Joint and *LPA**: COMBINATION OF APPROXIMATION AND SEARCH

Daniel Ratner and Ira Pohl

Computer & Information Sciences
University of California Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

This paper describes two new algorithms, Joint and LPA*, which can be used to solve difficult combinatorial problems heuristically. The algorithms find reasonably short solution paths and are very fast. The algorithms work in polynomial time in the length of the solution. The algorithms have been benchmarked on the 15–*puzzle*, whose generalization has recently been shown to be NP hard, and outperform other known methods within this context.

I. INTRODUCTION

In this paper we describe two new algorithms, Joint and *LPA**, which can be used to solve difficult combinatorial problems heuristically. The algorithms find reasonably short solution paths and are fast. The main idea behind these algorithms is to combine a fast approximation algorithm with a search method. This idea was first suggested by S. Lin (Lin, 1965; Lin, 1975), when he used it to find an effective algorithm for the Traveling-Salesman problem (TSP). His approximation techniques were strongly related to the TSP. Our goals are to develop a problem independent approximation method and combine it with search.

An advantage of approximation algorithms is that they execute in a polynomial time, where many other algorithms have no such upper bound. Examples where there is no polynomial upper bound can be found for various models of error in tree spaces (Pohl 1977); or under worst case conditions, where the error in the heuristic function is proportional to the distance between the nodes, the number of nodes expanded by A^* is exponential in the length of the shortest path (Gasching 1979).

In the following sections we state conditions that assure that the new algorithms will finish in polynomial time. Later we describe the algorithms and give some empirical results. Our test domain is the 15–*puzzle* and the approximation algorithm is the *Macro–Operator* (Korf, 1985a). The need for an approximation algorithm in the case of the 15–*puzzle* has been demonstrated in (Ratner, 1986) by a proof that finding a shortest path in the (n^2-1)–*puzzle* is NP-hard.

The empirical results, which come from test on a standard set of 50 problems (Politowski and Pohl, 1984), show that the algorithms outperform other published methods within stated time limits. Empirical results in two recent reports are related but reflect different goals. The Iterative-Deepening-A^* method found optimal solutions to randomly generated 15–*puzzles*, but it generated on average nearly 50 million nodes (Korf 1985b). In (Politowski 1986) excellent search results are achieved by an improved heuristic found through a learning algorithm.

II. GENERAL CONCEPTS

Let $G_n(V_n,E_n)$ be a family of undirected graphs, where n is the length of the description of G_n. Suppose there is an approximation algorithm that finds a path in a graph G_n between an arbitrary $x \in V_n$ (start node) and an arbitrary $y \in V_n$ (goal node) and runs in a polynomial in n time. Since the algorithm is polynomial, the length of the path is also polynomial. Once we have a path, we can make local searches around segments of the path in order to shorten it. If each local search is guaranteed to terminate in constant time (or in the worst case in polynomial time) and the number of searches are polynomial in n, then the complete algorithm will run in polynomial time.

In order to bound the effort of local search by a constant, each local search will have the start and goal nodes reside on the path, with the distance between them bounded by $d_{\max}$, a constant independent of n and the nodes. Then we will apply A^* with admissible heuristics to find a shortest path between the two nodes. The above two conditions generally guarantee that each A^* requires less than some constant time. More precisely, if the branching degrees of all the nodes in G_n are bounded by a constant c which is independent of n then A^* will generate at most $c \cdot (c-1)^{d_{\max}-1}$ nodes.

Theoretically $c \cdot (c-1)^{d_{\max}-1}$ is a constant, but it may be a very large number. Nevertheless, most heuristics prune most of the nodes (Pearl 1984). The fact that not many nodes are generated, is supported by our experiments reported in the result section.

The goal of the local search is to find a new path between two nodes which is shorter than the existing subpath on the original path. Hence, if there is a shorter path its new length will be at most $d_{\max}-1$. These two paths create a cycle of length $2 \cdot d_{\max}-1$ at most. Thus if the length of the smallest (non-trivial) cycle in G_n is CL, we want $d_{\max} \geq (CL+1)/2$. This means that CL has to be a constant, independent of n. Moreover, we expect that cycles of length CL (or a bit larger) exist throughout the graph. This is the case for many combinatorial and deductive problems.

Once we know that there is such a constant CL, we pick $d_{\max}$ that satisfies the condition $d_{\max} \geq (CL+1)/2$ in G. We would like to select segments of length $d_{\max}$ on the solution path given by the approximation algorithm and to try to shorten these segments by the local searches. There are many of ways to pick the segments. The algorithms that we suggest, LPA^* and Joint, pick a segment in a way that is based on our experiments and motivated by the following two facts:

a. Assume that node y is on the path between x and z nodes. Then if we cannot shorten the path between x and y and we cannot shorten the path between y and z, it does not mean that we cannot shorten the path between x and z (see Figure 1.).

b. Replacing a path by another path of the same length can later yield a shortening (see Figure 1.).

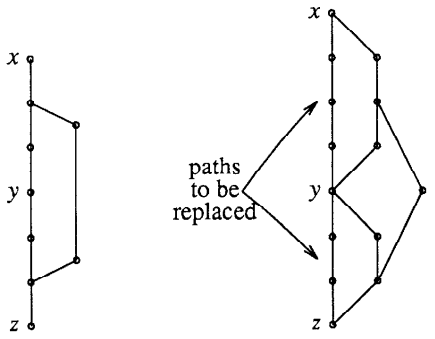

Figure 1. Examples of possibilities to shorten the Path(x,z).

Assume that the Global.Path consists of the following three consecutive segments l_1, l_2, l_3 and we try to shorten l_2. Note that there is no mutual influence among the searches and the segments. Thus when we replace the segment l_2 by a new one, l_n, (shorter or not), the beginning of l_n may be exactly as the end of l_1 but in the opposite direction (see Figure 2.) and the end of l_n may be exactly at the beginning of l_3 but in the opposite direction. Hence after replacing one segment by the other, we check whether there are such trivial cycles and cancel them. The process of erasing these cycles within the Global.Path is called Squeeze. The Squeeze process saves some local searches, and was shown, for both algorithms, to be useful in reducing execution time.

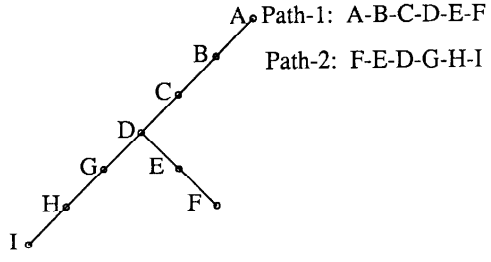

Figure 2. Two paths that partially cancel each other.

III. The LPA^* algorithm

In this section we define the algorithm LPA^* (Local Path A^*). First the algorithm finds a path by some approximation algorithm. Then it starts searching for an improvement from the global.start.node ($x \in V_n$). If the local search fails to shorten the current subpath, we advance the start node (anchor.node) along the path by a small increment, called δ_{anchor}. Then we try again to shorten the subpath starting at anchor.node and repeat this process until we succeed. The reason we advance the start node by only a small increment is motivated by fact (a) of the previous section. Once we succeed in shortening the subpath, we divide the remaining subpath (between the anchor.node and the global.goal.node ($y \in V_n$)) into consecutive segments of length $d_{\max}$. Then for each segment, we make a local search and replace it by the result of the search. The result of the local search is never longer than the original segment. The replacement is done, whether a shortening occurs or not, to increase the randomness in attempted improvements. This is a standard method in search to avoid repeating minima, which is given in fact (b) of the previous section. Upon finishing the replacement, the algorithm returns to the anchor.node as if it is the global.start.node and repeats the process.

In the following we present the LPA^* algorithm, using the following notations:

$d(x,y,P)$ is the distance between the nodes x and y along P;
G.P = Global.Path; L.P = Loacal.Path;
$g.s.n$ = global.start.node; $g.g.n$ = global.goal.node;
$l.s.n$ = local.start.node; $l.g.n$ = local.goal.node;
$a.n$ = anchor.node;

The LPA^* algorithm.

```
G.P ← Approximation(G , g.s.n , g.g.n) ;
a.n ← g.s.n ;
while d(a.n , g.g.n , G.P) ≥ d_max
begin
    L.P ← A*(G , a.n , l.g.n) ;
    l.g.n is the node s.t. d(a.n , l.g.n , G.P) = d_max;
    replace the segment from a.n to l.g.n in G.P by L.P
    if length of L.P = d_max
    then
        a.n ← the node with distance δ_anchor from a.n
                along G.P ;
    else begin
        l.s.n ← l.g.n ;
        while d( l.g.n , g.g.n , G.P) ≥ d_max
        begin
            l.g.n is the node s.t.
              d(l.s.n , l.g.n , G.P) = d_max;
            L.P ← A*(G , l.s.n , l.g.n) ;
            replace the segment from l.s.n to l.g.n in G.P
                by L.P ;
            l.s.n ← l.g.n ;
        end;
    end;
end;
L.P ← A*(G , a.n , g.g.n);
if length of L.P < d_max
  then replace the segment from a.n to l.g.n in G.P by L.P ;
```

In order to show that LPA^* is polynomial time in n, we have to show that the number of times we use A^* is polynomial in n. Since the length of the approximation's solution is polynomial in n, it is enough to show that the number of times we call A^* is polynomial in the length of solution. Let L_{app} be the length of the path generated by the approximation algorithm and let L_{opt} be the length of a shortest path between the global start and goal nodes. Then the number of times LPA^* calls A^* is not more than $\sum_{i=L_{opt}}^{L_{app}} \left\lfloor \frac{i}{d_{\max}} \right\rfloor + \left\lceil \frac{L_{opt}-d_{\max}}{\delta_{anchor}} \right\rceil$ (Ratner, 1986), which is quadratic in L_{app}. Practically, for the *15–puzzle*, the number of searches is much less than the worst case. According to our experiments the number of calls is about $\frac{L_{app}}{2\cdot\delta_{anchor}} + \frac{L_{app}}{d_{\max}}$, as reported in the result section.

IV. The Joint algorithm

In this section we present the Joint algorithm. The main ideas behind the Joint algorithm are:

Starting with a solution path found by the approximation algorithm, we divide it into segments of length $d_{\max}$. Then we shorten each segment by a local search and replace the segment with the path found by the search. As a result, the new Global.Path is composed from optimal subpaths. Since each segment is optimal, the most promising place to look for shortening is around the nodes that connect these segments. We name these nodes "joints". The algorithm always try to shorten the path around the first joint. The local start and goal nodes are picked as symmetrically as possible around the first joint and a new local search takes place. The path found by the local search replaces the corresponding segment on the Global.Path, even when no improvement is made. This is done to increase the randomness in attempted improvements. We found that it is worthwhile to define a parameter, called δ_{joint}, that depends on the problem. The algorithm will erase all the joints along the segment except those which are located in the last δ_{joint} nodes on the segment. If a shorter path was found the local start and goal nodes are added as new joints.

In the next column we present the Joint algorithm. We use the same notations we have used in the LPA^* algorithm.

The number of times Joint calls A^* is not more than $2(\,(L_{app}-L_{opt}) + \left\lfloor \frac{L_{app}}{d_{\max}} \right\rfloor\,)$ (Ratner, 1986), which is linear in L_{app} and therefore polynomial in n. Practically, for the *15–puzzle*, the number of searches is much less than the worst case. According to our experiments the number of calls is about $\frac{L_{app}}{d_{\max}} + \frac{2\delta_{joint}+d_{\max}}{4}$, as reported in the result section.

The Joint algorithm

Initialization: list of joints is empty ;
G.P $\leftarrow$ Approximation(G , $g.s.n$, $g.g.n$) ;
$l.s.n \leftarrow g.s.n$;
while $d(l.s.n$, $g.g.n$, $G.P) \geq d_{\max}$
begin
  $l.g.n$ is the node s.t. $d(l.s.n$, $l.g.n$, $G.P) = d_{\max}$;
  append $l.g.n$ to the list of joints;
  L.P $\leftarrow A^*$ (G , $l.s.n$, $l.g.n$) ;
  replace the segment from $l.s.n$ to $l.g.n$ in G.P by L.P
  $l.s.n \leftarrow l.g.n$;
end;
while there are more joints do
begin
  $l.s.n$ is the node s.t. $d(l.s.n$, $first.joint$, $G.P) = d_{\max}/2$;
  $l.g.n$ is the node s.t. $d(l.s.n$, $l.g.n$, $G.P) = d_{\max}$;
  remove all the joints that are on G.P and satisfy
    d(joint, $l.g.n$) $> \delta_{joint}$;
  L.P $\leftarrow A^*$(G , $l.s.n$, $l.g.n$) ;
  replace the segment from $l.s.n$ to $l.g.n$ in G.P by L.P ;
  if length of L.P $< d_{\max}$
    then prepend $l.s.n$ and $l.g.n$ to the list of joints ;
end;

V. *Macro–Operator* as an approximation algorithm

For our algorithm to be time efficient, we need to choose a fast approximation algorithm, that gives a "reasonable" solution path. The *Macro–Operator* Algorithm is such an algorithm. It runs in linear time in the length of the path it produces. The path generated by the *Macro–Operator* algorithm is a sequence of segments. In many cases each segment is optimal, which is the goal of the first loop in the Joint algorithm.

The idea behind the *Macro–Operator* is to predefine a set of subgoals such that any instance of finding a path in the graph can be viewed as a sequence of some of the predefined subgoals. For each of the subgoals there is a known macro (a subpath) that solves it. There is a restriction on each macro, namely, if a macro was used to solve a subgoal, then it must leave the previously solved subgoals intact.

Finding a path in a graph induced by permutations on *n–tuples* is an example of using a macro-operator. A graph induced by a permutations on *n–tuples* is a graph where each node represents a distinct permutation, and the edges are defined by some rules that relate the permutations. For example the 15-puzzle can be viewed as graph induced by permutations on 16-tuple. If we rename the right lower corner as 0 and give the blank tile the value 0 the standard goal node in this game is the permutation (0,1,2,.....15) and the start node will be some other permutation (i_0,i_1,i_2,....,i_{15}). The meaning of this permutation is that the tile with value i_j is in location j. An edge between two nodes exists iff by a single sliding of the blank tile one can move from one permutation to the other.

In this case the subgoals can be defined as follows:
sub.start.node: $(0,1,2,...j-1,x,...,x,j,x,...,x)$
sub.goal.node: $(0,1,2,....j-1,j,x,x...x)$ where x is don't care.
Only 119 such macros are required for the 15-puzzle, although about ten trillion different problems (start configurations) exist.

As explained above, the *Macro-Operator* can be chosen as the algorithm that initially approximates the solution, and then the first loop of Joint is redundant. In the Joint algorithm, after decomposing the solution to subpaths, each of them optimal, the Squeeze process is executed. The Squeeze is linear time in the length of the Global.Path, and generally the more squeezing the fewer searches will be later executed. Hence we would like to predefine the macros, a shortest path that is a solution to the subgoal, with some mutual influence such that the squeezing will be maximal . Looking for the most appropriate shortest subpath is a meaningful target since in general there is more than one shortest path. Especially, in this case of graphs induced by permutations where the subpath (macro) is a path between a set of start nodes and a set of goal nodes there are many shortest paths. We have no general scheme for how to find the macros that will guarantee maximum squeezing on the average. Yet we know how to do it in our test case, the 15-puzzle. A *Macro-Operator* with the macros picked randomly among the candidates give an average solution length of 149 moves, which is reduced by Squeeze to 139. If the macros were generated according to maximum squeezing we can achieve an average solution length of 124 after squeezing. Naturally after the squeezing process we will continue with the Joint algorithm for reducing the length of the resulting path.

VI. Experimental results

In this section, we report the results from using the *LPA** and Joint algorithms with *Macro-Operator* as an approximation algorithm. We compare our results with some of the well known search methods that generate thousands of nodes.

We selected the 15-puzzle as the domain for testing the methods because of the following three reasons. We wished to have a domain where, in theory, finding a shortest path is computationally infeasible. The second reason is that there is a lot of available data about this puzzle. The third reason is that the generalization of this puzzle satisfies the condition that the length of the smallest non-trivial cycles (CL) in the search graphs is small (< 30). These cycles are spread uniformly over the graph space.

The average solution length for the test data, using only *Macro-Operator*, with the macro designed for maximum squeeze by itself, is 143 moves before the squeeze. After applying the squeeze the average solution length is reduced by 26.2 moves to 116.8 moves.

For the Joint algorithm, with $d_{\max} = 24$ and $\delta_{joint} = 6$, the average solution length is 86.7 moves, where 5950 nodes were expanded and 9600 nodes were generated on the average.

For the *LPA** algorithm, that expands and generates about the same number of nodes as the Joint algorithm, the average solution length is 86.3 moves. This was achieved with $d_{\max} = 24$ and $\delta_{anchor} = 9$, where 6580 nodes were expanded and 10240 nodes were generated on the average.

We tested both algorithms with the following two well known admissible heuristics:
$h_1(a,b)$ = the sum of the Manhattan distance of the non-blank tiles between the start (a) and goal (b) nodes.
$h_2(a,b) = h_1(a,b) + 2 \cdot R(a,b)$

$R(a,b)$ is the number of reversals in a with respect to b. A reversal means that two tiles exist in the same row (or column) in a and b , but in an opposite order.

The following four tables correspond to the two heuristics and two algorithms. They show the reduction achieved by the local searches, the number of nodes that were generated and expanded and the number of searches as a function of $d_{\max}$ and δ_{joint} (or δ_{anchor}). All the data is the average for the 50 problems.

Table 1. *LPA** algorithm with heuristics h_1 .

$d_{\max}$	δ_{joint}	Solution Length	No. of expanded nodes	No. of generated nodes	No.of local searches
18	4	97.0	4400	6290	30.5
18	9	100.6	2790	4100	18.4
18	14	102.6	2210	3280	14.1
21	4	89.3	10120	14820	28.9
21	9	93.1	6750	10450	17.8
21	14	95.6	4730	7330	13.5
24	4	84.0	21980	32900	26.6
24	9	86.3	13860	21630	16.8
24	14	89.7	10380	16460	13.2

Table 2. *LPA** algorithm with heuristics h_2 .

$d_{\max}$	δ_{joint}	Solution Length	No. of expanded nodes	No. of generated nodes	No.of local searches
18	4	97.3	2890	4130	31.0
18	9	99.8	1810	2650	18.5
18	14	103.1	1370	2020	13.9
21	4	90.1	5250	7770	28.9
21	9	93.8	3460	5350	17.7
21	14	96.1	2650	4240	13.5
24	4	85.0	10340	15220	26.8
24	9	86.3	6580	10420	16.9
24	14	90.5	5310	8470	13.3

Table 3. Joint algorithm with heuristics h_1.

d_{max}	δ_{joint}	Solution Length	No. of expanded nodes	No. of generated nodes	No.of local searches
18	2	102.5	1640	2390	12.2
18	6	99.9	1960	2860	14.7
21	2	95.8	4420	6910	12.6
21	6	94.2	5090	7890	14.6
24	2	87.9	10170	16200	14.3
24	6	86.5	11790	18350	15.5

Table 4. Joint algorithm with heuristics h_2.

d_{max}	δ_{joint}	Solution Length	No. of expanded nodes	No. of generated nodes	No.of local searches
18	2	103.3	1080	1590	12.1
18	6	99.9	1300	1930	14.9
21	2	96.4	2440	3850	12.6
21	6	94.8	2740	4300	14.5
24	2	87.7	5340	8650	14.2
24	6	86.7	5950	9600	15.8

From the tables we can verify the following results.

1. Both algorithms generated only thousands of nodes.
2. There is no significant difference between the methods.
3. The bigger d_{max} the shorter the solution length.
4. The bigger δ_{joint} the shorter the solution length in the Joint algorithm
5. The smaller δ_{anchor} the shorter the solution length in the *LPA** algorithm.
6. Since h_2 is more informed than h_1 the number of nodes expanded (or generated) by h_2 is about half of the number of nodes expanded (or generated) by h_1.

In (Politowski and Pohl, 1984) there is a comparison between the performances of four methods using the same test data. The methods are:

a. The Heuristic Path Algorithm (HPA) (Pohl, 1971)- Unidirectional search with weighting.
b. The Heuristic Path Algorithm (HPA) (Pohl, 1971) - Bidirectional search with weighting.
c. The Bidirectional Heuristic Front to Front Algorithm (BHFFA) (De Champeux and Sint, 1977).
d. The D-node Algorithm (Politowski and Pohl, 1984).

Comparing the results obtained by the four methods and the results presented here we can conclude:

1. The other methods using "unsophisticated" heuristics cannot find a path at all or a "reasonable path", in contrast to our algorithms that always find a "reasonable" path.

2. Keeping running time the same, *LPA** and Joint algorithms yield a shorter solution than the other methods even when using "sophisticated" heuristics.

VII. CONCLUSION

The results in this paper demonstrate the effectiveness of using *LPA** or Joint. When applicable, these algorithms achieve a good solution with small execution time. These methods require an approximation algorithm as a starting point. Typically, when one has a heuristic function, one has adequate knowledge about the problem to be able to construct an approximation algorithm. Therefore, these methods should be preferred in most cases to earlier heuristic search algorithms.

REFERENCES

[1] De Champeaux, B. and Sint, L., "An improved bi-directional search algorithm," *JACM*, vol. 24, pp. 177-191, 1977.

[2] Gaschnig, J., "Performance measurement and analysis of certain search algorithms," *Ph.D thesis*, Department of Computer Science, Carnegi-Melon University, May 1979

[3] Korf, R. E., *Learning to solve problems by searching for Macro-Operators. Research Notes in Artificial Intelligence 5,* Pitman Advanced Publishing Program, 1985.

[4] Korf, R. E., "Iterative-Deepening-*A**: An Optimal Admissible Tree Search," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Vol. 2, pp. 1034-1035, 1985.

[5] Lin, S., "Computer Solutions of the Traveling-Salesman Problem," *BSTJ*, Vol. 44, pp. 2245-2269, December 1965

[6] Lin, S., "Heuristic Programming as an Aid to Network Design," *J Networks*, Vol. 5, pp. 33-43, 1975.

[7] Pearl, J., *Heuristics. Intelligent search strategies for computer problem solving,* Addison-Wesley Publishing Company, 1984.

[8] Pohl, I., "Bi-directional search," in *Bernard Meltzer and Donald Michie (editors) ,Machine Intelligence 6*, pp. 127-140, American Elsevier, New York, 1971.

[9] Pohl, I., "Practical and theoretical considerations in heuristic search algorithms," in *Bernard Meltzer and Donald Michie (editors), Machine Intelligence 8*, pp. 55-72, American Elsevier, New York, 1977.

[10] Politowski, G., "On Construction of Heuristic Functions," *Ph.D thesis*, University of California Santa Cruz, June 1986.

[11] Politowski, G. and Pohl, I., "D-Node Retargeting in Bidirectional Heuristic Search," *Proc. of the AAAI-84,* pp. 274-277, 1984.

[12] Ratner, D., "Issues in Theoretical and Practical Complexity for Heuristic Search Algorithms," *Ph.D thesis*, Department of Computer Science, University of California Santa Cruz, June 1986.