Search: The Core of AI

December 13, 2013

1 Design Search

Imagine that you have been given the (rather daunting) task of creating an origami sculpture of your best friend for her upcoming birthday party. Knowing nothing of this technique, you begin an information search among your friends, at the public library, on the internet, etc.. Since no book will have the precise folding instructions for the birthday girl’s face (unless you have very well-known friends), your primary goal of this search process is a set of general origami principles and techniques. In your pursuit of knowledge, many of the intermediate points are both source and pointer: friends refer to other friends, books reference other books, and web pages link to other sites. Still, many of these sources could end up being the final target of your search: a friend may be an origami expert and agree to do the whole job himself, a book may have all the essential folds for producing faces, etc. And although this seems obvious, each source is an existing, complete entity, though it may only provide partial information toward your goal. In short, your search involves a transition among well-indexed sources (via the pointers), all of which are pre-existing, and any of which could turn out to be the solution in and of itself (i.e., completely independent of the other sources along your search path).

Assume that you have gathered and digested all the relevant materials and sit before a large sheet of paper. A second, quite different process now begins: design search. You fold the paper many times, notice a discrepancy between the intermediate result and a never-to-be-violated principle from an obscure Japanese guidebook, and need to undo the three most-recent folds. From that point, you set off along another path of bending and mutilation. Each intermediate point is far from complete and may hint as to what succeeding folds might bring, but certainly does not, on its own, point your fingers in the right direction (although an origami master may indeed get these tactile clues). Rather, you rely on heuristics, (a.k.a. rules of thumb) based on general principles (gleaned from your information search) to give some indication of whether a given configuration is a promising stepping stone on the path to a final solution. In general, the intermediate configurations do not exist a priori: you create them and then either build further upon them, undo parts of them, or eventually recognize one of them as the final design goal.

This type of design search occurs in a space of partial solutions: most stops along the way cannot provide a respectable solution as is. Other design-search protocols work with full solutions as their primitive units: each point in the virtual search space constitutes a complete (albeit often sub-optimal) solution to the problem. For example, you might (methodically or randomly) craft long sequences of paper-folding operations and just try them out, in their entirety, without thinking too much about the consequences until the sequence finishes. Then, you would evaluate the final product, decide whether it matches the goal (i.e. your friend’s face). If not, you would modify some (any) operations of the sequence, start with a new piece of paper, and carry out this new sequence to yield a new solution/attempt, evaluate that one, and then finish, or modify and repeat. Here, although each intermediate state does represent a full solution, it gives only implicit hints (guided by your heuristics) as to the next move, i.e., fruitful changes to the folding sequence.

Terms such as search and search engine make the technology-section headlines all the time. This is predominantly
information search, the undisputed power of the world-wide web and the process that has changed the way people work and think. Artificial Intelligence is all about design search, in virtual spaces of both complete and partial solutions. It too has changed the way people work and think (about thinking, in fact), and it also plays a huge rule in the internet.

This chapter is about design search and some of its many formalizations that have conquered challenging GOFAI problems over the past several decades. In addition, these same formalisms offer useful perspectives for understanding biology, Bio-AI, and emergent intelligence.

2 Classic AI Search Techniques

Now imagine a different problem: finding efficient connection patterns among a collection of sites. Figure 1 depicts the layout of 4 buildings that might, for example, compose the science complex of a small college. Assume that the college wishes to interconnect the 4 buildings with fiber optics in a manner guaranteeing that the computers in any building can send signals to those in another building using only the cable of this new network. Furthermore, since fiber-optic cables are not cheap, the college wants to minimize the total amount of cable.

As shown in the figure, some solutions utilize less cable than others; those using the least possible cable are known as minimal spanning trees of the network, and tasks of this sort are known as minimal spanning tree (MST) problems. Formally, an MST instance involves a graph \(G\) that consists of a set \(V\) of vertices (e.g., points, sites, locations, etc.) and a set \(E\) of edges (i.e. connections between the vertices), with each edge having a corresponding weight (typically representing a cost or distance associated with linking or traveling between the two vertices). The goal is to find a subset \(E^*\) of \(E\) that both minimizes the total weight and provides a fully-connected graph: there is a path along edges of \(E^*\) between any two vertices of \(V\). Finding solutions to MST requires only polynomial time, namely, a run-time proportional to \(|E| \times \log(|E|)\), where \(|E|\) is the size of \(E\). For example, \(|E| = 6\) in Figure 1 (top right).

The MST problem can easily be couched as a search problem in a space of partial solutions, each of which consists of \(V\) along with a subset of \(E\). As shown in Figure 2, MST search problems of this form begin with a single root node of a search graph (top of Figure) having no edges. Movement in search space involves applying state-transforming search operators, which in this case consist of the possible additions of single edges to the graph. When search states consist of partial solutions, the application of operators is known as node expansion, since the state associated with the parent search-tree node is typically supplemented in some way (in the child node), thus moving the state closer to...
a completed solution. Furthermore, parent nodes typically sprout many child nodes - one for each possible operator application - so the search tree appears to expand or branch out from each parent node.

The root node of Figure 2 can expand into 15 possible child nodes - since, in an unrestricted topology, 15 bi-directional edges exist for 6 vertices, but only 3 are shown. The search procedure may then decide to expand the 2nd of these children in 3 different ways (though 14 are possible), and the third of these children may be expanded using 2 (of the 13) edge-adding operators.

Searching intelligently involves both:

1. initially choosing a potent set of search operators,
2. deciding which of the open (i.e. unexpanded) nodes to next expand,
3. deciding which of the operators to apply to that node.

A well-designed search procedure for a given task will find a good (or optimal) solution without expanding too many nodes. In other words, a good search algorithm can start with a partial solution and complete it relatively quickly, without exploring too much of the (often vast) space of possible partial solutions.

The MST problem has two well-known solution/search procedures, both of which run in polynomial time: Prim’s algorithm and Kruskal’s algorithm. The former begins with any empty set of chosen edges, $E^*$, and a single component $C$ consisting of a randomly-chosen vertex $v_0$ in $V$. It then adds to $E^*$ the lowest-weight, unchosen edge that connects $v_0$ to any other vertex, $v_1$. Next, $v_1$ is added to $C$, and the process continues by adding the lowest-weight, unchosen edge connecting any member of $C$ to any non-member of $C$, which then joins $C$ when that edge is added to $E^*$. The loop continues until all vertices of $V$ are in $C$. $E^*$ is then guaranteed to be a minimal spanning tree.

Kruskal’s algorithm begins with a) the complete list of edges, $E$, sorted by ascending weight values: smallest-weighted edges first, and b) $|V|$ clusters, each containing one of the vertices of $V$. It then reads through the sorted edges until it finds one (e) that connects two vertices that are in different clusters. It then adds e to $E^*$ and merges the clusters that e connects. The algorithm repeats (walking further along the sorted-edge list, choosing edges, and merging clusters) until only one (large) cluster exists, indicating that all vertices are connected to one another. Again, $E^*$ is guaranteed to be an MST.

Compared to the abstract search process of Figure 2, both Prim’s and Kruskal’s procedures appear extremely intelligent, since neither expands any nodes that are not along a path to the final solution. Granted, many edges are considered for addition to the partial graph, but none but the smallest component-enlarging (Prim) or cluster-joining (Kruskal) edge is added at each step. Partial solutions involving sub-optimal networks are never really generated, and certainly never added to a list of nodes that need to compete to be next in line for expansion. In essence, the search tree is one long vine, and MST is hardly considered a search problem at all.

Although the shortest path between two points is a straight line - and thus the MST of a two-vertex graph is just the edge between those two vertices - the step up to 3 or more vertices complicates things. The minimum total length of connections between 3 vertices is not always a 2-element subset of the 3-edge set, e.g. the edges A-B and B-C for vertex set $\{A, B, C\}$. In some cases, a shorter MST can be found by adding strategically-located vertices to $V$. This relaxation of the constraints of the original MST problem (which restricted edge and vertex choices to those in $V$ and $E$) expands the space of possible designs to topologies such as those of Figure 3, two of which use less cable than the optimal solutions to the corresponding 4-vertex MST problem of Figure 1.

An MST task that permits the addition of new vertices is known as a Steiner Tree Problem (STP)[2, 5], which has a long history in the study of NP-Completeness [4]. Though new, improved solutions to special types of STP problems continually crop up in the operations-research community [11, 10], no general-purpose, silver-bullet solutions (akin to
Figure 2: Search-space exploration for a 6-vertex MST problem, where search states are the complete vertex set along with a subset of edges. The only search operators are the addition of single edges to the graph.

Figure 3: Alternate Steiner trees for connecting the original 4 sites (A-D).
Prim’s and Kruskal’s algorithms for MST) exist. Hence, STP remains a legitimate search problem to which concerns of node expansion and operator selection still apply.

2.1 Best-First Search

AI search techniques are geared toward these NP-complete problems; easier problems are typically solved by highly problem-specific techniques, whereas the AI methods offer useful generality in design-search domains where the available problem-specific expertise is either sparse, or only supportive of partial information or ad-hoc rules.

AI search methods try to exploit this partial or ad-hoc knowledge of the problem domain in order to make intelligent moves in search space. The core of this knowledge consists of a rough estimate of the distance from a given state to the goal. This is known as heuristic information, computed by a heuristic function that takes a search state as input and produces a distance-to-goal measure.

Search that exploits heuristics is often called best-first search. It is typically applied to search spaces in which solutions are gradually pieced together by expanding states via operator applications. It contrasts with depth-first and breadth-first search, which ignore nearness-to-goal estimates and simply expand states based on a pre-defined, knowledge-free, protocol. In depth-first search, a host of operators are applied to a single state, in series, until either a solution or failure state is reached. In case of the latter, search backtracks a level or 2 up the search tree before attempting another plunge to a solution or dead-end. Conversely, in breadth-first search, all immediate successors of a single state (i.e. those achievable by one operator application) are generated before more distant neighbors are considered.

In terms of search trees, breadth-first search involves a methodical downward expansion of the tree, with each horizontal level produced before the next. On the other hand, depth-first search begins with the expansion of a single vine from the root to a leaf, followed by repeated plunges to leaf levels from intermediate nodes.

Breadth-first search has the advantage of never overlooking a state, due to its methodical sweep of search space, but it suffers the drawback of taking a long time to get to solutions, since these typically reside at significant depths of the tree. Depth-first search can quickly find a solution, sometimes, but it can also waste a lot of time exploring false leads.

Best-first search combines the advantages of depth- and breadth-first variants by permitting either type of expansion pattern, depending upon heuristic information (along with exact information about the distance from the root to the current state). Thus, a best-first search might plunge halfway to the leaf before realizing that the current avenue of exploration (i.e. vine) is less promising than an unexpanded node higher up in the tree. At a later date, it might return to the abandoned path as other states reveal themselves to be less-than-promised. At any rate, a best-first search is not a simple series of plunges or level expansions, but a heuristic-based movement along the horizon of the tree, i.e., the set of unexpanded states, in search of promising nodes. Figure 4 illustrates the different expansion patterns for the three search types.

The classic best-first search procedure is the A* algorithm, designed by Hart et. al. in 1968 [6]. The heart of A* is the following equation for evaluating a state, s, in the search space:

$$f(s) = g(s) + h(s)$$  \hspace{1cm} (1)

Here, $g(s)$ denotes the cost or distance of getting from the root of the search tree to s, while $h(s)$ represents the heuristic: an estimate of the distance from s to a goal state. Thus, $f(s)$ represents the total expected cost of a solution path going from the root, through s, to a goal. In a nutshell, A* works by expanding the node with the lowest $f(s)$ value, i.e. the node whose state has the lowest combination of known ($g$) and expected remaining ($h$) cost.
A* applies well to problems in which:

1. An initial state, $s_0$ is given.
2. There exists a well-defined, finite set of operators, $\pi$, for mapping states into other states.
3. At least one goal state, $s_g$ can be completely described.
4. The problem requires a minimum-cost solution, which usually implies that as few operators as possible should be used. In cases where operators have varying costs, the emphasis shifts to exploiting the lower-cost operators when possible.

For the Steiner Tree Problem, one key piece of knowledge involves the addition of new vertices into the graph: they should only be placed at the Steiner point of the triangle formed by 3 of the pre-existing vertices. As shown in Figure 5, the Steiner point of a triangle ABC resides along its longest side, creates an angle of 120 degrees or more with the 2 vertices of that side, and gives the minimum total distance from itself to A, B and C.

By judiciously adding Steiner points to neighbor triples (i.e., points closer to one another than to most of the other vertices), a spanning tree can often be found whose total edge weight significantly undercuts that of the MST over only the original vertices. As detailed in [2], the maximum number of Steiner points is $|V| - 2$, where $V$ is the set of original vertices. However, there are $T = \binom{|V|}{3} = \frac{|V|(|V|-1)(|V|-2)}{6}$ possible triples of points to consider. Thus, the entire search space of possible choices of Steiner points contains $\Upsilon$ states, an exponential function of $|V|$:

$$\Upsilon = \sum_{i=0}^{|V|-2} \binom{|V|}{i}$$  \hspace{1cm} (2)

That is, for each possible number (i) of Steiner points, from 0 to $|V| - 2$, there are $\binom{|V|}{i}$ ways to pick them.

A few values of $\Upsilon$ for corresponding $|V|$ are given in Table 1.
Figure 5: (Top) Construction of Steiner point for the vertices A, B and C (as described in [2]) involves finding the longest side of triangle ABC (AC), then drawing an equilateral triangle AQC opposite to point B. The intersection of line BQ and the circle that circumscribes AQC is the Steiner point (P). If angle ABC ≥ 120 degrees, then B is the Steiner point. (Bottom) The Steiner point enables a shorter spanning tree of points A-D.

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>( \Upsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>176</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>903601306070</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>14667964292635233849320188625720221593</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>328917620589704695603909426561568338979699166834420325964162029401355216</td>
</tr>
</tbody>
</table>

Table 1: Sizes of vertex sets |V| and the corresponding number of states (\( \Upsilon \)) in an STP search space.
Hence, S is larger than the number of sand grains on earth for a mere 20-vertex problem. This astronomical size precludes the use of brute-force methods, such as breadth-first search, that exhaustively evaluate all possibilities. The constraint imposed by Steiner points reduced the search space from infinite to finite, but still exponential in $|V|$.

Still, these numbers are only rough estimates. For one thing, many STP algorithms include data structures that keep track of the proximity of vertices to one another. Only triples of nearby vertices are selected as the basis for Steiner points, although the ease of discerning relative nearness could vary considerably with the problem instance. As another confounding factor, some members of these triples may be Steiner points themselves, thus increasing the full number of choices from $\binom{|V|}{3}$ to $\binom{|V|+|S|}{3}$.

### 2.2 Steiner Trees using Best-First Search

Best-first search provides a feasible, general-purpose alternative for solving STPs. It cannot guarantee finding a minimal Steiner Tree, but the problem fits nicely into the A* protocol and thus serves as an illustrative example of the algorithm. Namely, a) a start state consisting of all vertices and no edges is an intuitive starting point, b) the problem requires a minimum-cost solution, c) a well-defined operator set exists, consisting of adding edges or Steiner points to an existing, partial solution, and d) a goal state is formally described as one in which $|E^*| = |V| + |S| - 1$ (i.e. the number of chosen edges is one less than the number of original (V) plus Steiner (S) vertices) and only one vertex cluster remains (using Kruskal’s algorithm).

In A-STP, a simple version of A* applied to STP, each node in the A* search tree denotes a state of the search space, and each such state has a representation housing a partial solution to the problem, or a snapshot of an intermediate stage of the overall computation, which is a mixture of Kruskal’s algorithm and Steiner-point generation. This state consists of a list of chosen edges (i.e. those participating in the spanning tree so far), a list of remaining edges (sorted in ascending order by weight), and list of Steiner points.

An arc between a parent and child node in the A* tree then signals that the child node’s state is achieved by applying a search operator to the parent node’s state. Parent nodes are expanded by either a) adding anywhere from 1 to J of the next edges recommended by Kruskal’s algorithm to a parent state, or b) adding anywhere from 1 to K new Steiner points (where J and K are user-chosen parameters). The arc cost for the former operators is simply the summed weight of all new edges, while Steiner-point additions incur no arc cost. However, the addition of Steiner points will increase the h value, since the number of required edges increases with the addition of each node. As is standard with A*, the g value for any node (n) is the sum of the arc costs from the root on down to n.

The heuristic (h) value of n is an estimate of the distance from n to a goal state. For STPs, the total number of edges in the final tree should be $|V| + |S| - 1$, so the number of additional edges is $R = (|V| + |S| - 1) - |E^*|$, where $E^*$ is the current set of chosen edges. The cost of these edges is unclear, but a reasonable underestimate is the summed cost of the next R edges on the sorted edge list. As it turns out, A* works best when h(n) never overestimates the actual distance-to-goal, as discussed more thoroughly in AI textbooks [8].

Figure 6 sketches the beginning of an A* run on a 6-point STP. The vertical bars and horizontal triangles give a rough estimate of arc costs and h(n) values, while the contents of each node display a wide range of partial states. The algorithm randomly selects triples for Steiner-point creation, but as indicated by the combinatorics above, all such triples cannot feasibly be tested for problems of non-trivial size. Hence, node expansion involves only a relatively small subset of all possible subsets of triples. In addition, the algorithm does keep track of neighborhood relationships between all vertices, thus insuring that Steiner triples only contain vertices located relatively close to one another.

Figure 7 displays several Steiner Trees found by quick runs of A-STP, where only 14-16 nodes are expanded for

---

1 As an implementational note, all states share the list of original points, so a separate copy is not needed in each state. However, separate versions of chosen edges, remaining edges and Steiner points are necessary, thus adding to the computational load of best-first search.
Figure 6: Abstract depiction of the A* algorithm applied to a 6-point Steiner-Tree problem. The node-expansion operators are to add the next 1, 2 or 3 edges using Kruskal’s algorithm or to add 1 or 2 new Steiner points (drawn as stars). Vertical bars on search-tree arcs denote relative arc costs, while horizontal triangles above each node represent relative heuristic (h) values. For clarity, not all operators are applied to each expanded node.

each run. The same method works well for STP’s involving a few hundred vertices, but larger problems, involving thousands of vertices, call for special-purpose algorithms. These typically use sophisticated geometric scaffolding to keep track of neighboring point triples. However, the point of this example is not to compare and contrast these specialized methods for STP, but rather, to illustrate the essence of search among a space of partial solutions using an example problem domain (STP), which happens to have interesting ties to neural development.

2.3 Search among Whole Solutions

Whereas the breadth-first, depth-first and best-first search techniques all work with partial solutions - the most promising of which become further elaborated as search continues - a whole host of alternate approaches, often called local methods, deal primarily with whole solutions that are gradually modified in pursuit of improvement. These approaches may work with a single focal solution or a collection of them; and their modification procedure can involve merely random tweaks of current solution components or purposeful changes based on domain-specific knowledge.

Most local search methods follow the same basic scheme:

- Begin with a set or population (P) containing M individuals, each of which constitutes a complete (though probably not optimal) solution.

  REPEAT

- Use an objective function (F) to evaluate each individual in P.
Figure 7: Resulting best-found Steiner Trees for 4 independent runs of A-STP. Original vertices of the ladder are circles, while Steiner points are stars, and L denotes the total length of the Steiner Tree. The minimum spanning tree (MST) of the 6 original points is 24 units in length. The bottom left tree is optimal.

- If an individual \( p^* \in P \) produces an optimal value for \( F \), return \( p^* \) and halt.
- Produce \( N \) children of solutions in \( P \).
- Update \( P \) by including some or all children and removing some or all previous solutions in \( P \).
  UNTIL a pre-defined maximum number of iterations have been performed.
- Return the best (though not necessarily optimal) solution found.

Algorithms then vary as to the value of \( M \), the manner in which children are produced, and the criteria for adding children to \( P \) (and removing existing members).

For example, in standard hill climbing, \( M=1 \), kids are produced by modifying only one feature of \( p_1 \) (the only member of \( P \)) at a time, and the child with the best evaluation replaces \( p_1 \). Note that many children can be produced on each step, since \( p_1 \) may have many features, each of which may take on several possible values. One of the drawbacks of hill climbing is that search halts if no children can improve upon \( p_1 \), which may not be a global optimum in the search space, only a local optimum.

In Simulated Annealing (SA), again \( M=1 \) and kids are produced by a host of random modifications to \( p_1 \), but now there is a probability \( p_T \) that the best child can replace \( p_1 \) even if its evaluation is inferior to \( F(p_1) \). This allows search to continue past local optima by jiggling over to points in search space that, though inferior to their parent, may serve as points along a path to the global optimum. In SA, a temperature (T) parameter controls the amount of jiggle such that \( p_T \) is directly proportional to T. SA runs typically begin with a high T value that gradually declines; Thus, SA explores more routes that may not appear too promising (i.e. takes more chances) early on but then settles into a hill-climbing mode as T and \( p_T \) decrease.

In Beam Search, \( M \) is larger than 1, and the best \( M \) children simply replace the current population. Alternatively, in Stochastic Beam Search (SBS), the \( M \) new members of \( P \) are chosen stochastically from the child set, where a kid’s evaluation is directly proportional to the probability that it will be chosen. But still, children with poor evaluations can get lucky (and highly-rated kids get unlucky) during the selection process.
Evolutionary Algorithms (EA) are popular relatives of stochastic beam search. They normally have $M > 1$ and produce kids via local modifications (a.k.a. mutations) to members of $P$, but many EA variants include a genetic operator akin to recombination (i.e. crossover) in biology. This entails the combination of 2 or more members of $P$ to produce a child, which may vary considerably from each parent and thus represent a point in search space quite distant from the parents. This is the bold new improvement upon stochastic beam search that EAs provide, but many practitioners use it only sparingly. An even bolder supplement to SBS comes from the EA subfield of Genetic Programming (GP), where solutions are no longer vectors of values but complete computer programs. GP adds whole new dimensions to automated design search.

### 2.3.1 Local Search for Steiner Trees

Returning to the Steiner Tree Problem, note that if one knows the correct set of Steiner points ($S^*$) with which to supplement a vertex set, then generating the Steiner tree is a simple two-step process:

1. $V^* \leftarrow V + S^*$
2. Apply Prim’s or Kruskal’s algorithm to find the MST of $V^*$

In short, there is no need to gradually add in the Steiner points to a developing spanning tree. Of course, knowing $S^*$ ahead of time must include knowing any Steiner points for triplets that also include Steiner points. This anticipation of Steiner points by other Steiner points must be baked into $S^*$. This sounds a bit far-fetched but is relatively straightforward to implement in a local-search method.

A simple representation for applying any (of the many) local-search methods to STP is a list of Steiner points, each encoded as a triple. For example, the vector representation of equation 3 encodes groups of 3 point indices (where points are labelled from 0 to $|V| - 1$).

$$A_S = \{[p_1^1, p_1^2, p_1^3], [p_2^1, p_2^2, p_2^3], \ldots, [p_k^1, p_k^2, p_k^3]\} \quad (3)$$

As mentioned earlier, the maximum number of Steiner points is proven to be $|V| - 2$, so for any STP, let $k = |V| - 2$ when designing the basic structure of $A_S$.

Since these vectors may be randomly generated and/or modified by the search algorithm, the possibility for creating nonsensical triples certainly exists. An STP-specialized decoding process can relieve this problem.

To decode the triple $[p_i^1, p_i^2, p_i^3]$ into a potential Steiner point, begin by converting each index:

$$p_i^j \leftarrow p_i^j \mod (|V| + |S_{i-1}|) \quad (4)$$

where $|S_{i-1}|$ is the size of the set of valid Steiner points generated by the decoding of the first $i-1$ triples of $A_S$. This insures that each index refers to an existing vertex, either an original or a newly-created Steiner point.

Once converted, a triple ($i^t$) can still be meaningless if it:

1. includes duplicate integers (and thus fails to specify 3 unique points), or
2. codes for 3 vertices whose Steiner point is one of those 3 vertices, or
3. duplicates another triple \( t^j \) \((j < i)\) in \( A_S \).

Hence, even though \( A_S \) has length \( k = |V| - 2 \), it may produce less than \( k \) meaningful Steiner points. In this way, \( A_S \) can encode anywhere from 0 to \( k \) meaningful triples. Thus, the presence of meaningless triples fortuitously provides needed flexibility to this representation, since, prior to running of the algorithm, we only know that somewhere between 0 and \( k \) Steiner points are needed.

A slight modification to \( A_S \) can improve search efficiency by allowing triples to be easily ignored or included based on the flip of a switch, i.e., the flag bit \( f_i \). Random modifications to these flags then permit speedy transitions in search focus between longer and shorter vectors.

\[
A_S = \left\{ [f_1^1, p_1^1, p_1^2, p_1^3], [f_2^2, p_2^1, p_2^2, p_2^3], \ldots, [f_k^k, p_k^1, p_k^2, p_k^3] \right\}
\]  

(5)

Two basic operators, for modifying \( A_S \), are sufficient to enable movement about the Steiner-tree search space:

1. Flip a flag bit from 1 to 0 or vice versa.
2. Replace any integer of a triple by a randomly-generated positive integer.

As discussed above, all local search methods rely upon an objective function \( F \) to evaluate individual solutions. These functions are a vital prerequisite to success in design search, and though they can be difficult to formulate for some problem domains, STP seems to require nothing more than a simple comparison of the MST of the original vertices \( V \) to the MST of \( V \cup S \). Hence, a useful objective function for STP is:

\[
F(i) = \frac{W_V - W_{V \cup S_i}}{W_V}
\]  

(6)

where \( W_V \) is the total MST edge weight over the original \( V \) vertices, and \( W_{V \cup S_i} \) is the MST of \( V \) supplemented with the Steiner points of solution \( i \) (\( S_i \)). \( F(i) \) is therefore the fractional reduction in the MST when solution \( i \)'s Steiner points are included. For most STPs, the optimal value of \( F(i) \) is 0.13 or less.

The entire process involved in evaluating an individual is summarized in Figure 8. First, a sequence of quads are converted into potentially viable triples of points, some of which correspond to meaningful Steiner points (\( S \)). A well-defined, deterministic process (Kruskal’s algorithm) is then used to generate a minimum spanning tree of \( V \cup S \). Then, an even simpler deterministic process is employed to sum up the edge weights of that Steiner tree and compare it to the summed edge-weight of the MST over \( V \), yielding an assessment of the tree, which, in turn, serves as an evaluation of the original vector of quads. This evaluation then helps the search process move in the direction of more promising solutions.

Given a representation for complete solutions, a set of operators for manipulating representations, a means of translating a representation into a meaningful spanning tree, and an objective function, the STP problem is now properly packaged for consumption by any of the local-search methods described above (and many more). Figure 9 shows a collection of Steiner trees found by an EA using this STP packaging. The same or similar solutions were also found using the incremental solution-building approach of the A* algorithm.

As shown in Figure 10, the search space is often depicted as a k-dimensional landscape in which, together, the first \( k-1 \) dimensions signify the representation of an individual solution, while the \( k \)th dimension denotes its evaluation. Search
Figure 8: The translation and evaluation of a parameter vector during local search. The vector (top) consists of 4 quads, (as formalized in equation 5). All quads with active (1) flags have their indices scaled relative to the vertex set’s size (which varies from 6 to 10 during the process). The integer above each quad indicates this size in effect when the given quad is decoded. Each scaled triple deemed meaningful is converted into a Steiner point. Kruskal’s algorithm is then used to find the MST of the original plus Steiner vertices (assuming all and only direct connections between any two points constitute the edge set). The total weights of the two MSTs (with and without the Steiner points) are then compared to yield an objective-function evaluation.
Figure 9: Various Steiner Trees found by an evolutionary algorithm (EA) using a population of 20 individuals and run for 200 generations. Black circles denote the original vertices, while blue pentagons represent Steiner points.
constitutes movement in that landscape via modifications to the individual representations. The search algorithm cannot see the entire landscape; if it could, the process would more closely resemble map-based navigation than actual search. It only has *spotty* information: the evaluations of some individuals.

The misleading nature of this 2-dimensional depiction of a k-dimensional space cannot be overemphasized. In a real search space, each solution can have thousands or millions of potential children/neighbors. Generating and evaluating all of them can carry extreme computational demands. The hallmark of an efficient search algorithm is the ability to find a global optimum while evaluating only a small subset of the entire search space.

![Figure 10: Rough sketch of the search space of Steiner-Tree solutions. Each parameter vector \((A_S)\) on the x axis translates (dotted line) into a Steiner tree, whose evaluation is plotted on the y axis. Local search mechanisms involve the exploration of different parameter vectors in a manner strongly biased by their evaluations.](image)

Figure 11 (left) shows a commonly-used *footprint* of the progression through search space of an EA looking for a Steiner tree similar to one of those in Figure 9. Here, the x axis denotes the generation of the EA (i.e., the round of population-wide evaluation), while the y axis represents the evaluation/fitness of the best individual of that generation. This plot is typical of an EA run, with stasis punctuated by progress: the algorithm finds an individual that cannot be improved upon for many generations. However, beneath each fitness plateau lies a population undergoing constant modification; but the best fitness in each generation cannot beat that found at the plateau’s inception. Often, exploration of these *neutral landscapes* eventually leads to improvement when the algorithm finds an positive slope at one of the many edges of the plateau.

Figure 11 (right) illustrates both the size and neutral regions of fitness landscapes. The genotypes in this example consist of 417 bits, since both flags and integers rely on a binary encoding in this EA. To get an impression of the local topography, 417 neighbors are generated (where each differs from the original in exactly one of the 417 bits) and evaluated, with each fitness plotted in the figure. This gives some indication of the complexity of search in these landscapes. In this case, climbing up one of the mini peaks requires generating one of the 3 advantageous bit mutations (of 417 possibilities). And these are quite good odds compared to many local-search problems!
Figure 11: (Left) Best-of-generation fitness progression for an EA run on an STP problem for 300 generations and using the objective function of Equation 6. (Right) The fitness values of all solutions in the immediate neighborhood of the best individual of generation 300, i.e., their genotypes differ by only one (of 417) bits. The x axis indicates the deviant bit with the resulting fitness plotted on the y axis. Note that the three mini peaks rising from plateaus represent improvements over the best-of-generation individual, whose fitness is 0.0236, the level of each broad plateau.

3 Steiner Brains

Neuronal arbors (for both axons and dendrites) have been shown to display Steiner trees [3] resulting from self-organizing fluid-dynamic effects during brain development. These trees minimize total cable volume, not length, which makes sense in networks with heterogeneous branch thickness.

Consider the simple graphs of Figure 12. If edge thickness is constant throughout the network, then the 4-node Steiner tree atop the figure minimizes both channel length and volume. However, if, as in most natural flow systems, the downstream distributing branches are thinner than upstream conduits, then the standard Steiner tree no longer minimizes volume (middle left). As shown on the middle right, moving the branch-point (P) closer to the supplying node A produces the volume-minimizing Steiner tree.

The bottom of Figure 12 depicts a flow system reminiscent of an axonal segment. In these networks, the goal is to supply the target nodes (small black circles) with material from the upper left node (A) using an optimal network structure. Cherniak et. al. [3] found that neuronal arbors of this sort come within a few percentage points of minimizing total conduit volume, while they often deviate by 20-50% from length- or surface-area-minimizing networks over the same set of target nodes. Thus, the search processes involved in brain development actually produce Steiner trees!

Though neural network topologies self-organize from an intricate growth-and-signalling process involving axons and their targets [9], the Steiner-tree structure seems to emerge from simple hydrodynamics. In fact, signal gradients typically interfere with Steiner formations. In a nutshell, hydrodynamic principles determine the optimal thickness of branch points, such that:

\[ P^k = C_1^k + C_2^k \]  

where P is the diameter of the parent conduit, \( C_1 \) and \( C_2 \) are those of the child conduits, and k is a constant, typically between 2.5 and 3.
These diameters affect the branch angle: thicker branches require a larger angle (between the two branches). Examples of these angles are $\theta_1$ and $\theta_2$ in the middle of Figure 12. Conversely, these principles also restrict larger conduits (which typically carry more material at higher speeds) from feeding into branches with a large $\theta$, since the abrupt change of direction creates turbulence, which reduces the system’s throughput and stresses the tubing itself. As an analogy, compare the sharpness of turn angles found on small backroads (i.e. roads with fewer cars, typically traveling at lower speeds) to those on super-highways.

Hence, for hydrodynamic efficiency, larger conduits must branch at smaller angles. Similarly, in systems involving competing, but balanced, forces, where all pull on a central point, the branch angles opposite the strongest forces are smaller. Imagine a 3-way tug-of-war between a strong participant A and two weaker opponents, B and C. For the weaker two to have any chance of neutralizing A, they must reduce the angle between themselves to one that maximizes the resultant (in the direction of A) of their combined forces. Natural systems as diverse as water basins, tree roots, cardiovascular networks and brains all display this topological feature [1], which emerges from the interplay of these and other forces during network development, a process easily characterized as a search for both neural targets and equilibrium.

Interestingly, these mechanical and hydrodynamic interactions bode well for volume-minimizing Steiner trees, as shown in Figure 12. In the simple 4-node networks, note that by moving point P closer to A, the branch angle (BPC) naturally decreases ($\theta_2 < \theta_1$) and total conduit volume decreases. Thus Steiner trees in natural neural networks emerge from nothing more than the interactions of basic physical forces among growing branches. Development does not search for Steiner trees, but it finds them!
4 Biological Search and Problem Solving

The bio-inspired approach to solving problems differs quite dramatically from traditional AI. The difference is most easily summarized by Figure 13. Problem solving is often characterized as a search through solution space in which solutions/hypotheses are generated and then tested for feasibility, optimality, etc. Traditionally, an efficient problem solver has been viewed as one that exploits as much intelligence as possible to generate reasonably good solutions and thereby avoids wasting time testing bad hypotheses. In fact, a popular measure of problem-solving improvement (i.e., learning) in knowledge-based systems was the degree to which test knowledge or constraints could be re-expressed (or operationalized) in the generator [7].

![Figure 13: The generate-and-test view of problem solving with respect to both bio-inspired and more traditional Artificial Intelligence (AI) and Operations Research (OR) approaches. The size of each octagon and pentagon indicates the amount of problem-relevant knowledge that it employs, while the size of the Hs cloud denotes the number of active hypotheses (full solutions) under consideration.](image)

The A* algorithm illustrates knowledge operationalism at a few different levels. First, most non-exhaustive searches among partial-solution states (e.g. best-first search) attempt to prioritize states for expansion, where A* favors those with the lowest combination of current and projected-future cost. A*’s heuristic function (h) embodies knowledge of the problem domain operationalized as expected future costs of partial solutions. It enables A* to give low priority to many nodes, effectively pruning them, and thereby preventing many (sub-optimal) complete solutions from being generated. Thus, a good h function incorporates a good deal of intelligence into the generator by carefully prioritizing nodes for expansion.

The node-expansion process is another story, involving the application of operators. In naive A* applications, the algorithm simply throws the whole gauntlet of transformation operators at each state that is chosen for expansion, immediately producing C new children, where C can be quite large. More sophisticated versions of A* may only generate one or a few children at a time, but keep the generator on hold, ready to produce more as needed. The naive approach employs little context-appropriate knowledge: it simply uses the set of operators originally designed for the problem, whereas the delayed-generation model may only produce children that have the most promise (based on knowledge of what operators work best on particular types of parent states), or it may just produce the children randomly, but just one at a time.

In A-STP, an important piece of compiled knowledge is that any Steiner point (s) will probably link up to 3 or more
vertices, at least one of which should be part of the triple of pre-existing points for which $s$ was generated. Since each new Steiner point adds greater complexity to the search process, by moving a partial state farther from a solution (in terms of the number of remaining spanning-tree edges to add), it is imperative to reign in this complexity by adding edges whenever possible. Immediately after a Steiner-point addition, some or most of the points to which it should connect already exist, so A-STP restricts the expansions following a Steiner-point addition to those involving one or more edge additions. Hence, the operator set is context sensitive, as an important piece of Steiner-tree knowledge gets operationalized as a constraint on the legal subset of expansion operators for different classes of partial solutions. This helps reduce the number of partial and (thereafter) complete solutions generated.

Now contrast this with the approach of the local-search methods. These work with complete solutions, some of which are randomly generated: just thrown out there for evaluation; others are random variations on parent states. The number of ineffective (if not completely worthless) solutions produced and evaluated can be quite high, particularly for bio-inspired approaches such as EAs. However, these solutions admit concrete evaluation, unlike partial solutions, which demand evaluation of their potential, a process fraught with inaccuracy. As a simple illustration, the Steiner tree of Figure 9 (bottom left) found by an EA requires the evaluation of 20 solutions in each of 200 generations (i.e., a total of 4000 evaluations), whereas A-STP rarely finds a good solution in cases where it produces and evaluates 4-5 thousand states. For problems such as STP, the evaluations provided by complete solutions may provide more useful information to the search engine than do assessments of partial states. Good heuristics ($h$) are often hard to produce, whereas objective functions like Equation 6 give an accurate quantification of the entire spectrum from failure to success.

In general, most GOFAI systems can take a hypothesis (an example of a full solution, as discussed above) and its test results (i.e., the basis for its evaluation) as inputs and produce a new hypothesis that is almost guaranteed to be an improvement. For example, if a logic-based system uses disjunctions of primitive terms to produce classifications of input examples, and if the test results indicate that a hypothesis is too specific (in that it classifies many positive examples as negatives), then the system will normally add an extra disjunction in order to generalize the hypothesis. This intelligence that the generator contains is simply a knowledge of basic logic. Alternatively, if the representational form is an artificial neural network, then the knowledge of the backpropagation algorithm (a standard learning procedure for neural networks) leads it to increase and decrease connection weights in ways that decrease the network’s output error. In this case, the intelligence lies in a very basic understanding of the effects of change in a system of interlinked equations. In either case, the intelligence is representation dependent: the backpropagation algorithm could not handle a logical disjunction written in standard propositional or predicate calculus, and a logical engine would be lost if given the weights and activation values from a neural network.

Bio-AI algorithms tend to have very little intelligence in the generator. In general, the greater the effort required to convert the basic syntactic representation (e.g., the integer vector above) to a meaningful semantic form (e.g., a Steiner tree), the less intelligent the algorithm will appear in terms of its ability to generate good hypotheses/solutions. This stems from the simple fact that solutions are created by operators working at the syntactic level, which, in these algorithms, have little awareness of the corresponding semantic translations. Bio-AI methods (particularly EAs) permit a large representational gap between syntax and semantics, thus allowing the generator to operate relatively independently of the problem-solving context. For example, the same EA can mutate and recombine bit-string representations that encode everything from logical expressions to neural networks to Bayesian probability tables at the semantic level. In this sense, EAs can be extremely task-independent. But, again, the manipulations made to the syntax have no guarantee of producing improved solutions.

From a different perspective, standard AI problem solvers include a good deal of information about how a good solution should be created, and they may even possess meta-knowledge about why they perform certain hypothesis manipulations. This bias helps avoid the generation of bad hypotheses; and the why knowledge may even enable the system to explain its choices to a human user.

In contrast, an EA has little how or why information but a good deal of knowledge about what properties a good solution should have. It just tosses solutions out there and filters out the bad attempts. This is exactly how nature works: many random variations are tried, and the environment determines who survives based solely on what they
A major point of this book is that many of the other neural processes underlying intelligence also work that way: by the relatively random production of attempts followed by selection of the successes, by processes reflecting little more knowledge than trial and error, and by mechanisms that more closely resemble local search than A*. And again, it works, quite well.

References


