kNN LRTA*: Simple Subgoaling for Real-Time Search*

Vadim Bulitko
Department of Computing Science
University of Alberta
Edmonton, AB, T6G 2E8, CANADA
bulitko@gmail.com

Yngvi Björnsson
School of Computer Science
Reykjavik University
Kringlan 1, IS-103 Reykjavik, ICELAND
yngvi@ru.is

Abstract

Real-time heuristic search algorithms satisfy a constant bound on the amount of planning per action, independent of problem size. As a result, they scale up well as problems become larger. This property would make them well suited for video games where Artificial Intelligence controlled agents must react quickly to user commands and to other agents' actions. On the downside, real-time search algorithms employ learning methods that frequently lead to poor solution quality and cause the agent to appear irrational by revisiting the same problem states repeatedly. The situation changed recently with a new algorithm, D LRTA*, which attempts to eliminate learning by automatically selecting subgoals. D LRTA* is well poised for video games except it has a complex and memory-demanding pre-computation phase during which it builds a database of subgoals. In this paper, we propose a simpler and more memory-efficient way of pre-computing subgoals thereby eliminating the main obstacle of applying state-of-the-art real-time search methods in video games. In the domain of pathfinding on eight video game maps, the new algorithm used approximately nine times less memory than D LRTA* while finding solutions 9% worse.

1 Introduction

Artificial Intelligence (AI) controlled agents in video games must react quickly to player's commands and to other agents' actions. In order to scale up to large maps and massive number of simultaneously acting agents, AI algorithms used for such agents have to be real-time. The distinctive property of real-time operation is that an agent must repeatedly plan and execute actions within a constant time interval that is independent of the size of the problem being solved.

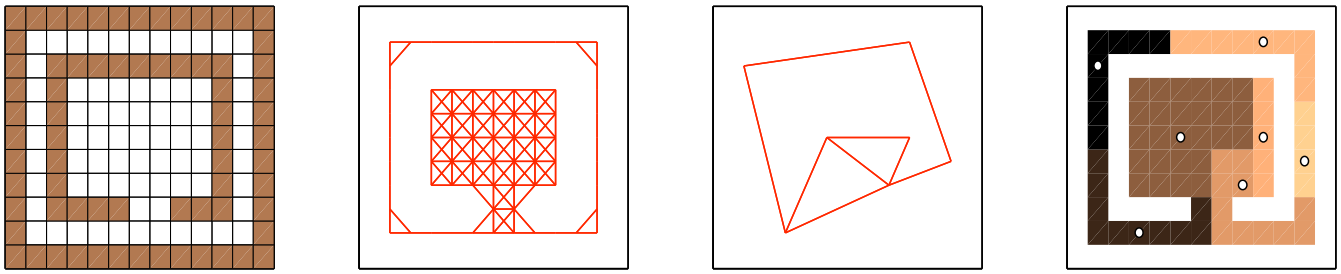
A common problem in video games is searching for a path between two locations. In this context, the real-time restriction severely limits the range of applicable heuristic search algorithms. For instance, static search algorithms such as A* (Hart, Nilsson, & Raphael 1968), IDA* (Korf 1985) and PRA* (Sturtevant & Buro 2005; Sturtevant 2007), re-planning algorithms such as D* (Stenz 1995), anytime algorithms such as ARA* (Likhachev, Gordon, & Thrun 2003) and anytime re-planning algorithms such as AD* (Likhachev *et al.* 2005) cannot guarantee a constant bound on planning time per action. This is because all of them produce a complete, possibly abstract, solution before the first action can be taken. As the problem increases in size, their planning time will inevitably increase, exceeding any predetermined limit.

Real-time search addresses the problem in a fundamentally different way. Instead of computing a complete, possibly abstract, solution before the first action is to be taken, real-time search algorithms compute (or plan) only a few first actions for the agent to take. This is usually done by conducting a lookahead search of a fixed depth (also known as "search horizon", "search depth" or "lookahead depth") around the agent's current state and using a heuristic (i.e., an estimate of the remaining travel cost) to select the next few actions. The actions are then taken and the planning-execution cycle repeats (Korf 1990). Since the goal state is not reached by most such local searches, the agent runs the risks of heading into a dead end or, more generally, selecting suboptimal actions. To address this problem, real-time heuristic search algorithms update (or learn) their heuristic function with experience.

The learning process has precluded real-time heuristic search agents from being widely deployed for pathfinding in video games. The problem is that such agents tend to "scrub" (i.e., repeatedly re-visit) the state space due to the need to fill in heuristic depressions (Ishida 1992). As a result, solution quality can be quite low and, visually, the scrubbing behavior is perceived as irrational.

A number of researchers have attempted to speed up the learning process. Approaches varied from backtracking (Shue & Zamani 1993) to heuristic weighting (Shimbo & Ishida 2003) to prioritized sweeping (Rayner *et al.* 2007) to state abstraction (Bulitko *et al.* 2007b) to multiple updates per action (Koenig 2004; Hernández & Meseguer 2005; Koenig & Likhachev 2006). However, the best performance was achieved by virtually eliminating the learning process. This can be done by computing heuristic with respect to a near-by subgoal as opposed to a distant goal. Since heuristic functions usually relax the problem (e.g., a straight-line distance heuristic ignores obstacles on a map), they tend to be more accurate closer to a goal. As a result, a heuristic function with respect to a near-by goal tends to be much more accurate and, therefore, requires little adjustment (i.e.,

*The support of RANNIS and NSERC is acknowledged.



(a) (b) (c) (d)

Figure 1: Clique-based abstraction employed by D LRTA*.

learning). The first real-time heuristic search algorithm to implement this idea was D LRTA* (Bulitko *et al.* 2007a; 2008). In a pre-processing phase, D LRTA* analyzes a state space and computes a database of subgoals which are then used at runtime. As a result, it significantly outperforms previous algorithms even with the myopic lookahead of 1.

Despite setting a new performance bar for real-time search, D LRTA* had substantial memory requirements (e.g., in pathfinding on video game maps its database took up to 70 times as much memory as the map for which it was built and the said database is loaded in the memory in its entirety) and a complex pre-processing module. The primary contribution of this paper is a simpler and more memory efficient algorithm for pre-computing subgoals for real-time heuristic search (Section 2). We base our algorithm on the "lazy learning" idea of k nearest neighbors (kNN). Namely, instead of analyzing a map and building a clique-based abstraction, the new algorithm builds a database of subgoals. During the problem-solving, we search the database for cases most similar to the problem at hand and use them to suggest a subgoal for LRTA* (Section 3). The resulting algorithm (called kNN LRTA*) is provably real-time and complete (Section 4). We empirically compare the new algorithm to D LRTA* in Section 5. We analyze the results, draw conclusions and propose future work in Section 6.

For the rest of the paper we focus on the problem of pathfinding on video game maps and use it to illustrate the ideas and to evaluate the new algorithm. Nevertheless, kNN LRTA* can be applied to other real-time search tasks.

2 Subgoal Database Construction

Our motivation is to retain D LRTA* excellent performance while simplifying the algorithm and relaxing memory requirements. The most memory-consuming part of D LRTA* lies with its subgoal database (Bulitko *et al.* 2008). To build the database, D LRTA* first abstracts the map using the clique abstraction scheme of PRA* (Sturtevant & Buro 2005). Figure 1 illustrates the process. A grid-based map (a) is converted to a graph (b) which is then abstracted so that every clique of original states becomes a single abstract state (c)[1]. Once the abstraction is complete, a ground-level state closest to the centroid of a clique is chosen as a representative for the abstract state. The representatives are marked with small white circles in part (d) of the figure. Ground-level states abstracting to different abstract states are shown as regions of different shades.

After this partitioning of a map into regions, D LRTA* considers all pairs of abstract states. For each pair, it finds an optimal path between ground-level representatives of the two abstract states. It then traces the path from the start representative and marks the ground-level state where the path enters the next region. The marked state is stored as the subgoal for the pair of the abstract states (Figure 2). The memory requirements are $O(V + V_\ell^2)$ where V is the number of ground-level states and V_ℓ is the number of abstract states at abstraction level ℓ.

D LRTA*'s code is non-trivial as one has to build the clique abstraction, find ground-level representatives and then run Dijkstra[2] from the ground-level representative corresponding to abstract goal state and trace the path from the ground-level representative corresponding to the abstract start state. Furthermore, because multi-level abstraction is used in practice ($\ell > 1$), the code has to be implemented for general graphs as opposed to rectangular grids.

We set out to come up with a simpler way of building a subgoal database. As a result, our approach is based on the following idea. The simplest and fastest version of real-time search algorithms is LRTA* with a lookahead of 1. Such a myopic LRTA* is effectively simple hill-climbing augmented with a Bellman update rule for the heuristic. To avoid learning and the problems it causes, we would like

[1]The abstraction procedure is applied successively three times.

[2]Running Dijkstra once from each abstract goal was found faster than running A* for each *pair* of abstract states.

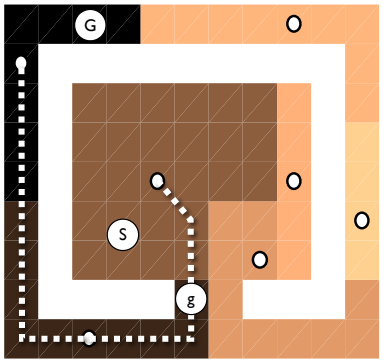


Figure 2: Subgoals as defined by D LRTA*. 'S' is the agent's state, 'G' is the global goal, 'g' is the subgoal.

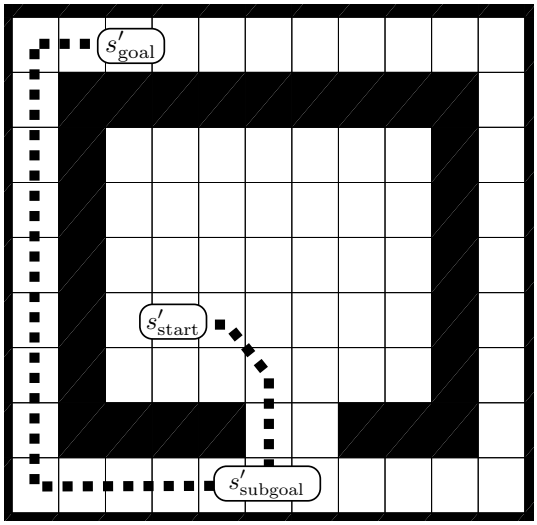


Figure 3: Subgoals as defined within kNN LRTA*.

our LRTA* *not* to learn most of the time. This can be accomplished by selecting a subgoal which is close enough to the agent to be reachable via simple hill-climbing. This intuition suggest the following definition of a subgoal: given a current state s'_{start} and a goal s'_{goal}, the best subgoal is the farthest state along an optimal path from s'_{start} to s'_{goal} which is still reachable from s'_{start} via simple hill-climbing (Figure 3).

The resulting algorithm constitutes the off-line database-building part of kNN LRTA* (Figure 4). We take a map and a desired number of subgoals N. We then generate N random pairs of ground-level states which are reachable from each other and are at least two moves apart (lines 3–5). We run A* for each pair to compute an optimal path p between the pair's states $(s'_{\text{start}}, s'_{\text{goal}})$ (line 4). Suppose $p = [s'_{\text{start}}, s'_2, \ldots, s'_{\text{goal}}]$. We then go through states $s'_i, i \geq 3$ (line 6). For each s'_i we check if it is reachable from s'_{start} by using simple hill-climbing with the heuristic computed with respect to s'_i (line 7). The subgoal s'_{subgoal} for $(s'_{\text{start}}, s'_{\text{goal}})$ is defined as the farthest state on the path p which is still hill-climbing-reachable from s'_{start}. The N records in the subgoal database Γ are of the form $(s'_{\text{start}}, s'_{\text{goal}}, s'_{\text{subgoal}})$ (line 8).

computeSubgoals(N)

```
1  load the map
2  for n = 1, ..., N do
3    generate a random pair of states (s'_start, s'_goal)
4    compute an optimal path p from s'_start to s'_goal with A*
5    if p = ∅ ∨ |p| < 3 then go to step 3 end if
6    for i = 3, ..., |p| do
7      if s'_i is not hill-climbing-reachable from s'_start then
8        store Γ_n = (s'_start, s'_goal, s'_{i-1}) in the subgoal database
9        break out of the inner for loop
10     end if
11   end for
12 end for
```

Figure 4: kNN LRTA* offline: computing subgoals.

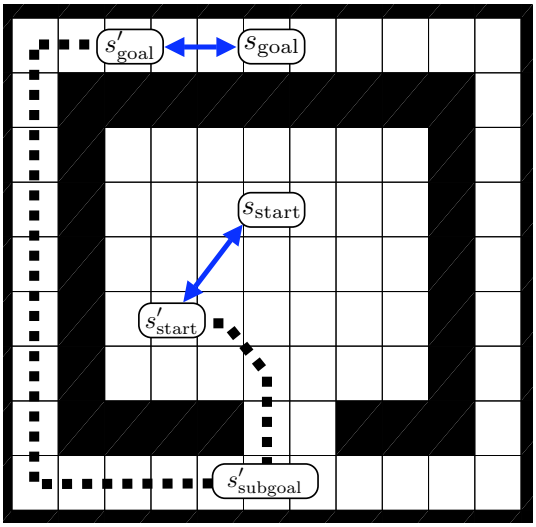


Figure 5: kNN LRTA* at runtime: finding the closest database entry. Double-ended arrows indicate heuristic distances between the agent's situation $(s_{\text{start}}, s_{\text{goal}})$ and the closest database entry $(s'_{\text{start}}, s'_{\text{goal}}, s'_{\text{subgoal}})$.

3 Runtime Search

Computing a subgoal for *every* pair of states is intractable for any non-trivial map. Thus, most of the time, the state of the agent and its goal will not be in the database of subgoals. Consequently, the agent will not be able to look up its exact goal but will have to generalize from available (state, goal, subgoal) records. One widely used and yet simple mechanism for run-time generalization is k nearest neighbors (Cover & Hart 1967). The idea lies with finding k closest cases to the problem at hand, combining their precomputed answers and using the combination as the solution.

We apply this approach for subgoal selection and, for simplicity, set $k = 1$. Thus, whenever an agent is in state s_{start} and is going to state s_{goal}, it scans the subgoal database and find the most similar record $(s'_{\text{start}}, s'_{\text{goal}}, s'_{\text{subgoal}})$ (line 4 in Figure 7). The similarity is defined as the sum of heuristic distances between s_{start} and s'_{start} and between s_{goal} and s'_{goal} as illustrated in Figure 5. The heuristic distance on maps is typically a procedurally specified function such as Euclidean distance, Manhattan distance or the octile distance (a natural generalization of Manhattan distance to diagonal actions). All of these distances ignore map obstacles.

Once the agent locates the database record most similar to its current situation, it sets the subgoal s'_{subgoal} from that record as its own subgoal and proceeds towards it using the standard LRTA* (lines 8 and 12 in the pseudocode).

There are several issues to address. First, the global goal may be hill-climbing reachable from the agent's current state and yet the agent may select a subgoal leading it away from the goal. We address this by checking if the global goal is hill-climbing-reachable from the agent's current state. If so, then we head for the global goal and forgo the subgoal selection completely (lines 3 and 10).

Second, although the subgoal s'_{subgoal} is by construction hill-climbing-reachable from s'_{start}, it is not necessarily so

kNN LRTA*($s_{\textbf{start}}, s_{\textbf{goal}}, \Gamma, m, d$)

```
1  s ← s_start
2  T ← ∅
2  while (s ≠ s_goal)
3    if s_goal is not m-hill-climbing-reachable from s then
4      (s'_start, s'_goal, s'_subgoal) ←
           arg min_{(s•_start, s•_goal, s•_subgoal) ∈ Γ} dist(s, s•_start) + dist(s_goal, s•_goal)
5      if s'_subgoal ∈ T ∨
           s'_start is not m-hill-climbing-reachable from s ∨
           s'_goal is not m-hill-climbing-reachable from s_goal then
6        go to step 4 and find the next best (s'_start, s'_goal, s'_subgoal)
7      end if
8      s_subgoal ← s'_subgoal; T ← T ∪ {s'_subgoal}
9    else
10     s_subgoal ← s_goal
11   end if
12   s ← LRTA*(d) starting in s with s_subgoal as the goal
13 end while
```

Figure 7: kNN LRTA* at runtime. Γ is the subgoal database computed off-line; m is the limit on the number of steps hill-climbing reachability checks are allowed to make; d is LRTA*'s lookahead depth; dist is the heuristic distance.

from the agent's actual state s_{start}. As a result, the agent may run into obstacles trying to reach it. We address this in several ways. First, instead of simple hill-climbing we use LRTA*, which guarantees that the subgoal will (eventually) be reached. To minimize negative manifestations of the resulting learning process, we run LRTA* with deeper lookahead d which is known to reduce learning (Bulitko & Lee 2006). Second, we can be proactive by increasing the number of records in the database in an attempt to increase similarity between agent's situation and the closest matching record in the database.

The third issue lies with possible cycles. It can so happen that the agent reaches its chosen subgoal s'_{subgoal} and then sets the next subgoal which brings it back. After reaching that subgoal, it may end up selecting s'_{subgoal} *again* thereby entering an infinite loop. We address this problem by maintaining a taboo list T of the subgoals used by the agent on a problem. As a result, no subgoal can be selected twice (lines 4 and 8 in Figure 7).

Fourth, because our similarity metric is based on the octile distance, it ignores obstacles. As a result, the most similar record may end up being on the other side of an obstacle causing the agent to spend a lot of time working its way around the obstacle. We alleviate this by checking if s'_{start} is hill-climbing-reachable form s_{start} and if s'_{goal} is hill-climbing-reachable form s_{goal} (line 5). If they are not then we go onto the second most similar record in the database and so on (as indicated by "*next best*" in line 6). If all records are exhausted, the agent defaults to the global goal (line 10).

Notice that we use hill-climbing reachability checks to deal with issues one and four. In order to satisfy the real-time constraint and prevent such checks from taking longer as the map increases in size, we limit all hill-climbing checks to a constant number of steps (m) as shown in lines 3 and 5.

4 Properties

At any time, kNN LRTA* either runs LRTA* (line 12 in Figure 7) or computes a subgoal (lines 3 through 7). The former satisfies the real-time constraint (Korf 1990). We will now show that the latter also satisfies the real-time constraint insomuch as the amount of CPU time taken by lines 3 through 7 does not increase unlimitedly with map size. This is due to the fact that the database size (N) is independent of the map size and all hill-climbing checks are limited by m. In the worst case, the agent will end computing similarity and checking hill-climbing-reachability for all records in the database (O(mN) time).

In addition to being real-time, kNN LRTA* is also complete insomuch as it finds its goal in a finite amount of time if the map is finite and the goal is located in the same connected component as the agent's initial state. Indeed, each subgoal s'_{subgoal} the agent sets out to go to (line 8) is reachable from agent's state s. This is because s'_{subgoal} lies in the same connected component as s'_{start} (by database construction) and s'_{start} is in the same component as s (checked in line 5). So the agent will always reach its subgoal and each subgoal can be used at most once (lines 5 and 8). The number of subgoals is finite, so in the worst case the agent will visit all of them before running LRTA* to the global goal.

Asymptotic memory complexity of kNN LRTA* is O(N) for the subgoal database, O(V) for LRTA*'s heuristic and O(N) for the taboo list T. Thus, in the worst case the agent consumes no more than O($N + V$) memory. Comparing to the standard LRTA*, the overhead is O(N).

5 Empirical Evaluation

For the empirical evaluation we used eight maps from popular video games: five from *Baldur's Gate* (BioWare Corp. 1998) and three from *Warcraft III* (Blizzard 2002). The maps were represented with grids sizes ranging from 150×141 to 234×274 cells (Figure 6). The timings are reported for a 2.33GHz Intel Core 2 Duo computer.

On each map we ran 128 different pathfinding problem instances with the start and goal locations chosen randomly, although constrained so that the optimal solution costs of the paths were similar, more specifically, in the range 100 to 150. Each data point we present is an average of 1024 different pathfinding searches (8 maps $\times$ 128 searches on each). The pathfinding algorithms used the octile distance as the heuristic. They were tested with several different parameterizations: for D LRTA* abstraction levels $\ell \in \{3, 5, 6, 7\}$ and for kNN LRTA* subgoal database sizes $N \in \{1000, 5000, 10000\}$ were used. The parameters m and d were set to 25 and 3, respectively.

Figure 8 presents the empirical results. The two subplots in the figure show the suboptimality of paths traversed by agents controlled by either D LRTA* (hollow markers) or kNN LRTA* (filed markers). The suboptimality is expressed as a percentage of how much more costly the paths are than optimal ones (e.g., a path costing 110 has suboptimality of 10% if the optimal path has a cost of 100). Points closer to the center of origin are preferred.

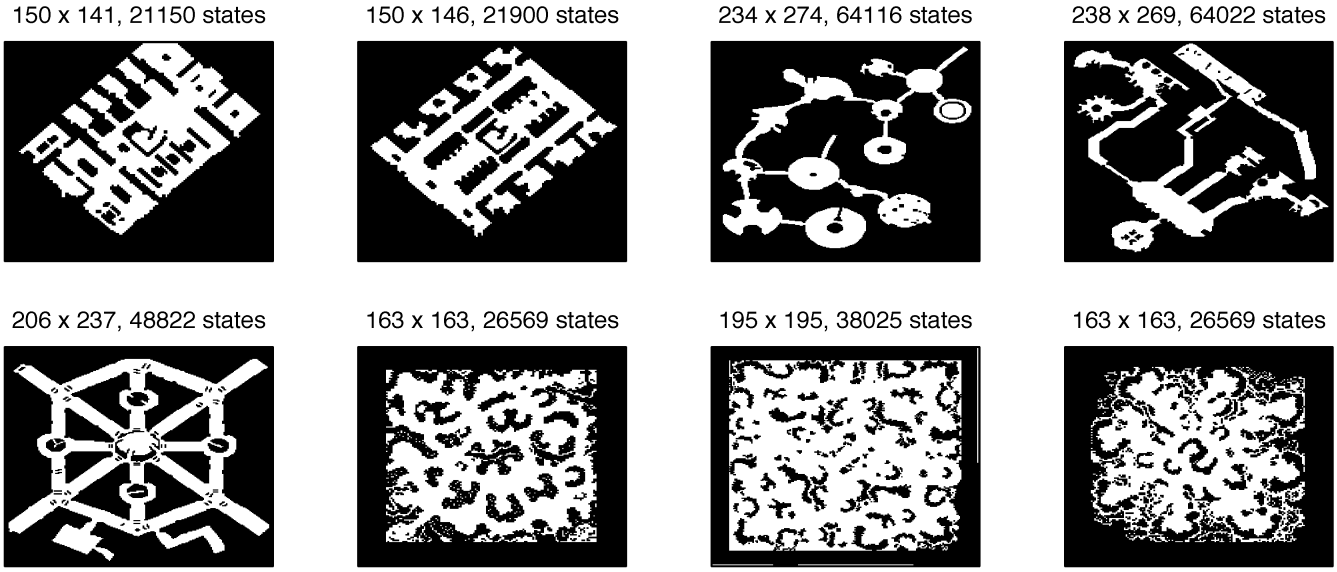


Figure 6: The eight maps used for empirical evaluation.

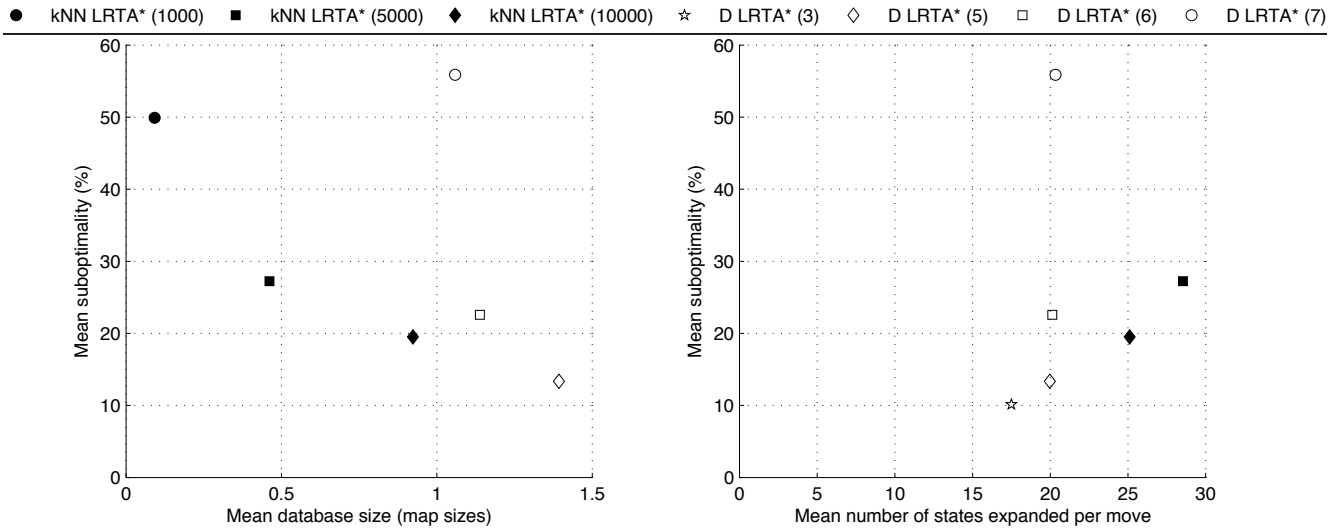


Figure 8: Suboptimality of paths as a function of database size and number of state expansions per move.

The left subplot presents the suboptimality as a function of database size. Database sizes are reported relative to the map size (e.g., relative database size of 1 means that the database takes as much memory as the map itself). Observe that kNN LRTA* subgoal databases are generally considerably smaller than their D LRTA* counterparts. For example, kNN LRTA* database with 1000 subgoals requires approximately 9 times less storage than the D LRTA* subgoal database producing comparable quality paths ($\ell = 7$). This ratio is also sizable for other parameterizations, although not as profound. Of a particular interest is the comparison between kNN LRTA* ($N = 10000$) and D LRTA* ($\ell = 6$): the former dominates the latter as it uses less memory *and* generates better paths.

The right subplot presents suboptimality as a function of the number of states expanded per move. kNN LRTA* is more expensive per move due to its reachability checks.

Table 1: Resources and performance metrics (means).

Algorithm	**Time** (min)	**Size** (rel)	**Subopt.** (%)	**Exp.** (nodes)
LRTA*	0	0	447.95	20.44
kNN LRTA* (1000)	0.62	0.09	49.91	42.6 8
kNN LRTA* (5000)	3.08	0.46	27.24	28.54
kNN LRTA* (10000)	6.23	0.92	19.52	25.11
D LRTA* (7)	0.29	1.06	55.87	20.34
D LRTA* (6)	0.35	1.14	22.57	20.14
D LRTA* (5)	0.50	1.39	13.35	19.98
D LRTA* (3)	1.91	8.22	10.13	17.50

D LRTA* suboptimality varies, exceeding that of kNN LRTA* only for abstraction levels 3 and 5. However, D LRTA* requires excessive amounts of memory (8 map sizes) for its level-3 subgoal database.

Two data points fall outside the range plotted in the

graphs: D LRTA*(3) in the left subplot (the database size is over 8 map sizes) and kNN LRTA*(1000) in the right subplot (it expands over 42 states per move). Data for all parameterizations of D LRTA* and kNN LRTA* as well as for the classic LRTA* are found in Table 1. The columns are: subgoal database precomputation time (in minutes), its size (relative to map size), suboptimality of the solution and the number of states expanded per move.

Note that the times provided are estimates of how long the precomputations would take if run with an optimized implementation written in a fast high-level programming language — a setting typical in video-game development.[3] The exact timings are implementation and map specific. Relative precomputation times for different algorithms are more important. As we can see, kNN LRTA* subgoal databases take longer to compute, although the computation times are still quite reasonable — a matter of minutes — and therefore do not exclude the approach from being used even by gamers for indexing their custom made game maps.

To summarize, we can state that kNN LRTA* performs slightly worse than D LRTA*, while requiring considerably less memory for its subgoal databases. The tradeoff is attractive for pathfinding in video games, especially on consoles where memory is often a scarce resource. This benefit comes at the expense of a somewhat longer precomputation times for generating the subgoal databases and a more complex runtime module.

6 Conclusions

The primary drive behind this work was to develop a new high-performance real-time search algorithm suitable for pathfinding in video games, by relaxing memory requirements of D LRTA*, a current state-of-the-art real-time search algorithm. The new algorithm, kNN LRTA*, offers clear benefits for pathfinding applications in video games: not only is its database computation simpler to implement than that of D LRTA* but it also offers nearly as good runtime real-time performance while requiring substantially less memory for storing the subgoal databases. Simplicity of implementation and a light memory footprint are desirable attributes of AI algorithms in video games.

As for future work, we believe that kNN LRTA* can be further enhanced to fully match D LRTA* runtime performance. We have identified several avenues for potential improvements. First, instead of selecting the start-goal pairs at random, a more intelligent way of dealing out the pairs can be envisioned, for example, by using influence maps to ensure a better coverage. Secondly our similarity metric does not take obstacles into account. The adverse side-effects it entails are mitigated by requiring a hill-climbing reachability between neighbors at the cost of additional runtime computation. Instead, an obstacle-aware similarity metric should be investigated. Finally, currently the algorithm looks for only the single nearest neighbor ($k = 1$); combining subgoal recommendations from several database records ($k > 1$) may reduce suboptimality. A natural approach would be to select a valid map state closest to the weighted average of the k subgoals.

[3] We computed the subgoal databases using MATLAB, which, on this code, is estimated to be 16.4 times slower than C++. For the estimate we ran A* on a set of pathfinding problems using both a C++ testbed (Sturtevant 2005) and our MATLAB testbed.

Acknowledgements

References

BioWare Corp. 1998. Baldur's Gate. http://www. bioware. com/ bgate/.

Blizzard. 2002. Warcraft 3: Reign of chaos. http://www. blizzard. com/ war3.

Bulitko, V., and Lee, G. 2006. Learning in real time search: A unifying framework. *JAIR* 25:119–157.

Bulitko, V.; Björnsson, Y.; Luštrek, M.; Schaeffer, J.; and Sigmundarson, S. 2007a. Dynamic Control in Path-Planning with Real-Time Heuristic Search. In *ICAPS*, 49–56.

Bulitko, V.; Sturtevant, N.; Lu, J.; and Yau, T. 2007b. Graph Abstraction in Real-time Heuristic Search. *JAIR* 30:51–100.

Bulitko, V.; Luštrek, M.; Schaeffer, J.; Björnsson, Y.; and Sigmundarson, S. 2008. Dynamic control in real-time heuristic search. *JAIR* 32:419 – 452.

Cover, T. M., and Hart, P. E. 1967. Nearest neighbor pattern classification. *IEEE Trans. on Information Theory* IT-13:21–27.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Hernández, C., and Meseguer, P. 2005. LRTA*(k). In *IJCAI*, 1238–1243.

Ishida, T. 1992. Moving target search with intelligence. In *AAAI*, 525–532.

Koenig, S., and Likhachev, M. 2006. Real-time adaptive A*. In *AAMAS*, 281–288.

Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *AAMAS*, 864–871.

Korf, R. 1985. Depth-first iterative deepening : An optimal admissible tree search. *Artificial Intelligence* 27(3):97–109.

Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2–3):189–211.

Likhachev, M.; Ferguson, D. I.; Gordon, G. J.; Stentz, A.; and Thrun, S. 2005. Anytime dynamic A*: An anytime, replanning algorithm. In *ICAPS*, 262–271.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *NIPS*.

Rayner, D. C.; Davison, K.; Bulitko, V.; Anderson, K.; and Lu, J. 2007. Real-time heuristic search with a priority queue. In *IJCAI*, 2372–2377.

Shimbo, M., and Ishida, T. 2003. Controlling the learning process of real-time heuristic search. *Artificial Intelligence* 146(1):1–41.

Shue, L.-Y., and Zamani, R. 1993. An admissible heuristic search algorithm. In *ISMIS*, volume 689 of *LNAI*, 69–75.

Stenz, A. 1995. The focussed D* algorithm for real-time replanning. In *IJCAI*, 1652–1659.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.

Sturtevant, N. 2005. HOG - Hierarchical Open Graph. http://www.cs.ualberta.ca/~nathanst/hog/.

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