Goal-Oriented-Action-Planning

Alex Ceberio, Iñaki Soler

1. Introduction

Game AIs are having a lot of increasing impact in modern games for it's a way to make agents and systems of the games look smart, having a big variety of algorithms depending on the needs of the games. But in many cases, the time put into developing these AIs is minimal, resulting in simplistic and predictable behaviors, which in turn makes the player experience worse overall. With AI being such a big part of many games, it's something that should not be lightly designed, but something that requires planning ahead and investing more time in a good base.

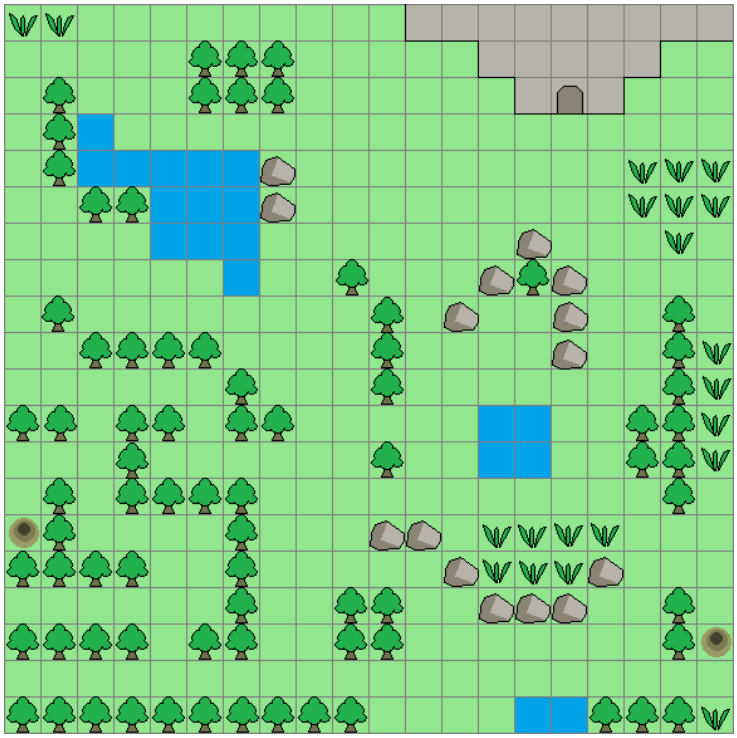
Our research aims to solve this problem by using the GOAP (Goal-Oriented Action Planning) technique, a declarative technique designed by Jeff Orkin in the game F.E.A.R. This will be showcased in a natural environment like simulation, having the animals in it be driven by this algorithm to make them achieve their goals and plan their way to survival.

2. The problem

Many game developers, when they need an AI for their game, immediately gravitate to procedural techniques like State Machines or Behavior trees, as they are easy to understand, implement and use. Although there's nothing inherently wrong with the techniques themselves, and even if on many occasions they're the best approach, sometimes the nature of the desired AI is too hard to be implemented with them, or downright impossible, due to the sheer number of actions and conditions the AI has to face.

Let's put a game like The Sims and its AI as an example. In it, the NPCs have a set of needs to keep in charge of hunger, energy, social meter, hygiene, bladder, and fun meter. For filling each of these, there are a lot of possible actions a sim can take to increase these values. Let's say that there's a sim that is hungry. It could try to enter the Eat state to decrease its hunger. But he is also very tired, so he can't cook a full dinner as it would take too much time, and get even more tired. And he's also starting to get bored, so he must deal with that as well. And then another sim enters the house, so he must go meet him, and… you get the idea. You cannot realistically represent all the range of situations a sim can be in without ending with a spaghettified mess. And let's not even talk about trying to expand it with updates or a DLC. A different technique is needed to tackle AIs like this.

3. The Demo



It consists of a top-down 2D map of the environment the animals live in. It's represented in the form of a grid, with different types of cells for the animals to interact with. The various actions and goals will read from this map to make decisions, let it be with the grass to eat, the small lakes to drink water, and the cave and burrows to sleep in and reproduce. All three animals share a lot of the goals, with small variations between what they eat or where they sleep.

Table 1, list of all actions

Actions	Effect val.	Affects	For what animals
Eat grass	20	Hunger	Rabbit
Eat apple	50	Hunger	Bear
Eat meat	50	Hunger	Bear, wolf
Drink	50	Thirst	Rabbit, Bear, wolf
Go to (food, water, cave…)	Depends on each go to	Nothing	Rabbit, Bear, wolf
Sleep	60	Energy	Hide_Action
Hide	can't be attacked	Nothing	Rabbit, wolf
Mate	creates child	Nothing	Rabbit, Bear, wolf
Attack	Claw: 20 Bite: 40	Health (for enemy)	Bear, wolf

Wander	Nothing	Nothing	Rabbit, Bear, wolf

Table 2, list of all goals

Goals	Condition	For what animals
Eat	Current Hunger + 40	Rabbit, Bear, wolf
Drink	Current Thirst + 50	Rabbit, Bear, wolf
Sleep	Current Energy + 100	Rabbit, Bear, wolf
Kill	Enemy health = 0	Bear, wolf
Mate	Is safe, Has partner	Rabbit, Bear, wolf
Hide	Is safe	Rabbit, wolf
Wander	none	Rabbit, Bear, wolf

4. GOAP representation

There are four important concepts that need to be understood before explaining the decision-making stuff, and those are: “States”, “Actions”, “Goals” and “Agents”.

4.1 States

A state is a way of simplifying the information of the world or the agent into a list of descriptions of what’s being represented and its logical or numerical value. Some examples of states are: (“HasFood”, true), (“Thirst”, 40), (“IsSafe”, false) …

4.2 Actions

It is defined as a set of conditions and effects, represented by using states, for example, the action for “HuntPrey” would look like:

[Cond: {(“PreyInSight”, true), (“PreyHealth”, 0)}, Effects: {(“PreyInSight”, false), (“HasFood”), true}]

4.3 Goals

They are defined by a set of Conditions, (example: “Reproduce” = (“IsSafe”, true), (“HasMate”, true), (“Hunger”, 80)), and a level of priority, which changes depending on the state of the world and the animal.

4.4 Agents:

They are composed of a memory of states and a list of available actions and goals to take.

5. Decision-making

In the initialization part of the demo, all the possible goals and actions an animal can take are created and assigned. This leads to the main bulk of the decision-making of the entities. In it, the animals follow a loop based on the "Sense-Think-Act" principle.

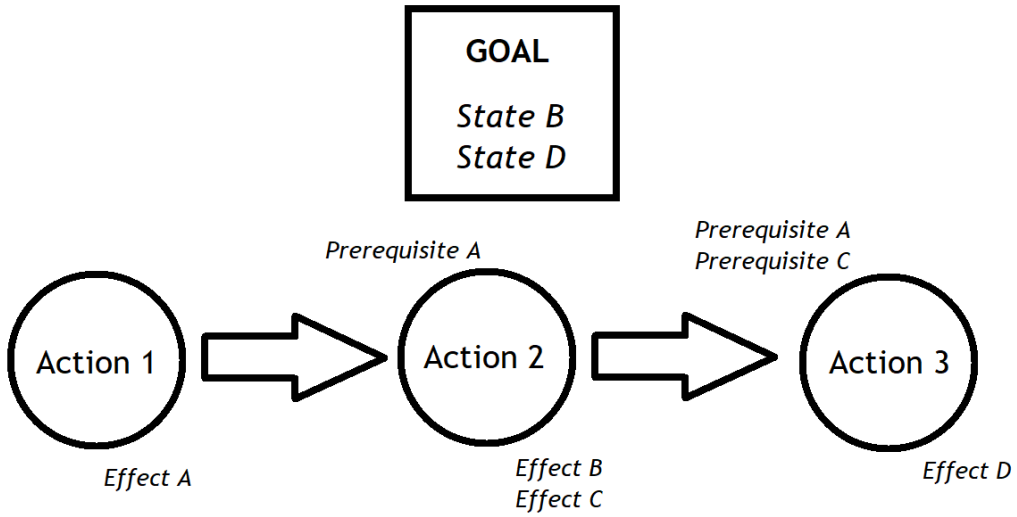
The "Sense" part consists of analyzing all the available goals the animal has. For each one of them, and depending on the logic of each goal, their priorities are calculated depending on the current state of the world and ordered by priority. Here ends the "Sense" part and begins the "Think" part, in which it will search for a series of actions that satisfies the highest priority goal and store it. If this part fails and no plan is found, it will drop to the next goal on the list. Finally, for the "Act" part, it will run the actions on the plan until it finishes, is interrupted by an external source, or fails to execute an action, then proceed with the cycle again.

Listing 1, Part of the Sense-Think-Act cycle.

```
Void Plan(Agent* agent)
{
     for all goals in agent
          curGoal->UpdatePriority();
          if (curGoal->IsPossible())
               possibleGoals.push_back(curGoal);

     possibleGoals.sort();     // sort by priority

     for all possible goals
          startNode = agent->GetMemory();
          agent->plan = AStar->Compute(startNode, curGoal);

          if (!agent->plan.empty())
               return;
     }
}
```

6. Planning: Computing the actions path

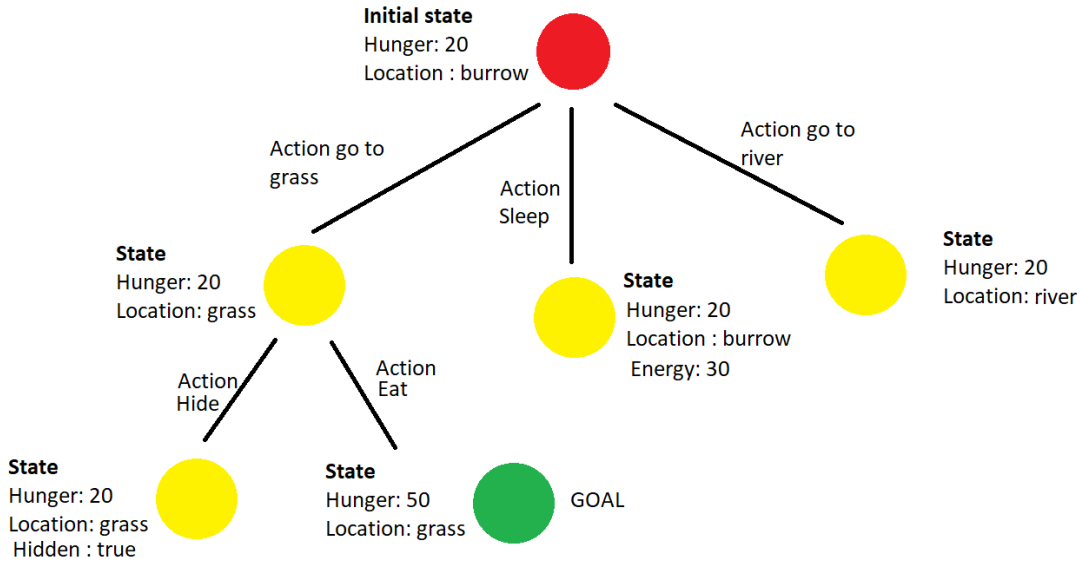
This is one of the most important parts of the GOAP architecture, where the decision-making takes place. After running the algorithm, what we obtain is a directed, weighted tree formed by actions.



In this tree, each node contains an action the animal can take, a link to its parent, a cost, and a list of effects, which is the addition of the effects from its own action plus the one from its parent. The root node is however an exception, as it does not represent any action, and is used to contain the current state of the world. There's also a depth value that serves as a way for the algorithm to not be able to run indefinitely, as we assume that once reached a high enough depth, finding a valid path is unlikely or too costly.

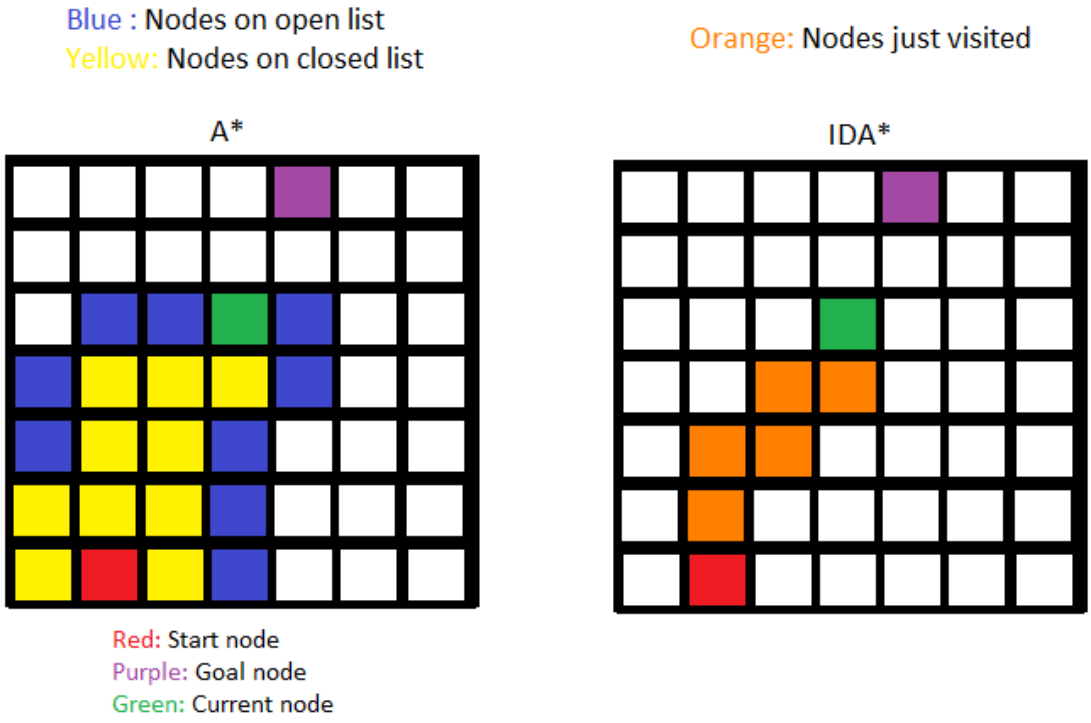
The state-space algorithm used for this project is A*, in which the leaf nodes are stored in an open list, and we explore them based on current cost, determined by the type of action, and a heuristic value, whose computation varies from goal to goal. The closed list comes implicit with the link to the parent node. Finally, for creating the successor nodes, two checks are made before adding the node to the open list. First, the new effects after applying the action have to be different from the parent node (or else we would just create the same node again), and the carried effects from the parent node must satisfy the conditions of the new action

It's important to mention that although we are talking about edges modifying the state



of the world node, at no moment should the actual world state be modified during the planning phase. This presents a new layer of complexity as we have to act on copies of a world model to reach the goal model. Creating exact copies of each piece of data on the world, as accurate as the plans created would be, is unfeasible memory-wise, so a lot of simplification is needed. We can use states to solve this problem, abstracting game data into simple boolean or numerical states. It has the additional effect of being easier to read and write for programmers.

Although for our demo we are using A* as the state-space algorithm, it has some inherent problems in using it here. The first reason why is that, when there's no actual path to the goal, A* would just keep exploring until it runs out of options. This may work with normal pathfinding, but for us, there are always more actions to be taken, which is why a depth limit must be set, as we mentioned before. The second reason is that the number of actions to be taken can be quite large, so memory consumption will exponentially grow as entities become more complex. Because of these problems, IDA* (iterative deepening A*) is likely the best choice. Although slower than A*, it has a space complexity of O(d), with d being the depth of the first solution, while A* has O(b^d) complexity, with b being the branching factor. So it's a speed for memory trade-off.



7. Possible improvements

What is shown in our demo is only a very small fraction of what can be achieved with GOAP. With more time and resources, more features can be implemented in the architecture to make it even more modular and powerful.

- Creating a sensors system: Since agents lack complete knowledge of the world, they may create a plan that runs into a situation mid-execution that requires changing goals and scraping the current plan, like an enemy blocking your way. For these situations, instead of polluting the code with a lot of if-else statements and logic, “Sensors” can be created and added to the agent. They are easy to manage, add and remove, and can also be shared between other types of agents to avoid having duplicated pieces of code. Some examples of this are “EyeSensors”, “SoundSensors” or “DangerSensors”.

- A plan debug mode: Although this is not something that has any impact on the game, a mode of interacting with the architecture, via GUIs or editors, it’s a must-have for anyone planning on using GOAP a lot. Things like being able to manually create plans and execute them, visualizing the created trees by the planner, or checking onto the agents to see their current memory, goals, and actions will help a lot when making a big game or project.
- Speed and memory optimizations: There are a lot of ways in which the performance of our implementation can be improved. Apart from the already mentioned change from A* to IDA*, creating a memory manager to handle node creation and

deallocation, having better cost and heuristic functions, and improving the pruning of new action for nodes, are some of the most obvious changes that should be made to optimize it.

- Improving the heuristics used: For calculating the heuristics of actions and goal priorities we are purely using linear functions and simple if checks. Although it did its job well enough, there's still more thought needed to make the AI feel less rigid and more intelligent.

8. Conclusion

After having worked on this project, we must clarify that the GOAP technique is not a magic tool to suddenly make an AI super intelligent and improve your game overnight. Things such as designing good heuristic functions, designing action effects and conditions, and making the AI act in a way you like are the hardest challenges to be faced when implementing it. A lot of resources are needed for this to show results. Nonetheless, with caution and good planning ahead, it eventually pays off and becomes a really powerful tool at your disposal.

9. References

[Ian Millington and John Funge] Artificial Intelligence for Games:
https://epdf.mx/artificial-intelligence-for-games-second-edition59239.html

[Jeff Orkin] Agent Architecture Considerations for Real-Time Planning in Games
https://alumni.media.mit.edu/~jorkin/aiide05OrkinJ.pdf

[Jeff Orkin] Symbolic Representation of Game World State: Toward Real-Time Planning in Games
https://alumni.media.mit.edu/~jorkin/WS404OrkinJ.pdf

[Panagiotis Peikidis] Demonstrating the use of planning in a video game.
https://pekalicious.github.io/StarPlanner/