LPI: Approximating Shortest Paths using Landmarks

Kevin Grant [1] and **David Mould** [2]

Abstract. We present LPI, a novel approach for finding approximate shortest paths in weighted graphs. The basis of the algorithm is the presence of *landmarks* placed at vertices throughout the graph. Each landmark stores the shortest path from itself to all other vertices. LPI works by searching the intersection points along the landmark paths to approximate a path between two points. We show that the paths computed by LPI are quite accurate: within 1% of optimal in the majority of cases, while reducing the cost of exact search.

1 Introduction

Finding least-cost paths through weighted graphs is one of the classic problems in computer science. In many computer game applications, numerous path queries will be undertaken on the same graph. When games are the application domain, it may not be critical to report the optimal path, but performance (time and memory usage) is of enormous importance.

We propose a novel linear-time algorithm that can find high-quality approximations to optimal paths. The algorithm depends on preprocessing to create *landmarks*: nodes for which the distance to every other node in the graph is known [4]. An approximate path from start to destination is obtained by concatenating a partial path from one landmark to the start node with a partial path from another landmark to the destination node. We call our algorithm LPI (Landmark Pathfinding between Intersections), since it finds its path between the intersections of landmark paths.

2 Background and Previous Work

We consider the task of finding the shortest path between two vertices in the graph with n vertices and m edges; a request for a shortest path is called a *query*. We use S and G to refer to the start vertex and end vertex of our path, respectively, and the optimal distance between two nodes A and B is written $d(A, B)$.

There are many algorithms for finding least-cost paths between pairs of vertices in a graph. Beginning at vertex S, the well known BFS algorithm searches through vertices by increasing distance from S, until G is found. The A^* algorithm [5] is based on BFS, but uses a heuristic to estimate remaining distance from each vertex to G; vertices are expanded in increasing order of $g + h$, where g is the calculated distance from S to a vertex, and h is the heuristic's estimated distance from that vertex to G. A^* can provide considerable speedup over BFS, but requires the existence of a good heuristic.

Our algorithm requires solving the *single-source shortest path* (SSSP) problem for several vertices. The SSSP problem refers to finding the shortest path between one vertex (say S) and all other vertices in the graph, and can be solved by Dijkstra's algorithm [3]. Dijkstra's algorithm terminates with known values for $d(S, V)$ for every vertex V; greedy hill-climbing from V can reconstruct the shortest path from S to V in linear time in the number of edges on the path. Dijkstra's algorithm computes this information in **O**$(m + n \log n)$ time and **O**(n) space. Once this information is computed, finding the shortest path from S to any other vertex V requires no search, just a linear series of lookups: begin at V, and follow the previous pointers back to S, recording the path as you go. The operation requires **O**(k) time, where k is the length of the path. The memory required for storing the path information is **O**(n), storing two values for every vertex (distance and a pointer to the next node).

An algorithm with similar goals to LPI is HTAP [7], which also relies on preprocessing to accelerate runtime path planning and produces approximations to the optimal path. HTAP uses its preprocessing time to create a hierarchy of abstracted graphs; runtime queries are handled by creating one query for each level of the hierarchy, and using the results of searching the low-resolution graphs to constrain subsequent higher-resolution searches. Because search takes place in a constant-width "corridor" at each level, HTAP's time complexity is linear in the number of edges in the final path. HTAP uses **O**$(n \log n)$ space to store the hierarchy. However, the constraint means that the optimal path may not be found.

Similar to HTAP, the HPA* algorithm [1, 6] preprocesses a graph by subdividing it into a set of smaller subgraphs. Within each subgraph, the shortest paths are precomputed and stored. This allows path queries to be solved by searching a smaller global graph, as each subgraph can be crossed in a single step. Additional levels can be added to the hierarchy. A key difference between HTAP and HPA* is their target domains: HPA* computes over obstacle graphs, where vertices are either blocked or unblocked, traversal to blocked vertices is forbidden, and the cost of traversal between two unblocked adjacent vertices is essentially uniform across the graph. HTAP solves queries for weighted graphs. It is the latter case that we are interested in, and our algorithm will be evaluated against HTAP.

The ALT class of algorithms [4] uses landmarks to compute heuristic values for guiding an A* search. Specifically, let V and G be two vertices in the graph, and L be a landmark. By the triangle inequality, $|d(V, L) - d(L, G)|$ is a lower bound for the distance $d(V, G)$. The largest lower bound over all landmarks is used to estimate the cost between a vertex V and the goal vertex G; this estimate is used as the heuristic value to guide the A* search. The ALT algorithms are similar to LPI in that they both use landmarks. The primary difference between the ALT algorithm is that the landmarks are used only to guide the search, but the algorithm is free to explore any path between vertices. By contrast, the paths found by LPI are restricted to follow the paths stored at each landmark. This means that the paths of LPI are not guaranteed to be optimal, but it effectively

[1] Dept. of Math and Computer Science, University of Lethbridge, Lethbridge, AB, email: kevin.grant2@uleth.ca

[2] Department of Computer Science, University of Saskatchewan, Saskatoon, SK, email: mould@cs.usask.ca

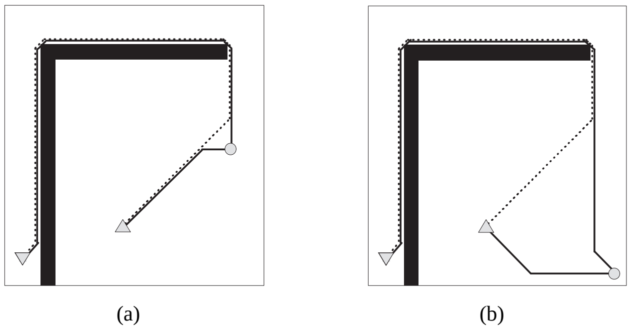


Figure 1. Some example approximation paths generated by traversing along the landmarks's shortest paths. An optimal path is shown with a dotted line. The approximate path is shown with a solid line. The start vertex is shown as a $\triangle$, while the goal vertex is shown as a $\triangledown$. The landmark's location is shown as a $\bigcirc$.

eliminates the "search" employed by ALT. In the evaluation section, we give a cost comparison of LPI to ALT.

3 LPI: Approximate Pathfinding using Landmarks

A *landmark* L is a vertex for which a shortest path is known to all other vertices in the graph. For a path query between nodes S and G, a simple and quick method for path generation is to concatenate the path from S to L (called the *upswing*) and the path from L to G (called the *downswing*). The length of the concatenated path is $d(S, L) + d(L, G)$. Figure 1 shows some examples of the paths generated by such a strategy, where obstacles are shown in black. As the figure demonstrates, the quality of the path depends on the map topology, as well as the landmark's location relative to a shortest path between A and B.

Once the landmarks are constructed, the time and space required for finding the path to and from the landmark is linear in the length of the path. To compute the information for a landmark, we use Dijkstra's algorithm, although any SSSP algorithm will suffice. If the graphs are unchanging, then the landmark information can be computed offline, and its running time can be amortized over all runs of the algorithm.

This idea of quickly generating paths from landmark paths will be the basis of our approach; the following subsections will improve upon this basic design. The motivation behind this approach rests on the observation that in many graphs, the shortest paths between many pairs of nodes favour a small subset of the edges. For instance, in the graph from Figure 1, the shortest path between any two nodes on opposite horizontal sides of the obstruction will traverse along the top edge of the obstruction. The same observation was used in designing the HTAP algorithm when constructing its hierarchy of graphs: nodes that survived into a higher level were "those which lie on the greatest number of optimal paths among the entire ensemble of paths" [7].

Not surprisingly, the landmark paths do not guarantee good approximations to the shortest path. The path generated in Figure 1(b) deviates a considerable amount from the optimal path. Figure 3(a) shows a near-pathological case – the path through the landmark is roughly 60 times as long as the optimal. In this section, we consider some extensions to the intial algorithm design. We refer to the final result as the LPI algorithm.

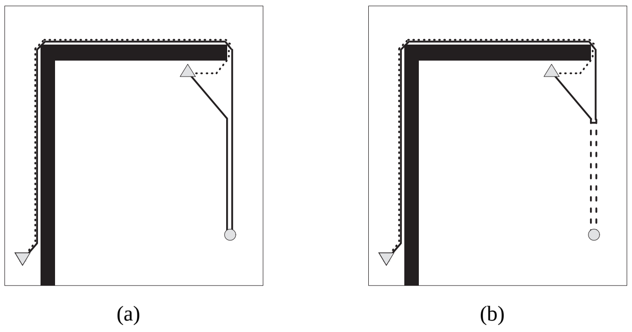


Figure 2. Further examples of approximation paths generated by traversing along the landmark's shortest paths, demonstrating path overlap. The approximated path is substantially improved by avoiding this overlap (shown by the dashed line).

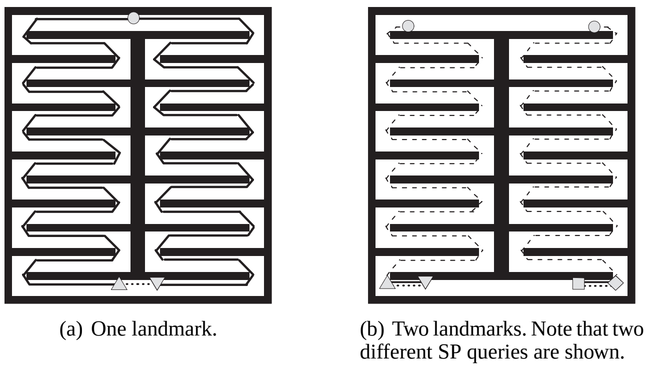


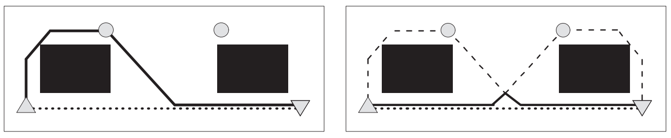
Figure 3. Further example paths generated through a maze-like terrain, demonstrating the importance of landmark placement and number.

Path Overlap. For each landmark, the union of the stored paths form an implicit tree rooted at the landmark's vertex. Rather than traversing from S to root, we need only traverse from S to the *least common ancestor* (LCA) of S and G in this implicit tree (the same argument applies to G). Following the path past the LCA creates overlap, as shown in Figure 2(a). By avoiding this overlap in the upswing and downswing, the quality of the approximated path can be improved substantially (Figure 2(b)).

Traversing to the LCA and traversing to the landmark have the same time complexity: both algorithms are linear in the number of vertices of the path. Traversing to the LCA requires a bi-directional traversal, starting at both S and G, where at each step, we move from the point that is farthest from the landmark along the stored path towards the landmark. This algorithm ensures that both points will meet at the LCA without moving past it.

Multiple Landmarks. The quality of a path depends on the placement of the landmark relative to the path endpoints, but it is not trivial (and perhaps not even possible) to place a landmark so as to produce good paths for all endpoints.

An optimal landmark placement for one path query might be a near-worst case placement for another path query. Strategic placement of the landmarks can reduce the likelihood of bad paths. However, we can alleviate the problem simply by allowing *multiple* land-



(a) Path construction – one landmark. (b) Path construction – two landmarks.

Figure 4. Demonstrating the effects of combining landmark information.

marks to exist in the graph simultaneously. When searching for a path between two points, one could consider the paths generated by each landmark, and take the best one. More generally, we will store a set of landmarks throughout our graph. We use the algorithm from the previous section to compute the path generated by each landmark (up to the LCA), and then taking the minimum amongst all paths. Figure 3 shows how pathological cases are eliminated from a simple maze by introducing a second landmark.

While multiple landmarks increase the likelihood of finding a short path, they bring with them some concerns. For one, each landmark requires memory linear in the number of nodes in the graph, which can be quite expensive; each landmark in a graph of 10^6 nodes would require space in the order of megabytes. Hence, the host architecture of the application may constrain the number of landmarks to be very few. Another concern with keeping multiple landmarks in the graph, and a further reason for minimizing their number, is running time. Finding the path associated with each landmark takes **O**(k) time (k being the length of the path), which means that r landmarks require **O**(rk) time. Note that this is an upper bound - not all paths need be expanded in some cases. For instance, in some fortunate cases, S will lie on the path from G to landmark L, or vice versa, meaning that the optimal path has been found. Finding such a path will terminate the search for a better one. As well, when one solution is found, it places an upper bound on the cost of a future solution - we can terminate the expansion of a path through a particular landmark as soon as its length exceeds that of the best path found so far. An assessment of the effect of landmark count on path quality will be considered in the *Evaluation* section.

Combining Landmark Information. Consider the path query presented in Figure 4. Figure 4(a) shows the path generated by the left landmark. The path generated by the right landmark would be a mirror image of the left path. Neither path is better than the other, and neither is of particularly high quality.

Previously, we discussed the LCA of two vertices in the context of a single landmark. Recall that the LCA in the context of a landmark L was the location where the path from S to L meets the path from G to L. With multiple landmarks, we have further intersections to consider. Consider where the upswing to the right landmark meets the downswing to the left landmark (call this point I). The path from S to I to G represents a much better approximation to the optimal path than was generated from each landmark individually (see Figure 4(b)). It turns out that we can locate these intersection points while traversing the paths to each LCA, with no effect on time complexity.

To find these intersections, we expand each upswing and downswing exactly as before, but we label each vertex that we pass with two values. In the case of an upswing expansion towards landmark L, when a vertex V is reached, we label that vertex with $d(S, V)$, since this value is readily available during this traversal (as $d(S, L) - d(V, L)$); we'll refer to this label as $d_u(V)$. In the case of a downswing expansion, when a vertex V is reached, we label that vertex with $d(G, V)$; we'll refer to this label as $d_d(V)$.

As the traversal is made, we can simultaneously check for intersections between upswings and downswings. When we label vertex V on a downswing, we can check to see if $d_u(V)$ has been given a value (and vice versa when checking an upswing). If this is the case, then the cost from traversing from S to G through V is $d_u(V) + d_d(V)$. Remembering the intersection with the lowest cost, we approximate our path by following backwards back from this spot to S and G (this is easily accomplished in the same manner as following Dijkstra paths).

Landmark Subsets. Each landmark in our graph adds to the number of paths searched when looking for a solution. However, some landmarks are more likely candidates to produce short paths for some queries. To reduce the cost of search, we employ the same strategy as ALT: we do not necessarily search all landmarks – only a subset of the landmarks are considered. We use the heuristic of first investigating landmarks whose distance to the start or goal vertex is shortest; in practice, the landmarks suggested by this heuristic do yield high-quality paths. More sophisticated landmark selection strategies are a topic for future work.

4 Evaluation

To evaluate LPI, we use the same terrains used to test the HTAP algorithm. These include four 512×512 images (Mandrill, Lena, Peppers, and Terrain), one simple maze (243×243), one complex maze (340×340), and several random terrains of varying size.

We first look at the effects of landmark count on path quality. To do this, we distribute 16 landmarks on each map by dividing the map into 16 equal-sized squares and placing a landmark at the center of each square. We then compute the LPI path for 5000 randomly generated queries, using the 2,4,8,16-closest landmarks.

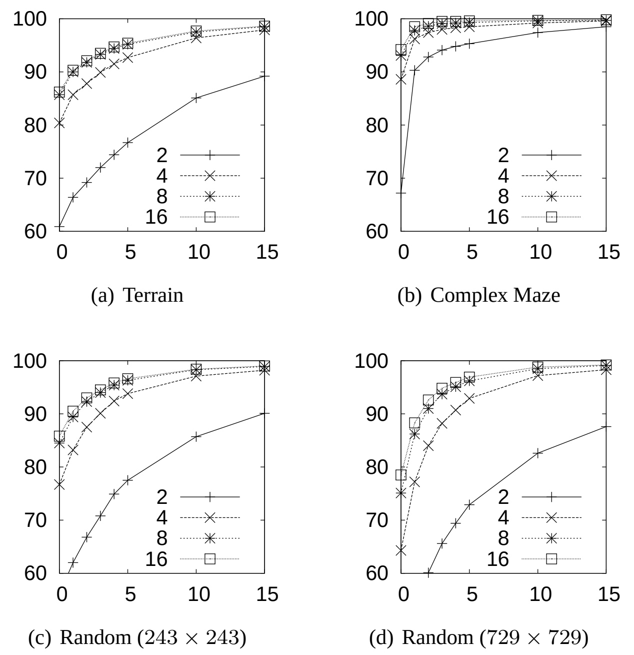
Figure 5 shows the results for four of our test maps, with one map per graph. Each line in the graph represents the number of closest landmarks used to compute the path. The graphs show path quality: the horizontal axis is the perentage difference of the approximated path's cost from the optimal path, while the vertical axis is the percentage of queries that fall within the suboptimality range shown on the horizontal axis.

As we increase the number of landmarks used to generate a path, the quality of the path increases, on average. However, the rate of increase in quality drops off rapidly as we add more landmarks. Doubling the number of landmarks from 2 to 4 has a much greater effect than doubling from 8 to 16. This suggests that good paths can be generated using only a few landmarks.

We next consider how the overall number of landmarks affects the path quality. To do this, we place 4, 9, 16, and 25 landmarks on each map. We then compute the LPI path for the same 5000 random queries, using the 4 closest landmarks. We plot these results using the same graphing procedures as in Figure 5. Figure 6 shows the results.

We see that increasing the number of landmarks in the map increases the likelihood of good approximations. In this case also, the marginal improvement decreases as the number of landmarks increases. This is an encouraging result, suggesting that reasonable paths can be generated with relatively few landmarks.

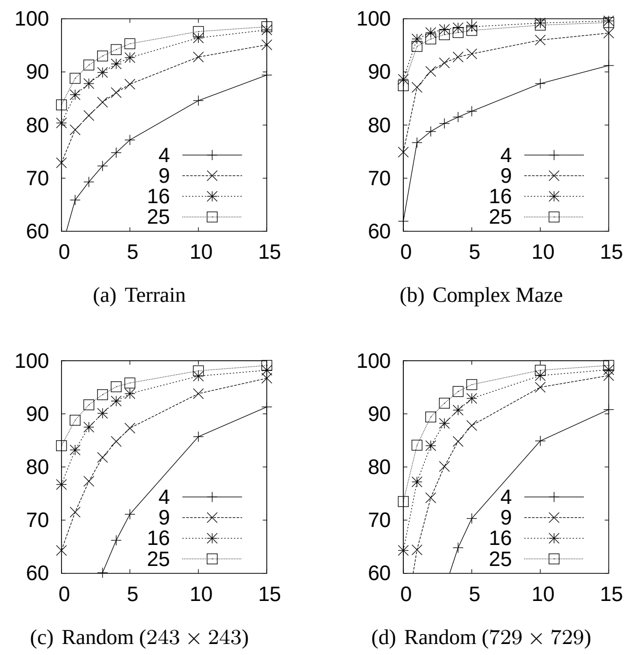
Next, we analyze LPI's scalability. That is, we empirically evaluate how well the algorithm performs with increasing graph size. To test this, we generated several random images (each pixel takes a



(a) Terrain (b) Complex Maze

(c) Random (243 × 243) (d) Random (729 × 729)

Figure 5. Path quality (percentage difference from optimal) vs. proportion of queries (percentage). The lines represent number of landmarks *used*.



(a) Terrain (b) Complex Maze

(c) Random (243 × 243) (d) Random (729 × 729)

Figure 6. Path quality (percentage difference from optimal) vs. proportion of queries (percentage). The lines represent number of landmarks *present*.

random value between 0 and 255) of size 243×243, 512×512, 729×729, and 1024×1024. Figure 7 shows the change in accuracy over 5000 test queries as the size of the random image is increased. For this test, we place 9 landmarks across the map, and use the closest 4 to generate each path. Each line in the graph is labeled with a percentage $x\%$, and the value reported by that line is the percentage of queries that were within $x\%$ of optimal, for each image size.

Not surprisingly, the path quality degrades as the graph size increases. However, the rate of degradation is slow. For instance, the number of paths found that were within 1% of optimal was 71.5% for images of size 243×243, and 56.3% for images of size 1024×1024. Increasing the number of landmarks decreased this rate of degradation. Also, when we reduce the accuracy threshold, the numbers also degrade much slower as image size increases.

Finally, we compare the LPI algorithm to other algorithms for computing shortest paths in terrain maps. We include comparisons to two algorithms: ALT and HTAP, both discussed in previous sections. We choose ALT as it uses landmarks in a similar way to LPI (albeit to compute an exact path), and HTAP as it is an approximation algorithm that exploits preprocessing. Table 1 compares the quality of the paths generated. The table does not include an explicit comparison to ALT, as ALT returns an optimal path; path quality is expressed with respect to the optimal, so ALT's path quality is implicitly compared. We compare HTAP to LPI with 6 and 9 landmarks, as 6 landmarks use roughly the same memory as the 3×3 hierarchy of HTAP for these test terrains. As with previous tests, the 4 closest landmarks are used to compute a path. Each row in the table corresponds to a test terrain, and each entry refers to the percentage of query paths that were within $x\%$ of the optimal path length, where $x\%$ refers to the column label. For example, top left entry in the table (15.3) means that for image 1 (Mandrill), 15.3% of the paths found by HTAP had cost that were within 1% of optimal.

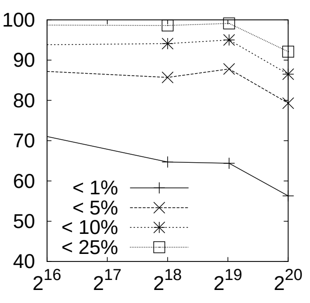


Figure 7. Path accuracy vs. image size.

LPI performs favourably in comparison to HTAP. The majority of LPI paths are typically within 1% of optimal, while the numbers are typically lower for HTAP. LPI performs particularly well in the random images, finding paths within 1% of optimal in over 45% of its test cases with 6 landmarks (by comparison, HTAP found paths of the same accuracy less than 1% of the time). When 9 landmarks are used, LPI finds paths within 1% of optimal in over 65% of cases. As regards runtime, LPI and HTAP require very similar costs, with HTAP typically performing slightly better on lower-cost paths, and LPI outperforming HTAP on higher-cost paths.

Table 2 compares the computation costs of ALT and LPI for finding the same paths reported in Table 1. When calculating computation costs, we use expanded vertices for ALT, and visited vertices for LPI. As with Table 1, each row corresponds to a terrain, and each entry refers to the percentage of query paths whose cost to find were less than $x\%$ of ALT's cost, where $x\%$ refers to the column label.

While ALT and LPI are similar in their use of landmarks, the al-

Table 1. Path length comparison between HTAP and LPI.

	HTAP			LPI (6 landmarks)			LPI (9 landmarks)		
	1%	5%	10%	1%	5%	10%	1%	5%	10%
1	15.3	67.0	84.7	65.0	78.8	87.1	75.8	87.9	93.4
2	25.3	78.4	89.4	74.8	86.4	91.4	77.2	89.0	93.4
3	17.4	71.4	87.1	78.5	87.8	92.1	86.0	91.9	94.6
4	1.3	30.3	72.4	70.7	82.2	89.4	79.1	87.7	92.8
5	57.1	92.1	96.3	83.1	86.2	89.0	83.2	86.0	88.8
6	57.1	68.6	77.5	91.5	96.4	98.1	87.1	93.4	96.0
7	0.8	3.4	16.2	61.8	82.1	90.3	71.5	87.3	93.8
8	0.1	0.6	3.8	46.2	79.3	90.0	64.4	87.8	95.0

Row labels:{1=Mandrill, 2=Lena, 3=Peppers, 4=Terrain, 5=Simple Maze, 6=Complex Maze, 7=Random (243x243), 8=Random (729x729)}.

gorithms are actually quite different, and we make several notes regarding this comparison. First, the costs associated with ALT refer to our implementation of the ALT algorithm. Our implementation employs a uni-directional A* search (as "for a small number of landmarks, regular search often has higher efficiency than bidirectional search" [4]), using the same landmarks as the LPI algorithm. Second, we count unique nodes visited by the LPI algorithm, akin to expanded nodes by ALT. Although LPI can visit nodes multiple times, the number of duplicate visits to a node is bounded by the number of landmarks, so the cost per node is **O**(k) for k landmarks; ALT's heuristic evaluation demands a calculation for every landmark, also **O**(k). Unique nodes visited is therefore the closest equivalent. Finally, we do not apply the "terminate-early" optimizations discussed previously when reporting LPI's cost. We do not terminate upon finding an optimal path early, nor do we use early solutions to bound our search for later solutions. We felt that this approach avoids biasing the results in LPI's favor.

Table 2. Computation cost comparison between LPI and ALT. Row labels are the same as for Table 1.

	LPI (6 landmarks)				LPI (9 landmarks)			
	10%	20%	30%	50%	10%	20%	30%	50%
1	58.9	77.9	84.7	90.7	52.0	73.5	82.6	89.0
2	52.4	73.8	82.1	89.3	50.3	72.6	81.4	89.0
3	61.4	79.8	86.6	92.5	65.2	81.5	87.5	92.5
4	57.7	76.5	84.2	90.6	58.9	77.4	85.1	91.6
5	51.7	74.9	80.7	85.9	55.1	77.5	83.4	88.0
6	6.5	26.6	45.9	67.6	10.3	36.2	54.9	74.0
7	18.8	43.7	58.9	73.9	13.3	42.5	58.8	74.7
8	53.4	73.8	83.2	90.0	53.1	75.4	84.0	90.3

LPI requires a fraction of the search cost incurred by ALT in most cases, improving on ALT by at least an order of magnitude in many cases. Regarding memory, the cost of ALT and LPI is roughly the same, as they both store the same landmark information.

Regarding offline costs, the LPI algorithm computes each landmark's information in roughly 2 seconds on a 729×729 graph (using a 3.0 GHz Intel processor). The time to precompute the landmark information for the previous experiments was roughly 12 seconds and 18 seconds, respectively. These times would be the same for ALT, but are about an order of magnitude less than the reported time to compute the HTAP hierarchy, albeit on a faster computer.

The landmarks of LPI are in some respects similar to the concept of *waypoints*, another technique used in pathfinding [2]. Waypoints are points placed along good paths, and path-planning algorithms will typically search for optimal routes through a system of waypoints. The search component can be optimized by pre-computing short paths between each pair of waypoints. However, clever game players can often ascertain the location of these waypoints and exploit this knowledge to their advantage (for instance, setting up defences at waypoints). By contrast, the LPI algorithm uses the *intersections* of the paths from landmark to endpoints when searching for short paths. These intersections can vary amongst different pairs of start and end points, making it more difficult to exploit known paths.

5 Discussions and Future Work

We present LPI, a novel approach for finding approximate shortest paths in terrain maps. The basis of the algorithm is the presence of landmarks, placed at different vertices throughout the graph. Each landmark stores the shortest path from itself to all other vertices. LPI works by searching for intersection points along the landmark paths to approximate a path between two points. On evaluation, the paths computed by LPI are reasonably accurate – within 1% of optimal in roughly 75% of cases with 9 landmarks placed in the graphs. The LPI algorithm reduces the online runtime of ALT considerably – an important property in an environment like a computer game, where processing time is scarce. Furthermore, LPI offers a dynamic time/quality tradeoff: more landmarks can be considered if processing is available, but if little processing is feasible, the results from few landmarks can still provide a path. Precomputing the landmark's information is quite fast – less than one minute in all experiments. And unlike waypoints, the dynamic nature of the path intersections makes the computed paths less predictable and less prone to exploitation in a game setting.

There are many avenues of future research to consider. The memory requirements of landmark storage is an important issue, especially when we consider that more landmarks are required to maintain accuracy when the graph size increases. We are currently evaluating methods for reducing the amount of storage required by each landmark (e.g., storing a subset of the shortest paths at each landmark). Additionally, we have contemplated optimizations to the search component of LPI, such that we could find the best intersections sooner and terminate our search earlier. We will also consider other approaches to landmark placement, such as those considered in [4], and look at placement strategies for non-planar graphs.

ACKNOWLEDGEMENTS

We wish to thank Michael Horsch for his assistance with the HTAP algorithm, and the reviewers for their helpful comments. This research was supported by NSERC RGPIN 258424 and 299070-04.

REFERENCES

[1] A. Botea, M. Müller, and J. Schaeffer, 'Near Optimal Hierarchical Path-Finding', *Journal of Game Development*, **1**(1), 7–28, (2004).

[2] M. A. DeLoura, *Game Programming Gems*, Charles River Media, 2000.

[3] E. W. Dijkstra, 'A Note on Two Problems in Connexion with Graphs', *Numerische Mathematik*, 269–271, (1959).

[4] A. V. Goldberg and C. Harrelson, 'Computing the Shortest Path: A* Search meets Graph Theory', in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 156–165, (2005).

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths in Graphs', *IEEE Trans. Syst. Sci. and Cybernetics*, **SSC-4**(2), 100–107, (1968).

[6] M. R. Jansen and M. Buro, 'HPA* Enhancements', in *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 84–87, (2007).

[7] D. Mould and M. C. Horsch, 'An Hierarchical Terrain Representation for Approximately Shortest Paths', in *Proceedings of Ninth Pacific Rim International Conference on Artificial Intelligence*, pp. 104–113, (2004).