*Geoff Sutcliffe (*S3352626@student.rmit.edu.au*)*
*Rhys van der Waerden (*s3134601@student.rmit.edu.au*)*

An Implementation of "Deadline-Aware Search, Using On-line Measures of Behaviour"

Introduction

The task given was to provide an agent that can compute a path between two points on a grid world map. The incumbent approach for this task is the A* algorithm. A* is a well-documented algorithm that will not be explored in any depth here. It does however, have a major shortcoming. The solutions that it retrieves are optimal, but there is often also a large amount of search space that does not contribute to the solution, but is explored in order to guarantee optimality. This is due to the imperfection of heuristics – the actual distance to the goal from each state (h*) is not known a-priori, otherwise search would hardly be necessary. Heuristics can lead the search pattern astray.

This extra effort to ensure optimality may not be appropriate in all application domains. In video games for example, a short pre-ordained time may be allotted for path finding by in-game characters. In a robot driving application, real time decisions must be made, that may not be perfect, but that are very good solutions. The commonality of all of these sub-optimal domain requirements is that a trade-off must be made. A sub-optimal search algorithm is able to trade search time for optimality.

Deadline Aware Search

Deadline Aware Search, based on [1], is the search algorithm that has been selected for this project. Briefly, Deadline Aware Search (DAS) uses online-calculated data to determine various metrics about the *search behaviour*. It then uses this data to determine how far the search would be able to continue in its current behaviour, and if this maximum depth (dMax) is deemed 'unreachable' according to the heuristic (d^cheapest(n), based on h(n)) then the node (n) is not expanded – it is pruned - and only searched if there is still time remaining by the time the entire open list has been explored. In this way, the DAS algorithm reacts to an impending deadline by not exploring those parts of the tree that it determines to be unreachable.

Search behaviour is mentioned in the previous paragraph, and warrants some explanation. An observation on A* search trees with admissible heuristics is that a child of node n, n' may have a higher value of f(n), such that f(n') > f(n). Intuitively, if our search follows this path, it causes the frontier to move away from the target. In the DAS world, we define this phenomenon as vacillation – traversing subtrees that we are likely to switch out of. Whilst A* would proclaim "for completeness", DAS would claim "If we do not expect to find a goal down one of these subtrees, then why explore it".

As we are given a time-limit in which to complete a search, and the agent has full knowledge of this time limit, it was felt that that DAS had the most to offer, when compared with the brute force efforts of the anytime algorithms. As is shown however, this is not necessarily the case. There is a lot of overhead associated with the DAS algorithm, and measurement of search performance is an imprecise task.

The implementation follows the reference material quite closely. The main deviation that has been experimented with, is on the sorting of the pruned list. The DAS paper [1] suggests to sort the pruned list on f(n). It was found that this was not progressing the search towards the goal strongly enough. This is explained in more depth later. The other change is that any node selected for expansion is not expanded if it's g(n) cost is greater than the incumbent solution.

Additional Optimizations

Pruning on G(n) cost

One simple optimization that we have applied to our search is to reject any node selected for expansion whose g cost is greater than the length of the incumbent solution. It is not possible that this node can lead to a shorter solution. Often this can create the impression that DAS is unable to find a solution in the provided deadline, but this is an effect of the software rejecting those paths that will achieve no improved result.

Speedier Search and Hybrid solution

As advised by the DAS paper [1], the search is preceded by a speedier search to cover the case where DAS finds no solution. This search is intended to find a path to the goal as quickly as possible - to that end, it sorts it's open set on h(n), checking for, and ignoring nodes that have already been expanded.

One interesting extension that has been made to this is the use of a hybrid solution. In the initial submission, this was purely used to 'stitch together' the initial greedy solution, and the in-progress DAS search, if the DAS search was unable to find any solutions.

This extension has been improved so that the hybrid solution is only created and returned to Apparate if it is actually better than the purely greedy solution. A further improvement that has been made is to stitch together the most recent completed DAS search, with any incomplete DAS path.

This 'stitching' occurs by back-tracking through the completed path, and checking each state for intersection with the current search tree. It does *not* perform any new searching.

Preferred Operators

Some investigation and experimentation went into applying the use of preferred operators [3] to our search. Three approaches were considered.

The first implementation is very simplistic. Operators that generate a node closer to the goal state than its parent were considered to be "preferred". Any node generated in this way was added to a sorted list additional to the open list - the preferred list.

When a node is selected for expansion, it is chosen from either the preferred list or the open list. For this implementation a constant was provided that defined the frequency with which the open (ie. non-preferred) list was used. If the constant was set to twenty, nineteen nodes would be taken from the preferred list for each one from the open list.
The effect of this approach is effectively the same as putting a huge weight on the heuristic cost. The search was take a needlessly greedy search through expensive terrain.

The next approach uses the paper's suggestion of keeping the open list as per a standard A* search, but adding an additional preferred list. The same criteria for preferred nodes was used. Node selection alternates between the two lists.

There is a conceptual problem to address with this approach. A node that has already been generated, but is not in the preferred list, can then be generated again with a preferred operator. This means that the node's metadata needs to be updated, and the node needs to be resorted.

Implementing this actually helped to reduce the search space quite significantly, but the cost of updating all the nodes had so much overhead that the search performed about four times slower than a normal A*.

If nodes are not updated as described above, the effect is to prune away large sections of the open set, so that they will not be considered. This would mean that even when a huge number of nodes were closed, the algorithm would leave “walls” of pruned nodes - segmenting the closed set so that a reasonable solution could not be found.

The third approach was to define a separate list for each operator type, and grade them by how successful they have been during the search. We dismissed this approach without testing it. There is a fundamental problem with using preferred operators in this context; An operator is deemed to be preferred when it takes the search towards the goal, but then evaluated in the same terms. It is a vicious cycle that cannot be applied in the grid world domain.

A logical conclusion is that the preferred operator approach would be better suited to a domain where the operators are less tightly coupled with the search heuristic. Also they make more sense as a method to defer evaluation. Deferred evaluation makes no sense in this context because our heuristic value was needed to decide whether the operator is preferred. Calculating the heuristic is also a cheap operation, so there is no gain in deferring its calculation.
Considering this, we did not use preferred operators in our DAS implementation.

Software Architecture

Class List

MyCoolAgent

This is our Apparate PlanningAgent class. It contains the logic for the DAS search, as well as the logic for our greedy "Speedier" search and the hybrid thereof.

FastDasMapInfo

Related Classes: HComparator, FComparator, WeightedHFComparator, WeightedGFComparator

This is a data structure that holds all the state data used of DAS during its search. It keeps track of the nodes in the different sets DAS uses (open, closed and pruned) and the ordering of those sets. It also can perform certain operations on its data such as the "recover pruned states" method.

Meta data held for each generated node:

- current set
- h cost
- g cost
- d cheapest
- d^ cheapest (d cheapest with error correction)
- parent node
- heuristic error
- expansion number
- cumulative error
- depth

To optimize the performance of this data structure, it has been written in the style prescribed by the data-oriented design paradigm[2]. An individual two dimensional array representing every coordinate in the map is created for each node attribute. This saves time by not requiring a separate memory allocation for each generated node, and also speeds up reads by putting data in contiguous memory (as opposed to in the heap).

The cost of this approach is in both maintainability and storage. All states are pre-allocated at the beginning of the search. This is an assumed optimisation that we are able to make because of the static size of the state space. In other, more dynamic, domains, this approach would not be possible, requiring the pushing and popping of dynamically allocated nodes, as and when they are generated by the search. There is an impact on maintainability, as any attribute added to the nodes requires it's own bi-dimensional array pre-allocated.

The open and pruned set are stored in PriorityQueue objects, which are sorted on f cost. The pruned sets comparison function uses a large weight on h to encourage the search to beam towards the goal

until the first solution is found. After the first solution is found, the weight is switched to 1.0 to guarantee convergence on optimality. Different sort orders are provided by the relevant Comparator classes.

GreedyMapInfo

Related classes: GreedyCellInfo

Similar to FastDasMapInfo, but stores data required by Speedier Search.

Sliding Window

A container for storing a capped number of values and returning their average. This is used by DAS to measure average expansion delay and average expansion time.

HRTimer

High resolution timer utilising C code via the Java Native Interface. This allows DAS to measure its own performance with much higher precision than allowed by Java library functions. Note that it utilizes Linux specific C code, and therefore sacrifices Java's natural portability.

A Windows version of this timer would be a trivial addition, and conditionally built into a dll (dynamic link library) instead of a so (shared object)

See [Appendix A] for benchmarks

GridUtil

Related Classes: DistanceCalculator, ManhattanDistanceCalculator, ChessboardDistanceCalculator

Utility class for checking the connectivity of a graph ("manhattan" or "chessboard"), and returning the required DistanceCalculator object for calculating d cheapest in the given domain.

FloatUtil

This is a simple utility class for comparing floating point values. It uses an epsilon value of 0.001 to eliminate floating point error.

Benchmarks

Map	Start	Goal	DAS 100ms	DAS 200ms	DAS 400ms	DAS 800ms	DAS 1000ms	DAS 1500ms	DAS 2000ms	Optimal	A* Time
Blasted Lands	85, 65	334, 468	3376	1509	1500	1450	1450	1450	1450	1450	1493
Blasted Lands	392, 85	289, 395	841	835	493	491	491	491	491	491	346
Blasted Lands	392, 85	42, 381	5053	1124	1146	1093	1074	1074	1074	1047	1947
Battleground	113, 91	459, 424	2730	2640	1111	1102	1100	1100	1100	1100	721
Donkey Kong	131, 59	460, 425	9818	7274	7274	7274	7286	6218	6326	2726	5238
Diamond 512	131, 59	460, 425	2275	2275	2275	1577	879	879	879	879	821
Diamond 512	256, 5	256, 507	2442	2442	2430	996	994	994	994	994	1216
Diamond 1024	512, 0	512, 1023	2949	2949	2943	2933	2931	2859	2787	1987	14677
Diamond 1024	637, 83	659, 931	2296	2204	2204	2184	2184	2114	2000	1539	5607
Bloodvenom	131, 59	460, 425	4510	2062	1625	1639	1447	1159	1148	1148	1500
Tranquil Paths	437, 353	136, 229	3692	2681	573	567	567	567	567	567	400

Table 1 Raw data for benchmarks

Blasted Lands 1	1	2.237243	2.250667	2.328276	2.328276	2.328276	2.328276
Blasted Lands 2	1	1.007186	1.705882	1.712831	1.712831	1.712831	1.712831
Blasted Lands 3	1	4.495552	4.40925	4.623056	4.704842	4.704842	4.704842
Battleground	1	1.034091	2.457246	2.477314	2.481818	2.481818	2.481818
Donkey Kong	1	1.349739	1.349739	1.349739	1.347516	1.578964	1.552008
Diamond 512 1	1	1	1	1.442613	2.588168	2.588168	2.588168
Diamond 512 2	1	1.005397	1.010362	2.46504	2.47	2.47	2.47
Diamond 1024 1	1	1	1.002039	1.005455	1.006141	1.03148	1.058127
Diamond 1024 2	1	1.041742	1.041742	1.051282	1.051282	1.086093	1.148
Bloodvenom	1	2.187197	2.775385	2.751678	3.116793	3.891286	3.928571
Tranquil Paths	1	1.377098	6.443281	6.511464	6.511464	6.511464	6.511464

Table 2 Normalised solution quality values

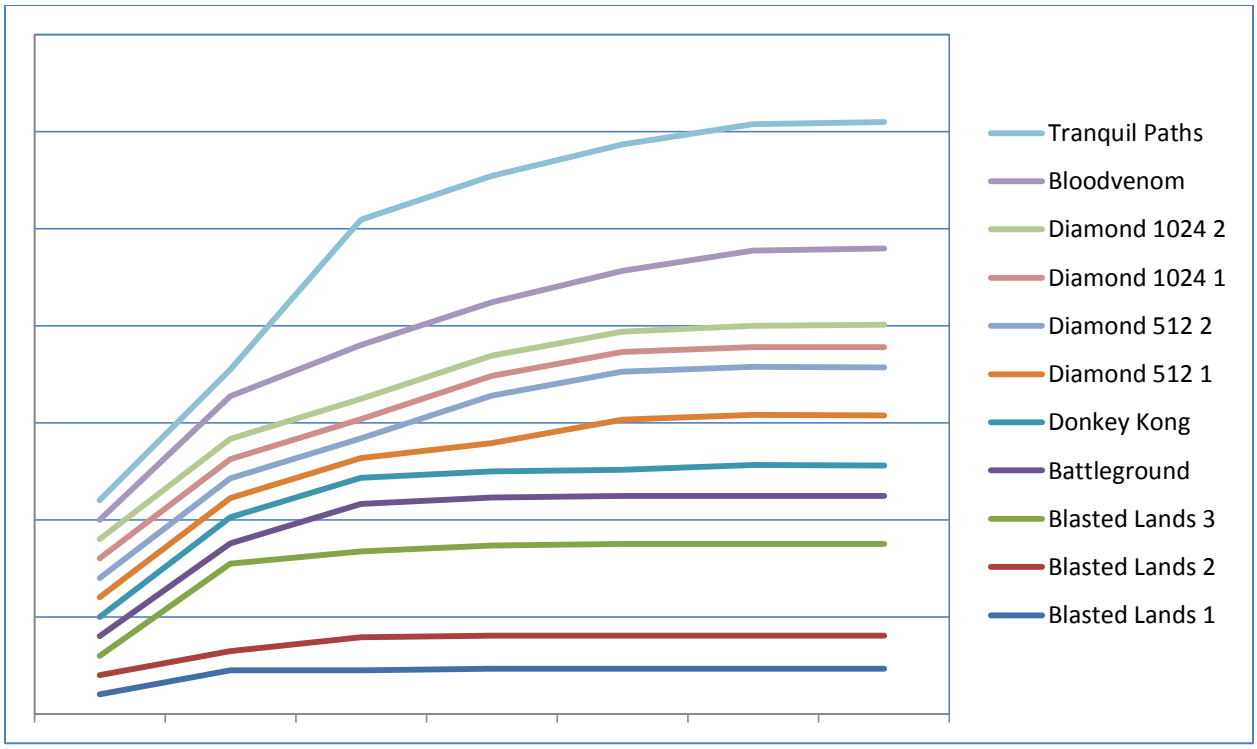

Figure 1 Normalised Solution Quality Improvement (y) vs. Deadline (x)

The above charts show the improvement of solutions over time for a variety of maps and deadlines.

In figure 1 each line represents a map, and the x axis represents the deadline given. The y-axis represents the normalised solution quality on relative to that particular map. We can see the intended gradual improvement of solutions as the available time for planning is increased.

Each of the samples flattens out, i.e. converges on optimality, which is surely a requirement of all time constrained sub-optimal planners.

It should also be noted that the DAS results are entirely non-deterministic, as the performance is measured in real time, and the algorithm responds in kind. One execution of a test will likely show different numbers to the next execution, so should be taken as representative only.

Potential Improvements / Investigations

End with Greedy Search

If the DAS search is about to end and has not found an optimal solution, a greedy search from the current head of the open list could be performed. This was not experimented with, as we deemed it unknowable when the search is 'about to end' and how much time to allot for the greedy search, in general terms. It was felt that this approach would not be robust, although it could dramatically improve the hybrid solution detailed previously.

Dynamic sorting of pruned/open lists based on time remaining

We did make initial efforts to change the sorting of the pruned list every time a solution was found by DAS. The sorting was changed by setting the weight of h to 1.0 + % time remaining.

This had minimal effect, and there was concern that it would impede upon the natural behaviour of DAS. It was also not clear how searches would still converge on optimality.

This could be a potential improvement if, instead of changing the weight w=1.0 + %time, we use a factor such as that proposed by Restarting Weighted A* [4], such that w= <factor> * (1.0 + %time), where <factor> is greater than 1.

Furthermore, we could apply this to both the open and pruned sets. This brings with it the extra overhead of resorting both priority queues, which is a comparatively expensive operation.

The disadvantage with this approach would be that it brings in all of the tuning issues associated with setting the weights that require customisation for and experimentation with each domain. This eliminates part of the benefit of DAS which is that it is domain-agnostic and does not require tuning of parameters.

Base the deadline on number of expansions, rather than time

In [1] it is suggested that an implementation might use the maximum number of expansions as a more stable / robust deadline metric. This would not be permissible in this application, as we have a real time limit, but it might provide some interesting results, and may help to normalise the performance measure.

Discussion & Conclusion

One of the key observations is that any form of greedy search is a good starting point for a sub-optimal search algorithm.

Whilst the initial solutions returned by the DAS algorithm do not improve upon the greedy solution, they do improve over time, which is entirely the objective of suboptimal search algorithms. The benchmarks show that in this implementation, the solutions do improve, given larger time allowances. Furthermore, they show that the search does converge on optimality, given enough time.

Unfortunately, as great as the algorithm sounds conceptually, the amount of time spent *measuring things* is possibly to the detriment of the DAS algorithm, at least in the example application, where the time limits are quite small. It is not certain if this is a flaw in the algorithm, or just in this Java implementation. The paper acknowledges that the measurement process itself becomes part of the measurements, so this was not unexpected by its writers. We have tried to counteract any potential negative impact incurred as part of performing the implementation in Java - namely, the C High Resolution timer, and using data-oriented design techniques.

Indeed the sample deadlines given by the paper to compare it with other algorithms *start* at 250ms, where it is less performant than its competitors. According to the paper, DAS only starts to outperform other suboptimal algorithms for deadlines over 1 second (1000ms) – See figure 2 in [1]. Granted, these numbers are specific to the writers' test platform, but the results seem similar.

Part of the issue experienced with DAS in this domain may be that the absolute value of d_max is often a degree of magnitude outside of the values of d^cheapest, to which it is compared (this concept was tested by printing these values at the time they are calculated). This means that every node is pruned. This is likely not the intent of DAS, and it causes its own vacillation, something that DAS is expected to curtail! As the paper [1] mentions "Depending on the given deadline, the value should typically fall somewhere within the range of current d^(s) values such that some pruning will occur when necessary and not all states will be pruned unnecessarily". It is not clear how to perform this normalisation practically, but it may be the necessary key to unlock DAS performance.

Beyond the recommendations and possible improvements offered in this report, we would greatly like to compare our results to comparable restarting weighted A* (RWA*) implementations on the same domain. An analysis of this comparison would be a good point by which to further understand the strengths and weaknesses of the DAS algorithm, and this implementation.

Appendix A - HRTimer Benchmarks and Instructions

See test harness within HRTimer class for the implementation of this benchmark.
We get the following output:
Average difference between two calls of timing function of 1000 runs
C: ***173*** *(getCurrentNanotime)*
ThreadMX: ***761*** *(getCurrentThreadCpuTime)*

This is a comparison of the average difference in system time retrieved via subsequent calls to the respective timing functions. We tried to reverse the order of the timing calls to check if cache-hits were responsible, but it made no observable difference.
What these numbers do not show is the variance, which from experimentation, is minimal for the C timer, and relatively high for the ThreadMX implementation.
For these reasons, the C Timer is used as it is always better than the ThreadMX implementation.

Compilation instructions for libHRTimer

```
export  JAVA_HOME=$(readlink -f /usr/bin/javac | sed "s:bin/javac::")
export  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.

gcc -I${JAVA_HOME}/include  -shared libHRTimer.c -o libHRTimer.so -fPIC -lrt
```

Note that -lrt needs to be at the end of the command!

Note also that libHRTimer.so needs to be in your LD_LIBRARY_PATH

A shared object for x86_64 GNU/Linux has been included with the delivery package.

Appendix B - Dynamic Maps

Some minor changes have been made to correctly handle dynamic maps.
Note that any change either to the map, goal position, or to the time available, will trigger a complete replan.

This is an assumption that has been made - if the map changes, the agent will be given sufficient time to create a new plan. Likewise, even if the map has not changed, but extra time is given, a complete replanning is performed.

This is an entirely simplistic approach to dynamic maps, but this has not been the focus in the development of the agent. This approach could provoke issues, for example if there is no change to the map, but 1ms extra is provided. We will perform an entirely fresh replanning operation! These conditions of when to replan are a different (albeit related!) class of problem, from that of search, one that we have not tried to solve beyond the simple case.

Note also that a static buffer of 20ms (*SEARCH_END_TIME_OFFSET*) is given to the DAS search to return the relevant data to Apparate. Any less than this on the sample maps provided, would occasionally leave too little time for walking the path. It is not clear what other methods could be used in place of this admittedly uninformed approach to setting a deadline.

Appendix C - Programmable Parameters

Although DAS is intended to be entirely self sufficient and make use of online learning, there are some parameters which may vary the performance of the system.

SEARCH_END_TIME_OFFSET : Time before the Apparate nominated deadline, by which a solution must be returned. This is specific to the Apparate framework, and should not be necessary in a general DAS implementation. By default it is set to 20000000 (ns).

SETTLING_EXPANSION_COUNT : After all open nodes are exhausted, a de-pruning operation occurs. Because this so drastically affects the make-up of the open set, our performance metric needs to be restabilised. This parameter is the number of expansions which are required for the system performance to be considered ‘stable’ once more. By default it is set to 10.

EXPANSION_DELAY_WINDOW_LENGTH : Related to *SETTLING_EXPANSION_COUNT*, this is the size of the sliding windows which are used to calculate average expansion delay and average expansion time deltas. By default, it is set to 10.

switchPrunedAfterSolutionFound : This is more of a development / experimentation tool, used to flag whether the first solution found triggers a switch of the sorting of the pruned list. By default it is enabled.

References

[1] Austin J Dionne, Jordan T Thayer, Wheeler Ruml, "Deadline Aware Search Using On-line Measures of Behavior" – Association for the Advancement of Artificial Intelligence, 2011

[2] Noel, "Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)" - http://gamesfromwithin.com/data-oriented-design

[3] Silvia Richter, Malte Helmert, "Preferred Operators and Deferred Evaluation in Satisficing Planning" – Association for the Advancement of Artificial Intelligence, 2009

[4] Silvia Richter, Jordan Thayer, Wheeler Ruml, "The Joy of Forgetting: Faster Anytime Search via Restarting" - Association for the Advancement of Artificial Intelligence, 2010