Taking Learning out of Real-time Heuristic Search for Video-game Pathfinding

Ramon Lawrence[1] and Vadim Bulitko[2]

[1] University of British Columbia Okanagan, ramon.lawrence@ubc.ca
[2] University of Alberta, bulitko@ualberta.ca

Abstract. Real-time heuristic search algorithms are useful when the amount of time or memory resources are limited or a rapid response time is required. An example of such a problem is pathfinding in video games where numerous units may be simultaneously required to react promptly to player's commands. Classic real-time heuristic search algorithms cannot be deployed due to their obvious state-revisitation ("scrubbing"). Recent algorithms have improved performance by using a database of pre-computed subgoals. However, a common issue is that the pre-computation time can be large, and there is no guarantee that the pre-computed data adequately covers the search space. In this work, we present a new approach that guarantees coverage by abstracting the search space using the same algorithm that performs the real-time search. It reduces the pre-computation time via the use of dynamic programming. The new approach has a fast move time and eliminates learning and "scrubbing". Experimental results on maps of millions of cells show significantly faster execution times compared to previous algorithms.

1 Introduction

As search problems become larger, the amount of memory and time to produce an optimal answer using standard search algorithms such as A* [5] increases substantially. This is an issue in resource-limited domains such as video game pathfinding. In real-time search, the amount of planning time per move is bounded independently of the problem size. This is useful when an agent does not have time to compute the entire plan before making a move. Recent real-time search algorithms such as D LRTA* [3], kNN LRTA* [2], and TBA* [1] satisfy the real-time constraint. D LRTA* and kNN LRTA* both use pre-computed subgoal databases to guide the search. However, as the search space grows, the pre-computation time is prohibitively long in practice.

In this paper, we describe a new real-time search algorithm called HCDPS (Hill Climbing and Dynamic Programming Search) that outperforms previous state-of-the-art algorithms by requiring less pre-computation time, having faster execution times, and eliminating state-revisitation. This contribution is achieved with two ideas. First, instead of using a generic way of partitioning the map (e.g., into cliques [10] or sectors [9]), we partition the map into reachability regions. The reachability is defined with respect to the underlying pathfinding algorithm which guarantees that when traversing within such regions, our agent can never get stuck. This fact allows us to replace a learning algorithm (e.g., LRTA* [7]) with simple greedy hill climbing. Doing so simplifies the algorithm, eliminates scrubbing, and allows a minimal online memory footprint per agent which, in turn, enables many more agents to path find simultaneously.

The second idea is an applying dynamic programming to database pre-computation. Once we partition the map into regions of hill-climbing reachability, we use dynamic programming to approximate optimal paths between representatives of any two such regions. This is in contrast to computing optimal paths for all region pairs with A* [3]. In our experiments, the benefits of this approximation are substantial: a two orders of magnitude speed-up in the database pre-computation time. In summary, in the domain of pathfinding on maps of over ten million states, HCDPS takes about five minutes of pre-computation per map, has a path suboptimality of about 10%, a move time of $0.23\mu s$, and overall execution time two orders of magnitude faster than A* and TBA*.

2 Problem Formulation

We define a heuristic search problem as a directed graph containing a finite set of states and weighted edges and two states designated as *start* and *goal*. At every time step, a search agent has a single *current state*, a vertex in the search graph which it can change by taking an action (i.e., traversing an out-edge of the current state). Each edge has a positive cost associated with it. The total cost of edges traversed by an agent from its start state until it arrives at the goal state is called the *solution cost*. We require algorithms to be *complete* (i.e., produce a path from start to goal in a finite amount of time if such a path exists). We adopt the standard assumption of safe explorability of the search space (i.e., there are no reachable vertices with in-edges only).

In principle, all algorithms in this paper are applicable to any such heuristic search problem. However, the presentation and experimental evaluation focus on pathfinding on grid-based video game maps. In such settings, states are vacant square grid cells. Each cell is connected to four cardinally and four diagonally neighboring cells. Out-edges of a vertex are moves available in the cell, and we use the terms *action* and *move* interchangeably. The edge costs are 1 for cardinal moves and 1.4 for diagonal moves.

An agent plans its next action by considering states in a local search space surrounding its current position. A *heuristic function* (or simply *heuristic*) estimates the optimal travel cost between a state and the goal. It is used by the agent to rank available actions and select the most promising one. We consider only *admissible* and *consistent* heuristic functions which do not overestimate the actual remaining cost to the goal and whose difference in values for any two states does not exceed the cost of an optimal path between these states. In grid maps we use the standard *octile distance* as our heuristic. The octile distance uses 1 and 1.4 as the edge costs and is equivalent to the optimal travel cost on a map without walls. An agent can modify or *learn* its heuristic function to improve its action selection with experience.

The defining property of real-time heuristic search is that the amount of planning the agent does per action has an upper bound that does not depend on the total number of states in the problem space. We measure the *move time* as the mean planning per action in terms of CPU time. The second performance measure of our study is *sub-optimality* defined as the ratio of the solution cost found by the agent to the optimal solution cost minus one and times 100%. To illustrate, suboptimality of 0% indicates an optimal path and suboptimality of 50% indicates a path 1.5 times as costly as the optimal path.

3 Related Work

Many search algorithms such as A*, IDA* [6] and PRA* [9] cannot guarantee a constant bound on planning time per action as they produce a complete solution before the first action is taken. As the problem size increases, the planning time and corresponding response time will exceed any set limit. Real-time search algorithms repeatedly interleave *planning* (i.e., selecting the most promising action) and *execution* (i.e., performing the selected action). This allows actions to be taken without solving the entire problem which improves response time at the potential cost of suboptimal solutions. LRTA* was the first algorithm and updates/learns its heuristic function with experience. The learning process may make the agent "scrub" (i.e., repeatedly re-visit) the state space to fill in heuristic local minima or *heuristic depressions* [8]. This degrades solution quality and is a show-stopper for video game pathfinding.

Improved performance is possible by pre-computing path information. In its pre-computation phase, D LRTA* abstracts the search problem using the clique abstraction of PRA* [10] and then builds a database of optimal paths between all pairs of ground-level representatives of distinct abstract states. The database does not store the entire path but only the ground-level state where the path enters the next region. Online, the agent repeatedly queries the database to identify its next subgoal and runs LRTA* to it. The issues with D LRTA* are the large amount of memory used and the lengthy pre-computation time. Further, D LRTA* repeatedly applies the clique abstraction thereby creating large irregular regions. As a result, membership of every ground state to the regions has to be explicitly stored which takes up as much memory as the search problem. Additionally, the abstract regions can contain local heuristic depressions thereby trapping the underlying LRTA* agent and causing learning and scrubbing.

kNN LRTA* attempts to address D LRTA*'s shortcomings by not using a state abstraction and instead pre-computing a set number of optimal paths between randomly selected pairs of states. On each optimal path, the farthest state that is still reachable from the path beginning via hill climbing is then stored as a subgoal. Online, a kNN LRTA* agent uses its database in an attempt to find a similar pre-computed path and then runs LRTA* to the associated subgoal. While kNN LRTA* is more memory efficient than D LRTA*, its random paths do not guarantee that a suitable pre-computed path will be found for a given problem. In such cases, kNN LRTA* runs LRTA* to the global goal which subjects it to heuristic depressions and the resulting learning and scrubbing. Additionally, pre-computing D LRTA* and kNN LRTA* databases is time-consuming (e.g., over a hundred hours for a single video game map).

TBA* forgoes LRTA* learning and runs a time-sliced version of A*. It does not pre-compute any subgoals and has to "fill in" heuristic depressions online with its open and closed lists. Thus, it consumes more memory per agent and is slower per move.

Our algorithm combines the best features of the previous algorithms. Like D LRTA* and kNN LRTA*, we run our real-time agent toward a near-by subgoal as opposed to a distant global goal, but we also guarantee that any problem will indeed have a suitable series of subgoals each of which is reachable from the preceding one via simple hill climbing. Like TBA*, we do not store or update heuristic values thereby simplifying the implementation, eliminating any scrubbing and saving memory. Unlike TBA*, we also do not use memory for open and closed lists.

4 Intuition for Our Approach

HCDPS operates in two stages: offline and online. Offline, it analyzes its search space and pre-computes a database of subgoals. The database covers the space such that any pair of start and goal states will have a suitable series of subgoals in the database. This is accomplished by abstracting the space. We partition the space into regions in such a way that any state in the region is mutually reachable via simple hill climbing with a designated state, called the *representative* of the region. Since the abstraction builds regions using hill climbing which is also used in the online phase, we are guaranteed that for any start state a our agent can hill climb to a region representative of some region A. Likewise, for any goal state b, there is a region B that the goal falls into which means that the agent will be able to hill climb from B's representative to b. All we need now is a hill-climbable path between the representative of region A and the representative of region B. Unlike canonical states used to derive better heuristics [11], our region representatives are used as subgoals. Unlike visibility polygons used in robot pathfinding, our regions are applicable to arbitrary search graphs as well as grids.

We could pre-compute such paths by running A* between representatives of any two regions on the map. However, this pre-computation is expensive as it scales quadratically with the number of regions. To speed up the pre-computation, we adopt a different approach. Specifically, for every pair of immediately neighboring regions, we run A* in the ground-level space to compute an optimal path between region representatives. We then use dynamic programming to assemble the optimal paths between immediately neighboring regions into paths between more distant regions until we have an (approximately optimal) path between any two regions. To save memory, the resulting paths are compressed into a sequence of subgoals so that each subgoal is reachable from the preceding one via hill climbing. Each such sequence of subgoals is stored as a record in the subgoal database. We then build an index for the database that maps any state to its region representative in constant time.

Online, for a given pair of start and goal states, we use the index to find their region representatives. The subgoal path between the region representatives is retrieved from the database. The agent first hill climbs from its start state to the region representative. We then feed the record's subgoals to the agent one by one until the end of the record is reached. Finally, the agent hill climbs from the region representative to the goal state.

5 Implementation Details

5.1 Offline Stage

The hill-climbing agent used offline and online is a simple greedy search. In its current state s, such an agent considers immediately neighboring states and selects the state s_{next} that minimizes $f(s_{\text{next}}) = g(s, s_{\text{next}}) + h(s_{\text{next}})$ where $g(s, s_{\text{next}})$ is the cost of traversing an edge between s and s_{next} and h is the heuristic estimate of the travel cost between s_{next} and the agent's goal. Ties in f are broken towards higher g. Remaining ties are broken randomly. The agent then moves from s to s_{next} and the cycle repeats. Hill climbing is terminated when a plateau or a local minimum in h is reached: $\forall s_{\text{next}}\,[h(s) \leq h(s_{\text{next}})]$. If this happens before the agent reaches its goal, we say that the goal is *not* hill-climbing

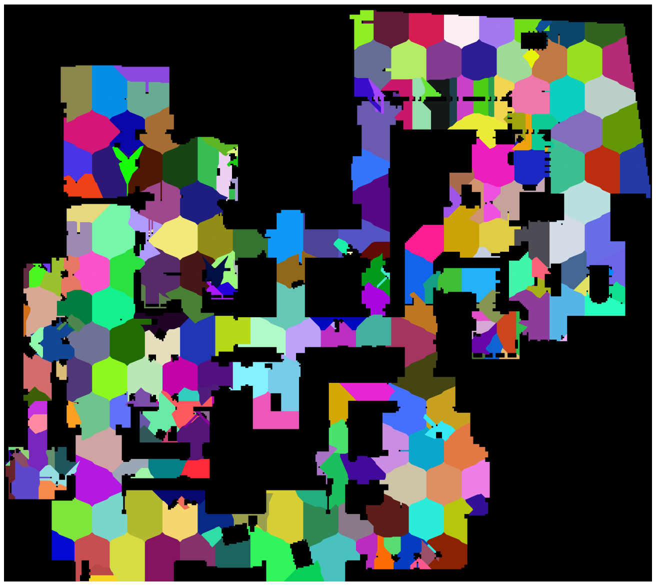

Fig. 1. Region partitioning of a grid map.

reachable from the agent's position. The agent does not use memory for heuristic values or open and closed lists.

Our partitioning mechanism is as follows. Each region R starts with a seed (representative) state r selected among yet unpartitioned states. Then, for each existing state in the region, we form a queue of candidate states to be added. Each candidate state is an immediate neighbor of some existing state in the region. For each candidate state s we check if s is mutually hill-climbing reachable with r and is closer to r than its currently associated seed state. If so, we add s to R. The distance check allows an already assigned state to change regions if its heuristic (octile) distance is closer to another region's seed. Partitioning stops when every ground state is assigned to a region. As the online part of HCDPS starts by heading for the region representative of its start region, we keep the regions fairly small to reduce suboptimality by imposing a cut-off c such that any state assigned to a region is no more than c steps from the region representative. We place seeds regularly along grid axes. In Figure 1 each region is a different color. Region shapes are based on the map properties and the initial starting seeds.

Given the regions and their representatives, we compute approximately optimal paths between all pairs of distinct region representatives with the Floyd-Warshall algorithm [4, 12] which incorporates dynamic programming. Specifically, the paths are stored in a two-dimensional array indexed by region numbers. The array is initialized with actual optimal paths computed using A* from each region's representative to representatives of the immediately neighboring regions, or in general a neighborhood depth up to L regions away. We iteratively update elements of the array until they stabilize.

Note that this problem does not exhibit optimal substructure. Specifically, an optimal path between a representative for the region A and a representative for the region B does not necessarily contain optimal ground-level paths between A and C and between

C and B even if the path passes through the region C. Thus, the computed paths are approximations to optimal paths but the savings in time are considerable.

Then we compress each computed path into a series of subgoals. The compression algorithm we use is an extension of the one used in kNN LRTA*. Specifically, given an actual path p, we initialize a compressed path p' by storing the beginning state of p in it. We then use binary search to find the state $s_i \in p$ such that s_i is not hill-climbing reachable from the end state of p' but the immediately preceding state $s_{i-1} \in p$ is. We add s_{i-1} to p' and repeat the process until we reach the end state of p which we then add to p' as well. Each compressed path is a record in our database.

The offline stage finishes with building an index over the database records to allow record retrieval in constant time. A two dimensional array is used to store a path record between each pair of region representatives. Entry (i, j) stores the database record from region representative i to j. Second, the mapping between each ground-level state and its region representative is compressed using run-length encoding (RLE) into an array sorted by state id (a unique scalar). To guarantee constant access time, we build a hash table which maps every k-th state id to its corresponding record in the RLE array. Probing the hash table involves dividing the ground-level state id G by k to get a hash table entry that maps to the RLE entry for ground-level state $\lfloor \frac{G}{k} \rfloor k$. If this RLE range does not contain G, a linear search is performed to find the correct range. In the worst case, this searches k entries if each entry represents only one ground-level state.

As an example, let the RLE table entries be: $(0, 1)$, $(625, 4)$, $(1200, 3)$, $(1600, 1)$, $(2100, 6)$. The first two entries mean that states with ids from 0 to 624 map to region 1. If $k = 1000$, the hash table has three entries: $(0, 0)$, $(1000, 1)$, $(2000, 2)$. The record $(1000, 1)$ means that id 1000 maps to entry 1 in the RLE table which is $(625, 4)$ (indexing starts at 0). Id 1000 maps to region 4 as it falls in the range $[625, 1200)$. State id 1500 maps to hash table entry $\lfloor \frac{1500}{1000} \rfloor = 1$ which is $(1000, 1)$. This gets us to RLE entry $(625, 4)$. 1500 is not in the range $[625, 1200)$ but we scan forward to find RLE entry $(1200, 3)$. Thus, state 1500 is mapped to region 3 as it is in the range $[1200, 1600)$.

5.2 Online Stage

Given a problem $(s_{\text{start}}, s_{\text{goal}})$, the HCDPS agent searches its database to find the record (r_i,r_j) where s_{start} is hill-climbing reachable to r_i and r_j is hill-climbing reachable to s_{goal}. r_i and r_j are region representatives for s_{start} and s_{goal} respectively and have a pre-computed path between them. The agent hill climbs from s_{start} to r_i, follows the subgoals in the path from r_i to r_j, and then from r_j to s_{goal}.

There are several enhancements to this basic process designed to improve solution optimality at the cost of increasing planning time per move. First, we check if the s_{goal} is hill-climbing reachable from s_{start}. If so, then the database is not used at all. Second, when we use a record, we check if its first subgoal is hill-climbing reachable from s_{start}. If so then we direct the agent to go to the first subgoal instead of the record's start. Third, when the agent reaches the last subgoal, it checks if s_{goal} is reachable from its current position. If so then it heads straight for the goal. Otherwise, it goes to the end of the record and then to the goal. Finally, to keep all such checks real-time, we limit the number of hill-climbing steps to a constant, map-independent cutoff c based on the desired response time and the amount of planning time available per move.

6 Theoretical Analysis

HCDPS has several desirable properties including:

1. Guaranteed hill-climbability within a record. For each record (i.e., a compressed path), its first subgoal is hill-climbing reachable from the path beginning. Each subgoal is hill-climbing reachable from the previous one. The end of the path is hill-climbing reachable from the last subgoal.

2. Guaranteed suitable record. For every state s there is a representative r_i state reachable from s via hill climbing. Every pair of region representatives r_i and r_j are connected by a compressed path in the database. Thus, an HCDPS agent can hill climb from s to r_i and then to r_j. From there it can hill climb to its goal state.

3. Completeness. For any solvable problem (i.e., a start and an end state that are reachable from each other), HCDPS will find a path between the start and the end in a finite amount of time with at most visiting any state twice.

Proof. By Property 1, for any problem $(s_{\text{start}}, s_{\text{goal}})$ there is a suitable database record with the start r_i and the end r_j such that r_i is hill-climbing reachable from s_{start} and s_{goal} is hill-climbing reachable from r_j. By Property 2, r_j is hill-climbable from r_i which means that HCDPS can hill climb from s_{start} to r_i to r_j to s_{goal}. Note that there are no state re-visitation within each hill climb. So the only possible state re-visitations can occur when a state visited on the climb from s_{start} to r_i gets re-visited on the climb from r_i to r_j. Likewise, a state visited on the climb from r_i to r_j can be re-visited on the climb from r_j to s_{goal}. □

4. Offline Space Complexity. Let N_R be the number of regions built by HCDPS offline. Then the number of compressed paths in the database is $O(N_R^2)$. Each path is at most $d_{\max}$ states where $d_{\max}$ is the diameter of the space and hence the worst-case database size is $O(d_{\max} N_R^2)$. Mapping between all states and their regions adds $O(N)$ space where N is the number of states. Thus, the total worst-case space complexity is $O(N + d_{\max} N_R^2)$.

5. Offline Time Complexity. An average region has N/N_R states and takes $O(N\sqrt{N/N_R})$ hill climbing steps to build, as a state can be added to at most N_R regions due to the distance check. Thus the total partitioning time is $O(NN_R\sqrt{N/N_R})$. A* is run for no more than $N_R B^L$ problems when each of the N_R regions has no more than B immediately neighboring regions. L is the depth of the neighborhood considered. Thus, the total A* run time is $O(N_R B^L N \log N)$ in the worst case. Running dynamic programming takes $O(N_R^3)$. Each of the resulting N_R^2 paths requires no more than $O(d_{\max} \log d_{\max})$ to compress. Building the compressed mapping table requires a scan of the map and is $O(N)$. Hence the overall worst-case offline complexity is $O(NN_R\sqrt{N/N_R} + N_R B^L N \log N + N_R^3 + N_R^2 d_{\max} \log d_{\max} + N)$.

6. Online Space Complexity. HCDPS uses $O(b)$ for hill climbing where b is the maximum number of neighbors of any state. However, it needs to load the database $O(d_{\max} N_R^2)$ and the index $O(N)$ resulting in the total space complexity of $O(d_{\max} N_R^2 + N)$. Note that the database is shared among $K \geq 1$ of simultaneously pathfinding agents. Thus, per-agent worst-case space complexity is $O(\frac{1}{K}(N + d_{\max} N_R^2))$.

7. Real-timeness. The worst-case online time complexity is $O(b)$. Using the hash table, database query time is $O(k)$.

7 Results

The HCDPS algorithm was compared against D LRTA*, kNN LRTA* and TBA* for pathfinding on game maps from *Counter-Strike: Source* (Valve Corporation), a popular first-person shooter. The grid dimensions varied between 4096×4604 and 7261×4096 cells giving these maps between 18 and 30 million grid cells, which is a two to three orders of magnitude increase in size over most previous papers. We did not compare against LRTA* due to its inferior performance. We did not compare to weighted A*and other approximate search algorithms as they are not real-time. Algorithms were tested using Java 6 under SUSE Linux 10 on an AMD Opteron 2.1 GHz processor.

We used 1000 randomly generated problems across four maps (one such map is in Figure 1). There were 250 problems on each map, and they had a solution cost of at least 1000. For each problem we computed an optimal solution cost by running A*. The optimal cost was in the range of $[1003.8, 2999.8]$ with a mean of 1882, a median of 1855 and a standard deviation of 550. We measured the A* difficulty defined as the ratio of the number of states expanded by A* to the number of edges in the resulting optimal path. For the 1000 problems, the A* difficulty was in the range of $[1, 200]$ with a mean of 63, a median of 36 and a standard deviation of 64.

HCDPS was run for neighborhood depth $L \in \{1, 2, 3, 4, 5\}$. D LRTA* was run with clique abstraction levels of $\{9, 10, 11, 12\}$. kNN LRTA* was run with database sizes of $\{10000, 40000, 60000, 80000\}$ records. We used a cutoff $c = 250$ steps for hill climbing and $k = 1000$ for RLE indexing. kNN LRTA* used reachability checks on the 10 most similar records. TBA* was run with the time slices of $\{5, 10, 50, 100, 500, 1000\}$ states expanded. Its cost ratio of expanding a node to backtracking was set to 10.

We chose the space of control parameters with three considerations. First, we had to cover enough of the space to clearly determine the relationship between control parameters and algorithm's performance. Second, we attempted to establish the *pareto-optimal frontier* (i.e., determine which algorithms *dominate* others by simultaneously outperforming them along two performance measures such as time per move and suboptimality). Third, we had to be able to run the algorithms in a practical amount of time (e.g., building a database for D LRTA*(8) is not practical as it takes over 800 hours).

7.1 Database Generation

Two measures for database generation are generation time and database size. Generation time, although offline, is important in practice, especially when done on the client side for player-made maps. Database generation statistics averaged per map are given in Table 1. HCDPS is one to two orders of magnitude faster than kNN LRTA* and D LRTA* (levels 9 and 10) in database generation time (**DBTime**). Additionally, HCDPS databases are two orders of magnitude smaller than those of D LRTA* and smaller than kNN LRTA* with better performance (**DBSize**). TBA* does not compute a database.

We vary how many levels of neighbors (L) are considered at the initialization of the Floyd-Warshall algorithm. More levels of neighbors improves the suboptimality performance at the cost of a longer generation time, specifically the A* time to compute paths between additional region pairs. For any value of L tried, partitioning the map, dynamic programming and path compression take about the same amounts of

Algorithm	**DBTime** (hours)	**DBSize** (KB)	**Online mem** (KB)	**Total mem** (KB)	**Move time** (μs)	**Overall Time** (ms)	**Subopt.** (%)
D LRTA* (12)	0.25	87000	19	87019	3.73	1449.37	15999.2
D LRTA* (11)	1.57	87008	11	87019	3.93	814.66	8497.1
D LRTA* (10)	11.95	87058	8	87066	4.26	662.40	6831.7
D LRTA* (9)	89.88	87453	3	87456	3.94	72.38	819.7
kNN LRTA*(10K)	13.10	256	9	265	7.56	665.00	6851.6
kNN LRTA*(40K)	51.89	1029	5	1034	6.88	93.71	620.6
kNN LRTA*(60K)	77.30	1544	4	1548	6.40	11.10	12.9
kNN LRTA*(80K)	103.09	2058	4	2062	6.55	11.30	12.0
TBA*(5)	0	0	1354	1354	14.31	579.77	1504.5
TBA*(10)	0	0	1354	1354	26.34	532.04	666.5
TBA*(50)	0	0	1354	1354	83.31	488.59	131.1
TBA*(100)	0	0	1354	1354	117.52	487.38	64.7
TBA*(500)	0	0	1354	1354	205.92	458.78	11.4
TBA*(1000)	0	0	1354	1354	229.21	459.81	5.3
HCDPS (1)	0.08	2254	0	2254	0.22	0.74	12.1
HCDPS (2)	0.09	2246	0	2246	0.23	0.78	10.6
HCDPS (3)	0.12	2229	0	2229	0.23	0.72	10.1
HCDPS (4)	0.21	2231	0	2231	0.23	0.74	10.0
HCDPS (5)	0.42	2223	0	2223	0.23	0.73	10.0
A*	0	0	1354	1354	335230	335.23	0

Table 1. Offline and online results.

time (140, 0.5 and 130 seconds respectively). However, A* takes 4 seconds for immediate neighbors ($L = 1$) and 1250 seconds for a neighborhood of depth $L = 5$. Of the total database size, approximately 58% is for compressed record storage, 38% is for the abstraction index mapping ground-level states to abstract states, and 4% is for the hash table on the abstraction index to guarantee constant time access. The abstraction mapping size is less than 1% of the number of map states. The hash table has an entry for every $k = 1000$ states resulting in a hash table of fewer than 30000 entries.

7.2 Online Performance

As per Table 1, HCDPS is greatly superior to D LRTA* in terms of suboptimality and better than kNN LRTA* as well. Furthermore, it is more robust than kNN LRTA* because it never fails to find a suitable database record and thus never resorts to the global goal. It is also more robust than D LRTA* because its core agent never gets trapped in a heuristic depression within a region. This advantage can be quantified in terms of *maximum* suboptimality over the 1000 test problems: 277000% for D LRTA*(9), 2600% for kNN LRTA*(60000) but only 49% for HCDPS(3).

Online memory is reported as the maximum size of the open and closed lists plus the storage for updated heuristics. HCDPS uses no such memory. Even when considering the total memory (i.e., adding the database size), HCDPS is substantially better than D LRTA* and approximately the same as kNN LRTA* to achieve similar suboptimality performance. It uses about 50% more memory than TBA* and A*. However, the memory advantage of TBA* and A* disappears with two or more agents pathfinding simultaneously on the same map and sharing the HCDPS database. This is very common in video game pathfinding with anywhere from half a dozen to a thousand agents pathfinding at once. Furthermore, the HCDPS database is read-only which is advantageous on certain hardware such as flash memory.

Finally, HCDPS has the fastest move (response) time of all algorithms and, in particular, is about 60 to 1000 times faster than TBA* and about 1.5 million times faster than A*, which is not a real-time algorithm and needs to compute an entire path before taking the first move. Even if HCDPS is not used as a real-time algorithm, its overall planning time per problem is still around 450 times faster than A*.

8 Conclusion and Future Work Directions

In this work we have presented HCDPS, the first real-time heuristic search algorithm with neither heuristic learning nor maintenance of open and closed lists. Online, HCDPS is simple to implement and dominates the current state-of-the-art algorithms by being simultaneously faster and better in solution quality. It is free of learning and the resulting state re-visitation — which tends to be a show-stopping problem with all previously published real-time search algorithms. This performance is achieved by computing a specially designed database of subgoals. Database pre-computation with HCDPS is two orders of magnitude faster than kNN LRTA* and D LRTA*. Finally, its read-only database gives it a smaller per-agent memory footprint than A* or TBA* with two or more agents. In summary, we feel that HCDPS is presently the best practical real-time search algorithm for video game pathfinding on static maps. Supporting dynamic search spaces by modifying the database in real-time is an avenue of future research.

References

1. Björnsson, Y., Bulitko, V., Sturtevant, N.: TBA*: Time-bounded A*. In: IJCAI. pp. 431 – 436 (2009)
2. Bulitko, V., Björnsson, Y.: kNN LRTA*: Simple subgoaling for real-time search. In: AIIDE. pp. 2–7 (2009)
3. Bulitko, V., Luštrek, M., Schaeffer, J., Björnsson, Y., Sigmundarson, S.: Dynamic control in real-time heuristic search. JAIR 32, 419 – 452 (2008)
4. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM 6(5), 345 (1962)
5. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. on Sys. Sci. and Cybernetics 4(2), 100–107 (1968)
6. Korf, R.: Depth-first iterative deepening: An optimal admissible tree search. AI 27(3), 97–109 (1985)
7. Korf, R.: Real-time heuristic search. AIJ 42(2–3), 189–211 (1990)
8. Shimbo, M., Ishida, T.: Controlling the learning process of real-time heuristic search. AI 146(1), 1–41 (2003)
9. Sturtevant, N.: Memory-efficient abstractions for pathfinding. In: AIIDE. pp. 31–36 (2007)
10. Sturtevant, N., Buro, M.: Partial pathfinding using map abstraction and refinement. In: AAAI. pp. 1392–1397 (2005)
11. Sturtevant, N.R., Felner, A., Barrer, M., Schaeffer, J., Burch, N.: Memory-based heuristics for explicit state spaces. In: IJCAI. pp. 609–614 (2009)
12. Warshall, S.: A theorem on boolean matrices. J.ACM 1(9), 11–12 (1962)