Using GOAP to aid level creation

Danilo Aimini, Moodie Ghaddar

1 Introduction

In the past decade and a half, the advancement in game AI architectures has not shown a lot of progress. The most widely-used architecture today, is still the behavior trees that was introduced in 2004. There have been attempts at creating other architectures such as Goal Oriented Action Planning(GOAP), but it failed to take the world by storm. It seems like the next step in game AI is not necessarily in creating new architectures but perhaps implementing expert heuristics in development tools to aid developers to create content more easily. Heuristics such as hints, classifications, visualizations, interactivity and even the ability to experiment with design creations via tools are all methods that can enhance the development process.

In this paper we will talk about how one can use GOAP to collect the data required to achieve this. GOAP is a convenient architecture which can plan a sequence of actions for an AI to perform dynamically in real time. Another way to utilize such an architecture, would be to perform planning from the players point of view. Usually in games players have a set of actions that they can perform. If we allow GOAP to perform those actions, we will have access to all possible world states that can be reached from a specific starting point. This is especially interesting for game genres that have actions with specific requirements and outcomes. Using a GOAP based algorithm we can aid the creation of levels for designers by offering an insight about the space of all possibilities the player can reach from their design.

2 Research

Our research progress followed several steps. First, we settled on a game idea that allowed us to clearly visualize all elements defining the problem and its solutions. Once this setup was made, we proceeded with writing the GOAP algorithm and adapt it to the specifics of our design. Lastly, we analyzed the results of our research to determine the interesting implications of the data we collected.

2.1 The game idea

To showcase our ideas for GOAP-based tools, we settled on a simple game design that could help in conveying our purpose. The structure resembles old school *The Legend of Zelda* games: players need to traverse a maze-like dungeon full of enemies and puzzles. Some of the connections between rooms, though, are blocked by obstacles: players will need to retrieve objects hidden in the rooms and use them to overcome the obstacles and make their way to the end of the dungeon.

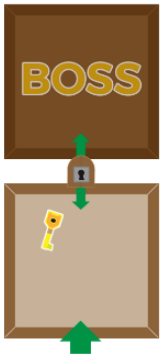

Figure 1 A very simple dungeon map.

Ideally, an interesting game would have different rooms with combinations of enemies and puzzles that reward players with objects once completed. For the purpose of this paper though, we will be focusing on the underlying structure of dungeons disregarding these game design details.

2.2 Items

Players will need items to overcome specific obstacles. In our model, all items are consumable: the player can own any amount of any item, and they are decremented upon usage. Also, objects can only be used to overcome obstacles, and they cannot be wasted or used in any other way. These rules are quite strict and do not pair well with the items we picked in this sample but are needed to keep the model consistent.

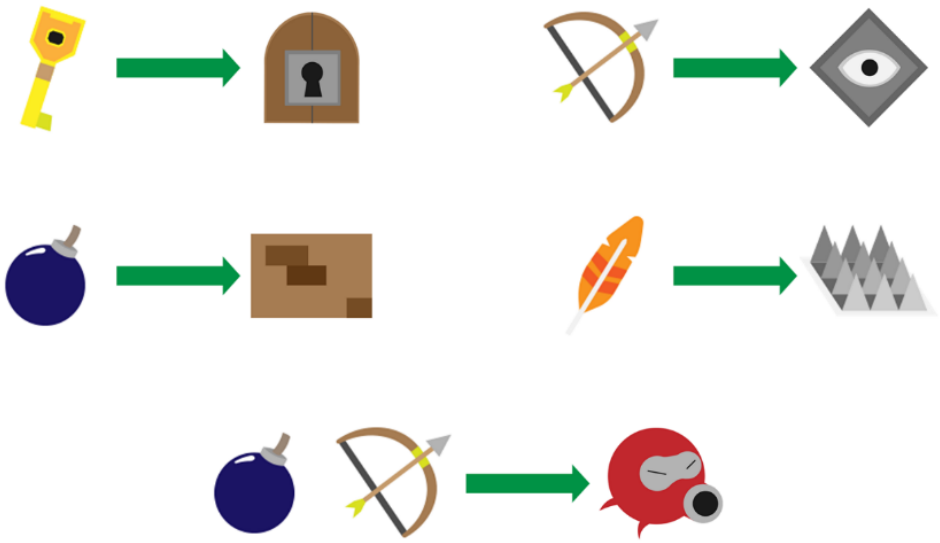

Figure 2 Item – Obstacle correspondence.

Figure 2 shows the correspondence between items and the obstacles that the player can overcome by using those items. As shown, keys can open doors; arrows can open eye-shaped switches; bombs can blow up walls; feathers allow players to jump over spikes; bombs or arrows can be used to defeat enemies. The same objects can be used to overcome different obstacles, and obstacles can be tied to different objects.

In our demo, the data structures determining the number of items and obstacles, their names and the correspondences between them are all exposed to Unity and can be edited with an easy to use user interface. Of course, altering these values will change the results of some of the operations in the random dungeon generation, meaning that the same seed will result in radically different map layout when these settings are changed.

2.3 Random Generation

Although our project does not necessarily require randomly generated maps, we decided to implement it for two reasons:

1. To speed up development time, as the alternative would have creating a fully-fledged level editor;
2. As it highlights some of the possible developments (see later chapters).

The random level generator requires three unsigned integers as inputs: the number of rows and columns the room grid for the dungeon will use and a seed that will be used to initialize the rand function. While number of rows and columns are required, and default to 3 and 3, the value for the seed can be omitted – in that case, a random seed will be selected.

The procedure for generating the maps starts by setting up a basic dungeon with size rows*cols, with all connections enabled with no obstacle set. The room in the lowest row, center column is set as the entrance, while the Boss room is randomly placed in a room in the upper half of the dungeon. With this setup, we then proceed in removing a random number of connections between (rows + cols) and (rows * cols). Of course, this could result in the Boss room being impossible to reach from the start room, so we run a simple Dijkstra search: if the boss room is found, then we can proceed to the next step; else, we will need to add some connections back and try again until the Boss room is reachable from the start room again.

Whenever we are sure that we have a path between start and Boss rooms, it's time to add some obstacles to the mix! A random number is generated between count/2 and count, where count is the number of rooms that can be reached from the start room. That number represents the number of obstacles we are going to add to our dungeon: each of them will be placed on a random connection between two rooms that are reachable. Moreover, an item that can be used to overcome that obstacle will be placed in a random reachable room in the dungeon; if the obstacle can be overcome with more than one item, a random one will be selected among that list.

At the end of this process we will have a complete dungeon map. Of course, since the obstacles and items were placed randomly, there is no guarantee that the Boss room can be reached from the entrance of the dungeon, even if we are sure they are connected. We will need to use GOAP to verify whether the dungeon can be solved by the player or not.

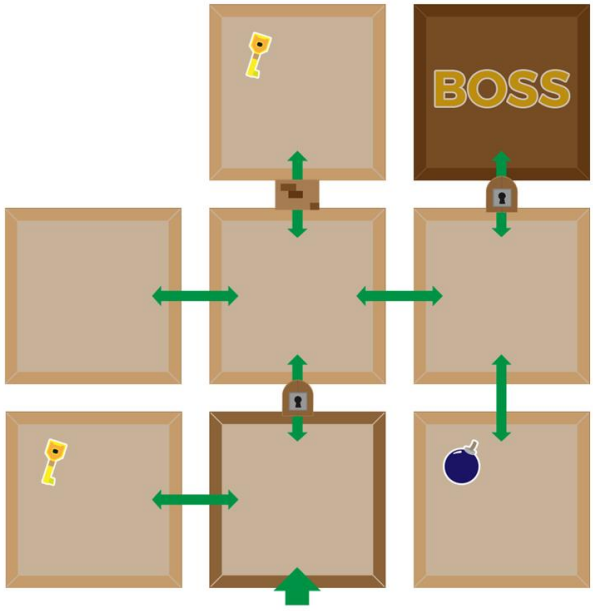

Figure 3 The result of a random dungeon generation. Note that in our demo we do not show rooms that cannot be reached from the entrance, or that can only be reached by traversing the boss room.

3 GOAP

The idea behind GOAP AI architecture is that instead of specifying which AI actions transitions to other actions or which actions are associated with which goals, the connection is made dynamically in real time. This is achieved by making every action in a game to have preconditions and effects. To perform an action, the agent must first meet all the preconditions necessary. If the preconditions are not met, then the algorithm will look for an action with an effect that meets the preconditions required.

Once all preconditions are met to achieve a certain goal, then the actions are chained together for the agent to perform. Effects and preconditions are stored as a world state. A world state represents the state an agent is in at any point of time. Table 1 presents a simple example of what a world state and goal state could look like.

Table 1 World state example

State	Kill Enemy	Gun Loaded	Has Gun
World State	False	False	True
Goal State	True	NA	NA

Using the above states as an example, the goal of the agent is to kill the enemy. GOAP will look for an action that has the effect of changing the state “Kill enemy” from false to true.

Let us say that action is called “Shoot Enemy”. To perform that action two preconditions are required. The agent must have a gun and the gun must be loaded. In this case the “Gun Loaded” state fails. Hence the algorithm will look for an action that changes

the state “Gun Loaded” from false to true. That action could be “Reload Gun” which has no preconditions. Since all preconditions are now met, the plan the agent will receive is to “Reload Gun” first and then “Shoot Enemy”.

3.1 World State

The world state differs vastly depending on the type of game you are creating. In our project, the state of the player is determined by how many items in the inventory, rooms visited and if the boss room is reached or not. We used an unsigned integer array to represent the inventory items, stored a list of ids of rooms visited and a boolean for the boss room.

In addition to those variables we also decided to store a list of all potential actions in a world state. In the traditional GOAP however, actions are assigned to agents beforehand and are independent of the world state. For optimization purposes and ease of use in our case, actions are removed and added as the player explores the dungeon.

3.2 Algorithm description

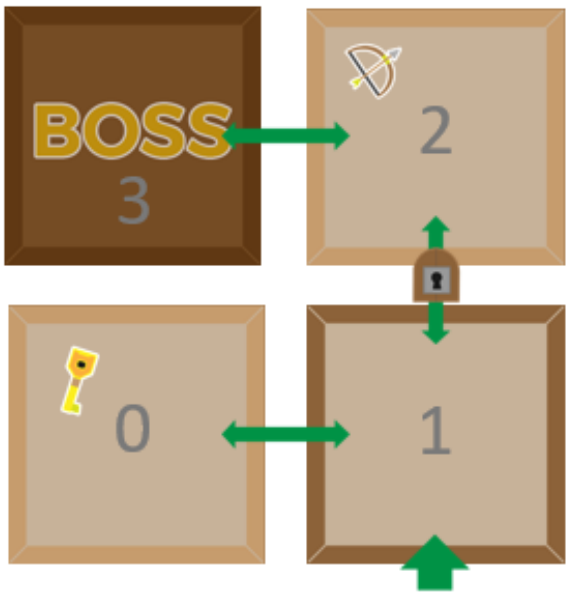

Figure 4 Example dungeon layout.

As shown in figure 4, the starting room in the layout is labeled as 1. Since there are no items in the room the player starts with no items in their inventory. The starting room provides the player with two actions as shown in the table below:

Table 2 Initial World State Actions.

Action	Requirement
Go from room 1 to room 2	Key
Go from room 1 to room 0	None

The algorithm goes through every action and, if it meets the requirement, it performs it and creates a new world state. In this case, the first action is not performed since the player does not have a key yet. The second action however has no requirement, so it performs it and removes it from the action list of the new world state. Since room 0 contains a key, a key is added to the player inventory. The algorithm runs again on the new world state which only has the first action of table 2 left to perform. This time the requirement is met, so the door obstacle is removed by consuming 1 key. The arrow item is added to the player inventory along with a new action which is to go from room 2 to room 3. Since the action has no requirement, it will be performed on the next iteration which results in reaching the boss room.

We run the algorithm until all world states have been exhausted. World states that end up reaching the goal are added as solutions while the others are discarded. Once all solutions have been found they are sorted from best to worst depending on how many actions it took to reach the goal. Before prompting all the solutions to the user, we filter out all redundant solutions that are similar to each other. This is done by taking two solutions with the same cost and removing all actions that have no requirements from them. Then if the actions remaining are the same we discard one of the solutions. Filtering can be enhanced by allowing the user to enter constraints on costs and allowed actions.

4 Project Results

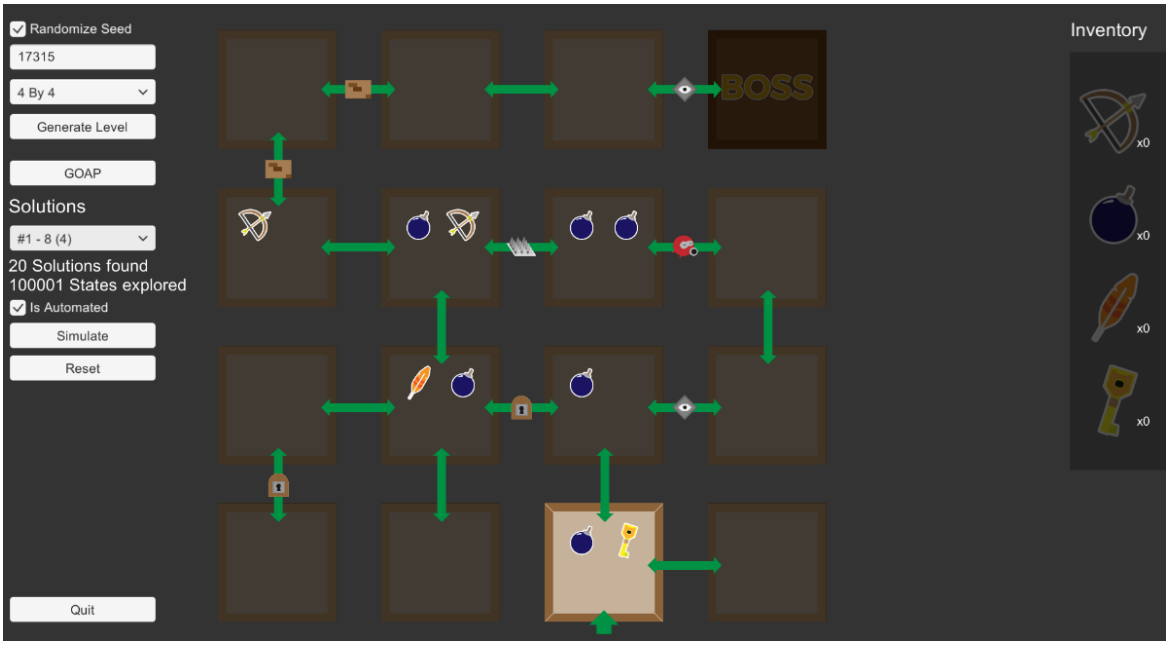

Figure 5 Demo Screenshot

Figure 5 shows a screenshot of our demo application developed in Unity. Thanks to the UI, the user is able to randomly generate dungeons of different sizes by either inserting a seed, which will always return the same map layout as a result, or by letting the system automatically provide a new one for them at every click of the *Generate Level* button.

At a glance, it is difficult to know if the randomly generated dungeon is solvable or not. One would have to either play through the entire level, but a faster solution is to use some sort of search technique, like GOAP in our case. With a click of a button, all possible solutions are listed in a dropdown menu saving precious time for the designer. In the above figure, 20 different solutions were found after exploring 100,000 world states. The disadvantage however is the huge number of world states that need to be searched. The time complexity is exponential as more items and actions are introduced.

Once a solution is selected the user can simulate it and watch an AI traverse the dungeon. Unexplored rooms are lit up when the AI visits them. All items inside are then collected one at a time updating the inventory on the right. Obstacles are also removed after consuming the item required.

Since we gathered such huge amount of data though, it would be a waste to just use it to figure out if a dungeon is solvable or not. Fortunately, if we examine the data closely, we can get much more out of this.

5 Further developments

On top of finding solutions to the problem at hand, this algorithm allows us to gather a lot of interesting data. After testing our project for a while we started to notice interesting patterns and behaviors that can be used to enhance level design further. The following are some examples.

5.1 Loop avoidance

One of the interesting cases we noticed was that even if a dungeon is solvable, if the player uses items in the incorrect order they may not be able to solve the dungeon anymore.

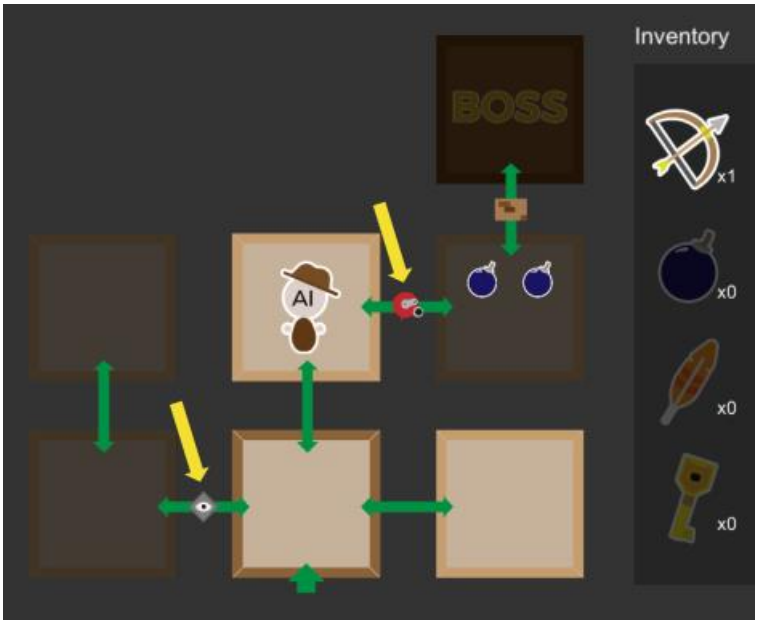

Figure 6 Potential of being locked in a dungeon

In figure 6, the player has only one arrow in his inventory. They can either use it to defeat the enemy or to overcome the eye obstacle on the lower left of the dungeon. If the arrow is used on the eye obstacle, the player does not have any more items that allow them to defeat the enemy, locking himself up in the dungeon.

This scenario is not a rare case in video games and is usually detected through in-depth testing of the level. Some games may even ship without noticing that a sequence of actions could lock the player in a level.

To detect such a case, we can create two sets A and B. Set A will contain all the rooms explored in sequences that lead to the boss room, while set B will contain rooms explored in sequences that do not lead to the boss room. If we subtract the intersection of those two sets from B, we would get the set of all the rooms that render the dungeon unsolvable if the player visits them.

$$B - (A \cap B) \tag{1}$$

After examining all world states, we can apply the above formula and then we can hint to the designer which rooms may potentially cause issues. The designer can then either manually tweak the level or even have the tool suggest what can be modified to solve the issues.

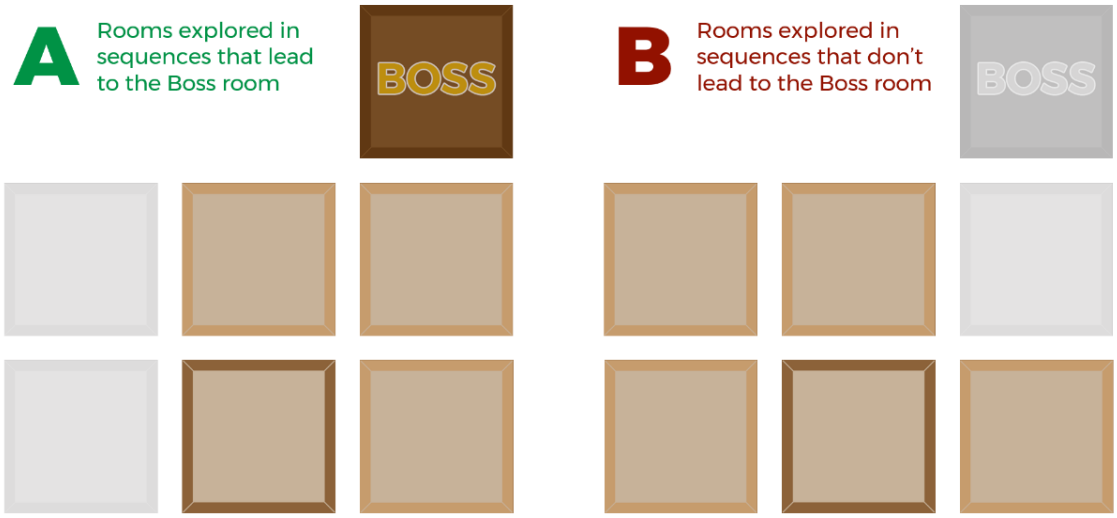

Figure 7 Map subsets for the example dungeon seen in figure 6.

5.2 Guided level creation

As programmers have come to understand in the last few years, auto completion and hints are very helpful when writing code. It's a huge time saving tool, because it prevents rewriting long words over and over again, looking up function arguments for seldom used libraries and provides templates that allow us to write better code. This same logic could be used in level creation: the data gathered from running automated tests can be used to empower the designer in several ways.

A simple tool integration could generate random solvable levels given a series of constraints; this way, a designer could quickly have a working base for a level to start iterating on. Of course, this would be highly dependent on the game design and require some understanding of which constraints lead to interesting gameplay. In a game like the Legend of Zelda, a highly appreciated feature for a dungeon is the need for backtracking; that is, when the sequence of events required to solve the dungeon, forces the player to visit rooms they have been to again in order to open new paths. This is the type of structural feature that could easily be detected by analyzing the sequence of actions that leads to solving a dungeon generated with GOAP.

Another interesting feature in our specific demo game would be the ability to localize "hidden" rooms to hide special rewards into. This is also easily done: by analyzing all possible solution, we could give each room a score based on how many solutions traverse that room. The lower the score, the lower the chances a player will get to that room making it a perfect hiding place for a secret bonus item.

5.2 Bug and exploit detection

Another application could be related to games with user generated content created through a level editor. Games that put a focus on levels created by their players usually do not need an automated system to verify whether a level can be completed or not, as they can simply require players to complete their own levels before submission.

This does not mean that a GOAP-based level solver should be considered useless though! An algorithm like the one we described could actually be used along the usual "self-completion" method to highlight issues with the game: if a user is able to complete and submit a level, but the algorithm actually returns that no solutions can be found for that map layout, then there has to be an issue in any of the parts involved. Either the algorithm is not taking some specific solution into account, or the player actually used a hack or bug to solve a level that should technically be unsolvable. In either case, a level presenting this type of mismatch should be marked and presented to the development team so that the issue can be solved.

6 Conclusion

Albeit with a small scope determined by the specific rules of our example design, we feel like our research gave interesting results on many separate levels.

First, it allowed us to see how many of the techniques and algorithms that we learn in "classic" artificial intelligence contexts can be altered and bent to different purposes depending on our target result. As GOAP uses A* to find sequences of actions instead of actual movement paths as intended by design, in the same way we ended up using GOAP to predict possible user behaviors instead of planning AI actions. This is especially interesting as it highlights how AI development is not just about using the same toolset over and over again, but about picking the right set of tools for the problem at hand and using them in a way that makes sense.

On top of this, we feel like working on this project gave us a better insight on using AI to empower the developers, creating useful tools that do not necessarily end up in the game itself but can make a difference in its production workflow. Getting this kind of data and analyzing it before the game is even out, we can give the designer a strong understanding of the space of all possibilities the player can reach from a specific design and allow them to explore a game's mechanics to their fullest potential.

To sum it up, we are very proud of the results of this research project, as it allowed us to delve into the GOAP algorithm and understand its inner workings by implementing it directly, but also to spin it around enough to get some new and interesting results. For sure, we will be taking much of this acquired knowledge with us into future projects, especially in regards to AI-powered tools for game development.

7 References

7.1 Online Documents:

Orkins, J. 2006. *3 States and a Plan: The AI of F.E.A.R.*. Paper, MIT Media Lab. http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf