Path Planning Course Project

Assignment 3 - Report
Agent-Oriented Programming and Design

Pulkit Karwal
3360413

Contents

Introduction

The submitted agent uses Deadline-Aware Search[1] (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.

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.

[1] DAS

The Algorithm

Deadline-Aware Search

Put forward by Dionne, Thayer and Ruml in [2], 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	Not only does this act as a contingency solution, it is also used to test the time we need to reserve for generating a	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

[2] sdfds

enough time to complete the iteration.	path object.	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 of 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.

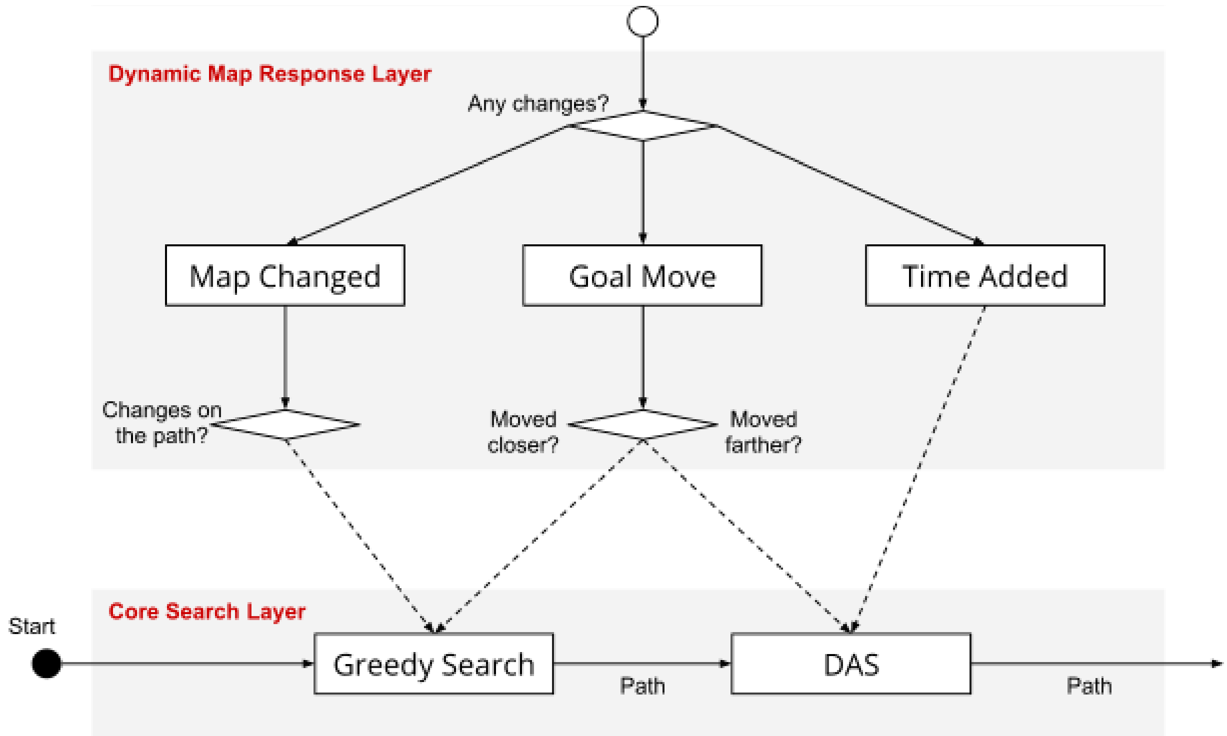


Figure 1: Agent Design

Dynamic Weights

To improve the performance of DAS, the concept of automatically varying weights has been adopted from Anytime Repairing A*[3] (ARA) algorithm.

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. The weights are applied to the costs as follows:

$$g = edgeCost + weightG * gParent$$

$$f = g + weightH * h$$

weightH
The weight depends on time and increases automatically as time runs out.

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

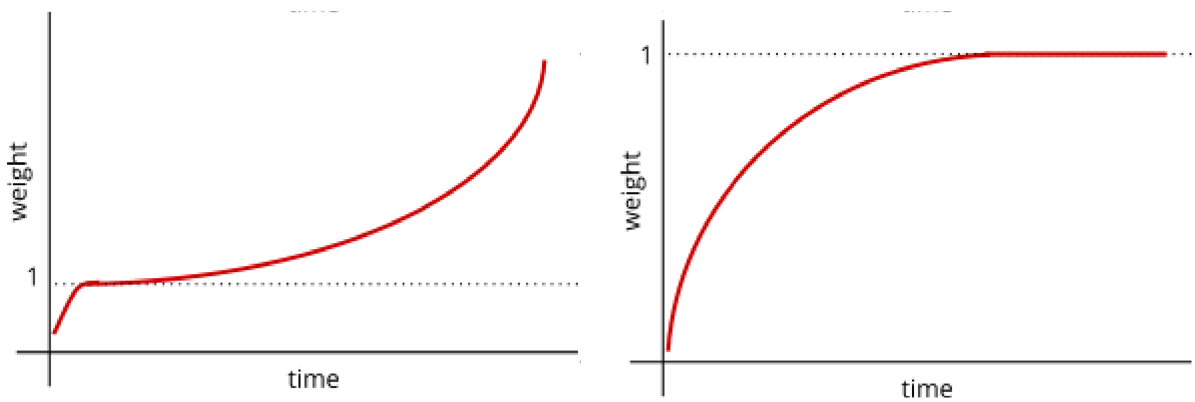


Figure 2: Gradually increasing weights for h (left) and g (right)

weightG
This weight serves to reduce the influence of the parent's *g* cost on the successor nodes. It is desirable to have a low influence initially, compounding the high influence of *h* (this prevents the agent from unnecessarily exploring vast areas at the start). As the algorithm progresses, the weight can be increased until it reaches 1. This allows the actual costs to become increasingly important and determine solution quality, as we near the goal.

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

Weights are reset to 1 each time a goal is obtained, so that the next round of search begins afresh and unbiased.

Performance

Diamond Map

[3] ARA

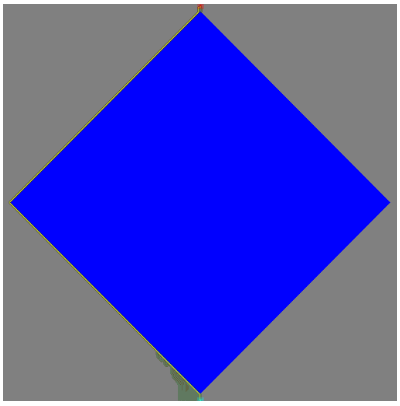
Benchmark

Map: diamond-512
Start position: (256, 2)
Goal position: (256, 510)

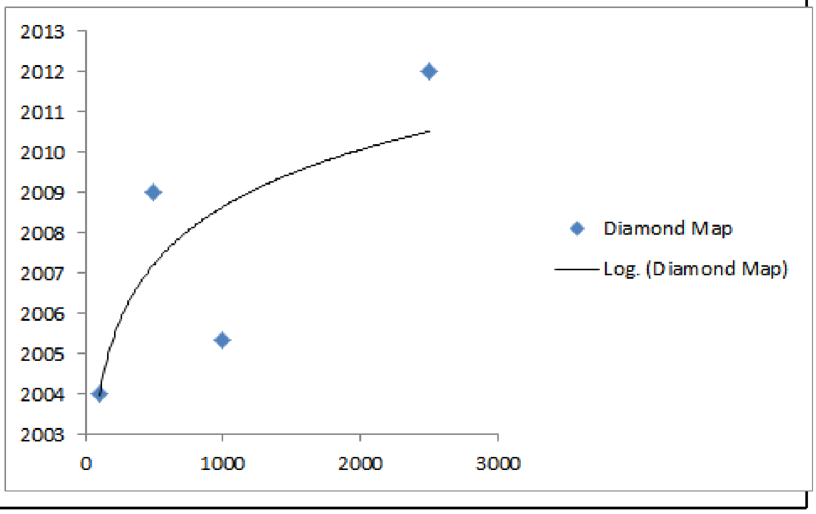
Results

Deadline	Time Taken	Cost
100	88ms	2004
500	353ms	2009
1000	831ms	2005
2000	2048ms	2012

Search Space



Graph



Donkey Kong Map

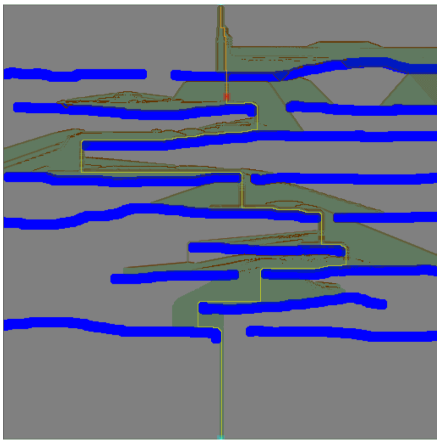
Benchmark

Map: donkeykong
Start position: (256, 2)
Goal position: (256, 510)

Results

Deadline	Time Taken	Cost
250	234ms	11894
500	395ms	8326
1000	831ms	8550
2000	2256ms	7919

Search Space



Graph