Domande e risposte teoria AI

Uninformed algorithms

- Breadth-First Search algorithm is an appropriate choice when all actions have the same cost. The root node is expanded first, then its children, then their successors, and so on. It is implemented using a FIFO queue. It is complete even on infinite state space (????) and it is optimal. It always finds the optimal solution in time O(b^d). The space occupied in memory is O(b^d) because it keeps in memory all nodes expanded.

- Depth-First Search algorithm is a tree search algorithm in which the deepest node in the frontier is always expanded first arriving where nodes have no successors; then the search backs up to the next deepest node that has unexpanded successors. It resolves the memory problem of breadth-first search, infact it keeps in memory only O(bm) nodes (where m is the maximum depth of the tree). It is neither complete nor optimal, infact it does not find a solution if the state space is infinite or if it gets stuck in cycles and it returns the first solution that finds even if it is not the cheapest. In general, it finds a solution in O(b^m).

- Depth-Limited Search is a version of depth-first search which avoid the possibility of wandering down an infinite path treating nodes at depth limit "l" as if they have no successors. Thus, the search process is the same of depth-first but with the "l" limit. It is not complete, because the solution could be deeper than the "l" limit of depth. For the same reason it is neither optimal. The space occupied in memory is O(bl) and its complexity is O(b^l)

- Iterative deepening search solves the problem of picking a good value for the limit "l" in depth-limited search. It can be implemented as an iterative call to depth-limited search where before every iteration "l" is increased of 1 and the search restarts from the root. It combines many of benefits of depth and breadth first search. Like the first its memory requirements are modest: O(bd) or O(bm) when there is no solution. Like breadth-first search instead it is optimal for problems where all actions have the same cost, and it is complete for acyclic finite state space. Its complexity is O(b^d) when there is a solution and O(b^m) otherwise.

Heuristic search

- Without previous information is very difficult to solve interesting problems. So, we approach them with heuristic algorithms. A heuristic is an estimation of the cost from a node to the closest goal node. A heuristic is admissible if it always underestimates the real cost to reach the closest goal node. Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem. Some other strategies to find heuristics are learning by looking at the "meta-level" computation tree, pattern databases, learning heuristics by successfully solving several instances of the problem.

- Greedy best-first search is a heuristic search algorithm which expands first the nodes with lowest *h(n)*, thus the nodes that appears to be the closest to the solution and that can lead to it more quickly. The function that evaluates the desirability of a node is only given by the value of h for that node. It is neither complete (it can get stuck in cycles) nor optimal. Time used for finding a solution is O(b^m) and same for space occupied in memory, but both can be improved with a good heuristic.

- A* is the most used informed algorithm. It is based on the evaluation function f(n) = g(n) + h(n) where g is the cost for having reached the node n and h is the underestimated cost for reaching a goal from that node. The search proceeds visiting nodes with lowest f(n). The algorithm keeps track of the cost of every node that it has explored and if while exploring a path it becomes more expensive of a node previously visited it backs up to this node exploring its successors. A* is complete and optimal under an admissible heuristic. Time for finding the solution is exponential in [relative error in h * length of solution].

- IDA* gives us the benefits of A* but without the requirement to keep all reached states in memory, at a cost of visiting same states multiple times. In IDA* the cutoff is the f-cost (g+h); at each iteration the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration

- RBFS resembles a recursive depth-first search, but rather then 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 the 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 of the forgotten subtree and can therefor decide whether it’s worth expanding it again later.

- SMA* it proceeds just like A*, expanding the best nodes until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the worst leaf node, the one with the highest f value. Its value is kept in its parent node and that subtree will be regenerated when all other paths are worst. It is complete and optimal if the available memory allows to record respectively the shallowest solution and the optimal one.

Local search algorithms

- In many optimization problems path is irrelevant, the goal state is itself the solution. Local search algorithms operate by searching from a start state to neighboring states, without keeping track of paths or reached states. In this kind of problems an objective function assigns a value to each state in the state space. These algorithms are not systematic but use very little memory and can often find a reasonable solution even on infinite or large states space.
- Hill-climbing search keeps track of one current state and on each iteration moves to the neighboring state with highest value trying to reach the goal (state with the highest value in all the state space). It terminates when reaches a peak. It does not look ahead beyond the immediate neighbors of the current state. It can get stuck in local maxima, ridges (sequence of local maxima) and plateaus. For these reasons it is not complete neither optimal. Some ways to improve its performances are:
    - Side moves with a limit on the maximum number
    - Stochastic hill climbing which chooses random moves among the uphill nearby states.
    - First choice hill climbing which generates random successors of the current state choosing the first with uphill value.
    - Random restart which conducts a series of hill climbing searches from randomly generated initial states, until a goal is found.
- Simulated annealing has an overall structure like hill climbing, but instead of picking the best move, it picks a random move. If the move improves the situation is always accepted, otherwise, the

algorithm accepts the move with some probability that decreases exponentially with the badness of the move.

- Local beam search keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any of these is the goal the algorithm stops. Otherwise it selects the k best successors from the complete list and repeats. In this algorithm differently from random restart useful information is passed among the parallel search threads. It can suffer from a lack of diversity among the k states, so a variant (called stochastic beam search) chooses not the best successors but the successors with probability proportional to the successors value
- Genetic algorithms are a variant of stochastic beam search, explicitly motivated by metaphor of natural selection in biology. States are a population of individuals in which the fittest (the one with the highest value) produce off-spring (successor states) that populate the next generation, a process called recombination. This kind of "reproduction" can follow two different process:
    - Crossover, also called recombination, in which information of two parents are combined to produces the offspring
    - Mutation that instead involves only one parent state, which produces its offspring through the mutation of some of its information

Constraint satisfaction problems

- Constraint satisfaction problems are based on a factored representation of the world in which each state is described by a set of variables each of which has a value. A CSP is solved when all the variables have a value that satisfies the constraints on it. These problems take advantage of the structured representation and use general rather than domain-specific heuristics.
  CSP structure consists of three components:
    - X which is the set of the variables
    - D which is the set of the domains, one for each variable, specifying their allowed values
    - C which is the set of constraints that specify allowable combinations of values

  Each constraint C_j consists of a pair *<scope,rel>*, where scope is a tuple of variables that participates in the constraint and *rel* is the relation that defines all combination of values that those variables can assume.

  In a standard search formulation states are defined by the values assigned to the variables so far. The initial state is the empty assignment {}. The successor function assign values to unassigned variables that does not conflict with the current state. It fails when there are no legal assignments. Goal test check is the current assignment is complete and consistent.

  An example is the problem of coloring the map of Australia such as two nearby regions never have the same color.

- Backtracking search is an algorithm for solving csp. It repeatedly chooses an unassigned variable, and then tries all values in its domain 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 previous state and the algorithm tries a new value for that variable. If there are no more legal values the algorithm ends, failing.

- 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 that is involved in the largest number of constraints on other unassigned 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

- Forward checking is a technique usable for improving the search. The idea is to keep track of remaining legal values for each variable after every assignment. But this method does not provide early detection for all failures. It terminates the search when a variable has no more legal values.

- Suppose we are representing our search space as a graph in which every node is a variable and every arc (edge) is a constraint between the two nodes it links. A variable X is arc-consistent with another one Y if for every value in X domain there is at least one legal in Y domain. In other words if for every value in X domain there is at least one in Y domain which satisfies the binary constraint (X,Y). We can exploit this principle for running before the search, or after each assignment, an algorithm that checks all the arcs in the graph to make every node arc-consistent with its linked nodes deleting values from its domain if they violate the arc-consistency. This can sometimes solve the search before the real search algorithm.

Planning

- Both planning and searching are techniques useful to look for a solution of a problem but they have some main differences.
    - In searching the world is represented through the use of ad hoc data structures while in planning we only use "fluents" that are simple Boolean variables
    - In searching the actions are coded instead in planning they are defined, only specifying their preconditions and effects.
    - Even the goal in searching is coded, while in planning it is a conjunction of fluents
    - In searching a plan is a sequence of action, and even if also in classical planning it is in this way, in some relaxed versions of planning it is a bit different.
- Search forward algorithm applies all actions whose preconditions are satisfied until a goal is found or all the states have been explored. The irrelevant actions cause the search space to blow-up, so even if it is problem independent (and this is good), it can be inefficient even with small problems.
- In backward search (also called regression search) we start from the goal applying actions backward until we found a sequence of them that leads to the initial state. At each step we consider only relevant or consistent actions. The first type includes all the actions whose execution add fluents required for reaching the goal. The consistent actions instead are all those actions that do not negate any fluent useful for reaching the goal. Previous states are computed following the rule
    - Result(s, a)$^{-1}$ = (s – (ADD(a)) U Pre(a)

POP

- Partial order planning is a variation of the classical planning based on:
    - The principle of least commitment according to which we can have a partial ordering between actions (instead of a total one) and not fully instantiated plans.
    - A change of problem representation. In POP a state of the search is no more a state representing the world but it is a partial plan. In the first case (state space) 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 second case (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.
- Plan computed in this representation are partially ordered collections of actions with:
    - Start action, which has the description of the initial state as effect
    - Finish action, which has the description of the terminal state as preconditions
    - Temporal ordering between pairs of actions

  In order to characterize the planning process two additional elements are needed:

    - The notion of open precondition, a precondition of an action which has not already casually linked
    - The notion of causal link, which relate two actions specifying that the effects of one of them satisfy the precondition of the other

  The 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.

  Actually, conflicts are called clobberers. A clobberer is a potentially intervening action which would destroy a condition already achieved by a causal link. More specifically a conflict between the causal link A -> p -> B and an action C holds when C has effect not p. There are two solution for these conflicts:

    - Promotion, which consists in placing the clobberer after the causal link (B < C, B comes before then C)
    - Demotion, which consists in placing the clobberer before the causal link (C<A, C comes before then A)

  The searching process could be improved through the use of some general heuristics:

    - Number of open preconditions
    - Most constrained variable
        - Open precondition satisfied with the smallest number of actions

  A planning graph also could be useful

Hierarchical task network planning

In real problems searching for a plan at the level of primitive actions could be very complex and inefficient. Sometimes primitive actions make very small changes with respect to the goal blowing up the search space. For this reason, we introduce the concept of high-level actions, actions that cannot be executed directly from the agent and that can be decomposed in a sequence of smaller subtask until primitive tasks (actions) are reached. These sequences are called refinements. A refinement completely composed by primitive actions is called implementation.

A high-level plan achieves the goal if at least on of its implementations does.

The hierarchical search refines HLA from the initial abstract specification into sequence of primitive actions to determine whether the current plan is feasible. To completely exploit the hierarchical decomposition we have to find a way of planning at the high level, without reasoning at the primitive level.

In order to do this the approach is to model HLA as primitive actions, defining their preconditions and effects. Because of the multiplicity of refinements that each HLA could have, its effects will include only the positive effects achieved by every refinement and the negative effects achieved by any refinement.

Demonic non-determinism allows us only to know if HLAs' refinements makes the goal reachable, but this implies that every refinement allows this, that is a strong condition.

Angelic non-determinism instead looks for at least one refinement that makes the goal reachable.

- Every HLA has a reachable set REACH(s,h) which specifies the states reachable from state s with the HLA h. The reachable set of a sequence of HLA is the union of all the reachable sets obtained by executing the action n from all the reachable states of the action n-1. A sequence of HLA reaches a goal if its reachable set intersect the set of possible goal states. So the search process proceeds as follows:
    - Search among high level plans
    - If one of the HLP's reachable sets intersects the goal return yes
    - Else refine the plan further
        - If refinement turns yes, return yes
        - Else return no

  Multiplicity of refinements of an HLA makes some fluent not sure to be always true. Their truth depends on the refinement we choose. For this reason, we can define an optimistic reachable set, which considers always true even the fluents that are not sure to be true, and a pessimistic reachable set, which instead includes only the surely true fluents. If the pessimistic set intersect the goal the plan is correct, if the optimistic one does, the plan may be correct. If neither the optimistic nor the pessimistic intersect, plan is wrong.