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: Purser, Evader, Pursuer with Updates Graph (PUG), Hierarchal 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 length(f) associated with each edge $f \in F$.
2) A pursuer p and an evader e whose initial locations are l_p^{init} and l_e^{init} , 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_e^{init} and ending at l_e^{final}.
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.

The algorithms Probabilistic Pursuit Edge (PPE) , and Probabilistic Pursuit Region (PPR) [9] are based on guessing the evader's location at some point in the future. The guess is based on a probability function. The function computes for each edge the probability that the evader is on that edge at a given time t. The experimental results in [9] show that time-parameterized techniques such as PPE and PPR do very well with respect to capture time, but not as well on the number of updates and run time as the graphs grow larger. We observe that there is an inherent trade-off between these two quantities. If the time parameter t is small, the run time is likely to be small, but the numbers of updates high since the algorithms are designed to request updates after t time units. Similarly, if t is large, the run time will be high, but the number of updates is likely to be lower. In this paper, we address this by employing hierarchical graphs to solve the PUG problem. The hierarchical graph has so far been principally used as an efficient database structure to maintain large graphs and to efficiently solve shortest path problems [1,2,6,7,8,10]. Our approach consists of pre-processing the original (large) flat graph by partitioning it into subgraphs called *fragments* and then building a two-level hierarchical graph. In order to capture the evader, we first require the pursuer to travel to the same fragment as the evader by repeatedly using a shortest-path algorithm on the top-level graph. Once the pursuer and evader are in the same fragment, the algorithm switches to the lower-level graph, where we use methods for flat graphs devised in the [9]. We describe our hierarchical approach in greater detail below. In section 2 we describe the hierarchical graph construction. Solving PUG using hierarchical graph is described in section 3. We present the experimental results in section 4.

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.

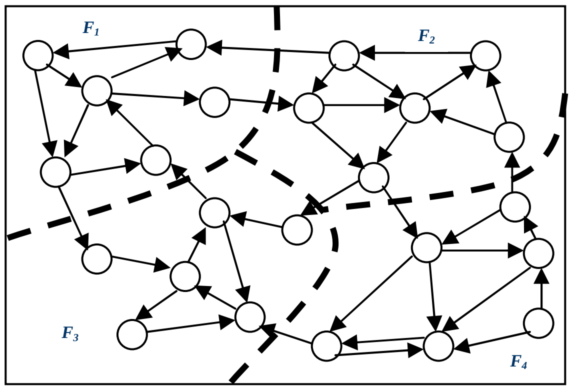

Figure 1: Partitioning: the graph consists of 24 vertices. The dashed lines divide them into four fragments consisting of six vertices each.

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.

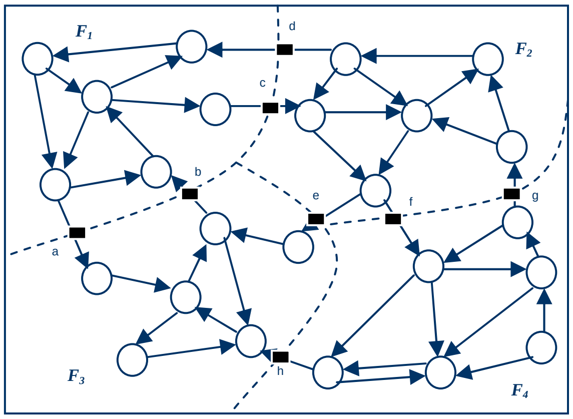

Figure 2. Border vertices *a*-*h*, denoted by filled boxes, are inserted in the middle of the corresponding border edges. We have SourceSet(F_1)={*a,c*}, TargetSet(F_1)={*b,d*}, SourceSet(F_2)={*d,e,f*}, T2argetSet(F_2)={*c,g*}, SourceSet(F_3)={*b*}, TargetSet(F_3)={*a,e,h*}, SourceSet(F_4)={*g,h*}, 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.

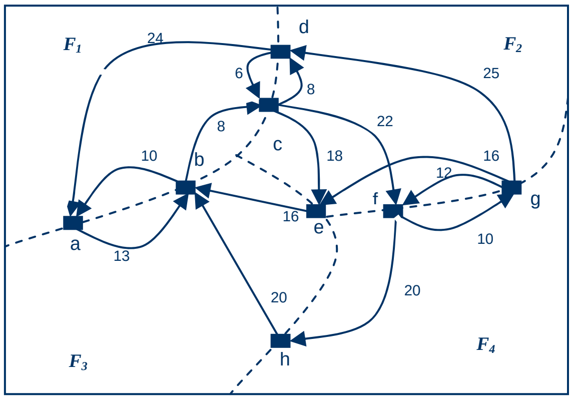

Figure 3. A top-level graph for the graph partition and border vertices of Figures 1 and 2. The edges are added as per Approach A1. Recall that fragment F_1 has TargetSet(F_1)={*b,d*} and SourceSet(F_1)={*a,c*}. The Cartesian product of these sets results in the edges (*b,a*), (*b,c*), (*d,a*), and (*d,c*). 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.

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
       n $F$;
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

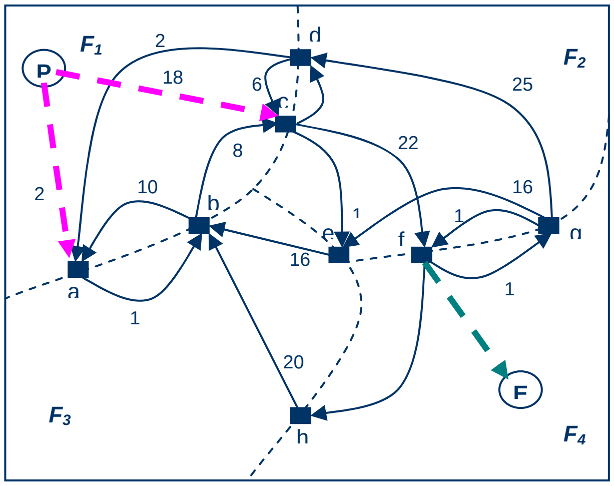

Figure. 5 Illustration of algorithm in top-level graph. Vertices P and E denote the location of the pursuer and evader, respectively. Edges (P,c), (P,a) and (f,E) are added to the top-level graph. The lengths of these edges are the lengths of the shortest paths between the vertices in the original graph. 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. totaTime = 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. I3 differs from I1 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 I2 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

[1] A. Car, H. Mehner, and G. Taylor, "Experimenting with hierarchical way finding," Tech. Rep. 011999, Department of Geomatics, University of Newcastle, England, 1999.

[2] E. P. F. Chan and N. Zhang, "Finding shortest paths in large network systems," in GIS '01: Proceedings of the 9th ACM international symposium on Advances in geographic information systems, (New York, NY, USA), pp. 160-166, ACM Press, 2001.

[3] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: application in vlsi domain," in DAC '97: Proceedings of the 34th annual conference on Design automation, (New York, NY, USA), pp. 526-529, ACM Press, 1997.

[4] G. Karypis and V. Kumar, "Multilevel k -way hypergraph partitioning," in Design Automation Conference, pp. 343-348, 1999.

[5] G. Karypis and V. Kumar, "hMETIS A Hypergraph Partitioning Package version 1.5.3. University of Minnesota, 1998.

[6] J. Feng and T. Watanabe, "Search of continuous nearest target objects along route on large hierarchical road network," in Proceedings of the Sixth IASTED International Conference ON Control and Applications, pp. 144-149, Mar. 2004.

[7] N. Jing, Y. W. Huang, and E. A. Rundensteiner, "Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation," IEEE Trans. Knowl Data Eng., vol. 10, no. 3, pp. 409-432, 1998.

[8] N. Jing, Y. W. Huang, and E. A. Rundensteiner, "Hierarchical optimization of optimal path finding for transportation applications," in CIKM '96, Proceedings of the Fifth International Conference on Information and Knowledge Management, Rockville, Maryland, USA, pp. 261-268, ACM, Nov. 1996.

[9] Sahar Idwan, Dinesh Mehta, "Fast Probabilistic Pursuit of Mobile Entities in Road Networks", Grace Hopper Celebration of Women in Computing Proceedings, SanDiego, USA, pp. 437-444, 2006 .

[10] W. Yimin, X. Jianmin, H. Yucong, and Y. Qinghong, "A shortest path algorithm based on hierarchical graph model," in Proceedings of 2003 IEEE International Conference on Intelligent Transportation Systems, pp. 1511-1514, Oct. 2003.

[11] Y. W. Huang, N. Jing, and E. A. Rundensteiner, "Effective graph clustering for path queries in digital map databases," in CIKM '96: Proceedings of the fifth international conference on Information and knowledge management, (New York, NY, USA), pp. 215-222, ACM Press, 1996.

PursuerNode	EvaderNode	Clock			
		I1	I2	I3	I4
5144	10467	21520	**570**	21530	890
9456	4520	7070	**260**	5800	410
9118	10178	57440	**900**	48260	1230
4696	14265	13460	**540**	13320	850
3414	8717	8900	**300**	7690	430
3375	5801	33240	**590**	24620	920

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

Method	Evader's Speed	Capture Time		No. Update		Clock	
PPE		Hier	Flat	Hier	Flat	Hier	Flat
	60	48.48529(19.51550)	48.48529	14(1)	23	1680(480)	1320
	100	55.43330(19.51550)	55.43330	28(1)	42	2030(490)	2430
	150	64.75460(19.51550)	64.75460	46(1)	65	2460(490)	2890
PPRSS	60	46.90367(19.51550)	46.90367	13(1)	22	1800(480)	3100
	100	55.43330(19.51550)	55.43330	28(1)	42	2340(490)	4380
	150	64.20440(19.51550)	64.20440	48(1)	67	3310(480)	5580
PPR	60	46.90367(19.51550)	4690367	13(1)	22	2260(490)	3180
	100	55.43830(19.51550)	55.43830	28(1)	42	2800(480)	3700
	150	75.68012(19.51550)	75.68012	51(1)	70	3740(480)	4700

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

Method	Evader's Speed	Capture Time		No. Update		Clock	
PPE		Hier	Flat	Hier	Flat	Hier	Flat
	60	8.63911(6.74768)	8.29346	5(2)	14	4490(4220)	132
	100	9.53438(6.74768)	9.52949	8(2)	22	5320(4220)	3720
	150	12.94784(8.98089)	11.19560	22(3)	36	8230(6170)	11730
PPRSS	60	8.63911(6.74768)	8.23981	5(2)	14	5070(4220)	4560
	100	9.32891(6.74768)	9.32023	12(2)	29	5970(4150)	7390
	150	11.09655(8.98089)	11.01700	12(3)	46	7400(6070)	11200
PPR	60	8.52603(6.74768)	8.23981	5(2)	14	5140(4240)	5400
	100	9.32891(6.74768)	9.32023	8(2)	22	6960(4220)	12380
	150	11.09655(8.98089)	11.32723	12(3)	39	8690(6160)	29130

Table 3: Results on 100x100 grid with six parts.

No. Of Parts	Evader's Speed					
	60		100		150	
	CaptureTime(U)	Clock	Capture Time(U)	Clock	CaptureTime(U)	Clock
4	8.63911(4	410	9.32891(	4790	11.30946(	8680
6	8.63911(5	507	9.32891(	5970	11.09655(	7400
50	8.23981(8	641	9.32023(	7720	11.13522(	9660
100	8.23981(1	743	9.32023(	8940	11.01697(	12130
150	8.33384(1	897	9.32023(	9820	11.08737(	10876
200	8.24863(1	107	9.44643(	1298	10.89578(	15840

Table 4. Results on a 100 x100 with different partition sizes using PPRSS.