DnD-like game using GOAP

Emmanuele Villa

January 2022

Contents

1 Introduction

This relation will describe the implementation phases and details of a dnd-like game: a turn-based **tactical rpg** played on a grid map.
For each part of the project, there will be a description of the **problem**, the proposed **solution** in pseudo-code and the benchmark and **optimization** of the implemented solution.

2 Project Overview

The project codebase is splitted in two C# solutions, one containing the **Unity3d scripts**, and one containing **game logic**, unit tests and benchmarks.

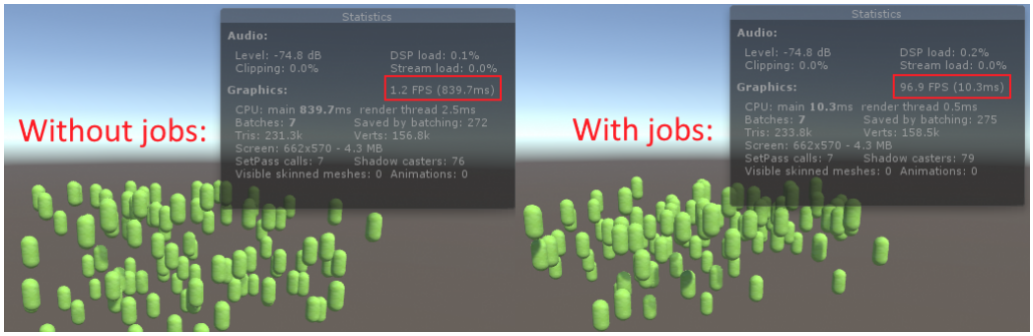
- The logic solution is composed by a **.NET Standard 2.0 library** that is copied in the Unity project folder after the build, a **.NET 5.0 Nunit** project, that contains all the unit tests related to the Core library, and a **Console Application** project used for benchmarking the algorithms. All projects have the Unity3d engine as a dependency to enforce decoupling of logic and UI.
- The unity3d solution is the standard Visual Studio solution created by the Game Engine

2.1 Communication between the Core and Unity3d

As per the project structure, the Unity3d project is not referenced in the Core, but the Unity3d scripts can freely access all public classes and methods of the Core library.
The problem is that these methods are always called from the **main thread**, and if they are too slow, they will lock the frame until their conclusion, dropping the frame rate.
To avoid this problem without relying on the Unity coroutines system in the Core library, Unity3d offers a **Job system**.

2.1.1 The Job system

From the Unity3d documentation: *"The Unity C# Job System lets you write simple and safe multi-threaded code that interacts with the Unity Engine for enhanced game performance."*
Long story short, it's a simple system where you create a Job that behaves like a **new thread**: it executes without blocking the main thread, in background, and you can wait for its completion.
In addition, jobs can be scheduled, ran in parallel or one after another with a **dependency system**.



2.1.2 Job example

Jobs are created in the Unity3d project and they act as bridges from the engine to the Core library.
For example, let's have the following in the Core library:

```
public int TimeConsumingMethod() {
    var sum = 0;
    for(int i=0; i<100000; i++)
    {
        for (int j = 0; j < 100000; j++)
        {
            sum += i*j;
        }
    }
    return sum;
}
```

This method will for sure lock the frame for some seconds, so we'll declare a job to handle it. First thing first, a job is declared as a **struct** that implements the **Execute** method of the **IJob** interface, and a public **NativeArray** where the execution result will be put.

```
struct VeryTimeConsumingJob : IJob
{
    public NativeArray<int> result;
    public void Execute()
    {
        var executor = new VeryTimeConsumingClass();
        result[0] = executor.VeryTimeConsumingMethod();
    }
}
```

Now that the job is ready, we need to schedule it and wait for its result, avoiding to lock the thread with the usage of the Unity3d Coroutines:

```
IEnumerator StartJob()
{
    NativeArray<int> result
        = new NativeArray<int>(1, Allocator.Persistent);

    // Set up the job data
    var jobData = new VeryTimeConsumingJob
    {
        result = result
    };

    // Schedule the job
    JobHandle handle = jobData.Schedule();

    // Wait until it's completed
    while (!handle.IsCompleted)
    {
        yield return null;
    }

    // Free the access to the result
    handle.Complete();

    // All copies of the NativeArray point to the same memory,
    // so we can access the result in our copy of the NativeArray
    int res = result[0];
```

```
        // Free the memory allocated by the result array
        result.Dispose();
}
```

3 The map

The game will be played on a grid map, where each cell has two parameters: **terrain** and **height**:

- **Terrain** represents the type of the terrain in that cell of the grid, which can be Grass, Water, Stone and so on
- **Height** represents the height of that cell, transforming the 2d terrain information to a 3d representation of the map.

3.1 Map editor

Before thinking about which data structure will represents our map in the code, we'll "develop" a simple map editor using the best map editor of all time: **Google sheet**.
Google sheet isn't only super fast and simple to use, but it can also help us visualize the map using the conditional formatting on the cells.
After creating the map on Google sheet, we'll download it in a **csv** format (but saving the file in .txt because Unity3d's TextAsset inspector field only accept *.txt files) and we'll have it ready for parsing.
In the next image, you can see an example of a map created in this way and fed to the Core library:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	G9	G9	G6	G6	G6	G7	G6	G6	G7	G9	R0	R0	G7	G7	G7	G9	G9	G9	G9	G9	
2	G9	G9	G6	G6	G6	G6	G6	G6	G7	G9	R0	R0	G4	G7	G7	G6	G6	G6	G9	G9	
3	G9	G9	G6	G6	G6	G4	G4	G6	G7	G9	R0	R0	G4	G4	G4	G6	G6	G6	G9	G9	
4	G9	G9	G6	G6	G6	G4	G4	G5	G5	G3	R0	R0	G4	G4	G4	G4	G4	G6	G9	G9	
5	G9	G9	G4	G4	G4	G4	G4	G5	G3	G3	R0	R0	G4	G4	G4	G3	G3	G5	G9	G9	
6	G9	G9	G4	G4	G4	G3	G3	G3	G3	S4	S4	S4	S4	G4	G4	G3	G3	G3	G3	G3	
7	G9	G5	G4	G4	G4	G3	G3	G3	G3	R0	R0	R0	G4	G4	G4	G3	G3	G3	G2	G2	
8	G3	G3	G3	G3	G3	G3	G3	G3	R0	R0	G1	R0	G2	G2	G2	G3	G3	G2	G2	G2	
9	G3	G3	G3	G3	G3	G3	G3	G3	R0	G1	G1	R0	G2	G2	G2	S2	G3	G2	G2	G2	
10	G3	G3	G3	G3	G3	G3	G3	G3	R0	G1	G1	R0	R0	G2	G2	S2	R0	R0	G2	G2	
11	G1	G3	G3	G3	G3	G3	G3	R0	R0	G1	G1	R0	R0	R0	R0	S2	R0	R0	R0	R0	
12	G1	G3	G3	G3	G3	G3	G3	R0	G1	G1	G1	G1	R0	R0	R0	S2	G1	G1	G1	G1	
13	G1	G2	G2	R0	R0	R0	R0	R0	G1	G1	G1	G1	G1	G1	G1	S2	G1	G1	G1	G1	
14	G1	G2	G2	R0	R0	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	
15	G1	G1	G1	R0	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	G1	
16																					

Each cell contains a string with **two chars**, the first one representing the **terrain type** (G=grass, R=river, S=stone) and the second one representing the cell's **height**. By using conditional formatting with colors, we can clearly visualize both values at glance.

3.2 Map data structure

Many things can be done to optimize the map size in memory: taking the previous map as an example, we can notice that the majority of the tiles have type "G", so it could be intuitive to take

"G" as the base value for the terrains of the map, and retain in memory **only the differences**.
However, let's have a look at the simplest way in which we could encode the map: **a matrix of cells**.
Given the following structure for the cell (which is only the basic information that will be enhanced):

```
public class CellInfo
{
    public string Terrain { get; private set; }
    public int Height { get; private set; }
}
```

A **15x20** map will use (15*20) * (4 + 2) = **1800 bytes** in a multidimensional array.
Given the fact that in the sample map there are 250 cells with the "G" value, by storing them only once we could reduce the amount by (249 * 4) = **996** bytes, effectively halving the memory usage of the map.
However, aside from the fact that 1Kb is a derisory amount of memory in the Unity3d environment (this reasoning is just an example), what really matters is the speed of the algorithm, and even if both memory access (in Arrays and Dictionaries) are **O(1)**, the time needed to access the value can vary a lot based on the **language** we are using and the underlying **implementation** of arrays and dictionaries
As an example of that, we've implemented two different **IMap**: one that uses a multidimensional array and one that uses a Dictionary (respectively **ArrayDndMap** and **DictionaryDndMap** in the code) and compared the time needed to:

- Create a map with size 25x25
- Read all cells 1000 times

And here are the results:

Method	Mean	Error	StdDev
ReadDictionaryMapValues	3.105 ms	0.0062 ms	0.0055 ms
ReadArrayMapValues	1.506 ms	0.0045 ms	0.0037 ms

The solution with the Dictionary is **two times slower** than the multidimensional array solution: even if the time spent for a single run is just 1.5 microseconds vs 3 microseconds, we can say that it's not worth it to implement some memory optimization that will worsen the runtime execution and make the code way less readable, since this particular method may be called **hundreds of time** in a single search run.
What we can do fast and with no cons is change the **class** to a **struct** (which gives us some speed in other instances), change the **int** value to a **byte** to decrease the space from 4 to 2 (giving us a straight 30% reduction on the total size) and change the **String** field to a **char**, which retains the same used space but who knows if the compiler will use this information for some internal optimizations. Also, changing the **properties** to simple **fields** will give us a bit of speed while accessing it.
Unfortunately, best practises don't go well with optimization!
So, the final structure of the map will be a multidimensional array with:

```
public struct CellInfo
{
    public char Terrain;
    public byte Height;
}
```

3.3 Map UI

The map in the game will be drawn using a free set of isometric tiles on the Unity Asset Store, named **2D Pixel Art - Isometric Blocks**.
This is the in-game representation of the previous google sheet map:

4 Position in map

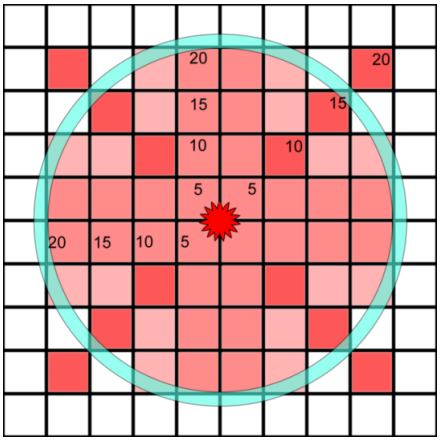
Characters position in the map will be described with a **list**, composed by:

- The **character** information
- The position in the form of a **CellInfo**; To support the addition of flying characters in the future, the CellInfo contained in the pair will be a copy of the one in the map, so that the "height" value could change without affecting the map.
- A list of **CellInfo**, representing the **threat area** of the character, meaning all the cells that they can reach with a melee attack. Caching this value will help us optimize the movement algorithm described in the next sections.
- Another list of **CellInfo**, representing the cells occupied by the creature. Since we can have creatures that occupies more than one cell, we'll only occupy the top-left cell (namely the **root-cell**) of the creature in the map array, and store the others apart.

5 Movement

5.1 Speed

From the D&D 5e player's handbook: *"In combat, Characters and Monsters are in constant motion, often using Movement and position to gain the upper hand.*
On Your Turn, ***you can move a distance up to your speed****. You can use as much or as little of your speed as you like on Your Turn, following the rules here.*
Your Movement can include ***Jumping, climbing, and Swimming****. These different modes of Movement can be combined with walking, or they can constitute your entire move.*
However you're moving, ***you deduct the distance of each part of your move from your speed*** *until it is used up or until you are done moving.".*
This speed -in feet- is always multiple of 5 (which is the size of a square), so we'll divide the speed by 5 to obtain the speed in squares, and store that value.



5.2 Special type of movement

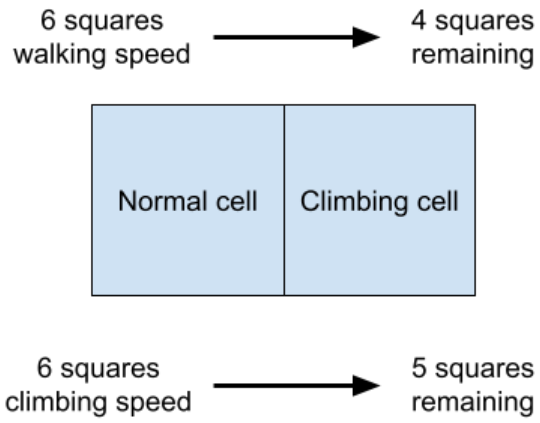
There are special types of movements require extra speed to be crossed, and will be handled in the project.
From the D&D 5e player's handbook: *"Combat rarely takes place in bare rooms or on featureless plains. Boulder–strewn caverns, briar– choked forests, treacherous staircases—the setting of a typical fight contains Difficult Terrain.*
*Every foot of Movement in Difficult Terrain costs **1 extra foot**. This rule is true even if multiple things in a space count as Difficult Terrain.*
*While climbing or Swimming, each foot of Movement costs **1 extra foot** (2 extra feet in difficult terrain), **unless** a creature has a climbing or Swimming speed."*

5.3 Calculating Graph Edges

In this game we don't need to find the path from a starting cell to a goal, but given a list of speeds for a character and his position in the map, we need to calculate every cell that they can move in using that speed.
To calculate this we'll use a modified Uniform Cost Search algorithm, but before that let's dive in in how the graph for the algorithm is created: since every character will have different connections and node weights based on their abilities and speed types we need to re-calculate the graph every time the game needs it, that is to say when a PC wants to move, or when an NPC needs that information to elaborate their strategy.

5.3.1 Needed speed calculation

To **create the graph** to be used in the search algorithm, we first need to calculate the speed needed to go from one cell to an adjacent one: we'll have a function that takes the **character's speeds** and **two adjacent cells** as an input and returns the amount of speed that must be consumed to travel to there. If it returns null, then there's no edge between the cells.
There's no need to explicit the speed type in the return type, because **all speeds are decremented regardless of the used type**.
For example, let's assume that character A has only 6 normal speed and must travel to a climbing cell. It uses 2 speed and 4 speed remains available. Now, character B has 6 normal speed and 4 climbing speed and must travel to a climbing cell. Since they have climbing speed, they only use 1 unit of it, and remains with 5 normal speed and 3 climbing speed.



The pseudo code that handles this cases would be:

```
int getSpeedUsed(Start, Destination, Player) {
    If Destination.isOutsideMap() {
        Return Null
    }
    int amount = 1;
    If Destination.isTooHigh() {
        If Player.hasClimbingSpeed {
            amount += ((Destination.Height - Origin.Height + 1) / 2) - 1
        } else {
            amount += Destination.Height - Origin.Height - 1
        }
    }
    if(Destination.requiresSwimming && Player.cantSwim) {
        amount++;
    }
    return amount;
}
```

5.3.2 Moving around other creatures

The previous algorithm must be enhanced, because there are special rules for when you move around other creatures:
"You can move through a non-hostile creature's space. In contrast, you can move through a Hostile creature's space only if the creature is at least two sizes larger or smaller than you.
Remember that another creature's space is Difficult Terrain for you.
Whether a creature is a friend or an enemy, you can't willingly end your move in its space.
If you leave a Hostile creature's reach during your move, you provoke an opportunity Attack."

To handle this rule, the only information about the amount of speed needed is no more sufficient, we need to enhance it:

```
public struct Edge
{
    public CellInfo Destination;
    public int SpeedUsed;
    public float DamageTaken;
    public boolean CanEndMovementHere;
}
```

The complete algorithm will look like this:

```
Edge getEdge(Start, Destination, Player) {
    If Destination.isOutsideMap() {
        Return Null
    }

    If Destination.isOccupiedByEnemyWithSimilarSize(Player.size) {
        Return Null
    }

    float damage = 0;
    If Destination.isTooLow() {
        damage += CalculateFallDamage(distance)
        amount = heightDifference
```

```
    }

    Bool canEndMovementHere = Destination.isNotOccupied()
    int amount = 1;
    If Destination.isTooHigh() {
        If Player.hasClimbingSpeed {
            amount += ((Destination.Height - Origin.Height + 1) / 2) - 1
        } else {
            amount += Destination.Height - Origin.Height - 1
        }
    }
    if(Destination.requiresSwimming && Player.cantSwim) {
        amount++;
    }
    if(Destination.isOccupied()) {
        amount++;
    }

    For(enemy in enemies) {
        If (character.leftReachOf(enemy, Start, Destination)
            && character.notDisangaged()
            && enemy.hasReaction()) {
                damage += enemy.Damage(character)
            }
    }

    return Edge(Destination, amount, damage, canEndMovementHere)
}
```

With this code we can handle the movement of all creatures that occupy one square of the map from one cell to an adjacent one, but what about creatures with bigger size?
First of all, we need to transform the Start cell in a list of all the cells occupied by the creature.
The algorithm works only with two input cells, Start and Destination, that are adjacent. This means that their difference in coordinates, named "delta", is between -1 and +1 on both axis.
We can apply this delta to all the Start cells in the list, and obtain in this way a Destination cells list.
After safely remove the Destinations cells that are also contained in the Start cells list, we can invoke the getEdge method on the remaining Start-Destination pairs and merge the resulting edges in a new edge that contains the worst cases.
In this way, I can check if one of the cell is entering difficult terrain, or it's leaving the threatening area of an enemy:

```
Edge getEdge(Start, Destination, Player, Map) {
    If(Player.CellSize < 2) {
        return getEdgeInternal(Start, Destination, Player)
    }
    Starts = Map.GetCellInRect(Start.Coord, Player.CellSize)
    Delta = Start.Coord - Destination.Coord
    Destinations = Map.GetCellInRect(Start.Coord + Delta, Player.CellSize)
    Destinations.RemoveIf(cell => Starts.Contains(cell))
    Edges = EmptyList()
    Foreach(D in Destinations) {
        Edges.Add(getEdge(D - Delta, D))
    }
    If(Edges.AnyNull()) {
```

```
        return null
    }

    return new Edge(
        DamageTaken = Edges.MaxDamageTaken
        SpeedUsed = Edges.MaxSpeedUsed
        CanEndMovementHere = Edges.AllCanEndMovementHere
        Destination = Destination
    )
}
```

5.4 Custom Uniform Cost Search algorithm

As mentioned previously, we need to search the map graph until we find all cells that are in the reach of the creature movement speed.
To do so, we'll assume that all cells are connected to the 8 surrounding cells and we'll calculate the edge cost during the search, since it cannot be cached between searches for too many variables changes between creatures and between turns.

```
List<Edge> Search(Creature, Start, Map) {
    Result = ListOf()
    Visited = ListOf(Start)
    Queue = ListOf(Start)
    while(Queue.NotEmpty() {
        Current = Queue.GetCellWithLessCost()
        RemainingMovement = Creature.Movement - Current.UsedMovement
        Foreach(Neighbor in Current.Neighbors) {
            If(Visited.Contains(Neighbor) {
                Continue
            }
            Edge = getEdge(Current, Neighbor, Creature, Map)
            If(Edge != null) {
                Edge.SpeedUsed += Current.UsedMovement
                Edge.DamageTaken += Current.DamageTaken
                Result.Add(Edge)
                Queue.Add(Neighbor)
                if(Edge.DamageTaken == 0) {
                    // if I took damage going there,
                    // I want to try again from another cell
                    Visited.Add(Neighbor)
                }
            } else {
                Visited.Add(Neighbor)
            }
        }
    }
    return Result
}
```

The neighbors are visited first horizontally and vertically, and then in diagonal. In this way we always prefer a straight path

5.4.1 A Last edit

Since we don't have a target cell and we'll need to be able to invert the path for every cell in the list that we found, it will be too performance heavy to make an algorithm that retrieves the inverted path for a given cell. Instead, we'll build the paths during the same search, by remembering all the path in the resulting edges.
However, we don't want to load too much the "getEdge" function: it will still return a "base" edge, and then the search algorithm will enhance it to a "memory" edge after having verified that it's valid.
So we just need to change the row:

```
Result.Add(Edge)
```

to

```
Result.Add(EnhancedEdge(Edge, PathUntilHere))
```

So for example, if the previous version returned as a result: (a,b) (b,c) (c,d)
Now, it will return ([a], b) ([a,b],c) ([a,b,c],d)
The "PathUntilHere" part is made by a list of "MovementEvents" that not only resumes the path, but also other events that will happen during the movement, like attacks of opportunity of falls.

5.5 Performance

With the help of **BenchmarkDotNet**, we can start various simulations of games to measure the performance of the search graph algorithm. Since the number of edges grows with the size of the creature, we'll setup a map with four types of creatures, mixed between enemies and allies, and calculate for each one the movement graph.
The test will simply be:

```
Setup() {
    Map = InitMap()
}
BenchmarkMedium() {
    Map.FinPathMediumCreature()
}
BenchmarkLarge() {
    Map.FinPathLargeCreature().Search()
}
BenchmarkHuge() {
    Map.FinPathHugeCreature().Search()
}
BenchmarkHuge() {
    Map.FinPathGargantuanCreature().Search()
}
```

BenchmarkDotNet will take care of calling these methods multiple times, excluding too high or too low values, to give us the average time of each method.
And here are the result of the first implementation:

Method	Mean	Error	StdDev
FindPathMediumCreature	980.3 us	3.48 us	3.09 us
FindPathLargeCreature	3,916.7 us	18.31 us	17.12 us
FindPathHugeCreature	5,044.4 us	28.47 us	23.78 us
FindPathGargantuanCreature	31,411.2 us	176.03 us	164.66 us

As we can see, calculating the path of a medium creature takes around 3ms, while a gargantuan creature can take up to 30ms on average, which means 3 frames in a 60fps game.
This may seem low, but there are few things to consider:

- This is just the **first part** of the AI implementation, and much more calculations will be needed, especially if the creature has a lot of possible actions or aoe spells
- This benchmark has been executed on a **high-end CPU desktop computer**, and the result will be a lot higher on less powerful devices such as low-end smartphones, for example
- Since this is just the first draft of the algorithm, written without optimization in mind, there should be a lot of room to drop down these numbers

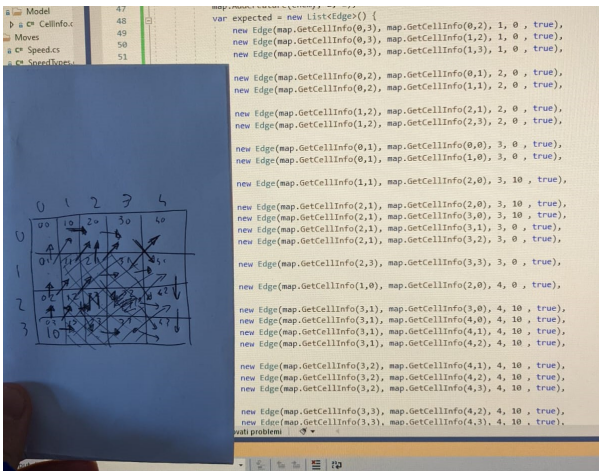
But before proceeding to the optimization, there is something crucial about both optimization and stability of the game: Unit Tests.

5.6 Unit Tests

Until now we have map creation and creatures handling with the map, meaning not also movement, but cells occupied and threatened areas too.
Unit tests are the answer of two questions:

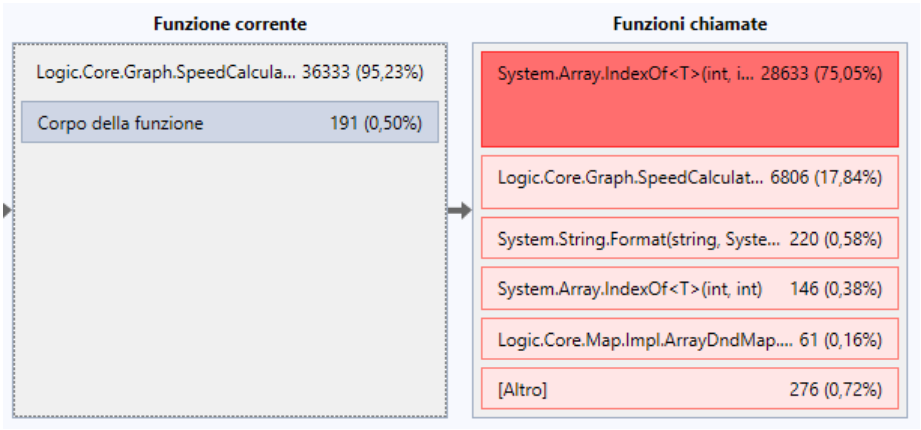
- How can I be sure that everything works as intended?
- How can I be sure that everything *still* works as intended after a refactor due to optimization?

We saw that the edge calculation depends on a lot of factors: creature size, terrain type, terrain height, if the destination cell is occupied by an ally or an enemy, the size of that enemy, if that movement is a fall or has me exiting a threatened zone, and so on..
Testing all this cases manually is simply not feasible, and even less feasible is testing everything after a huge optimization refactor, that usually implies big changes to the data structures.
To write this tests, we create an ad-hoc map and check that all methods returns what we expect. Writing them is not trivial, but in this way we are 100% sure (if we managed to test all possible cases) that if the tests pass, the game works as intended.
For example, to validate the search algorithm, we can setup a map where a cell can be reached in two ways: one requires less movement but gives an attack of opportunity to an enemy, and the other requires more movement but gives no attacks to the enemy. We then call the search algorithm, and since the edge search is in a predictable order, we can check if the list of the edges matches what we expect:



5.7 Optimization

To find **bottlenecks** in the code, since we are developing a .NET Standard library, we can easily use the Visual Studio Profiler.
To have significants results we'll call the same methods as the benchmark, but this time we'll take care of executing them lot of times, to enhance the slower parts over the rest.
This is the result of calling the search on a Gargantuan creature two thousand times (which should take around 1 minute):



From the result, we can surprisingly see that 75% of the running time has been spent to calculate the starting and ending cells of the creature, and not the calculation of the actual edges. Also, the search algorithm excluding the edges calculation occupies only 5% of the running time.
This is the culprit line:

```
if(myCells.Contains(newTo))
{
    continue;
}
```

Since in this part we need only to check coordinates, we can change it to:

```
if(myCells.Any(my => my.X == newTo.X && my.Y == newTo.Y))
{
    continue;
}
```

And this is the new benchmark after the change:

Method	Mean	Error	StdDev
FindPathMediumCreature	1.201 ms	0.0084 ms	0.0078 ms
FindPathLargeCreature	1.895 ms	0.0092 ms	0.0086 ms
FindPathHugeCreature	3.439 ms	0.0344 ms	0.0305 ms
FindPathGargantuanCreature	8.557 ms	0.0331 ms	0.0258 ms

Changing just a single line made the algorithm 3 times faster for bigger creatures, and unit tests guarantee us that the game logic hasn't been touched for the worst!
Running the profiler again will give us a new bottleneck, and we can continue this procedure until there's nothing left to optimize or we are happy with the result.

Some tricks used to speed up the algorithm:

- Using **Hashset** or **Dictionary** instead of List where the order doesn't matter

- Using **SortedList** with a custom **IComparer** instead of using a normal List and order it after all the item are present
- Using a custom **IComparer** for searching in the lists when we know that we can only look at a subset of fields
- **Manually calculating** the list of the "next" cells in case of big creatures, instead of creating the whole square and removing the starting cells
- Cutting the loops with **early continues, breaks and returns** instead of waiting for normal execution
- Calculating the maximum value in an array when inserting a new item instead of using **.Max()**
- For the map dictionary, instead of using String.format("0,1", x, y) for the key, we'll use **(x <<6) + y**, which guarantees a unique key for a map up to 64x64 cells. This speed up key creation, insertion and also item retrieval (the key are computed with .GetHashCode(), that just returns the value in case of an int).

And this is the final result:

Method	Mean	Error	StdDev
FindPathMediumCreature	817.8 us	1.25 us	1.17 us
FindPathLargeCreature	332.6 us	1.11 us	1.04 us
FindPathHugeCreature	400.2 us	2.08 us	1.85 us
FindPathGargantuanCreature	299.8 us	0.91 us	0.81 us

The search is now taking between 0.25 to 0.8 milliseconds to complete! It may be counter-intuitive that the bigger creature take less time, but this is a simulation in a game-like map and not in a plain desert, so bigger creatures have way less possibility of maneuver that smaller ones.

5.8 Search job

Jobs in unity can only contain **unmanaged types** (no strings, class instances, delegates...) so there are a couple of solution to avoid this limitation:

- The job could invoke the search and then convert the data in a way that can be read from the unity main thread
- The job could invoke the search and the core lib will cache the data that could be retrieved later

The first solution seems more structured, but involves a lot of logic on Unity-side. The second solution may not be clean if not implemented well, but it's in the direction of modern design patterns like **Flux/Redux**, and while a proper *redux store* won't be implemented in the project, it still is a solid solution when handled correctly.
In the game UI, when selecting to move, the greyed out cells are **unreachable**, and the possible paths are marked with **colored circles**. In the following example, the creature can choose to move along the green path, or move along the yellow path and take a bit of fall damage to use less movement:

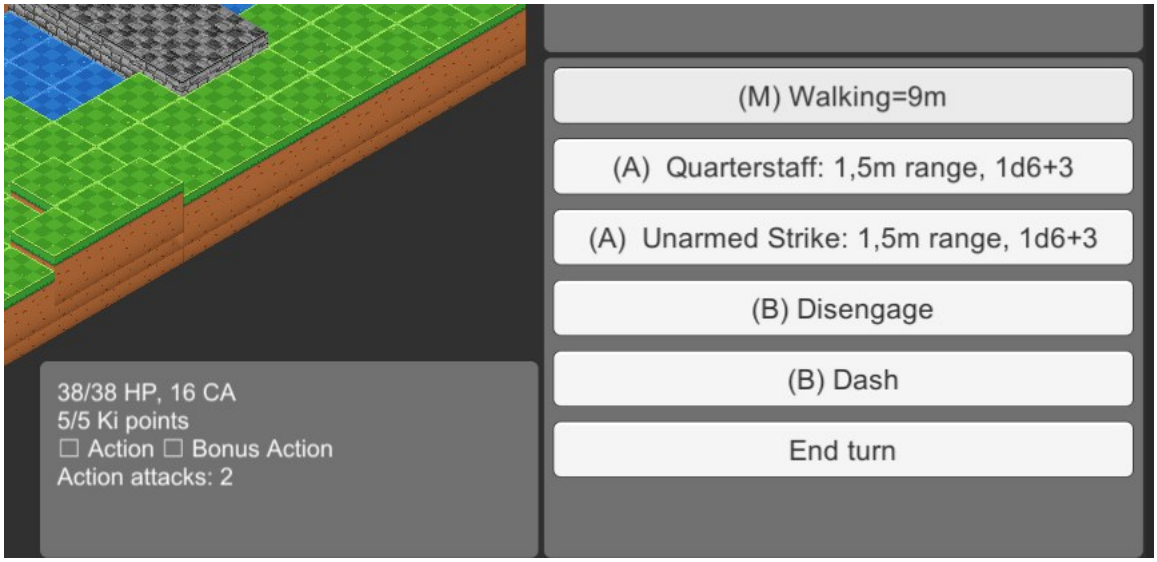
5.9 Smoothing movement

Since D&D is a game **based on cell movement** and the interaction with the cells is fundamental to calculate attacks of opportunity, the movement is not smoothed and the creatures moves from the center of a cell to the center of the very next cell.

6 Making it playable

6.1 Actions

The movement action is just one of the many things that a character can do. Each creature has a standard set of properties that describes the basic actions, in common for everyone, like having Movement, an Action, a Bonus Action and a Reaction each turn.
If the creature has additional actions or abilities, those are described by interfaces that are joined together to create the list of the creature skills, for example a spellcaster will implement the *ISpell-Caster* interface that declares how many spell slots and spells the creature has at their disposal.
The method **GetAvailableActions** of **DndBattle** delegates the creation of the available actions to a list of **IActionsBuilder**. Each of them checks conditions and collaboratively create the list of the action that a given creature can execute in that situation (position, temporary effects, already used actions...).
This method will be used from the Unity3d code to show the list in the UI, but also from the AI algorithm to evaluate the available actions.
In the game UI, each action is labeled as (M), (A) or (B), meaning that it will consume movement, actions or bonus actions. There's also a bottom pop-up that highlights information about the creature currently in turn.



7 Goals Oriented Actions Planning

At this version of the game, it can be played in a hot-seat way, where all the creatures are controllable by the users. As the escape or give-up of a team is not contemplated, a team wins when all the other creatures are dead.
The main general **goal** of each creature would then be to reduce the enemies HP, but there are other factors to be taken into consideration when deciding which action to take:

- Will I or an ally of mine also lose HP during the action?
- Will my action put a negative condition on the enemy or a positive condition or an ally?
- If I have healing, should I instead heal a low-hp ally?

To evaluate which actions to take, we will search all the possible courses of actions and choose the one that fulfils better our list of goals.

7.1 Actions sequences

As said in the previous paragraphs, each creature has an **Action** and a **Bonus Action** at their disposal. The attack action could even consist in more than one attack, and a creature can have additional abilities to replenish a used Action or Bonus Action.
Also, the Movement Action can be split as a creature pleases, so for example the monk can executes 2 attacks with the main action, then use the "Flurry of blows" ability and executes another 2 attacks with their bonus action, splitting movement between all those so that the actions sequences is:
Move - Attack 1 - Move - Attack 2 - Move - Flurry of blows - Move - Bonus attack 1 - Move - Bonus attack 2 - Move - End turn
Anyway this is very unlikely, because moving after an attack means to offer an attack of opportunity to the enemy, that will reduce the utility of the "Don't lose hp" goal. A more likely sequence of action would be:
Move - Attack 1 - Attack 2 - Bonus Attack 1 - Flurry of blows - Bonus Attack 2

7.2 Building the action sequences

7.2.1 Copying the game state

The first step of the implementation of the AI algorithm is to build the **actions sequence** that an agent can perform during their turn.

For an algorithm like that to work, we need to make a deep copy of the whole game state (map, creatures, battle managers etc) and execute the action in the new copy.
Unfortunately, c# doesn't have a built-in method to copy complex object like Kotlin or Javascript, so we'll apply a little hack to make the copy to be sure there will be no bugs or inconsistencies. Being DndBattle the entry point of all the game logic, we'll have:

```
public IDndBattle Copy()
{
    var serialized = JsonConvert.SerializeObject(this);
    return JsonConvert.DeserializeObject<DndBattle>(serialized);
}
```

By serializing to json and deserializing we are 100% sure that the new object contains **only** copies of the original data, with no shared reference whatsoever. Unfortunately this approach is not feasible in "production" because it's **extremely slower** than a property-by-property manual copy, but we'll use it for the sake of (developing) speed and potential bugs in the copy.
Starting the development with this approach lets us develop knowing that there are no bugs in the object copy, and when the algorithm is completed and working, we'll swap it with a custom manual deep copy.

7.2.2 Breadth-first search

As per the movement problem, we can image the sequences of actions as a **graph** and we can calculate the actions tree by using a similar algorithm, with some tweaks to speed it up, like not permitting step-by-step movement if it's not interrupted by other actions:

```
List<ActionList> GetActionLists(Battle battle) {
    Result = ListOf()
    Queue = ListOf(ActionList.Empty)
    while(Queue.NotEmpty() {
        Current = Queue.First()
        Foreach(Action in NextActions) {
            if(Action is Movement && PreviousAction is Movement) {
                continue;
            }
            if(Action is EndTurn) {
                Result.Add(Current)
                break;
            }
            Foreach(Target in Action.AvailableTargets) {
                GameCopy = Current.Game.Copy()
                GameCopy.Execute(Creature, Action, Target)
                Queue.Add(GameCopy, Action)
            }
        }
    }
    return Result
}
```

After more than an **hour** of execution, the algorithm (still going) passed thought more than 300k edges between action combinations, with the longest one being composed by 6 actions.
This algorithm is clearly too slow to be used as-is, but we can now try a custom clone method and re-run it to check the difference.
After updating the clone logic, the algorithm passed thought 2 million edges, and then passed out because it filled 32 GB of ram.

7.2.3 Branch pruning

To increase the speed of the algorithm, we can **observe the actions that have been produced**, keeping in mind that the best course of actions would be "Move to enemy, Attack 1, Attack 2, Bonus attack 1, Increase bonus attacks, Bonus attacks 2", for example this is one of the generated actions:

- Move to enemy, Attack the enemy, Move away, Use the dash action, Go back to the enemy, Attack it again, End turn

This sequence is wrong and illogical on so many levels, and while it would be certainly not chosen by the GOAP we can **prune** this decision beforehand, saving a lot of branch search.
First of all, since moving away from an enemy causes an attack of opportunity and a potential loss of health points, we don't want to move if it would cause an attack of opportunity.
This leads to a very static combat, but this is also the case in the true Dungeons & Dragons game, so we can exploit this flaw in the game design to speed up the branches without the player being surprised by the behaviour.
In general, even if we can split the movement between attacks it doesn't make sense to move in two different instances in the same turn. So we can reduce the load number by a lot if we ignore any movement action when I've already moved.
The only time where a creature would move after attacking is when the attacked creature dies mid-turn. In that case the course of action is invalidated because the game state changed too much. In this case we can simply run the algorithm again to find a new list of valid actions.
Another instance of wrong course of action would be:

- Move to enemy, Use the Dash bonus action, Attack 1, Attack 2, End turn

In this case, using the dash at the beginning of the sequence is not only useless because I don't need the extra movement to go anywhere, but also counterproductive because then I don't have a bonus action to make my attack.
This case can be solved by adding a priority to the actions and only pursue the higher priority actions:

```
NextActions.RemoveIf(Action.Priority < NextActions.MaxPriority())
```

We define the following priorities:

- Cast a spell: priority 5
- Attack an enemy: priority 4
- Use a special ability: Priority 3
- Move: Priority 2
- Fallback actions: Priority 1

To apply this check we need to swap from a BFS to a DFS, so we'll have the result of the high-priority actions sooner. In C# this is very easy, it only means we replace the Queue with a Stack.

7.3 Evaluating Goals

The previous algorithm can still be optimized, but since we are creating a list of possible course of actions, we'll implement the **GOAP** first so we can only retain in memory the best course of action instead of the whole list.
For each course of actions found, we need to **evaluate the fulfillment** of our goals.
Since in this game we have actions outcome decided by dice, the goal fulfillment will be calculated using an average projection of the game in the next turn.

7.3.1 Agent knowledge

For the sake of this sample project, the creatures will **metagame**, which in the DnD slang means that they access information that normally they don't have at their disposal.
For example, when evaluating an attack action the goal will calculate the average damage based on the probability of the creature to attack the other one. Doing this in a game legit way is not trivial; for example to evaluate a physical attack action we would need to:

- Evaluate if the targeted creature has some characteristic that reveals their Armor Class, for example if it wears a full plate armor or a robe
- Evaluate the intelligence of the attacking creature. Based on the INT ability score we can try to guess the correct AC of the target and use that value for the projection
- When dice are rolled and the attack either misses or hits, we can update the guessed AC based on the result

This is only an example of how agent knowledge can be used to **enhance and customize the behaviour** of each creature: a clever enemy will ignore the full-plate tank paladin and try to take down the dangerous wizard first and that *seems* intelligent, even if the real reason is only because the probability to hit the wizard is higher.

7.3.2 Evaluation integration

The goals are represented by a very simple interface:

```
interface IGoal
{
    float EvaluateGoal(
        ICreature creature,
        IDndBattle oldState,
        IDndBattle newState
        );
}
```

This is all we need to add the **actions evaluation** in the previous algorithm. After changing the List of results into a single result, we'll set it if the course of actions in that cycle has a higher evaluation:

```
if(Action is EndTurn) {
    Evaluation = 0
    foreach(Goal in Goals) {
        Evaluation += Goal.evaluate(old battle, new battle)
    }
    if(Evaluation > Result.Evaluation) {
        Result = Current
    }
    break;
}
```

7.4 Subsequent optimizations

Now that we have joined the search algorithm with the GOAP, we can see that there are some flaws in it:

- The path search algorithm takes a **huge part** of the CPU time

- Since the GetAvailableAction() method is also used for player characters, there are resulting courses of actions that **don't make sense**

7.4.1 CPU time of the path search

Since per-se the path search is not so slow, the GetAvailableAction() calculates the available paths every time because it's also used for the player characters. Of course, this is useless for our algorithm, because the same creature can reach the same cells in all the turn, except if it uses the dash action. As we can see from benchmarking the process, in one minute of play 84% of the CPU time is spent in calculating the reachable cells.
After adding a **simple cache** that it's invalidated if the character uses the dash action, the CPU time spent on that method drops to 66%
, which is still a lot, but we already knew that in a GOAP with pathfinding, that's the major entity in the CPU time. We could continue with the optimization since there are slow low level list-related methods, but we will wait after we have solved the other issues with the main algorithm and we can avoid discussing them in detail

7.4.2 Courses of actions that don't make sense

A human player can easily **think in advance**: if I have an ability that grants me an additional attack, I will only use it if I want to attack afterwards, or I will not use it at all if I can see that there are no targets available for the attack.
Unfortunately, our search algorithm can't think like this, so we currently end up with actions that waste resource without any gain.
We could avoid this problem with 3 solutions:

- Since we are using GOAP, we could add a special **"don't waste resources goal"** that returns an infinite negative result, to discard the action sequence. While this works, we are just discarding the leaves of the search
- To prune more branches, we can modify the "GetAvailableAction()" method to only return the actions that are valuable now, and not after others. For example, if there are available attacks to do, we won't tell the AI that it can increase the number of available attacks until it uses all of them. While we are removing **"ownership"** of the validation from the GOAP, there's a lot of CPU time saved this way
- Another approach could be to **batch some actions together**. For example, we won't return to the AI the action "get another attack", but instead we'll return "get another attack + attack", forcing that branch search. While this solution sounds cleaner, it doesn't take into account all possible edge cases and requires more effort in the implementation

In the projects I've implemented the second solution, and the number of branches visited dropped by a lot. Taking the monk as an example (which is the creature with the most combinations and abilities) the algorithm passed from evaluating 61226 possible course of actions to only 6270, with the same output.

7.5 Defined goals

Our goals stays pretty much the same for the duration of the game, and there's no need to move insistence around during it, so insistence is a **fixed number**, then multiplied by the fullfillment of the goal to reduce its incidence on the final value.
In an evolution of the algorithm, different creature should have different insistences for their goals.

7.5.1 Reduce enemy health points

This is the most important and priority goal: all other actions doesn't matter if the enemy is dead. Because of this, this goal has always the same insistence of 1.

The algorithm sums the hp of the enemies before and after the action projection, and the difference is the fulfillment of the goal. This doesn't ensure that the best target is attacked, but it ensures that the maximum damage is.
A smarter evaluation would be to attack the reachable enemy with the lowest hp, but we'd also need to take into account the AC probability.

7.5.2 Increase ally health points

At the opposite side of the previous one, this works as a dual goal: on one side we grant points to heals, on the other side we ensure that our health isn't reduced by other attacks on our turn, meaning attacks of opportunity.
The discontentment generated by losing hp is implemented by returning a negative fullfillment. Since losing hp is a big deal and we won't to avoid attack of opportunity in pretty much all cases, the insistence of this goal is 10x in case of negative result.

7.5.3 Buff ally

This goal counts the buffs of the ally team and it works both as a deciding factor in case I have multiple abilities with the same outcome but one applies also a buff, and as a fallback course of actions, because a creature can always take the "Dodge" action which counts as a buff.
So for example at the start of the game when no enemy is reachable, a creature will choose the dodge action because it's a simple +1 buff to the team and no other action is available.

7.5.4 Melee/Ranged position

With only the previous goals, no action would be taken by the melee creatures at the beginning of the game because they would just take the dodge action and maybe move randomly since the resulting fullfillment would be the same.
With the Position goal we have two different goals: one for the creatures that prefer to attack melee and one for the creatures that prefer to attack at range.
This goal check which is the nearest enemy and returns a positive or negative fullfillment based on the preferred creature's position.
Since combat in D&D is very static, we want this goal to only urge the creatures to engage at the beginning of the game, so this goal has a fixed incidence of 0.1

8 Conclusion

8.1 Performance

Performance is a huge con of this approach: even with our in-depth analysis using the Visual Studio Profiler and all the improvements, branch pruning, cache and little hacks, the GOAP algorithm struggles a lot in a game with so many possible actions.
While waiting a couple of seconds is no big deal in a game like this one, we tried it on a high-end computer and this few seconds will sure be a lot more if we were to run the game on a low-end mobile device.
Also, to increase CPU performance we heavily applied ram caches, which again are not a problem on a PC but are a problem on low end devices.

8.2 Behaviour

As a veteran D&D player, the resulting behaviour feels good, aside some improvements that could be applied to the attack target selection and a bit of positioning.
The creatures aren't taking non-sense actions and spreading attacks or not positioning at their best

could even add a sense to not be playing with an optimized AI, since they are common mistakes usually done by real players.

8.3 Technology

The Unity Job system has pros and cons: the main pro is that we could take the logic library and move it to a full-authoritative server not based on unity to implement online multiplayer. The big con that I've discovered halfway is that the Job system doesn't work on the webgl platform.
Also, the impossibility of passing managed classes obliges the developer to write bad code around it.
In hindsight, it may be better to integrate the code into the unity project and use the coroutine system to manage the "background" execution (but the algorithm may take longer in that way)

8.4 Next steps

We could add a lot of things to improve the AI:

- Each creature could have different insistences for goals: a feral creature would care only to target the nearest enemy, while a clever creature should have a higher insistence on attacking the most dangerous enemy
- A creature could have different insistence based on its state: a FSM could change the insistences of the goals after some event occurs. For example a pirate could switch from being aggressive to being defensive, reducing the "deal damage" insistence and increasing the "buff" and "positioning" ones.

8.5 Code references

In the project DndCore, the following folders and classes contains the highlight of the algorithm:

- Core/Battle/ActionBuilders contains the classes that return the valid action for a given creature
- Core/Creatures contains the creatures and their abilities
- Core/GOAP contains the course of actions builder and the goals
- Core/Graph contains the Uniform Cost Search related classes
- Core/Map contains the map classes