Search Summary

Author: **Nicolò Brandizzi**

Contributors:

DIAG
Sapienza
October 2018

Contents

Abstract

This is **free** material! You should not spend money on it.
This notes are about the *Search* part taught by professor Daniele Nardi in the Artificial Intelligence class. Everyone is welcome to contribute to this notes in any relevant form, just ask for a pull request and be patient. Remember to add your name under the contributors list in the title page when submitting some changes (if you feel like it).

Contents

1 Intro

Some stuff you should remember:

- **Completeness**: The algorithm is guaranteed to find a solution when there is one.
- **Optimality**: The strategy found is optimal.
- **Time Complexity**: The time spent to find a solution
- **Space Complexity** : The memory needed to perform the search.
- **Branching factor** [b]: the maximum number of successors of any node.
- **Depth** [d]: the depth of the shallowest goal node.
- **[m]** : The maximum length of any path in the state space.

2 Uninformed Search

Only use information available in the problem definition.

Breadth-first It expands the root node first and then all it's successors, and then it's successor and so on. It can be implemented using a *FIFO* queue where the goal condition is applied to each node when it is **generated**.
Proprieties:

- Complete if $d < \infty$
- Optimal if the path cost is a non-decreasing function of the depth (all action have same cost)
- Time and Memory complexities are $O(b^{d+1})$

Uniform-cost Expands the node with the lowest path cost $[g(n)]$. It can be implemented using a **Priority Queue** ordered by $g(n)$. The goal condition is applied to the node when this one is selected for *expansion* because the generated one can be a sub-optimal path.
Proprieties:

- Complete if $g(n) > \epsilon$
- Always Optimal since it expands the nodes in order of their optimal path cost
- Time and Memory complexities are $O(b^{C*/\epsilon})$, where $C*$ is the cost of the optimal solution

Depth-first Expands the deepest node in the frontier, using a LIFO queue.

- Complete for *graph version* if space is finite, not complete for *tree version*
- Not Optimal
- Time Complexity is $O(b^m)$ where m is the maximum depth of any node
- Space Complexity is $O(b \cdot m)$

Depth-limited Is the same as *depth-first* but there is a limit for the maximum depth l.

- Complete for $l \geq d$
- Optimal for $l \leq d$
- Time Complexity is $O(b^l)$
- Space Complexity is $O(b \cdot l)$

Iterative deepening depth-first Same as *Depth-limited* but gradually increases l until we have $l = d$.

- Complete when path cost is non decreasing.
- Space Complexity is $O(b \cdot d)$
- Time Complexity is $O(b^d)$
- Optimal if the path cost is a non-decreasing function of the depth (all action have same cost)

Search Direction Can be

- **Forward**: or data driven
- **Backward**: or goal driven
- **Bidirectional**: Uses two simultaneous searches, one from start to goal and one from goal to start, hoping the searches will meet in the middle. It is less intensive since $b^{d/2} + b^{d/2} << b^d$.

3 Informed Search

Some algorithm make use of an evaluation function $[f(n)]$ to decide which node to expand first (lowest). This information can be joint with an heuristic function $[h(n)]$ which **estimate the cost of the cheapest path from the current state (node n) to the goal**.

3.1 Heuristic functions

Dominance A heuristic function $h_1(n)$ **dominate** another $h_2(n)$ if

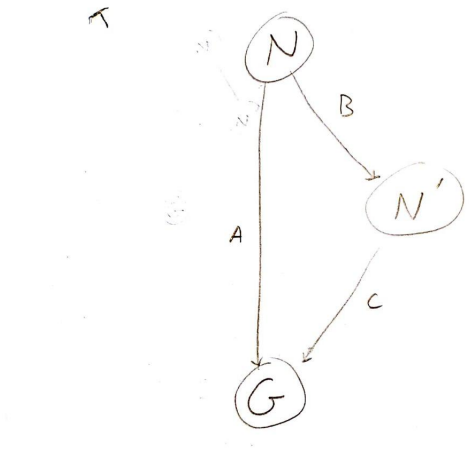
$$\forall n : h_1(n) \geq h_2(n)$$

This leads to the following propriety: **the number of nodes expanded by A* with a dominant h is always less**.

Admissibility For tree search the heuristic function needs to be **admissible**, that is the function never overestimate the cost to reach the goal. For example the path to get from Rome to Madrid being a straight line (euclidean distance) rather than following the roads and avoiding the sea will be an underestimation of the real path.

Consistency For graph search the heuristic function must be **consistence** [1], i.e. the estimate cost to reach the goal from n is not grater than the cost to reach the node n' (child of n) plus the estimated cost from n' to the goal[2]

$$h(n) \leq c(n, n') + h(n')$$



A: h(N)
C: h(N')
B: c(N, @, N')
↳ ACTION

[1] Every consistent heuristic is also admissible
[2] Also known as triangle inequality

Relaxed problem The optimal solution of a relaxed problem is not grater than the optimal solution of the real one. So you can derive a heuristic function from the relaxed problem and apply it to the real one.

Combination Given multiple heuristics $h_1(n), h_2(n), ..., h_m(n)$ you can combine them into a heuristic function which dominates all of them:

$$\forall n : h(n) = max(h_1(n), h_2(n), ..., h_m(n))$$

Effective Brancing Factor [EBF] Given an algorithm whose looking for a goal node and two heuristics associated with it, $h_1(n), h_2(n)$. This algorithm expands $N_1 = 17$ nodes before reaching the solution with the first heuristic and $N_2 = 14$ with the second. Moreover the expansion tree has a brancing factor [3] of $d = 3$. To measure the effectiveness of a heuristic we estimate the **effective brancing factor** [EBF]:

$$N + 1 = 1 + EBF + EBF^2 + ... + EBF^d$$

For our example we have:

$$17 + 1 = 1 + EBF_1 + EBF_1^2 + EBF_1^3 \rightarrow EBF_1 \approx 2.165$$

$$14 + 1 = 1 + EBF_1 + EBF_1^2 + EBF_1^3 \rightarrow EBF_1 = 2$$

The closer EBF is to one the better the heuristic is.

Greedy best-first search This algorithm uses just the information from the heuristic function. The proprieties are the following:

- **Not complete**: can get stuck into loops, can be complete only in finite state with checking.
- **Not Optimal**
- **Time** $O(b^m)$ where m is the maximum depth of the node and b is the branching factor.
- **Space** $O(b^m)$, keeps every node in memory.

3.2 A*

If we combine the information given by the heuristic function, i.e. the cost to reach the goal from the current node, **h(n)** and the cost to reach the node **g(n)**, we get an **estimated cost of the cheapest solution through the node n**

$$f(n) = g(n) + h(n)$$

Optimality of A* Proving that A* is optimal if h(n) is consistent is fairly easy, we just need to use the formulation of consistency:

$$f(n') = g(n') + h(n') = g(n') + c(n, n') + h(n') \geq g(n) + h(n) = f(n)$$

[4] Moreover the sequence of nodes expanded by A* is in non-decreasing order, hence the first node must be an optimal solution because f is the true cost for the starting point $(h(n_s) = 0)$

[3] The number of successors generated by a given node.
[4] Guarantees that f is not decreasing (monotonic).

Proprieties of A*

- **Is complete** unless there are infinitely many nodes with $f \leq f(G)$
- **Is Optimal**
- **Time is exponential**
- **Space**, keeps every node in memory

3.3 Other implementation of A*

The A* algorithm has some limitations, the most important is the exponential use of the memory (we keep every node in memory).

Iterative Deepening A* [IDA*] The cutoff value is the value of $f(n) = g(n) + h(n)$ rather than the three depth. This behave well for unit cost path finding problems, and keeps a linear memory consumption. But has the same problem of the uniform-cost search

Recursive Best First Search [RBFS] It goes down a node recusivley until the *f-value* increases above the alternative nodes. When this happens it rewinds back to the original node storing the *f-value* of its children. Each rewind and change of mind is an IDA* iteration.
Is has **O(bd)** complexity for space, but its time complexity is not easy to characterize, moreover it uses *too few memory* (only the *f-values* of nodes are stored) thus it can expand a path multiple times.

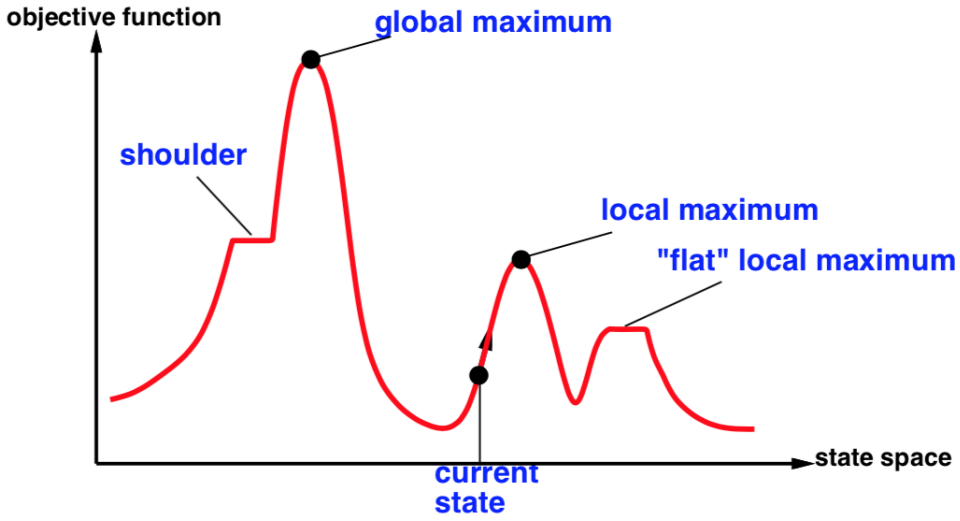
Simple Memory Bounded A* [SMA*] It behaves just like A*, but when its memory is full it drop worst leaf node [5] propagating its *f-value* up to its parent. When there are same *f-valued* nodes it always expand the newest and deletes the oldest. This algorithm is always complete if the depth (d) of the shallowest solution is in memory size reach.
Proprieties:

- Uses all available memory
- It avoids to repeat states until the memory is full
- It is complete and optimal if d can fit in memory

[5] The one with the highest *f-value*.

4 Local Search Algorithms

Sometimes we don't know the entirety of the state space or we just don't care of the path to reach the goal. For this kind of problem we adopt the local search algorithm whose objective is to gradually optimize a state.



4.1 Hill-climbing search

Hill climbing is a simple loop which moves in the direction of the increasing value [6]. Since this algorithm is a **greedy local search** it grabs a good neighborhood without thinking ahead, for this reason it can get stuck in the following cases:

- **Local Maxima**: it won't come down from the maximum (even if it's only local)
- **Ridges** : are a sequence of local maxima which are difficult to navigate [7].
- **Plateaux** : a flat surface which can be either a local minimum or a shoulder.

Stochastic hill-climbing Chooses a randomly from the possible uphill moves, it is slower but can find better solutions.

First choice hill-climbing Implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state

[6] The value is given by an heuristic function.
[7] Look it up on google.

Random restart hill-climbing Randomly generates initial states until a goal is reached. If each hill-climbing has a probability p of succeed then the expected number of restarts required will be $\frac{1}{p}$.

4.2 Simulate annealing

It is a midway between the hill-climbing algorithm and a randomize search [8]. The algorithm picks a random moves every time, if the moves decreases the total cost then it is accepted, on the other hand it has a probability p to be accepted. The probability p depends on two factor:

- The **schedule T** which is a sort of time measure from the start of the algorithm, the more time has passed the lower p will be
- The **badness** δE of the choice to be accepted, i.e. how much worst is the cost after accepting the choice.

If the schedule is slow enough then the algorithm approaches probability 1 to reach a goal.

4.3 Beam Search

This algorithm keeps track of k random generated states, compares them with each other and picks the k best from their successors. Useful information is passed between the k threads so that unfruitful states are abandoned to prefer best ones.
This can bring the search to be concentrated in a small region of the state space. To fix this there is another implementation called the **stochastic beam search** which chooses k successors at random with a probability which is a function of the increasing value.

4.4 Genetic Algorithm [GA]

GAs combine uphill tendency with a random exploration of the state space and information exchange between parallel states. The algorithm works like this:

1. At first k random states are generated, called the **population**.
2. The GA rate states using a **fitness function** which return a higher value for better states.
3. A pair of states are chosen according to the rating and a **crossover**[9] point is defined randomly from the states.
4. Finally a random mutation occurs in the children.

[8] Moving to a successor chosen uniformly at random from the set of successors
[9] The crossover usually take large steps at the beginning of the algorithm

5 Constraint Satisfaction Problems [CSP]

A CSP has three elements:

- A set of **variables** $X_1, X_2, ..., X_n$, which becomes states when X_i takes its values from D_i.
- A set of **domains** for each variable $D_1, D_2, ..., D_n$
- A set of **constrains** C for legal variable combinations. Which is later called *goal test*.

It is helpful to visualize a CSP as a constraint graph in which every node is a variable and the edges are the constrains. This is done in order to apply *constraint propagation* and delete large portions of search space from the searching.

5.1 Variations of CSP

The simplest kind of CSP have a discrete finite domain. When the domain is not finite (although is discrete) we must use a constraint language to express the link between variables without enumerating all the legal combinations.
As long as the domain is discrete this types of CSP fall under the category of **linear programming** which can be solved in time polynomial to the number of variables.

Aryties constrains There are different kind of constrains:

- **Unary constrains** where a variable is bounded to be a single value ($X = 1$), they can be treated as a domain restriction.
- **Binary constrains** that relates two variables together ($X \neq Y$)
- **Higher order constrains** like $Y \leq X \leq Z$, can be reduced increasing the number of variables and constraints.
- **Global constrains** which involves an arbitrary [10] number of variables (like $AllDiff(X_1, X_2, .., X_n)$)

Constraint Optimization Problems [COP] Indicates which solution is preferred, this kind of constrains are not forcefully respected but rather are costly to be neglected.

5.2 Inference in CSPs

We can use **constraint propagation** to reduce the number of legal values for a variable.

Node consistency A single variable is node-consistent if all the values in the variable's domain satisfy the variable's unary constraints. It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

[10] Not necessarily all the variables.

Arc consistency A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints. For example given $Y = X^2$ with :

$$\langle (X, Y), [(0, 0), (1, 1), (2, 4), (3, 9)]$$

If we want to make Y arc-consistent in respect to X we remove from the Y's domain those value which cannot be taken from the equation, so the Y domain will become $[0, 1, 4, 9]$.
The **Ac-3** algorithm is used to propagate arc-consistency in a CSP, it works like this:

1. Initially the queue contains all the arcs in the CSP.
2. AC-3 pops arbitrary arc (X_i, X_j) from the queue.
3. It makes X_i consistent with X_j
4. If this leaves D_i unchanged it continues, else D_i got smaller and the AC-3 gets all the arcs neighborhood of X_i and adds them to the queue
5. If D_i ends up empty the AC-3 return a failure
6. Else it returns the subset of the original domain.

The algorithm lead to faster search at the cost of executing the AC-3 which is $O(cd^3)$ in the worst case scenario.

Path consistency Almost the same as the arc-consistency, in this case we look at a triple of variables X_1, X_2, X_3. A two-variable set X_1, X_3 is path-consistent with respect to a third variable X_2 if, for every assignment $X_1 = a, X_3 = b$ consistent with the constraints on X_1, X_3, there is an assignment to X_2 that satisfies the constraints on X_1, X_2 and X_2, X_3. This is called path consistency because one can think of it as looking at a path from X_1 to X_3 with X_2 in the middle.

K-consistency A CSP is strongly k-consistent if it is k-consistent and is also *(k 1)-consistent, (k 2)-consistent, . . . all the way down to 1-consistent.* Now suppose we have a CSP with n nodes and make it strongly n-consistent, for each X_i we need to find the values in its domain which are consistent with $X_1, X_2, ..., X_{i-1}$. We will find a solution in $O(n^2 d)$ time using exponential space!

Global constrains **Resource constraints** (also called *atmost* constraint), for example the constraint that no more than 10 personnel are assigned in total is written as $AtMost(10, P_1, P_2, P_3, P_4)$ and can be checked estimating the sum. But for large resource-limited problems with integer value we use **bounds propagation**. For example:
F_1 and F_2 are to flights, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then:

$$D_1 = [0, 165] \qquad D_2 = [0, 385]$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$, propagating bounds constraints leads to :

$$D_1 = [35, 165] \qquad D_2 = [255, 365]$$

5.3 Backtracking for CSP

We know that CSPs are **commutative** since the assignment of a variable to a value reach the same partial assignment given any order [11]. The *backtracking search* is the same as a depth-search but when a leaf has no legal value left it backtrack the information to its root.

Values and variable order There are some ways to choose the order of variables/values to optimize the time spent building the three.

- **Minimum Remaining Value** [**MRV**] : choosing the variable with the fewest *legal* values.
- **Degree Heuristic** : it attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constrains.
- **Least constraining value**: it prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

Search and Inference There are some techniques to check whenever a choice is better than another. One of them is **forward checking** in which we chose a values for a variable $X_i = c$ and we remove c from all the domains of the variables connected to X_i. This is the case of the constrain *AllDiff*, but the technique works for other examples too.
Another one is **Maintaining Arc Consistency** [**MAC**]: when a value is decided for the variable X_i the algorithm calls AC-3 [5.2] to check arch consistency on all the variables connected with X_i.

Looking Backwards A problem that we may face is the choosing of the right variable X_i which is in direct conflict with the considered variable X_j. There might be cases of an illegal assignment between this two variables but an arbitrary number of other variables $(Y_1, Y_2, ..., Y_k)$ were chosen in the meanwhile. So the previous algorithm will just jump back from $X_j \to Y_K \to Y_{k-1} \to ... \to Y_1 \to X_i$ arriving at the actual problem after k futile approaches. **Backjumping** is used to store a **conflict set** of each variable so that, when an illegal assignment occurs, we can directly jump back to the cause of it.
Remember that every branch pruned by backjumping is also pruned by forward checking.

[11] This brings the possible combination in Sudoku from $n! * d^n$ to d^n