Accelerated Pursuit Using Hierarchical Graphs

Sahar Idwan +, Dinesh P. Mehta ++
+ Computer science Department, Hashemite University, Zarqa, Jordan
++Mathematical and Computer Sciences Department, Colorado School of Mines, Golden, USA

Abstract: This paper improves on solutions to the PUG (Pursuit with Updates on Graph) problem by employing hierarchical rather than flat graphs. The flat graph is partitioned into fragments. If the pursuer and evader are in different fragments, the algorithm operates on the top-level graph and attempts to place the pursuer and evader in the same fragment with minimal updates. Once the pursuer and evader are in the same fragment, the algorithm reverts to the low-level flat graph. Our experimental results show that this results in a significant improvement in the number of updates and a modest improvement in run time without sacrificing the high-quality capture times obtained for the flat graph.

Keywords: Pursuer, Evader, Pursuer with Updates Graph (PUG), Hierarchical Graph.

1. Introduction

The Pursuit with Updates on a Graph (PUG) problem assumes that the following is provided [9]:

1) A graph \( G=(V,F) \) with length \( \text{length}(f) \) associated with each edge \( f \in F \).

2) A pursuer \( p \) and an evader \( e \) whose initial locations are \( l_{\text{init}}^p \) and \( l_{\text{init}}^e \), respectively. The location of an evader or pursuer refers either to a vertex or to a specific point on an edge. For example, suppose \( G \) has an edge \( (u,v) \) of length 5. Then, a location on the edge may be defined as “3 units from \( u \) on edge \( (u,v) \)”.

3) The evader is oblivious to the pursuer and is traveling along a pre-determined path at \( l_{\text{init}}^e \) and ending at \( l_{\text{final}}^e \).

4) The pursuer and evader travel with constant speeds \( s_p \) and \( s_e \), respectively.

The objective is to compute a path for the pursuer that captures the evader in minimum time. A capture is said to occur when the pursuer and evader are at the same location at the same instant of time. The algorithm used to compute this path is not permitted to use knowledge of the evader’s path, but is permitted to request updates of the evader’s location. These updates only provide the evader’s location and are communicated instantaneously to the pursuer. A secondary objective of the problem is to minimize the number of updates. We make the following additional assumptions: 1. The evader is not permitted to pause. 2. Neither the pursuer nor the evader is permitted to change directions on an edge. This is clearly not possible in a directed graph, but we are also explicitly forbidding it on undirected graphs. 3. Updates do not provide the direction of travel of the evader, only its location.
2. Hierarchical Graph Construction

2. a. Graph Partitioning

The original flat graph is partitioned using the hMetis software package that was developed for partitioning VLSI circuits [3,4,5]. Graph partitioning is NP-complete and consequently the VLSI design automation community has explored several approaches for practical graph partitioning. hMetis is considered to be the best VLSI partitioner at the current time. (It obtains high-quality solutions and has fast run times.) VLSI circuits are actually hypergraphs. A hypergraph is a generalized graph in which an "edge" can connect several vertices. In a traditional graph, each edge connects two vertices. hMetis permits the user to specify a parameter K, which represents the desired number of fragments. It partitions the graph into K fragments such that each fragment contains approximately the same number of vertices, while minimizing the number of edges that cross fragments. Figure 1 illustrates a graph that has been partitioned.

![Figure 1](image1.png)

Figure 1: Partitioning: the graph consists of 24 vertices. The dashed lines divide them into four fragments consisting of six vertices each.

There are other methods to partition the graph that depends on spatial proximity. One approach [11] is to sort all edges by the x coordinates of their source nodes and then apply a plane-sweep technique to sweep all x-sorted edges from left to right. The sweeping process stops periodically to sort the edges swept since the last storage by the y-coordinates of their origin nodes. This technique is applicable only if coordinates are available for the vertices of the graph, and usually yields partitions that are worse than those obtained by other partitioning techniques.

2. b. Hierarchical Graph Construction

Our strategy for constructing a hierarchical graph is similar to that used in [7,8] although there are some key differences:

1. We use hMetis to partition the graph whereas they use a spatial partitioning technique.
2. We partition vertices of the graph, whereas they partition edges into fragments.
3. We construct shortest paths between vertices and their border nodes on the fly, whereas they perform an all-pairs shortest path precomputation within each fragment.

In the following discussion, we will describe the construction of a two-level hierarchical graph. The hMetis algorithm partitions the vertices of a graph into fragments. The edges of the graph are of two types: local and border. A local edge is one whose vertices both belong to the same fragment, whereas a border edge is one whose vertices belong to different fragments. To build the hierarchical graph, we insert a border node at the half-way point of each border edge. Figure 2 illustrates these concepts.

![Figure 2](image2.png)

Figure 2. Border vertices a-h, denoted by filled boxes, are inserted in the middle of the corresponding border edges. We have \( \text{SourceSet}(F_1) = \{a,c\} \), \( \text{TargetSet}(F_1) = \{b,d\} \), \( \text{SourceSet}(F_2) = \{d,e,f\} \), \( \text{TargetSet}(F_2) = \{g\} \), \( \text{SourceSet}(F_3) = \{a,e,h\} \), \( \text{TargetSet}(F_3) = \{a,c\} \), \( \text{SourceSet}(F_4) = \{g,h\} \), \( \text{TargetSet}(F_4) = \{f\} \).

This border node is conceptually viewed as being on the common "border" of the two fragments corresponding to the endpoints of the border edge. Two fragments are said to be adjacent if they have at least one common border node. For example, in the figure, \( F_1 \) and \( F_2 \) are adjacent, but \( F_1 \) and \( F_4 \) are not. All of the border nodes in the graph are "moved up"
to the top level, where they form the vertices of the top-level graph (Figure 3). It remains to build the edges of the top-level graph. This is done as follows: Consider a fragment \( F \) of the graph. The border nodes belonging to \( F \) are of two types, those that belong to edges leaving \( F \) and those that belong to edges entering \( F \). (Recall that the original graph is directed.) We refer to the former as the SourceSet (because the source vertices of the corresponding border edges belong to \( F \)) and the latter as the TargetSet. Our objective is to add edges to the top-level graph so that if the shortest-path distance between a pair of border nodes in the original flat graph is some quantity \( x \), then there is a shortest-path of length \( x \) in the top-level graph. In other words, the top-level graph should leave shortest path distances between pairs of border vertices unchanged. We call this the SP Invariant. We discuss two approaches for adding edges to uphold this principle. Both approaches consist of adding a top-level edge to every pair of vertices \( (u,v) \) such that \( u \) is in SourceSet(\( F \)) and \( v \) is in TargetSet(\( F \)) for some fragment \( F \).

**[A1]** Compute the length of the shortest path from \( u \) to \( v \) in the original graph and add a top-level edge \((u,v)\) of the same length. **[A2]** Perform the same shortest-path computation as in [A1]. However, we only add edge \((u,v)\) if all of the edges in the shortest path belong to fragment \( F \).

**Theorem:** Both A1 and A2 result in a top-level graph that satisfies the SP Invariant.

**Proof:** Consider a pair of border vertices \( u \) and \( v \) in a top-level graph. We want to prove that these vertices satisfy the SP Invariant. If there is an edge \((u,v)\) in the top-level graph, then the SP Invariant is immediately satisfied for \( u \) and \( v \) because the length on this edge is based on the length of the shortest path between them (in both A1 and A2). Now, consider the case where there is no edge \((u,v)\) in the top-level graph. Suppose that there is a path from \( u \) to \( v \) in the original graph that is shorter than the one in the top-level graph. This shorter path OPT cannot be completely contained in a fragment as both A1 and A2 would then have used its length to establish an edge between \( u \) and \( v \). OPT must therefore traverse two or more fragments. OPT must enter a fragment through a border vertex in its TargetSet and exit the fragment through a border vertex in its SourceSet. But, from the principle of optimality, this portion of the path is a shortest path between the two border vertices and therefore should be represented by an edge of the same length in the top-level graph. Thus, any path OPT, can be broken down into a sequence of edges that are contained in the top-level graph.

An algorithm for building the hierarchical graph based on A2 is presented in Figure 4.

![Figure 3. A top-level graph for the graph partition and border vertices of Figures 1 and 2. The edges are added as per Approach A1. Recall that fragment \( F_i \) has TargetSet(\( F_i \))=\{b,d\} and SourceSet(\( F_i \))=\{a,c\}. The Cartesian product of these sets results in the edges \((b,a), (b,c), (d,a), \) and \((d,c)\). The length of each edge is the length of the shortest path between the two vertices in the original graph. Edges in other fragments are similarly computed.](image)

**3. Solving PUG using Hierarchical Graphs**

Having created a hierarchical graph, we are now ready to address the PUG problem. If the pursuer and the evader are in different fragments, we compute the shortest path between them. The shortest path is computed using the top-level graph if both the pursuer and evader are border vertices. Otherwise, edges have to be added to the top-level graph as follows. An edge is added from the pursuer's location to each vertex in its fragment's SourceSet. The length of this edge is the length of the shortest path between the pursuer's location and the border node in the flat graph.

Similarly, an edge is added from each vertex in the evader's fragment's TargetSet to the evader location. The shortest path between pursuer and evader is now computed in this graph. The pursuer is sent to the first top-level vertex on this path, where it requests an update. Figure 5 illustrates these concepts. We repeat this process until the pursuer and evader are in the same fragment. At this stage, the algorithm switches to the original graph, where one of our earlier PUG methods (DPAA,PPE,PPRSS,PPR) [9] is employed.
until the evader is captured. The algorithm is presented in Figure 6.

BuildHierarchicalGraph(G)
1. fragmentList = hMetis(G);
2. Foreach (fragment F in fragmentList) Do
3. TargetSet(F)=Border nodes whose targets are in F;
4. SourceSet(F)=Border nodes whose sources are in F;
5. Foreach u in TargetSet(F) Do
6. Foreach v in SourceSet(F) Do
7. spath=ShortestPath(G,u,v);
8. If (all edges of spath are in F)
9. length(u,v)=length(spath) + (length(edge(u)) + length(edge(v)))/2;

Figure 4. Algorithm Build Hierarchical Graph

Figure. 5 Illustration of algorithm in top-level graph. Vertices P and E denote the location of the pursuer and evader, respectively. Edges (P,c), (P,a) and (f,E) are added to the top-level graph. The lengths of these edges are the lengths of the shortest paths between the vertices in the original graph. The shortest path from P to E may now be computed using Dijkstra's algorithm. In the example, there are two paths: P-a-b-c-f-E and P-c-f-E, of which the latter is shorter. The pursuer is sent to the first vertex on this path; i.e., to vertex c.

4. Experimental Results
Our algorithms were developed in C++ and LEDA and our experiments were implemented on an Intel Pentium 4 processor with clock speed 3.2GHz running on Fedora core 2 system. In all experiments, we have the following:
1. The pursuer has speed 200.
2. The evader's speed is one of 60, 100, and 150.
3. The results contain three pieces of information:
   a. The Capture Time (the amount of time required for the pursuer to capture the evader.
   b. The number of updates (when appropriate).
   c. The run time of the algorithm in milliseconds.
4. In the hierarchical column of each table, we display two values, one of which is in parentheses. For example, in the first line of Table 2, we have 48.48529(19.51550) under capture time. This means that the total capture time was 48.48529 and that of this; the pursuer required 19.51550 units of time at the top level graph. The remaining capture time (48.48529-19.51550), was incurred at the lower level of the graph when both pursuer and evader are in the same fragment. This convention is used throughout.

HierarchicalGraphMethod(l_p, l_e)
1. numUpdates = 0;
2. totalTime = 0;
3. While (fragment(l_p) != fragment(l_e)) Do
4. spath = ShortestPath (HG, l_p, l_e);
5. Move the pursuer to the first node U in spath;
6. totalTime = totalTime + dist (l_p, U)/ s_p ;
7. l_p = U;
8. l_e = update();
9. numUpdates++;
10. Call any PUG method;

Figure. 6 Hierarchical Graph Algorithm

Table 1 evaluates four implementations of our top-level shortest path problem. Implementations I2 and I4 explicitly add edges from the pursuer to source border vertices and from target border vertices to the evader and call Dijkstra's algorithm once. The difference between them is that I2 adds an edge whether or not the shortest path lies entirely within the fragment, whereas I4 only adds an edge when all of the edges of the shortest path lie in the fragment. Implementations I1 and I3 do not explicitly add edges. They compute the shortest path by running Dijkstra's algorithm for every pair of border vertices (U,V) such that U is a source vertex in the fragment
containing the evader and \( V \) is a target vertex in the fragment containing the evader. \( I_3 \) differs from \( I_1 \) in that a combination \((U, V)\) is eliminated if the shortest path from the pursuer to \( U \) (or the shortest path from \( V \) to the evader) does not completely lie within its fragment. The remaining tables use Implementation \( I_2 \) as it had the best run time. The result of experiments comparing hierarchical and flat graphs are displayed in Tables 2 and 3.

In every one of the 21 experiments for grids and 36 experiments on the Denver area road network, the number of updates required by the hierarchical graph was less (and often significantly less) than the number of updates required by the flat graph. In 13 out of 57 experiments, the flat graph algorithm was faster than the hierarchical graph, usually by a small amount. In the remaining 44 experiments, the hierarchical graph was faster, sometimes by as much as a factor of 2-3. The capture times in both cases are almost identical. Finally, Table 4 was designed to compare the results when different partition sizes were used in order to obtain an optimal partition size. The results shown below are not very conclusive. In general, both the run time and number of updates appear to increase when the number of partitions increases. This suggests that using fewer partitions (perhaps 4 or 6) is the optimal strategy.

Reference


<table>
<thead>
<tr>
<th>Pursuer Node</th>
<th>Evader Node</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>5144</td>
<td>10467</td>
<td>21520</td>
</tr>
<tr>
<td>9456</td>
<td>4520</td>
<td>7070</td>
</tr>
<tr>
<td>9118</td>
<td>10178</td>
<td>57440</td>
</tr>
<tr>
<td>4696</td>
<td>14265</td>
<td>13460</td>
</tr>
<tr>
<td>3414</td>
<td>8717</td>
<td>8900</td>
</tr>
<tr>
<td>3375</td>
<td>5801</td>
<td>33240</td>
</tr>
</tbody>
</table>

Table 1. A comparison of different implementations of the top-level shortest path algorithm. I2 is the best implementation.

<table>
<thead>
<tr>
<th>Method</th>
<th>Evader's Speed</th>
<th>Capture Time</th>
<th>No. Update</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hier</td>
<td>Flat</td>
<td>Hier</td>
</tr>
<tr>
<td>PPE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>48.48529(19.51550)</td>
<td>48.48529</td>
<td>14(1)</td>
<td>23</td>
</tr>
<tr>
<td>100</td>
<td>55.43330(19.51550)</td>
<td>55.43330</td>
<td>28(1)</td>
<td>42</td>
</tr>
<tr>
<td>150</td>
<td>64.75460(19.51550)</td>
<td>64.75460</td>
<td>46(1)</td>
<td>65</td>
</tr>
<tr>
<td>PPRSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>46.90367(19.51550)</td>
<td>46.90367</td>
<td>13(1)</td>
<td>22</td>
</tr>
<tr>
<td>100</td>
<td>55.43330(19.51550)</td>
<td>55.43330</td>
<td>28(1)</td>
<td>42</td>
</tr>
<tr>
<td>150</td>
<td>64.20440(19.51550)</td>
<td>64.20440</td>
<td>48(1)</td>
<td>67</td>
</tr>
<tr>
<td>PPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>46.90367(19.51550)</td>
<td>46.90367</td>
<td>13(1)</td>
<td>22</td>
</tr>
<tr>
<td>100</td>
<td>55.43830(19.51550)</td>
<td>55.43830</td>
<td>28(1)</td>
<td>42</td>
</tr>
<tr>
<td>150</td>
<td>75.68012(19.51550)</td>
<td>75.68012</td>
<td>51(1)</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 2: Results on Denver area road network with four parts.

<table>
<thead>
<tr>
<th>Method</th>
<th>Evader's Speed</th>
<th>Capture Time</th>
<th>No. Update</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hier</td>
<td>Flat</td>
<td>Hier</td>
</tr>
<tr>
<td>PPE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>8.63911(6.74768)</td>
<td>8.29346</td>
<td>5(2)</td>
<td>14</td>
</tr>
<tr>
<td>100</td>
<td>9.53438(6.74768)</td>
<td>9.52949</td>
<td>8(2)</td>
<td>22</td>
</tr>
<tr>
<td>150</td>
<td>12.94784(8.98089)</td>
<td>11.19560</td>
<td>22(3)</td>
<td>36</td>
</tr>
<tr>
<td>PPRSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>8.63911(6.74768)</td>
<td>8.23981</td>
<td>5(2)</td>
<td>14</td>
</tr>
<tr>
<td>100</td>
<td>9.32891(6.74768)</td>
<td>9.32023</td>
<td>12(2)</td>
<td>29</td>
</tr>
<tr>
<td>150</td>
<td>11.09655(8.98089)</td>
<td>11.01700</td>
<td>12(3)</td>
<td>46</td>
</tr>
<tr>
<td>PPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>8.52603(6.74768)</td>
<td>8.23981</td>
<td>5(2)</td>
<td>14</td>
</tr>
<tr>
<td>100</td>
<td>9.32891(6.74768)</td>
<td>9.32023</td>
<td>8(2)</td>
<td>22</td>
</tr>
<tr>
<td>150</td>
<td>11.09655(8.98089)</td>
<td>11.32723</td>
<td>12(3)</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 3: Results on 100x100 grid with six parts.
<table>
<thead>
<tr>
<th>No. Of Parts</th>
<th>Evader's Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td>Capture Time (U)</td>
<td>Clock</td>
</tr>
<tr>
<td>4</td>
<td>8.63911(4)</td>
</tr>
<tr>
<td>6</td>
<td>8.63911(5)</td>
</tr>
<tr>
<td>50</td>
<td>8.23981(8)</td>
</tr>
<tr>
<td>100</td>
<td>8.23981(1)</td>
</tr>
<tr>
<td>150</td>
<td>8.33384(1)</td>
</tr>
<tr>
<td>200</td>
<td>8.24863(1)</td>
</tr>
</tbody>
</table>

Table 4. Results on a 100 x 100 with different partition sizes using PPRSS.