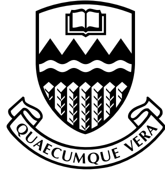

University of Alberta

Coarse-to-Fine Search Techniques

by

Ken Anderson, Nathan Sturtevant, Robert Holte, and Jonathan Schaeffer

Technical Report TR 08-05
April 2008

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Coarse-to-Fine Search Techniques

Ken Anderson, Nathan Sturtevant, Robert Holte, and Jonathan Schaeffer

April 5, 2008

Abstract

The following report describes some techniques that generalize and attempt to improve upon the Coarse-to-Fine Dynamic Programming (CFDP) algorithm developed by Christopher Raphael. CFDP uses a hierarchy of abstraction graphs, iteratively searches to find optimal paths, and refines those paths until a base level is found. Our most successful approach introduces a cached heuristic that reduces the effort spent re-searching in a graph.

1 Brief Introduction and Problem Domain

The generic domain we are interested in is single-agent search. Single-agent search encompasses a variety of problems such as pathfinding, DNA sequence alignment, and planning. Generally stated, single-agent search can be used to solve problems structured as finding a minimal-cost path from a start node to a goal node in a weighted, directed graph.

The domain in which we tested our algorithms was the pathfinding domain. We used 8-connected grid worlds where the undirected edges have cost one. This domain was used for two reasons: it made our techniques easier to visualize for the purpose of checking validity, and a pre-existing framework, Nathan Sturtevant's Hierarchical Open Graph (HOG), facilitated rapid development.

2 Related Work

2.1 CFDP

Christopher Raphael's Coarse-to-Fine Dynamic Programming technique can be used to find optimal paths on a trellis graph [4]. In our work, an optimal path is a minimal-cost path. Christopher Raphael specifically used trellis graphs, but we notice that this technique can be extended to general weighted, directed graphs.

Given a graph consisting of *base-level* nodes and edges, nodes can be grouped together to form abstract nodes. We retain the graph edges between (abstract) nodes, but make sure that each edge cost is not an overestimate of any of the base-level edges. The resulting abstract graph is smaller than the original graph and ideally takes less time to search through. We construct a hierarchy of progressively coarser-grained abstract graphs where the most abstract graph consists of a single node.

CFDP uses a search technique to find an minimal-cost path through an abstract graph, G. G consists of nodes from various abstraction levels. Let us say that the search technique used is Dijkstra's Algorithm because we are given no heuristic at the start. The path that is found, however, consists of abstract nodes. What we really are after is a path at the base level. Therefore, we take all nodes on the minimal-cost path and *refine* them to a lower level. Refining a node removes the node from the graph and replaces the

node with its corresponding *pre-image* nodes from the finer-grained level of abstraction. The edges are also replaced correspondingly.

CFDP then iteratively repeats the search and refinement steps until the minimal-path solution consists completely of base-level nodes.

2.1.1 Properties

This search technique requires that an abstract graph, G, be held in memory. Recall that at any time, this graph can consist of abstract nodes from arbitrary levels of abstraction. But since every node at the base level maps to exactly one node in G, our memory requirements for holding G can be no more than for holding the original graph at the base level. By keeping some nodes in G at an abstract level, it is possible achieve memory savings.

Single-agent search problems are generally formulated as finding a minimal-cost path through a weighted, directed graph from a start node to a goal node. This formulation implies that the graph (or problem space) be of finite size. However, CFDP can be used to solve problems in continuous problem spaces to arbitrary accuracy.

One typical way to solve continuous problem spaces is to discretize or dissect the state space into individual connected nodes. The problem is then solved with traditional graph-based search methods. If a hierarchical layering of abstractions can be patterned upon a continuous state space, we can successively solve the more difficult problems until we reach some arbitrary accuracy.

2.2 Hierarchical A*

Holte's Hierarchical A* search similarly uses a hierarchy of abstract graphs to speed search. At the base level we search from a start node to a goal node using A*. However, we do not have a pre-existing heuristic for nodes at the base level. To get a heuristic for node n, we calculate its image n' in the coarser-grained abstraction level. Solving this smaller problem (finding the exact distance from n' to the abstract goal g' in the abstract graph) gives us an lower bound on the exact distance from s to g in the base-level graph. Now, in order to efficiently find the distance from s' to g', we need a heuristic. So we recursively retrieve a heuristic from the next coarser level of abstraction.

In order to save repeated search effort, Holte introduced three caching techniques. These techniques exploit the fact that the goal never changes

- **h*-caching**: After every search in the abstract graph, the exact distance to the goal is known for every node on the optimal path. This distance is cached and used as a heuristic for the next search. h*-caching produces inconsistencies in the heuristic but these inconsistencies are proved not to affect search.
- **optimal-path caching**: When expanding a node where we know the exact heuristic value, we add the goal plus the heuristic value to the open list.
- **P-g caching**: After every search in the abstract graph, we cache the value $f_{sol} - g$ for all expanded nodes, where f_{sol} is the solution length for that abstract problem and g is the distance from the start to the node. P-g-caching only works given a consistent heuristic; the g values of nodes on the closed list correspond to the minimal-cost distance to the start node. Additionally, P-g-caching subsumes h*-caching.

3 Novel Algorithms

3.1 Coarse-to-Fine Dynamic Programming with caching

Our most successful technique merged the ideas of CFDP with the caching developed for hierarchical search. The approach is exactly the same as CFDP; a graph G consists of abstract nodes (of arbitrary

abstraction level) connected by directed weighted edges. Search finds an optimal solution. Then all nodes on this path are refined to the finer-grained level of abstraction. This repeats until our optimal path is found.

Since the whole graph is in memory already, we assign a heuristic value to every node in the graph. The heuristic for every node at the beginning is 0. After the first search finishes, we use P-g-caching to assign a heuristic value to every expanded node. During the refinement step, every refined node passes its heuristic to all of its corresponding pre-image nodes.

The heuristic retains admissibility on the refinement step as long as the refinement step guarantees that the optimal path cannot decrease in length. In practice the heuristic remains consistent, but we have not yet proved this property.

This technique was tested against normal CFDP in Section 4.1.

3.2 CFDP with caching while switching directions

A second algorithm was developed with the aim of improving the heuristic to save search effort on each iteration. If A* search uses a consistent heuristic, when a node n is expanded, the g-value is correct. That is, we know the exact distance from n to the start. If we store (cache) these g-values, we can use them to guide search on the next iteration.

In order to use these values efficiently, we switch search directions (search from the goal to the start) and use the stored g-values as a heuristic for the search in the opposite direction. After the forward search we cache the g-values and refine the necessary nodes. Each new node created from the refinement step retains the g-value of its image. After the refinement step, the g-values from the previous search are no longer correct. However, the g-values still do not over-estimate the distance to the start, and so remain an admissible heuristic for the reverse search.

The original conception of this idea was to continually switch directions using the stored g-values as a heuristic for the opposite direction. However, there is a theoretical flaw with this approach. Let us follow an example to see why.

On the first search iteration (in the forward direction), there is no stored heuristic. Therefore, the heuristic is zero for every node and is consistent. We search forward and store the g-values of all expanded nodes for use as a heuristic in the backwards direction. Because the heuristic is consistent, the g-values are guaranteed correct.

On the second search iteration we search in the backwards direction using the stored heuristic. On the first iteration we only stored the g-values of expanded nodes. So all unexpanded nodes from the previous search have a heuristic of zero. Therefore our heuristic is potentially inconsistent. A* using an inconsistent heuristic will still give an optimal path *if* we allow for node re-expansion (re-opening nodes already on the closed list). However, because our heuristic is inconsistent, we are no longer guaranteed that the g-values of all expanded nodes are correct. Because of this, using the cached g-values as a heuristic on the third iteration can result in an inadmissible heuristic and a suboptimal solution.

Our analysis did not find this theoretic flaw until after implementation, when suboptimal solutions were being found. Therefore, there are no results using this technique. However, some possible avenues for further research are listed in Section 5.1.

3.3 Single-node refinement with heuristic propagation

This technique merges search, heuristic evaluation, and refinement into one algorithm. Its motivation lies in the idea that iteratively re-searching in order to refine nodes can be expensive. Therefore we developed an algorithm that does not search, but rather selects nodes to refine based on heuristic estimates, then propagates the heuristic changes through necessary parts of the graph.

Each node n in the graph has a g and an h value, where $g(n)$ is an under-estimate of the shortest distance from the start to n and $h(n)$ is an under-estimate of the shortest distance from n to the goal. The h value of a node n can be updated by examining h value of each of its neighbors plus the cost to n. The

g value can be updated similarly. This is heuristic update method is used in Korf's Learning Real-Time A* [2].

Each node has an implicit f value, where $f(n) = g(n) + h(n)$. This f value is an under-estimate of the cost to get from the start to the goal through the node n. Our algorithm examines nodes in a prioritized order (based mostly on the f-cost) and chooses to either update the node's g and h values or refine the node.

Appendix C shows the full algorithm. This algorithm was implemented, but the performance was so poor that thorough testing was judged to be unnecessary. Results are listed in Section 4.2.

4 Results

4.1 CFDP vs CFDP with heuristic caching

To test this domain, random mazes of variable size were created. The edges were undirected with weight one. An example maze is listed in Appendix A.

Table 1 compares the average number of expanded nodes using CFDP vs CFDP with caching. The first column lists the dimensions of the square-shaped maze. The second column shows the total number of nodes in the graph. The third and fourth columns represent the average number of nodes expanded when using CFDP and CFDP with caching over 100 trials. The final column shows the percent reduction in number of nodes expanded when caching is used with CFDP.

The number of nodes in the graph can be used as a worst-case scenario for the number of nodes expanded using Dijkstra's algorithm (no heuristic). Note that CFDP and CFDP with caching on average expands more nodes than the total number of nodes in the graph. This indicates that Dijkstra's algorithm will, on average, outperform CFDP in this domain.

In all cases, caching improves the performance of CFDP, reducing the average number of nodes expanded. This reduction improves with problem size. Further testing with larger state spaces would be worthwhile to examine.

x and y dimensions	Number of nodes in graph	CFDP	CFDP with caching	% reduction
15	117	156	130	16.62%
30	517	1,353	1,030	23.84%
45	1,121	3,294	2,427	26.32%
60	2,080	9,436	6,751	28.46%
75	3,182	18,039	12,402	31.25%
80	3,668	21,392	14,699	31.29%
95	5,086	31,671	21,438	32.31%
110	6,960	47,618	31,504	33.84%
125	8,903	61,897	40,982	33.79%
140	11,279	108,716	71,351	34.37%

Table 1: Average number of nodes expanded using CFDP vs CFDP with heuristic caching.

4.2 Single-node refinement with heuristic propagation

In practice, the refining of a single node results in the update of many of the nodes in the graph. Each update is very small, so the g and h values of nodes are incrementally increased. This propagation step

ends up being extremely slow and if g and h are real numbers, each incremental update can be minuscule, exacerbating the slow-propagation problem.

5 Future Work

5.1 CFDP with caching while switching directions

Using an initial search in the reverse direction for the purpose of creating a heuristic for the forward search gives impressive performance benefits [5]. But as we have seen in Section 3.2, using this technique does not easily lend itself for use in our coarse-to-fine algorithm. There are some possible adaptations that we have not tried but will mention here.

5.1.1 Enforced Heuristic Consistency

The first approach is to try to force the heuristic to be consistent. One naive way to do this would be to continue the A* search until all nodes have been expanded, caching the g-values along the way. However, this technique expands all nodes at every iteration so having a heuristic and using A* saves us no search effort at all (except on the last iteration).

Another similar approach to enforce heuristic consistency is to propagate heuristic values after each search iteration using a pathmax-based idea [3] This idea would assign a consistent heuristic value to a node n on the open list based on its closed neighbors, $n_{1..k}$:

$$h(n) = max_{n_i \in n_{1..k}} \{h(n_i) - c(n, n_i)\}$$

Re-organize the open list to assign higher priority to higher h-values. To propagate heuristics, 'expand' node n and add/update its neighbors to the open list with their new h values if they aren't already on the closed list. The resulting heuristic will hopefully be consistent. This technique potentially touches fewer nodes than the entire graph because the propagation of values stops when $h = 0$.

5.1.2 Inconsistent Heuristics

The second technique allows heuristic inconsistencies, but deals with them carefully. As we saw, if our forward search uses a consistent heuristic, our backward search can use the cached g-values as an admissible, but inconsistent, heuristic. For a number of subsequent iterations, the search can occur in the reverse direction using the heuristic values found in the first iteration. At some point, we decide to search in the forward direction again (to improve the heuristic). As long as the heuristic is consistent, we can cache and re-use the g-values again for backward searches. A simple approach is to make the heuristic zero everywhere.

An slight modification to this approach updates the heuristic values after each iteration in the reverse direction. As you may recall, one of the caching techniques for Hierarchical A* is to cache the h-values for nodes on the optimal path, h*-caching [1]. Searching in the reverse direction with A* using an inconsistent heuristic, we can still find optimal paths (if we re-open closed nodes). The g-values for closed nodes are not guaranteed correct; however, the g-values for nodes on the optimal path are correct and we can still use h*-caching. This may introduce inconsistencies in the heuristic, but the heuristic is most likely inconsistent already. Note that we cannot use P-g-caching because the g-values may be incorrect. Nor can we use optimal-path caching because the graph is always changing.

5.2 Combining CFDP with Hierarchical Search

The previous approaches have all concentrated on complete search techniques that refine nodes in an abstract graph until we find an optimal path. Let us call this class of algorithms *coarse-to-fine*. Alterna-

tively, hierarchical search starts at the base level, and queries coarser abstraction levels for heuristics in a *fine-to-coarse* approach.

Coarse-to-fine algorithms store an abstract graph in memory wherein each node in the base graph always maps to a node in the abstract graph. Because of this property, we can combine the two approaches as follows. A coarse-to-fine algorithm can refine the abstraction graph until a memory limit (or node limit) is reached. At this point we can search using A* at the base level. Our heuristic can be acquired by getting the heuristic of the corresponding node in the abstract graph.

One implementation detail to be addressed is how to efficiently find the abstract node that corresponds to a given base-level node.

6 Conclusions

In this document we have proposed a number of new search techniques and we have implemented two of them. One technique in particular, albeit the simplest one (CFDP with caching), has shown promise in improving upon CFDP on our chosen domain, pathfinding.

Caching is fairly simple to implement as an augmentation of CFDP. The majority of our development time was spent implementing CFDP for use with our domain; the additional effort to implement caching was minimal. Naturally, care should be taken to ensure that caching does not result in suboptimal solutions.

6.1 Appropriate Domains

Adding caching to CFDP may have considerable beneficial results in domains where CFDP performs well, that is to say in domains where the refinement process quickly narrows down on a particular solution.

We believe that pathfinding may not be an appropriate domain for Coarse-to-Fine Dynamic Programming, but it is a domain we have particular interest in. When Christopher Raphael originally presented CFDP [4], he demonstrated the approach on domains where refining the graph changed the edge weights minimally. Thus (1) the total number of iterations is low and (2) the majority of effort is spent refining nodes near the optimal path. In other words, the refinement process quickly narrows down on a particular solution. In pathfinding, this is not necessarily true.

Adding caching to CFDP does not change the number of iterations or the number of refined nodes, but it will decrease the search effort required for each iteration. In a domain where the refinement effort is concentrated near the actual optimal path, using cached heuristics should produce more cutoffs in the search tree, thereby enabling the optimal path to be found more quickly.

6.2 Recommendation

Adding caching to an existing implementation of CFDP is an easy add-on and the extra memory requirements are low. We have demonstrated beneficial results for pathfinding in this document and we believe that the application of this technique to more suitable domains will improve upon the state-of-the-art CFDP solutions.

References

[1] Robert C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI*, pages 530–535, 1996.

[2] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[3] Laszlo Mero. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1):13–27, 1984.

[4] Christopher Raphael. Coarse-to-fine dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(12):1379–1390, 2001.

[5] David Silver. Cooperative pathfinding. In *Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 1, 2005.

[6] Nathan Sturtevant. Hierarchical open graph (hog). http://www.cs.ualberta.ca/ nathanst/hog.html.

A Sample maps used for experiments

The domain used in our tests was the pathfinding domain. The graphical world has undirected edges of weight one. The world is eight-connected, where each node can have up to eight possible edges. Maps are randomly generated.

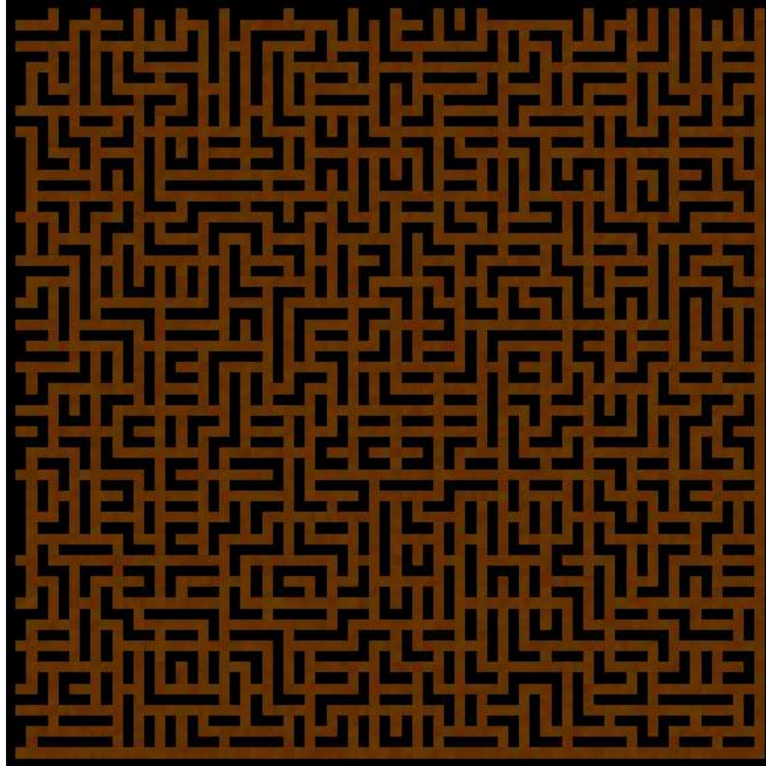

Figure 1: The 70 by 70 maze used for experiments.

B Overview of implementation in HOG2

The following three algorithms are implemented in the HOG2 framework which can be found at http://code.google.com/p/hog Please contact Nathan Sturtevant for access and information.

The CFDP algorithm is implemented under the name ”IRDijkstra” (for Iterative-Refinement Dijkstra). This algorithm uses Dijkstra's algorithm to find optimal paths, then refines all optimal paths. This process is repeated until all solution nodes are at the base level.

The CFDP caching algorithm is under the name ”IRAstar” (for Iterative-Refinement A*). This algorithm uses an A* search, but each graph node has a heuristic value. At the completion of each A* search we update the heuristic values of every node on the closed list. There are three possible caching strategies: no caching, P-g caching, or h*-caching (one is enabled by passing in a parameter). Of the three, P-g caching consistently produces the best results; in fact, it dominates the other two strategies. Consequently, our experiments use P-g caching.

The single-node refinement algorithm with heuristic propagation is under the name ”CFOptimalRefinement” (for Coarse-to-Fine Optimal Refinement). This algorithm is implemented as described in Appendix C. Our initial tests showed the algorithm to be working correctly, but recent review has indicated that it is not properly converging on a solution. It is unsure at this point if this is an algorithmic or implementation error.

All algorithms were called from one file: ”Sample.cpp”. The actual algorithm call must be changed manually. This program takes the following parameters:

- *-map filename* which specifies a map
- *-seed integer* sets the seed for the randomized number generator
- *-size integer* creates a maze in the shape of a square with the same x and y dimensions
- *-batch integer* runs the specified number of tests in a row.

To reproduce the results listed in this paper for the 75 by 75 maze, use the following line:

```
./hog2/bin/debug/sample -seed 5 -size 75 -batch 100
```

C Single-Node Heuristic-Propagation Algorithm

The following is documentation of the Single-Node Heuristic-Propagation Algorithm described previously.

C.1 Symbols

- S – problem space
- $|S|$ – size of problem space
- ϕ – hierarchical set of abstract graphs
- ϕ_x – abstract graph, which covers the entire space S
  - higher x = coarser-grained abstractions
  - ϕ_M is the coarsest-grained abstraction and consists of only one node
  - ϕ_0 is the finest-grained abstraction and consists of the original (base) graph
- $start$, $goal$ – start and goal node in the original graph (ϕ_0)
- n,m – arbitrary nodes

C.2 Terminology

- *subnodes* of $n \in \phi_x$ – the set of nodes in abstraction ϕ_{x-1} that directly map to n. This is also the *pre-image* of n.
- *supernode* of $n \in \phi_x$ – the node in abstraction ϕ_{x+1} that n maps to. This is also known as the *image* of n.
- *successors* – nodes connected by an out-edge
- *predecessors* – nodes connected by an in-edge

C.3 Preconditions

- all edges have non-negative edge weights
- an edge that doesn't exist can be assumed to have weight ∞. This algorithm requires a lower bound on the cost of an edge between two nodes. This can be any positive number less than or equal to the actual cost of the edge between the nodes. If no edge exists, then the lower bound can still be represented by any positive number less than or equal to ∞.
- abstractions must be hierarchical – every node in every abstraction, except ϕ_M, must have, at most, one supernode.

C.4 Data Structures

The relavant data structures are:

- $\phi = \phi_{0..M} \equiv$ Hierarchical levels of Abstractions. Each level covers the entire state space S and contains abstract nodes and edges between abstract nodes (which is a lower bound on the minimum cost edge between the nodes in the real space). Can be explicit or implicit. ϕ_0 is the base level and ϕ_M is the coarsest-grained level.
- Explicit abstract graph G. G is a set of abstract nodes that partitions the state space S; every node in S maps to a corresponding node in G. The edges in G are a lower bound on the minimum cost between nodes in S.
- Every node $n \in G$ has an associated:

- abstraction level ϕ
- set of successor abstract nodes $\in G$
- set of predecessor abstract nodes $\in G$
- g – lower-bound on optimal distance from $start$ to n in the base level
- h – lower-bound on optimal distance from n to $goal$
- $optimalH$ flag stating whether the nodes h value is optimal ($true$/$false$)
- $list$ – is either $OPEN$ or $CLOSED$

- G is stored as a Prioritized Queue where nodes have priority based on the following criteria. Ties of the same f-value are broken by the $list$ criteria.
  1. f-value (low f = high priority)
  2. $list$ (OPEN = high priority, CLOSED and not at base level = medium priority, CLOSED and at base level = low priority)
  3. (optional) g/h values (either way)
  4. (optional) abstraction level (either way)
- $start$, $goal$ are the start and goal nodes in the base level.

C.5 Operations

The operations for the queue are

- Remove$(n, Queue)$ – removes node n from $Queue$ (random access)
- Remove$(Queue)$ – removes node with highest priority of $Queue$
- Add$(n, Queue)$ – adds n to prioritized queue (priority based on node's associated extra data)

Other operations:

- $\phi_x(n)$ – find n's supernode or subnodes at the specified abstraction level.
- Successors$(n, graph)$ – find n's successor nodes in $graph$ with the restriction that $graph$ must contain n.
- Predecessors$(n, graph)$ – find n's predecessor nodes in $graph$ with the restriction that $graph$ must contain n.
- $c(n, m, graph)$ – the cost of an edge from n to m in $graph$ ϕ_x. $graph$ must contain n and m.
- Connect(n, m, c, G) – add m to the successor list of n with cost c and add n to the predecessor list of m with cost c. Assume n and m are in G.
- Disconnect(n, m, G) – remove m from the successor list of n and remove n from the predecessor list of m. Assume n and m are in G.
- Reconstruct$(G, start, goal)$ – finds the sequence of operators or states of the optimal path.

C.6 Algorithms

Note about graph operations - if G is not listed, it is implied.

```
MAINALGORITHM(start, goal, φ)

     // Initialize Graph G to hold one node
1    n ← φ_M // φ_M has one node
2    n.h ← n.g ← 0
3    n.list ← OPEN
4    ADD(n, G)

     // Find optimal path
5    until start.optimalH = true
6         n ←REMOVE(G)
7         if n.list = OPEN then
8              UPDATE(n, G, start, goal)
9         else // n.list = CLOSED
10             REFINE(n, goal, G, φ)
11        endif
12   return RECONSTRUCT(G, start, goal)
```

Figure 2: Refinement algorithm to find optimal path.

```
UPDATE(n, G, start, goal)

// Update h
minH = ∞
13  forall s ∈SUCCESSORS(n, G)
14      h ← c(n, s) + s.h
15      if minH > h then
16          minH ← h
17  if minH ≠ n.h and n ≠ φ_{n.φ}(goal) then
18      n.h ← minH
19      forall p ∈ PREDECESSORS(n)
20          REMOVE(p, G)
21          p.list ← OPEN
22          ADD(p, G)

    // Update g
23  minG ← ∞
24  forall p ∈PREDECESSORS(n, G)
25      g ← p.g + c(p, n)
26      if minG > g then
27          minG ← g
28  if minG ≠ n.g and n ≠ φ_{n.φ}(start) then
29      n.g ← minG
30      forall s ∈ SUCCESSORS(n)
31          REMOVE(s, G)
32          s.list ← OPEN
33          ADD(s, G)

    // Update OptimalH flag
34  optH ← false
35  if n.φ = 0 and n.optimalH = false then
36      if n = goal
37          optH ← true
38      forall s ∈ SUCCESSORS(n, G)
39          if s.optimalH = true and c(n, s) + s.h = n.h then
40              optH ← true
41      if optH = true then
42          n.optimalH ← optH
43          forall p ∈ PREDECESSORS(n)
44              REMOVE(p, G)
45              p.list ← OPEN
46              ADD(p, G)

    // Close node
47  n.list ← CLOSED
48  ADD(n, G)
```

Figure 3: Update Method.

```
REFINE(n, goal, G, φ)
subnodes ← φ_{n.φ−1}(n)

    // Connect Subnodes of n to nodes in G
49  forall n_i ∈ subnodes
50  // Connect subnodes to themselves (internally)
51      forall m ∈ subnodes
52          if c(n_i, m, φ(n_i)) ≠ ∞ then
53              CONNECT (n_i, m, c(n_i, m, φ(n_i)))
54  // Connect subnodes to successors of n (externally)
55      forall s ∈ SUCCESSORS (n, G)
56          if s.φ < n_i.φ then
57              if c(n_i, φ_{n_i.φ}(s), n_i.φ) ≠ ∞ then
58                  CONNECT (n_i, s, c(n_i, φ_{n_i.φ}(s), n_i.φ))
59          else if s.φ = n_i.φ then
60              if c(n_i, s, φ_{n_i}) ≠ ∞ then
61                  CONNECT (n_i, s, c(n_i, s, n_i.φ))
62          else // s.φ > n_i.φ then
63              if c(φ_{s.φ}(n_i), s, φ_s) ≠ ∞ then
64                  CONNECT (n_i, s, c_G(n, s))
65          endif

    // similar for Predecessors ...  *

    // Put subnodes of n into G
66  forall n_i ∈ subnodes
67      n_i.g ← n.g
68      n_i.h ← n.h
69      n_i.list ← OPEN
70      n_i.optimalH ← false
71      ADD(n_i, G)

    // Disconnect n from graph G
72  forall s ∈ SUCCESSORS(n, G)
73      DISCONNECT(n, s, G)
74  forall p ∈ PREDECESSORS(n, G) *
75      DISCONNECT(p, n, G) *
```

Figure 4: Refine Method. *not necessary in undirected graph