Shabbir Hussain

Amy Hoover

Foundation of Artificial Intelligence CS-5100

8 April 2016

AI Algorithmic Contrast: Introduction to Local Team Search

1.1 Abstract

This paper starts by comparing and contrasting various informed and uninformed search methods in AI with a goal of highlighting similarities and peculiar differences amongst them. We will also try to understand shortcomings of each algorithm and how their successors tried to alleviate them. Main focus of this paper will be on required 7 algorithms, but we'll try to explore other relevant algorithms as well. Then, we would explore a proposed variant of local beam search called a Local Team Search (LST). This algorithm addresses problem of stochastic search like incompleteness and local maxima while maintaining majority of benefits from non-deterministic approach. Finally, we would touch upon adversarial search method.

1.2 Introduction

What is the meaning of term Artificial Intelligence? Most intellectual communities have disparate views on the true meaning of the term, yet none of them can be deemed as ideal. My version of definition is "*A goal driven logical agent, searching intellectually to find a solution*". In order to achieve this goal an agent has to come up with a conclusion from its knowledge base (KB). This can be achieved by broadly two ways searching for a solution in KB or training a Neural Network or by a combination of those two.

This paper tries to simply explanations by abstracting implementation details and Math like time and complexities for existing algorithms. In our paper we'll consider an example of hungry Sampson who is trying to find a pizzeria (without a smartphone ☺)

2 Searching for a Solution

Searching is an effective strategy in finding a solution. The problem comes with the size of search space. If we have a very small search space, we might not have enough information to answer all questions on the other hand if we have a very large search space, then it may need days if not weeks to find a solution sequentially. There are many search methods available that tend towards optimizing this process. Most of these search methods can be broadly divided into 2 broad categories namely

- Uninformed search
- Informed Search

2.1 Uninformed search

This class of algorithms searches for solution blindly, without having any idea of which route to take. In our finding a pizzeria in a city example, troubled by the ghosts of his hunger Sampson starts haphazardly looking for an outlet by exploring in any available direction. He might choose any book keeping strategy to keep track of which roads he has travelled and which are still unexplored. There are two major uninformed search strategies as listed below:

- Breadth first search
- Depth first search

2.1.1.1 Breadth First / Uniform Cost

This algorithm works by exploring all nodes at a particular cost first. A cost could be a depth, distance or any other parameter that separates parent from child. This algorithm is the most optimal and guaranteed to yield the best possible solution in a tree, given enough time and memory.

2.1.1.2 Depth First Search

This search algorithm works by completely searching one till end before moving to next path. In our pizzeria example suppose you start with I90 from east coast you might end up at west coast before looking for a pizzeria that could be just next door. This kind of search is useful as it does not store all possible combinations of paths so is pretty light on memory. One most appropriate application could be just finding a path where optimality is not must.

2.1.1.3 Iterative Deepening

The above DFS might seem to go to un-necessary lengths just to find a solution next door. This problem can be addressed by limiting depth of DFS. So we say go to just end of one city block at one time then try another path. However, this approach could lead to failure if no pizzerias are available in one city block radius. Solution, increase search depth if no solution is found at first run, iteratively.

To understand better lets take an example of a simple tree shown in Figure 2.1-1 Simple Graph.

- Our start node is the root A
- Goal is the node labeled G

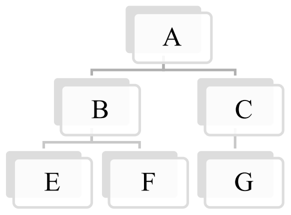


Figure 2.1-1 Simple Graph

Table 2.1-1 Uninformed search contrast describes various uninformed search methods

Table 2.1-1 Uninformed search contrast

Search Method	Traversal Order	Best Use	Completeness/Time
Breadth First	A-B-C-E-F-G	Get optimal solution	And other boring attributes can be found on Wikipedia
Depth First	A-B-E-F-C-G	Get solution in large space	
Iterative Deepening	C1: A-B-C C2: A-B-E-F-C-G	Get solution in somewhat better time than DFS (in most cases)	

2.2 Informed Search

The other half of search problems are the informed search algorithms. As search spaces grow larger and larger, finding a most optimal solution using through search becomes no longer feasible. So we enter into realm of guessing game with introduction of informed searches.

These algorithms have some kind of perception of goal and maybe a heuristic that can guess how far the goal is from each node. In our hungry pizza searcher example if our agent could smell the pizza and guess which direction would have the strongest scent. Then it would become some kind on informed decision.

These decisions, however informed they may be, could sometimes mislead. In our example aroma could be a result of gust of whirlwind which could result in our agent taking wrong direction. In a non deterministic world problems, this heuristic is almost never accurate. Another problem is the size of continuous data set which could be enormous to search sequentially even in one direction. Informed search could help in reducing search space only if we could learn to tackle misleads. In the following section we will explore some informed search algorithms that deal with problems step by step.

2.2.1 A* Search

One of the most basic search form is A* search. This method sequentially evaluates paths to select the best path from given perception and cost incurred so far. This algorithm is pretty good at finding an optimal solution as long as perception is dependable. It explores one best node and stores all of the unexplored nodes. This comes in handy if percept/heuristic turns out to be untrue to expectation. Even if it finds a solution it is not guaranteed to be optimal as it is only as dependable as it's optimality of percept. Same is applicable for all following algorithms.

2.2.2 Hill Climbing

A* is good but not that efficient in continuous search space. As going through data sequentially is painfully CPU intensive. Hill Climber tries to solve this problem by using a sampling method which takes random neighbors at arbitrary distance and if any node seems more promising than current then search continues through that route. Problem is good neighbor generation is luck dependent with random neighbors. Another temporary fix is to re-run random search a few times just to be a little surer that no better routes actually exists. Second loss is the loss of optimality. As search could get stuck at local maxima and couldn't find better nodes in near proximity.

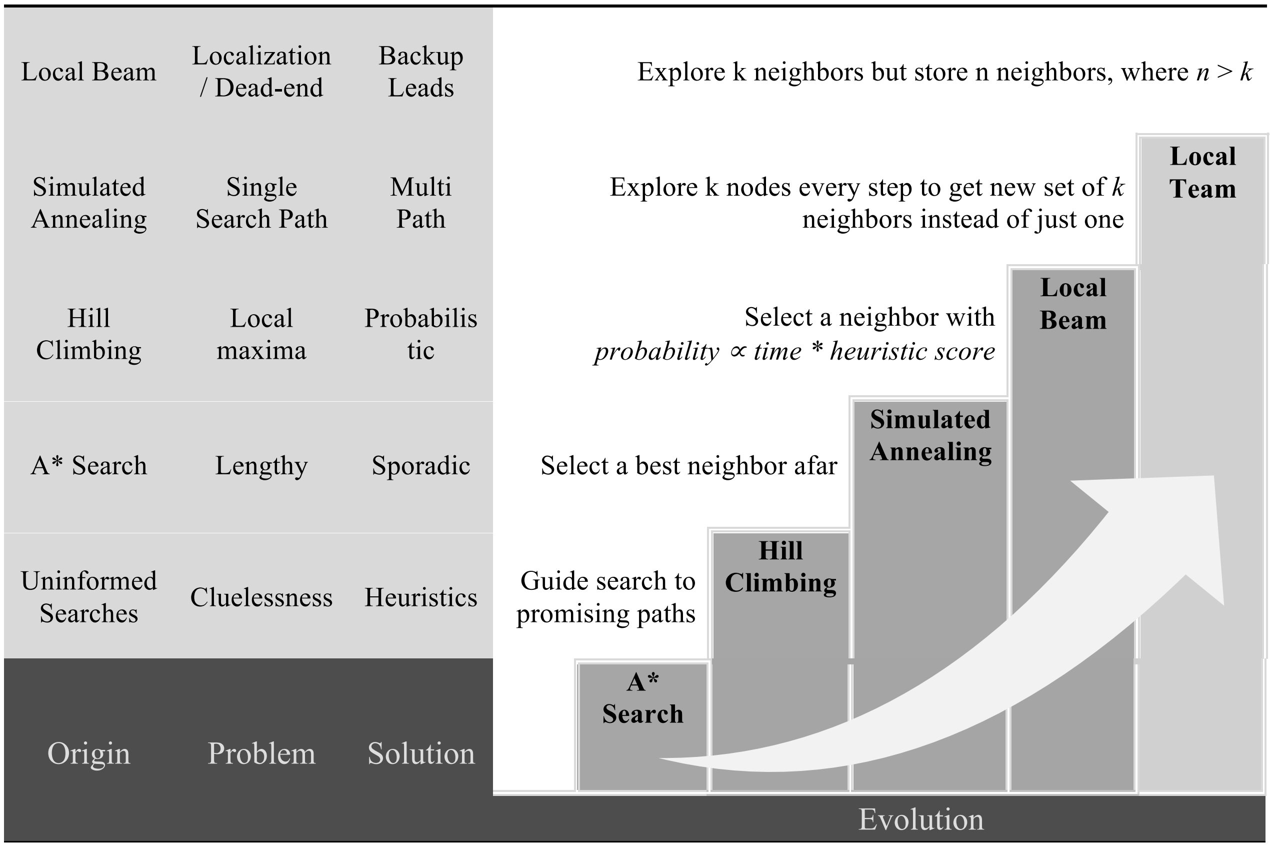
2.2.3 Simulated Annealing

Simulated Annealing is like a hill climber mostly. Instead of just selecting best child this algorithm uses probabilities to select new neighbor. These probabilities tend to favor more to fitter children and grows steeper with passage of time. So at first algorithm might deviate from target at first but converges with passage of time.

2.2.4 Local Beam Search

Local beam is multi threaded cousin of above search which tends to keep a pool of best nodes instead of selecting just one then exploring each node and merging results into the pool so all threads can benefit from findings of others. Problem here is search could get localized fast and suffer from lack of diversity as one thread might produce lot of promising children that don't hold up to expectations once all threads are concentrated towards these nodes at later stage in time.

Figure 2.2-1 Informed Search Contrast below shows summary of different informed search algorithms and what problems they solved.



Origin	Problem	Solution
Local Beam	Localization / Dead-end	Backup Leads
Simulated Annealing	Single Search Path	Multi Path
Hill Climbing	Local maxima	Probabilistic
A* Search	Lengthy	Sporadic
Uninformed Searches	Cluelessness	Heuristics

Explore k neighbors but store n neighbors, where $n > k$
Local Team
Explore k nodes every step to get new set of k neighbors instead of just one
Local Beam
Select a neighbor with *probability ∝ time * heuristic score*
Simulated Annealing
Select a best neighbor afar
Hill Climbing
Guide search to promising paths
A* Search
Evolution

Figure 2.2-1 Informed Search Contrast

3 Local Team Search: Custom Algorithm

In a local beam search *k* neighbors are explored simultaneously to generate set of *k* best. All other leads generated are considered useless and thrashed. Success rate of this algorithm is

- Directly proportional to number of simultaneous processes (*k*)
- Inversely proportional to number of dead ends (or accuracy of heuristics)

A thread working on misleading heuristics path would quickly generate seemingly good leads and add them to the common leads pool. This will invite other searches to abandon their paths and start converging on this lead. While this might be worth exploring it might not always turn out to be the best path. This quickly turns into a ripple effect with all threads abandoning their search paths that could have actually lead to success. Thus local beam search suffers from localization problem.

Theoretically it is possible to solve this problem by using an unbounded *k* would lead to highest success rates but it is impractical to increase computing power. Proposed variant Local Team Search builds on the assumptions that memory is cheaper than processing power in many cases and builds on local beam search. Search starts like a local beam where *k* threads start by exploring and merging results into a common pool. The difference is the amount of leads retained in the pool. Here instead we keep *n* leads in memory, where $n > k$. Like a local beam even if search gets concentrated to region with misleading heuristics temporarily, other good leads are not immediately discarded. Thus once threads realize path is not living up to the expectation they can quickly revert to leads generated before taking the detour

```
/** Performs team search over given start to goal. Where,
 * beamWidth is the width of beam (k)
 * backupLimit is the backup limit (n)
 */
function localTeamSearch(startState, goalState, beamWidth, backupLimit) {
        beam = {startState}                          // List of nodes
        while(beam.top() != goalState){
                // Probabilistic approach can be adopted to select which state to explore
                // Generate successors for every node within beam limit
                for(each state in beam list whose position is less than beamWidth){
                        beam <- beam U state.getAllSuccessors()
                }
                beam.sort()                          // Sort all state in the beam
                beam.tuncateTail(backupLimit) // Discard the tail beyond backup limit

        if(time limit has reached) break the loop
        }
        return beam.top()                            // Return the best solution until now
}
```

Code 2.2-1 Local Team Pseudocode

Figure aside shows a sample maze where our Sampson (🐹) wants to get to the nearest pizza(🍕). There Sampson has three paths to choose in total that out of which only one could head him to the pizza.

Search threads, *beamWidth* (*k*) = *2*
Lead pool size, *backupLimit* (*n*) = *10*

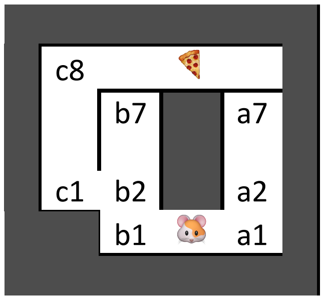


Figure 2.2-1 Sample maze local team

Consider a simple city example as shown in above figure, with the same goal to help Sampson to nearest pizzeria. A typical local beam with two simultaneous process will start with neighbors as shown in the table walkthrough below. Our heuristic is simple Euclidian distance to the pizzeria

The first table shows a walkthrough of Local Beam Search with 2 threads working in parallel. As you see after iteration 2 an important lead c2 gets dropped leading to failure at the end. While on the other hand in Local Team search c1 is retained as pool size is 10

Table 2.2-1 Local Beam Walkthrough

Iteration	Pool	Neighbors Generated		Pool
	Before	T 1	T 2	After
1	a1, b1	a2	b2	a2, b2
2	a2, b2	a7	b7, c1	a7, b7, c1
3	a7, b7	-	-	-
Result :	Failure			

Table 2.2-2 Local Team Walkthrough

Iteration	Pool	Neighbors Generated		Pool
	Before	T 1	T 2	After
1	a1, b1	a2	b2	a2, b2
2	a3, b3	a7	b7, c1	a7, b7, c1
3	a5, b5, c1	-	-	c1
4	c1	c8		
Result:	Success			

Thus more memory we add to leads pool more chances we get to success than a local beam search while keeping computing requirements roughly proportional. This approach adds a some overhead of sorting lists every time a longer list is used for leads pool. This effort can be justified by savings in the reduction of number of threads that are needed to explore number of paths. This algorithm uses a team of fittest members from a pool of leads thus the name Local Team Search.

4 Adversarial Search

Finally, we would like to explore last section of this paper that is adversarial search. This kind of search strategy is used in game playing where a decision has to be reached based on opponents moves. A more general definition could be making an optimal decision in a competing multi agent environment where actions of an agent can affect another agent's performance.

4.1 Alpha-Beta MiniMax

Any algorithm working in adversarial environment would have to anticipate opponents move in order to select its own move. Problem arises with number of moves opponents can make. This is called branching factor, which is responsible for width of a decision tree. A typical game of chess has an average branching factor of about 35. Which means for every move an agent takes opponent can make 35 moves. So any agent hoping to contract opponents moves must consider all possibilities. Thus for a look ahead of j moves decision tree size becomes $(35)^j$. Which quickly becomes unmanageable for most systems working on move-move based algorithm like MiniMax.

In MiniMax there are two types of players(agents) competing. The MAX type tries to maximize its profit, while the MIN tries to minimize MAX's profit. However, considering both players play optimally MAX will never play a move that is sub-optimal, so would MIN. Having this in mind we start Alpha-Beta pruning. We don't go to end of each and every sub-tree created if we already know optimal evaluation of previous branch is less than the current. Because the minimizing player MIN will never let MAX go to this new path so any other evaluations of this branch will lead to a result that is more than current optimal which is never going to be reached in an optimal game play and vice versa.

5 Beyond Search

We have seen searching can solve many problems, however, it requires painstaking efforts of going through input set every-time you need to answer another question. In out example world it would be searching for pizzeria every-time you feel hungry. A wiser approach would be storing this information as some kind of knowledge representation. There are many approaches used for this purpose like knowledge graphs. These graphs are much more compact than the knowledge they represent,. Yet, as knowledge grows this again becomes search problem.

5.1 Perceptron Learning

Instead of storing in graphs we can also store knowledge as weights in structures similar to biological neurons. Together they can conceptually build a graph as complex as one representing human brain.

Neural Networks suffer from mystical obscurity due to lack of visibility in internal working similar to ones in our brain. Thus making them more complex to build and train.

6 Summary

As we have seen there are many algorithms which work for informed and uninformed modes. While informed search has a great speed benefit over their blind counterparts, their success is highly dependent on accuracy of fore sighting function (heuristic) they use. It is just like having a sight for a human being. As our vision suffers from different illusions and environmental conditions so does any search playing the guessing game.

There are different versions of informed searches which tries to alleviate problem of sight with heuristics by juggling with depth and accuracy. Each with its own pros and cons. Ultimately for search strategies it all boils down to the grain of search. The deeper you search the better but slower results you get.

The proposed variant Local Team Search extends the idea of memory vs processing tradeoff. Separating these two to get a clearer distinction and fine tuned customized search approach.

Finally, never write a paper with hungry stomach. You get stuck with silly examples like Sampson.