. Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for $t$ = 1 to ∞ do
     $T$ ← schedule($t$)
     if $T = 0$ then return current
     next ← a randomly selected successor of current
     $\Delta E$ ← next.VALUE – current.VALUE
     if $\Delta E > 0$ then current ← next
     else current ← next only with probability $e^{\Delta E/T}$
```

2.1.3 LOCAL BEAM SEARCH

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The **local beam search** algorithm3 keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.
At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. *In a local beam search, useful information is passed among the parallel search threads.* In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.
In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps alleviate this problem. Instead

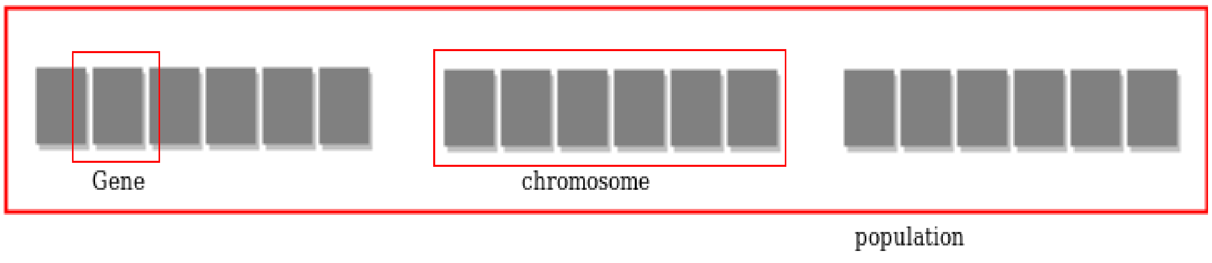
of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value. Stochastic beam search bears some resemblance to the process of natural selection, whereby the “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).

2.1.4 GENETIC ALGORITHMS

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. **They are commonly used to generate high-quality solutions for optimization problems and search problems.**
Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. **Each generation consist of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Search space

The population of individuals are maintained within search space. Each individual represent a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).



```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
inputs: population, a set of individuals
FITNESS-FN, a function that measures the fitness of an individual
repeat
new population ←empty set
for i = 1 to SIZE(population) do
x ←RANDOM-SELECTION(population, FITNESS-FN)
y ←RANDOM-SELECTION(population, FITNESS-FN)
```

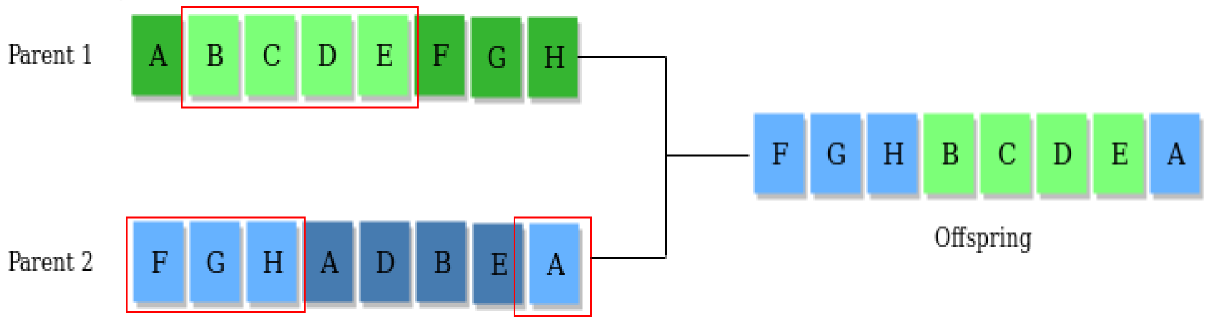
```
child ←REPRODUCE(x , y)
if (small random probability) then child ←MUTATE(child )
add child to new population
population ←new population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN
function REPRODUCE(x , y) returns an individual
inputs: x , y, parent individuals
n←LENGTH(x ); c←random number from 1 to n
return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Operators of Genetic Algorithms

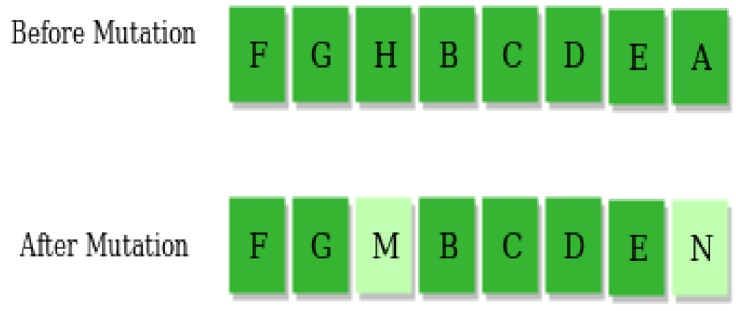
Once the initial generation is created, the algorithm evolve the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass there genes to the successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). For example –



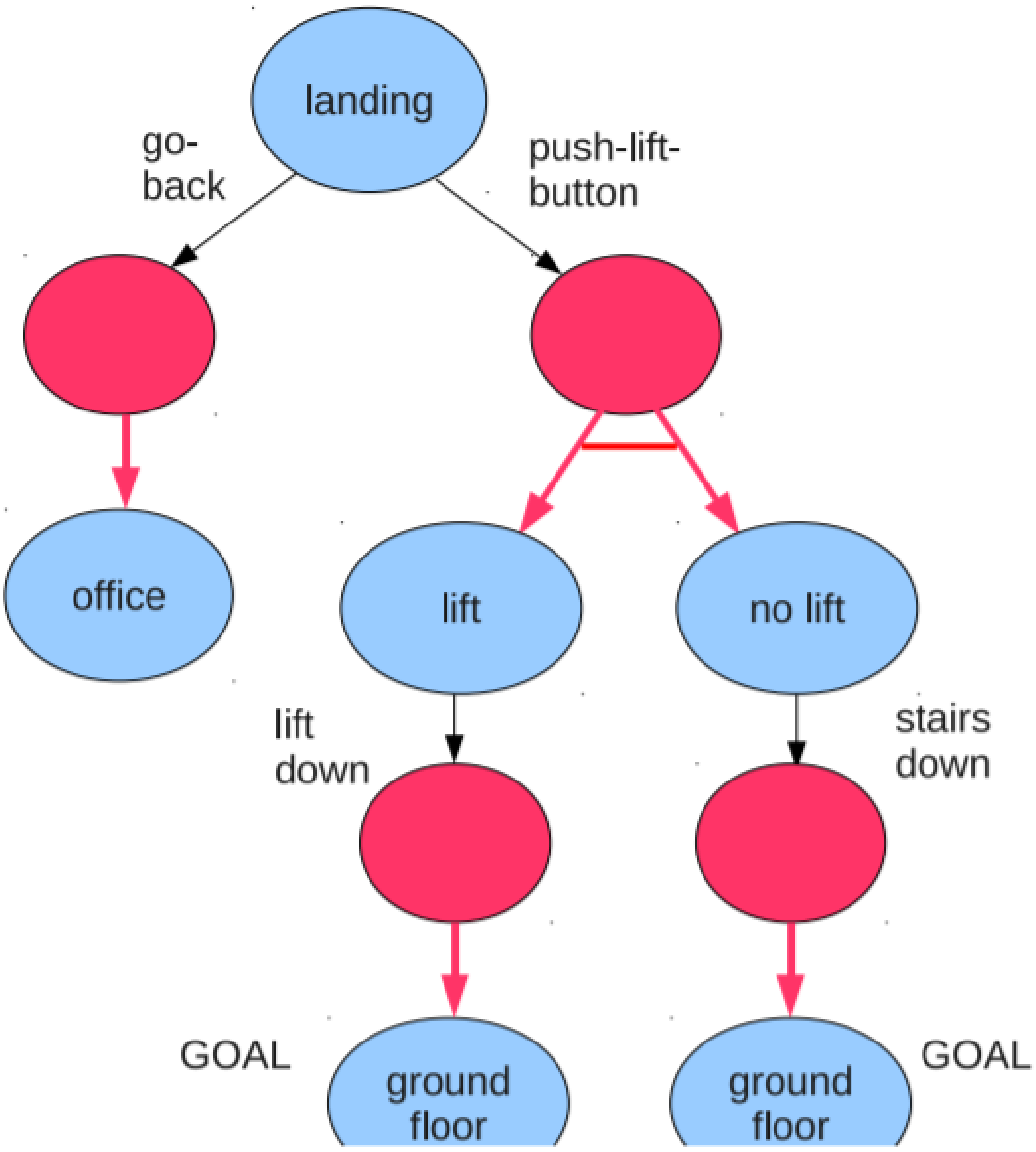
3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence. For example –



2.2 SEARCHING WITH NONDETERMINISTIC ACTIONS

Each action has a set of possible outcomes (resulting states) A solution is not a sequence of actions, but a contingency plan, or a strategy: if after pushing the lift button, lift arrives, then take the lift; else take the stairs.

Previous search trees: branching corresponds to the agent's choice of actions Call these OR-nodes Environment's choice of outcome for each action: AND-nodes



A solution for an AND-OR search problem is a subtree that (1) has a goal node at every leaf (2) specifies one action at each of its OR nodes (3) includes every outcome branch at each of its AND nodes

finds non-cyclic solution if it exists:

```
**function** AND-OR-GRAPH-SEARCH(problem) **returns** a conditional plan, or failure
OR-SEARCH(problem.INITIAL-STATE, problem, [ ])
```

```
**function** OR-SEARCH(state, problem, path) **returns** a conditional plan, or failure
if problem.GOAL-TEST(state) then return the empty plan
if state is on path then return failure
for each action in problem.ACTIONS(state) do
plan ←AND-SEARCH(RESULTS(state, action), problem, [state | path])
if plan _= failure then return [action | plan]
return failure
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
for each si in states do
plani←OR-SEARCH(si, problem, path)
if plani = failure then return failure
return [if s1 then plan1 else if s2 then plan2 else . . . if sn−1 then plann−1 else plann]
```

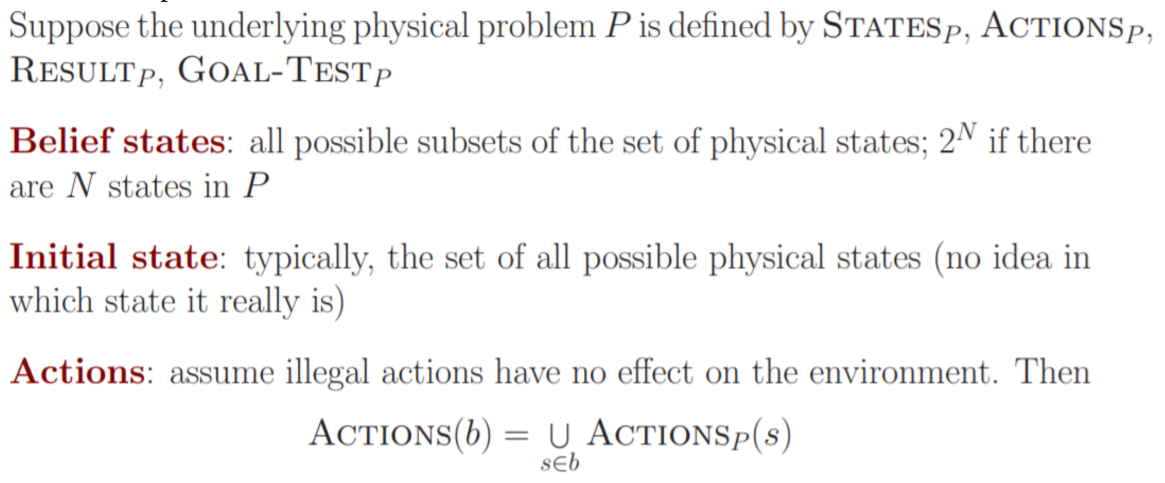
else, finds cyclic solution:
add a while loop; if in state where the action failed, repeat until succeed) this will work provided that each outcome of non-deterministic action eventually occurs

2.3 SEARCHING WITH PARTIAL OBSERVATION

When the agent's percepts provide *no information at all*, we have what is called a **sensor less** problem or sometimes a **conformant** problem. At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensorless problems are quite often solvable.

The agent knows that it is in one of a set of possible physical states This set of physical states is called a belief state.

Belief state space search:



Suppose the underlying physical problem P is defined by $\textsc{States}_P$, $\textsc{Actions}_P$, $\textsc{Result}_P$, $\textsc{Goal-Test}_P$

Belief states: all possible subsets of the set of physical states; 2^N if there are N states in P

Initial state: typically, the set of all possible physical states (no idea in which state it really is)

Actions: assume illegal actions have no effect on the environment. Then

$$\textsc{Actions}(b) = \bigcup_{s \in b} \textsc{Actions}_P(s)$$

Results: the set of outcomes of actions applied to different physical states.

For deterministic actions,

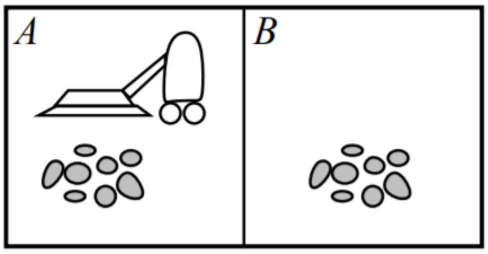
$$b' = \textsc{Result}(b, a) = \{s' : s' = \textsc{Result}_P(s, a) \text{ for } s \in b\}$$

For non-deterministic actions,

$$b' = \textsc{Result}(b, a) = \bigcup_{s \in b} \textsc{Results}_P(s, a)$$

Goal test: b satisfies the goal test if **all** physical states in b do

Example:



Two physical states: the agent is in room A (s_1) or in room B (s_2)

Actions: $Left$, $Right$

If the agent starts in the belief state $b = \{s_1, s_2\}$ and performs action $Right$, the resulting belief state is $b' = \{s_2\}$

Bigger example

Now the agent can also clean dirt, (action $Suck$), and the states differ also on whether the rooms are clean or dirty.

Problem representation

The number of possible belief states is huge

Problem representation gets quite complex (working with sets of states)

Later in the lectures on planning we will see how first-order representation (describing sets of states by properties which hold in all of them) is much easier to work with and more efficient

Searching with observations

Suppose an agent can sense the environment: can tell whether a room is dirty or not (but only the room where it is, not the next one)

$\textsc{Percept}(s)$ returns a percept for the given state, for example $[A, \mathit{Dirty}]$ (if s is the state when the agent is in room A and it is dirty)

Transitions are now 3-step:

1) given belief state b we are in and an action a, compute the next belief state (all physical states we may end up in) $\hat{b}$=$\textsc{Predict}(b, a)$

2) for each physical state in $\textsc{Predict}(b, a)$, compute the percept in that state: $\textsc{Possible-Percepts}(\hat{b})= \{o : o = \textsc{Percept}(s) \text{ and } s \in \hat{b}\}$

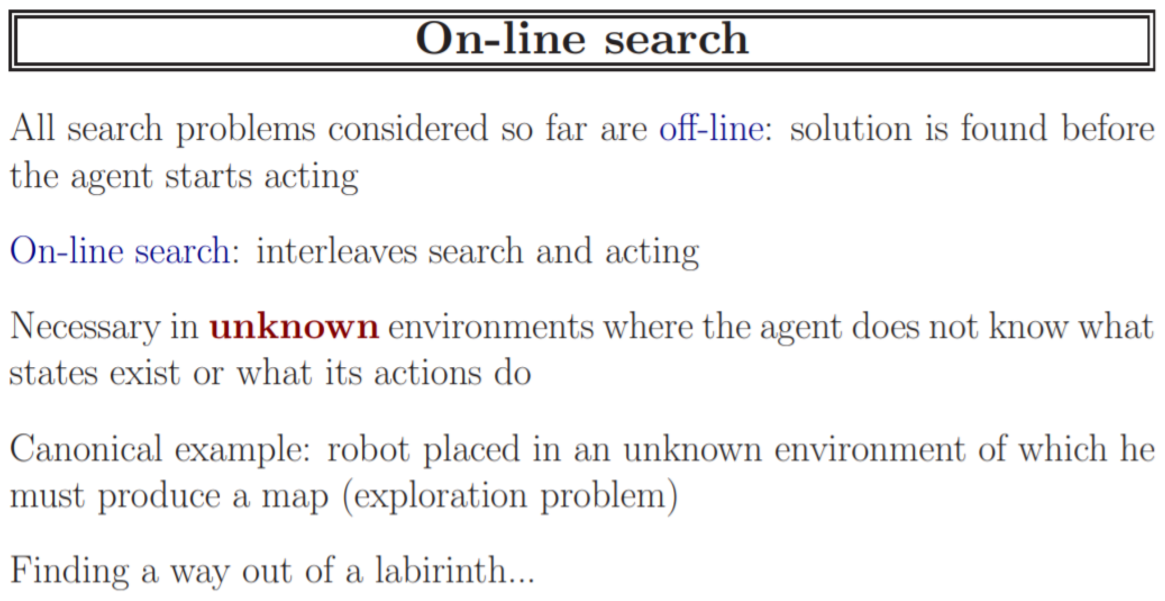
3) for each possible percept o, determine the belief state (after executing a and observing o): $\textsc{Update}(\hat{b}, o)= \{s : o = \textsc{Percept}(s) \text{ and } s \in \hat{b}\}$

Searching with observations contd.

$\textsc{Results}(b, a)$=$\{b_o : b_o = \textsc{Update}(\textsc{Predict}(b, a), o)$

and $o \in \textsc{Possible-Percepts}(\textsc{Predict}(b, a))\}$

Can now apply AND-OR search algorithm



On-line search

All search problems considered so far are off-line: solution is found before the agent starts acting

On-line search: interleaves search and acting

Necessary in **unknown** environments where the agent does not know what states exist or what its actions do

Canonical example: robot placed in an unknown environment of which he must produce a map (exploration problem)

Finding a way out of a labirinth...

2.4 GAMES

competitive environments, in which the agents' goals are in conflict, giving rise GAME to **adversarial search** problems—often known as **games**.

In AI, the most common games are
of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information** (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules.

We begin with a definition of the optimal move and an algorithm for finding it. We
then look at techniques for choosing a good move PRUNING when time is limited. **Pruning** allows us
to ignore portions of the search tree that make no difference to the final choice, and heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete
search.

games such as backgammon include an element of
chance; we also discuss bridge, which includes elements of **imperfect information** because not all cards are visible to each player. Finally, we look at how state-of-the-art game-playing programs fare against human opposition and at directions for future developments.

A game can be formally defined as a kind of search problem with the
following elements:
• S0: The **initial state**, which specifies how the game is set up at the start.
• PLAYER(s): Defines which player has the move in a state.
• ACTIONS(s): Returns the set of legal moves in a state.
• RESULT(s, a): The **transition model**, which defines the result of a move.
• TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
• UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves.

the term **search tree is** for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

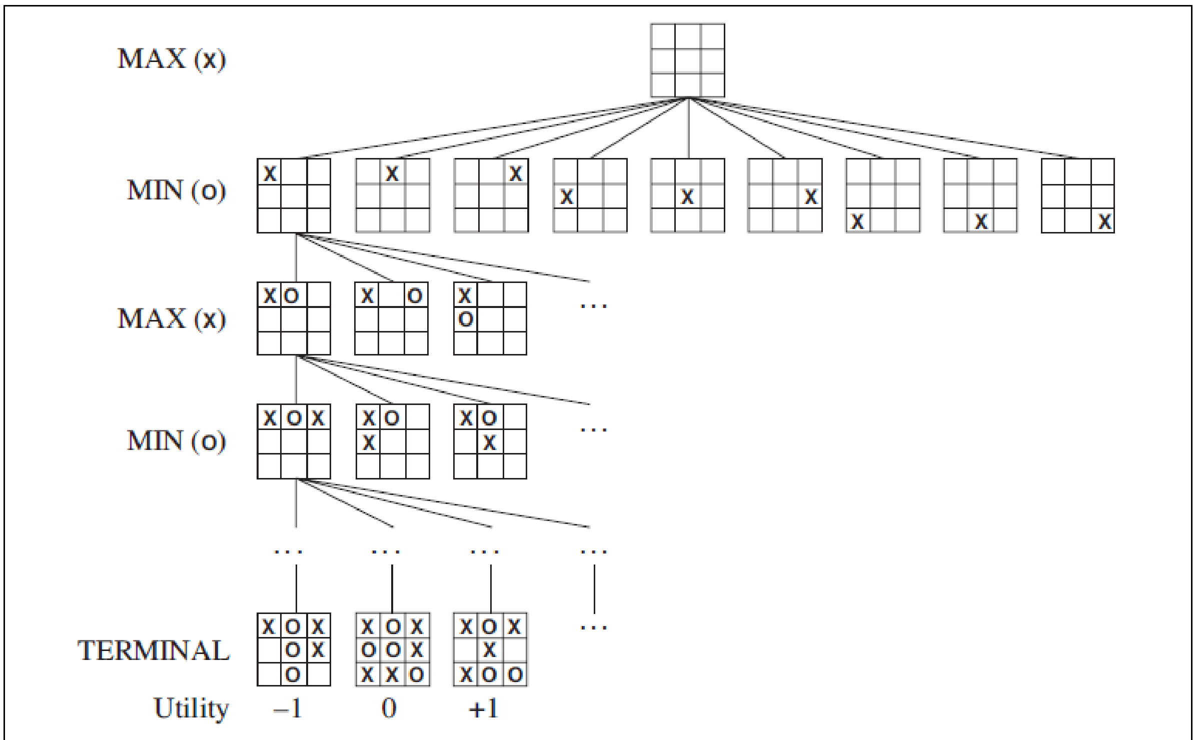


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial

<u>2.5 OPTIMAL DECISION IN GAMES</u>

The minimax algorithm
The **minimax algorithm** (Figure 5.3) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in Figure 5.2, the algorithm first recurses down to the three bottomleft nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backedup value of node B. A similar process gives the backed-up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.
The minimax algorithm performs a complete depth-first exploration of the game tree.
If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O(bm). The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time (see page 87). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

```
function MINIMAX-DECISION(state) returns an action
  return argmax
  a ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ←−∞
  for each a in ACTIONS(state) do
    v ←MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v←∞
  for each a in ACTIONS(state) do
    v ←MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Multiplayer games can allow communication between players.
They can form alliances.
Breaking alliances might have out-of-game payoffs.

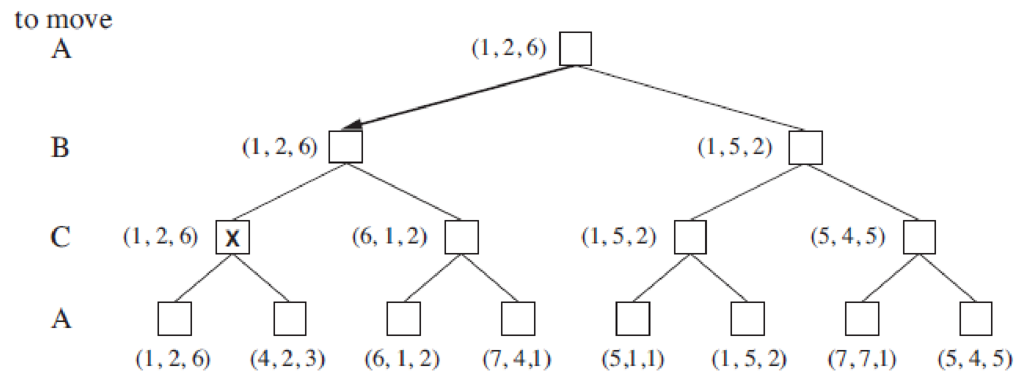


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

2.6 ALPHA BETA PRUNING

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
    a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
    b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

The main condition which required for alpha-beta pruning is:

α>=β

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

```
function ALPHA-BETA-SEARCH(state) returns an action
v ←MAX-VALUE(state, $-\infty, +\infty$)
return the action in ACTIONS(state) with value v
function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← $-\infty$
for each a in ACTIONS(state) do
v ←MAX(v, MIN-VALUE(RESULT(s,a), $\alpha$, $\beta$))
if v $\geq \beta$ then return v
$\alpha$ ←MAX($\alpha$, v)
return v
function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← $+\infty$
for each a in ACTIONS(state) do
v ←MIN(v, MAX-VALUE(RESULT(s,a) , $\alpha$, $\beta$))
if v $\leq \alpha$ then return v
$\beta$ ←MIN($\beta$, v)
return v
```

Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.

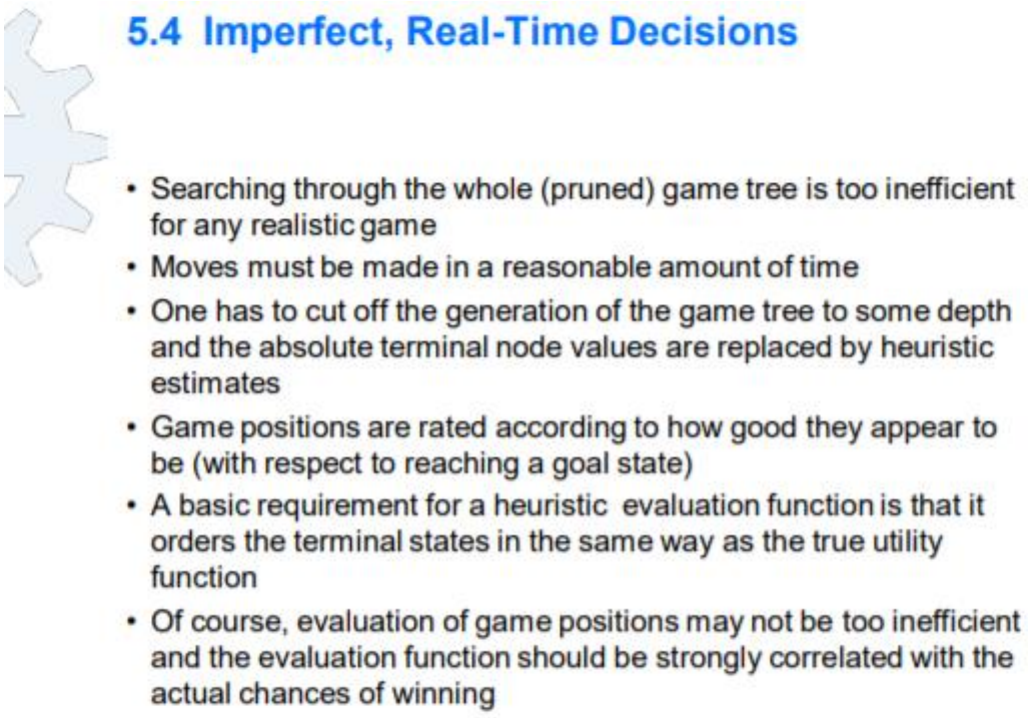
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is O($b^{m/2}$).

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

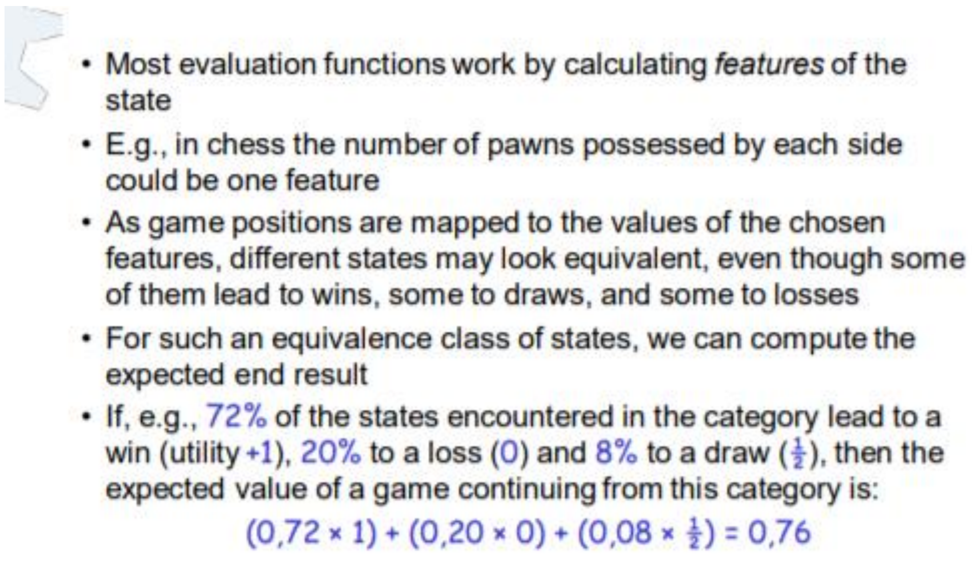
- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

2.7 REAL TIME DECISIONS



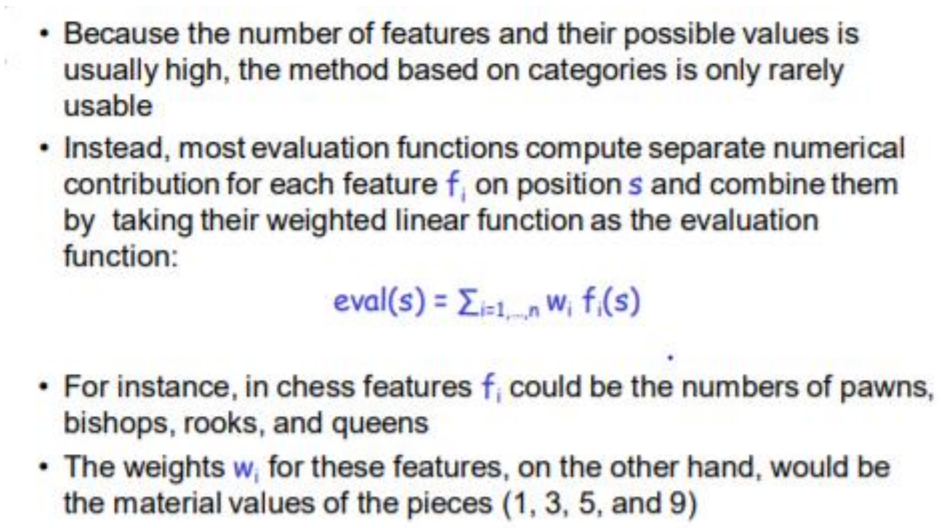
5.4 Imperfect, Real-Time Decisions

- Searching through the whole (pruned) game tree is too inefficient for any realistic game
- Moves must be made in a reasonable amount of time
- One has to cut off the generation of the game tree to some depth and the absolute terminal node values are replaced by heuristic estimates
- Game positions are rated according to how good they appear to be (with respect to reaching a goal state)
- A basic requirement for a heuristic evaluation function is that it orders the terminal states in the same way as the true utility function
- Of course, evaluation of game positions may not be too inefficient and the evaluation function should be strongly correlated with the actual chances of winning



- Most evaluation functions work by calculating *features* of the state
- E.g., in chess the number of pawns possessed by each side could be one feature
- As game positions are mapped to the values of the chosen features, different states may look equivalent, even though some of them lead to wins, some to draws, and some to losses
- For such an equivalence class of states, we can compute the expected end result
- If, e.g., 72% of the states encountered in the category lead to a win (utility +1), 20% to a loss (0) and 8% to a draw ($\frac{1}{2}$), then the expected value of a game continuing from this category is:

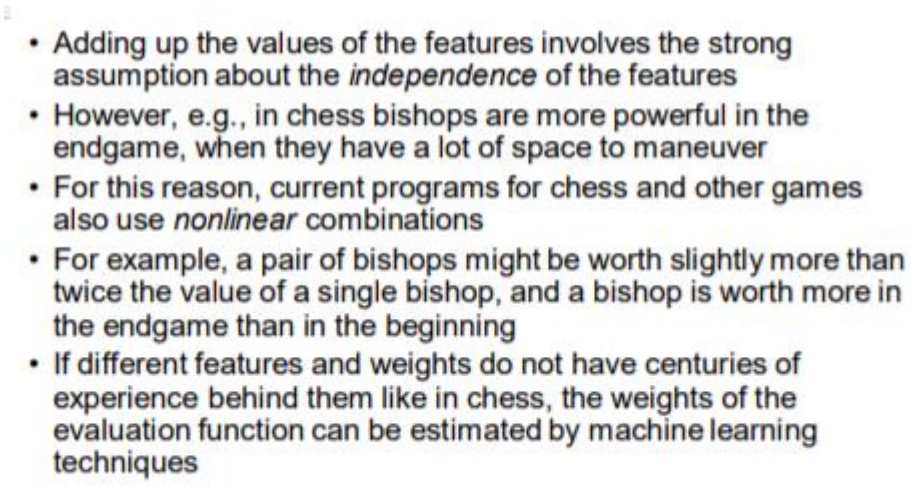
$$(0{,}72 \times 1) + (0{,}20 \times 0) + (0{,}08 \times \tfrac{1}{2}) = 0{,}76$$



- Because the number of features and their possible values is usually high, the method based on categories is only rarely usable
- Instead, most evaluation functions compute separate numerical contribution for each feature f_i on position s and combine them by taking their weighted linear function as the evaluation function:

$$eval(s) = \sum_{i=1,\dots,n} w_i\, f_i(s)$$

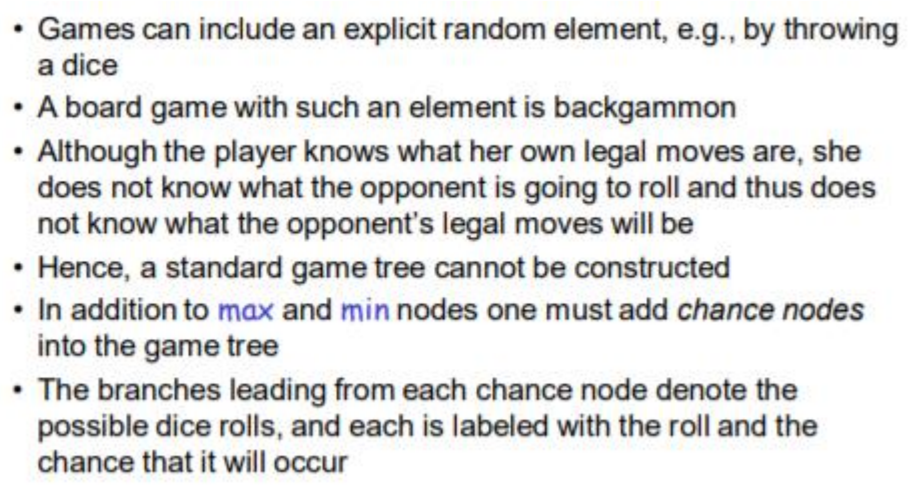
- For instance, in chess features f_i could be the numbers of pawns, bishops, rooks, and queens
- The weights w_i for these features, on the other hand, would be the material values of the pieces (1, 3, 5, and 9)



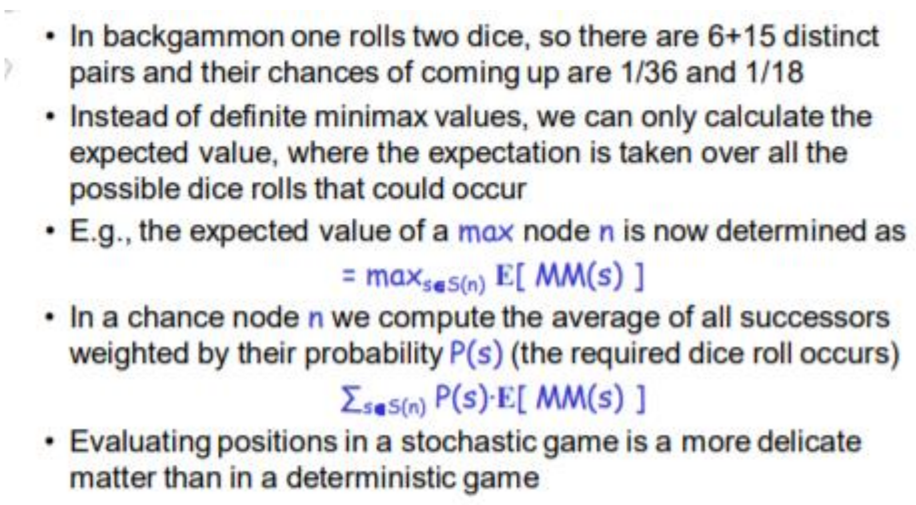
- Adding up the values of the features involves the strong assumption about the *independence* of the features
- However, e.g., in chess bishops are more powerful in the endgame, when they have a lot of space to maneuver
- For this reason, current programs for chess and other games also use *nonlinear* combinations
- For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame than in the beginning
- If different features and weights do not have centuries of experience behind them like in chess, the weights of the evaluation function can be estimated by machine learning techniques

2.8 SCHOTASTIC GAMES

In real life, many unpredictable external events can put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as the throwing of dice. We call these **stochastic games**.



- Games can include an explicit random element, e.g., by throwing a dice
- A board game with such an element is backgammon
- Although the player knows what her own legal moves are, she does not know what the opponent is going to roll and thus does not know what the opponent's legal moves will be
- Hence, a standard game tree cannot be constructed
- In addition to max and min nodes one must add *chance nodes* into the game tree
- The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur



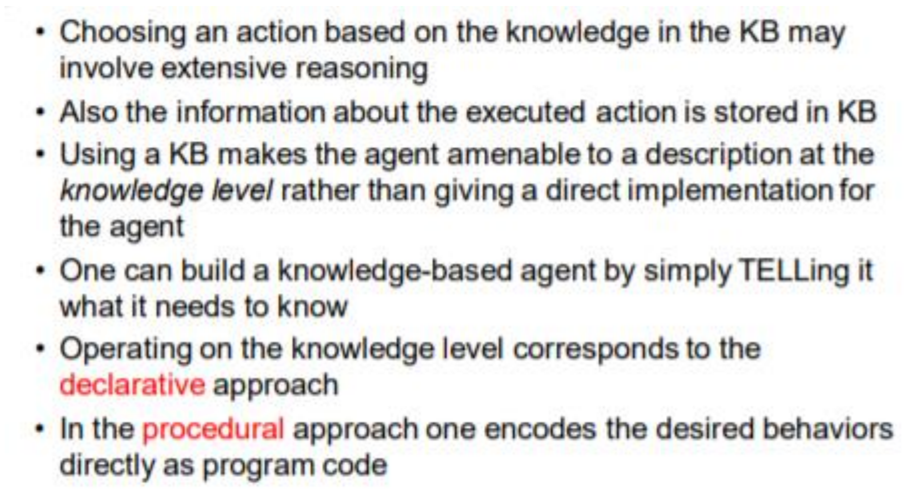
- In backgammon one rolls two dice, so there are 6+15 distinct pairs and their chances of coming up are 1/36 and 1/18
- Instead of definite minimax values, we can only calculate the expected value, where the expectation is taken over all the possible dice rolls that could occur
- E.g., the expected value of a max node n is now determined as

$$= \max_{s \in S(n)} \mathrm{E}[\ MM(s)\ ]$$

- In a chance node n we compute the average of all successors weighted by their probability P(s) (the required dice roll occurs)

$$\sum_{s \in S(n)} P(s) \cdot \mathrm{E}[\ MM(s)\ ]$$

- Evaluating positions in a stochastic game is a more delicate matter than in a deterministic game



- Choosing an action based on the knowledge in the KB may involve extensive reasoning
- Also the information about the executed action is stored in KB
- Using a KB makes the agent amenable to a description at the *knowledge level* rather than giving a direct implementation for the agent
- One can build a knowledge-based agent by simply TELLing it what it needs to know
- Operating on the knowledge level corresponds to the declarative approach
- In the procedural approach one encodes the desired behaviors directly as program code

2.8 INTRODUCTION TO CSP AND INFERENCE???/

Inference

Constraint propagation: Inference in CSPs

A number of inference techniques use the constraints to infer which variable/value pairs are consistent and which are not. These include node, arc, path, and k-consistent.

constraint propagation: Using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

local consistency: If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph.

There are different types of local consistency:

Node consistency

A single variable (a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraint.

We say that a network is node-consistent if every variable in the network is node-consistent.

Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.

X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j).

A network is arc-consistent if every variable is arc-consistent with every other variable.

Arc consistency tightens down the domains (unary constraint) using the arcs (binary constraints).

2.9 BACKTRACKING SEARCH FOR CSP

A problem is commutative COMMUTATIVITY if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only consider a *single* variable at each node in the search tree.

The term **backtracking search** BACKTRACKING is used for a depth-first search that chooses values for
SEARCH
one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 6.5. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is
detected, then BACKTRACK returns failure, causing the previous call to try another value. Part
of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA,NT,Q, ... Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state,

action function, transition model, or goal test.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
return BACKTRACK({ }, csp)
function BACKTRACK(assignment, csp) returns a solution, or failure
if assignment is complete then return assignment
var ←SELECT-UNASSIGNED-VARIABLE(csp)
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
if value is consistent with assignment then
add {var = value} to assignment
inferences ←INFERENCE(csp, var , value)
if inferences _= failure then
add inferences to assignment
result ←BACKTRACK(assignment, csp)
if result _= failure then
return result
remove {var = value} and inferences from assignment
return failure
```

In Chapter 3 we improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge.
Instead, we can add some sophistication to the unspecified functions in Figure 6.5, using them to address the following questions:
1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
2. What inferences should be performed at each step in the search (INFERENCE)?
3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?
The subsections that follow answer each of these questions in turn

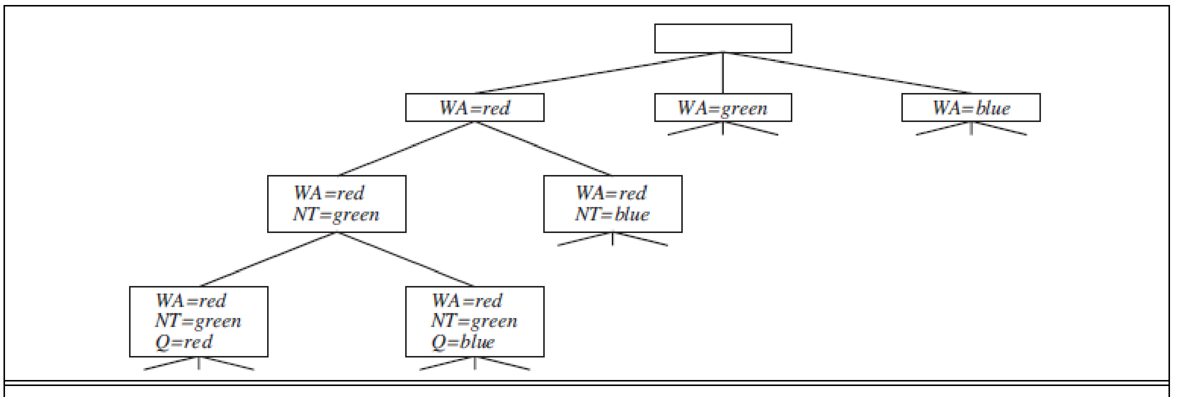


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

2.10 LOCAL SEARCH FOR CSP’s

Local search algorithms (see Section 4.1) turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. For example, in the 8-queens problem (see Figure 4.3), the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column. Typically, the initial guess violates several constraints. The point of local search is to eliminate the violated constraints.

In choosing a new value for a variable, the most obvious heuristic is to select the value MIN-CONFLICTS that results in the minimum number of conflicts with other variables—the **min-conflicts**
Is heuristic.

```
function MIN-CONFLICTS(csp,max steps) returns a solution or failure
inputs: csp, a constraint satisfaction problem
max steps, the number of steps allowed before giving up
current ←an initial complete assignment for csp
for i = 1 to max steps do
if current is a solution for csp then return current
var ←a randomly chosen conflicted variable from csp.VARIABLES
value←the value v for var that minimizes CONFLICTS(var, v, current , csp)
set var =value in current
return failure
```

Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n-queens problem, if you don’t count the initial placement of queens, the run time of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment).

All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective. The landscape of a CSP under the minconflicts
heuristic usually has a series of plateaux. There may be millions of variable assignments that are only one conflict away from a solution. Plateau search—allowing sideways moves to another state with the same score—can help local search find its way off this plateau. This wandering on the plateau can be directed with **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those states. Simulated annealing can also be used to escape from plateaux.
Another technique, called CONSTRAINT **constraint weighting**, can help concentrate the search on the
WEIGHTING
important constraints. Each constraint is given a numeric weight, Wi, initially all 1. At each

step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems.