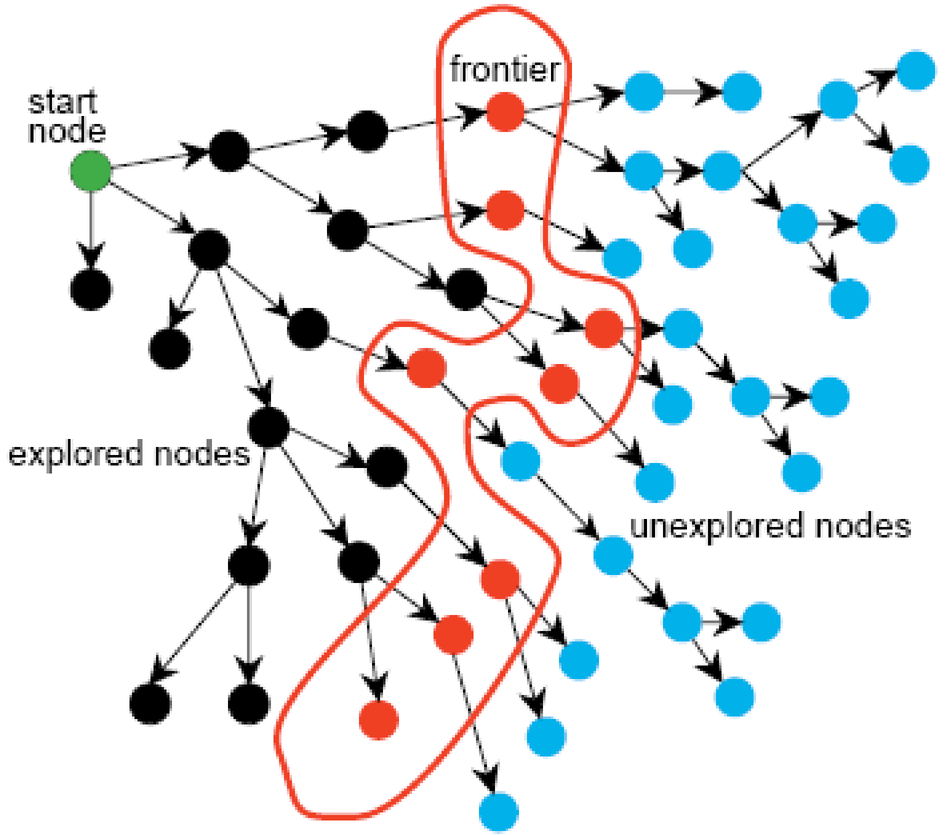
Searching for a solution

The process of searching for a solution may be represented in terms of the following concepts, and visualized on a **search graph**:

- start node corresponds to the initial state of the agent
- other nodes correspond to the subsequent states through which the agent traverses the space
- graph edges correspond to the state transitions or actions of the agent; they may have costs assigned
- part of the space is explored and known to the agent, while the rest is yet undiscovered



Searching is a component of all methods of artificial intelligence, and the ability of efficient searching seems to be an inherent attribute of the intelligence proper.

Exponential search spaces

Let's try to make some numerical characterization of the search space. For simplicity let's assume there is a constant set of state transitions available in each state, denote their number as b, as in **branching factor**.

How many states does the agent need to examine to explore the space to the depth d? It is b for $d = 1$, b^2 for $d = 2$, b^3 for $d = 3$, ... and generally b^d for the depth d.

And how deep will the agent generally need to search? That depends on how deep the solution state is located. It may be close to the starting state, which means the problem is easy, or it may be far away from the starting state, or deep in the search space, which signifies a hard to solve problem.

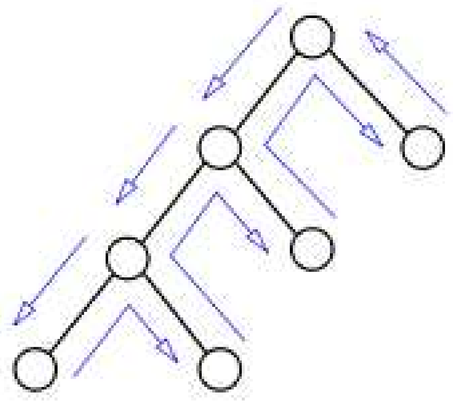
Assuming the agent will need to search a significant part of the size of the space for each specific problem, we can say that the amount of search grows exponentially with respect to the difficulty of a problem, so the search process will have complexity $O(b^d)$.

This is bad news. For exponential problems, assuming $b = 10$, solving an easy problem, like $d = 5$, will require exploring 10^5 states, which might take as little as 1 second. A more difficult problem with $d = 10$ will involve 10^{10} states and 10^5 seconds, or just over one day of computing time, but a problem with $d = 15$ will require 317 years of compute time. Even with massively parallel processors this will likely be unacceptable.

Backtracking search (BT)

```
FUNCTION BT(st)
BEGIN
   IF Term(st)            THEN RETURN(NIL)    ; trivial solution
   IF DeadEnd(st)         THEN RETURN(FAIL)   ; no solution
   ops := ApplOps(st)                         ; applicable operators
L: IF null(ops)           THEN RETURN(FAIL)   ; no solution
   o1 := first(ops)
   ops := rest(ops)
   st2 := Apply(o1,st)
   path := BT(st2)
   IF path == FAIL        THEN GOTO L
   RETURN(push(o1,path))
END
```

The BT algorithm efficiently searches the solution space without explicitly building the search tree. The data structures it utilizes to hold the search process state are hidden (on the execution stack). It is possible to convert this algorithm to an iterative version, which builds these structures explicitly.
The iterative version is more efficient computationally, but lacks the clarity of the above recursive statement of the algorithm.



Backtracking search — properties

BT has minimal memory requirements. During the search it only keeps a single solution path, and discards all information for the explored and abandoned parts of the search space. Its space complexity is $O(d)$, where d — the distance from the initial state to the solution (measured in the number of the operator steps). The space complexity is the same for the best case, average case, and worst case.

The time complexity is much worse, but critically depends on how lucky the search is. In the worst case scenario the BT algorithm may visit all the states in the space before finding the solution. And even the average case has exponential $O(b^d)$ time complexity.

But perhaps the most important problem with the BT algorithm is that it might not find a solution, even if one exists. If the state space is infinite, the algorithm may execute an action leading to an infinite subspace which contains no solution. In this case, the algorithm will never backtrack from the wrong operator choice, and keep searching forever.

Even with finite spaces, the algorithm may fail to find a solution. If the algorithm gets into a loop, it may get trapped in it and never be able to break out. This depends on how the algorithm processes the successor states that it reaches by its actions.

Checking for repeated states

One of the problems with BT — as well as with all search algorithms — is the potential for looping. If the algorithm ever reaches a state, which it has already visited on its path from the initial state, and just continues the state space exploration, then it will repeatedly generate the same sequence of states and may never break out of this loop.

In principle, it is quite easy to avoid this problem. The simplest way is to check, after reaching any state, whether that state is not present on the current path from the start state. And if it is, then prevent the exploration of this just arrived at, revisited state.

However, such checking incurs a significant computational overhead. Each state exploration step, instead of being constant time and possibly quite fast, would require d steps of checking, at depth d. It is therefore a compromise choice, whether to significantly extend the running time, and avoid the loops, or skip this checking to save time, but risk looping.

In fact, it is possible to do a more extensive repetition check. The state the algorithm arrives at may not be on the current path, but may have been previously discovered, and explored, on another search branch, already backtracked from. For such test a set of all visited stated must be kept, a so-called *Closed* list. In the recursive version of the BT algorithm this list would need to be global for all the invocations of the procedure.

Search depth limiting with iterative deepening

A serious problem for the BT algorithm are infinite, or extremely large spaces, which it generally cannot handle. If the algorithm makes a wrong turn, and starts exploring an infinite, or a very large, subtree which contains no solution, it may never backtrack and will not find the solution. Particularly costly, or fatal, are bad choices made at the very beginning of the search.

This is a problem with all "optimistic", or depth-first, algorithms, such as BT, which prefer to pursue any path as long as possible, rather than worry about alternatives.

A general and effective solution to this problem is to set a search depth limit. It protects from the consequences of taking a wrong turn in a large space, but also of getting trapped in a loop. The depth limit needs to be set to some "reasonable" value.

It is, however, generally not easy to determine such "reasonable" value. Setting it unreasonably high reduces the benefits of having a limit, while setting it too low prevents the algorithm from finding a solution when one exists.

Therefore search depth limiting needs to be augmented with **iterative deepening**. With this modification BT is **complete** — as long as a solution path to the goal state exists, the algorithm will find it. However, iterative deepening is very inefficient with BT. If the depth limit is set too low, BT searches the space to that limit, but when it backtracks, it deletes all search results. So after each deepening step the whole search process essentially starts from scratch.

Heuristics and static evaluation functions

The algorithms presented so far are simple and do not generally require an informed strategy to work. Having and using such strategy is however always desirable.

A heuristic we will call some body of knowledge about the problem domain which:

- cannot be obtained from a syntactic analysis of the problem description,
- may not be formally derived or justified, and which may even be false in some cases, and may lead to wrong hints for searching,
- but which in general helps make good moves in exploring the search space.

Having a heuristic should permit one to build informed search strategies. A general and often used scheme for constructing strategies using heuristic information is a **static evaluation function**. This function, defined on all states, expresses either:

- the perceived "goodness" of a state, or a chance that the solution can be reached via this state efficiently, or:
- the estimated distance from this state to the solution.

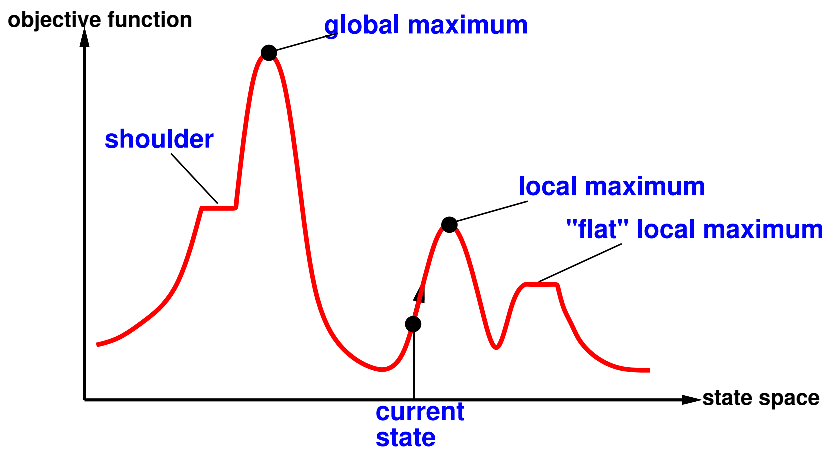
These two metaphors for formalizing heuristics actually have opposite meanings: in the first case, a higher value is better, while the opposite is true in the second. The latter will be mostly assumed here.

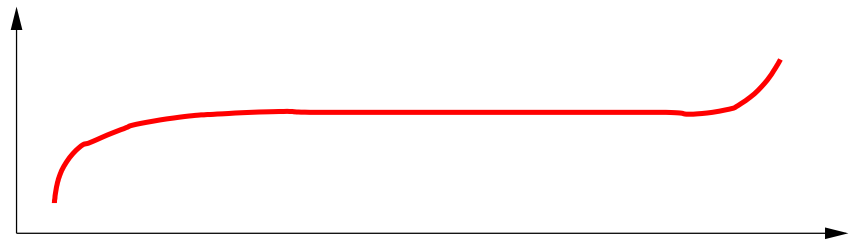
Hill climbing approaches

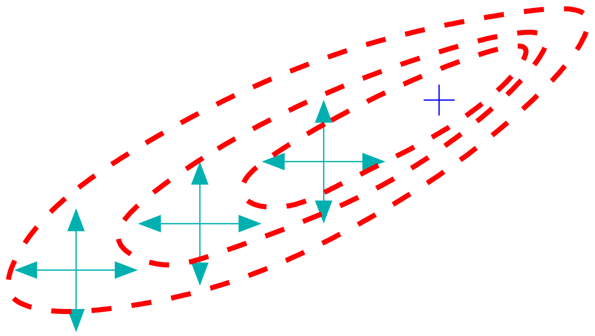
An evaluation function can be applied directly in searching. This leads to a class of methods called **hill climbing**. Hill climbing methods generally belong to the class of greedy algorithms.

Direct application of these methods is limited to domains with a very regular evaluation function, eg. strictly monotonic one. Applying hill climbing in practical cases typically leads to the following problems:

1. local maxima of the evaluation function
2. "plateau" areas of the evaluation function
3. oblique "ridges" of the evaluation function





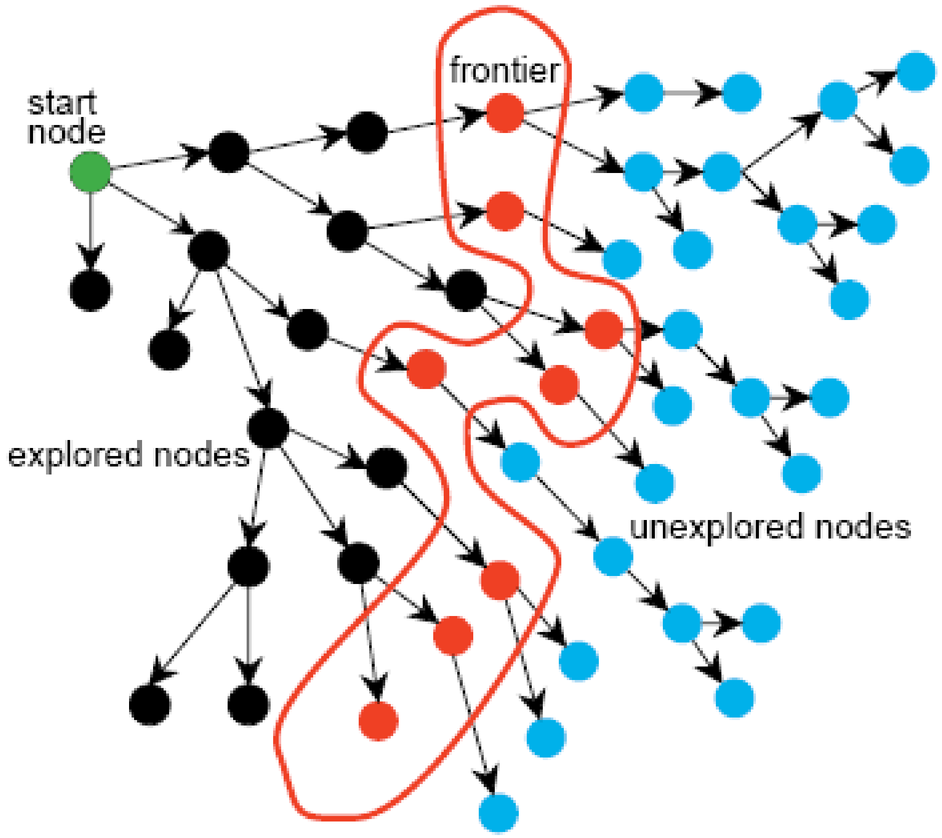


Graph searching

Recall the the iterative deepening version of the backtracking (BT) algorithm, and the problem of repeated explorations of the initial part of the search space. In order to avoid such repeated exploration one might introduce an explicit representation of the search graph, and keep in memory the explored part of the search space. Algorithms which do this are called **graph searching** algorithms.

General graph searching strategies (blind):

- breadth-first search strategy (BFS),
- depth-first search strategy (DFS),
- other strategies.



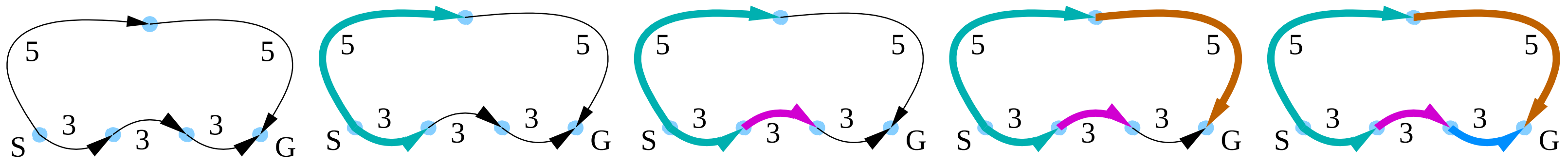
Graph searching algorithms

The graph searching algorithms perform a simulated exploration of the state space by generating new yet-unknown states as successors of the states already known. This is the process of repeatedly **expanding** a selected state by trying all possible state transition operators on it. The strategy determines which state is selected for expansion first.

```
FUNCTION TreeSearch(Problem,Strategy)
BEGIN
   initialize the search tree using the initial state of Problem
   LOOP
      IF there are no candidate nodes on Open
         THEN RETURN failure
      choose a node from Open according to Strategy
      IF the node corresponds to a goal state
         THEN RETURN the respective solution
      expand the node and add the child nodes to the tree
      transfer the node from Open to Closed
END
```

Uniform-cost search (UC)

In those cases, when the costs of all moves are not equal, the breadth-first search, which is based on the number of moves, obviously no longer guarantees optimality. A simple extension of the breadth-first algorithm finds the optimal path for any (positive) cost of a single move. This algorithm is called the **uniform-cost** (UC) search, and works by always selecting for expansion the graph node of the lowest path cost.



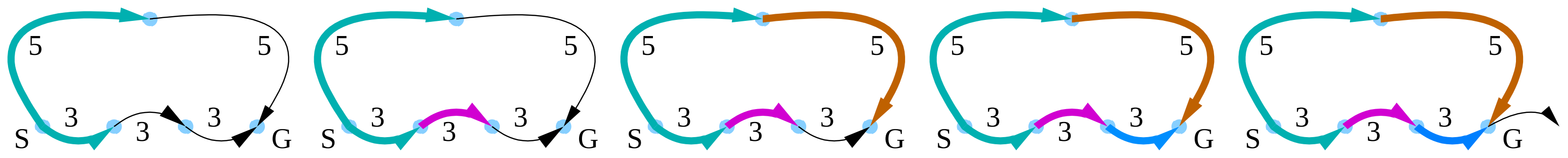
In the case of equal costs, this is identical to breadth-first search.

The optimality of the algorithm can be (trivially) proved as long as the cost of a single move is some positive value ($\geq \epsilon$). But since the algorithm is guided by the path cost, its complexity cannot be characterized as a function of b and d. Instead, if C^* denotes the cost of the optimal solution, the worst case complexity of the algorithm — both time and memory — is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

In the case of equal costs, this is $O(b^d)$.

Search termination

The goal of searching might be just to find some path to the goal, or to find the optimal path. In the former case, the algorithm may terminate when it discovers, that the state it has just reached, ie. that has been placed on the *Open* list, is the goal state. But can we do the same when searching for an optimal solution?



The optimal search should be terminated when the algorithm has just chosen a goal node (possibly one of a few already reached goal nodes) for expansion. The expansion can then be abandoned, and the algorithm terminated, but the best known path to the goal node is the optimal solution. Since the algorithm systematically finds all cheapest paths, a decision to expand a node means, that there may not exist any cheaper paths to it.

Before that happens, however, the algorithm explores cheaper paths, and there is no guarantee that it would not find a new, better path to the goal node.

Adding heuristics: the best-first search

The most straightforward application of a heuristic state evaluation function to graph searching leads to the **best-first search**. At any point, it chooses to expand states with the best heuristic value. With a good evaluation function, such which correctly evaluates states, and decreases in value along the path to the solution, the best-first search algorithm proceeds directly toward the goal state, wasting no time exploring any unnecessary states (graph nodes).

Also with slight defects in the evaluation function, with a few values a little off, but no systematic errors, this scheme works very well in guiding the search space exploration process.

The problems start when the evaluation function is wrong in a larger (perhaps infinite) part of the search space, and consequently indicates as good some states which do not lead to a solution. In such cases the best-first strategy exhibits the same problems as the depth-first search, even if the evaluation function may correctly evaluate many, or most, states.

A modified node selection — the already incurred cost

Consider the following deterministic state (node) evaluation functions:

$h^*(n)$ – the cost of the cost-optimal path from n to the goal
$g^*(n)$ – the cost of the cost-optimal path from s_0 to n

Therefore:

$f^*(n) := g^*(n) + h^*(n)$
$f^*(n)$ – the cost of the cost-optimal path from s_0 to the goal, going through n

Having access to the $f^*(n)$ function would allow one to always select the nodes on the optimal path from start to the goal. In fact, it would suffice to use the $h^*(n)$ function. In both cases, the agent would go directly to the goal.

Unfortunately, these functions are normally not available. We are forced to use their approximations to select nodes in the graph. However, when using the approximations, then the search based on the $f^*(n)$ function does not necessarily proceed exactly like that based on the $h^*(n)$ function.

A modified node selection — the A* algorithm

Consider the following heuristic (approximate) state evaluation functions:

$h(n)$ – a heuristic approximation of $h^*(n)$
$g(n)$ – the cost of the best known path from s_0 to n; note $g(n) \geq g^*(n)$
$f(n) := g(n) + h(n)$

How does the strategy using the $f(n)$ approximation work? If $h(n)$ estimates the $h^*(n)$ value very well, then the algorithm works perfectly, going directly to the goal. If, however, the $h(n)$ function is inaccurate, and eg. reports some states to be better then they really are, then the algorithm will first head in their direction, lured by the low values of $h(n)$, while $g(n)$ was negligible.

After some time, however, such erroneously estimated paths will stop being attractive, due to the increasing $g(n)$ component, and the algorithm will switch its attention to more attractive nodes. The attraction of a node here is not affected by how far it is from start or from the goal. Instead it is determined only by the combined estimate of the total cost of a complete start-to-goal path running through that node.

An algorithm using a strategy with the above $f(n)$ function is called the **A* algorithm**.

The evaluation function in the A* algorithm

The $h(n)$ and $g(n)$ components of the $f(n)$ function represent the two opposite trends: the optimism ($h(n)$) and the conservatism ($g(n)$). We can freely adjust the strategy one way or the other by using the formula:

$$f(n) := (1-k) * g(n) + k * h(n)$$

By increasing the weight coefficient k we can bias the search toward more aggressive (and risky) when, eg. we trust the $h(n)$ function and want to proceed rapidly. On the other hand, by decreasing this coefficient, we enforce a more careful exploration of the search space, moving ahead slower, but possibly compensating for some of the $h(n)$ function's errors.

Note that in the extreme cases, $k = 1$ yields the best-first search, while $k = 0$ yields the uniform-cost search.

But it is the quality of the $h(n)$ function that has the biggest influence on the search process.

The h(n) function properties in A*

The heuristic evaluation function $h(n)$ in the A* algorithm is called **admissible** if it bounds from below the real cost function $h^*(n)$, ie. $\forall n\ h(n) \leq h^*(n)$. Admissibility means chronic underestimating of future costs, so it is also referred to as optimism. It can be proved, that whenever there exists a path from the start node to the goal, the A* with an admissible heuristic will always find the best such path.

This sound nice, so is it hard to find such an admissible heuristic? Not necessarily. For example, $h(n) \equiv 0$ indeed bounds $h^*(n)$ from below for any problem. And can such a trivial heuristic be useful? The answer is: not really. Such algorithm always selects the nodes with the shortest path from s_0, so it is equivalent to the breadth-first (more generally: uniform-cost) search which indeed always guarantees to find the optimal solution, but, as we already know, it is not such a great algorithm.

Naturally, the better $h(n)$ approximates $h^*(n)$ the more efficient the search is. In fact, it can be proved that for any two evaluation functions $h_1(n)$ and $h_2(n)$, such that for all states $h_1(n) < h_2(n) \leq h^*(n)$ using h_1 in search leads to the exploration at least the same number of states as it does using h_2.

The h(n) function properties in A* (cntd.)

Admissibility of the heuristic function $h(n)$ is an interesting property, which can frequently be proved for functions coarsely approximating $h^*(n)$, but not always can be proved for painstakingly elaborated function, such as using numerical learning from a series of examples (which is one method of constructing heuristic functions, which we will look at later).

An even stronger property of a heuristic evaluation function $h(n)$ is its **consistency**, also called the **monotone restriction**, or simply the triangle property:

$$\underset{n_i \rightarrow n_j}{\forall} \; h(n_i) - h(n_j) \leq c(n_i, n_j)$$

It can be proved that for a function h satisfying the monotone restriction the A* algorithm always already knows the best path to any state (graph node) that is chooses for expansion. In practice this makes it possible to simplify the search algorithm implementation, if we know that the evaluation function is consistent.

A* algorithm complexity

For most practical problems the number of nodes of the state space grows exponentially with the length of the solution path. Certainly, an efficient heuristic could decrease the computational complexity of the algorithm.
A good question is: when could we count on such a reduction?

It can be proved, that for this to happen, ie. for the A* algorithm to run in polynomial time, the estimate error of the heuristic evaluation function should not exceed the logarithm of the actual solution length:

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

In most practical cases one cannot count on finding such good heuristics, so the A* algorithm should be considered to be exponential. However, most often this bad time performance is not even the biggest problem with A*. Just as with most other graph searching algorithms, it stores all the discovered states in memory, and usually fills up the available computer memory a long time before running out of its time limit.

Memory-considerate variants of A*

There are variants of the A* algorithm which cope with the memory problem.

The IDA* (Iterative-Deepening A*) algorithm sets a limit on the f value to which the algorithm is allowed to proceed. After that the limit is extended, but the explored nodes are deleted from memory.

The RBFS (Recursive Best-First Search) algorithm is more like the recursive version of the BT algorithm. It explores the search graph recursively, always keeping in mind the estimated cost of the second-best option (at all levels of recursion). When the currently explored path estimate exceeds the memorized alternative, the algorithm backtracks. And when it does backtrack, it loses all memory of all the explored part of the space (but keeps the estimate of that path in case it is later necessary to also backtrack from the original alternative).

The SMA* (Simplified Memory-Bounded A*) proceeds just like A*, but only up to the limit of the currently available memory. After that, the algorithm continues, but deleting the least-promising node to make space for each newly encountered state. However, it stores in the parent of each deleted node its heuristic estimate, so in case all preserved nodes get their estimates higher, the algorithm may come back, and re-generate the deleted node.

Algorithm A* in practice

A good question is whether the heuristic search algorithms, such as A*, have important practical applications.

The answer is: yes, in some constrained domains, such as planning the optimal travel path of autonomous vehicles, or finding the shortest paths in computer games.

The A* algorithm is the heuristic version of Dijkstra's algorithm (1959) finding the shortest paths from a selected node to all the other graph nodes.

The Dijkstra's algorithm is also used in many technical applications, such as network routing protocols like OSPF, or finding the routes in the GPS navigation systems. In the latter domain, due to the graph size the Dijkstra's algorithm must be augmented by additional techniques. These can be heuristics, or introducing abstraction and path hierarchy. However, due to the commercial nature of this still developing application, the detailed solutions are rarely published.

Short review

1. What is the difference between A* and best-first search algorithms?
   How does this difference affect the search process?

2. What are admissible heuristics for the A* algorithm?
   What is their practical significance?

3. The heuristic search algorithm A* with an admissible evaluation function h guarantees finding an optimal solution, whenever one exists. Consider the following modifications of the f function, and answer whether they preserve the this optimality property of the A* algorithm. Justify your answer.

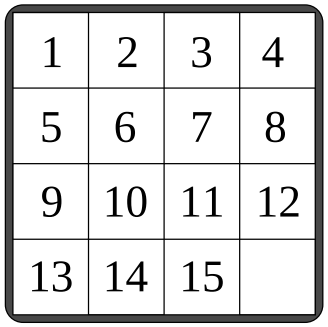
   (a) introduction of an upper bound on the value of the h(n) function
   (b) introduction of a lower bound on the value of the g(n) function

Constructing useful heuristics

How in general can one go about constructing a useful heuristic function, without a sufficient knowledge of the problem domain to design it from first principles?

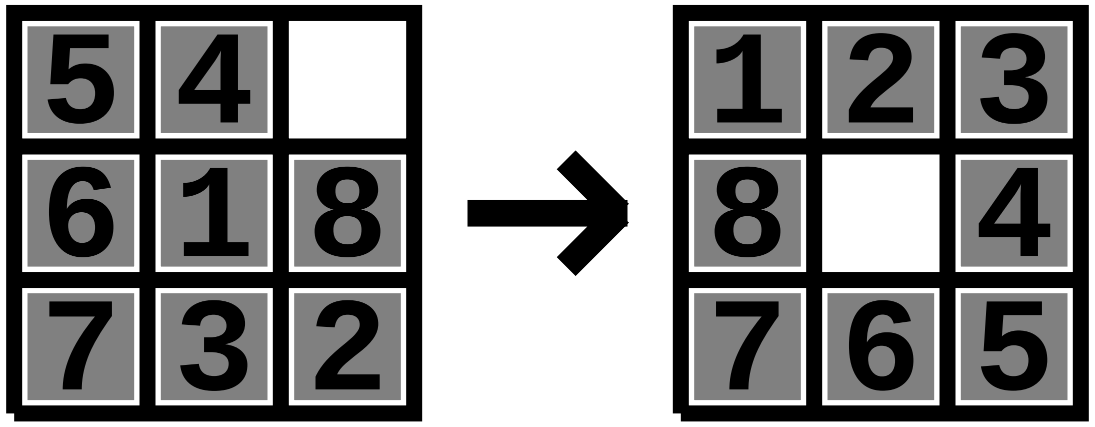
Experiment, experiment, experiment!

An example: the 8-puzzle



The 15-puzzle is popular with school children.

8-puzzle — a reduced version, suitable for testing various artificial intelligence algorithms and strategies, and presenting their operation.



Heuristics for the 8-puzzle

Heuristic 1: count elements in wrong places, the function $h_1(n) = W(n)$

Heuristic 2: for all the elements in a wrong place, compute and add up their distances from their proper place. The number thus derived will certainly be less than the number of moves of any complete solution (so is a lower bound of the solution). Call it the function $h_2(n) = P(n)$

Heuristic 3: $h_3(n) = P(n) + 3 * S(n)$
where the function $S(n)$ is computed for the elements on the perimeter of the puzzle taking 0 for those elements which have their correct right neighbor (clockwise), and taking 2 for each element which have some other element as their right neighbor. The element in the middle scores 1, if it is present.

In general, neither $S(n)$ nor $h_3(n)$ are lower bounds of the solution length. However, the $h_3(n)$ function is one of the best well-known evaluation functions for the 8-puzzle, resulting in a very focused and efficient search strategy.

On the other hand, the $h(n) \equiv 0$ function is a perfect lower bound solution estimation, satisfying the requirements of the A* algorithm, and always finding the optimal solution. This illustrates the fact, that *technically correct* is not necessarily *heuristically efficient*.

The heuristic function quality vs. the cost of A* search

The table shows a comparison of the search costs and effective branching factors for the Iterative-Deepening-Search and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

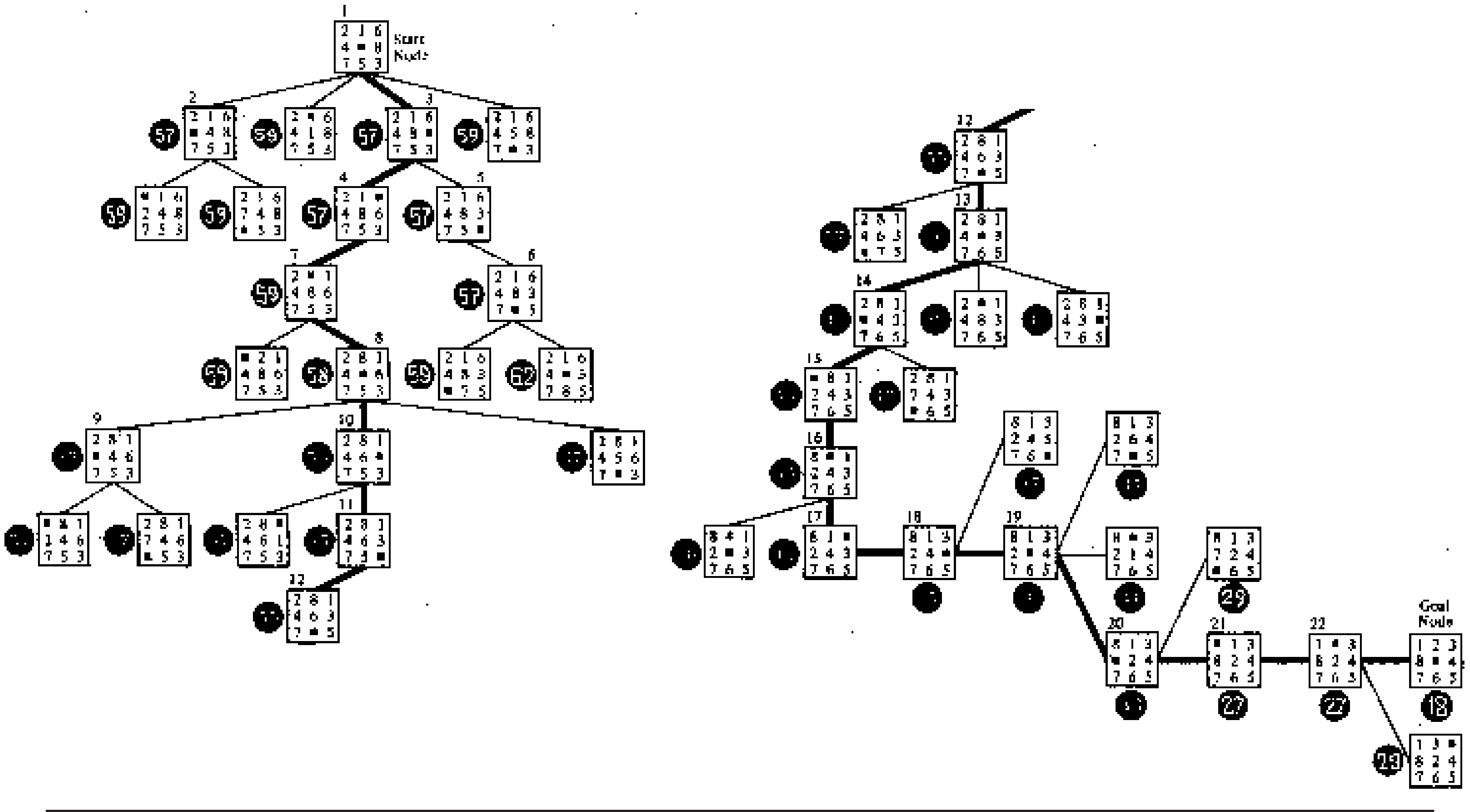
	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

(Table copied from the Russell&Norvig textbook.)

An approximate number of IDS nodes for d=24 is 54,000,000,000.

Heuristic search of the 8-puzzle search tree

The presented comparison of the heuristic functions for 8-puzzle does not contain the best h_3 function. Some illustration for its performance is the following example search tree, where the solution is at level 18, and the total number of nodes is 44. Its effective branching factor is 1.09



Constructing heuristics: the relaxed problem approach

One of the general approaches to constructing heuristic functions is the following. Consider a simplified problem, by giving up on some requirement(s), to make finding a solution easy. For each state generated during the search for the original problem, a simplified problem is solved (eg. using a breadth-first search). The cost of the optimal solution for the simplified problem can be taken as an estimation (lower bound) of the solution cost for the original problem.

For example, if the state space is defined with n parameters, so the states are the elements of the n-dimensional space, then one of the parameters can be eliminated, effectively mapping the states to $(n-1)$ dimensions.

If there are a few different ways, that this simplification can be achieved, and we cannot choose between them (eg. which state variable to drop), then we can use their combination for the evaluation function: $h(n) = \max_k(h_1(n), ..., h_k(n))$

Let us note, that in the case of the 8-puzzle heuristics, if one allowed a teleportation of the elements to their proper place in one move, it would be an example of such approach, and give the evaluation function $h_1(n)$. Further, the agreement to move elements by single field, but regardless of other elements possibly in the way, would give the function $h_2(n)$.

Constructing heuristic functions (cntd.)

Another approach to developing a heuristic function is to work it out statistically.

We need first to define some state attributes, which might be related to the distance to the solution. Having these, we take a heuristic function to be a linear combination of such attributes, with some unknown coefficients, which can be learned. This is done by running some experiments to determine some solution distances, using a full search, or another heuristic function.

The derived optimal solution distances can be used to construct a set of linear equations, which can be solved for the unknown coefficients.

Note that this is the way the $h_3(n)$ function for the 8-puzzle could possibly be found. The $W(n)$ and $P(n)$ functions could be assumed useful for constructing a good heuristic. The $S(n)$ function also estimates the difficulty of reaching the goal state. Using the function $h(n) = a * W(n) + b * P(n) + c * S(n)$ and running many experiments to compute $h(n)$, possibly we could have determined the approximate optimal values as: $a \approx 0$, $b \approx 1$ and $c \approx 3$, in effect obtaining the function $h_3(n)$.

Short review

1. Name and briefly describe the methods you know for creating heuristic evaluation functions.

Searching in two-person games

Games are fascinating and often intellectually challenging entertainment. No wonder they have been the object of interest of artificial intelligence.

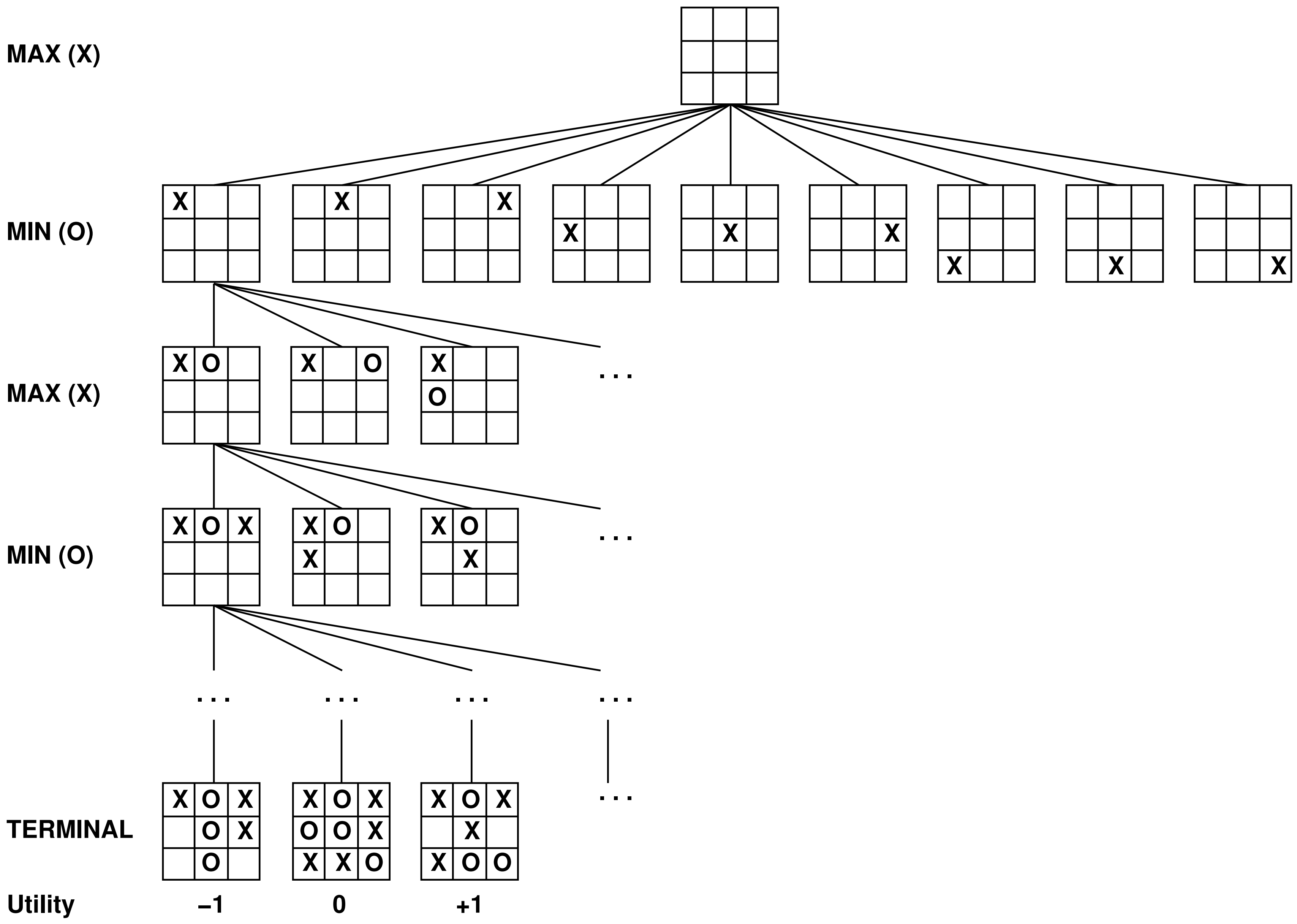
State space search methods cannot be directly applied to games because the opponent's moves, which are not known, must be considered. The "solution" must be a scheme considering all possible reactions of the opponent.

Additionally, in some games the full state information is not available to either player.

Types of games:

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon, monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble

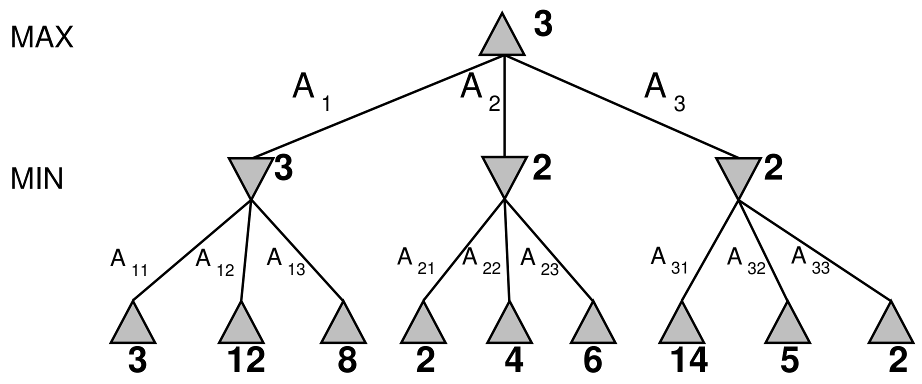
Two-person game tree



The minimax procedure

A complete strategy for a deterministic perfect information game can be computed using the following **minimax** procedure. It computes the value of the starting node by propagating the final utility values up the game tree:

1. the levels of the tree correspond to the players' moves: MAX's and MIN's; assume the first move is MAX's,
2. assign the MAX's win value to the terminal states in the leaves (negative, if they actually represent a loss to MAX)
3. tree nodes are successfully assigned the values: the maximum of the branches below if the current node corresponds to MAX, and the minimum of the branches below if the node corresponds to MIN,
4. the top tree branch with the highest value indicates the best move for MAX.



Resource limiting — using heuristics

The minimax procedure defines an optimal strategy for the player, assuming the opponent plays optimally. But only, if it can be fully computed.

For a real game tree this might be a problem. Eg., for chess $b \approx 35, m \approx 100$ for a reasonable game, and a complete game tree might have about $35^{100} \approx 10^{155}$ nodes. (The number of atoms in the known part of the Universe is estimated at 10^{80}.)

To solve this problem, a heuristic function estimating a position value can be used, like in standard state space search, to determine the next move without having an explicit representation of the full search space. In the case of a two-person game this facilitates applying the minimax procedure to a partial game tree, limited to a few moves.

For chess, such heuristic function can compute the **material value** of the figures on the board, eg. 1 for a pawn, 3 for a rook or a bishop, 5 for a knight, and 9 for the queen. Additionally, position value can be considered, such as „favorable pawn arrangement", or a higher value of the rook in the end-game (higher yet for two rooks).

Special situations in heuristic-based search

Limiting the depth search sometimes leads to specific issues, which require special treatment.

One of them is the concept of **quiescence search**. In some cases the heuristic evaluation function of some states may be favorable for one of the players, but the next few moves — which extend beyond the minimax search limit — inevitably lead to serious shifts, like exchanging some pieces in chess. It would be useful to detect such situations, and extend the search in the corresponding part of the game tree to reach a more stable configuration, or so-called *quiescent states*.

Another issue is the **horizon effect**. It occurs when an inevitable loss for one of the players approaches, but she can postpone its onset by making insignificant moves.

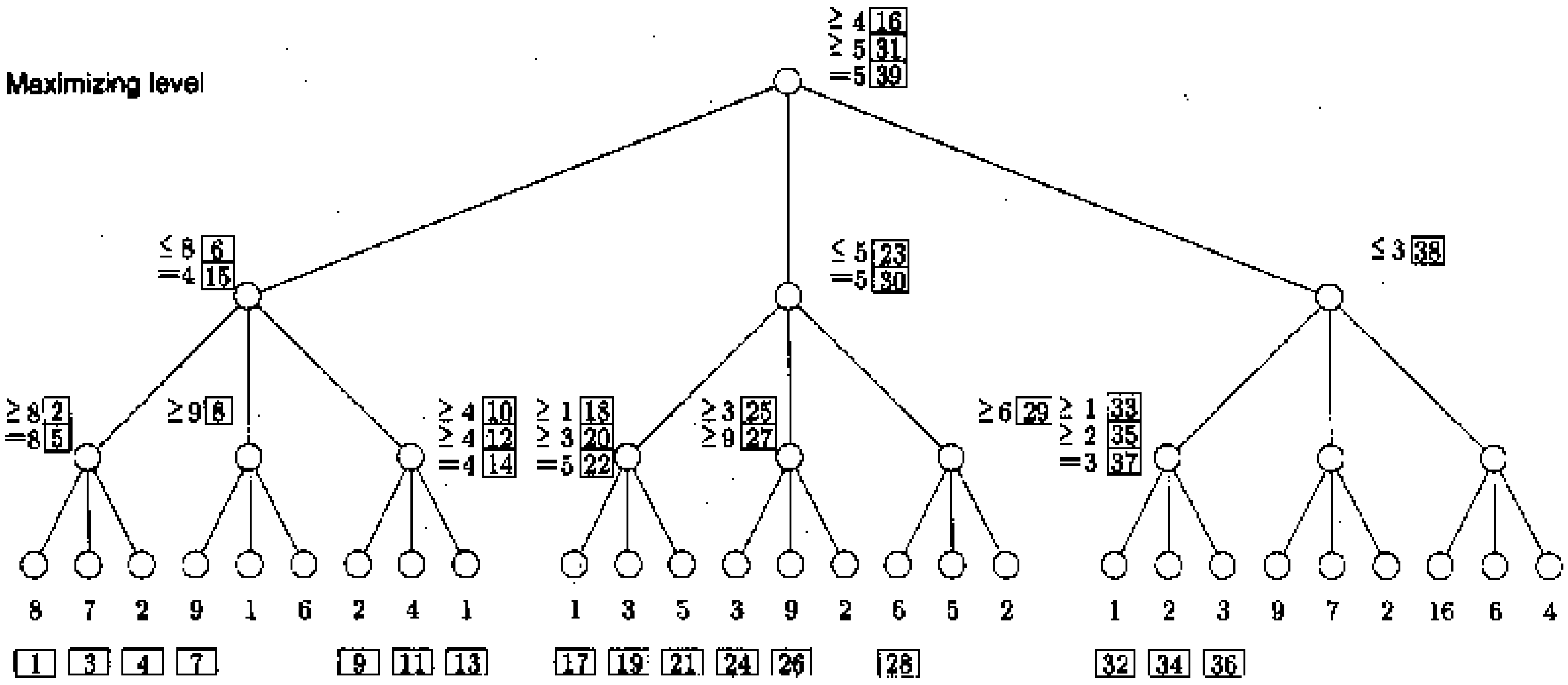
Minimax search — cutting off the search

What practical effects can be obtained with the heuristic search limited to a few steps?

Eg., for chess, assuming 10^4 nodes per second and 100 seconds for a move, $10^6 \approx 35^4$ positions can be explored, which amounts to 4 moves. Unfortunately, for chess this corresponds to only the most elementary play. Additional techniques for increasing the search efficiency are needed.

It turns out it is easy to make additional savings in the minimax. The most common approach is called the **alpha-beta pruning**.

α–β pruning — an example

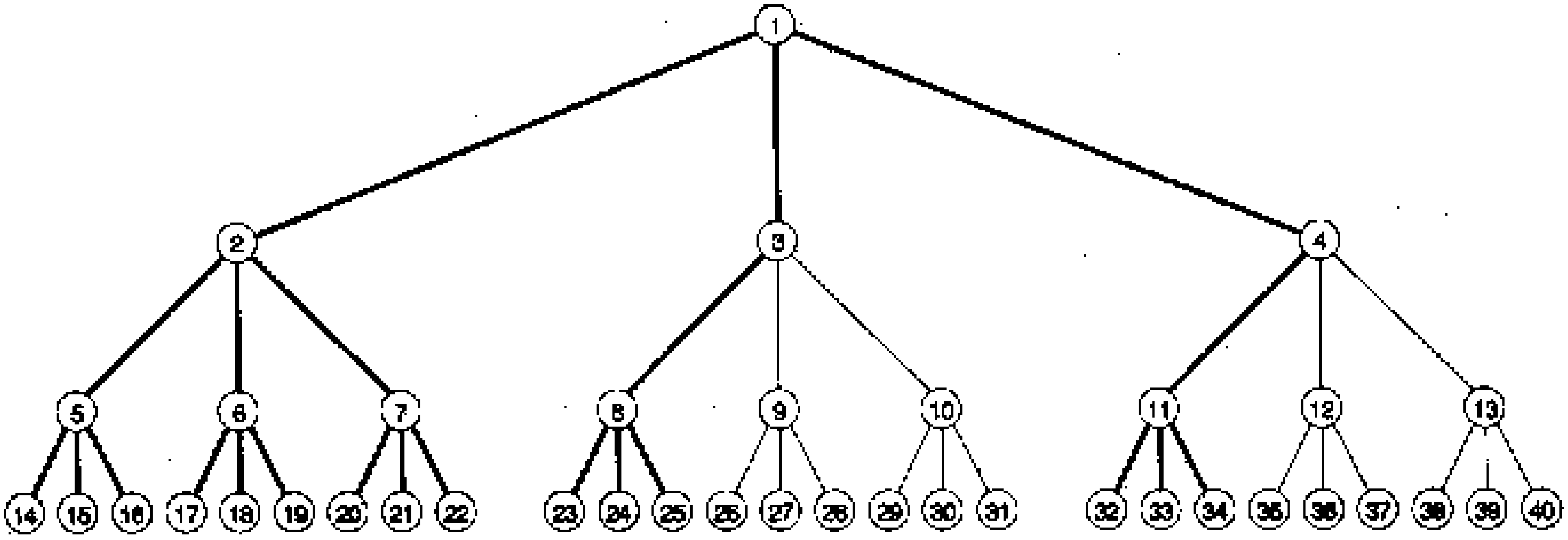


Exercise: find an error in the above tree (source: Patrick Henry Winston, *Artificial Intelligence*, 3rd ed.).

Answer: step $\boxed{10}$

α–β pruning — the optimal case

The optimal case of the minimax search with the alpha-beta cuts is when at each tree level the nodes are examined starting from the most favorite, for the given player. In such case only one “series” of nodes are evaluated in each subtree, and a cut occurs on each return up the tree.



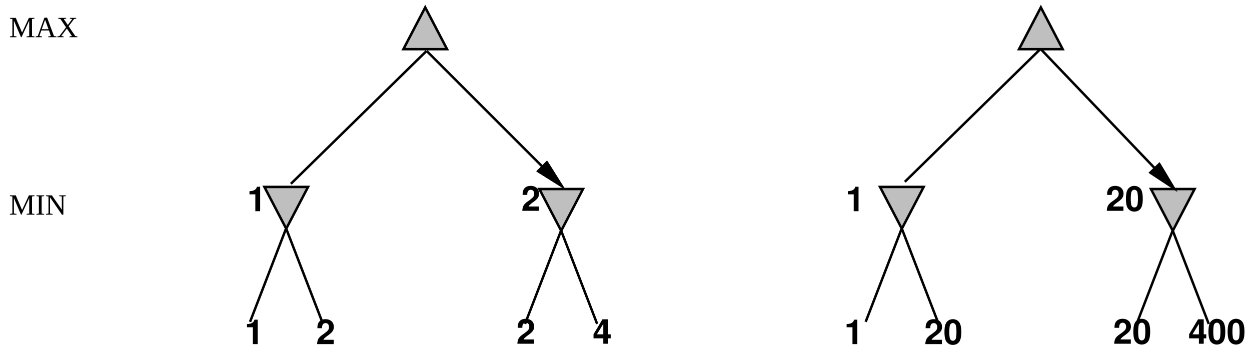
In the above diagram the savings is 16 nodes. Out of 27 nodes at the lowest level of the tree only 11 must be evaluated.
Source: Patrick Henry Winston, *Artificial Intelligence*, 3rd ed. (note an error: the nodes 18, 19, 21, and 22 could also be cut off).

The properties of the α–β algorithm

The basic idea of the α–β algorithm is that the cuts it makes do not affect the optimal move of the player.

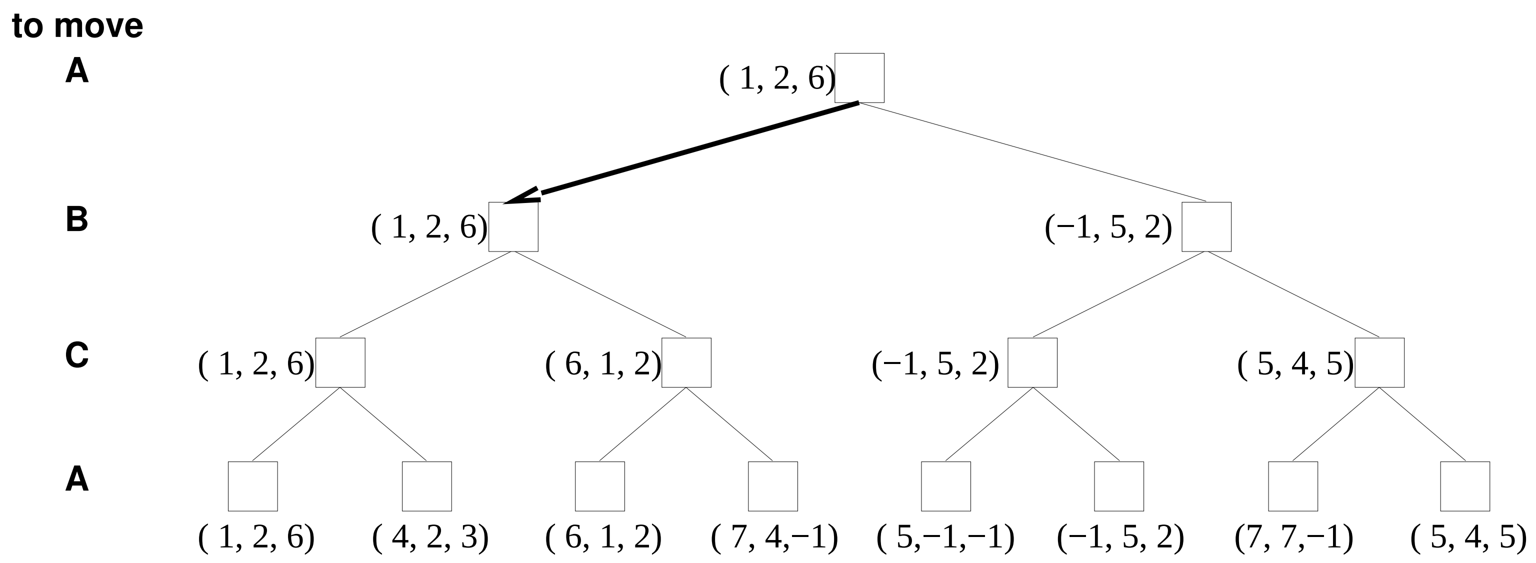
Introducing a favorable ordering allows better cut-off efficiency. In the limit, the optimal cuts achieve $O(b^{m/2})$ algorithm complexity. In practice this doubles the effective search depth.

The results of the min-max/α–β analysis does not depend on the specific values of the evaluation function, only on their ordering. This means that an arbitrary monotonic transformation of the evaluation function works as well and gives the exact same results.



Minimax — a multi-player generalization

The minimax algorithm can be generalized to a multiplayer case. In this case, a vector evaluation function must be employed, which evaluates the position from the point of view of each player. Each player maximizes her element of the vector, and the value propagation proceeds like in two-player case.



There are other factors that have to be considered in multi-player games, such as alliances. Sometimes it is advantageous for players to make alliances against other players, or even change these alliances dynamically during the game.

The practice of two-person computer games

Checkers: the Chinook program terminated in 1994 the 40-year long domination of the world champion Marion Tinsley[1] A year later the program defeated the subsequent champion Don Lafferty.

Chess: during the decade of 1990-2000 the chess programs rose to the level of the best human players. In 1997 Deep Blue first defeated the world master Garri Kasparov in an open tournament. Subsequently, chess programs running on normal computers entered. In 2006 Deep Fritz defeated the world master Vladimir Kramnik. After this, the excitement of the man-machine chess competition started to drop.

Othello: the human masters refuse to play the computers, which are superior.

Go: the human masters refuse to play the computers, which are too weak. Typical games include a handicap made by humans to the computers. The branching factor in go $b > 300$ so instead of a systematic search of the game tree most programs use a rule knowledge base to generate moves.

Lastest news: in March 2016 AlphaGo defeated a 9-dan master in an even game on a full-size board.

[1] Although the champion withdrew from the competition for health reasons and died shortly thereafter.

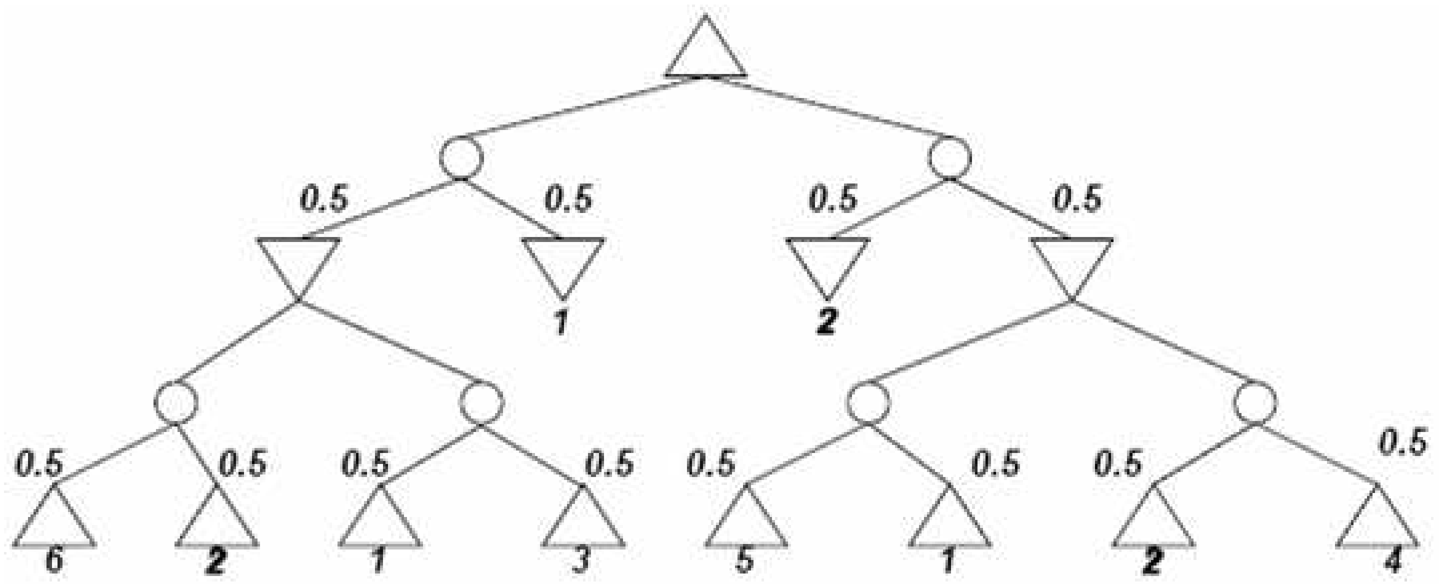
Games with chance elements — expectimax

With chance elements, the set of available actions at each step is dependent on some random variable, such as throwing the dice. The analysis is more complicated and requires considering all the options, and computing the expected values of the distributions of the random variables.

For example: one-person games with chance elements. Every other move is the player's choice, who maximizes her position evaluation, and the other moves are chance (or treated as chance), with a known probability distribution of the results. The algorithm modified to analyze such games is called **expectimax**.

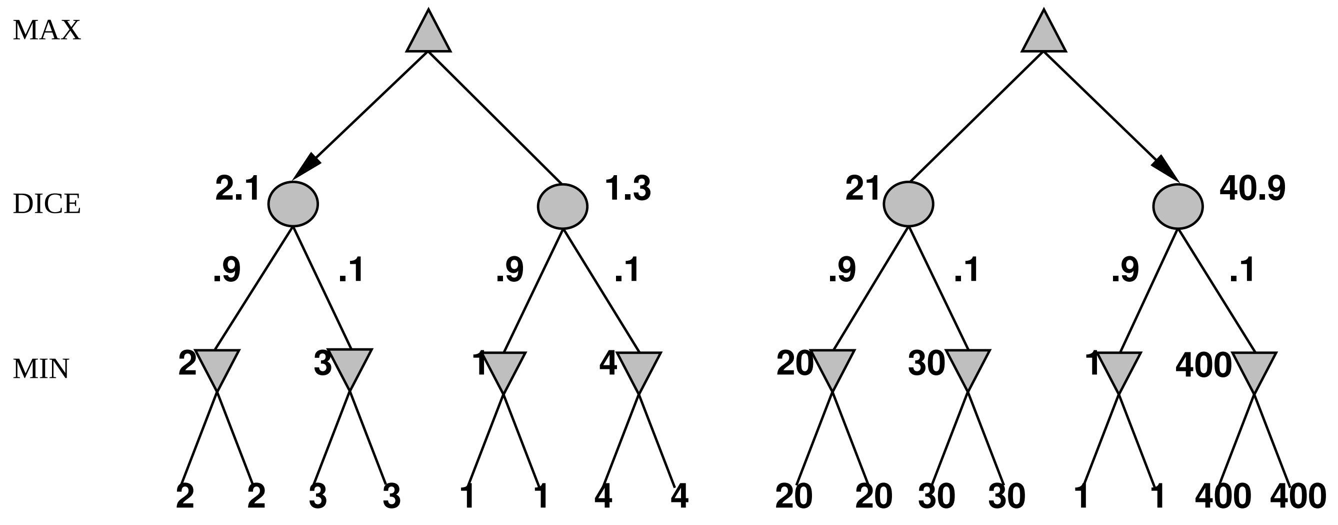
A further generalization — expectiminimax

A complete probabilistic generalization of the minimax algorithm considers alternating moves of the players and random draws. The algorithm for the analysis of such game search tree is called the **expectiminimax**.



The heuristic evaluation in expectiminimax

Let us note a different property of an evaluation function. For minimax, the move choice is the same for all functions with the same order of the graph nodes. This property does not hold for expectiminimax, as seen in the figure below. The moves designated in the presented trees are different, while it would be identical if not for the chance moves.



In the case of expectiminimax the evaluation function may not be an arbitrary function correctly sorting the positions. In fact, it should reflect the expected win (or its linear transformation).

Games with incomplete information

Example: various card games.

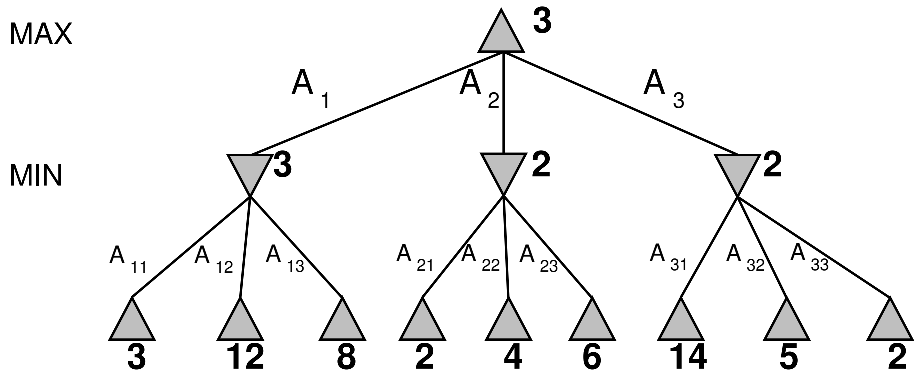
One can compute the probability distribution of all combinations of the deal.

The idea: compute the minimax value of each possible action for each possible deal, and then select the action maximizing the expected value computed over all possible deals.

Best bridge programs implement this by generating many deals consistent with the knowledge gained from the bidding and play so far, and select the action which maximize the number of those won.

Short review

1. For the following two-person game search tree, write a precise sequence of the evaluation function values computed by the minimax algorithm with alpha/beta cuts (order left to right).



Constraint satisfaction problems

The **Constrained Satisfaction Problems** (CSP) are a special group of state space search problems defined as follows:

- a finite set of variables $X = \{x_1, x_2, ..., x_n\}$
- for each variable x_i, a finite set of its possible values, called its **domain**
- a finite set of **constraints** for the combination of values of the variables, eg. if $x_1 = 5$, then x_2 must be even, and the combination $(x_1 = 5, x_2 = 8, x_3 = 11)$ is disallowed

A solution of a CSP problem is any combination of variable values satisfying all the constraints.

Let us note, that the CSP problems are really a special case of a general state space search problems if we treat the set of constraints as a goal specification, and assigning values to variables as state transition operators. Therefore, all algorithms introduced earlier can be applied to these problems.

Constraint satisfaction problems (cntd.)

Examples of CSP problems are: graph or map coloring, the 8-queen problem, the SAT problem (assigning 0 or 1 values to variables in a logical formula to satisfy the formula), cryptoarithmetic, VLSI design, the node labeling problem (for object recognition in images after edge detection), task queueing, planning, and many others.

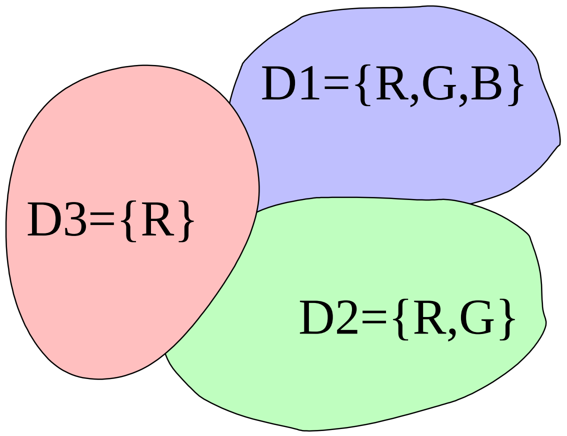
Many of them are NP-hard problems.

A CSP problem may have a solution or not, or there may exist many solutions. The goal may be to find one solution, all of the solutions, or the best solution according to some cost function.

The constraints in a CSP problem may be assumed to be binary, ie. constraining pairs of variables. If there are other constraints in a CSP problem, then n-ary constraints (for $n > 2$) can be converted to equivalent binary constraints, and unary constraints can be built into their respective variables' domains and dropped.

Local constraint satisfaction

Let's consider the map coloring problem. We have to assign colors to areas in a given map from the sets of allowed colors, possibly different for different areas, so that adjacent areas have different colors.



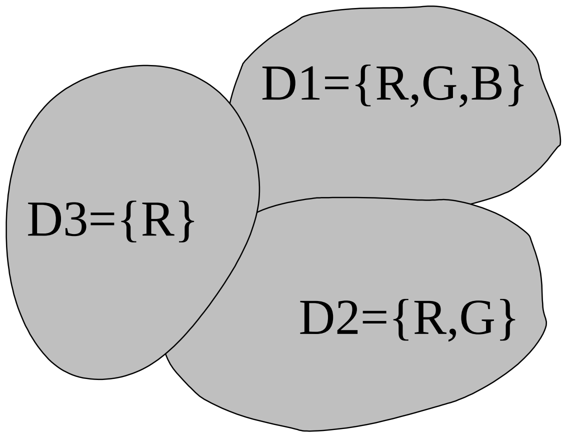
Before we start searching the space of possible value assignments to variables, we can conduct some **local** constraint satisfaction analyzes.

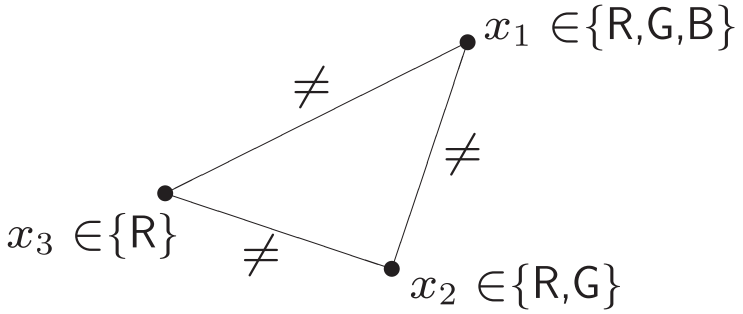
Let's consider the **constraint graph** of a CSP problem, whose nodes correspond to the variables, and edges to the (binary) constraints of the original problem. We consider an edge in this graph as a pair of complementary directed edges, and define a directed edge $x_i \longrightarrow x_j$ of the graph to be **arc consistent** iff $\forall x \in D_i \; \exists y \in D_j$ such that the pair (x, y) satisfies all the constraints existing for the edge.

An inconsistent arc can be brought into consistency by removing specific values from the domains of some variables (specifically, those $x \in D_i$ values for which there does not exist a $y \in D_j$ value satisfying some specific constraint).
This works to reduce and simplify the original problem.

Arc consistency

Let's consider the following example map coloring problem:
$D_1 = \{R, G, B\}$,
$D_2 = \{R, G\}$,
$D_3 = \{R\}$,
$\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\}$.





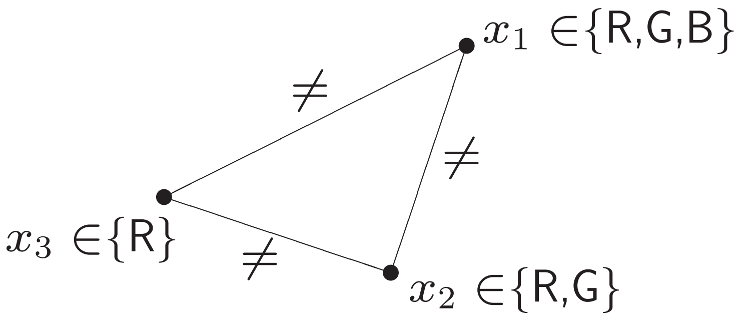
The arc $(x_1$—$x_2)$ is arc consistent, since both $\forall x \in D_1\ \exists y \in D_2\ x \neq y$ and $\forall y \in D_2\ \exists x \in D_1\ x \neq y$ hold.

The fact that arc consistency holds is a mixed blessing. It means that the constraint satisfaction checking of a specific arc in the graph does not contribute to solving the problem. However, a full analysis of the whole CSP constraint graph can sometimes give quite useful results.

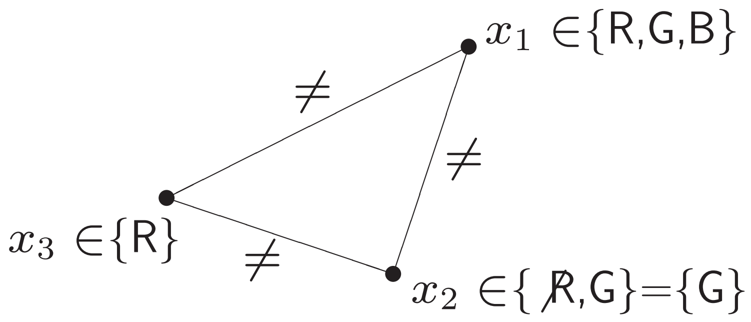
An example: map coloring

We again consider the map coloring problem: $D_1 = \{R, G, B\}$, $D_2 = \{R, G\}$, $D_3 = \{R\}$, $\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\}$.

Analyzing the first constraint ($x_1 \neq x_2$) gives nothing because, as previously noted, this edge is arc consistent. (For each value from D_1 there is a value in D_2 which satisfies the constraint, and the other way around.)

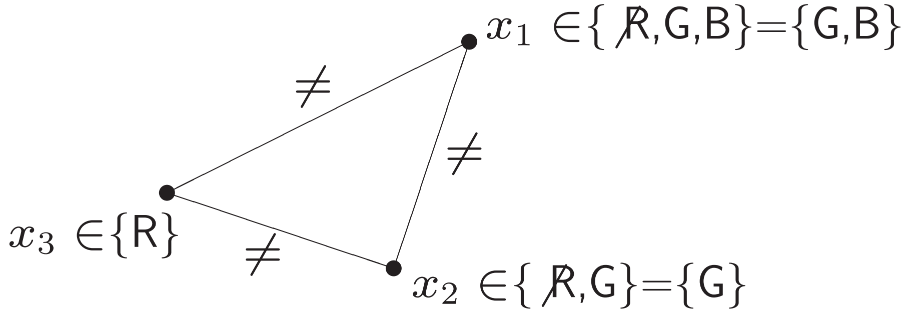


However, analyzing the second constraint ($x_2 \neq x_3$) gives some useful results. Even though for $x_3 =$ R there exists corresponding values for x_2, for $x_2 =$ R there is not a value for x_3 satisfying that constraint. So the value R can be removed from the domain of x_2.



An example: map coloring (cntd.)

A similar analysis for the constraint $(x_1 \neq x_3)$ permits to strike from the domain of x_1 the value R:



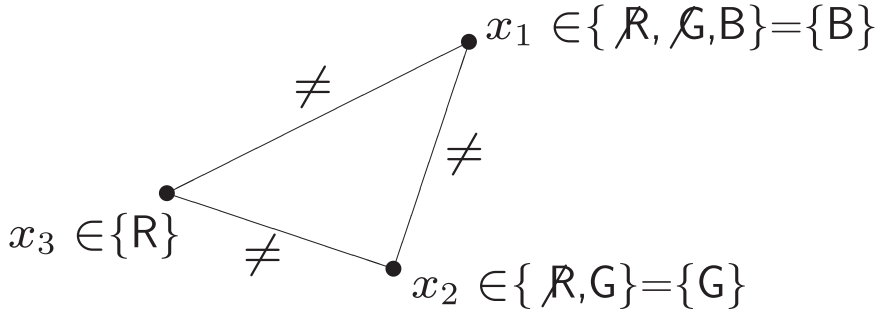
Analyzing all the constraints ended with a partial reduction of the variables' domain. The problem has been simplified (there are fewer possible value assignments to variables), but there still exists more than one potential solution.

But it is easy to observe that the arc consistency checking could, and should, be continued.

Constraint propagation

Since the arc consistency checking results in the reduction of the domains of some variables, it makes sense to repeat the process for the constraint graph edges which were originally consistent, or which have been made consistent. This leads to the **constraint propagation**, which means repeating consistency checking as long as values continue to be removed from variables' domains.

The constraint propagation in the map coloring example causes the edge (x_1, x_2) — originally consistent — to remove the value G from the domain D_1:

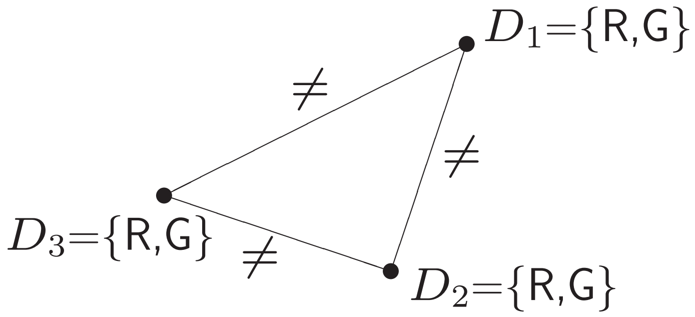

$x_1 \in \{\cancel{R}, \cancel{G}, B\} = \{B\}$
$x_3 \in \{R\}$
$x_2 \in \{\cancel{R}, G\} = \{G\}$
$\neq$
$\neq$
$\neq$

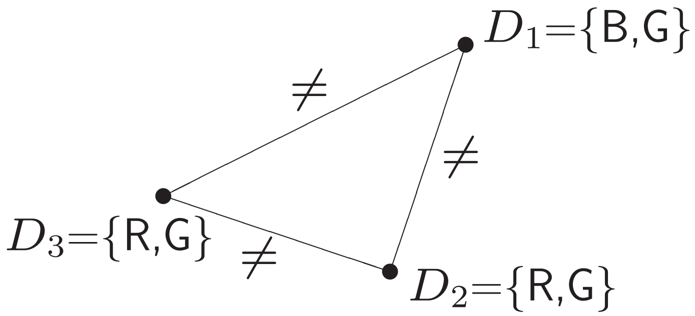
Finally, all the variables have singleton domains, and, furthermore, all the values satisfy all the constraints. Thus the constraint propagation in this case helped solve the problem and determine the unique solution.

In general, consistency checking and constraint propagation lead merely to a simplification, and not necessarily to a complete solution, of a problem.

Constraint propagations — the unsolved cases

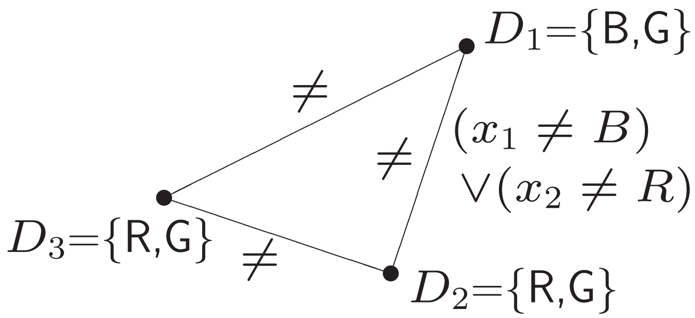
It is easy to notice, that in another instance of the map coloring problem presented here, all arcs are consistent. Nevertheless, the problem has no solution.





In still another instance all arcs are again consistent. The problem has two solutions, and the constraint propagation does not help in determining them explicitly, not does it result in any reductions.

By adding to the previous problem the constraint: $(x_1 \neq B) \vee (x_2 \neq R)$, we obtain yet another instance, in which only one solution is valid, but it still cannot be determined by constraint propagation.



So computing arc consistency and constraint propagation do not by themselves guarantee determining a solution of a CSP problem. It is necessary to search.

Algorithms for computing arc consistency

The easiest approach to compute the arc consistency is to take each constraint, in turn, and testing the logical conditions of the constraints. But since this may have to be repeated due to propagation, even for a single edge, there are a lot of computations. Some savings are possible.

It can be observed, that after a reduction of some domain D_i the propagation can give new results only by checking the edges of the form (D_k, D_i), so just these needs to be checked. What's more, with any reduction in D_k there is no need to check the edge (D_i, D_k), since the elements removed during this reduction from D_k were not necessary for any constraint satisfaction for any of the elements of D_i. The algorithm computing the constraint propagation this way is called AC-3 (1977).

When an arc's consistency is checked again, the same conditions are evaluated for the same pairs of values. Memorizing these verified value pairs (in an proper data structure) could help refrain from recomputing them during subsequent propagations. This is accomplished by yet another algorithm called AC-4 (1986).

Non-binary constraints

We initially assumed to restrict the analysis to binary constraints, ie. such binding exactly two variables. The non-binary constraints can be converted to binary ones.

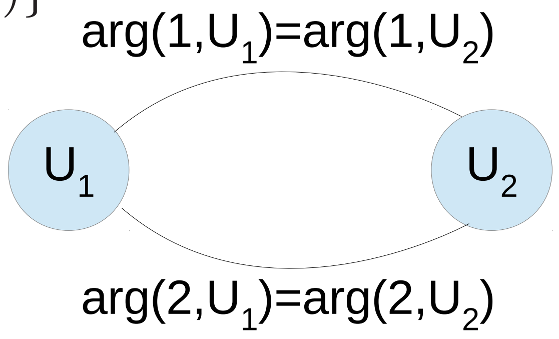
One of the simplest conversion schemes is the **dual encoding**. It works by introducing a new variable for each constraint of the original problem. The domain of a new variable is the set of n-tuples of values of the original variables satisfying the original constraint.

This way the values contained in the new variables by definition satisfy each constraint (original) in separation. We just need to make sure the values satisfy all the constraints. For this, the dual encoding introduced new constraints, which occur between each pair of new variables, containing the same variables (original). The constraints state, that the same variables (original) must have the same values.

Non-binary constraints — example

Consider a CSP example with three variables: $X = \{x, y, z\}$, their domains $D_{x,y,z} = \{1, 2, 3\}$, and two constraints: $\mathcal{C} = \{x + y = z, x < y\}$. The dual encoding of this problem contains two variables and two constraints:

$$\begin{aligned}
U_1 &: \langle oc_1, [x, y, z]\rangle, D_{U_1} = \{(1,2,3), (2,1,3), (1,1,2)\} \\
U_2 &: \langle oc_2, [x, y]\rangle, \quad D_{U_2} = \{(1,2), (1,3), (2,3)\} \\
C_1 &: arg(1, U_1) = arg(1, U_2) \\
C_2 &: arg(2, U_1) = arg(2, U_2)
\end{aligned}$$



Unfortunately, the consistency analysis of this problem does not yield anything, because for each value of one variable there exist values of the other with satifying subsequent arguments. However most values othe dual variables are n-tuples failing the original constraints.

Generally, converting multi-variable to binary constraints sometimes leads to problems that do not lend themselves to consistency analysis. For this reason, several algorithms have been developed for arc consistency with multi-variable constraints. These algorithms will not be discussed here.

Path consistency

We define for a CSP constraint graph the notion of **K-consistency**. A graph is K-consistent (for some K), if for any (K-1) variables, which among themselves have all the constraints satisfied, for any (K-1)-tuple of values of these (K-1) variables satisfying all the constraints for the (K-1) variables, in the domain of any selected K-th variable a value such, that the so-obtained K-tuple of values satisfies all the constraints for the K variables.
A constraint graph is **strongly K-consistent** if it is K-consistent for any J, J<K.

Note that the previously defined arc consistency is equivalent to the strong 2-consistency of a constraint graph.

The strong 3-consistency of a graph is also called a **path consistency**.

The significance of K-consistency is such, that if a CSP problem constraint graph with n nodes is strongly n-consistent, then the problem can be solved without searching. However, the algorithms for enforcing K-consistency are exponential, so it is seldom worthwhile to do that. An exception is a weaker version of path consistency — the **restricted path consistency**, for which there is an algorithm which is sometimes computed.

Searching in the CSP problems

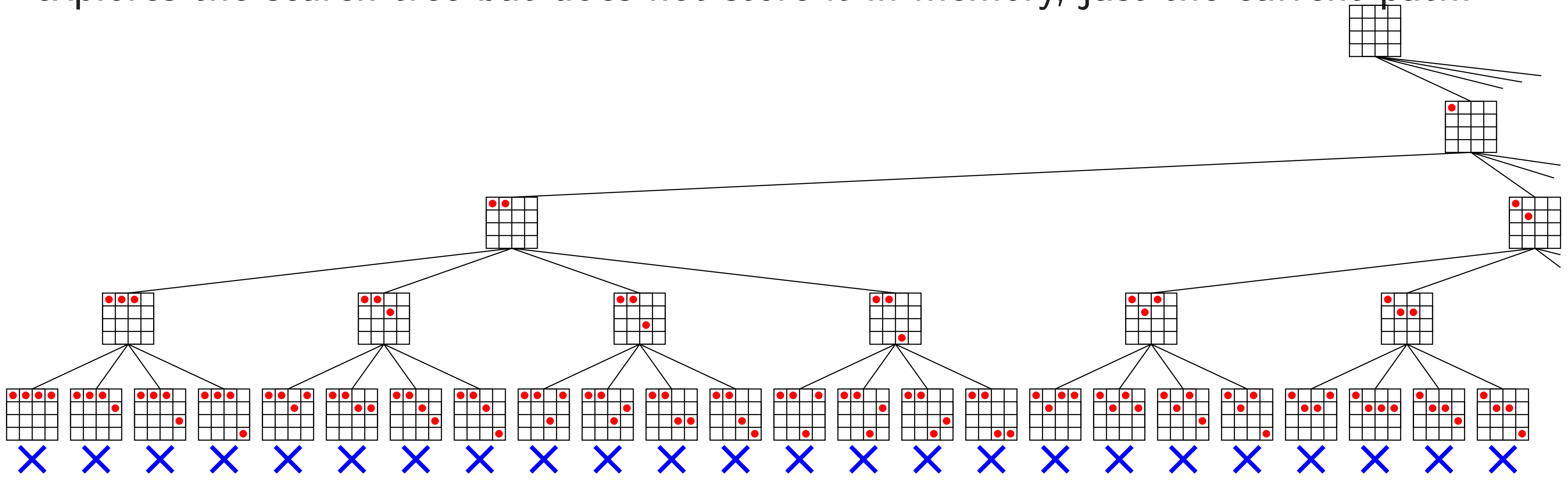
Any of the previously discussed searching algorithms may be used for the CSP problems. However, in most really hard CSP problems, where the constraints have the nature of hard to meet, tight compromises, the most important is just the analysis of these constraints, both syntactic and semantic.

On the other hand it is typically hard to come up with a useful heuristic, capable of guiding the process of searching the space of value assignments to the variables.

Therefore often used is the simplest of the searching algorithms, the backtracking search (BT). In place of a good heuristic prioritizing the best choices to be at the front of the list, this algorithm may be augmented by a local constraint satisfaction checking. This reduces the number of choices for the subsequent steps. In the extreme case, when the domain of some variable got reduced to an empty set, the algorithm would immediately backtrack to the alternative values in earlier assignments.

Example: the 4 queen problem

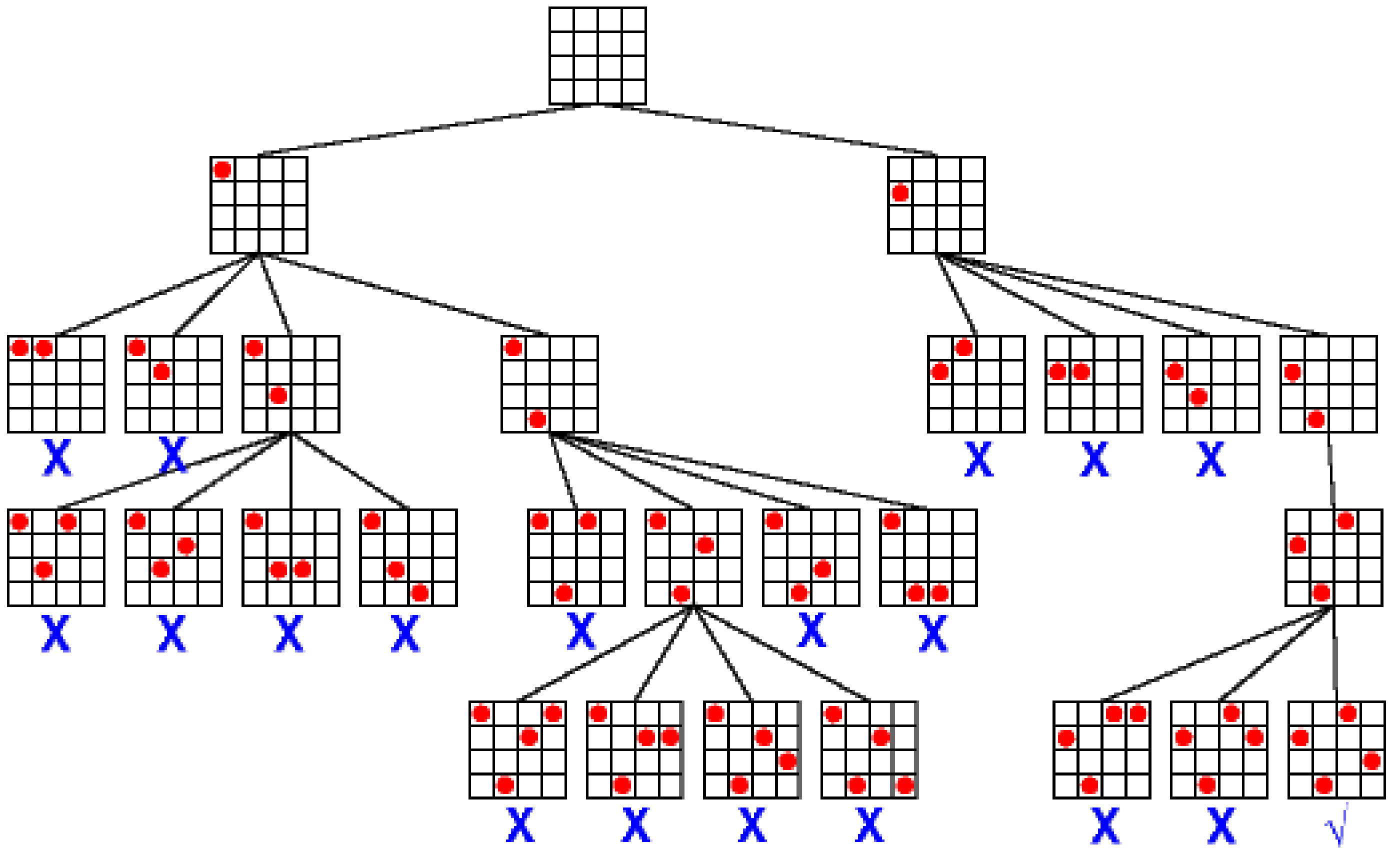
Let's now consider the application of the BT (backtracking) algorithm to the 4 queen problem. We formulate the problem to assign the row positions to the 4 queens belonging to the different columns of the 4×4 chessboard. Note the BT algorithm explores the search tree but does not store it in memory, just the current path.



The algorithm checks the constraints after placing all the queens on the board. It will surely solve the problem, but makes many unnecessary steps, which could be eliminated. For example, all the terminal configurations are invalid due to the placement of the second queen. This can be seen at depth level 2 already.

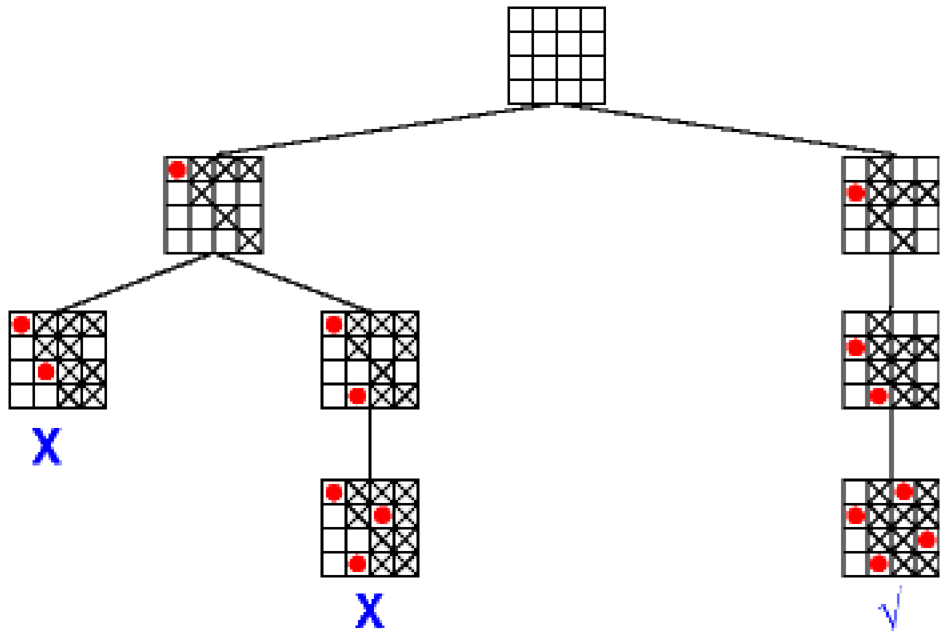
Example: the 4 queen problem (cntd.)

An obvious improvement to the algorithm is then to test the constraints on all variables as soon as they have been assigned values. Should any constraint be found to be violated, the value assignment most recently made would immediately be dropped, and the algorithm would backtrack. This algorithm will be called **early checking** (BT-EC). It is obviously advantageous to the BT algorithm, since the tested constraints would have to be later checked anyway.



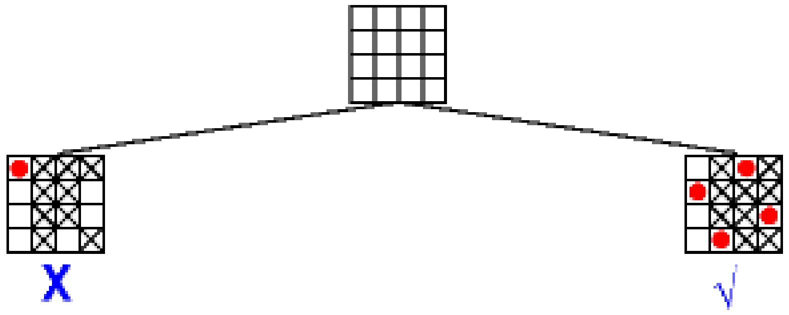
Example: the 4 queen problem (cntd.)

Combining the backtracking search with just the minimal form of the local constraint satisfaction checking is called the *forward checking* (BT-FC) algorithm. All the constraints for any variable assigned a value are checked, and only those. In most cases this algorithm is advantageous to BT-EC, and certainly to BT.



Example: the 4 queen problem (cntd.)

It is possible to apply the full arc consistency checking, with propagation. The algorithm doing that is sometimes called the *look-ahead* (BT-LA) algorithm. It may significantly reduce the size of the explored search space, as it does in the 4-queens example here. However, the cost performing those checks is significant, and the BT-LA may not always be advantageous to the BT-FC algorithm.



Dependency-directed backtracking

In searching the CSP tree we may encounter a failure, causing the BT algorithm to backtrack, whose cause was not the most recently selected assignment, but one of the earlier steps. In such case the algorithm will continue trying various possibilities, generating only failures, until it backtracks sufficiently, and changes the assignment of the offending variable.

It is possible to detect such cases, when the set of variables involved in constraints with the current variable — the **conflict set** — does not include the most recently assigned variable. In these cases, the algorithm could backtrack, not just a single step, but all the way to the most recently assigned variable from the conflict set. Such algorithm is called **backjumping** (BJ).

Simple backjumping currently has only historical value, since it solves the problem, which does not arise in practice, since the arc consistency checking starting from BT-FC eliminate those cases completely. However, backjumping is still useful with a slightly extended concept of the conflict set, defined as a set of those variables, whose assigned values caused a constraint failure of the current variable, along with the subsequently assigned variables. A version of BJ based on such definition is called **conflict-directed backjumping**, and it is capable of determining the backjumping steps where consistency checking does not help.

Dynamic ordering

We have noted earlier, that is is difficult to obtain good heuristics indicating good moves in searching the space of most CSP problems. There do exist, however, other techniques augmenting this search, based on **dynamic ordering**, both of variables to select those which should first receive assignments, and of values, which should be tried first.

The **most constrained variable** heuristic (or MRV, for Minimum Remaining Values), suggests to first select those variables with the smallest domains. Such choice gives the best chance of encountering inconsistencies, and taking advantage of the resulting reductions. This heuristic also works well within the BT-FC algorithm.

Another heuristic which may be useful in selecting a variable is the **degree heuristic**, suggesting the variable occurring in the highest number of constraints with unassigned variables.

Once a variable to assign is chosen, the **least constraining value** heuristic may be used which prefers to choose those values, which exclude the least values of other variables.

Local search for CSP

Another approach which works well with some CSP problems is based on local search. After more or less random choice of an initial value assignment for all variables, an incremental repair is attempted. Greedy hill-climbing search may be used, which does not explore the search space systematically, unlike the BT family of algorithms.

Often successful in such search for CSP problems is the **min-conflict** heuristic which works by randomly selecting a variable violating some constraint, and selecting another value for it, so that it would minimize conflicts (number of failed constraints) with other variables.

Some CSP problems can be solved with surprising efficiency using this approach. The key element to success is the randomness, which helps to escape the local maxima, and other traps, and to select the right variable to repair, or to skip an unfortunate variable choice, for which the right value would better be assigned later.

Short review

1. Consider the CSP problem with four variables: A, B, C, D, with domains: $\{1, 2, 3\}$ for each, and the set of constraints given below. Draw the constraint graph for the problem, and then try to solve it using constraint propagation (arc consistency). Show each step of the solution (no picture). Show the graph after the termination of constraint propagation. How many possible CSP problem solutions does it represent? Write down one of them.

   The constraint set:
   $\mathcal{C} = \{C \neq D, B > D, B > C\}$

Useful resources

A good elementary introduction to CSP problems by Roman Barták
`http://ktiml.mff.cuni.cz/~bartak/constraints/constrsat.html`