Automated Playing of Survival Video Games with Commonsense Reasoning

Bryant Hargreaves[1], Dan N. Nguyen[1], Keegan Kimbrell[1] and Gopal Gupta[1]

[1] *The University of Texas at Dallas, Richardson, USA*

Abstract

Don't Starve is a survival video game where the objective is for the player to survive as long as possible without dying. The game is challenging to play due to new situations being randomly generated making survival quite hard. However, the game follows certain rules, that can be employed to automate its playing. Our preliminary effort reported in this paper uses the answer set programming framework to model the logic employed by the game-playing agent to survive in the game for as long as possible. The agent learns the environment around it and translates the knowledge as facts represented as predicates. A set of commonsense reasoning rules represented in ASP capture the logic that must be used to survive. The facts and rules together are used to compute the best next action that the agent must take. We use the s(CASP) goal-directed predicate ASP engine to compute the game-playing actions. The game engine has been customized to interface with the s(CASP) system. Our results indicate that our preliminary automated game playing system is able to outperform novice players. Further refinement should allow our system to play at expert level.

Keywords

Video Games, Answer Set Programming, Commonsense Reasoning

1. Introduction

Video games are popular for children, teenagers, college students, and adults worldwide; however, playing them can require complex thought and reasoning, especially in randomly generated open-world survival games, where the player is presented with countless choices and unique situations. Historically, this complex thinking has been difficult to mimic with software as too many unique scenarios have to be accounted for [1]. Attempts at solving this issue have been made through heuristic algorithms such as the Monte Carlo algorithm or with reinforcement machine learning models, which have achieved limited success in this field. One increasingly popular technique is using pre-trained large language models to simulate human reasoning given facts about the environment, however, its responses can be limited by hallucinations, and its large memory requirements to infer these models make it unfeasible for use in the game industry.

Our motivation for this paper is to develop an autonomous agent that can play a game in which events are randomly generated. The agent will use commonsense reasoning to compute the response. Commonsense reasoning is performed using answer set programming, in particular, the s(CASP) predicate answer set programming engine [2]. Our goal is to show that automated commonsense reasoning realized through ASP and s(CASP) is an excellent tool for automated survival video game playing.

2. Background

2.1. Don't Starve

Don't Starve is a survival video game where the player has to learn to survive in a harsh environment. Don't Starve starts a player out as Wilson, a scientist, who has been mysteriously transported to a

4th Workshop on Goal-directed Execution of Answer Set Programs (GDE'24)

bryant.hargreaves@utdallas.edu (B. Hargreaves); dan.nguyen@utdallas.edu (D. N. Nguyen); keegan.kimbrell@utdallas.edu (K. Kimbrell); gupta@utdallas.edu (G. Gupta)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

strange and deadly world by a demon gentleman. After a quick greeting, the adversaries vanish, and the player has to figure out how to survive [3].

The player's objective is to survive as long as possible. To survive, a player must carry out operations such as scavenging for food, building a campfire to stay warm, etc. If not fed, for example, the player will eventually die. After playing and dying a few times with no help at all from the games, the player will figure out some important mechanics. The core game mechanics that are important when creating Agent Wilson [4] are discussed next.

Figure 1: Player's normal game screen

Hunger: Hunger is represented as a yellow icon, and when it reaches 0, the player's health will drop drastically until the player dies. There are many actions that players perform, e.g., running and chopping trees, that deplete a player's energy. Just staying alive also slowly depletes it. The player's energy is displayed in a *hunger bar*. Players can always eat food to replenish energy and slow down starvation.

Health: Health is represented as a red icon, and when it reaches 0, the player dies. Usually, that means the game is over, except for some scenarios. Health is drained due to hostile mobs, natural disasters like lightning and fire, or hunger. Eating foods might improve the player's health, as shown in the health icon.

Sanity: Sanity is represented as an orange icon, and when it gets near 0, players will be in a hallucinated state, and hostile mobs will keep spawning to attack the player. They can increase/decrease passively due to the Day/Night Cycle, but the most common causes that drain sanity are Evil Flowers, eating raw meat, fighting hostile mobs, and killing animals. The best way to replenish it is to pick up flowers and/or make a garland from these flowers and wear them [5].

Day/Night Cycle: The Day/Night Cycle of Don't Starve is a constant factor. It acts as a sort of clock, and each day will have three cycles: Day (yellow section), Dusk (red section), and Night (blue section). It is generally pretty safe during the day, but when Dusk comes, it will get darker, and the player's Sanity will start to drain slowly. When night arrives, it will become completely pitch dark unless the player has a light source like a torch or campfire. The player's sanity will also start draining during the night, but it will passively recoup some small amounts back when the day comes. Players should have a light source when the Night arrives because an invisible mob will attack and kill the players in two hits.

Inventory: The player's inventory keeps a present record of items gathered, used, and crafted. The inventory also shows if perishable foods (carrots, berries, ...) will go bad soon in 3 colors (green, yellow, and red, with red usually means the food is spoiled and should not be eaten). The player's inventory is limited to 15 items unless the player is equipped with a backpack.

Equipment: The player has 3 slots for equipped items: tool, torso, and hat. The player can use equipped

tools like an axe to chop trees for wood, a spear to hunt animals, or a torch to light up the pitch-dark night. The player can wear a torso piece like armour to reduce the damage they take from hostile mobs or keep warm during the winter season, or backpack to expand their inventory slots. Lastly, the player can wear a hat to increase/keep their sanity, stay warm during winter, etc.

Interaction and Crafting: Interacting with the world is one of the most important elements of Don't Starve. Interacting can mean a variety of different actions, including collecting items, picking berries, gathering flowers, and chopping wood. These actions generally allow the player to obtain items in exchange for the time and stamina used to perform the action. Furthermore, players can use these items in crafting. Crafting is another core mechanism in Don't Starve that enables players to obtain tools and structures for survival. This allows players to dynamically strategize the allocation of randomly generated entities to better prepare for any scenario. The usual action to start is to gather flint and twigs, then craft an axe to chop trees and collect logs, which can be used for making a campfire to survive the night.

2.2. ASP and s(CASP)

Answer Set Programming (ASP) [6, 7] is a logic programming paradigm suited for knowledge representation and reasoning that facilitates commonsense reasoning. The s(CASP) system [2] is an answer set programming system that supports predicates, constraints over non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and compute partial models by returning only the fragment of a stable model that is necessary to support the answer.

Complex commonsense knowledge can be represented in ASP and the s(CASP) query-driven predicate ASP system can be used for querying it [8, 9]. Commonsense knowledge can be emulated using (i) default rules, (ii) integrity constraints, and (iii) multiple possible worlds, according to [7] and [10]. Default rules are used for jumping to a conclusion in the absence of exceptions, e.g., a bird normally flies, unless it's a penguin.

```
1 flies(X) :- bird(X), not abnormal_bird(X).
2 abnormal_bird(X) :- penguin(X).
```

Integrity constraints allow us to express impossible situations and invariants. For example, a person cannot be dead and alive at the same time.

```
1 false :- person(X), dead(X), alive(X).
```

Finally, multiple possible worlds allow us to construct alternative universes that may have some parts common but other parts inconsistent. For example, the cartoon world of children's books has a lot in common with the real world (e.g., birds can fly in both worlds), yet in the former birds can talk like humans but in the latter they cannot.

A large number of commonsense reasoning applications have already been developed using ASP and the s(CASP) system: [11, 8, 9]. Justification for each response can also be given as the s(CASP) system can generate justifications for successful queries as proof trees as shown by [12].

3. Designing Agent Wilson

Our goal in this paper is to automate the playing of the Don't Starve game. For this purpose, we created an autonomous agent that uses automated commonsense reasoning to compute what actions it must take, given the surrounding environment, in order to survive.

The main parts of creating the autonomous agents are (i) understanding the current environment and the agents' characteristics, (ii) converting the environment and characteristics into predicates, (iii) combining the predicates, rules, and constraints to compute the action(s) that the agents should take, and (iv) applying actions, or one of the best actions in order to continue surviving.

The agent will keep a feedback loop going as the game is played. Metrics are always updated after the agent applies an action or the game-state changes on its own. Therefore, these dynamic changes lead to a cycle in which the player understands the world, and uses the knowledge gained to execute an action. The agent repeats the cycle until the agent dies.

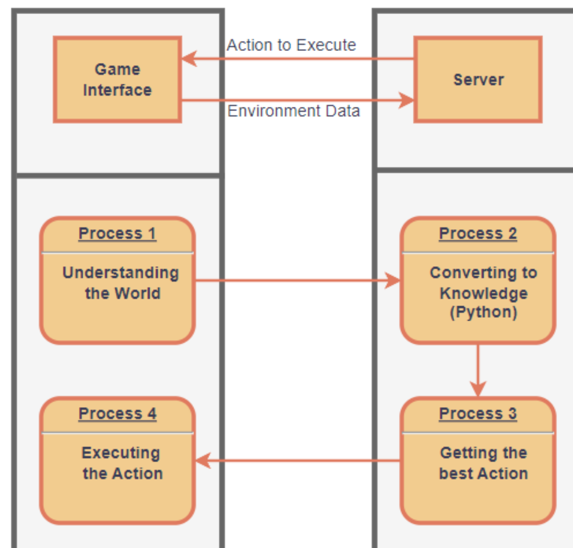

Figure 2: Diagram of Agent Wilson

3.1. Understanding the World

The base version of Don't Starve relies on a day/night system. During the day, the player can safely explore the world and gather materials[13]. For the agent to explore the world, it needs to understand the world around it. Our custom game modification lets the agent know what entities are on the screen and their characteristics. For example, these factors can include whether or not a campfire is still burning, if a berry bush has berries left to pick, or even the size of an oak tree.

Following the safety of daytime is the complete darkness of the night, the player should settle down by creating a fire source and cooking food items [13]. Our custom game modification relays the necessary ingredients needed to craft items such as campfires. This allows the agent to know what is possible to be created. As for understanding food, some materials might be consumable—and to combat food waste, the agent should only consume foods if it is low on points in health, hunger, and sanity. Eating extra food won't help if the agent is at the maximum level for these three measures.

Our custom game modification will send these facts and statistics so the agent can understand them. Don't Starve was created with modifications in mind, allowing developers to create Lua scripts that can influence the game[14]. In our Lua script, we create a repeating event that reads the environment's data, cleans it, and sends it as JSON-formatted text to a Python server that handles the logic coded in s(CASP).

3.2. Converting to Knowledge

To communicate both the game data and the agent's decision-making between each other, we need to transform the information collected from the game world into a logical form that s(CASP) can process. s(CASP) (Constraint Answer Set Programming) is well-suited for reasoning about incomplete information and constraints, making it ideal for complex decision-making scenarios like those in Don't Starve.

The JSON data obtained from the game world is converted into a set of s(CASP) predicates. These predicates represent the state of the world, the agent's current status, and the entities representing its

characteristics. The facts below show that there is an evergreen (a type of tree) that has a GUID(Global Unique Identifier) of 108017. It is workable(can be interacted with via a mouse click), and choppable with an axe since it is a tree, and there is only 1 tree for that entity.

```
1  item_on_screen(evergreen, 108017).
2  guid(108017).
3  workable(108017).
4  choppable(108017).
5  quantity(108017, 1).
```

Second, the agent's inventory data will be converted to predicate. For this example, the agents currently have 2 twigs in one of their inventory slots. Twigs are a fuel source that can keep a campfire burning. If twig(s) are on the ground, they can be collected.

```
6  slot_in_inventory(twigs, 111163).
7  guid(111163).
8  fuel(twigs).
9  collectable(twigs).
10 item_in_inventory(twigs, 2).
```

Third, the agent's equipment data will also be converted to predicate. This told us that the agent is equipping an axe.

```
11 equipment(axe).
12 equipment_guid(axe, 111191).
13 guid(111191).
14 quantity(111191, 1).
15 equippable(axe).
16 collectable(axe).
```

Then, the agent's statistics are mapped to predicates, so if the agent is low on health, it can eat some food in order to regain it. Currently, the agent has a high amount of health, sanity, and hunger points (a low hunger bar point means the agent is hungry).

```
17 sanity(high).
18 hunger(high).
19 health(high).
```

Finally, the world characteristics will also be mapped to predicates. These characteristics are time of day, season, and current biome that the player is standing on.

```
20 time(day, early).
21 season(summer).
22 biome(forest).
```

Once the environment has been converted into these logical forms, the s(CASP) engine can reason to take actions given the current game state. This allows the agent to query possible actions and outcomes based on the current world configuration. The engine evaluates all possible actions, returning a set of feasible solutions sorted by priority that satisfy the agent's survival requirements, such as staying alive, finding food, or avoiding hostile entities.

This conversion process is key to allowing the agent to understand its world in a structured, logical way. By using s(CASP), the agent can not only react to immediate needs but also reason about future states, crafting an intelligent survival strategy in a dynamic, unpredictable environment.

3.3. Computing the Possible Actions

Selecting the possible actions for the agent to perform comes after the world state has been transformed into s(CASP) predicates. Using the current environment, player statistics, and resource availability as inputs, the s(CASP) engine is queried to start the decision-making process. In assessing potential courses of action, the engine considers the agent's objectives: survival through self-management of

hunger, health, and sanity and ensuring necessary resources to survive through the future (nights, seasons, and hostile monsters).

The results of each query are a list of potential actions the agent could take, including "pick_entity," "build," or "run_away." There should always be one or more possible action to take at any time (we will just letting the agent wander around the world if there are no actions to complete). For example, one of the possible actions that can help survive the nights is gathering logs to make a campfire, and to get logs, trees must be chopped. Therefore, one of the rules is:

```
action(chop_tree, chop_tree, GUID) :- choppable(GUID), equipment(axe), good_amount(log, 10), not time(night)
```

Note that this action is taken if there is no better action that is possible. Using the sample from when world data is converted to knowledge:

```
1  item_on_screen(evergreen, 108017).
2  ...
4  choppable(108017).
5  ...
11 equipment(axe).
12 ...
13 time(day)
14 ...
```

This action will hold true, with the model output as:

```
action(chop_tree,chop_tree,108017)
```

Chopping a tree requires the agents to do at least more than 1 chop (or more than one feedback loop). If there is more than one tree that the agents can see, the agent might do 1 chop on this tree and 1 chop on the other tree, and it has to walk in between, or the agent might not notice that tree anymore, and the axe will lose durability with no logs in return. Therefore, we implement a history system that saves the last action the agent took in order to prevent these needless action modifications. The final course of action is re-selected if it is still feasible given the current condition of the world. However, the agent should also stop chopping the tree and run away if they encounter hostile mobs. Here, the agent will ignore the history and discard the previous action so it can "run away" from these hostile mobs.

Sometimes, there might be 2 actions that are the same, but just on different entities—like either foraging a berry bush on the left or foraging a berry bush on the right. In this case, Python will be used to determine which action was pushed first in the queue, and that action will be executed by the agent.

Furthermore, the "sufficient_amount" predicate uses s(CASP)'s powerful negation abilities to accurately model whether or not the agent has enough of a given resource. This predicate holds if the agent has less than the maximum amount of desired items and false otherwise. This way, the agent will stop gathering the same resource it has enough of and instead go for other resources.

```
1 sufficient_amount(X, MAX) :- not item_present(X).
2 sufficient_amount(X, MAX) :- item_in_inventory(X, N), N .>. 0, N .<. MAX.
```

3.4. Executing the Chosen Action

The final step in creating the agent is taking the output from the s(CASP) predicates and acting on the environment. For this, we use a script written in Lua that sends the data to the server to listen for responses. Our system relies on using a set number of predefined actions as stated previously, however, they can be customized by passing arguments specifying entities in the world. These actions include wandering around the world, walking towards entities, picking up items, foraging entities for items, cooking food, equipping and unequipping items, chopping down trees, running away from hostile entities, eating food, adding fuel to campfires, staying close to entities, and crafting items. These actions were chosen to obtain food and shelter while following basic human survival logic.

4. Evaluation

First, our goal is that Agent Wilson must be able to survive for at least 7 days. This is because we utilized the guide from Don't Starve wiki [4]. In creating the agent we discovered that while the guide was useful in general, it didn't include some of the details necessary to catch all cases. For example, the guide instructs readers to avoid spiders and other hostile entities, however, it does not specify the distance a player should keep or how to properly disengage from them. Additionally, the guide instructs the player to collect everything in sight, however, it does not suggest what amounts of resources are necessary or if there should be a balance or limit to the items collected. These small details which are intuitively understood by human reasoning must be represented in s(CASP) logically and consistently. Our ultimate interpretations of these instructions can be found in the final action script run by the server.

Below is a table of Agent Wilson's unassisted attempts at surviving in randomly generated worlds. In testing, we made some minor interventions to the logic of the s(CASP) code when we found a flaw in prioritizing actions, but there were no interventions through the following attempts.

Test	Causes of Death	Days Lived
1	Darkness	4
2	Terrorbreak (Out of Sanity)	7
3	Frog (Hostile Mob)	2
4	Starvation	5
5	Hound (Hostile Mob)	5
6	Starvation	6
7	Frog (Hostile Mob)	7
8	Frog (Hostile Mob)	6

With respect to related work, efforts have been made to automate the playing of the Don't Starve game. Techniques such as machine learning and reinforcement learning have been applied in the past. However, machine learning or reinforcement learning doesn't allow for the understanding of the reasoning behind their actions. For example, deep neural networks create their weights, manage their hidden layers, and only work with numerical values. With reasoning, the logic can be easily understood semantically which can be useful to the majority of video game players with little to no programming experience. Additionally, to make decisions given a vast and complicated world, they must have a large number of internal nodes to understand and output an action to each input scenario whereas with reasoning the rules can be general. This means that it can quickly output actions without fear of undesired results from under-fitting or over-fitting. Finally, with both reinforcement and machine learning, the large requirements for training and storage can deter both developers and players of video games from using reasoning and artificial agents in games altogether. While machine learning may be more scalable in this sense, reasoning is significantly more cost-effective.

5. Conclusion

In this paper we reported on our experience designing an autonomous agent that can play the survival game "Don't Starve" automatically. The agent simulates the commonsense knowledge that a human would use to play the game. It uses answer set programming and the s(CASP) goal-directed predicate ASP system to perform commonsense reasoning. Our evaluation shows that our preliminary implementation can play slightly better than the novice level—surviving for 7 days. Future work includes making our agent's reasoning stronger so that it can survive indefinitely.

References

[1] N. Justesen, P. Bontrager, J. Togelius, S. Risi, Deep learning for video game playing, IEEE Transactions on Games 12 (2017) 1–20. URL: https://api.semanticscholar.org/CorpusID:37941741.

[2] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, 2018. URL: https://arxiv.org/abs/1804.11162. doi:10.48550/ARXIV.1804.11162.

[3] N. Meunier, Don't starve review (2014). URL: https://www.gamespot.com/reviews/dont-starve-review/1900-6407882/.

[4] D. S. Wiki, Guides/getting started guide (2024). URL: https://dontstarve.fandom.com/wiki/Guides/Getting_Started_Guide.

[5] D. S. Wiki, Guides/starting out: A guide for newbies (2024). URL: https://dontstarve.wiki.gg/wiki/Guides/Starting_Out:_A_Guide_For_Newbies.

[6] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103.

[7] M. Gelfond, Y. Kahl, Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach, Cambridge University Press, 2014. doi:10.1017/CBO9781139342124.

[8] Z. Chen, K. Marple, E. Salazar, G. Gupta, L. Tamil, A physician advisory system for chronic heart failure management based on knowledge patterns, Theory Pract. Log. Program. 16 (2016) 604–618.

[9] Z. Xu, J. Arias, Others, Jury-trial story construction and analysis using goal-directed answer set programming, in: Proc. PADL, volume 13880 of *LNCS*, Springer, 2023, pp. 261–278.

[10] G. Gupta, Automating common sense reasoning with ASP and s(CASP), 2022. Technical Report, https://utdallas.edu/~gupta/csr-scasp.pdf.

[11] G. Sartor, J. Davila, M. Billi, G. Pisano, G. Contissa, R. Kowalski, Integration of logical english and s(casp), in: Proc. ICLP Workshops: GDE'22, volume 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022.

[12] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for Goal-Directed Constraint Answer Set Programming, in: Proceedings 36th ICLP (Tech. Comm.), volume 325 of *EPTCS*, 2020, pp. 59–72. doi:10.4204/EPTCS.325.12.

[13] M. Sliva, Don't starve review (2013). URL: https://www.ign.com/articles/2013/05/02/dont-starve-review.

[14] R. Ierusalimschy, L. H. de Figueiredo, W. C. Filho, Lua—an extensible extension language, Software: Practice and Experience 26 (1996) 635–652.