El proyecto entregado es una simple demostración técnica del funcionamiento de GOAP, aplicado a un personaje que busca hacerse con una casa propia. El personaje puede:

* Tomar un hacha: Siempre que no la tenga.
* Tomar un martillo: Siempre que no lo tenga.
* Talar árboles: Si cuenta con energía suficiente. Le permite conseguir madera para otras acciones.
* Construir una cama: Si cuenta con madera y energía suficientes.
* Dormir: Siempre y cuando haya construido una cama, el personaje podrá dormir en ella para recuperar energía.
* Construir una casa: Si cuenta con energía, un martillo, y suficiente madera, el personaje podrá construirse su propio hogar.

Cada tarea que realiza el personaje consume un poco de su paciencia, por lo que puede llegar a las acciones "límites":

* Atacar a un enemigo: Si el personaje se encuentra cansado, pero le queda suficiente energía para un ataque, podrá atacar a otro NPC, eliminándolo y robándole la llave de su casa.
* Abrir la casa del enemigo: Una vez que lo haya eliminado, el personaje podrá quedarse con la casa de su enemigo, gracias a la llave que le quitó.
* Invocar a Satanás: La desesperación lo puede todo. Si el personaje se queda sin paciencia, puede rezarle a Satanás para recuperar su energía y ver su casa construída instantáneamente.

A la hora de programar este proyecto, mi primera idea fue hacer algo sumamente escalable, lleno de datos genéricos que yo pudiera moldear para cualquier proyecto que quisiera. Parte del motivo por el cual quería esto era que, al no tener una idea de juego en la cual aplicar GOAP, quería crear algo que pudiera llevarme a otros juegos en forma de librería fácilmente. Sin embargo, terminé complicando de más el proyecto, utilizando diccionarios de objetos y montones de casteos, que llegaban a estar metidos incluso en el código base de GOAP.

Tras corregir estos problemas, pude llegar a un resultado que considero más apropiado, en el cual tenemos una clase de estado con variables tipadas, y una clase de acción que almacena funciones para determinar las precondiciones, efectos, y acciones a ejecutar in-game. Todas estas funciones son determinadas por la entidad en sí, de forma que cada entidad puede tener una idea diferente de qué acciones realizar, con sus propias prioridades, y objetivos y heurísticas únicos. Aún así, me hubiera gustado hacer esto un poco más "modular", de forma que se puedan arrastrar acciones a realizar a cada entidad, o algo similar, para simplificar más el trabajo.

A la hora de implementar A\*, comcencé reutilizando una versión del algoritmo que había adaptado anteriormente, que no utilizaba Linq, simplemente para verificar que las cosas funcionaran sin tener que perder tanto tiempo. Lamentablemente, al hacer el nuevo algoritmo con Linq, para el cual me basé en una de las versiones del algoritmo enseñadas por el profesor Wain, me encontré con el problema de que el algoritmo parece seguir una y otra vez sin fin, por lo que, tras un tiempo intentando encontrar la solución, decidí volver al algoritmo viejo para poder llegar a terminar lo más posible. Como decía Wain, "divide and conquer: primero aseguren el 4, después suban hasta el 10".

Para el movimiento del personaje decidí ir por algo simple; un sistema de Waypoints conectados formando un grafo, y una clase ActionRoutines que contiene las funciones de movimiento de cada acción (todas corrutinas excepto construir la cama). Para poder realizar la progresión de forma fácil y rápida, decidí hacer que cada corrutina llame a la siguiente al terminar. Tal vez hubiera sido más apropiado que el personaje se entere que terminó una reacción y llame a la siguiente, pero esta forma era más rápida.