Path Planning Course Project

Assignment 3 - Report
Agent-Oriented Programming and Design

Pulkit Karwal
3360413

Contents

Introduction

The submitted agent uses Deadline-Aware Search (DAS) algorithm to perform efficient and time-conscious path planning. The agent is based on the Apparate platform, designed to run on Apparate using JPathPlan.

The algorithm for the agent, DAS, has been sourced from the paper [1] by Dionne, Thayer and Ruml.

The agent uses octile distance measure as heuristic, supported by a basic implementation of preferred operators.

Usage Instructions

The agent can be executed by calling `agents.MyCoolAgent` from the supplied JAR file (`MyCoolAgent.jar`), or from the source code archive (`MyCoolAgent-src.zip`).

The agent uses an additional class `agents.pulkit.GridCellInformed` for operation. The class acts as a `GridCell` extension allowing storage of DAS-specific parameters (costs and expansions), and includes a comparator for automatic tie-breaking.

The Algorithm

Deadline-Aware Search

Put forward by Dionne, Thayer and Ruml in [1], this algorithm adds a layer over vanilla A* to check for approaching deadlines. As the deadline closes in, the search tree gets constricted and more directed towards the goal, ultimately resulting in an "all-or-nothing" rush to find a solution within the time limit. Given that limited time is an important constraint for this assignment, I decided to adopt DAS for this agent.

Algorithm in Literature *Summarized From the Paper*	**Modifications** *Implemented in the Agent*
while **current time < deadline :** if open list is not empty : compute dmax node := best node in open list	Track the **average time** taken to execute one iteration. Don't run loop if this amount of time is not left. Prevents over-consumption of time.
if node is a **goal** and **better than previous** : goal node := node else:	Use **Greedy search** to compute a goal first.
if **dCheapest** < dmax : expand node else prune node	Use **Manhattan distance** for quick estimation.
else **recover pruned nodes**	Add **weights** to cost values as time runs out.

Specifically, the modifications are as follows:

Average Time Taken	**Greedy Search**	**Dynamic Weights**
Computed as the time elapsed between node expansions. This ensures that DAS doesn't enter a loop when it knows it won't have enough time to complete the iteration.	Not only does this act as a contingency solution, it is also used to test the time we need to reserve for generating a path object.	Weights are applied to the heuristic and actual costs (*h* and *g*) for increasing the influence of *h* as we run out of time. This hastens progress towards the goal.

Implementation

Data Structures

The agent uses the following data structures to handle the problem information:

Priority Queues

Priority queues offer quick add, retrieve and remove methods and are inherently sorted. By passing a comparable element, priority queues have all ties already broken at the time of node retrieval.

Open List – maintains the list of nodes yet to be examined

Pruned List – tracks the nodes that the DAS algorithm believes it won't be worth exploring

Fixed Arrays

For non-sorted lists, fixed arrays are unmatchable in efficiency. The agent initializes the arrays as per the map's width and height, in order to store any per-coordinate information.

Closed List – tracks all grid locations which have already been visited during search

agent.pulkit.GridCellInformed

A custom encapsulation of grid cells with a fixed number of extra fields, including costs (*f*, *g* and *h*), expansion counter *e*, and a reference to the parent cell.

Unlike `SearchNode`, `GridCellInformed` does not use variable-length arrays to hold these data, and hence, is more efficient.

`GridCellInformed` is used to store all nodes examined during the search.

The class implements a comparator, allowing Priority Queues to readily sort `GridCellInformed` objects.

Agent Design

The architecture of the agent is divided into two layers:

Core Search Layer

While the search is performed only once for a given problem state, it is carried out in two calls to the agent's `getNextMove()` function. The agents performs the following actions in those calls, respectively:

1. **Greedy Search**
   First, the agent generates an incumbent solution. The agent doesn't stop if the Greedy search can't find a solution.
   A path is generated and stored, and the time taken to do so is recorded.

2. **Deadline-Aware Search**
   Next, DAS is executed over whatever time is left (with some time reserved to generating the path from the solution, using the time obtained above).
   Each time DAS tries to recover pruned states, the weights over the costs are adjusted to further

propel the agent towards the goal.

Dynamic Map Response Layer

In order to cater to the varying nature of the problem, this layer is added on top of the core. It determines the action required (the degree to which we need to re-plan, for example), depending on the situation.

1. **Map has changed**
   If the changes affect the current path, the agent will begin to re-plan from its current position towards the goal.

2. **Goal has moved**
   If the goal has moved away from the agent, then ignore the change until we reach close to the goal (or alternatively, plan a path from the old to the new goal position and append it to the current path plan). The agent will, thus, need a much shorter time for re-planning.
   Else, all heuristic information is unusable now, and it is better the agent initiates a complete replan.

3. **Time has increased**
   In the event that the time allotted is increase without any of the above accompanying changes, the agent will take this opportunity to resume DAS where it left off, to try and improve the solution.

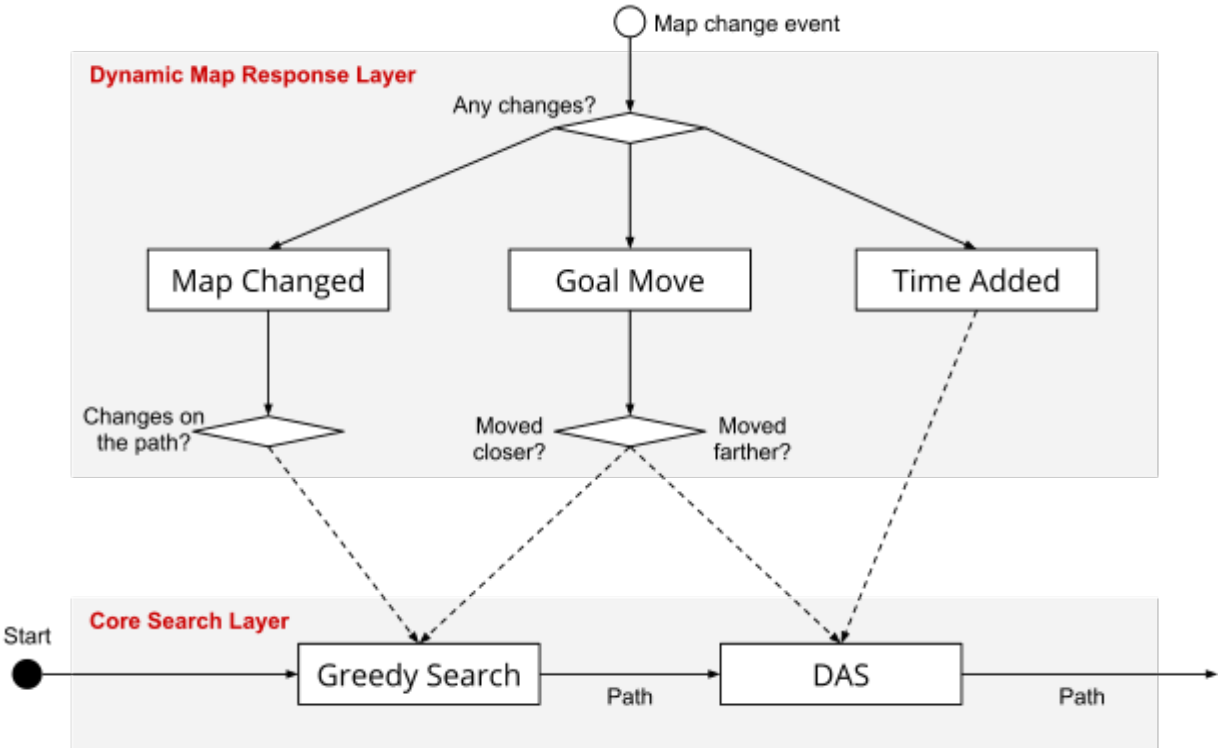

Figure 1: Agent Design

Tie-Breaking

Equally attractive nodes are distinguished using the *f*, *g* and *h* costs in the following order:

1. Weighted costs: *f*, then *h*, then *g*
2. Unweighted costs: unweighted *f*, unweighted *g* (if any)
3. Expansion time: *e* – allowing nodes that have been opened later to be given higher preference.

Heuristic and Weights

Heuristic is computed incrementally (see [2]), i.e., once the start node's heuristic value is known (using Apparate's inbuilt Manhattan/Euclidean heuristic estimate) the *h* costs for the children are computed by adding +1 or -1 to the parent's *h* value, depending on the direction of motion.

The directions are represented as operators each with a preference (**preferred operators**): any direction of motion in the general (and thus, preferred) direction of the goal is rated higher than otherwise. Having avoided recalculation of heuristic for every child node, this algorithm performs considerably faster.

To improve the performance of DAS, the concept of **automatically varying weights** has been adopted from Anytime Repairing A* (ARA) algorithm [3]. If the DAS has pruned everything and finds itself trying to recover pruned states, the agent adjusts the weights of *g* and *h* suitably for the next DAS iteration.

Weights increase automatically as time runs out.

$$w_h = initial\ time\ allotted\ /\ time\ left$$

$$w_g = 1 - time\ left\ /\ initial\ time\ allotted$$

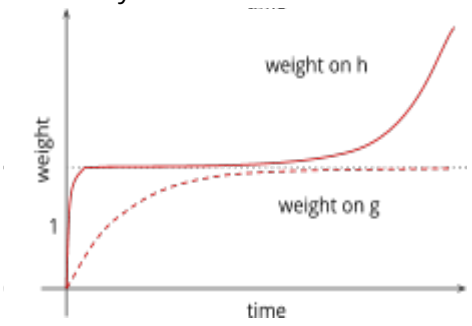

Figure 2: Gradually increasing weights

Tested over the Diamond map with fixed initial and goal positions, the impact of these dynamic weights can be observed and plotted:

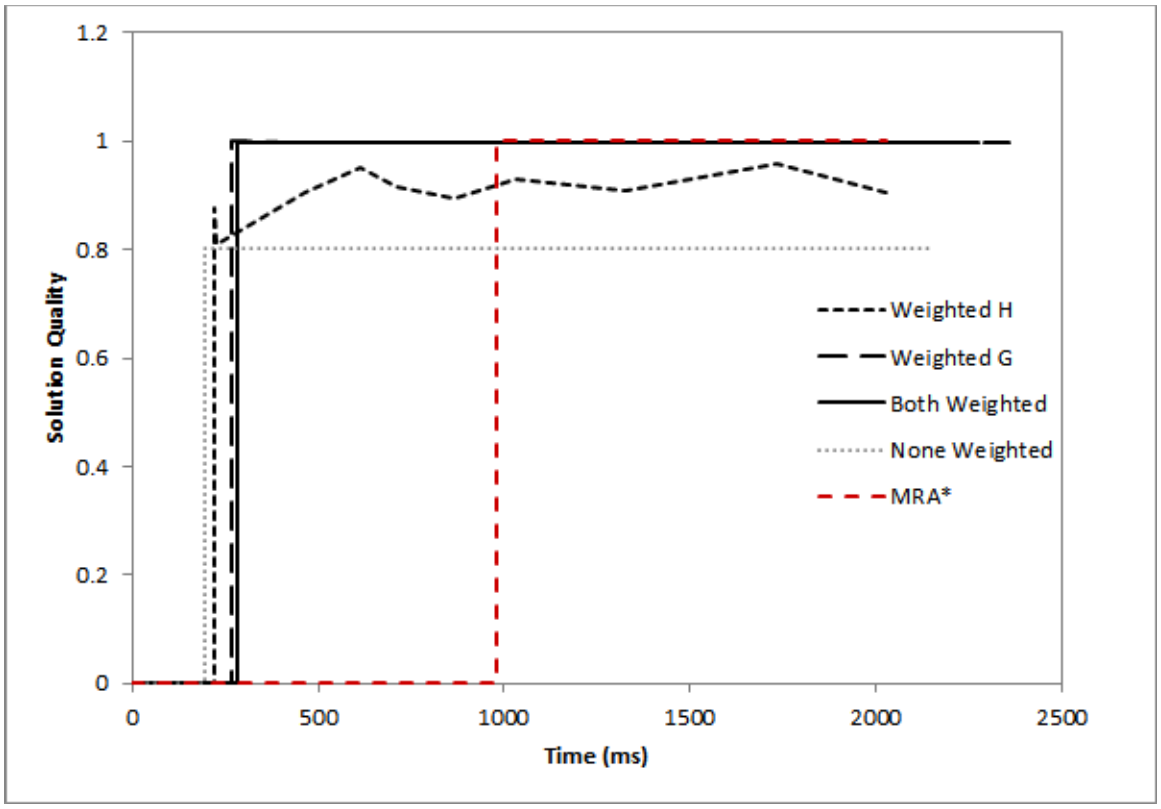

Figure 3: Impact of weighted costs. Adding weights on both g and h seems to be the most effective

Performance on Benchmarks

To test the agent's average performance on this map, it was run with several time allotments and the results depicted in a chart. Time allocations range from 50ms to 2500ms (2.5 seconds).

For each time allocation, the agent was executed 10 times and the costs were averaged. To assess the performance, the result is calculated as:

$$Solution\ Quality = \frac{Cost\ of\ Best\ Path\ by\ MyRestartAstarAgent\ (MRA)}{Average\ Cost\ of\ Path\ by\ MyCoolAgent\ (DAS)}$$

Solution Quality approaches 1 as the solution returned by the agent nears the best possible path.

Diamond Map

Benchmark: diamond-512.map with start position (256, 2) and goal(256, 510) on a Manhattan grid.

Results

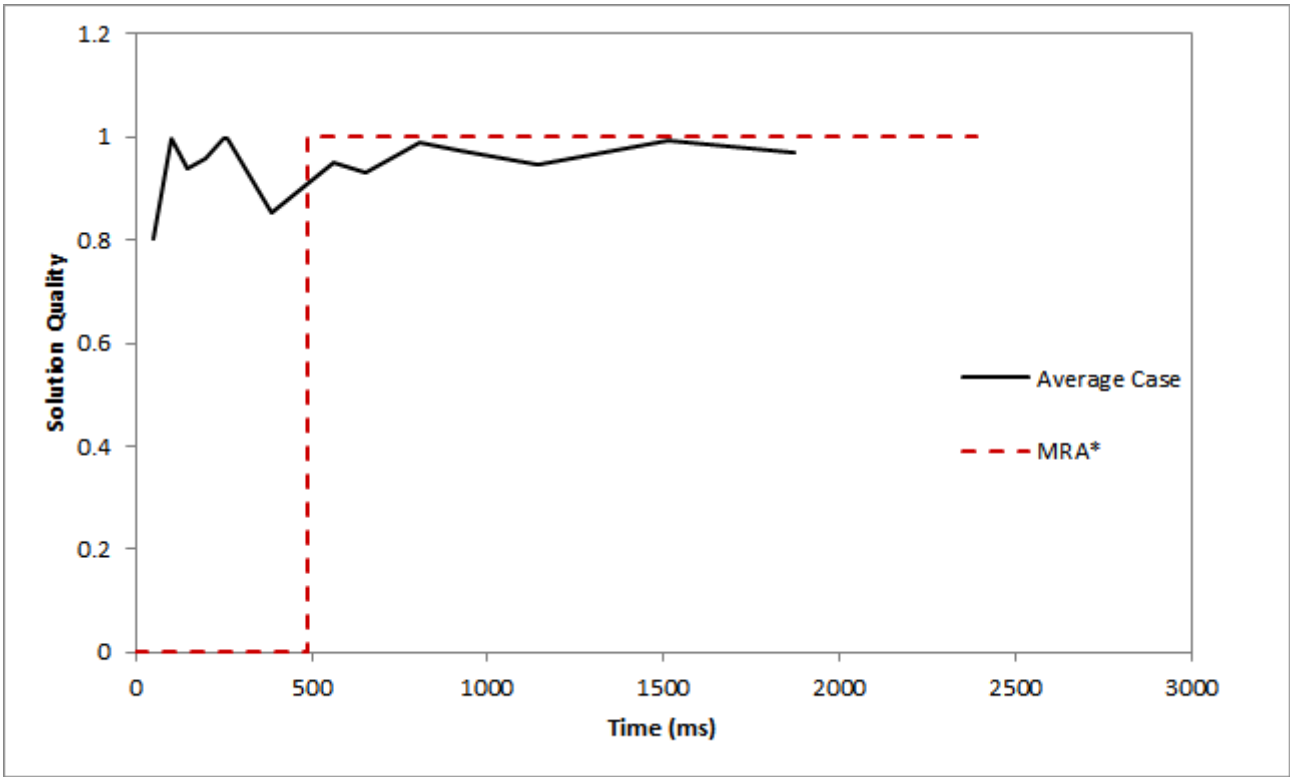

Figure 4: Agent performance on the Diamond Map is volatile

On average, the agent is able to return almost perfect solutions, with Solution Quality approaching 1. It is also able to find a solution in half the time taken by MyRestartA* agent.

Of note is the volatility of the average case graph showing that the agent gives widely varying performance results across the 10 runs of each time allocation. This could be attributed to the nuances of the testing operating system, or due to the weights, which are dynamically computed based on precise amounts of time left (in nanoseconds).

BlastedLands Map

Benchmark: blastedlands3.map with start position (103, 95) and goal (478, 422) on a Manhattan grid.

Results

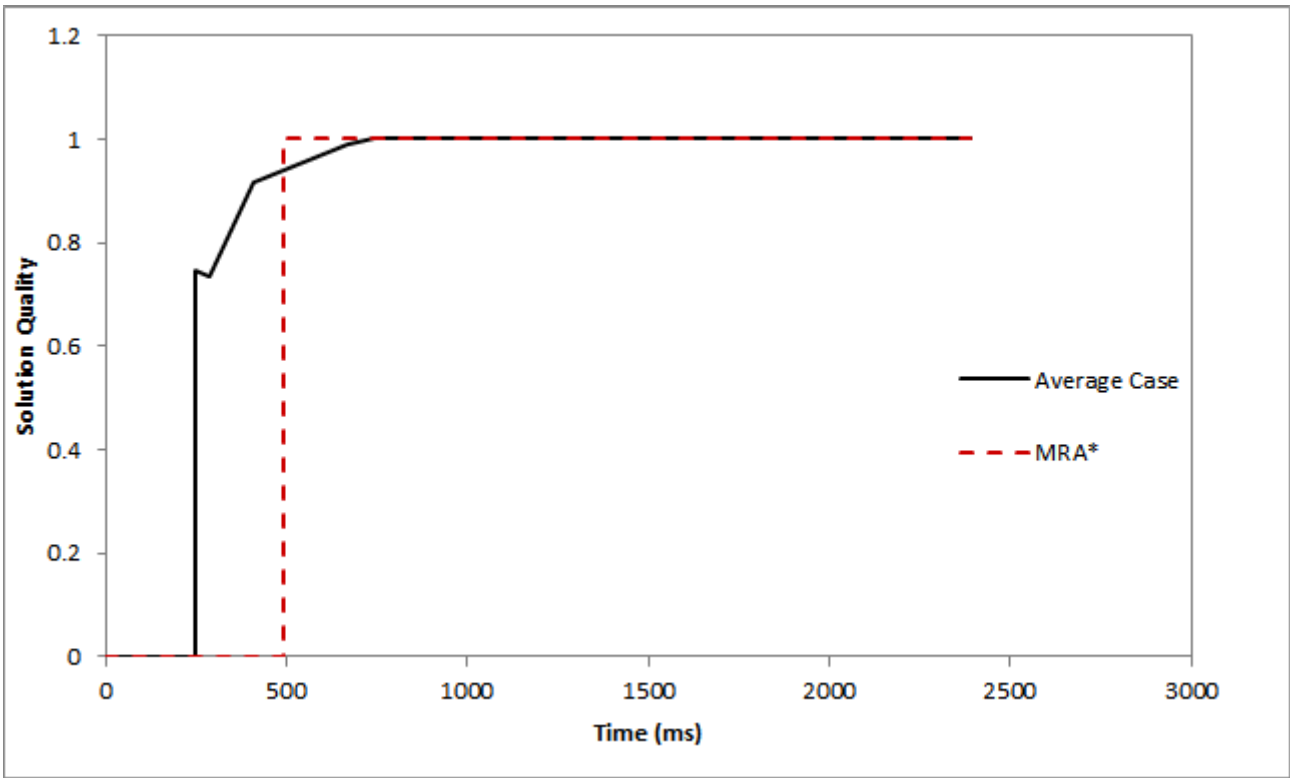

Figure 5: Agent performance on the BlastedLands map is near-optimal

The results are far more promising on this map. The average case approaches the optimal solution as determined by MyRestartA* agent.

It is interesting to note that Solution Quality lags a little in comparison with MyRestartA*.

For smaller deadlines, the amount of time left is fairly small. Thus, the value of the weight w_h grows large very early into the search process, resulting in a highly focused, good quality solution. On the other hand, with larger deadlines the weight computation has a negligible impact on *h*, thereby depriving the algorithm of strong focus towards the goal. This is augmented by the behaviour of DAS – with large deadlines, the estimates for d_{max} are too accommodating for any pruning to take effect.

Donkey Kong Map

Benchmark: donkeykong.map with start position (256, 2) and goal (256, 510) on a Manhattan grid.

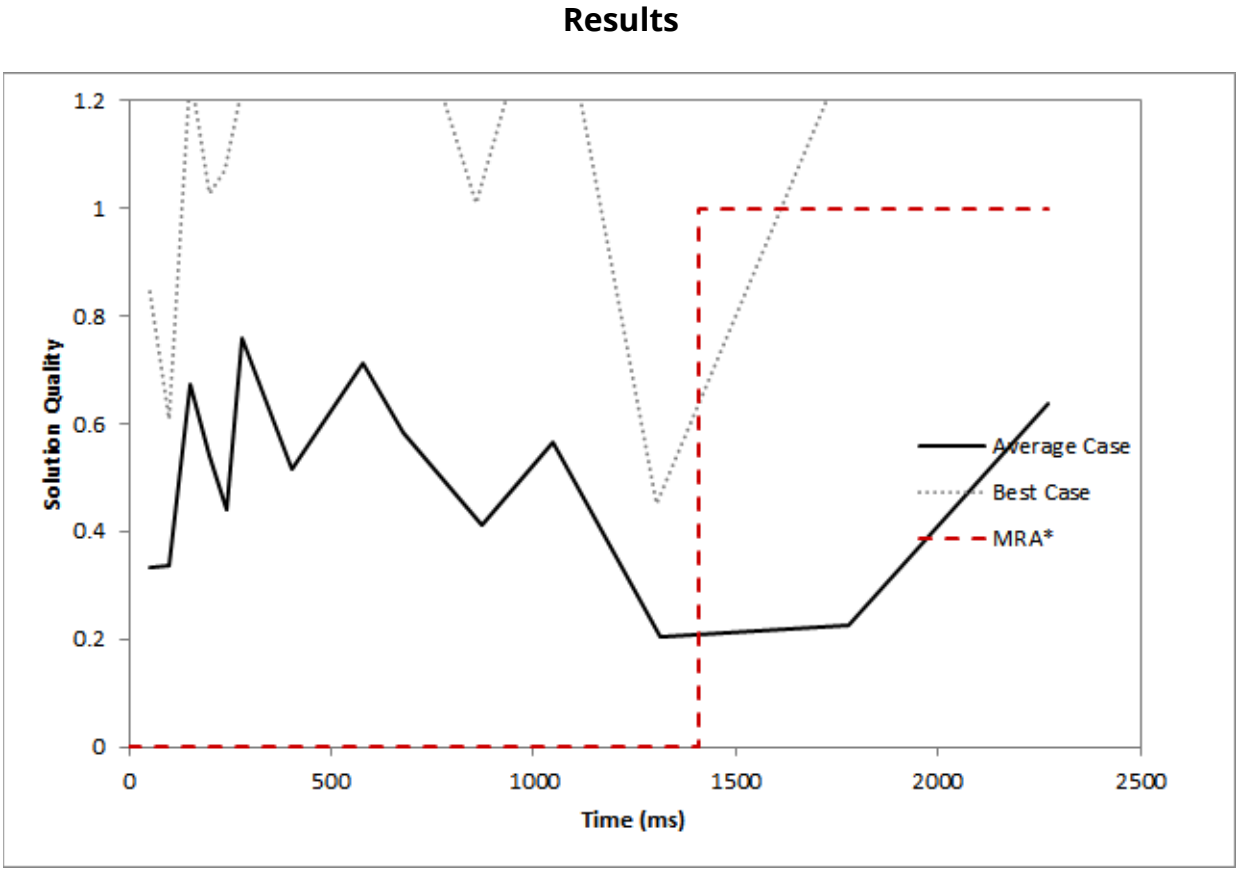

Figure 6: The agent performs poorly on this map, though the solution quality is rising

This map presents a special case where the agent's performance varies vastly over successive runs, as observable from the large gap between the average case and the best case graphs. Nevertheless, the best case shows a rising trend, and can be extrapolated to approach the optimal solution eventually.

Due to excessive pruning, and the increasing weight on *h* as time runs out, the search path tends to skip past the water channels, deciding instead to directly head to the goal. This is an intended behaviour of DAS.

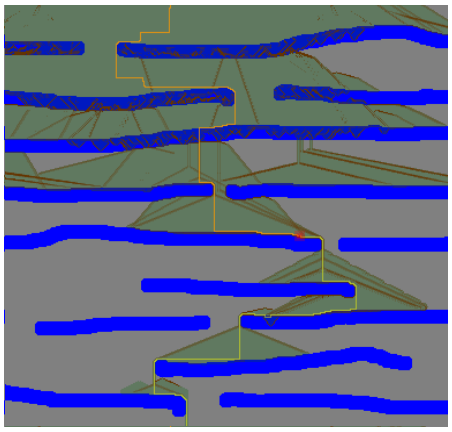

Figure 7: An illustrative solution produced by the agent on the DonkeyKong map (time allotted: 2000ms)

Conclusion

This was an excellent opportunity to study and implement DAS. Although the algorithm doesn't perform fantastically on the current domain, it is theoretically sound for dealing with problems where time plays a crucial role. Yet, as the authors of DAS note, the computation of some of the parameters (particularly d_{max}) needs to be experimented with for finding better solutions.

Much of the work on this agent went into getting this factor working optimally. An improperly computed value would result in large chunks of the search tree getting prematurely pruned. To work around this issue, I implemented weighted costs to add to the search's directed progress.

Rhys and Geoffrey were able to develop a much more efficient program that does not need weights to augment the search. My own implementation lacked the speed to be able to show appreciable results.

As is evident from my current and previous assignments, I lack the skills to make effective, speed-conscious algorithms with perfect choice of data structures. These tasks seemed to be primary area of focus for the last two assignments, as anyone with a lacklustre search algorithm obtained excellent results with carefully chosen data structures. In this assignment, I've tried to address that with improved data structures as far as I could. Nevertheless, I took this opportunity to learn the nuances of pathfinding algorithms, the difference between theoretical conjecture and practical application, and how minor aspects of problem domains can strongly impact performance.

Acknowledgements

Rhys and Geoffrey – For the numerous insightful discussions held to exchange ideas and delve deeper into the operation of DAS, and for the suggestion to replace dynamic SearchNodes with fixed-size arrays.

Rene – For the suggestion of preferred operators.

References

[1] Dionne, A. J., Thayer, J. T., Ruml, W (2011) *Deadline-Aware Search Using On-line Measures of Behavior*

[2] Burns, E., Hatem, M., Leighton, M. J., Ruml, W (2012) *Implementing Fast Heuristic Search Code*

[3] Richter, S., Thayer, J. T., Ruml, W (2010) *The Joy of Forgetting: Faster Anytime Search via Restarting*