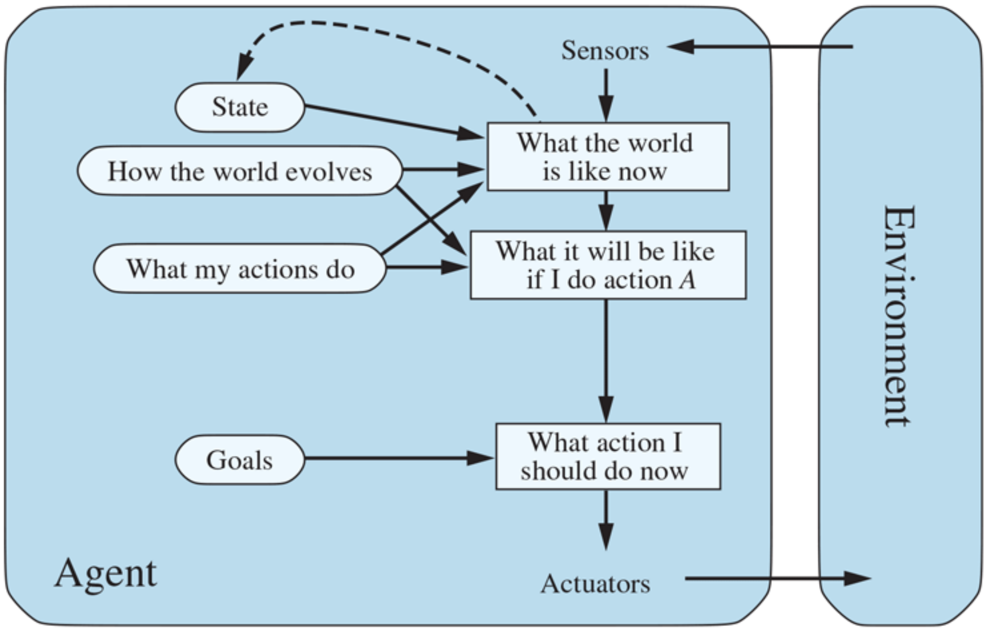
AI FIRST MID-TERM

Goal-Based Agent

The agent needs some sort of **goal** information that describes situations that are desirable. The agent program can combine this with the model to choose actions that achieve the goal.



Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action *A*
Environment
Goals
What action I should do now
Agent
Actuators

Sometimes it will be tricky when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal.
Notice that decision making of this kind is fundamentally different from the condition, action depends on the consideration of the future.
Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. For example, a goal-based agent’s behaviour can easily be changed to go to a different destination, simply by specifying that destination as the goal. The reflex agent’s rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

Utility-Based Agent

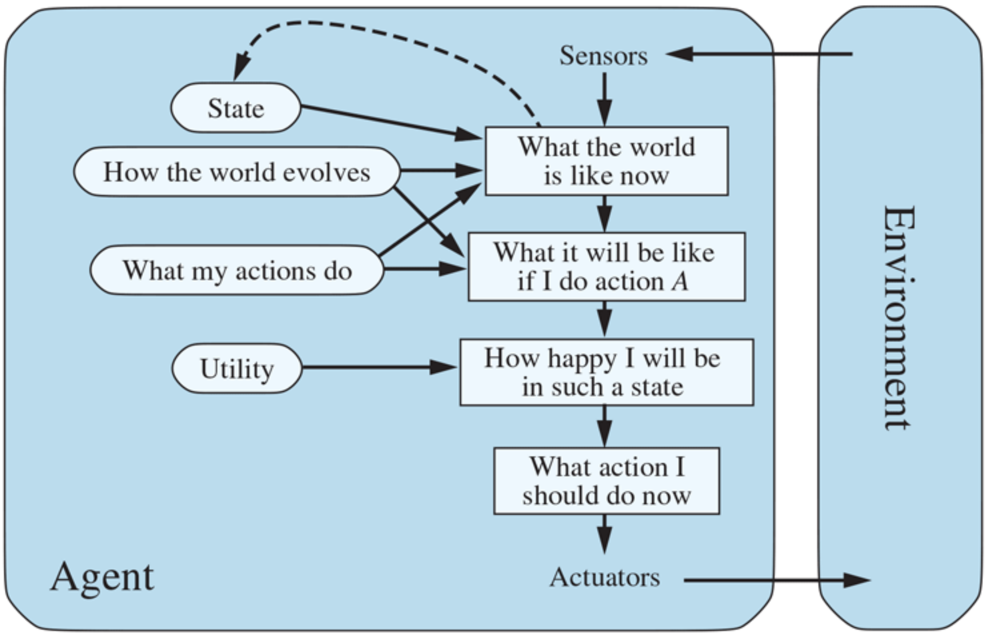
Goals alone are not enough to generate high-quality behaviour in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal), but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between “happy” and “unhappy” states.

We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi’s destination. An agent’s utility function is essentially an internalization of the performance measure. Provided that the internal utility function and the external performance measures are in agreement, an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate trade-off. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

Technically speaking, a rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes.

An agent that possesses an explicit utility function can make rational decisions with a general-purpose algorithm that does not depend on the specific utility function being maximized.


Sensors
State
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

It’s true that such agents would be intelligent, but it’s not simple. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning.

Uninformed algorithms

Uninformed strategies use only the information available in the problem definition:

- Breadth-first search
- Depth-first search
- Depth limited search
- Iterative deepening search

*[**b** = maximum branching factor of the search tree,*

***d** = depth of the least-cost solution,*

***m** = maximum depth of the state-space,*

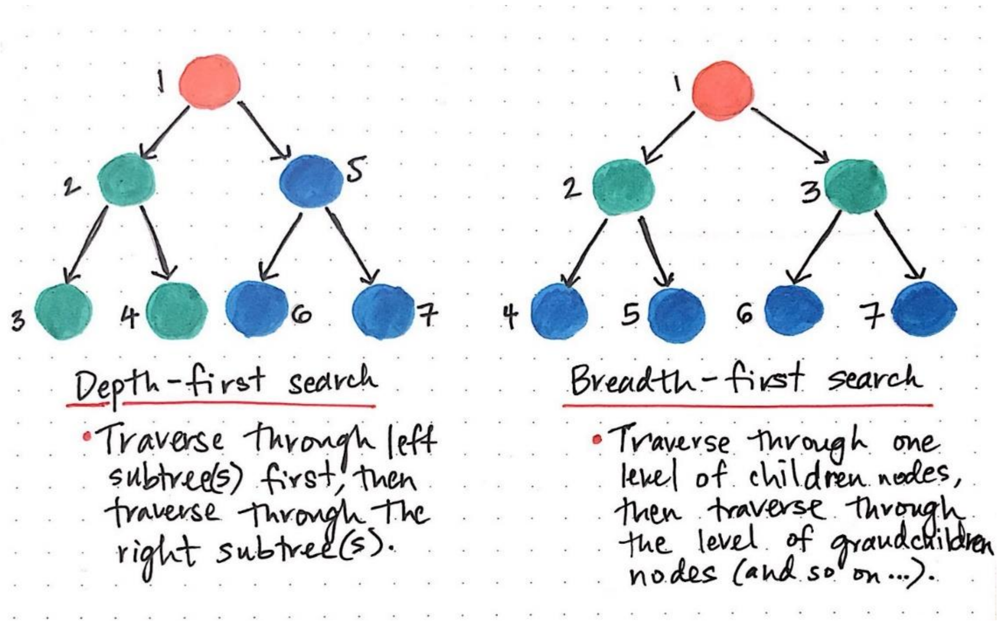
***l** = limited depth]*

Breadth-first search

The tree is explored looking first at the root-node, then at all children of the root node, then at their successors and so on and so for.

Due to explore nodes, the iterative implementation uses a FIFO strategy.

It is complete; it is optimal, that means it finds a solution with the minimum amount of actions, finding the shallowest solution; it finds an optimal solution in $O(b^d)$, and it occupies $O(b^d)$ of memory.


1
2
5
3
4
6
7
Depth-first search
• Traverse through left subtree(s) first, then traverse through the right subtree(s).
1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Depth-first search

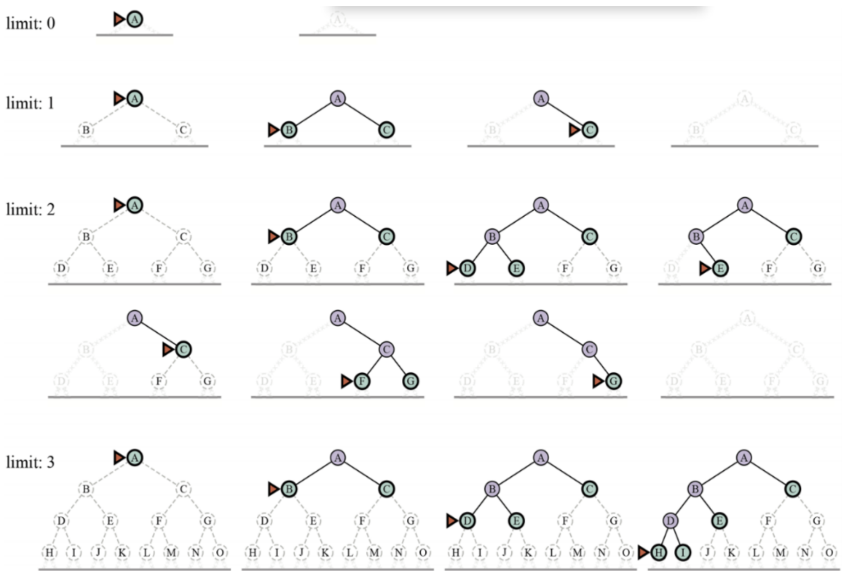
The search proceeds exploring first the deepest node of the frontier and then backing up to the second deepest node of the frontier and so on and so for, acting in a LIFO way. It is not complete because it could continue to expand nodes always deeper getting stuck in cycles or trying to explore infinite states. It is not optimal because going deeper first, it could miss some shallower solutions. It finds a solution in $O(b^m)$, and it occupies $O(b * m)$ of memory.

Depth limited search

The search is like depth-first search but there is a limit in the tree's deep. In this way, the searching process cannot continue infinitely. If there is not a solution to a depth minor than the depth limit, the algorithm is not able to find any solution, otherwise, yes; so it is not complete. It is not optimal for the same reason for its incompleteness. It finds a solution in $O(b^d)$, and it occupies $O(b * l)$ of memory.

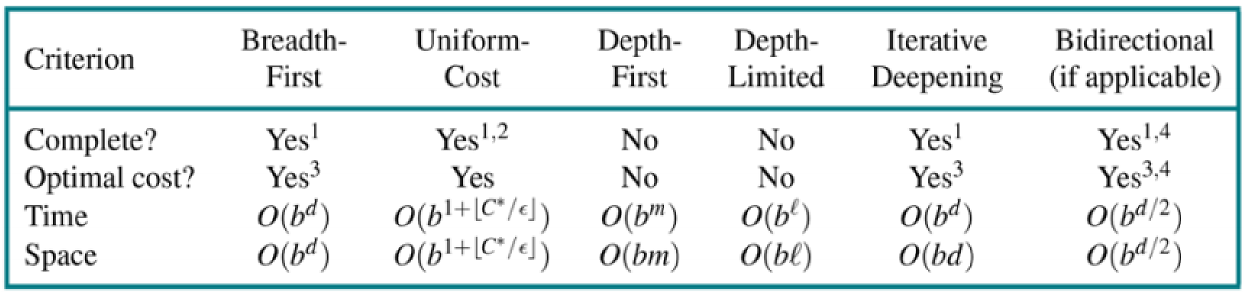
Iterative deepening search (IDS)

The search is an iteration of several depth limited searches, starting from l = 1 and increasing it until the shallowest solution is found. It is complete because it finds a solution acting in a similar way of breadth-first search, finding the shallowest solution, with more state visited but with lower memory occupation at the same time. The complexity is $O(b^d)$ for time and $O(b^d)$ for memory occupation.



Bidirectional search

An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d . For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state is a successor of in the forward direction, then we need to know that is a successor of in the backward direction. We have a solution when the two frontiers collide.



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \varepsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

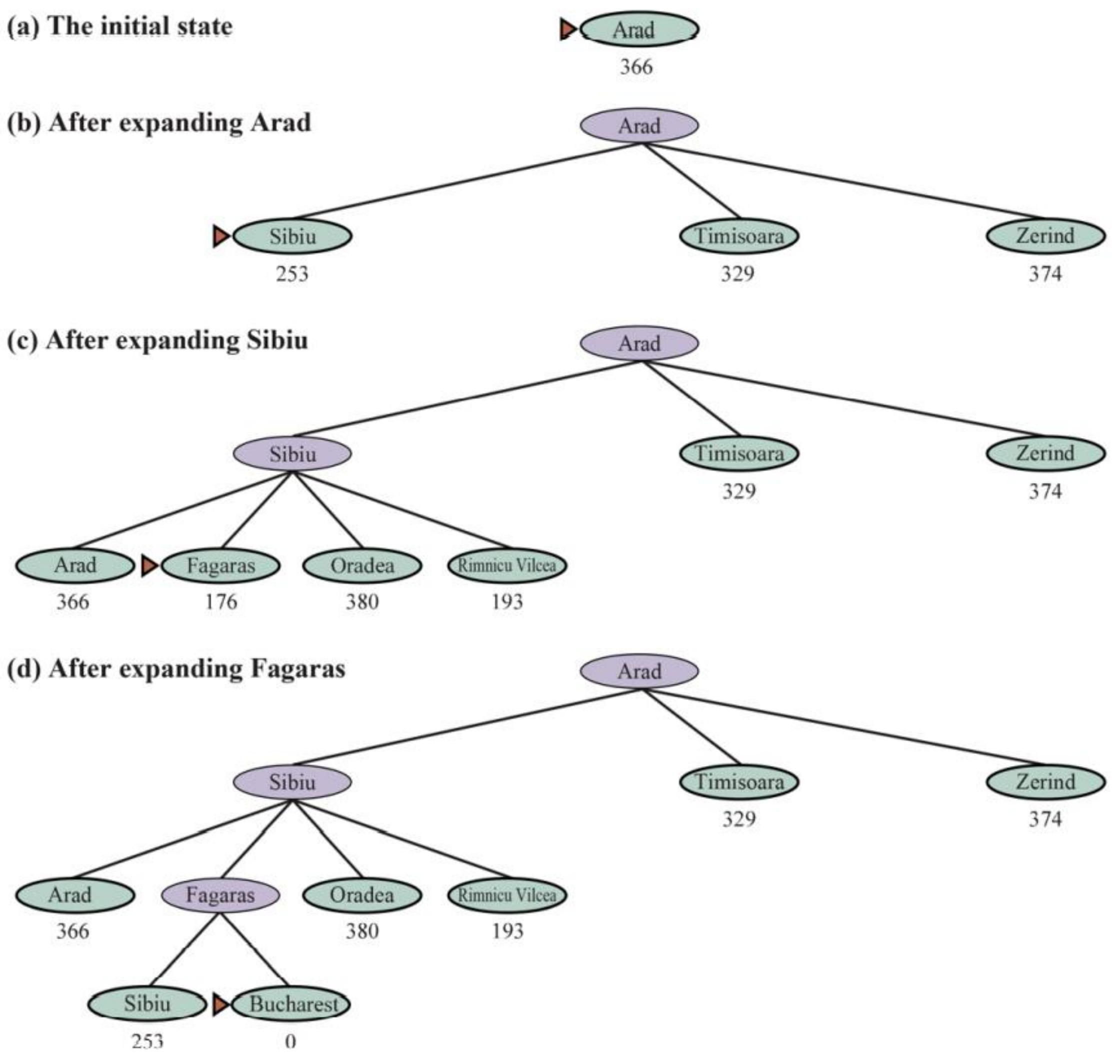
Heuristic search

Idea: use an evaluation function for each node – estimate of "desirability": expand most unexpanded node desirable. Evaluation function *h(n)* (heuristic) = estimate of cost from *n* to the closest goal.

Greedy best-first

Greedy best-first search is a form of best-first search that expands first the node with the lowest value *h(n),* ergo the most desirable node —the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. It means that the evaluation function *f(n)=h(n).* The fringe is a queue sorted in decreasing order of desirability.

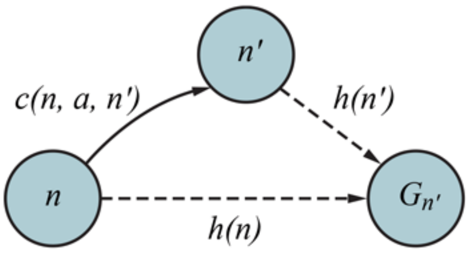
It is not complete, can get stuck in loops. Complete in finite space with repeated-state checking. It has time complexity *O(bm),* but a good heuristic can give dramatic improvement. It has space complexity *O(bm)*—keeps all nodes in memory. It is no optimal.

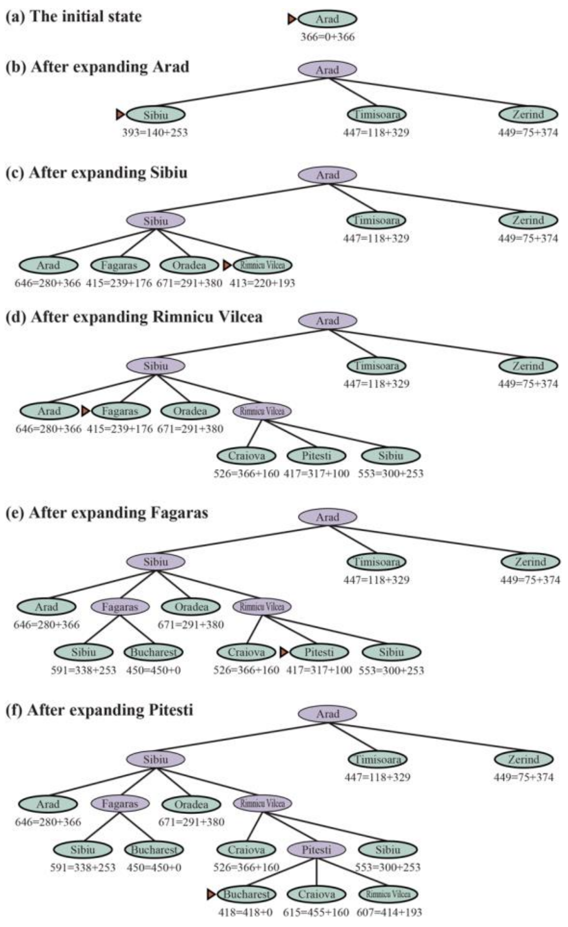

(a) The initial state
Arad
366
(b) After expanding Arad
Arad
Sibiu
253
Timisoara
329
Zerind
374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193
(d) After expanding Fagaras
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

A* search

The most common informed search algorithm is A* search, a best-first search that uses the evaluation function: $f(n) = g(n) + h(n)$

Where $g(n)$ is the path cost from the initial state to node *n*, and $h(n)$ the estimated cost of the shortest path from *n* to a goal state; so $f(n) =$ *estimated cost of the best path that continues from n to a goal.* NB: a key property is *admissibility*: an admissible heuristic is one that never overestimates the cost to reach a goal. A slightly stronger property is called consistency. A heuristic $h(n)$ is consistent if, for every node *n* and every successor *n′* of *n* generated by an action *a* we have: $h(n) \leq c(n, a, n') + h(n')$.





Properties of A*

- Complete - Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time - Exponential in [relative error in $h \times length\ of\ soln.$]
- Space - Keeps all nodes in memory
- Optimal - Yes—cannot expand $fi + 1$ until fi is finished
    - A* expands all nodes with $f(n) < C^*$
    - A* expands some nodes with $f(n) = C^*$
    - A* expands no nodes with $f(n) > C^*$

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.

Iterative-deepening A* (IDA*)

IDA* gives us the benefits of A* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems that do not fit in memory. In IDA* the cut-off is the $f - cost$ $(g + n)$; at each iteration, the cutoff value is the smallest $f - cost$ of any node that exceeded the cutoff on the previous iteration. In other words, each iteration exhaustively searches an $f - contour$, finds a node just beyond that contour, and uses that node's -cost as the next contour.

Recursive best-first search (RBFS)

RBFS resembles a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f_limit variable to keep track of the $f - value$ of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the $f - value$ of each node along the path with a **backed-up value**—the best $f - value$ of its children. In this way, RBFS remembers the $f - value$ of the best leaf in the forgotten subtree and can therefore decide whether it's worth re-expanding the subtree at some later time. RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. These mind changes occur because every time the current best path is extended, its $f - value$ is likely to increase $-h$ is usually less optimistic for nodes closer to a goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA* and could require many re-expansions of forgotten nodes to recreate the best path and extend it one more node.

Simplified Memory-Bounded A* (SMA*)

The problem with IDA* and RBFS is that they use too little memory:

- in IDA* at each iteration only the current cost limit for f is kept
- in RBFS the cost of the nodes in the depth first search are recorded.

SMA* instead can use all the available memory for the search. Idea: better remember a node that re-generate it when needed. When a new node must be generated the node with the highest f is discarded (forgotten node), while keeping its cost f in the parent node. A discarded node will be re-generated only when all other paths are worse than the forgotten node.

*Properties of SMA**

- It uses all the available memory
- It avoids repeated states until the available memory is exhausted.
- Complete – if the available memory allows the shallowest solution path to be recorded.
- Optimal – if the available memory allows the shallowest optimal solution path to be recorded. Otherwise it returns the best solution attainable with the available memory.
- Optimal Efficient - if the available memory can store the whole search.

Local search

Local search algorithms operate by searching from a start state to neighbouring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages: (1) they use very little memory, and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable. Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function.

Hill Climbing

It keeps track of one current state and on each iteration moves to the neighbouring state with highest value—that is, it heads in the direction that provides the steepest ascent. It terminates when it reaches a "peak" where no neighbour has a higher value. Hill climbing does not look ahead beyond the immediate neighbours of the current state. Hill climbing is sometimes called greedy local search because it grabs a good neighbour state without thinking ahead about where to go next. Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state. Unfortunately, hill climbing can get stuck for any of the following reasons:

- **LOCAL MAXIMA**: A local maximum is a peak that is higher than each of its neighbouring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
- **RIDGES**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **PLATEAUS:** A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible.

In each case, the algorithm reaches a point at which no progress is being made. There are some possible solutions:

- Side moves with a limit on the maximum number #.
- Stochastic Hill Climbing: random moves (choosing among the uphill successors).
- First choice: random generation of successors taking the first one uphill.
- Random restart.

Success is strongly related to the "shape" of the state space

Simulated annealing

In general

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum. In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum but will be extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing is such an algorithm. In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum—perhaps into a deeper local minimum, where it will spend more time. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature). It is similar to Hill Climbing Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted.

Local beam search

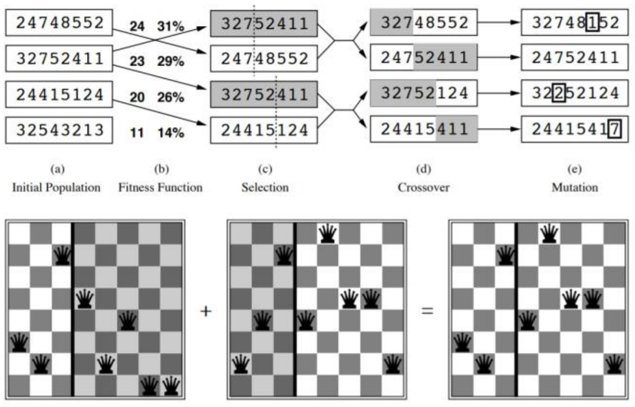
Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats. Problem: too quick convergence in the same region of the search space; the stochastic beam search randomly chooses k successors weighting more the most promising ones.

Genetic Algorithms

Idea: organisms evolve; those adaptable to the environment survive and reproduce, others die (Darwin).

1) initial population: individuals or chromosomes
2) selection: by fitness function
3) reproduction: crossover
4) reproduction: mutation

Search in the space of individuals. Steepest ascent hill-climbing since little genetic alterations are performed on selected individuals.
To deploy Genetic Algorithms, we must define the previous 4 points.



Constraint satisfaction problems (CSPs)

The search algorithms consider each state in an atomic way (indivisible, a black box with no internal structure). Now we break open the black box by using a factored representation for each state: a set of *variables*, each of which has a *value*. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a *constraint satisfaction problem*, or *CSP*.

CSP:

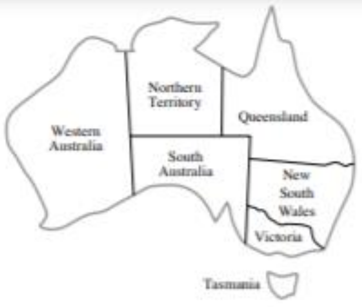
A CSP is a triple $< X, D, Ci >$, where:
X is a set of variables, $\{X1, \dots, Xn\}$.
D is a set of domains, $\{D1, \dots, Dn\}$, one for each variable.
C is a set of constraints that specify allowable combinations of values. Each constraint Cj consists of a pair $< scope, rel >$, where $scope$ is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

State is defined by variables Xi with values from domain Di
Goal test is a set of constraints specifying allowable combinations of values vi
for subsets of variables.
Solution is an assignment $\{Xi = vj, \dots\}$ that does not violate the constraints.
Key feature: general-purpose algorithms with more power than standard search algorithms.
Different variables have different domains:
Infinite domains can be reduced to finite by putting an upper bound to values. Unary constraints involve a single variable, e.g., SA 6= green. Binary constraints involve pairs of variables, e.g., SA 6= W A.

Binary CSP: each constraint relates at most two variables.
Constraint graph: nodes are variables, arcs show constraints.
General-purpose CSP algorithms use the graph structure
to speed up search.

<u>Classic approach/Standard formulation</u>

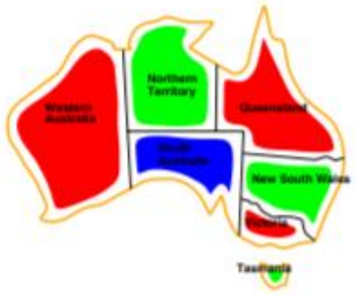
Straightforward approach: states are defined by the values assigned so far.
Initial state: the empty assignment, "{ }"
Successor function: assign value to unassigned variable that does not conflict with current assignment. It fails if no legal assignments (not fixable!). Goal test: the current assignment is complete.



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

AI (Section1) – AI1, Daniele Nardi, 2020-2021 Constraint Satisfaction Problems 10

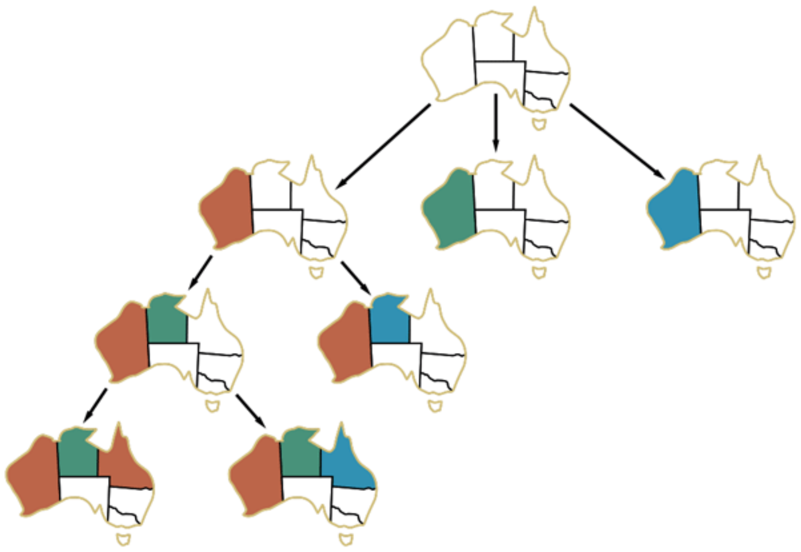
Example: Map-Coloring contd.



Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Backtracking search

Sometimes we can finish the constraint propagation process and still have variables with multiple possible values. In that case we need to search for a solution. *Backtracking search* keep in count the CSP's commutative property, so we need only to consider a single variable at each node in the search tree. At each level of the tree we have to choose which variable we will deal with, but we never have to backtrack over that choice. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call. If the call succeeds, the solution is returned, and if it fails, the assignment is restored to the previous state, and we try the next value. If no value works then we return failure.



General-purpose methods can give huge gains in speed:

- *Minimum remaining values (MRV)*: choose the variable with the fewest legal values, it picks a variable that is most likely to cause a failure soon.
- *Degree heuristic*: choose the variable with the most constraints on remaining variables. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.
- Given a variable, choose the *least constraining value*: the one that rules out the fewest values in the remaining variables (scegli un valore già assegnato in modo tale da lasciarne molti liberi non assegnati).

Forward checking

Idea: Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values. Forward checking propagates information from assigned to unassigned variables but do not provide early detection for all failures.

Arc consistency

Simplest form of propagation makes each arc consistent.
Can be run as a pre-processor or after each assignment: in Recursive-Backtracking after adding a new value an inference step is performed
– forward checking (removing values from the nodes constrained by the assigned var)
– arc consistency (starting with the arcs that connect the
assigned variable with unassigned ones)

PLANNING

State space: Each node represents a state of the real world and a transition is an addiction of an action in the action-sequence to reach a goal.
Plan space: One node represents a partial plan. The state of the search is a partial plan. When you go cross an arch you add an action from the action-set to the partial plan.

Possible method of searching for a plan:

- Progression: apply actions whose preconditions are satisfied until goal state is found or all states have been explored
- Regression: any positive effect of A that appears in G is deleted and each precondition literal of A is added unless it already appears.
- Heuristics search: it is very effective. "Relaxing the problem", there are two possibilities:
  - Removing preconditions -> all the actions are completely free with any constraints.
  - Removing negated effects

Partial Ordered Plans (POP)

Partially ordered collection of actions with
- Start action has the initial state description as its effect
- Finish action has the goal description as its precondition
- temporal ordering between pairs of actions

Two additional elements are needed to characterize the planning
process:

- Open precondition = precondition of an action not yet
causally linked
- Causal links from outcome of one action to precondition of another

A *plan* is *complete* if every precondition is achieved. A *precondition* is *achieved* if: it is the effect of an earlier action and no possibly intervening action undoes it.
A clobberer is a potentially intervening action that destroys the condition achieved by a causal link. A conflict can be solved by adding:
- $C \prec A$ (demotion) or
- $B \prec C$ (promotion)

Nondeterministic algorithm: backtracks at choice points on failure:
– choice of action (Sadd) to achieve open precondition (Sneed)
– choice of demotion or promotion for clobberer

Selection of open precondition (Sneed) is irrevocable: the existence of a plan does not depend on the choice of the open preconditions.
POP is sound, and complete.

General:
- number of open preconditions
-most constrained variable open preconditions that are satisfied in fewest ways
-a special data structure: the planning graph

Problem Specific:
Good heuristics can be derived from problem description (by the
human operator). POP is particularly effective on problems with many loosely related subgoals

HIERARCHICAL TASK NETWORK PLANNING

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together, and state-of-the-art algorithms can generate solutions containing thousands of actions.
Bridging this gap requires planning at higher levels of abstraction.
We concentrate on the idea of hierarchical decomposition, an idea that pervades almost all attempts to manage complexity. The key benefit of hierarchical structure is that at each level of the hierarchy is reduced to a small number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities
for the current problem is small.

HLA HIGH LEVEL ACTIONS

The basic formalism we adopt to understand hierarchical decomposition comes from the area of hierarchical task networks or HTN planning. We assume set of actions, now called primitive actions, with standard precondition–effect schemas. The key additional concept is the high-level action or HLA. Each HLA has one or more possible

refinements, into a sequence of actions, each of which may be an HLA or a primitive action. An HLA refinement that contains only primitive actions is called an implementation of the HLA. We can say, then, that a high-level plan achieves the goal from a given state if at least one of its implementations achieve the goal from that state. The “at least one” in this definition is crucial.
—not all implementations need to achieve the goal, because the agent gets to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning —each of which may have a different outcome—is not the same as the set of possible. Outcomes in nondeterministic planning. There, we required that a plan work for all outcomes because the agent does not get to choose the outcome; nature does.

Describe HLA in HTN and provide example (referring to the previous problem)

A plan can be decomposed in a huge set of atomic primitive action which give only a small informative contribute for the entire plan or they introduce only a small progress with respect to the goal, so we introduce a high-level action.
An HLA is a sequence of primitive action directly runnable by an agent. in real planning problem. Wrt the given problem drop or collect could be a HLA because they are a sequence of movement or also the action go can be modelled in a sequence of primitive action like openCar, turn on Car... or takeBycicle turnLeft ecc ecc... .

Discuss the difference between state-space and plan-space planning. Describe the basic approach to plan space planning (POP)

State space is a transition from a state A to a state B will be caused by the addition of an action which will follow the action that had led to the state A. Instead in the plan space a transition to another node corresponds to inserting an action in the plan that has not to be strictly the following of the previous. In this representation actions can be in every point of the plan, without a sequential order.

The POP search procedure proceeds as follows:
The initial plan is composed only by the Start action and the Finish action, specifying that the start < finish (start comes before then finish)
Pick the open precondition p of an action B and look for an action A whose effect satisfies p. Add the causal link A -> p -> B and specify the ordering A<B (A comes before then B). In addition to this if A is a new action add the orderings Start<A, B<Finish
Run the goal test which would succeeds if there are no more open preconditions.
Solve conflicts, otherwise backtrack.