# An Efficient Bidirectional Graph Search Algorithm *

**Andrew Caldwell (doctoral student at UCLA)**
**Marcelo de Oliveira Johann (doctoral student at UFRGS)**
**Andrew B. Kahng (co-advisor - UCLA)**
**Ricardo Augusto da Luz Reis (advisor - UFRGS)**
**johann@inf.ufrgs.br**

**Abstract**

*Bidirectional search and heuristic search are techniques that improve the performance of a shortest path graph search. Despite many attempts to use both simultaneously, this combination has been leading to worse results in average, if compared to a simple unidirectional heuristic search, the A* algorithm. This work presents a new graph search algorithm that is the first to combine both techniques effectively, resulting in a cooperative bidirectional search capable of reducing the number of search nodes when compared to A*. The algorithm is simple, fast and requires less storage space when used on specific applications such as VLSI routing.*

## 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* area allowed [1]. A *tie* occurs when two nodes $n_1$ and $n_2$ in the search's open list have the same

*This work was partially supported by the UCLA VLSI-CAD laboratory, under direction of Prof. Andrew B. Kahng. However, this paper was compiled at UFRGS. Any mistakes or omissions should not be attributed to all authors.

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 [6] it is introduced a bidirectional heuristic algorithm that avoids search overlapping with *nipping, pruning, trimming* and *screening* operations. However, this wasn't sufficient to make it always 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 counter-arguments were shown to be weak [5] [4] [7], but as yet no bidirectional algorithm is demonstrated to perform better than A*. This work introduces a new graph search algorithm that is the first to combine both bidirectional and heuristic techniques effectively, resulting in a cooperative search capable of reducing the number of search nodes when compared to A*.

## 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 $(n_1, n_2) \in E$ for nodes $n_1, n_2 \in V$ |
| $S_s(n), n \in V$ | immediate successors of $n \in V, \{x \vert x \in V, (n, x) \in E\}$ |
| $S_t(n), n \in V$ | immediate predecessors of $n \in V, \{x \vert x \in V, (x, n) \in E\}$ |

### 2.2. Optimal values

| | |
|---|---|
| $P^*_{n-m}$ | optimal path from $n \in V$ to $m \in V$ |
| $P^*(n)$ | optimal path from $s$ to $t$ constrained to pass through node $n \in V$, $(= P^*_{s-n} \cup P^*_{n-t})$ |
| $k^*(n, m)$ | cost of the optimal path from any two nodes $n, m \in V$ |
| $g^*(n)$ | $= k^*(s, n)$, cost of the optimal path from $s$ to $n, n \in V$ |
| $h^*(n)$ | $= k^*(t, n)$, cost of the optimal path from $t$ to $n, n \in V$ |
| $f^*(n)$ | $= g^*(n) + h^*(n)$, cost of $P^*(n)$ |
| $C^*$ | $= k^*(s, t)$, cost of the optimal path from $s$ to $t$ |

### 2.3. Properties of the optimal values

**Consistency** : $k^*(n_1, n_3) \leq k^*(n_1, n_2) + k^*(n_2, n_3)$

**Corollaries** : $g^*(n_1) \leq k^*(n_1, n_2) + g^*(n_2)$ and $h^*(n_1) \leq k^*(n_1, n_2) + h^*(n_2)$

**Ordering** : For any nodes $n_i$ and $n_{i+1}$ in any optimal path $P^*_{s-t}$, we have: $g^*(n_i) < g^*(n_{i+1})$ ; $h^*(n_i) > h^*(n_{i+1})$ ; $f^*(n_i) = f^*(n_{i+1})$ .

**Independence** : The values of $g^*(n), \forall\ n \in V$, are completely determined by the choice of source node $s$, and do not depend on the target node $t$. The values of $h^*(n), \forall n \in V$, are completely determined by the choice of target node $t$, and do not depend on the source node $s$.

## 2.4. Information used by heuristic search

| | |
|---|---|
| $g(n)$ | estimate of $g^*(n)$ |
| $h(n)$ | estimate of $h^*(n)$ |
| $f(n)$ | $= g(n) + h(n)$, estimate of $f^*(n)$ |

## 2.5. Properties of the estimator function $h$

$h$ is said to be *static* iff $h(n)$ has the same value at any time for every node $n \in V$
$h$ is said to be *dynamic* if $h(n)$ at time $t_1 \neq h(n)$ at time $t_2$ for some node $n \in V$
$h$ is said to be *admissible* iff $h(n) \leq h^*(n)$ for every node $n \in V$ (underestimation)
$h_1$ is said to be *no more informed than* $h_2$ iff $h_1(n) \leq h_2(n) \leq h^*(n)$ for every node $n \in V$
$h$ is said to be *monotone* iff $h(n_1) \leq c(n_1, n_2) + h(n_2)$ for all nodes $n_1, n_2 \in V$ with $n_2 \in S_s(n_1)$
$h$ is said to be *consistent* iff $h(n_1) \leq k^*(n_1, n_2) + h(n_2)$ for any nodes $n_1, n_2 \in V$[1]

## 2.6. Properties of A* [3] [1]

**Continuity** — Anytime before termination, for any nonclosed node $n$ there exists an open node $n'$ on $P^*(n)$ with $g(n') = g^*(n')$. As a consequence, anytime before termination, there exists an open node $n' \in P^*_{s-t} \mid f(n') \leq C^*$, since there is at least one node $n'$ with $g(n') = g^*(n')$, $h(n') \leq h^*(n')$, and then $f(n') \leq f^*(n') = C^*$.

**Completeness** — A* is complete for degree-bounded graphs, i.e., the algorithm terminates if there is a solution. A* is also complete for infinite graphs if there is a finite solution (bounded $C^*$) and if there is a value $\delta \mid c(n, m) > \delta > 0,\ \forall\ n, m \in V$.

**Admissibility** — A* is admissible if the heuristic estimator is admissible.

**Sufficient condition for expansion** — A* expands all nodes with $f^*(n) < C^*$.

**Necessary condition for expansion** — Only nodes with $f^*(n) \leq C^*$ can be expanded by A*.

**Indeterminism on critical ties** — The number of expanded nodes with $f(n) = C^*$ depends on the problem instance and on the tie-breaking rule.

**Monotonic Evaluation** — A* never reopens nodes when the heuristic estimator is consistent, and if node $n_2$ is expanded after node $n_1$, then $f(n_2) \geq f(n_1)$. In other words, an open node $n$ with the lowest $f(n)$ has $g(n) = g^*(n)$.

**Admissible Dominance** — If $h_2$ is more informed than $h_1$ and both are admissible, then A* using $h_2$ will expand less nodes than A* using $h_1$.

**Optimality** — If the estimates have the consistency property and if no ties are allowed, then A* is optimal among all other unidirectional admissible search algorithms that are not more informed than A*;

## 3. A SIMPLE 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.

As the boundaries of both searches overlap each other, we need to prevent their temporary data to be used by the opposite search. 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, what is not visible to the opposite search. Once a node $n$ is expanded, the algorithm knows its $g^*(n)$ value, and this value, along with a pointer, is stored in the public state space structure. LCS* is presented on figure 1.

The heuristic information available to the algorithm is:

| | |
|---|---|
| $c_{min}$ | $0 < c_{min} \leq c(n_1, n_2)\ \forall\ n_1, n_2 \in V \mid (n_1, n_2) \in E$ |
| $k(n_1, n_2)$ | consistent estimate of $k^*(n_1, n_2)$; For example, in 2D grid graphs, $k(n_1, n_2) = (\lvert x(n_1) - x(n_2)\rvert + \lvert y(n_1) - y(n_2)\rvert) * c_{min}$ |

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

| | |
|---|---|
| $Closed_s$, $Closed_t$ | set of nodes reached from $s$ or $t$ and already expanded |
| $Open_s$ | set of references $r$ to nodes $n(r)$ where $n(r) \in S_s(x)$ for some $x \mid x \in Closed_s$ and $n(r) \notin Closed_s$ |
| $Open_t$ | set of references $r$ to nodes $n(r)$ where $n(r) \in S_t(x)$ for some $x \mid x \in Closed_t$ 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 $n \in Closed_t$ |
| $p(n) \in V$ | parent of $n$ from where it was expanded with lowest cost, where $n \in Closed_s \oplus Closed_t$ |
| $g_s(r)$ | estimate of $g_s^*(n(r))$ where $n(r)$ is the node pointed to by $r$ |
| $g_t(r)$ | estimate of $g_t^*(n(r))$ where $n(r)$ is the node pointed to by $r$ |
| $p(r) \in V$ | parent node of $n(r)$ from where this reference was generated |
| $d(r)$ | number of arcs in the path from $s$ to $n(r)$ if $r \in Open_s$ or from $t$ to $n(r)$ if $r \in Open_t$ |
| $L_{min}$ | $\geq C^*$, cost of the best $s$-$t$ path found so far |
| $\Omega_s$ | minimum 'resistance' to reach $s$ through $Closed_s$ |
| $\Omega_t$ | minimum 'resistance' to reach $t$ through $Closed_t$ |
| $MeetingNode$ | $= n, n \in Closed_s \oplus Closed_t$, and $f^*(n) = L_{min}$ |
| $Mp_s, Mp_t$ | $MeetingNode$ pointers toward source and target |

### 3.1. LCS* is Admissible

We will assume initially that $\Omega_s = \Omega_t = 0$. Under these circumstances, it is possible to prove that $LCS^*$ is complete and admissible. The proof uses optimal values in the symmetric notation, and specific data stored by the algorithm, as just defined above. The estimated evaluation function $f()$ is defined only on references, as $f(r) = g_s(r) + g_t(r)$, while the optimal function $f^*()$ is defined only on actual nodes, as $f^*(n) = g^*_s(n) + g^*_t(n)$. This helps understanding the differences between instance properties (optimal values) and information stored by the algorithm. We shall use indexes $\varsigma$ to designate "this search" and $\tau$ to designate the "the opposite search". So, $\varsigma = s$ and $\tau = t$ in the search beginning at $s$, and $\varsigma = t$ and $\tau = s$ in the search beginning at $t$.

As in [6], 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 proof is 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 $\Omega_s$ and $\Omega_t$ while maintaining these properties.

**lemma 0** Anytime during execution $C^* \leq L_{min}$

Proof: At the beginning (step 0) $L_{min} = \infty$. $L_{min}$ is updated at the meeting nodes by $g_\varsigma(r) + g(n)$ where $n = n(r)$, $r \in Open_\varsigma$ and $n \in Closed_\tau$. Now $g^*_\varsigma(n) \leq g_\varsigma(r)$ and $g^*_\tau(n) \leq g(n)$. Therefore, $f^*(n) \leq g_\varsigma(r) + g(n)$. Since $C^* \leq f^*(n)\ \forall n \in V$, $C^* \leq g_\varsigma(r) + g(n)$ for all meeting nodes $n$.

**lemma 1** $L_{min}$ is non-increasing. Proof: conditions at steps 3.3, 4.2 and 4.5 of the algorithm. Thus, if at any given moment there is some value $x \geq L_{min}$, the same value $x$ will remain $\geq L_{min}$ from this moment on, until the algorithm terminates;

**lemma 2** No reference $r$ with $f(r) \geq L_{min}$ is ever expanded. Proof: terminating condition (3.2);

**lemma 3** Before LCS* **finds a meeting node on** $P^*_{s-t}$, there exist references $r'_s \in Open_s$ and $r'_t \in Open_t$ such that $n(r'_s)$ and $n(r'_t) \in P^*_{s-t}$, and $g_s(r'_s) = g^*_s(n(r'_s))$ and $g_t(r'_t) = g^*_t(n(r'_t))$ .

Proof for $r'_s$: Before the first node from source is expanded, there is a reference $r^0_s \in Open_s$ with $n(r^0_s) = s$, $g_s(r^0_s) = g^*_s(n(r^0_s)) = 0$ (step 1) and trivially $r'_s = r^0_s$ because $s \in P^*_{s-t}$. Each time a reference $r_s \in Open_s \mid g_s(r_s) = g^*_s(n(r_s))$ is selected for expansion, a new reference $r'_s \mid n(r'_s) \in P^*_{s-t}$ is inserted in $Open_s$ with $g_s(r'_s) = g^*_s(n(r'_s))$, for the following reasons: (i) if $n(r_s) \notin Closed_t$ it will certainly be expanded (condition at step 4.1). Otherwise, this reference is considered as a meeting node; (ii) $g_s(r'_s) = g_s(r_s) + c(r_s, r'_s)$, and then $g_s(r'_s) = g^*_s(n(r'_s))$ because $r_s, r'_s \in P^*_{s-t}$, which is optimal; (iii) $f(r'_s) \leq C^* \leq L_{min}$ (lemmas 0 and 1), and then $r'_s$ cannot be discarded by screening. Therefore, we prove this lemma by induction.

**lemma 4** Before LCS* **finds a meeting node on** $P^*_{s-t}$, every node $n \in P^*_{s-t}$ expanded by LCS* from a reference $r \mid n = n(r)$ have $g_\varsigma(r) = g^*_\varsigma(n)$.

Proof: Suppose the contrary: $r \in Open_s, f(r) \leq f(x)\ \forall x \in Open_s$ and $g_s(r) > g^*_s(n)$. Let $P^0_{s-n}$ be the path already found from $s$ to $n(r)$ Then, there should be an optimal path $P^*_{s-n}|P^*_{s-t} = P^*_{s-n} + P^*_{n-t}$ with cost $C(P^*_{s-n}) = g^*_s(n) < C(P^0_{s-n}) = g_s(r)$ and the algorithm did not find it (if $P^*_{s-n}$ was already found, $n \in Closed_s$, and $r$ would not be expanded because of the condition at 4.1). There is a reference $r'_s \in Open_s \mid n(r'_s) \in P^*_{s-n}$ with $g_s(r'_s) = g^*_s(n(r'_s))$ (from lemma3) because $n$ was not expanded before. Now, if $r'_s \in Open_s$ at this moment, $g_s(r'_s) < g_s(r)$ (from hypothesis), $g_t(r'_s) \leq g_t(r) + k(r'_s, r)$ (consistency) and therefore $f(r'_s) < f(r)$, contradicting the selection of $r$ for expansion.

**corollary 1** The condition at step 4.1 is always true for nodes along the optimal path. Therefore, we can remove it and the above mentioned properties still hold.

**corollary 2** Every meeting node $n = n(r) \in P^*_{s-t}$ corresponding to a reference $r$ have $g_\varsigma(r) + g(n) = f^*(n)$. This value is then assigned to $L_{min}$ (steps 3.3, 4.2, 4.5).

**theorem 1** LCS* is complete.

Proof: If there is a finite path $P^*_{s-t}$ with cost $C^*$ and the cost of any arc is at least $c_{min}$, then for any reference $r$ to a node $n(r)$ further than $M = C^*/c_{min}$ steps from $s$, we have $f(r) \geq g(r) \geq Mc_{min} = C^*$. Clearly, no reference $r$ to a node $n(r)$ further than $M$ steps from $s$ is ever expanded, for by the Lemma 3, the corollary 2 to Lemma 4, and the Lemma 2.

**theorem 2** LCS* is admissible.

Step a: From the Lemma 3 (there are references with $f(r) \leq f^*(n(r))$ in the open lists), Lemma 0 ($f^*(n(r)) = C^* \leq L_{min}$), and Lemma 2, we prove that at a certain time a reference to a node in the optimal path will be selected for expansion and recognized as a meeting node with $g_\varsigma(r) + g(n) = f^*(n)$.

Step b: After a meeting node $m$ in the optimal path is selected (comment on corollary 2), no other meeting node can be selected, for $L_{min} \leftarrow f^*(m) = C*$.

## 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 [9]. Values greater than 0 can be used for $\Omega_s$ and $\Omega_t$ 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 $\tau$ from anywhere outside $Closed_\tau$, including nodes at the boundary of $Closed_\tau$. So, it 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.

To make the estimates consistent among all nodes in a given open list, we have to use the same value $\Omega_\tau$ 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 $\lambda$. 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 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).

# 4. CONCLUSIONS

This paper has introduced a new algorithm for finding the shortest path by applying bidirectional and heuristic techniques effectively. Theorems show that the algorithm is complete and admissible. The pruning power is enhanced by using information from the "opposite search", and less space is required to store data in the public state space. Next steps include the implementation and test of the algorithm in generic and domain-specific graph problems, comparing it to other similar algorithms (to BS* in particular).

# 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] 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.

[5] 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.

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

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

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

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

[10] 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\. $L_{min} \leftarrow \infty; MeetingNode \leftarrow none;\ \Omega_\varsigma \leftarrow 0; \Omega_\tau \leftarrow 0;$

1\. put a new reference $r_s$ in $Open_s$ with: $n(r_s) = s, d(r_s) = 0, g_s(r_s) = 0, g_t(r_s) = dc(s,t);$
put a new reference $r_t$ in $Open_t$ with: $n(r_t) = t, d(r_s) = 0, g_t(r_t) = 0, g_s(r_t) = dc(s,t);$

2\. 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 $Open_\varsigma$ is empty // stop condition 1

3.2 or $f(r) \geq L_{min} - \Omega_\tau$ // stop condition 2
then
goto step 5 (stop);
endif

3.3 if $n = \tau$ and $g_\varsigma(r) < L_{min}$ // target meeting node
then
$L_{min} \leftarrow g_\varsigma(r);$
$MeetingNode \leftarrow n;\ Mp_\varsigma \leftarrow p(r); Mp_\tau \leftarrow none;$
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
endif

4.2 elsif $n \in Closed_\tau$ // nipping/pruning
then
if $(g_\varsigma(r) + g(n) < L_{min})$
then
$L_{min} \leftarrow g_\varsigma(r) + g(n);$
$MeetingNode \leftarrow n;\ Mp_\varsigma \leftarrow p(r); Mp_\tau \leftarrow p(n);$
goto step 2;
endif; // 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
$L_{min} \leftarrow g_\varsigma(r_m) + g(m);$
$MeetingNode \leftarrow m;\ Mp_\varsigma \leftarrow n; Mp_\tau \leftarrow p(m);$
endif
endif
endfor

4.6 remove and delete $r$ from $Open_\varsigma$;
if necessary, update $\Omega_\varsigma$; // computation of the improved lower bound
goto step 2.

5\. if $L_{min} < \infty$
then
the shortest path is traced from $MeetingNode$ with $Mp_s, Mp_t$
and $p(n)$ pointers until reaching $s$ and $t$, and has cost $L_{min}$;
else no path exists;
endif

Fig. 1: Basic $LCS^*$ Algorithm.