TRANSIT Routing on Video Game Maps

Leonid Antsfeld
NICTA and UNSW
first.last@nicta.com.au

Daniel Harabor
NICTA and ANU
first.last@nicta.com.au

Philip Kilby
NICTA and ANU
first.last@nicta.com.au

Toby Walsh
NICTA and UNSW
first.last@nicta.com.au

Abstract

TRANSIT (Bast, Funke, and Matijevic 2006) is a fast and optimal technique for computing shortest path costs in road networks. It is attractive for its usually modest memory requirements and impressive running times. In this paper we give a first analysis of TRANSIT routing on a set of popular grid-based video-game benchmarks taken from the AI pathfinding literature. We show that in the presence of path symmetries, which are inherent to most grids but normally not road networks, TRANSIT is strongly and negatively impacted, both in terms of performance and memory requirements. We address this problem by developing a new general symmetry breaking technique which adds small random ϵ-values to edges in the search graph, reducing the size of the TRANSIT network by up to 4 times while preserving optimality. Using our enhancements TRANSIT achieves *up to four orders of magnitude* speed improvement vs. A* search and uses in many cases only a small ($\leq$ 10MB) or modest ($\leq$ 50MB) amount of memory. We also compare TRANSIT with CPDs, a recent and very fast database-driven pathfinding approach. We find the algorithms have complementary strengths but also identify a class of problems for which TRANSIT is *up to two orders of magnitude* faster than CPDs using a comparable amount of memory.

Introduction

The AI and Game Development communities have devoted much attention to the study of both exact and approximate techniques that speed up forward state-space search algorithms such as Dijkstra and A*. These efforts range from: **(i)** abstraction-based near-optimal techniques such as (Botea, Müller, and Schaeffer 2004; Sturtevant 2007) **(ii)** precomputation algorithms for improving heuristic estimates; for example (Sturtevant et al. 2009; Goldenberg et al. 2010) **(iii)** online and offline pruning and symmetry breaking methods such as (Björnsson and Halldórsson 2006; Pochter, Zohar, and Rosenschein 2009; Harabor and Grastien 2011) and **(iv)** compressing the entire set of All-Pairs data as in (Botea 2011). Almost all involve a speed vs. memory tradeoff and typically deliver improvements in the range of one or (as in the case of (Botea 2011)) two orders of magnitude.

In addition to computing shortest paths it is sometimes desirable to efficiently calculate the distance between two units on a map or the distance to an object of interest. Support for such *distance queries* is found in popular game libraries, including Umbra 3 [1], where they are used for optimizing game logic and driving scripted events. Distance queries may also be useful for higher level AI; e.g. as described in (Champandard 2009).

TRANSIT (Bast, Funke, and Matijevic 2006) is a well known and influential technique that supports both shortest path and distance queries. Developed in the Algorithmics community for the purpose of routing on road networks, TRANSIT can be described as an optimality preserving abstraction technique whose primary advantages are typically modest memory requirements and the ability to eliminate almost entirely the need for state-space search. For example, on the 24 million node US road network, TRANSIT requires less than 500MB of storage and answers 99% of all distance queries without any state-space search in just 12 μs [2].

TRANSIT's performance is due to a natual property of most road networks: low *highway dimension* (Abraham et al. 2010). Intuitively, a graph has low highway dimension if, for all nodes inside an area of radius r, there are a small set of vertices that cover all shortest paths of length longer than $k \cdot r$, for some value k. It is unclear to what extent this characteristic applies to grid maps. Previous work (Goldberg, Kaplan, and Werneck 2006) has indicated that, for randomly-weighted grids at least, this property is unlikely to hold. However a recent study (Sturtevant 2012) suggests that for certain popular grid benchmarks, particularly those drawn from video games, the opposite may be true.

Our contribution is as follows: We give a first detailed evaluation of TRANSIT on popular grid domains from the AI literature. We find in the first instance that TRANSIT is strongly and negatively impacted, in terms of running time but also memory, by uniform-cost path symmetries that are inherent to many grid-based domains but not road networks. We address this with a new general technique for breaking symmetries using small additive ϵ-costs to perturb the weights of edges in the search graph. Our enhancements reduce the number of nodes in the TRANSIT network by up to 4 times and yield running times up to *4 orders of mag-*

[1] `http://www.umbrasoftware.com/`

[2] The remaining 1% are short local queries solved using state-space search that is reported to take 5 milliseconds on average.

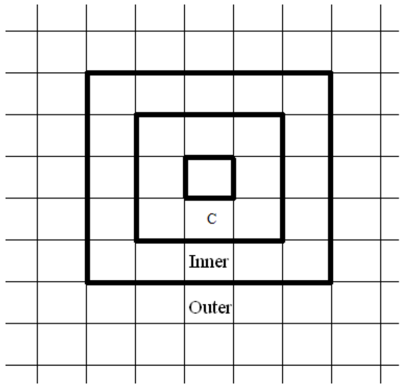

Figure 1: Example of the TRANSIT grid; also cells and inner and outer squares.

nitude faster than A* search. We also compare TRANSIT with CPDs (Botea 2011): a recent and very fast database-driven pathfinding approach. Our results indicate the two algorithms have complementary strengths and we suggest an approach by which they could be combined. However, we also identify a class of problems to which TRANSIT appears uniquely well suited. In these cases we report up to two orders speed improvement vs CPDs using a comparable amount of memory.

TRANSIT Routing

The TRANSIT algorithm is based on a very simple intuition inspired from real-life navigation: when traveling between two locations that are "far away" one must inevitably use some small set of edges that are common to a great many shortest paths (highways are a natural example). The endpoints of such edges constitute a set of so-called "transit nodes" for which the algorithm is named. TRANSIT proceeds in two phases: (i) an offline precomputation phase and (ii) an online query phase.

Precomputation

There are two steps to TRANSIT's precomputation phase. The first step identifies transit nodes and the second step builds a database of exact costs.

Identifying Transit Nodes: TRANSIT begins by dividing an input map into a grid of equal-sized cells [3]. To achieve this TRANSIT computes a bounding box for the entire map and divides this box into $g \times g$ equal-size cells. Let C denote such a cell. Further, let I (Inner) and O (Outer) be squares having C in the center, as depicted in Fig. 1. The size of the squares C, I and O can be arbitrary without compromising correctness. Their exact values however will directly impact factors such as TRANSIT's preprocessing time, storage requirements and online query times. In our experiments we report results from different combinations of square sizes. We also propose some simple heuristics for comparing between different values for these parameters.

[3]This grid is distinct from the one representing the input map.

In what follows we will compute shortest paths between nodes in C and O and look for transit nodes among the endpoints of edges that cross the border of I. Let V_C be set of nodes as follows: for every link that has one of its endpoints inside C and the other outside C, V_C will contain the endpoint inside C. Similarly, define V_I and V_O by considering links that cross I and O accordingly. Now, the set of transit nodes for the cell C is the set of nodes $v \in V_I$ with the property that there exists a shortest path from some node in V_C to some node in V_O which passes through v. We associate every node inside C with the set of transit nodes of C. Next, we iterate over all cells and similarly identify transit nodes for every other cell.

Computation and Storage of Distances: Once we have identified all transit nodes we store, for every node on the map, the shortest distance from this node to all its associated transit nodes. Recall from the previous section that every such node $v \in V$ is associated with the set of transit nodes that were found for its cell. In addition we also compute and store the shortest distance from each transit node to every other transit node. In an undirected map it is enough to compute and store costs in only one direction.

Local Search Radius

TRANSIT distinguishes between two types of queries: local and global. Two nodes for which horizontal or vertical distance (as measured in cells) is greater than some *local search radius* are considered to be "far away" and the query between them is global. We define local search radius to be equal to the size of the inner square I plus the distance from I to the outer square O. This definition guarantees that for each global query two important conditions are satisfied: (i) the start node src and destination node dst are not inside the outer squares of each other (ii) their corresponding inner squares do not overlap. Both conditions are necessary to ensure the TRANSIT algorithm is correct and optimal.

Online Query Phase

For every global query from src to dst we fetch the transit nodes associated with cells containing src and dst and choose those two that will give us a minimal cost of the combined three subpaths: $src \rightsquigarrow T_{src}$, $T_{src} \rightsquigarrow T_{dst}$, $T_{dst} \rightsquigarrow dst$. For all local queries we apply any efficient search algorithm; A* for example.

Shortest Path Extraction

By performing a series of repeated distance queries TRANSIT is able to efficiently extract actual shortest paths for any given input map. However, as path extraction is tangential to the current work, we omit the details of this procedure. The interested reader is instead encouraged to refer to the original work which contains a full description (Bast, Funke, and Matijevic 2006) of this method.

Symmetry Breaking Using Additive ϵ-costs

The TRANSIT algorithm does not appear to have been tested previously on grid-based maps of the type commonly found in video games. Upon a first attempt we observed

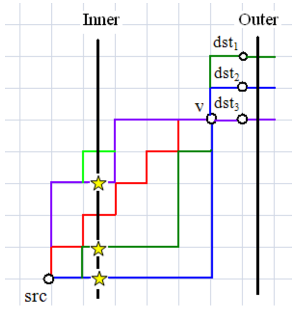

Figure 2: (a) Example of many symmetric shortest paths between src and v in 4-connected grid network. (b) Example of shortest paths from src to dst_1, dst_2 and dst_3 that share many common shortest subpaths from src to v.

the algorithm often has prohibitively large memory requirements and relatively long query times. This behavior can be traced to a property commonly found in grid maps but rarely in road networks: uniform-cost path symmetries (Harabor and Grastien 2011). To overcome this difficulty we propose a novel symmetry breaking technique which we apply during preprocessing and which involves the addition of small positive "noise" to all edge weights. This idea has been previously suggested in the context of symmetry breaking for Integer Linear Programming (Margot 2009) but the authors note that such perturbation "does not help much and can even be counter-productive". In our case, perturbation of edge weights significantly reduces the number of transit nodes and leads to faster preprocessing times, lower memory requirements and significantly better query times. To the best of our knowledge, we are the first to successfully apply such a technique to symmetry breaking in pathfinding search. Our idea is generally applicable to any kind of search graph; we exemplify it on the 4-connected grid in Figure 2.

Assume that during TRANSIT's preprocessing phase we are calculating shortest paths from the node src to each of dst_1, dst_2 and dst_3 – which all reside on the border of the Outer square O. Notice that each shortest path shares a common symmetric subpaths $src \rightsquigarrow v$ and crosses the Inner square I in three different locations. Therefore we identify T_1, T_2 and T_3 as transit nodes (starred locations in Figure 2). Our intuition is as follows: if we will add a "small" random ϵ-cost to each edge in the grid, there will be (with high probability)[4] only one shortest path to node v, e.g. through T_3. We assume here that $src \rightsquigarrow v$ will appear as a common subpath for two or more of $dst1$, $dst2$, $dst3$; although theoretically this is not guaranteed it occurs very often in practice.

We will now show that choosing ϵ sufficiently small will preserve optimality of all shortest paths.

Definition 1. *Let $G = (V, E)$ be a weighted graph with integer costs (if they're not integer, we just scale them up by a suitable factor so that they are) and let L be the length (number of links) of the longest shortest path in G. Then we define $\epsilon = \frac{1}{L}$.*

Note for all practical purposes there is no need to actually calculate the length of the longest shortest path L, but we can use $|V|$ as an upper bound.

Definition 2. *Let $G(\epsilon)$ to be an exact copy of G with the only difference that for every edge e in $G(\epsilon)$ we add a random number from the interval $(0, \epsilon)$.*

Lemma 1. *For every optimal path π_ϵ in $G(\epsilon)$ there exists a corresponding shortest path π in G that traverses through exactly the same nodes as π_ϵ.*

Proof. By contradiction. Let π_ϵ in $G(\epsilon)$ be an optimal path between two nodes src and dst. Let π be a path in G which travels through the same nodes as π_ϵ but is not optimal. This means there exists another path π' between src and dst in G which is strictly shorter than π. Let π'_ϵ be a non-optimal path in G_ϵ which travels through the same nodes as π'. Now we notice that the smallest difference between costs of any two paths in G can be at least 1. From Definition 1 this can only happen if π'_ϵ is longer than L, which is impossible. $\square$

A natural value for L in a 4-connected grid map is $\epsilon = \frac{1}{L}$; for 8-connected grids we define $\epsilon = \frac{2-\sqrt{2}}{L}$.

Corollary 1. *Number of transit nodes identified in $G(\epsilon)$ is no greater than in G*

Proof. Similar to Lemma 1. We omit it for brevity. $\square$

A direct conclusion from the latest discussion is that during the identification of transit nodes stage, we can safely substitute G with $G(\epsilon)$ and in practice eliminate all symmetric path segments. This will reduce number of transit nodes, precomputation time and storage space as well as final query time. Notice that perturbation of the graph weights in the manner described above is not specific to grid maps or indeed to any implementation of the TRANSIT algorithm. It is a general technique for reducing path symmetries in graphs.

Efficiently Approximating Network Size

TRANSIT's performance strongly depends on a set of preprocessing parameters: (i) the size of each grid cell C and (ii) sizes of the Inner square I and Outer square O. It is not clear apriori how to choose those parameters.

In this section we propose a very simple heuristic that we found useful for choosing those parameters and quickly estimating the size of the final preprocessing data as well as percentage of global vs. local queries. For a given overlay grid of size $m \times m$, we count the number of edges crossing each horizontal and vertical line of the grid. This number gives us an upper bound of the number of transit nodes and allows us to tune the grid size to match available computing resources and application requirements. Having selected a grid size S and sizes of I and O, for every query it is very

[4]In practice computer representation of real numbers is finite and commonly default to 64 bits. Assuming b_1 bits are required to represent the edges costs, and b_2 bits required to encode length of the longest shortest path in G, there are $c = 64 - b_1 - b_2$ bits available to represent ϵ-costs in order to preserve optimality. Therefore, probability of remaining with two different paths of length l with the same perturbed weight is equivalent to the probability that we draw 2 sets of size l of random numbers from 2^c numbers that sums up to the same.

Benchmark	Map	Size	States
BG	AR0602SR	299×308	23,314
	AR0700SR	320×320	51,586
DAO	orz100d	395×412	99,626
	orz900d	656×1491	96,603
Mazes-1	maze512-1-0	512×512	131,071
Random-10	random512-10-0	512×512	235,900
Rooms-32	32room_000	256×256	240,671

Table 1: Grid maps used during evaluation. We generate 10K valid problem instances for each map.

easy to verify whether it is global or local: we just need to check if the two nodes at hand are within local radius of each other. Using simple random sampling we can build a good estimate of the percentage of global vs.local queries and we can adjust our parameter values until we achieve the desired result. In our experience larger values for S, I and O yield a smaller number of transit nodes and require less memory but also cover a smaller number of global queries.

Experimental Setup

We evaluate TRANSIT on a subset (Table 1) of Sturtevant's popular and freely available [5] grid map benchmarks (Sturtevant 2012). These problem sets have appeared extensively in the literature; for example in (Botea, Müller, and Schaeffer 2004; Björnsson and Halldórsson 2006; Sturtevant 2007; Felner and Sturtevant 2009; Pochter, Zohar, and Rosenschein 2009; Goldenberg et al. 2010; Harabor and Grastien 2011). Some, such as BG and DAO, are taken from real video games. The others, comprising Rooms-32 and Mazes-1, are synthetic. We selected from each benchmark set maps we considered to be challenging; either due to the presence of extensive uniform-cost path symmetries (as discussed in the preceding section) or due to topographic features which are likely to induce significant error for standard heuristics such as Manhattan Distance and Octile Distance (e.g. narrow corridors, dead-ends etc).

For each map we generate 10K valid problems from across all possible problem lengths. Our implementation is written entirely in Java. We perform all experiments on an Intel Core2Duo with 8GB RAM. For comparative purposes we include results for a similar set of instances using Compressed Path Databases (CPD). This algorithm is originally described in (Botea 2011); its source code was kindly made available to us by the original author.

Results

We evaluate TRANSIT on each of our input maps and measure its performance in terms of: number of transit nodes per cell (T), database size (DB) and global query time. To provide a common point of reference we report the latter in terms of *speedup* which we define as relative improvement vs. standard A* search. The exception is Table 2 where we report times in μs. Database size is always in MB.

[5] http://movingai.com/benchmarks

Map	**(S, I, O)**	G		G_ϵ	
(* = no diag.)		TN	QT	TN	QT
32room_000	(32, 5, 9)	2482	14	1922	8
32room_000*	(32, 5, 9)	8858	183	1837	8
AR0602SR	(45, 5, 9)	4251	61	3618	38
AR0602SR*	(45, 5, 9)	4273	89	2173	14
AR0700SR	(40, 5, 9)	10173	205	9035	122
AR0700SR*	(40, 5, 9)	8769	268	4724	38
maze512-1-0	(38, 5, 9)	1707	2	1707	2
maze512-1-0*	(38, 5, 9)	1707	2	1707	2
orz100d	(43, 5, 9)	18852	643	16934	419
orz100d*	(43, 5, 9)	16189	844	10192	141
orz900d	(57, 5, 9)	4886	105	3732	74
orz900d*	(57, 5, 9)	2303	74	1177	29

Table 2: Effect of adding random ϵ-costs to edge weights. G is the original graph and G_ϵ is the graph with perturbed edge weights. TN = total transit nodes. QT = global query time (μs), S = grid size, I = V_I cell size, O = V_O cell size. Note that there are two versions of each map: one which allows diagonal transitions and the other which does not.

Symmetry Reduction

In Table 2 we give results for our ϵ-based symmetry breaking approach. We run TRANSIT on two variants of each input map: one where diagonal transitions are allowed and the other where they are not. Both are common in games and often studied in the literature.

In the case where diagonal transitions are disallowed the addition of random ϵ-costs to edge-weights has a dramatic effect, reducing the number of identified transit nodes by a factor of between 2-4 and reducing global query times by anywhere between 2.5 times to over one order of magnitude. When diagonal transitions are allowed (which is always the case in the remainder of this section) the improvement is less dramatic but remains strongly positive: we reduce the number of transit nodes by between 10-25% and improve global query times by up to a factor of 2.

Comparative Performance

We compare TRANSIT with Compressed Path Databases (Botea 2011) on each map in our test set [6]. CPDs are a new and highly effective procedure for compressing all-pairs data. As is often the case with TRANSIT, CPDs can solve distance queries optimally with no state-space search. Only a linear number of lookups into a compact database are required (the number is equal to the individual steps on the path). By comparison TRANSIT performs a single lookup operation which compares a up to a quadractic number of local transit nodes. Figure 3 summarises our findings. Note that values along the y-axis are log10.

We plot in most cases three curves for TRANSIT; each one represents a different set of preprocessing parameters including different abstract grid size (S) and different sizes for the Inner square I and Outer square O (we measure both of

[6] NB: diagonals are always allowed here.

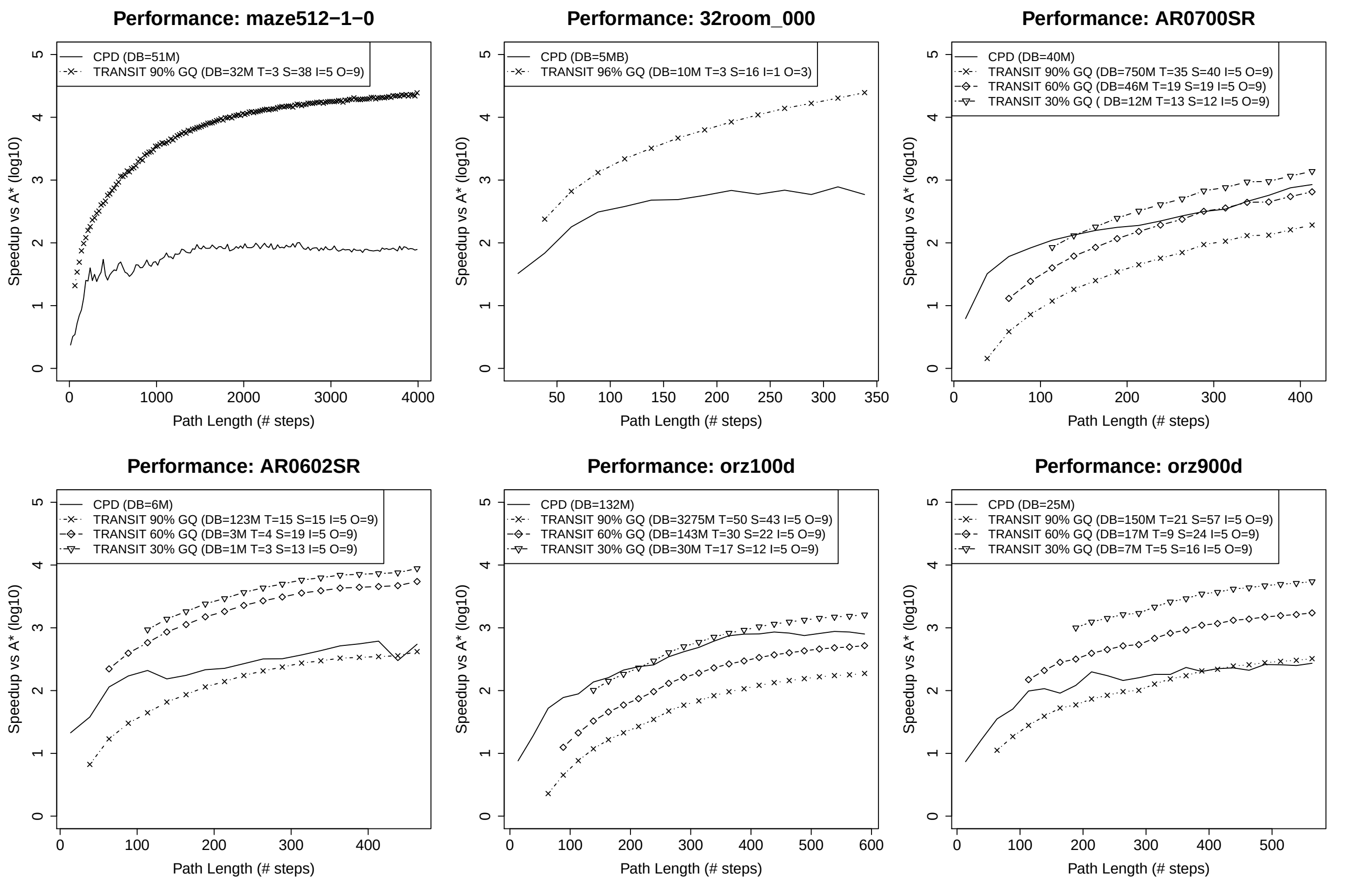

Figure 3: Search time speedup (i.e. relative improvement) of TRANSIT and CPDs vs. A*. Note the log10 scale on the y-axis.

these in terms of cells in the abstract grid). We also give the average number of transit nodes (T) for each cell in the abstract graph and the percentage of queries which are global (GQ) and do not require any state-space search. Remaining local queries are omitted (these paths are usually short and can be solved using any available search algorithm; for example Jump Point Search (Harabor and Grastien 2011)).

We observe that on domains containing no symmetries (Mazes) or only short symmetric path segments (Rooms) TRANSIT outperforms CPDs by between one and two orders of magnitude (and up to 4 orders improvement over A* search). TRANSIT requires a moderate amount of storage (32MB and 10MB respectively) and covers 90% of all queries without search (the remaining 10% are local). CPDs require 51MB and 5MB respectively and cover all queries. Notice that we store only a very small number of transit nodes per cell. We experimented with different preprocessing parameters beyond those given in Figure 3 but were unable to reduce this number further for additional speed gains.

The remaining domains, particularly orz100d and AR0700SR, are characterised by large open areas and long symmetric path segments that often span several cells in the abstract grid. These symmeries are not pruned effectively by TRANSIT using ϵ-costs. As a consequence, for 90% coverage, its performance is dominated by CPDs; both in terms of time and database size. We ran additional experiments on these problems using two smaller values for global query coverage: 60% and 30%. We used a coarser abstract grid in these cases, having larger (absolute) values for I and O. 60% global query coverage omits all paths of lengths between 50-125 (depending on the domain). 30% coverage can omit in some cases all paths up to length 175. In return for this tradeoff, we see a dramatic reduction in the average number of access nodes per cell (T) and a corresponding improvement in performance: TRANSIT is shown to be up to an order of magnitude faster than CPDs using 30% global query coverage and requires substantially less memory; in some cases just a few MB. For 60% global query coverage, TRANSIT is comparable with CPDs, both in terms of peformance and database size.

Discussion

We have shown that TRANSIT is able to compete with and even outperform CPDs for a certain class of distance query: those where the start and goal are not in close proximity. Such problems are usually considered difficult for state-space search algorithms but also for CPDs because more lookups are required and each lookup has an associated, and often linear-time, cost. On one hand, CPDs are attractive because they perform well in practice and offer complete coverage of all queries without resorting to any localized state-space search (as is sometimes the case for TRANSIT), as well naturally produce an actual path. On the other hand, CPDs have very long preprocessing times and, unlike recent variants of TRANSIT (Antsfeld and Walsh 2012), cannot incrementally repair the path database in the event of changes to the underlying network.

We find that the two methods have quite different strengths and characteristics and believe them to be orthogonal and easily combined. For example: compute a small TRANSIT database covering queries longer than some minimum length. To cover all remaining queries compute a CPD that contains only paths of lengths less than this minimum: i.e. during the all-pairs shortest path computation, do not generate any successors beyond the predefined limit. This is a much smaller subset of nodes than CPDs usually consider and we expect it will require proportionally less space to store and potentially less time to perform lookups. Once preprocessing is complete any given distance query is either global for TRANSIT, and we can extract it very fast, or we invoke TRANSIT's local query algorithm and extract the length from our local CPD. Using a similar procedure we can also very quickly extract the actual shortest path for any given distance query.

Conclusion

We report the first known results of TRANSIT (Bast, Funke, and Matijevic 2006) route-finding to grid-based benchmarks from video games. We find that on such domains the basic algorithm is impacted, in a strongly negative way, by the presence of uniform-cost path symmetries. To address this, we give a new general symmetry breaking technique involving the random perturbation of edges in the input graph with small ϵ-costs. We prove this technique is optimality preserving and show that it can reduce TRANSIT's memory overhead by several factors and improve performance by up to two orders. We undertake an extensive empirical analysis of TRANSIT on a range of popular grid-based pathfinding benchmarks taken from video games and give a first comparison of TRANSIT with CPDs (Botea 2011). We find the two have complementary strengths and identify a class of problems to which TRANSIT appears better suited: distance queries involving start and goal locations that are not in close proximity. An obvious direction for future work appears to be combining TRANSIT and CPDs. Another possibility is to reduce the number of nodes in the TRANSIT network; for example through the application of recent graph partitioning schemes (Delling et al. 2010) or the application of systematic symmetry breaking in the manner of Jump Point Search (Harabor and Grastien 2011). Finally we believe it may be possible to use a subset of the TRANSIT network as an accurate memory heuristic for A* search.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

Abraham, I.; Fiat, A.; Goldberg, A. V.; and Werneck, R. F. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, 782–793. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Antsfeld, L., and Walsh, T. 2012. Incremental updating of the transit algorithm. In *Vehicle Routing and Logistics Optimization (VeRoLog)*.

Bast, H.; Funke, S.; and Matijevic, D. 2006. Transit ultrafast shortest-path queries with linear-time preprocessing. In *In 9th DIMACS Implementation Challenge*.

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *AIIDE*, 9–14.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *J. Game Dev.* 1(1):7–28.

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *AIIDE*.

Champandard, A. 2009. Modern pathfinding techniques. In *AIGameDev.com*.

Delling, D.; Goldberg, A. V.; Razenshteyn, I.; ; and Werneck, R. F. 2010. Graph partitioning with natural cuts. Technical report, Microsoft.

Felner, A., and Sturtevant, N. R. 2009. Abstraction-based heuristics with true distance computations. In *SARA*.

Goldberg, A.; Kaplan, H.; and Werneck, R. F. 2006. Reach for a*: Efficient point-to-point shortest path algorithms. In *ALENEX*.

Goldenberg, M.; Felner, A.; Sturtevant, N.; and Schaeffer, J. 2010. Portal-based true-distance heuristics for path finding. In *SoCS*.

Harabor, D., and Grastien, Al. 2011. Online graph pruning for pathfinding on grid maps. In *25th Conference on Artificial Intelligence (AAAI-11)*.

Margot, F. 2009. Symmetry in integer linear programming. In Jümnger, M.; Liebling, T.; Naddef, D.; Nemhauser, G.; Pulleyblank, W.; Reinelt, G.; Rinaldi, G.; and Wolsey, L., eds., *50 Years of Integer Programming*. Springer. 647–681.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2009. Using swamps to improve optimal pathfinding. In *AAMAS*, 1163–1164.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*.