Artificial Intelligence (CAP 5625)

Definitions of AI:
- Automatically performing tasks that require human intelligence (problem solving, games, discovery, etc.)
- Emulating basic human skills (vision, language understanding, speech recognition, etc.)
- Something that has not been done yet
- Defined by a list of sub-areas: search, learning, language understanding, vision, etc.

Researchers do *not* agree on the definition of AI and its main goals.

Two key problems:
- Perform complex tasks (ultimately, make the human obsolete)
- Perform tasks like human (ultimately, pass the Turing test)

We consider several selected areas.
Search:
- Basic search (Chapter 3)
- Informed search (Chapter 4)
- Games (Chapter 5)
- Planning (Chapters 11 and 12)

Learning:
- Decision trees (Chapter 18)
- Neural networks (Chapter 19)
- Reinforcement learning (Chapter 20)

Basic search (Chapter 3)

Review graph algorithms:
Cormen, Leiserson, and Rivest, *Introduction to algorithms*
First edition: Chapter 23, Sections 25.1 and 25.2
Second edition: Chapter 22, Section 24.3

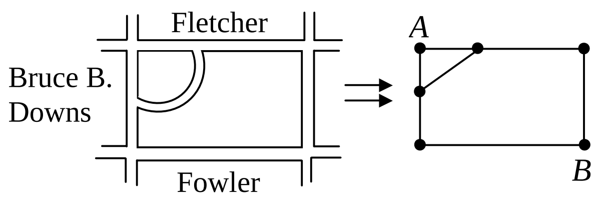
Search is a fundamental method of AI.

Intuition: Look through multiple candidate solutions, until one of them turns out a correct solution. Note that humans also use this method.

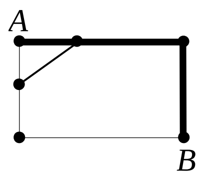
Basic search (a.k.a. blind, uninformed, exhaustive, brute-force): Define a (very large) space of candidate solutions, and implement some systematic procedure for exploring it.

Example 1: Map quest

Given a map, find a route from *A* to *B*.



Start from *A* and use some graph algorithm to look for a path to *B*.



- In the worst case, the time is proportional to the size of the graph: $O(V + E)$
- For a "clever" algorithm, it may be much smaller.
  For example, finding a route in Tampa should not be proportional to the size of a detailed map of the US.

Different search types:

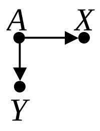
- Search for the shortest path; it is slow, but finds the best solution
- Breadth-first (BFS); it is faster, but may not find the best solution
  (unless all edges are of the same length)
- Depth-first (DFS); it is usually faster than BFS, but may produce a very lengthy route

If the map is not stored as a graph, then we need to construct an explicit graph; however, we may not need to construct the complete graph. For example, if we search for a path in Tampa, we do not need the complete map of the US.

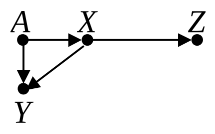
Idea: Build a graph in the process of search. For example, suppose that a function *roads* inputs an intersection and outputs adjacent intersections with distances:

roads(*A*) = {*X* (0.5 miles), *Y* (0.3 miles)}

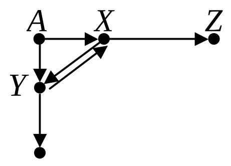
Start at *A*, call *roads*(*A*).



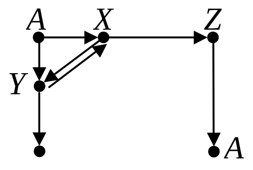
Call *roads*(*X*).



Call *roads*(*Y*).

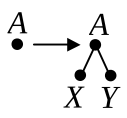


Call *roads*(*Z*).

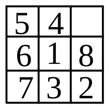


We have found a route from *A* to *B* without building the complete graph of the US (or Tampa).

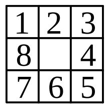
When we call *roads*(*A*) and get *X* and *Y*, we say that we *expand* node *A* and *create* nodes *X* and *Y*.



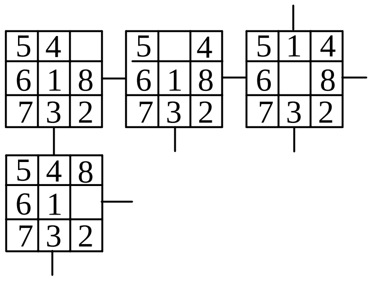
Example 2: Eight puzzle



Eight square tiles, numbered 1 through 8, and an empty square. We can slide adjacent tiles into the empty square. In a given position, there are at most four different moves.
The goal is to get to a specific final position.
The standard final position is:

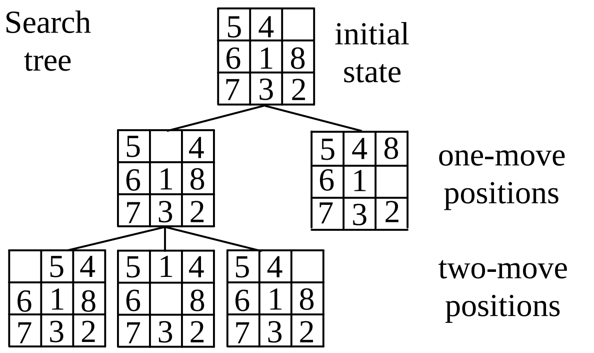


Construct a graph:
- Positions are nodes
- Moves are edges

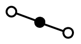
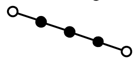
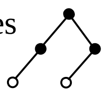


Terminology:
- Position is a *(world) state*
- Move is a *transition* or *operator*
- Set of all possible states with transitions between them is a *state space* (may be huge)

Start from the initial state and keep expanding the space until reaching the goal state.

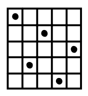
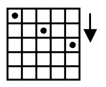


Problem: The search procedure creates duplicate states; it does not violate correctness, but takes extra time.

We can:
- Detect two-move loops (easy)

- Detect longer loops (harder)

- Detect duplicates in different branches (requires hashing)


Example 3: n-queen problem

Put *n* queens on an $n \times n$ chess board, in such a way that they do not attack each other.
Initial state: No queens on the board
Move: Add one queen
Stupid search: Consider all possible placements of *n* queens.
Smarter: Put first queen in the first row, second in the second row, etc.
Even smarter: Each queen takes a separate column; only *n*! possibilities.

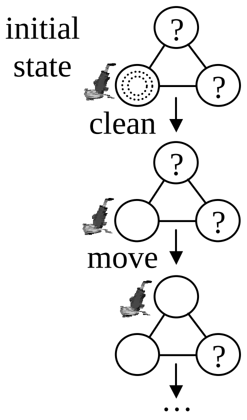



Example 4: House cleaning

We need to clean a house, where some places are dirty.
We use a vacuum and can perform the following actions:

- Move to an adjacent location
- Clean the current location

We see dirt only at our current location.



Since we do not know whether unvisited locations are dirty, each node in the search space corresponds to a *set* of world states.

Although we do not know which locations are dirty, we can develop a plan that is guaranteed to work. It is more effective to develop a tentative plan and keep re-planning as we explore the world.

- Planning for all possible situations is called a *contingency problem*
- Re-planning as we explore the world is called *interleaving* search and execution

Summary: A search problem has an initial state, goal description, and transition function. The task is to find a sequence of actions that leads from the initial state to one of the states that satisfy the goal description.

When writing a program for solving search problems, we consider several factors:

- Speed of finding the solution (time complexity)
- Memory requirements (space complexity)
- Length of a solution (that is, the number of transitions)
- Quality of a solution (usually measured as the total "cost" of moves, for example, the total length of a car route)
- Quality of a final state (some goal states that may be better than others)

Desirable properties of search algorithms:

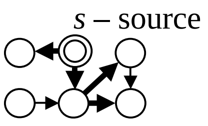
- High speed and low memory
- Completeness: Guarantee to find a solution if one exists
- Optimality (a.k.a. admissibility): Guarantee to find an optimal solution

Real-life problems solvable by search:
Map quest, robot navigation, logistics transportation,
assembly sequence, scheduling, VLSI layout.

When solving a real-life problem, we need to "formulate" it, that is, define states and transitions. We "abstract away" most features of the real world. Different applications require different levels of abstraction. For example, we may consider a map on the level of cities (for airplanes), intersections (for cars), or one-foot squares (for robot navigation).
Converting real world into a finite state space is often a research problem (for example, robot navigation).

Basic search algorithm (theory, not AI)

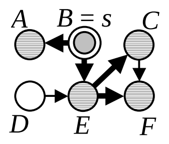
A search algorithm explores a directed graph, starting from a given node. The “frontier” of the exploration is often called the *fringe*.



Mark visited nodes:
- Initially, all nodes are white (“unexplored”)
- When a node *u* is first discovered, it becomes gray (“fringe”)
- After the adjacent nodes of *u* are processed, *u* becomes black (“explored”)

For each node *u*, we store:
color[*u*] – white, gray, or black
parent[*u*] – parent of *u* in the search tree
The gray nodes are stored in some fringe structure. When *u* is discovered, it is painted gray and added to the fringe. When the adjacent nodes of *u* are processed, it is painted black and removed from the fringe.

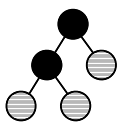
```
SEARCH(Graph, s)
for each node u in Graph
     do color[u] ← WHITE
          parent[u] ← NIL
color[s] ← GRAY
fringe ← {s}
while fringe is not empty
      do u ← “some fringe node”
          for each v adjacent to u
               do if color[u] = WHITE
                      then color[v] ← GRAY
                              parent[v] ← u
                              fringe ← fringe ∪ {v}
          color[u] ← BLACK
          fringe ← fringe – {u}
```



Fringe: *B*
AE
E
CF
F

Note that the algorithm does not distinguish between gray and black nodes; the difference is only for the explanation.

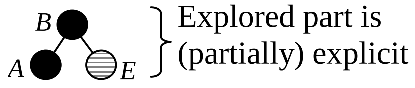
AI terminology:
- Graph is the state space or search space
- Parent pointers form the search tree
- Black nodes are expanded
- Gray nodes are newly created; they are leaves of the search tree



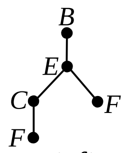
- White nodes are not explicitly represented; they are "somewhere" in the state space

AI is different from theory:

- A graph is huge and there is no explicit representation; it is expanded in the process of search



- The search algorithm may create many copies of the same node and add them to the search tree

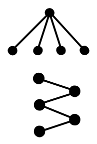


- The search usually terminates after finding a node that satisfies certain conditions, without exploring the whole graph. For example it terminates after finding some solution to the *n*-queen problem

- The search may proceed from several nodes in parallel

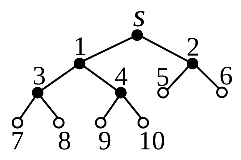
The search time depends on three main factors:

- *Branching factor b*, that is, the (mean) number of children of a node in the search tree
- *Search depth d*, that is, the length of the path from the source node to the goal node
- Time per node



Breadth-first search

The fringe in the search algorithm is a queue (FIFO); thus, the oldest fringe node is expanded first. If the solution is at depth d, the algorithm explores all nodes with depth $< d$ and some nodes with depth d.



Number of "$< d$" nodes: $1 + b + b^2 + \ldots + b^{d-1} = \dfrac{b^d - 1}{b - 1}$

Total number of "d" nodes: b^d

Number of explored nodes: $\dfrac{b^d - 1}{b - 1} \le N \le \dfrac{b^d - 1}{b - 1} + b^d$; thus, $N = \Theta(b^d)$.

For example, consider eighteen-move Rubic's cube solution:
$B = 12$, $d = 18$, $b^d = 12^{18} = 2.7 \cdot 10^{19}$ (a lot)

The search takes not only $\Theta(b^d)$ time, but also $\Theta(b^d)$ memory, which is even worse.

Advantages:

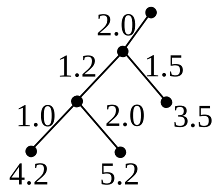
- Easy to implement
- Guaranteed to find a solution ("complete")

Drawbacks:

- Impractically slow
- Prohibitive memory requirements

Best-first (a.k.a. uniform-cost) search

For each node, determine its *cost*, that is, the distance from the root, computed as the sum of edge costs. The cheapest fringe node is expanded first. The fringe is a priority queue, which allows fast identification and deletion of the cheapest node. In algorithm theory, we call it the “Dijkstra” algorithm.



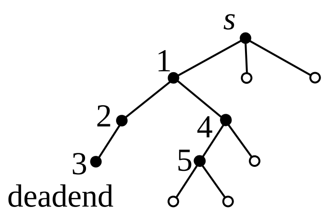
Number of nodes is about the same as in BFS; per-node time is larger since extracting the cheapest node from the fringe takes nonconstant time.

Advantages
- Guaranteed to find a solution (“complete”)
- Finds the best (i.e. cheapest) solution (“optimal,” a.k.a. “admissible”)

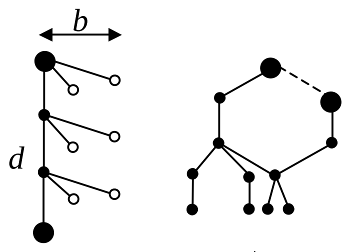
Drawbacks
- Even slower than BFS
- Same memory requirements as BFS

Depth-first search

The fringe is a stack (LIFO), and the newest fringe node is expanded first. If the algorithm runs into a deadend, it backtracks and considers a different branch.



The running time depends on many factors, including luck. For a depth-d solution, the algorithm may create from $b \cdot d$ to many more than b^d nodes. In particular, it can miss an “easy” solution and go into a deep branch.



best case worst case

The memory requirements are low: if the length of the expanded path is d, then the memory is $\Theta(b \cdot d)$.

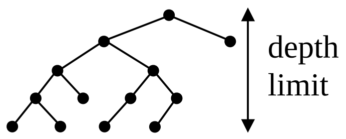
This strategy is good if
- there are a lot of solutions deep in the space or
- we have some guidance toward a solution

Advantages
- Easy to implement
- Low memory requirements
- Often faster than BFS

Drawbacks
- Low-quality solutions
- Usually not complete, that is, may fail on a solvable problem

Depth-limited search

Depth-first search that backtracks upon reaching a certain depth. Alternatively, it can backtrack upon reaching a cost limit.

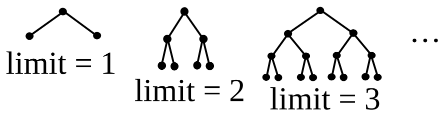


Advantages
- Easy to implement
- Low memory requirements
- Guaranteed to find a solution within depth limit (if any)
- Guaranteed to terminate

Drawbacks
- Low-quality solutions
- Does not find a solution if it is beyond depth limit

Iterative deepening

Repeat depth-limited search multiple times, with successively increasing depth limit.

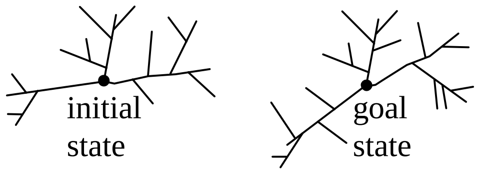


This search is equivalent to BFS, but it needs little memory.

Number of visited nodes $\Theta(b) + \Theta(b^2) + \Theta(b^3) + \ldots + \Theta(b^d) = \Theta(\frac{b^d + 1}{b - 1}) = \Theta(b^d)$;

thus, the running time is almost the same as BFS.
To summarize, it is an "improved BFS."
Alternatively, it can iterate on cost limit, which makes it near-equivalent to BestFS.

Bidirectional search



Start from both ends and continue search until meeting in the middle. If the standard search takes $\Theta(b^d)$ time, then the bidirectional search may take as little as $\Theta(2 \cdot b^{d/2}) = \Theta(b^{d/2})$. The memory is $\Theta(b^{d/2})$, which is often impractically high requirement.

Main problems:
- Need to know the exact goal state
- Need to have an efficient mechanism for comparing fringes

Summary

- Uninformed search works only for small problems
- Running time is the main bottleneck for large problems
- Different search techniques are effective for different problems; there is no "universally effective" search

Informed search (Chapter 4)

Idea: Consider likely solutions before unlikely ones; for example:
- If a key is lost, look in the pocket before looking in the microwave
- When trying to reach the downtown, move in the direction of the largest building

The hardest part is to find functions for evaluating candidate solutions. The second hardest part is to use them wisely, as even the best functions are only estimates. For example, the “move toward the largest building” function may lead to a deadend.

The functions for guiding the search are called *heuristic functions,* and a search algorithm that uses them is called a *heuristic search*. These functions estimate the *distance to a goal*.

Examples:
- The straight distance on the map is an estimate of the driving distance
- The number of misplaced tiles in the eight puzzle is an estimate of the solution length

General idea: Search in the direction of reducing the estimate function.

Greedy search

For each fringe node, estimate its distance to the goal, and expand the closest node. The distance estimate is called a *heuristic function*. This search is similar to BFS and BestFS, but it usually takes less time.

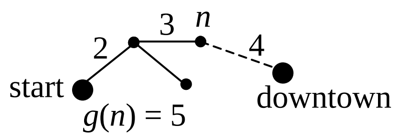
Advantages:
- Usually faster than uniformed search
- Solution quality is OK (but not the best)
- Usually complete (but not always)

Drawback:
- Prohibitive memory requirements

Famous A (impractical foundations of AI)*

For each node n, determine
- distance from the initial state, denoted $g(n)$
- estimated distance to the goal, denoted $h(n)$

The estimated cost of a path *through* the node is
$f(n) = g(n) + h(n)$.



$g(n) = 5$
$h(n) = 4$
Estimated path cost:
$f(n) = 5 + 4 = 9$

For example, consider Map quest.
- $g(n)$ is the shortest known path to location n
- $h(n)$ is the straight distance to the goal, which is an estimate of the path length

Expand the fringe node with the smallest $f(n)$, that is, the shortest path through it.

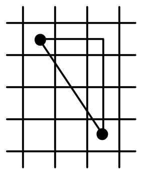
The strategy helps to search for a near-optimal solution; however, if *h* is not an accurate estimate, then the solution may not be optimal.

If *h* always *underestimates* the distance, then it is called an *admissible* heuristic function. For example, the straight-line distance always underestimates the shortest path. The search algorithm that expands the smallest-*f* node *and* uses an admissible *h* is called A*.

If the branching is finite and the edge costs have a positive lower bound, then A* is complete and optimal (a.k.a. admissible). With several semi-reasonable assumptions, it is proved to be the most efficient among optimal algorithms.

A good heuristic function should underestimate the actual distance to the goal, but be close to the actual distance. The more accurate the heuristic, the smaller the number of expanded nodes. At the same time, it must be efficiently computable.

For example, suppose that we are searching for a route in Tampa (or Manhattan), where all streets are "vertical" or "horizontal."

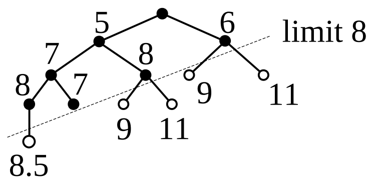


- Good heuristic: Straight distance, easy to compute
- Very good heuristic: Manhattan distance, easy to compute
- Perfect but impractical heuristic: Exact path length, hard to compute

Advantages:
- Complete and admissible
- Faster than uniformed search

Drawback:
- Prohibitive memory requirements

Iterative deepening A* (a.k.a. IDA*)



IDA* is similar to the iterative deepening algorithm, with the depth bound determined by *f*-cost. It searches all nodes up to some *f*-cost limit, and then increases the limit, repeating this increase until the solution is found. It increases the depth limit to the minimal *f*-cost of nodes outside the explored region. In our example, we increase the *f*-cost limit to 8.5. This algorithm is similar to A*, but it requires little space.

If the increments of depth limit are small (e.g. 8.0, 8.5, 8.8, 8.9, …), IDA* may re-expand the space too many times. In the worst case, it may re-expand the space for each new node. We may avoid this problem by increasing the bound in larger steps, but then we lose admissibility. For example, suppose that we fix some ε cost and increase the *f*-cost limit to "minimal *f*-cost outside the explored region" + ε. Then, we get a near-optimal solution that is within ε from the optimal.

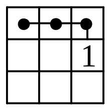
SMA*: Combination of A* and IDA*, which uses *all* available memory to improve efficiency; it keeps part of the search tree in memory, and re-expands the other parts.

Constructing heuristics

We need heuristic estimates that are close to actual path costs; good heuristics are crucial for the efficiency.

Some ideas:

- Count the number of features to be corrected; for example:
  - Number of out-of-place pieces in the eight puzzle (estimate of the solution length)
  - Number of known bugs in the code (estimate of the debugging time)
- Measure “distances;” for example:
  - Straight distances on a map (estimate of the route length)
  - Distance of each tile in the eight puzzle from its proper place (the sum of these distances is an estimate of the solution length)



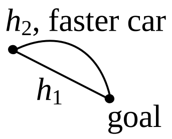
distance 3

- Number of moves in a “relaxed” problem, that is, a simplified problem with additional transitions; for example:
  - Eight puzzle where tiles can move over each other; the solution length in this simple puzzle is an estimate for the actual solution length
  - City map where we disregard traffic lights
- Pre-computed distances for some states; for example:
  - Pre-compute all states of the eight puzzle that are within three moves from the goal
  - Pre-compute distances from USF to major intersections

If we have several admissible heuristics, h_1, h_2, …, h_k, we can combine them by taking the maximum:

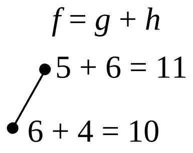
$$h(n) = \max(h_1(n), h_2(n), \ldots, h_k(n)).$$

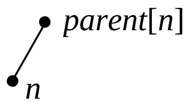
For example, suppose that h_1 is the time of straight-line driving to the goal, and h_2 is the time of actual driving by a faster car, known from past experience. Then, the heuristic underestimate of driving time is $h = \max(h_1, h_2)$.



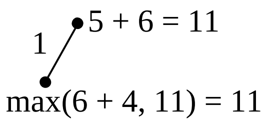
Sometimes, f of a node may be smaller than f of its parent. For example, you drive a mile toward the downtown, and suddenly it appears two miles closer. We say that such a heuristic is not *monotonic*. A better monotonic f can be constructed as follows:

$$f'(n) = \max(f(n), f'(parent[n])).$$





For example:



Example: Admissible search for Rubik’s cube (Korf).
The cube consists of two “sub-puzzles”: Corners and sides. The “sides” state space has $12! \cdot 2^{12} / 2$ nodes, and the “corners” space has $8! \cdot 3^8 / 6$ nodes. We can construct an explicit graph for each space, and compute exact distance to the goal for each node, which gives two admissible heuristics:

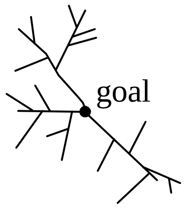
- Distance in the “sides” space, denoted h_1

- Distance in the "corners" space, denoted h_2

In addition, we can compute all states within certain number of moves from the goal; for example, eight moves give rise to about 11^8 states. Thus, we get a third heuristic, denoted h_3:

- If the state is within the expanded space, h_3 is the exact distance
- If not, $h_3 = 9$

Combined heuristic: $h = \max(h_1, h_2, h_3)$; run IDA* with it.

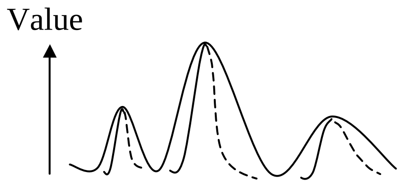


Iterative improvement

Suppose that we search for a state rather than a path (e.g. 8-queen problem, VLSI design), and need to find a sufficiently good state. We have a *Value* function that evaluates the quality of a state, and we need to find a state such that *Value*(*state*) ≥ *Threshold*, or *Value*(*state*) is "as large as possible."

For example, consider the 8-queen problem.
Value(*state*) = – "number of attacked queens"
Threshold = 0

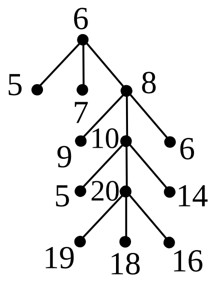
Intuition: Search space is a landscape, and *Value* is the altitude of a state. We need to find a high point.



Hill-climbing

- Start in a random place
- Go uphill until reaching a hilltop
- Repeat if necessary (when the hilltop is too low)

If the highest child is the same as the node, keep going.



- Always go to the child with the highest value
- If all children are strictly smaller than the current node, then stop.

Example:

- Start with an inefficient program for the homework
- Keep improving as much as you can
- If the result is not good enough, start all over

Unlike greedy search, hill-climbing does not keep the search tree.

Problems:

- *Local maxima;* random re-starts may or may not help
- *Plateaux* may cause a random walk
- *Ridges* may cause oscillation without progress, even if they slope up to a hilltop

Advantages:

- Simple
- Little memory

Drawback:

- Limited applicability

Simulated annealing

This algorithm is a modification of hill-climbing based on an analogy with random fluctuations in a cooling liquid. If the cooling process is slow, the liquid freezes in a low-energy state.

We allow random steps in the beginning of hill-climbing, and gradually reduce their probability. The algorithm uses simulated temperature, which gradually decreases.

```
ANNEALING
current ← initial state
T ← initial temperature
while T > 0
    do next ← random successor of current
       Δ ← Value(next) – Value(current)
       if Δ > 0
           then current ← next
           else with probability $e^{\Delta/T}$, current ← next
       T ← decreased T
```

Slow decrease of *T* gives better results, but it requires longer search. The algorithm is more effective than hill-climbing, but its applicability is also limited.

Games (Chapter 5)

History:

- Checkers:
  - Samuel's program, 1952
  - Schaeffer's Chinook: lost to Tinsley in 1992; became the world champion in 1994
- Chess:
  - Condon and Tompson's Belle reached the master level in 1982
  - Campbell and Hsu's Deep Though / Deep Blue became the world champion in 1997
- Backgammon:
  - Tesauro's TD-Gammon program became one of the top three players in 1992
- Go game is still unsolved

We consider two-player games:

- Players take turns making moves
- Their goals are opposite
- Usually *zero-sum,* that is, the gain of the first player is equal to the loss of the second

Games model a competition without complexities of the real world. When playing a game, the computer cannot construct a complete solution plan since it does not know the opponent's moves. Instead, it uses some *strategy,* which determines its move in each possible position.

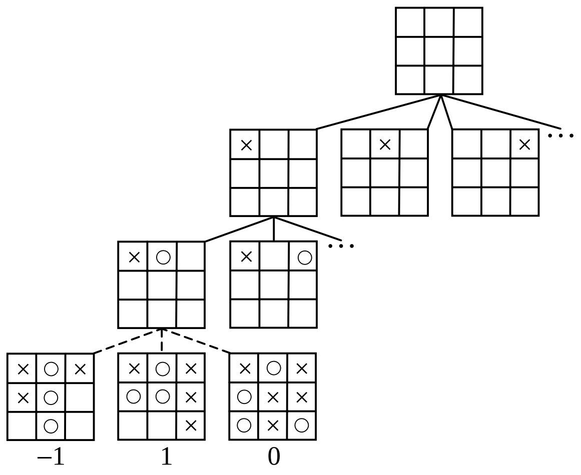
A game is defined by:

- Initial state

- Operators for changing the state (a.k.a. legal moves)
- Termination test for determining end-game states (for example, check-mate in chess)
- Utility function (a.k.a. payoff function) that determines the “quality” of each end-game state for the first player (for example, in chess, +1, 0, or –1; in poker, the final profit)

The first player, called MAX, tries to maximize the utility function; the second player, called MIN, tries to minimize it. If the players have enough time to explore the complete search tree of a game, they can find a perfect strategy, that is, an optimal strategy for playing against an opponent who also has an optimal strategy.

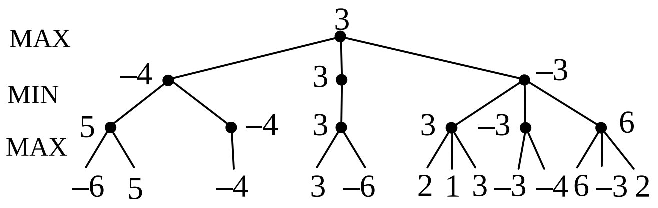
For example, consider the Tic-Tac-Toe tree:



The utility of a node in the search tree is equal to the utility of the leaf that will be reached if both players use a perfect strategy. In other words, a node’s utility is *u* if MAX can ensure that the end-game utility is at least *u,* and MIN can ensure that it is at most *u*.

Given enough time, we can use a bottom-up technique to compute the utility of each node.

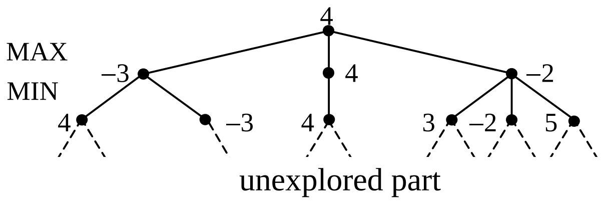
As another example, consider a three-level game tree:



In this example, if MAX uses an optimal strategy, it will win at least 3. If MIN uses an optimal strategy, it will lose at most 3.

The algorithm for computing utilities is called MINIMAX. The knowledge of utilities allows players to select optimal moves.

In practice, we cannot expand a complete tree. Instead, we “cut” the search at some depth and apply an evaluation function to estimate the utility. Then, we use MINIMAX to propagate the estimates to the root. If an estimate function is accurate, we usually select the best move, although the computed utility is not exact.



The role of a utility estimate is similar to the role of the *h*-function in A*: accurate estimates are essential. Estimates are usually weighted linear functions of the form $w_1 \cdot f_1 + w_2 \cdot f_2 + \ldots + w_n \cdot f_n$, where $f_1, \ldots, f_n$ are features of a state, and $w_1, \ldots, w_n$ are weights representing their importance.

features	weight
queen	9
rook	5
…	…
pawn	1
castling	0.5
…	…

A good evaluation function should give an average estimate for all positions with identical features $f_1, \ldots, f_n$. Good feature sets should identify positions that have close utilities. For example, features of a chess position include the available pieces and their interactions. Features must be selected by a human expert; weights can be learned by the computer.

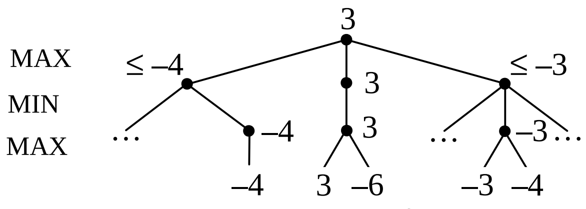
Problems:

- Evaluations should be applied only to *quiescent* positions, where features cannot significantly change in the next move; for example, opponent cannot capture the queen
- The algorithm should account for the *horizon problem,* that is, threatening moves by the opponent that can be delayed but not avoided

Alpha-Beta Search

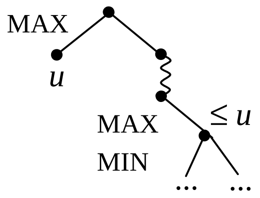
Sometimes, we can discard a node without considering all its children.

Idea: If we already know that some move is worse than another move, we do not need to know how bad it actually is. For example, if we know that some chess move will cause *at least* the loss of a rook, we skip it and consider another move.



skip the rest of the search

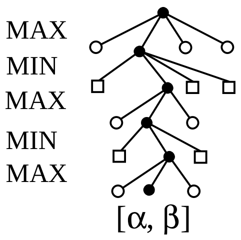
General picture:



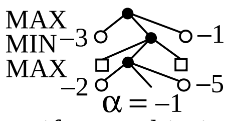
Skip this move, since there is a better alternative.

Same of MIN, with the opposite inequality.

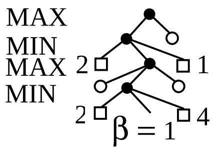
During the depth-first search, we keep two values:
α is the best alternative for MAX
β is the best alternative for MIN



α is the maximum of all circle nodes with known values
β is the minimum of all square nodes with known values



if ≤ -1, skip it



if ≥ 1, skip it

If MAX's is choice $\leq \alpha$, or MIN's choice is $\geq \beta$, skip it. In other words, if some subtree is outside $[\alpha, \beta]$, skip it.

Game search algorithm:

```
MAX-VALUE(state, α, β)
if CUTOFF(state)  ▹ depth limit?
   then return EVAL(state)  ▹ apply evaluation function
for each successor s of state
   do α ← max(α, MIN-VALUE(s, α, β))
      if α ≥ β  ▹ MIN will not go here
         then return β  ▹ prune the branch
return α
```

```
MIN-VALUE(state, α, β)
if CUTOFF(state)
   then return EVAL(state)
for each successor s of state
   do β ← min(β, MAX-VALUE(s, α, β))
      if β ≤ α
         then return α
return β
```

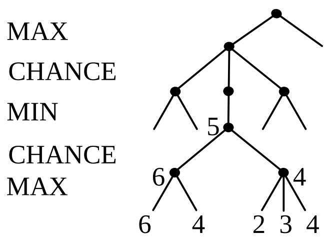
For the top-level call, $\alpha = -\infty$ and $\beta = +\infty$.
Note that $\alpha < \beta$ at all times; the actual value of a node is between α and β

If the program usually guesses an optimal move and considers it first, then it quickly narrows the $[\alpha, \beta]$ interval and skips poor moves. If guessing is almost always correct, the branching factor is reduced from b to $\sqrt{b}$, and the algorithm can search twice deeper in the same amount of time.

To make good guesses:

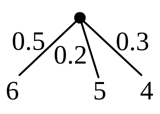
- Use simple heuristics
- Run a preliminary search with smaller depth

Element of chance



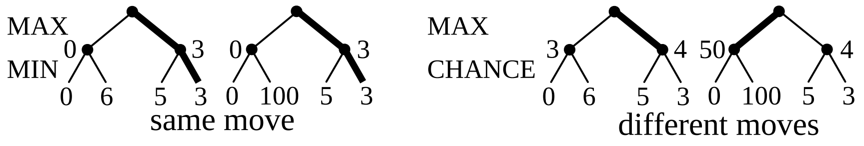
Some games include random choice; for example, backgammon and card games. We add random-choice nodes to the game tree. The utility of a random-choice node is determined as the mean utility of its children.

If different children have different probabilities, we need to compute the weighted mean; for example, $0.5 \cdot 6 + 0.2 \cdot 5 + 0.3 \cdot 4 = 5.2$.



If branching factor of random choice is small, we can apply MINIMAX, but the depth is very low for both computers and human players. The evaluation function must allow averaging; this requirement is different from no-chance games, where the function needs to show only the relative quality of nodes.

Example:



If branching of random choice is large (for example, cards), the MINIMAX algorithm is impractically slow. An alternative technique is reinforcement learning.

Some nontraditional approaches to games:
- Goal-directed reasoning (for example, "how to capture the opponent's queen")
- Recognition of standard situations and appropriate actions (for example, castling, pawn patterns, pieces in the right places)
- Meta-reasoning: Utility of different search branches, choice among different strategies

Planning (Chapter 11 and Prodigy reading)

The task is to find a sequence of steps that leads to a certain desired situation.

Historical motivation: Planning high-level actions of Shakey the Robot (SRI, early 70s).

Other examples:
- Plan a sequence of scientific experiments
- Schedule machine-shop operations
- Plan the delivery of UPS packages
- Plan military operations

Basic search algorithms, such as DFS or IDA* from the initial state, are impractically slow.
We need a representation that allows dividing a goal into subgoals, selecting operators relevant to the goal, and reasoning about properties of operators.

Two main approaches:
- Domain-specific systems, with specialized ad-hoc techniques
- General systems, which are re-usable but inefficient

Some systems are in-between.

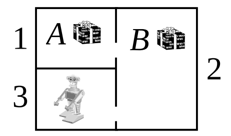
Problems:
- Representing the world state
- Describing actions and goals
- Generating appropriate sequences of actions

Trade-off:

- If a representation is simple, hard to describe the world
- If it is complex, hard to plan efficiently

Simple STRIPS representation

The world is a collection of literals:



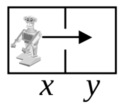
door(1, 2) robot-in(3)
door(2, 3) in(A, 1)
in(B, 2)

Rooms: 1, 2, 3
Boxes: A, B

Assume that we know everything relevant about the world.

Closed-world assumption: If a literal is part of the description, it is true. If not, it is false; for example, door(1, 3) is false. Thus, we do not explicitly represent negations.

An operator is defined by *preconditions* and *effects*, with typed variables.



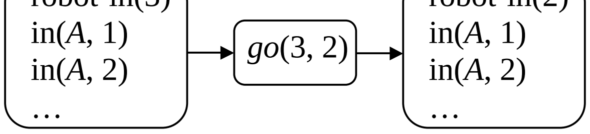
x and y are variables

go(x, y)
x, y: room
Pre: robot-in(x) $\wedge$ door(x, y) $\wedge$ $\neg$locked(x, y)
Eff: del robot-in(x)
add robot-in(y)

push(b, x, y)
b: box
x, y: room
Pre: robot-in(x) $\wedge$ in(b, x) $\wedge$ door(x, y) $\wedge$ $\neg$locked(x, y)
Eff: del robot-in(x)
del in(b, x)
add robot-in(y)
add in(b, y)

The *preconditions* of an operator are a conjunction of predicates and their negations; a *predicate* is a literal with variables. The *effects* are a collection of predicates to be added to or deleted from the world state.

Operator application:



Application means a simulation of an operator execution rather than the execution in the real world. We can apply an operator only if its preconditions are satisfied.

A *goal* is a conjunction of literals and negated literals; for example,

- robot-in(1)
- in(A, 3) $\wedge$ in(B, 3)

Standard extensions

(1) Disjunctions and quantifiers in preconditions and goals; thus, preconditions and goals may be arbitrary logical expressions.

go(x, y)
Pre: robot-in(x) $\wedge$ (door(x, y) $\vee$ door(y, x))
Goals:

- $\exists$ b: in(b, 3), which means "move any box to room 3"
- $\forall$ b: in(b, 3), which means "move all boxes to room 3"

(2) Conditional effects

```
push(b, x, y)
Eff: del…
     del…
     add…
     add…
     if fragile(b) ∧ ¬packed(b)
        then add broken(b)
```

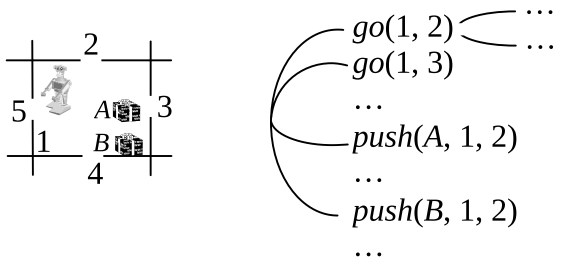
Other extensions

- Probabilistic effects: Changes happen with some probability, which depends on the world state.
- Partial world knowledge: We do not have complete knowledge, and use sensors to learn more.
- Random or hostile changes in the world: Someone else is changing the world state.
- Resource constraints: Operators may consume or produce resources, such as time or electricity; in addition, operators may have limited capacity, such as moving $\leq$5 boxes.

Different planning systems handle different features, with different degrees of efficiency.

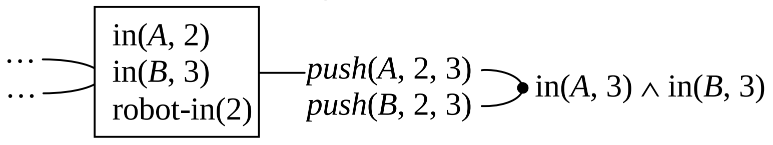
Search strategies

- Forward search from the initial state



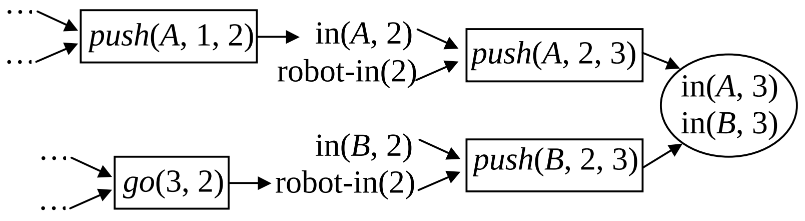
It works only in very small domains. The branching factor is huge since there are a lot of possible actions, most of which are irrelevant.

- Backward search from the goal



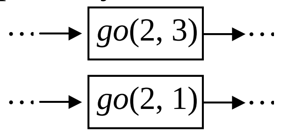
If we consider only goal-related operators, this technique is better than forward search, but the branching factor is still impractically large.

- Goal-directed reasoning

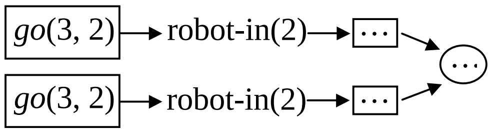


The branching factor is usually small; however, we no longer know the exact world state, which creates problems:

- Different parts of the plan may interfere with each other



- The plan may contain redundant operators

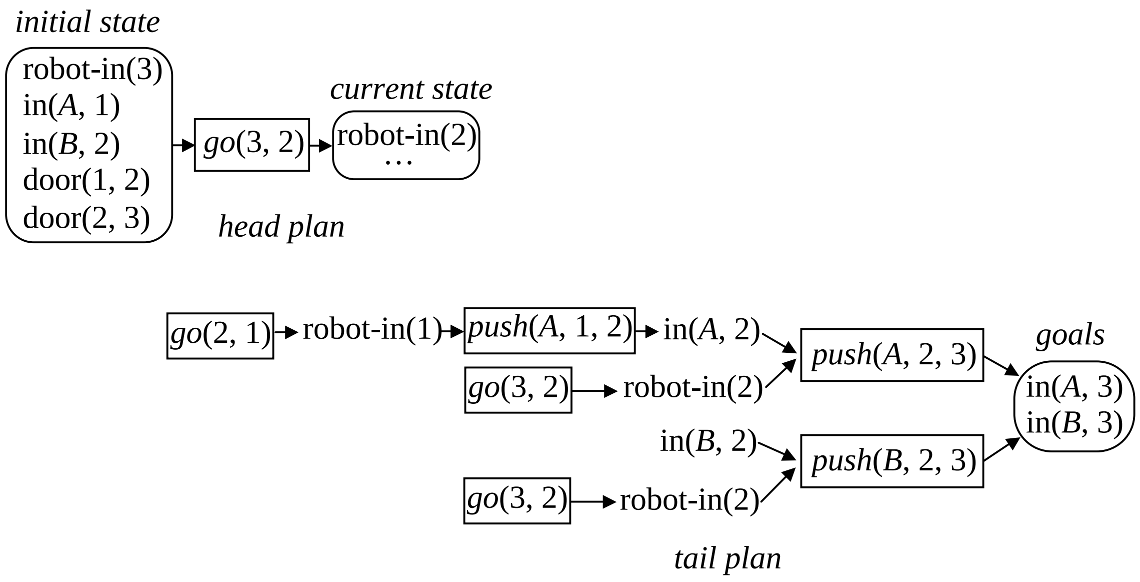


Alternative solutions:

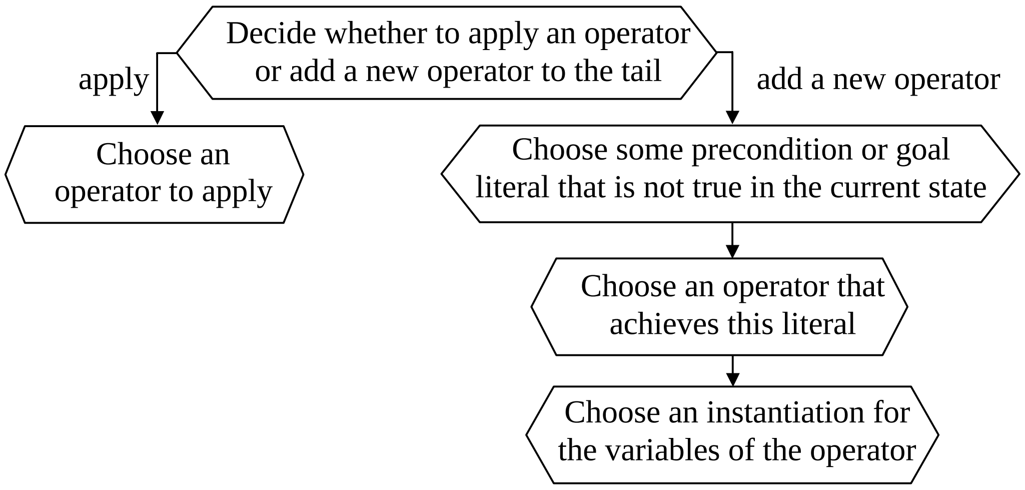
- Simulate the execution of some operators, thus gaining information about the world state (Prodigy)
- Determine all possible interactions of operators, and impose a partial order (UCPop)

The relative performance of these techniques depends on a specific domain; neither is universally better than the other.

Prodigy search

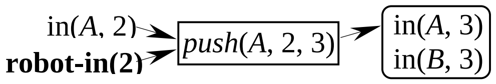


At each step of planning, we need to make several decisions that determine the next modification of the current plan:

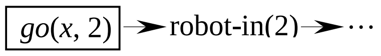

Decide whether to apply an operator or add a new operator to the tail
apply
add a new operator
Choose an operator to apply
Choose some precondition or goal literal that is not true in the current state
Choose an operator that achieves this literal
Choose an instantiation for the variables of the operator

Example:

- Choose a goal



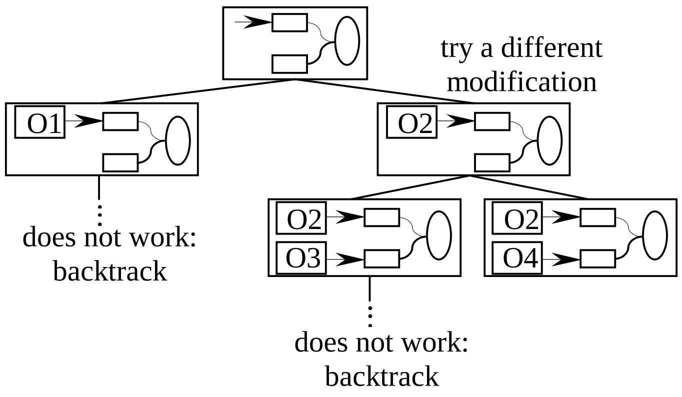
- Choose an operator



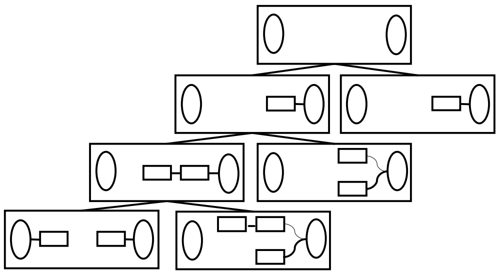
- Choose an instantiation

$\boxed{\textit{go}(3, 2)} \rightarrow \text{robot-in}(2) \rightarrow \cdots$

If the planner makes a wrong decision, it may need to backtrack and consider a different modification of the current plan.



Thus, the planner explores a search space, where each node is a plan. Each possible modification of a plan is a transition to a new node of the search space. The planner begins with an initially empty plan and searches the space of plans produced by modifications until generating a complete plan.



The planner performs a depth-first search with a pre-set depth bound. The user may choose different depth bounds for different domains. In addition, the user may choose a bound on the search time, to avoid long search. The planner uses general heuristic rules to choose promising branches of the search space.

Rule examples:

- Select operators whose preconditions are satisfied or almost satisfied

$\boxed{\textit{go}(3, 2)} \rightarrow \text{robot-in}(2) \rightarrow \cdots$

better than *go*(1, 2) because the robot is in room 3

- Select operators with fewer preconditions

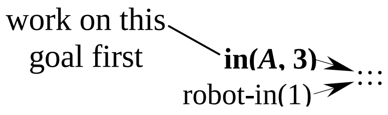
$\boxed{\textit{go}(x, 2)} \rightarrow \text{robot-in}(2) \rightarrow \cdots$

better than *push*(*x*, 2) because *go* has fewer preconditions

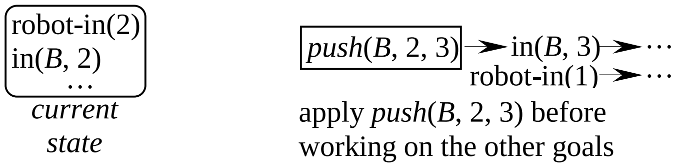
The planner may also use domain-specific if-then rules, called *control rules,* to select promising branches.

Examples:

- Prefer the in(b, x) goal to the robot-in(y) goal



- If some *push* can be applied, then apply it before adding new operators to the tail plan.

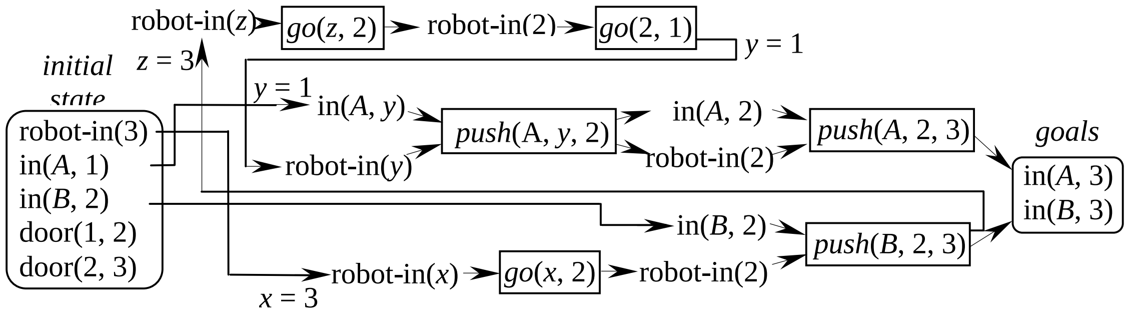


Control rules may be provided by the user of learned automatically.

To reduce the branching factor, we may forbid backtracking over some of the decisions. Different versions of Prodigy backtrack over different decisions. For example, Prodigy2 does not consider different goal choices, and it always chooses the latest goal. Also, it always prefers an operator application to adding a new operator. Prodigy4 does not consider different choices of operators to apply, and it always applies the last added operator. When we forbid some backtracking, we not only reduce branching factor, but also eliminate some solutions, which may have negative effect. Different limitations on backtracking are effective in different domains.

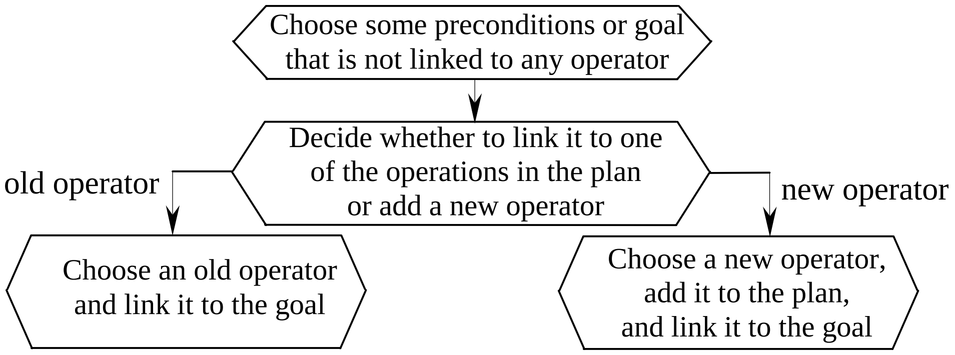
UCPop search

- A plan is a partially ordered sequence of operators
- The initial state is considered the first operator of the plan
- Some variables in the plan may not be instantiated



Result: *go*(3, 2), *push*(B, 2, 3), *go*(3, 2), *go*(2, 1), *push*(A, 1, 2), *push*(A, 2, 3)

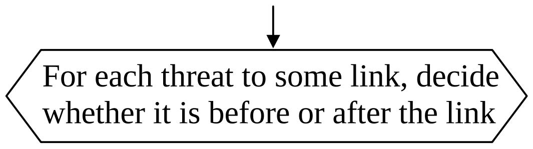
(1) Adding a new link.


Choose some preconditions or goal that is not linked to any operator
Decide whether to link it to one of the operations in the plan or add a new operator
old operator
new operator
Choose an old operator and link it to the goal
Choose a new operator, add it to the plan, and link it to the goal

E.g. $go(x, 2) \xrightarrow[\text{new link}]{} \text{robot-in}(2) \longrightarrow \boxed{push(A, 2, 3)}$

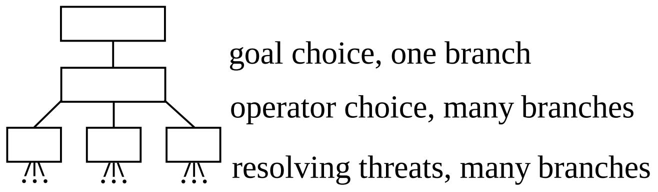
(2) Resolving threats.
A *threat* is an operator that may interfere with a link.
E.g. $\cdots \longrightarrow go(2, 1) \longrightarrow \cdots$
$go(x, 2) \Longrightarrow \text{robot-in}(2) \longrightarrow \boxed{push(A, 2, 3)}$

Every threat must be ordered with respect to the link; that is, if operator C is a threat for $A \to B$, then C must be either before A or after B.


For each threat to some link, decide whether it is before or after the link

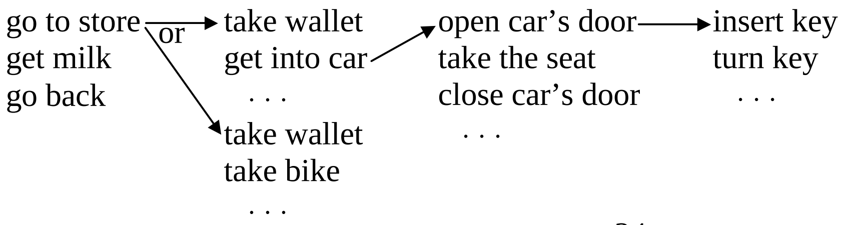
To summarize, the planner makes three main decisions at each step:
1. Choose a precondition or goal
2. Choose an operator to achieve it
3. Choose an ordering to resolve threats

The planner backtracks over Decisions 2 and 3, but it does not backtrack over Decision 1.


goal choice, one branch
operator choice, many branches
resolving threats, many branches

Hierarchical planning (Sections 12.2 and 12.3)

Idea: Construct a high-level plan and then refine it by replacing each operator with more specific operators; intuitively, a high-level operator is like a subroutine. Note that an operator may have many possible replacements.



We continue to replace operators until reaching the level of *primitive* operators, which have no decomposition. The primitive operators should be easily executable; the choice of primitive operators depends on an application.

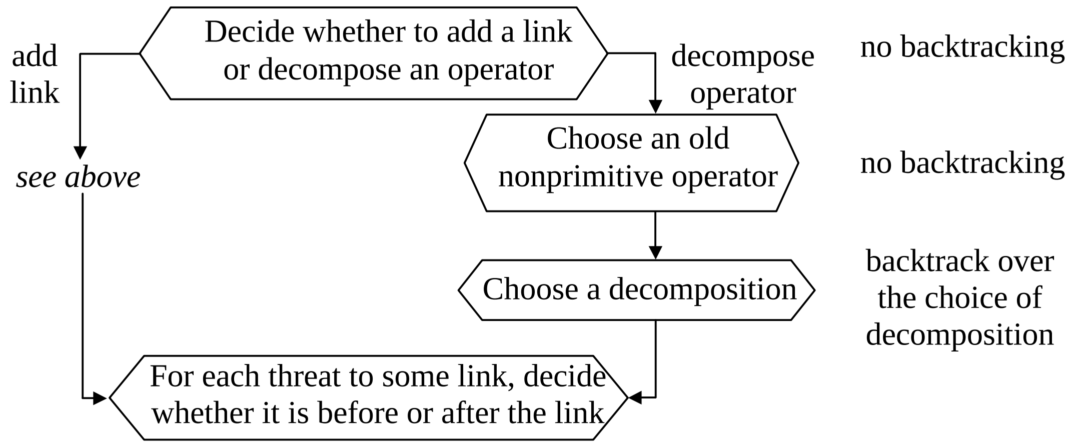
When encoding a domain, we separate operators into primitive and nonprimitive. For each nonprimitive operator, we specify all possible decompositions, which are partially ordered plans with appropriate links.

Example:

$$\begin{tikzcd}
\mathit{push}(b, x, y) \arrow[dr, Rightarrow] \arrow[dddr, Rightarrow] & & & & & \\
& \mathit{go\text{-}to\text{-}box}(b, x) \arrow[r] \arrow[dr] & \mathit{push\text{-}to\text{-}door}(b, x, y) \arrow[r] & \mathit{push\text{-}thru\text{-}door}(b, x, y) \arrow[r] & {} & \\
& & \mathit{notify\text{-}user}(b, x, y) \arrow[urr] & & & \\
& \mathit{go\text{-}to\text{-}box}(b, x) \arrow[r] \arrow[dr] & \mathit{lift\text{-}box}(b, x) \arrow[r] & \mathit{go}(x, y) \arrow[r] & \mathit{put}(b, y) \arrow[r] & {} \\
& & \mathit{notify\text{-}user}(b, x, y) \arrow[urrr] & & & \\
\mathit{go}(x, y) \arrow[dr, Rightarrow] \arrow[ddr, Rightarrow] & & & & & \\
& \cdots & & & & \\
& \cdots & & & &
\end{tikzcd}$$

The preconditions and effects of the replacement plan must match the higher-level operator. That is, if the preconditions of the higher-level operator are satisfied, then the preconditions of all operators in the replacement plan must also be satisfied, and the effects of the replacement plan must be the same as the effects of the higher-level operator. The choice of decompositions is the user’s responsibility.

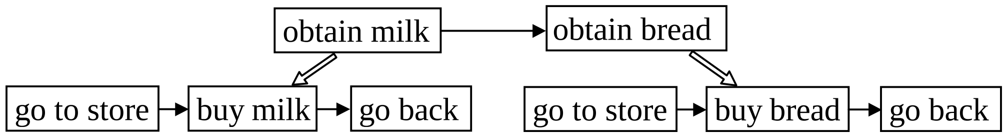
At each step, the planner has to decide between adding a new link and decomposing an old operator. If decomposing, it needs to choose an operator and its decomposition.


Decide whether to add a link or decompose an operator
add link
see above
decompose operator
no backtracking
Choose an old nonprimitive operator
no backtracking
Choose a decomposition
backtrack over the choice of decomposition
For each threat to some link, decide whether it is before or after the link

Note that decomposition of an operator may lead to new threats caused by side effects of the decomposition. Also note that the user may provide an initial high-level plan, and use the planner only for selecting low-level replacements of given high-level operators.

Problems:

- A high-level plan may not have refinements; for example, we may plan to go to a store, and on a lower level find out that the car key is lost, and the bike has a flat tire
- A good low-level plan may not have a corresponding high-level plan; for example:



Solutions:

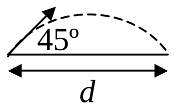
- Provide a decomposition that does not cause these problems, which is the user's responsibility for each domain
- Make sure that the problems do not occur too often, and the hierarchical planning is efficient on average
- Implement techniques for optimizing low-level plans

Learning

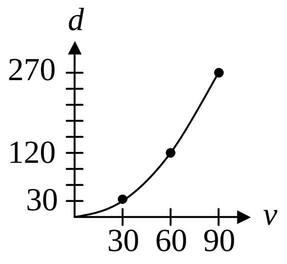
Machine learning is automated collection and analysis of new data. Usual goals:

- learn a new concept or behavior, or
- improve the performance (speed or solution quality).

Simple example: Throw a hard object with a known speed and predict the distance.



Experiments:
30 feet/sec, 30 feet
60 feet/sec, 120 feet
90 feet/sec, 270 feet



Memorization: Store the available data points, and use the experience to generate correct predictions for these known (or similar) cases.

Learning hypothesis: Find a function that fits the available data, and use it to generate predictions for other cases; for example, $d = v^2 / 30$ fits the available data. The learning process is search for an appropriate function; the space of candidate functions is called a *hypothesis space.*

Occam's razor: Prefer simple hypotheses to more complex ones. A hypothesis is considered simpler if it has fewer parameters. For example, $ax + b$ is simple, and ax^b is also simple; $ax^2 + bx + c$ is more complex, and $a_n x^n + a_{n-1} x^{n-1} + ... + a_0$ is even more complex.

Occam's razor prevents *overfitting,* that is, finding a complex function that fits available data but gives wrong predictions for other data.

The choice of hypothesis space and preferences among its elements is called a *learning bias.* It determines the hypotheses (functions) that the learner may consider.

The choice of an appropriate space is essential. If it is too small, it may not have the right functions; for example, the linear regression may be insufficient. If it is too big, the search may take forever.

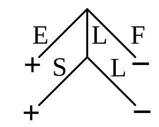
We may view any learned knowledge as a function that maps questions (problems, situations) into appropriate answers; that is, *f*(question) = answer. A *supervised learning* algorithm inputs some questions with appropriate answers, and tries to find a pattern for answering other questions.

Decision trees (Chapter 18)

Example

ELS +	FSR –	ESC **?**
FLR –	LLS +	FSC **?**
LSC –	ESR +	

Decision tree



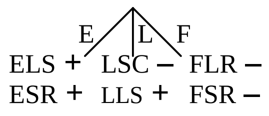
Meaning:

+ need to fill the tank	E empty tank	S short drive	S sunny
– enough gas	L low tank	L long drive	C cloudy
	F full tank		R rainy

A decision tree allows us to *classify* objects, that is, decide which objects belong to a certain class, and which do not. We construct it from a set of *training examples,* which are classified as positive (i.e. belonging to the class) and negative.

Algorithm

- Split examples on the first attribute
- If some nodes get both positive and negative examples, recursively split them on the next attribute
- Continue splitting until each leaf is + or –



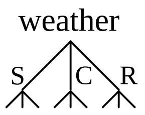
See Figure 18.7 in the textbook for a more detailed algorithm.

High-level view: We need to learn a Boolean function that maps examples into +/–. The tree encodes such a function, which is a learning hypothesis. The hypothesis space includes all possible trees, and the algorithm searches for an appropriate tree.

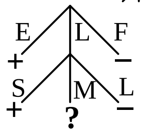
The resulting tree correctly classifies all training examples, but it may give wrong results for new examples. A related philosophical question is how we can ever be sure that a hypothesis derived from a training set will work for other examples.

Problems

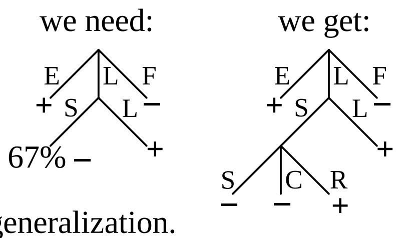
- *Splitting on an inappropriate attribute*
  We may still get a correct tree, but with more levels.
  According to Occam's razor, smaller trees are better.



- *Insufficient training data*
  For instance, suppose that there is a "medium-distance drive" scenario, but no data for LM∗.



- *Insufficient attributes*
  For instance, we may get training examples LSC–, LSC+, and LSC–.
- *Overfitting*
  For instance, suppose that the training examples are LSC–, LSR–, and LSS+. The LSS+ example comes from driving to Canada, where the gas is more expensive, but the destination is not among the attributes.
  The *overfitting* is building a tree that fits the training data "too well"; that is, an accurate fit does not lead to a generalization.

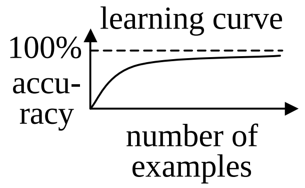


- *Limited expressiveness*
  - Hard to deal with real-valued attributes; usually, we discretize such attributes by defining appropriate ranges
  - Near-impossible to learn parity

To evaluate the quality of a tree:

- Divide the available examples into training set and test set
- Use the training set to build a tree
- Use the test set to evaluate its accuracy, that is, the percentage of correct classifications

To determine the quality of available data, repeat these steps for several random divisions into training and test sets, and take the average. To determine the necessary number of examples, evaluate the accuracy for sets of different sizes.



Selecting informative attributes

Different attributes provide different amount of information about the right answer; for example, the amount of fuel is very informative, and the weather is not at all. Intuitively, if we earn certain money for giving the right answer, then we will pay differently for different attributes. The "value" of information is measured in bits. The prediction of a fair-coin outcome is worth one bit. The prediction of an unfair coin is worth less because we already have a good idea about the outcome. If a "coin" has k sides, with the corresponding probabilities $q_1,...,q_k$, then the value of an accurate prediction is

$$I(q_1,...,q_k) = -\,q_1 \cdot \lg q_1 - q_2 \cdot \lg q_2 - ... - q_k \cdot \lg q_k.$$

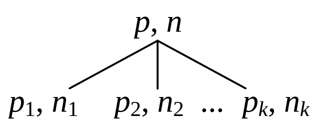
Fair coin: $-\,0.5 \cdot \lg 0.5 - 0.5 \cdot \lg 0.5 = 1$.
Unfair coin: $-\,q \cdot \lg q - (1 - q) \cdot \lg (1 - q)$.
One-sided "coin": $-\,1 \cdot \lg 1 = 0$ (no information at all).
Gas decision: $q = 3 / 6 = 0.5$; $I = 1$.

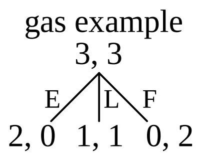
Each attribute gives part of the information; thus, one attribute is worth less than 1. We may determine the exact worth of an attribute by calculating how much information we need *after* the attribute test.

If we have *p* positive and *n* negative examples, the worth of an accurate classification is

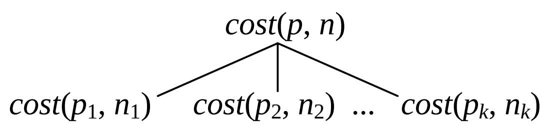
$$cost(p,n) = -\frac{p}{p+n}\cdot \lg\frac{p}{p+n} - \frac{n}{p+n}\cdot \lg\frac{n}{p+n}\,.$$

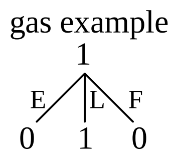
An attribute divides examples into several groups, each with positive and negative examples:





For each group, we need to make a decision, which incurs additional cost:





We need to adjust additional costs by their probabilities:

$$Additional\text{-}Cost = \frac{p_1+n_1}{p+n}\cdot cost(p_1,n_1) + \ldots + \frac{p_k+n_k}{p+n}\cdot cost(p_k,n_k)$$

Gas example:

$$Additional\text{-}Cost = 0.33 \cdot 0 + 0.33 \cdot 1 + 0.33 \cdot 0 = 0.33$$

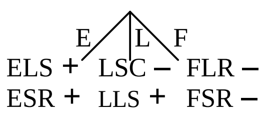
The worth of an attribute is

$$Gain = cost(p,n) - Additional\text{-}Cost$$

$$= cost(p,n) - \frac{p_1+n_1}{p+n}\cdot cost(p_1,n_1) - \ldots - \frac{p_k+n_k}{p+n}\cdot cost(p_k,n_k)$$

Gas example:

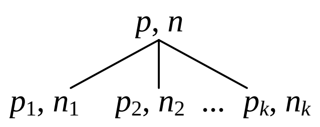
$$Gain = 1 - 0.33 = 0.67$$



The resulting value is called the *information gain* of the attribute. We compute it for each attribute, select the attribute with the highest gain, and put it at the root of the tree. This strategy helps to minimize the depth of the final tree.

When recursively constructing the next level of the tree, we use the corresponding subgroups of examples for computing the information gain. For instance, we need to compute the information gain of the driving distance and weather for the group "LSC–, LLS+."

If the information gain of all attributes is "too low," we should not split at all, thus preventing overfitting. If the gain is low, then the distribution of positive and negative examples in the leaves is about the same as in the parent:



If $p / n \approx p_1 / n_1 \approx p_2 / n_2 \approx \ldots \approx p_k / n_k$, then the split is not useful.

The gain is *not* low if the distribution deviates from this pattern with statistical significance as measured by the χ^2-distribution test. Thus, we use an attribute in splitting only if it passes the statistical χ^2 test. This strategy is called χ^2 *pruning*; it allows handling "noise" and insufficient data without overfitting.

Cross-validation: Set aside some of the available data and use them to evaluate the quality of a tree. We may also use them to prune inappropriate branches, thus avoiding overfitting.

Sample complexity

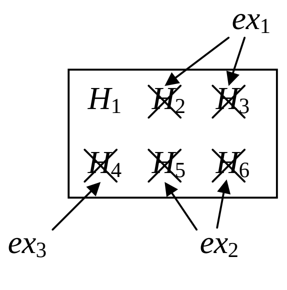
Intuitively, the sample complexity of a learning algorithm is the dependency between the number of training examples and the accuracy of the learned knowledge.

A learned hypothesis (e.g. a decision tree) is *approximately correct* if it correctly classifies "almost all" examples; that is, the error rate is within some small ε. A learning algorithm is good if it produces an approximately correct hypothesis with high probability; that is, the probability of not being approximately correct is within some small δ. In this case, we say that learning is probably approximately correct (PAC). The required number of training examples is a function of ε and δ; this function is called *sample complexity.*

The probability that the learned hypothesis incorrectly classifies a given example is within δ + ε, where δ is the chance that the learned hypothesis is no good, and ε is the chance that a good hypothesis gives a wrong result.

Stationarity assumption: Training examples have the same distribution as test examples. Without this assumption, we cannot guarantee good results.

The learner selects a hypothesis from a certain pre-defined space, for instance, from the space of possible decision trees. The hypothesis is consistent with all training examples; thus, each example eliminates wrong hypotheses from the space. The more examples, the fewer wrong hypotheses are left.



Suppose that H is a "bad" hypothesis; that is, its error rate is > ε. The probability that it is consistent with one training example is $< 1 - \varepsilon$. The probability that it is consistent with m training examples is $< (1 - \varepsilon)^m$. Suppose that the space includes h different hypotheses. Then, the probability that at least one bad hypothesis is consistent with all m examples is $< h \cdot (1 - \varepsilon)^m$. This probability must be bounded by δ:

$$h \cdot (1 - \varepsilon)^m \leq \delta,$$

which holds if

$$m \geq \frac{1}{\varepsilon} \cdot (\ln \frac{1}{\delta} + \ln h).$$

Thus, to ensure probable approximate correctness, we need $\left\lceil \frac{1}{\varepsilon} \cdot (\ln \frac{1}{\delta} + \ln h) \right\rceil$ training examples. The number of examples depends on error rates, ε and δ, and on the number of candidate hypotheses, *h*. The hypothesis-space size is usually exponential in the length of hypothesis description, and the required number of examples is usually polynomial in the description length.

Justification for Occam's razor: Shorter hypotheses require fewer examples for a given error rate; in other words, they allow more accurate learning with the available examples. Selecting a small hypothesis space is essential.

Space of decision trees:
Suppose that we consider *n* attributes, with *k* values for each attribute. For a given ordering of attributes, the tree has k^n leaves, and each leaf is + or –; thus, there are 2^{k^n} different trees. We may consider $n!$ attribute orderings; thus, the total number of trees is $n! \cdot 2^{k^n}$.

$$\ln(n! \cdot 2^{k^n}) = \ln n! + \ln 2^{k^n} \leq n \cdot \ln n + k^n \approx k^n$$

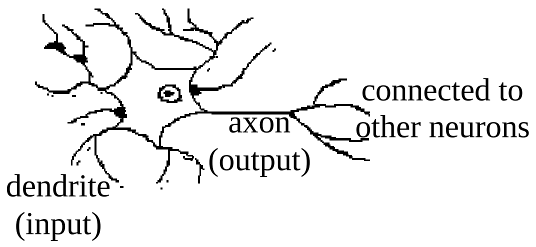
$$m = \frac{1}{\varepsilon} \cdot (\ln \frac{1}{\delta} + k^n)$$

Thus, the number of examples is exponential in the depth of the tree; however, it is reasonably small for a limited depth, for instance, if $n = 3$.

Neural networks (Chapter 19, except 19.6)

The neural networks are based on the analogy with the human brain; however, they do *not* provide a plausible model of human problem-solving abilities. The artificial neural networks are non-symbolic and massively parallel. On the other hand, human problem solving is probably based on symbolic sequential algorithms. Although a brain consists of neurons, they seem to form a close analog of a sequential digital computer.

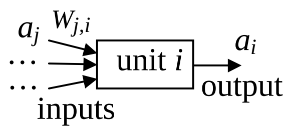
Human neurons



Some inputs increase the electric potential of the neuron, and some decrease the potential. When the potential reaches a certain threshold, the neuron sends an output impulse. The input/output connections and their strength can change over time, which may be viewed as "self-reprogramming," that is, learning.

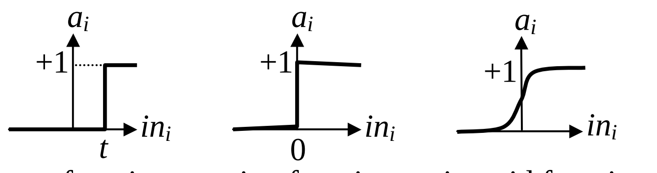
Artificial neurons

"Neurons" in artificial neural networks are called *units* or *nodes;* "axons" are *links* with weights. The output is usually 0 or 1; in some models, it is –1 or 1.



a_i is the output of unit i
a_j the output of unit j, connected to unit i
$W_{j,i}$ is the weight of the connection between units j and i

The "potential" is called the *input function:* $in_i = \sum W_{j,i} \cdot a_j$.
The "threshold" is called the *activation function:*

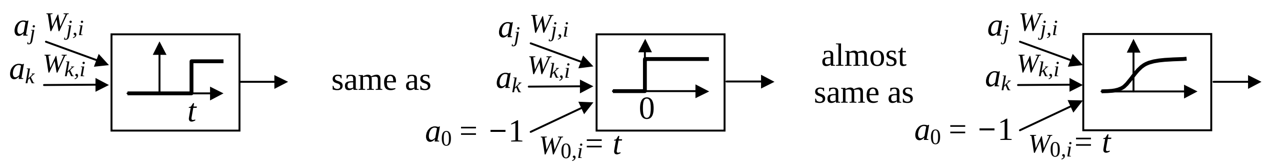


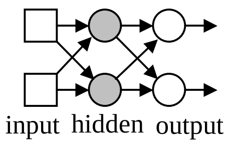
step function sign function sigmoid function

Sigmoid:

$$a_i = \frac{1}{1+e^{-in_i}}$$

We usually use the sigmoid function because it allows an easy computation of the derivative, which is required for learning. To simulate a step function using a sigmoid, we add unit 0 with the constant output −1.

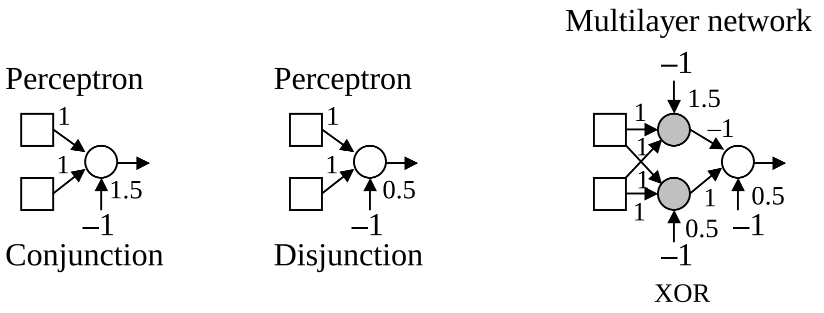




A neural network is a collection of units, some of which are connected. The units that represent the outside input are called *input units,* and the units that output the final result are called *output units;* all other units are *hidden units.* The input values may be different from 0 and 1.

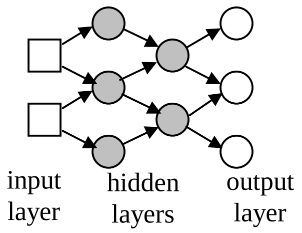
Neural networks operate with continuous real values, which is different from most other AI techniques.

Example



A network without cycles is called *feed-forward*. A network with cycles is *recurrent*. Note that recurrent networks require some "clock"; they have internal state and may lead to very long (or infinite) looping. Most applications use feed-forward networks.

If a feed forward-network has no hidden units, it is called a *perceptron;* otherwise, it is a *multilayer network.* We usually use feed-forward networks arranged in layers; each unit of a layer is connected only to the next layer. For example, the network in the picture is a three-layer network; note that we do not count the input layer.

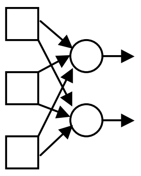


A network represents a specific function, which depends on the structure (units and links) and weights. A learning algorithm has to find a function that fits training examples. Usually, the human user defines the structure, and the learning algorithm adjusts the weights.

Some approaches to automatically changing the structure:

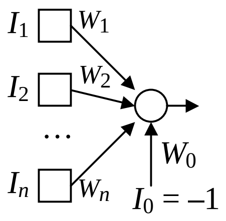
- Genetic algorithms: Simulate the "evolution" of neural networks
- Optimal brain damage: Remove the connections that seem unnecessary and readjust the weights of the other connections; repeat until the performance begins to degrade
- Tiling algorithm: Add new units to improve the classification accuracy using techniques similar to building a decision tree

Perceptrons



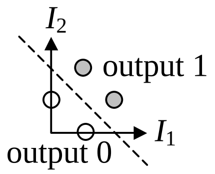
A *perceptron* is a one-layer feed-forward network; thus, it has no hidden units. Each output unit is independent of the other units, which means that we can limit the study to single-output perceptrons.

$$in = \sum W_j \cdot I_j$$



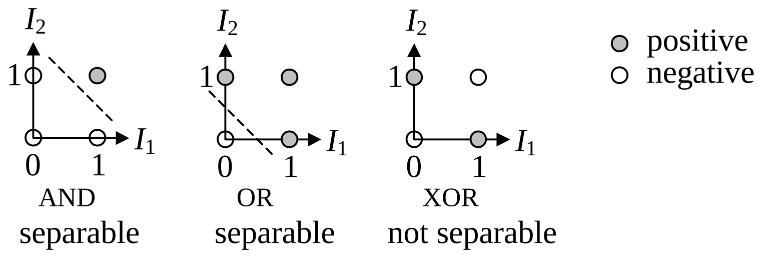
A perceptron can represent some Boolean functions, such as AND, OR, and majority, but it cannot represent XOR or parity.

If a perceptron uses the "sign" activation function, it outputs 0 when $\sum W_j \cdot I_j < 0$, and 1 otherwise. If we view possible inputs as points in n-dimensional space, then this inequality describes a halfspace.

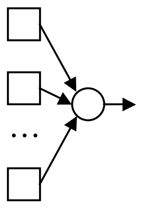


Thus, positive examples must be separated from negative by a hyperplane in the n-dimensional space; we say that they are *linearly separable.* If this condition does not hold, a perceptron cannot reliably distinguish between positive and negative examples.

Example



Thus, constructing a perceptron for a given set of examples is equivalent to finding a hyperplane that separates positive and negative examples. If the examples are separable, we can efficiently construct a perceptron that classifies them.



- Make a perceptron with an appropriate number of inputs
- Randomly assign weights (usually, between –0.5 and 0.5)
- Repeatedly recompute weights based on the training examples

Updating weights:
- If the perceptron correctly classifies a training example, do not change the weights
- If it gives 1 instead of 0 (negative error), increase the weights of the negative inputs and decrease the weights of the positive inputs
- If it gives 0 instead of 1 (positive error), do the opposite

Thus, we update each weights W_j as follows:

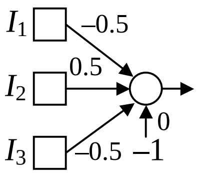
$$W_j \leftarrow W_j + \alpha \cdot I_j \cdot Err$$

I_j is input j for the current example
Err = *Correct-Output* – *Observed-Output*, which is –1, 0, or 1
- If the output is correct, $Err = 0$
- If 1 instead of 0, $Err = -1$
- If 0 instead of 1, $Err = 1$

α is the learning rate

We can re-use an example several times during the learning process. An *epoch* of learning is updating the weights for each of the examples; the learning process involves multiple epochs.

Example: Learning the majority function.

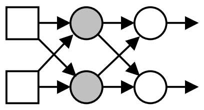


$\alpha = 0.1$

Examples			Weights			
I_1	I_2	I_3	W_1	W_2	W_3	W_0
			–0.5	0.5	–0.5	0.0
1	1	–1	–0.5	0.5	–0.5	0.0
–1	1	–1	–0.4	0.4	–0.4	0.1
1	–1	1	–0.3	0.3	–0.3	0.0

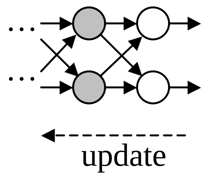
This search is hill-climbing toward appropriate weights, with no local maxima. If α is sufficiently small to avoid "jumping across the maximum," the perceptron converges to appropriate weights.

Multilayer networks



The learning algorithm is similar to perceptrons; it repeatedly modifies weights until the network correctly classifies (almost) all examples. The activation function in multilayer networks is usually the sigmoid.

Back-propagation: At each step, the update of weights propagates backward from the output layer.



- For each output unit i, determine its error:

  $Err_i = Correct\text{-}Output_i - Observed\text{-}Output_i$

  If the activation function is the sigmoid, Err_i may be fractional, between –1 and 1.
- Compute the corresponding error term:

  $\Delta_i = Err_i \cdot g'(in_i)$

  $g'(in_i)$ is the derivative of the activation function; for the sigmoid, $g'(in_i) = g(in_i) \cdot (1 - g(in_i))$
- For each unit j linked to i, update the weight $W_{j,i}$:

  $W_{j,i} \leftarrow W_{j,i} + \alpha \cdot a_j \cdot \Delta_i$

  The rule is similar to that for perceptron, but there are two differences:

  - we use Δ_i instead of Err_i
  - we update the weights of links outgoing from the next-to-last layer rather than from the input units
- Compute the error term for each unit j in the next-to-last layer; intuitively, we propagate the error term:

  $\Delta_j = g'(in_j) \cdot \sum W_{j,\,i} \cdot \Delta_i$ (the sum over output units; the weights are *before* the update)

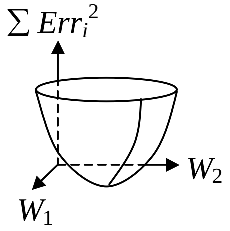
  Thus, we compute these error terms before updating the weights.
- Use these error terms to update the weights of incoming links, propagate error terms backward to the next layer, and so on.

Steps of back-propagation:

- Compute Δ values for the output layer
- Repeat until reaching the earliest hidden layer:
  - propagate Δ values back to the previous layer
  - update the weights between the two layers
- Update the weights between the earliest hidden layer and the input layer

For more details, see the algorithm in Figure 19.14 of the textbook.

Mathematical foundations



The error for a specific example is a function of all weights in the network. The back-propagation procedure is a gradient descend toward the minimum of this function. The learning rate α determines the size of steps.

The convergence time depends on a specific set of examples; in the worst case, it may be exponential in the number of inputs. Furthermore, the gradient descent can stop at a local minimum; for instance, if most examples are positive, it may converge to classifying all examples as positive. We can use simulated annealing to avoid local maxima.

Advantages and drawbacks

In theory, we can represent any continuous function by a two-layer network, and any function at all by three layers; however, the required number of nodes may be exponential. Note that the expressiveness of a network with a specific fixed structure is still limited.

Advantages:
- Effective for dealing with continuous values
- Insensitive to noise

Drawbacks:
- Provides yes/no answers without probabilities
- Does not provide explanations
- Does not use prior knowledge in learning

Some applications:
- Handwriting recognition (e.g. zip codes)
- Face detection and recognition
- Automated driving (NavLab)