

Computer Games Development CW208

Technical Design Document

Year IV

|  |
| --- |
| Ben Millar  C00236772  20/04/2022 |

[Declaration form to be attached]

Contents

[Ben Millar 0](#_Toc101127201)

[C00236772 0](#_Toc101127202)

[20/04/2022 0](#_Toc101127203)

[[Declaration form to be attached] 0](#_Toc101127204)

[Technical Design 2](#_Toc101127205)

[Background 2](#_Toc101127206)

[Environments 2](#_Toc101127207)

[1-Dimensional Bin Packing 2](#_Toc101127208)

[1-Dimensional Knapsack 3](#_Toc101127209)

[2-Dimensional Bin Packing 3](#_Toc101127210)

[2-Dimensional Knapsack 4](#_Toc101127211)

[3-Dimensional Knapsack w/ Real Data 4](#_Toc101127212)

[References 5](#_Toc101127213)

# Technical Design

## Background

The purpose of this project is to determine both the viability and the efficacy of using deep reinforcement learning (DRL) to solve multi-dimensional bin packing problems. To achieve this, the codebase was split into TWO categories:

1. **Environment code**: This is the code that models the environment we are working with – a 1-dimensional bin packing environment for example – and will be contained in a *python (.py)* file.
2. **Driver code**:This is the code that instantiates the environments described above, and trains DRL models on the same. Each environment will have it’s own driver code, which will be contained in a *Jupyter Notebook (.ipynb)*.

Each of the environments examined are comprised of a pair of files as described above.

## Environments

I chose to approach this problem with an iterative solution, gradually increasing the complexity of problem at hand and bringing it closer to the end goal. I will begin by describing the implementation of the simplest environment and from there, for the sake of brevity, I will describe only the delta from one environment to the next.

### 1-Dimensional Bin Packing

#### Overview

This environment aims to model a 1-dimensional bin packing problem in which we have some number of bins, or containers, each with some non-negative integer capacity.

**[IMAGE NEEDED]**

At each time step, an item is generated of some size between 0 andour maximum capacity(although typically, this was between 1 and ) and we may take one of the following actions

1. Discard the item without placing it.
2. Place the item in one of our bins, thus reducing its remaining capacity by the size of the item.

The environment will terminate once we have filled each of the bins to capacity (or to such a capacity that they cannot take any more items of the sizes available), or we have gone over some maximum number of steps (1000 for testing, with 10 bins of capacity 20).

#### Representation

The environment was modelled as a NumPy **[citation needed]** array of size *n+1*, where *n* is the number of bins. Indices 0 → *n-1* contained integers representing the remaining capacity of each bin, and index *n*contained an integer representing the next item to be placed.For example, if we had 5 bins of capacity 10, and the next item to place was of size 3, then our array would be as follows:

The action we take on this environment, then, is denoted by an integer value in the range of our array length. If the action value is in the range then the item will be placed in the bin at that index. Otherwise, if the action value is equal to *n,* then we will discard the value to be placed and choose a new one.

It should be noted that if we attempt to place an item in a bin which does not have sufficient capacity, then that action will have no effect.

Let us take the example above and say we take the action value *2*. This will place the item into the bin at index position 2 (third element), reducing its capacity accordingly.

A new item will then be created with a value in the range specified; in this case, .

This will continue until all bins have a capacity below the minimum item value (<1, in this case).

#### Implementation

### 1-Dimensional Knapsack

#### Representation

This environment is modelled as a 2-dimensional NumPy array of size *n+1*, taking the form:

where the top row represents the remaining capacities of each knapsack, and the bottom row represents the total value contained within each. As before, the final index of both arrays contains the next item to be placed.

#### Implementation

### 2-Dimensional Bin Packing

#### Representation

This environment is modelled as a 2-dimensional NumPy array of size *n+1*, like the 1D knapsack, but transposed such that each element of the outer array is itself an array of length 2: [x, y]. The final index, as before, holds the next item to be placed.

The behaviour of this environment is much like that of the 1-dimensional bin packing environment, except that each item has a dimensionality of 2 and thus must satisfy the condition

#### Implementation

### 2-Dimensional Knapsack

#### Representation

#### Implementation

### 3-Dimensional Knapsack w/ Real Data

#### Representation

#### Implementation

The purpose of this document is to communicate effectively the technical details and design decisions of the system/algorithm to the readers.

It could include software architecture, algorithm design, class specifications, pseudo code, etc. with tools such as UML, Class Diagram, CRC Cards.

# References

Book

Author(s) - family name, initials. (Year). *Title of book.* Edition. Place of publication: Publisher.

[1] Gregory, J. (2014). *Game Engine Architecture.* CRC Press.

Report

Author(s) - family name, initials. (Year). *Title of report.* Edition. Place of publication: Publisher. (Series and vol./no.).

[2] Burrows, D., *et al.* (2012). *Global Trends: Alternative Worlds.* Washington: National Intelligence. (Office of the Director of National Intelligence).

Web-site

Author(s) - family name, initials. (Year, month day). *Title of document.* [Online]. (URL). Place of publication: Publisher. (Date accessed).

[4] de Valk, J. (2014, May 12). AI for Games. [Online]. (URL https://www.somesite.com/articles/1234/). (Accessed 12 February 2014).