A New Bidirectional Heuristic Shortest Path Search Algorithm

Marcelo Johann[1], Andrew Caldwell[2], Andrew Kahng[2], Ricardo Reis[1]

[1]Federal University of Rio Grande do Sul (UFRGS) - Instituto de Informática
Av. Bento Gonçalves, 9500. Campus do Vale - Bloco IV
Phone.: +55 51 336-7036 or 336-6830 - Fax: +55 51 336-5576
P.O. Box 15064 - CEP 91501-970 - Porto Alegre - RS - BRAZIL
e-mail: {johann,reis}@inf.ufrgs.br

[2]University of California at Los Angeles (UCLA) - Computer Science Department
e-mail: {caldwell,abk}@cs.ucla.edu

***Abstract*— Bidirectional search and heuristic search are techniques that improve the performance of a shortest path graph search. Despite many attempts to use both at the same time, this combination has been leading to worse results in average, compared to the classic unidirectional heuristic A* algorithm. This work presents a new graph search algorithm, LCS*, that is the first to combine both techniques effectively. LCS* is a generic and simultaneous bidirectional heuristic algorithm which is faster than A* in most domains. This is achieved by using dynamic estimation with the separation of computed values for open and closed nodes. Experiments demonstrate the relative superiority of LCS* both in terms of expanded nodes and running times.**

1 Introduction

Given a degree-bounded (or, locally finite) directed graph with edge costs, the shortest path problem is to find, for a given pair of nodes s and t, a path from s to t with total cost less than or equal to the cost of any other path from s to t. An algorithm that always returns this exact result is *admissible*.

The first admissible search algorithms were Moore's (1957) and Dijkstra's (1959), which traverse the graph in a *breadth-first* manner starting from the source until the target is reached. Search algorithms work as follows. Starting with the node s, they explore some part of the graph, known as the *search tree*, by repetitive application of the *successor operator*. Each time a successor operator is applied to a node, we say that the algorithm has *expanded* that node. We say that nodes returned by the successor operator are *generated* by the algorithm. An already expanded node is called a *closed* node, and is stored in the *closed list*. Nodes generated but not yet expanded are called *open* nodes, and are stored in the *open list*. A node in the search tree is either open or closed at a given moment. If the algorithm cannot identify that a given node was already generated (or expanded), we say that the *search tree* is part of the *search space*, which differs from the *state space* because the same state can appear more than once.

In [3] Dijkstra's algorithm is extended into a more efficient algorithm called A* by including an estimate of the remaining path cost to the target. A* selects nodes for expansion in order of increasing $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to go from s to n and $h(n)$ is an estimate of the cost to go from n to t. This process searches nodes closer to the target first, and is called *heuristic search* or *best-first search*. A* is admissible if and only if the remaining path costs are *underestimated*. Although A* may be outperformed by other algorithms in some specific graph problems, it is proven to be optimal among all other unidirectional algorithms that are not *more informed* than A* and if no *ties* are allowed [1]. A *tie* occurs when two nodes n_1 and n_2 in the search's open list have the same evaluation result, e.g.: $f(n_1) = f(n_2)$. We say that there is a *critical tie* between n_1 and n_2 when $f(n_1) = f(n_2) = C^*$, where C^* is the cost of the optimal path.

A bidirectional search can be viewed as two simultaneous searches starting from source and target nodes

respectively. A new path is found when a given node is recognized to be at both search trees at the same time. This node is called a *meeting node*. The *terminating condition* of a bidirectional algorithm is not as simple as in unidirectional ones. The first complete path from s to t found by a bidirectional algorithm may not be the optimal one, and the algorithm needs to continue expanding nodes until all nodes n in at least one open list have $f(n) \geq min[f(m)]\ \forall$ meeting node m. This behavior leads to searches *overlapping*, which is the generation and/or expansion, by both searches, of the same nodes in the state space.

In [8] it is introduced BS*, a bidirectional heuristic algorithm that avoids search overlapping with *nipping, pruning, trimming* and *screening* operations. However, this wasn't sufficient to make it better than A*. There has been great difficulty in achieving improvements with heuristic bidirectional search. The main problems reported are the complexity of keeping the needed information and making both searches meet in the middle rather than each search finding essentially disjoint equivalent (optimal) $s-t$ paths (*missing fronts* problem).

Recently, the potential gain of bidirectional search has appeared again, as some previous counterarguments were shown to be weak [7] [5]. Perimeter search [9] presents better results under some conditions, but as yet no traditional bidirectional algorithm is demonstrated to perform better than A* in average. Another improvement appears in the work of [6]. As in perimeter search, searches are not simultaneous, but one following the other. This *non-traditional* bidirectional search allows dynamic estimation to be easily implemented as single added values. Statistical data show sensible reduction in number of expanded nodes and runing time. The limitation is the fact that the algorithm must know when to change direction in order to optimize its search, and if made wrong, this decision may nullify the gains or even make the search more expensive. It is best used when the available memory is not sufficient to perform a normal search, or when the average size of the searched space is known.

The present work is also based on the use of dynamic estimation to improve the pruning power of the algorithm. A new graph search algorithm is introduced, the first to combine both bidirectional and heuristic techniques effectively using simultaneous searches in a simple way. To better understand the improvements and the algorithm itself, a set of definitions and properties is reviewed in the next section. Then, in Sec. 3, the new algorithm LCS* is presented along with the formal proofs of its admissibility. Section 4 first explains some implementation details and then shows the results taken from three different domains: 2D grids, random graphs, and mazes. Finally, Sec. 5 present some concluding remarks.

2 Definitions

2.1 Graph definition

G	$= (V, E)$, directed degree-bounded graph
s, t	source and target nodes, with $s, t \in V$
$c(n_1, n_2)$	cost of arc $(n_1, n_2) \in E$
$S_s(n)$	successors of $n, \{x \mid x \in V, (n, x) \in E\}$
$S_t(n)$	predecessors of $n, \{x \mid x \in V, (x, n) \in E\}$

2.2 Optimal values

P^*_{n-m}	optimal path from $n \in V$ to $m \in V$
$k^*(n, m)$	cost of the optimal P^*_{n-m} path
$g^*(n)$	$= k^*(s, n)$, cost of the optimal $s - n$ path
$h^*(n)$	$= k^*(n, t)$, cost of the optimal $n - t$ path
$f^*(n)$	$= g^*(n) + h^*(n)$
C^*	$= k^*(s, t)$, cost of the optimal $s - t$ path

2.3 Estimates used by heuristic search

$g(n)$	estimate of $g^*(n)$
$h(n)$	estimate of $h^*(n)$
$f(n)$	$= g(n) + h(n)$, estimate of $f^*(n)$
c_{min}	$0 < c_{min} \leq c(n_1, n_2)\ \forall\ n_1, n_2 \in V \mid (n_1, n_2) \in E$
$k(n_1, n_2)$	consistent estimate of $k^*(n_1, n_2)$;

In 2D grid:
$k(n_1, n_2) = (|x(n_1) - x(n_2)| + |y(n_1) - y(n_2)|) * c_{min}$

2.4 Properties of the estimator function h

h is *admissible* iff $h(n) \leq h^*(n)\ \forall\ n \in V$ (underestimation)
h_1 is *no more informed than* h_2 if and only if $h_1(n) \leq h_2(n) \leq h^*(n)$
h is *consistent* iff $h(n_1) \leq k^*(n_1, n_2) + h(n_2)$

3 A New Bidirectional Algorithm LCS^*

This section introduces LCS*: Lower bound Cooperative Search. As any other bidirectional algorithm, LCS* requires the ability to identify the same node in the *state space*, and requires also *consistent* estimators. Improvements in LCS^* are due to two concepts: cooperation and visibility.

Cooperation is found by observing that the function $g()$ of one search corresponds to the function $h()$

of the other, and vice-versa. Then, it is possible not only to reduce the amount of information stored, but also to improve the estimates in one search based on the other's paths. This is known as dynamic estimation, and was already exploited in [6]. To use dynamic estimation, one has to make new estimation functions that account for information coming from the opposite search. But in order to keep the new estimation values consistent, it is not possible to change the ordering imposed by them. Therefore, in bidirectional algorithms, the new estimation is actually the same plain estimation, but additional values are used for the terminating conditions. This becomes clear looking at the algorithm code, and also observing the fact that the prunning power of an algorithm is the factor that makes it finish sooner or not.

The novelty of our algorithm is to show that such dynamic estimation can be used in simultaneous, or traditional algorithms. As the boundaries of both searches overlap each other, we need to prevent their temporary data to be used by the opposite search. The concept of *visibility* is to hide temporary (estimated) data inside each search, while sharing the values that are already proven to be optimal between both search fronts, in a public space. This is accomplished by storing g and h functions in *references*, but not in the actual nodes of the state space, as in previous search algorithms. Each search has its own private open list to store references generated from it, and they are not visible to the opposite search. Once a node n is expanded, the algorithm knows its $g^*(n)$ value (this is guaranteed by admissible and consistent estimators), and this value, along with a pointer, is stored in the public state space structure. LCS* is presented on figure 5. The algorithm has the same structure and steps of BS* [8].

The information maintained by the algorithm during its execution is the following:

$Closed_s$	nodes reached from s already expanded
$Closed_t$	nodes reached from t already expanded
$Open_s$	references r to nodes $n(r)$, $n(r) \in S_s(x)$ for $x \in Closed_s$ and $n(r) \notin Closed_s$
$Open_t$	references r to nodes $n(r)$, $n(r) \in S_t(x)$ for $x \in Closed_t$ and $n(r) \notin Closed_t$
$g(n)$	optimal path cost from s to n if $n \in Closed_s$ or from t to n if $n \in Closed_t$
$p(n)$	parent of n from where it was expanded with lowest cost, $n \in Closed_s \oplus Closed_t$
$g_s(r)$	estimate of $g_s^*(n(r))$, $n(r)$ is the node pointed to by r
$g_t(r)$	estimate of $g_t^*(n(r))$, $n(r)$ is the node pointed to by r
$p(r)$	parent node of $n(r)$ that generated it
$d(r)$	number of arcs in the path from s to $n(r)$ if $r \in Open_s$ or t to $n(r)$ if $r \in Open_t$
L_{min}	$\geq C^*$, cost of the best s-t path found so far
Ω_s	extra cost to reach s through $Closed_s$
Ω_t	extra cost to reach t through $Closed_t$
Py_s	Penalty: estimation of $P^*(p)$,$p \in Open_s$
Py_t	Penalty: estimation of $P^*(p)$,$p \in Open_t$
$MeetN$	meeting node $\in Closed_s \oplus Closed_t$
Mp_s	$MeetN$ pointer toward source
Mp_t	$MeetN$ pointer toward target

3.1 LCS* is Admissible

Assume initially that $Py_s = Py_t = \Omega_s = \Omega_t = 0$. Under these circumstances, it is possible to prove that LCS^* is complete and admissible. As in [8], it is not possible to prove that *every reference r to a node n expanded* by LCS* has $f(r) = f^*(n(r))$ because of *nipping/pruning* operations. The proofs, that appear in [4], are conducted as follows: There are always references to nodes *in the optimal path* in both search's open lists. Once these nodes are selected for expansion, we know their $f^*()$ values. Meeting nodes are found and the algorithm terminates with the optimal solution; In the next section we show how to use values greater than zero for Ω and Py while maintaining these properties.

3.2 Improved Dynamic Estimator

The power of a heuristic admissible algorithm is not how fast it goes toward the goal, but how efficiently it can compute a higher $f(n)$ $\forall n \mid f(n) \leq f^*(n)$. This is known as *pruning power* of the estimator or algorithm [11]. Values greater than 0 can be used for Ω_s, Ω_t and Py_s, Py_t to improve the pruning power of LCS^* and cause the algorithm to terminate sooner, without loosing consistency and admissibility properties.

Let $\Omega_\tau = min[g(p(m)) - k(p(m), \tau)]$ $\forall m \in Open_\tau$. This resistivity value is the minimum overhead to get to τ from anywhere outside $Closed_\tau$, including nodes at the boundary of $Closed_\tau$. It corresponds to the Min idea of [6], and can be added to any static estimate $k(n, \tau)$. Since the improved dynamic estimate $k(n, \tau) + \Omega_\tau \leq k^*(n, \tau)$ $\forall n \mid n \notin Closed_\tau$ or $n \in p(m) \mid m \in Open_\tau$, and no estimate is needed when $n \in Closed_\tau$ and $n \notin p(m) \mid m \in Open_\tau$, this value can be used by the algorithm (see Fig. 1).

To make the estimates consistent among all nodes in a given open list, we have to use the same value Ω_τ for all references $r \in Open_\varsigma$. By doing so, we do not break consistency, for $g_\tau(r_1) + \lambda \leq g_\tau(r_2) + \lambda + k(r_1, r_2)$ for any value of λ. It does not modify the ordering in which the references are stored (by $f(r)$) in the open list. The absolute values of $f(r)$ are only used

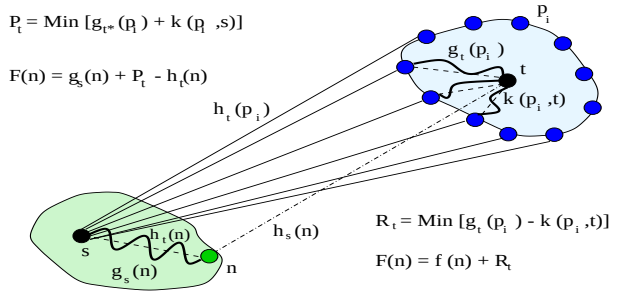

Figure 1: Dynamic estimation using Py and Ω (R) values

to compare to L_{min}, and higher values are desired to improve the pruning power. This is why $f(r) = g_\varsigma(r) + k(n(r), \tau)$ is still used in $Open_\varsigma$ (step 4.4), but $f(r) \geq L_{min} - \Omega_\tau$ is used as pruning condition (step 3.2).

Analogously, let $Py_\tau = min[g(p(m)) + k(p(m), \varsigma)]$. This penalty is the minimum cost of an $s - t$ path through node p. It includes the additional overheads of $k(p(m), \varsigma)$ and $k(p(m), \tau)$. Then, for any node n, $F(n) = g_\varsigma(n) + Py_\tau - h_\tau(n)$ is an improved admissible estimation of $f^*(n)$, and corresponds to the Max idea of [6]. It is used together with the resistivity estimator because in some cases it may be smaller than even the static f.

4 Results

In order to test the actual performance of LCS*, it was implemented in C++ and compared against the A*, which is the best generic algorithm in average.

4.1 Implementation Details

The implementations of both A* and LCS* are template classes that can operate on the space of any arbitrary domain, and use the same basic data structures. The STL priority queues were substituted by an specific Binary Heap implicitly stored in arrays. It guarantees logaritmic worst-case insertion and deletion times, and provide iterators to access all elements. The access by iterators is needed to make Resistivity and Penalty updates in amortized constant time. The period in which these values are updated is 2 times the sum of both open list sizes.

The ordering in which references and nodes are sorted is the lowest f, followed by the biggest g, what breaks ties in favor of the target. The criteria for choosing direction was basically the *cardinality principle*, but adapted with an inertial behaviour. In this case, while the expansions do not change the value of minimum f, the direction is kept unchanged. It produces small depth searches when many critical ties occur, and makes LCS* compete better with A* by preventing it from duplicating the optimal path (solving the missing front problem when it happens).

4.2 2D Grid Spaces

The first tests were performed on 2 dimensional grids filled up with random costs. 200 by 200 grids were generated using an average cost of 100, minimum and maximum limits of 50 and 150. The range (which is centered in the average 100) varies from 100 to 500, in steps of 20. When the range generates random values below the minimum, the minimum value is used, what represents a lower saturation, or empty regions. When values greater than the maximum are generated, infinity is used, what represents blocking.

For each grid on this range, 20 random pairs of points were selected and the shortest path was found using one LCS* and two A* runs, one starting from the source and the other from the target. Looking at Fig. 2, it is possible to see that LCS* expands less nodes in average than the best of A* runs. Figure 3 shows the ratios between the number of expansions in LCS* and in the average of the A* runs, what is expected in practical situations.

Figure 3 also shows the ratios of *running times* taken from the same experiments Gains are not only proportional, but in some cases a little bit better than the number of expansions. At the first time one would not expect this behaviour, as LCS* has more steps to do for a given expansion than A*. However, the time per node expanded is dominated by the insertion and deletion times in the open lists. Now, given that LCS* expands in two separated fronts, its open sizes are smaller compared to the size of the A*'s single open list, and then LCS* may spend less time per node expanded. In other domains where neighbour generation is more complex and dominates, the times will also keep proportional to the number of expansions.

4.3 ε-admissibility

In many applications, we seek for the shortest path but do not require the results to be 100% admissible. Although LCS* was not directly developed with ε-admissibility in mind, the fact that meeting occurs earlier during the search allows it to check for a prescribed error bound. This has shown to be a potentially powerful use of LCS*. After the first meeting, the allowed error is considered in the terminating conditions.

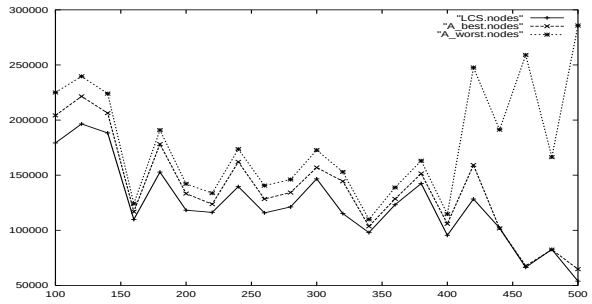

Figure 2: Number of nodes expanded by A* and LCS* in 2d grids

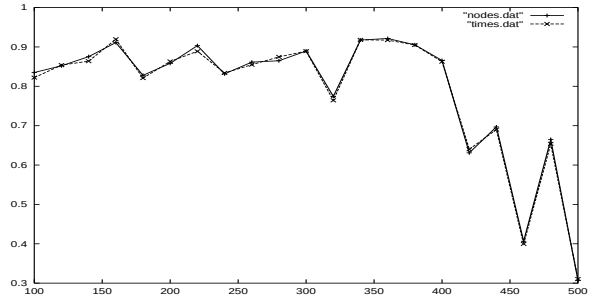

Figure 3: LCS* ratios in number of nodes and running times

Figure 4 show the results for the same problems but using now 96% admissibility both in LCS* and A*. When using LCS*, the reduction in terms of expanded nodes is much bigger than the allowed error. Yet in the A*, the error bound can only be used as an overestimating factor for h, and it must be static because there is no comparison against L_{min} (A* termination is automatic). As a consequence, the reduction in number of nodes in the ε-admissible A* is proportional to the nominal error.

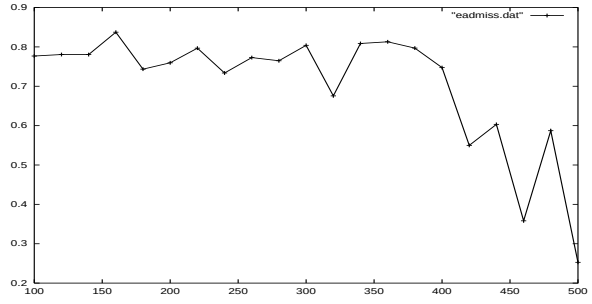

Figure 4: LCS*/A* ratios using ε-admissibility

In practice, however, the actual error is typically much smaller than the nominal guaranteed error bound. For a given application domain, one can measure typical errors by running the ε-admissible and fully admissible versions for the same sample problem instances.

The use of ε-admissible LCS* has a lower limit, however, which is given by the first meeting. Therefore, after some threshold, augmenting the allowed error bound will not make it faster anymore.

4.4 Random and Geometric Graphs

Geometric and Random graphs were generated by randomly selecting the positions of nodes in a plane, and assigning the minimum distance between two nodes to the cost of the arc that connect them, if any, and to the estimate of the shortest path between them. Then, we call random graphs the instances generated with only one arc per node, connecting it to a randomly selected partner, for which all nodes have the same chance. The true geometric graphs are those instances for which there are more arcs per node, and the arcs connect only nodes that are in a prescribed maximum distance.

Table 1 shows the superiority of LCS* for an instance of random graph with 70 nodes. The second column compute only the cases for which there was a path, because the graph is not necessarily connected. The experiment tested the shortest path for all node pairs (70 times 70), and thus represent the universe of problems in the generated graph.

Table 1: Random and Geometric Graph Results

algorithm	random	solved	geometric	easier
average A*	126156	78306	39095	28406
best A*	71576	61016	24438	21428
worst A*	180736	95596	53752	35384
LCS*	77710	59869	38662	34130

Yet in the geometric graphs, LCS* does not present a clear superiority. The experiment used 10% of probability of connecting two nodes that are in a maximum distance of 0.3 times the space side. In fact, when this geometric graph is incremented to have at least one arc per node, LCS* looses from A*, as it is shown in the fourth column. These cases must not be a concern. Geometric graphs, with arc costs corresponding to the distance, allow a very good estimation, and in most cases the nodes expanded by A* are only the nodes along the optimal path, what is absolutely optimal. On the other hand, LCS* needs to expand at least one more node in the reverse order, and the dynamic estimation is almost void. That said, we can expect

that for this easy problems A* is better, while LCS* is still better for all problem domains where finding the shortest path requires much bigger effort.

4.5 Mazes

A small set of mazes were generated with random DFS searches, producing minimum spanning trees in a 2D regular graph. In this domain, for any two nodes, the path connecting them is unique, and the problem is to find it (the word "shortest" becomes meaningless). The estimation may be good only in some portions of the problem, but are partially random, in such a way that we expected the comparison to be quite close to unidirectional versus bidirectional BFS. In fact, Table 2 shows that the bidirecitonal LCS* greatly outperforms A* on this domain.

Table 2: Maze Results

maze	LCS*	best A*	worst A*	ratio
0	13145	15025	15056	0.874003
1	5246	7205	9969	0.610923
2	7938	10081	13128	0.684074
3	6502	7959	12732	0.628516
4	6846	7486	15502	0.595615
5	5616	8628	10272	0.594286
6	5509	10020	10141	0.546528
7	11897	14234	16026	0.786319
8	3755	4055	9190	0.567049
9	5767	9401	11311	0.556875

5 Conclusions

This paper has introduced LCS*, a new algorithm for finding the shortest path by applying bidirectional and heuristic techniques effectively. The pruning power of this bidirectional algorithm is enhanced by using information from the "opposite search", and little information needs to be stored in the public state space. Practical experiments show that LCS* outperforms A* in most problem instances of many different application domains. Gains in terms of running time closely match the reduction in number of expansions. It was also shown how to exploit earlier meeting to get bounded ε-admissible searches with less effort. The new algorithm opens several possibilities for further research on cooperative searches, direction choosing criteria, among others.

References

[1] R. Dechter and J. Pearl, *Generalized Best-First Search Strategies and the Optimality of A**, J. Assoc. Comput. Mach., 32 (1985), pp. 505–536.

[2] S. Ghosh and A. Mahanti, *Bidirectional Heuristic Search with Limited Resources*, Information Processing Letters, 40 (1991), pp. 335–340.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems, Science and Cybernetics, SSC-4 (1968), pp. 100–107.

[4] M. Johann, A. Caldwell, A. Kahng, and R. Reis, *Admissibility Proofs for the LCS* Algorithm.* Accepted for publication in the proceedings of the 15th Brazilian Symposium on Artificial Intelligence, Atibaia, Brazil, Nov. 2000.

[5] H. Kaindl, G. Kainz, A. Leeb, and H. Smetana, *How to Use Limited Memory in Heuristic Search*, in Proceedings of the International Joint Conference on Artificial Intelligence, vol. 1, 1995, pp. 236–242.

[6] H. Kaindl, G. Kainz, *Bidirectional Heuristic Search Reconsidered*, in Journal of Artificial Intelligence, vol. 7, 1997, pp. 283–317.

[7] A. L. Koll and H. Kaindl, *Bidirectional Best-First Search with Bounded Error: Summary of Results*, in Proceedings of the International Joint Conference on Artificial Intelligence, 1993, pp. 217–223.

[8] J. B. H. Kwa, *BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm*, Artificial Intelligence, 38 (1989), pp. 95–109.

[9] G. Manzini, *BIDA*: An Improved Perimeter Search Algorithm*, Artificial Intelligence, 75 (1995), pp. 347–360.

[10] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill Book Co., 1971, ch. State-Space Search Methods, pp. 43–79.

[11] J. Pearl, *Heuristics*, Addison-Wesley, 1984, ch. Formal Properties of Heuristic Methods, pp. 73–87+.

[12] I. Pohl, *Bi-Directional Search*, in Machine Intelligence, 1971, pp. 127–140.

LCS* Shortest Path Algorithm

Input: $G = (V, E)$
Output: P^*_{s-t}

0. $L_{min} \leftarrow \infty$; $MeetN \leftarrow none$; $\Omega_s \leftarrow 0$; $\Omega_t \leftarrow 0$; $Py_s \leftarrow 0$; $Py_t \leftarrow 0$;
1. put a new reference r_s in $Open_s$: $n(r_s) = s, d(r_s) = 0, g_s(r_s) = 0, g_t(r_s) = dc(s,t)$;
   put a new reference r_t in $Open_t$: $n(r_t) = t, d(r_s) = 0, g_t(r_t) = 0, g_s(r_t) = dc(s,t)$;
2. if necessary, update Ω_s, Ω_t, Py_s, Py_t; // dynamic estimation updates
   if expand_from_source // direction choosing criteria
   then $\varsigma = s, \tau = t$
   else $\varsigma = t, \tau = s$; endif
3. select $r \in Open_\varsigma \mid f(r) \leq f(x)\ \forall x \in Open_\varsigma; n \leftarrow n(r)$; // selection
3.1 if $Open_\varsigma$ or $Open_\tau$ is empty // stop condition 1
3.2 or $f(r) \geq L_{min} - \Omega_\tau$ // stop condition 2
   then goto step 5 (stop);
   endif
3.25 if $g(r) - k(\varsigma, n(r)) > L_{min} - Py_\tau$ // stop condition 2
   then return to step 2; // screening 2
   endif
3.3 if $n = \tau$ [and $g_\varsigma(r) < L_{min}$] // target meeting node
   then
     $L_{min} \leftarrow g_\varsigma(r)$;
     $MeetN \leftarrow n$; $Mp_\varsigma \leftarrow p(r)$; $Mp_\tau \leftarrow none$;
     goto step 2;
   endif
4. // check/expand node $n = n(r)$ (all steps 4)
4.1 if $n \in Closed_\varsigma$ [and $g(n) \leq g_\varsigma(r)$]
   then return to step 2
4.2 if $n \in Closed_\tau$ // nipping/pruning
   then
     if $(g_\varsigma(r) + g(n) < L_{min})$
     then
       $L_{min} \leftarrow g_\varsigma(r) + g(n)$;
       $MeetN \leftarrow n$; $Mp_\varsigma \leftarrow p(r)$; $Mp_\tau \leftarrow p(n)$;
       goto step 2;
     endif;
     goto step 2; // trimming
4.3 else put n in $Closed_\varsigma$; $g(n) \leftarrow g_\varsigma(r)$; $p(n) \leftarrow p(r)$;
   endif
4.4 for each node m in $S_\varsigma(n)$ do: // expansion
     if $m \notin Closed_\varsigma$
     and $g(n) + c(n,m) + k(m,\tau) < L_{min} - \Omega_\tau$ // screening
     then
       put a new reference r_m in $Open_\varsigma$
           with: $n(r_m) = m, p(r_m) = n, d(r_m) = d(r) + 1$,
       $g_\varsigma(r_m) = g(n) + c(n,m), g_\tau(r_m) = k(m,\tau)$;
4.5   // earlier termination
       if $m \in Closed_\tau$ and $g_\varsigma(r_m) + g(m) < L_{min}$
       then
         $L_{min} \leftarrow g_\varsigma(r_m) + g(m)$;
         $MeetN \leftarrow m$; $Mp_\varsigma \leftarrow n$; $Mp_\tau \leftarrow p(m)$;
       endif
     endif
   endfor
4.6 remove and delete r from $Open_\varsigma$;
   goto step 2.
5. if $L_{min} < \infty$
   then
     the shortest path is traced from $MeetN$ with Mp_s, Mp_t
     and $p(n)$ pointers until reaching s and t, and has cost L_{min};
   else no path exists;

Figure 5: Basic LCS^* Algorithm.