Annales Univ. Sci. Budapest., Sect. Comp. 15 (1995) 93-108

WHICH OF GRAPHSEARCH VERSIONS IS THE BEST?

T. Gregorics (Budapest,Hungary)

Abstract. There is a critical point of graphsearch algorithms when a better path has been found to the node which was expanded before. The literature mentions two versions of the general graphsearch algorithm which handle this problem. One of them neglects descendants of the regenerated node of this sort, but the other one propagates the improvement downward to its descendants. In this paper we give a method for this latter idea. We compare the properties and complexities of the two versions and then the decision will be taken which algorithm is better.

1. Introduction

A number of problems in the artificial intelligence (AI) area can be related to the general problem of finding a path through a space of problem states from the initial state to any goal one. In this *state-space representation* any problem can be treated as a directed graph. Thus to obtain a solution to such problem means finding a path in the graph from the start node to a goal one.

A graph (N, A) is defined as a set of nodes (N) and a set of arcs (A). Each arc is directed from a node to another one. We use the notation $c(n, m)$ to denote the cost of an arc directed from node n to node m. We assume that these costs are all greater than any arbitrarily small positive number δ. Each node of the graph under consideration has only a finite number of arcs. We call these directed δ-graphs *the representation graphs.*

One of the wellknown search techniques solving AI problems is the graphsearch. The graphsearch algorithms discover step by step the representation graph of any problem starting from start node s. The subgraph that has been

learnt by the algorithm at any stage is called *the search graph (G)*. It contains two kinds of nodes: *closed nodes* and *open nodes*. The node which has already been expanded is the closed node, and the one which has been generated but not expanded is the open node. An *evaluation function (f)* helps to select one node from the set of open nodes *(OPEN)* for expansion. (At any stage the open node with the smallest value of function f is selected.) Taking stricter and stricter conditions on the evaluation function we get several classes of graphsearch algorithms. The *expansion* (Γ) of this node means that its successsors are built into the search graph with their arcs. The algorithm terminates if goal node is selected for expansion or there are no open nodes.

The basic scheme of graphsearch is the following:

Procedure *GRAPHSEARCH*
$G \leftarrow \{s\}$; $OPEN \leftarrow \{s\}$
$n \leftarrow s$
while *not* $(goal(n)$ *or* $empty(OPEN))$ **loop**
$OPEN \leftarrow OPEN \setminus \{n\}$
$OPEN \leftarrow OPEN \cup \Gamma(n)$
$G \leftarrow G \cup \Gamma(n)$
$n \leftarrow min_f(OPEN)$
endloop
end

where $OPEN \leftarrow OPEN \cup \Gamma(n)$ put all successors of node n into set *OPEN*, and $G \leftarrow G \cup \Gamma(n)$ moves them together with the arcs directed from node n into search graph G.

When a graphsearch algorithm expands a node it may occur that some of its successors have already been generated, since there may exist several paths from the start node to any node of the representation graph. As it may be important to give only one path from the start node to any goal node, at the termination we must record one path from the start node to any node of the search graph unambiguously. To record the paths we will use pointers (p) directed from all nodes to one of its parents. Thus we get a network determined by pointers over the search graph called *the spanning tree*.

When the graphsearch algorithm generates a node generated before, a better (cheaper) path may be found to the node different from the one already recorded in the spanning tree. (The cost of a path between two nodes can be computed by summing the arc costs.) Our aim is to preserve the cheapest path in the spanning tree from the start node to any other node. When a newly

found path is less costly than the older one, the spanning tree is adjusted by changing the parentage of regenerated node to its more recent parent.

Let g be a cost-function and $g(m)$ show the cost of a path from the start node to node m. When another path has been found to m (let n be the parent of m on this path), then we must compare the cost of the former path $g(m)$ and the cost of new path going via node n $g(n) + c(n, m)$. If this new cost is less than the cost of node m, it means that we have found a cheaper path to m, so we must redirect its pointer to the node n.

When a regenerated node has got descendants in the search graph, i.e. it is a closed node, and we have just found a better path to it, then one should propagate the improvement to its descendants, because there may exist a better path going via this node just regenerated to some of its descendants than the cost of the one recorded now. Otherwise the spanning tree loses its *optimality* over the search graph. (We remark that the optimal tree over the search graph means that it recorded the cheapest path to any node inside the search graph, but it is not always the cheapest path inside the representation graph.) Moreover these descedants can be inconsistent, i.e. the cost of the path recorded by pointers is not equal to the recorded value of cost-function g. (We can say that the spanning tree is consistent if all of its nodes are consistent.)

There are two versions of graphsearch for solving this problem. The first one published by Hart, Nilsson and Raphael [5] disregards the consistence of the spanning tree. In our paper we will call their algorithm the HNR-version. The other solution derived by Nilsson [8] maintains the consistence of the spanning tree. We will call his algorithm the N-version.

2. The N-version

The algorithm published by Nilsson [8] with minor formal modifications is the following:

Procedure *N-GRAPHSEARCH*

$G \leftarrow \{s\}$; $OPEN \leftarrow \{s\}$; $g(s) \leftarrow 0$; $p(s) \leftarrow nil$
$n \leftarrow s$
while *not (goal(n) or empty(OPEN))* **loop**
    $OPEN \leftarrow OPEN \setminus \{n\}$
    $M \leftarrow \Gamma(n)$
    while *not empty(M)* **loop**

$m \leftarrow member(M)$; $M \leftarrow M \setminus \{m\}$
if $m \notin G$ **then**
$g(m) \leftarrow g(n) + c(n,m)$; $p(m) \leftarrow n$
$OPEN \leftarrow OPEN \cup \{m\}$
elsif $m \in OPEN$ *and* $g(m) > g(n) + c(n,m)$ **then**
$g(m) \leftarrow g(n) + c(n,m)$; $p(m) \leftarrow n$
elsif $m \in G \setminus OPEN$ *and* $g(m) > g(n) + c(n,m)$ **then**
$g(m) \leftarrow g(n) + c(n,m)$; $p(m) \leftarrow n$
determine whether or not to correct values of g
and to redirect pointers
for each of descendants of node m in G
endif
endloop
$G \leftarrow G \cup \Gamma(n)$
$n \leftarrow min_f(OPEN)$
endloop
end *N-GRAPHSEARCH*

This algorithm maintains an *optimal* and *consistent* spanning tree over the search graph. It means that it contains the cheapest path in the search graph from the start node to each node (all paths are optimal inside the search graph), and all nodes are consistent, i.e. the value of cost-function g for every node of the search graph is equal to the cost of the path (from the start node to the given node) recorded by pointers. When a cheaper path may be found to any closed node then the optimality and the consistency of its descendants may be lost, but the N-version adjusts the optimality and the consistency of spanning tree. Unfortunately Nilsson does not give any information about how to walk over all these descendants. Rich [10] suggested a depth-first traversal for this problem, but her method may check a node more than once.

Now we introduce another method to solve the problem of walking over all descendants which checks a node only once. Let n be the expanded node. Since we have found a better path to its successors we must propagate the improvement to its descendants. So to propagate the new cost downward means a uniform-search [1] traversal of the subgraph of descendants of node n by starting from this node changing the value of function g and the pointer at each node. This propogation is terminated when each branch reaches either a node with no successors or a node to which an equivalent or better path has already been found than the one going via the node n. (Hence this checking guarantees that this method will terminate even if there are cycles in the graph.)

Otherwise the cost value and the pointer value of current closed node must be changed and all of its successors are reproduced (this reproduction is just like an expansion). We will call this change and reproduction as *correction.*

The uniform-search walks over all descendants in the order of the cost of the path from node n to them. Thus this method guarantees that each descendant will be checked at most once. Nevertheless the implementation of this method is not very simple. It is a new graphsearching method inside the N-version.

All brute-force algorithms suffer in efficiency from the fact that they are essentially blind searches, they use no domain knowledge to guide the choice of which nodes to expand next. The heuristic search takes advantage of the fact that most problem spaces provide relatively smaller computational cost, some information distinguishes states in terms of their likelihood to lead to a goal state. This information is called a *heuristic function.*

In the context of a single-agent problem a heuristic function is a mapping from a pair of states to a number that estimates the distance in the problem space between the two states. For example in navigating from one point to another in a network of roads the airline distance between a pair of locations gives a rough estimate of distance in the road network between the states at a low computational cost. Another example is a common heuristic function for the Eight Puzzle (It consists of a 3×3 square frame containing 8 numbered square tiles and an empty position. The legal operators slide any tile horizontally or vertically adjacent to the empty position. The task is to rearrange the tiles from some random initial configuration to a particular goal configuration), called Manhattan distance: for each tile not in its goal position the distance in number of moves along the grid from its goal position is computed, and these values are summed over all tiles. The Manhattan-distance is the *heuristic function*, i.e. an estimate of number of moves required to get from one state to another, and it can be computed much faster than actually solving the problem and counting the moves. If we fix the goal state, then the heuristic function becomes the function of one state computing the distance to the given goal state.

If the evaluation function is calculated for any node n as $f(n) = g(n)+h(n)$ (the heuristic function h is greater than or equal to the zero function) we call this graphsearch algorithm *algorithm A.* (The heuristic function estimates the merit of each node we generate.) This enables the algorithm to search more promising paths first.) We know that the algorithms A always find a path from the start node to a goal node if there exists a path between the start node and the goal nodes [2]. When the heuristic function of algorithm A for any node n is supposed to be a lower bound to the cost of a minimal path from node n to the nearest member of goal nodes, we call it *algorithm* A^*. It is wellknown

that algorithm A^* is *admissible*, i.e. it always finds an optimal path from the start node to the goal nodes whenever this path exists.

Finally let us examine the complexity of the N-version of algorithm A^*. Let k be the number of expanded nodes of the search graph at termination. It means that the number of expansions is also k for the N-version. The running time depends on the number of expansions and the number of corrections after each expansion. It is clear that the running time of one correction is almost equal to the time of one expansion. Hence one part of running time of the algorithm is measured by the sum of numbers of expansions and corrections. (Now we disregard the other part of running time, namely the running time of selection depending on the complexity of evaluation function.) In the worst case our algorithm corrects all closed nodes of search graph at any stage. At the first expansion there are no closed nodes, at the second one there is at most one, finally there are $k-1$ closed nodes. Thus the number of expansions and corrections is $k*(k+1)/2$, that is $\frac{1}{2}k^2+O(k)$.

3. The HNR-version

The HNR-version was made earlier than the N-version. The only difference between these algorithms is that the N-version saves the optimality and consistency of spanning tree, the HNR-version not. The HNR-version disregards descendants of the regenerated node, but it puts this node back to the set OPEN. Thus this node is open and closed at the same time, and it may be selected for expansion again. The algorithm discovered by Hart, Nilsson and Raphael [5] with minor modifications is the following:

```
Procedure HNR-GRAPHSEARCH
    G ← {s}; OPEN ← {s}; g(s) ← 0; p(s) ← nil
    n ← s
    while not (goal(n) or empty(OPEN)) loop
        OPEN ← OPEN \ {n}
        M ← Γ(n)
        while not empty(M) loop
            m ← member(M); M ← M \ {m}
            if m ∉ G or g(m) > g(n) + c(n,m) then
                g(m) ← g(n) + c(n,m); p(m) ← n
                OPEN ← OPEN ∪ {m}
            endif
```

endloop
$G \leftarrow G \cup \Gamma(n)$
$n \leftarrow min_f(OPEN)$
endloop
end $HNR-GRAPHSEARCH$

Let us look at an example which shows that the spanning tree of the HNR-version includes *non-consistent nodes*, in addition it terminates with finding a non-consistent goal node. In Figure 1 we can see the search graph with pointers after termination. We use s, a, b, t to denote the nodes (s is the start node, t is the goal node), and write the costs on the arcs. The evaluation function is computed as $f(n) = h(n) - g(n)$ for any node n.

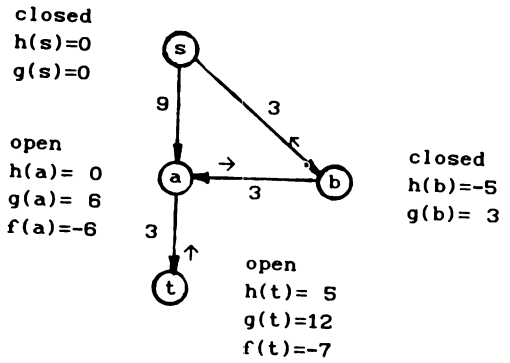

Figure 1.

Fortunately, if the evaluation function is calculated for any node n as $f(n) = g(n) + h(n)$, where the heuristic function h is greater than or equal to zero function, then the HNR-version never expands non-consistent nodes. It means when a goal node is selected for expansion the cost value of this goal node is equivalent to the cost of recorded path. Now we will prove this predicate in more general form.

Theorem 1. *The HNR-version using a decreasing evaluation function always expands consistent nodes.*

Remark. The *decreasing evaluation function* means that if any closed node gets into the set OPEN repeatedly, then the value of this node must decrease. The evaluation function of algorithm A is decreasing function since

the heuristic function h remains unchanged during the run of program and the value of function g may only decrease.

Proof. Let us imagine that an HNR-version has produced the nodes n and m of a representation graph, the node m is a descendant of node n. At first node n had been generated via path α from the start node ($\alpha \in \{s \rightarrow n\}$). Then the node m had been discovered along the path γ from the node n to it ($\gamma \in \{n \rightarrow m\}$). In the end a cheaper path β has been found for node n $\beta \in \{s \rightarrow n\}$ and $k^{\alpha}(s,n) > k^{\beta}(s,n)$, where $k^{\alpha}(s,n)$ means the cost of path α). Let $f(n)$ be the previous value and $f'(n)$ be the new value of evaluation function for node n, where $f'(n) < f(n)$. Now both nodes waiting for expansion are in the set $OPEN$ and node m is not consistent (Figure 2). It is enough to show that the algorithm must expand node n before the node m. Then it follows that node m cannot be expanded as long as any of its ancestors are in set $OPEN$ to which a better path has been found, i.e. until node m is consistent.

We suppose that the algorithm expands node m before the node n is expanded. It means that $f(m) \leq f'(n)$. Let us examine this process step by step.

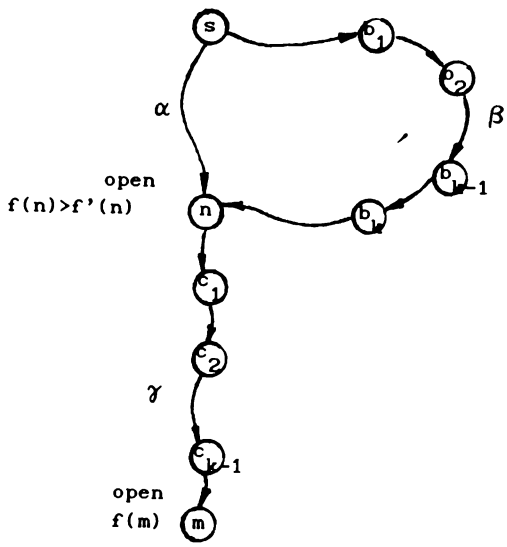

Figure 2.

At first the algorithm reaches node n along the path α. Since the algorithm has generated node m before path β is discovered, there must be a node b_1 on the path β, which is not expanded until node n is selected for expansion, that is $f(n) \leq f(b_1)$. After this path β and path γ are grown simultaneously until node m appears in the set OPEN. So there must be a node c_1 on path γ which is not expanded till the expansion of node b_1, that is $f(b_1) \leq f(c_1)$. Then it is clear, there must exist a node b_2 on the path β which is not expanded until node c_1 is expanded, i.e. $f(c_1) \leq f(b_2)$.

Continuing this deduction we have got

$$f(b_{k-1}) \leq f(c_{k-1}) \qquad \text{where } b_{k-1}, b_k \in \beta, \quad c_{k-1} \in \gamma, \quad k \in \mathbb{N},$$
$$f(c_{k-1}) \leq f(b_k),$$
$$f(b_k) \leq f(m).$$

Summarizing these inequalities we get $f(n) \leq f(m)$. According to our supposition $f(m) \leq f'(n)$. This inequality, however, contradicts our condition that the evaluation function is decreasing.

Now we are going to show if the spanning tree having lost optimality is rebuilt by the HNR-version with decreasing evaluation function after a couple of expansions.

Let S be the sequence of the values of function f for the nodes in the order in which they are expanded by the algorithm (Figure 3). It is evident that the first value is $f(s)$. Let us take now a subsequence $F^1, F^2, \ldots$ of S constructed as follows: F^1 is the first value and F^{i+1} is the first element of S following F^i such that $F^{i+1} \geq F^i$. Therefore $F^1, F^2, \ldots$ is a monotone nondecreasing sequence. We call the values of this subsequence the *limit-values* (F^i, $i \in \mathbb{N}$). The nodes belonging to these values are denoted as *limit-nodes* (n_i). Each node may be limit-node at most once, but just for its first expansion. We will call the part of expansions between node n_i and node n_{i+1} the i-th *ditch*.

Theorem 2. *The spanning tree is always optimal over the search graph whenever a limit-node is selected for expansion by the HNR-version with decreasing function.*

Proof. It will be sufficient to show that there are no closed nodes in the set OPEN at the expansion of any limit-node. We will prove it by induction on the number of limit-nodes. It holds on the first limit-node. Suppose there are no closed nodes at the expansion of node n_i. Then we can show that there are no closed nodes in the set OPEN at the expansion of node n_{i+1}. Let m be a closed node which is put back into the set OPEN inside the i-th ditch. It is obvious that $f(m)$ is not greater than the earlier value of function f of node

m, since we have a decreasing function. But this value must be less than F^i. Therefore $f(m) < F^i$, that is node m is expanded inside the i-th ditch.

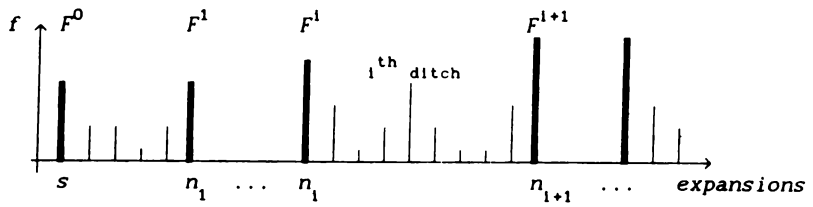

Figure 3.

Now we can introduce the same algorithm classes (algorithm A, algorithm A^*) embedded into the HNR-versions in the same way as in the case of the N-versions. Their definitions and main properties are identical.

Martelli [6] has shown that the HNR-version of the A^* algorithm requires $O(2^k)$ expansions when the search is performed on a graph with k nodes. We have to remark that some of them are not really expansions because if a closed node is being expanded, we know its successors.

4. Comparing the N-version and the HNR-version

Now we compare the N-version with the HNR-version of algorithm A^*. These algorithms can find an optimal solution if there is any solution. We remark that the following results hold on algorithm A, too. We assume that there is always a solution.

Theorem 3. *Both versions of algorithm A^* expand the same limit-nodes with the same limit-values, and these expansions record the same search graphs, the same spanning trees, the same open sets.*

Proof. We will prove it by induction on the number of limit-nodes. Before expansion of the start node both algorithms know just the start node. Suppose that the predicate holds at the expansion of limit-node n_i, that is both versions

select node n_i for expansion with the same limit-value F^i, record the same open set $OPEN^i$, the same closed set $CLOSED^i$, the same search graph G^i and the same cost-values g^i for each node ($i \in \mathbb{N}$). It will be sufficient to show that in the i-th ditch the set of nodes expanded by the HNR-version, which do not belong to the closed node at the expansion of the i-th limit-node, is equivalent to the set of nodes expanded by the N-version.

The following set describes the nodes expanded by the HNR-version in the i-th ditch:

$$HNR^i \stackrel{def}{=}$$

$$\stackrel{def}{=} \{n_i\} \cup \{m \in N \mid (1)\ \exists \alpha \in \{n_i \to m\} \text{ where } \forall n \in \alpha \setminus \{m\} : n \in HNR^i,$$
$$(2)\ g^i(n_i) + k^\alpha(n_i, m) < g^i(m) \quad \text{if} \quad m \in G^i,$$
$$(3)\ g^i(n_i) + k^\alpha(n_i, m) + h(m) < F^i\}.$$

The following set contains the nodes expanded by the N-version in the i-th ditch:

$$N^i \stackrel{def}{=} \{n_i\} \cup \{m \in N \setminus CLOSED^i \mid$$

$$\mid (1)\ \exists \alpha \in \{n_i \to m\} \text{ where } \forall n \in \alpha \setminus \{m\} : n \in N^i \cup CLOSED^i,$$
$$(2)\ g^i(n_i) + k^\alpha(n_i, m) < g^i(m) \quad \text{if} \quad m \in G^i,$$
$$(3)\ g^i(n_i) + k^\alpha(n_i, m) + h(m) < F^i\}.$$

It is easy to see by induction on the length of path α in the definitions above that

$$N^i = HNR^i \setminus CLOSED^i.$$

We know that both versions of algorithm A^* find an optimal solution. It is clear that their last limit-nodes are just the goal nodes found by them. Therefore both algorithms find the same goal node. Let us examine their complexity. Both algorithms generate the same nodes, i.e. both of them require the same amount of memory. But what about the number of expansions (about the running time)?

Now we will prove that the nodes expanded by the HNR-version in a ditch and the nodes expanded and corrected by the N-version in the same ditch are identical.

The next set describes the closed nodes corrected by the N-version after any expanded node n.

$$C^n \stackrel{def}{=} \{m \in CLOSED \mid (1)\ \exists \alpha \in \{n \to m\} \text{ where } \forall k \in \alpha \setminus \{m\} :\ k \in C^n,$$
$$(2)\ g^n(m) > g^n(n) + k^\alpha(n, m)\},$$

where $g^n(m)$ and $g^n(n)$ are the cost values of node n and node m at the expansion of node n, $CLOSED$ denotes the set of closed nodes.

Lemma. *Let n be a node expanded by the N-version in the i-th ditch. Then*

$$\forall m \in C^n : \quad g^n(n) + k^\alpha(n,m) + h(m) < F^i.$$

Proof. By the definition of set C^n

$$g^n(n) + k^\alpha(n,m) + h(m) < g^n(m) + h(m) \leq f(m) \leq F^i,$$

where $f(m)$ is the value of evaluation function at the expansion of node m.

Theorem 4. *Let HNR^i be the set of nodes expanded by the HNR-version of algorithm A^* in the i-th ditch ($i \in \mathbb{N}$). Let N^i be the set of nodes expanded by the N-version of algorithm A^* in the same ditch. Then*

$$HNR^i = N^i \cup \left(\bigcup_{i \in N^i} C^n \right).$$

Proof. At first we will show that

$$HNR^i \subseteq N^i \cup \left(\bigcup_{i \in N^i} C^n \right).$$

By Theorem 3 it will be sufficient to show that

$$HNR^i \cap CLOSED^i \subseteq \bigcup_{i \in N^i} C^n.$$

Let m be a node in $HNR^i \cap CLOSED^i$. There must be a path from node n_i to node m inside the set HNR^i. Let us select that not closed node at the expansion of limit-node n_i which is the closest one to m on this path, let us denote it by n (if there are several such paths then we take the firstly expanded by the HNR-version in the i-th ditch one). Obviously this node n belongs to the set N^i.

α denotes the cheapest path from n_i to m via the node n, that is

$$g^n(n) = g^i(n) + k^\alpha(n_i, n).$$

Denoting by β the part of α between n and m, it is obvious from the choice of node n that

$$k \in CLOSED^i \quad \text{and} \quad g^n(k) = g^i(k), \qquad \forall k \in \beta \setminus \{n\}.$$

Because of $\alpha \in HNR^i$ we have

$$g^i(k) > g^i(n_i) + k^\alpha(n_i, k), \qquad \forall k \in \alpha.$$

Summarizing

$$g^n(k) = g^i(k) > g^i(n_i) + k^\alpha(n_i, k) = g^i(n_i) + k^\alpha(n_i, n) + k^\alpha(n, k) =$$
$$= g^n(n) + k^\beta(n, k), \qquad \forall k \in \beta \setminus \{n\}.$$

Now we found a path (β) from node n to node m, where

$$\forall k \in \beta \setminus \{n\} : \quad k \in C^n, \quad \text{and}$$

$$g^n(m) > g^n(n) + k^\beta(n, m).$$

Therefore we have $m \in C^n$, since the node m had been a closed node at the expansion of n.

On the other hand it is enough to show that

$$C^n \subseteq HNR^i, \qquad \forall n \in N^i$$

($N^i \subseteq HNR^i$ follows from the result of Theorem 3).

Let β denote the cheapest path from node n_i to node n in the set HNR^i, that is $g^n(n) = g^i(n_i) + k^\beta(n_i, n)$. This path must exist since the node n is in the set HNR^i. Let m be any node in the set C^n. Let us prove that it belongs to the set HNR^i making an induction argument on the length of path γ from node n to node m, which exists according to the definition of set C^n.

If the length of γ is 1, i.e. $k^\gamma(n, m) = c(n, m)$, then

(1) it is clear that there exists a path α connecting the path β with γ ($\alpha = \beta \circ \gamma$), where $\forall n \in \alpha \setminus \{m\} : \; n \in HNR^i$;

(2) since $m \in C^n$

$$g^i(n_i) + k^\alpha(n_i, m) = g^i(n_i) + k^\beta(n_i, n) + c(n, m),$$
$$= g^n(n) + c(n, m) < g^n(m) \leq g^i(m);$$

(3) by the lemma we have

$$g^i(n_i) + k^\alpha(n_i, m) + h(m) = g^n(n) + c(n, m) + h(m) < F^i.$$

Therefore $m \in HNR^i$.

Suppose that the set HNR^i includes the node m if the length of path from n to m in C^n is not greater than any $l \in \mathbb{N}$. Let γ be a path starting from node n of length $l+1$. Denote the last node of this path by m. It is clear that all nodes except m on the path γ belong to the set HNR^i by the induction hypothesis.

(1) It is obvious that there exists a path α connecting the path β with the path γ ($\alpha = \beta \circ \gamma$), where $\forall n \in \alpha \setminus \{m\} : n \in HNR^i$.

(2) On the other hand

$$g^i(n_i) + k^\alpha(n_i, m) = g^i(n_i) + k^\beta(n_i, n) + k^\gamma(n, m),$$
$$= g^n(n) + k^\gamma(n, m) < g^n(m) \leq g^i(m).$$

(3) By the lemma we have

$$g^i(n_i) + k^\alpha(n_i, m) + h(m) = g^n(n) + k^\gamma(n, m) + h(m) < F^i.$$

This means that $m \in HNR^i$.

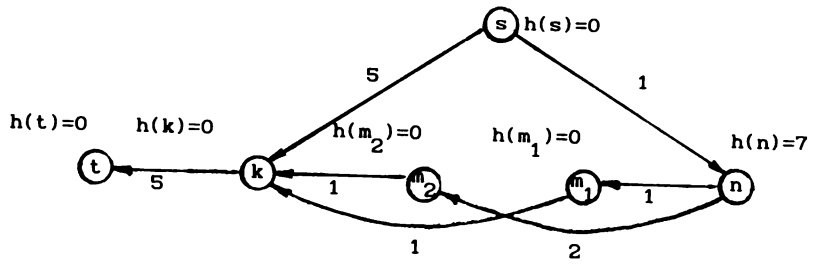

Figure 4.

The only difference between the HNR-version and the N-version of algorithm A^* is that how many times they expand or correct one node inside the same ditch. The N-version is not better than the special HNR-version of algorithm A^* (this is Martelli's algorithm B), which expands every node at most once in the same ditch. However we can give an example (Figure 4) where the algorithm B expands a less number of nodes (6) than the nodes (7) expanded and corrected by the N-version. (We can get a more strict estimate on the number of expansions of algorithm B. Since it is not greater than the number of expansions of the N-version, we can estimate it by $\frac{1}{2}k^2 + O(k)$.)

The last argument to decide which version is better is the complexity of their implementation. It is obvious that the HNR-version is simpler to be implemented. Thus the graphsearch version that we suggest to use is the HNR-version of the graphsearch algorithm.

6. Notations

$R = (N, A)$	- R is a directed graph, N is the set of nodes, A is the set of arcs;
s, T	- s is the start node and T is the set of the goal nodes;
$(n, m) \in A$	- directed arc from the node m to the node n;
$c(n, m)$	- cost of the directed arc from node n to node m;
$(n_0, n_1, \ldots, n_l)$	- the directed path from node n_0 to node n_l $(n_{l-1}, n_l) \in A$;
$n \rightarrow m$	- a directed path from node n to node m;
$\{n \rightarrow m\}$	- the set of directed paths from node n to node m;
$k^{\alpha}(n, m)$	- the cost of directed path α from node n to node m;
$g^*(n)$	- the cost of optimal path from the start node to node n;
G	- the search graph recorded by graphsearching;
Γ	- the operator of expansion;
$g(n)$	- the estimate of $g^*(n)$ by the algorithm;
$p(n)$	- the current parents of node n in the set G;
$f(n)$	- the estimate of cost of the path from the startnode to goal nodes containing the node n;
$OPEN, CLOSED$	- the sets of open nodes and closed nodes;
F^i, n_i	- the i-th limit-value and limit-node;
i-th ditch	- the part of extension, where the values of function f for the expanded nodes are less than F^i;
HNR^i, N^i	- the sets of nodes in the i-th ditch of the versions;
$OPEN^i, CLOSED^i$	- the set of open nodes, the set of closed nodes;
$g^i(n), G^i$	- the value of function g for any node and the search graph at the expansion of node n_i;
$C^n, g^n(m)$	- the set of closed nodes which are corrected by the N-version after the expansion of node n and the cost function value of node m at the same time.

References

[1] **Barr A. and Felgenbaum E.A.,** *The handbook of artificial intelligence I.,* HeurisTECH Press, Stanford, 1982.

[2] **Fekete I., Gregorics T. and Varga L.Zs.,** Corrections to graph-searching algorithms, *Proc.of "Fourth Conference of Program Designers", Budapest, June 1-3, 1988,* ed. A.Iványi, ELTE, Budapest, 1988, 25-30.

[3] **Gregorics T.,** Another introduction to consistent algorithms, *Proc.of "First Seminar on Artificial Intelligence", Visegrád, January 23-24, 1989,* eds. I.Fekete and S.Nagy, ELTE, Budapest, 1989, 137-144.

[4] **Gregorics T.,** Which graphsearch algorithm is better?, *Proc.of "Second Conference on Artificial Intelligence" vol. 2, Budapest, January 23-25, 1991,* eds. I.Fekete and P.Koch, ELTE, Budapest, 1991, 275-284.

[5] **Hart P., Nilsson N.J. and Raphael B.,** A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. System Science and Cybernetics,* **4** (1968), 100-107.

[6] **Martelli A.,** On the complexity of admissible search algorithms, *Artificial Intelligence,* **8** (1) (1977), 1-13.

[7] **Mérő L.,** A heuristic search algorithm with modifiable estimate, *Artificial Intelligence,* **23** (1) (1984), 13-27.

[8] **Nilsson N.J.,** *Principles of artificial intelligence,* Springer, 1982.

[9] **Pearl J.,** *Heuristics,* Addison-Wesley, Reading, Mass., 1984.

[10] **Rich E.,** *Artificial intelligence,* MacGraw-Hill Book Comp., 1983.

(Received July 1, 1992)

T. Gregorics
Department of General Computer Science
Eötvös Loránd University
VIII. Múzeum krt. 6-8.
H-1088 Budapest, Hungary