Regressive Arithmetic Goal-Oriented Action Planning

Isaac Hsu (Hsu, Chih-Kang)

Introduction

This paper is intended for game programmers interested in an AI planning technique known as **Goal-Oriented Action Planning** (**GOAP**). Ever since the groundbreaking masterpiece *F.E.A.R.*, this method rapidly spread among video games such as *Fallout 3*, *Empire: Total War*, *Middle-earth: Shadow of Mordor*, and *Rise of the Tomb Raider*. Unlike traditional game AI mechanisms such as finite state machines and behavior trees where the decision making is usually hardcoded, **GOAP** grants AI the ability to formulate plans for goals on their own at runtime and thus decouple decision-making logic from actions and goals. Moreover, the capability to look ahead makes it stand out from traditional approaches that can't predict based on prediction.

GOAP is a planning architecture for AI to find sequences of actions for changing game states to given goal states. However, the original **GOAP** modifies game states only by copying from the effects of actions. If an agent needs to perform an action several times for a goal (e.g. getting enough resources to do something), the relation of **partial satisfaction** between the action and goal cannot be directly formulated in the original algorithm. Therefore, I would like to propose an extended **GOAP** with arithmetic operations for expressiveness and flexibility, because this kind of use cases could be very common and useful, especially in RPG, strategy and simulation games.

I'll focus on only the planning algorithms of **GOAP** and won't elaborate on goal selection and plan execution since they are already mentioned in the original **GOAP**. Besides, developers can freely devise new processes according to their game design or the features of their game systems and engines. For example, goal selection could be based on user-defined action/goal types, utility scores, or cost of feasible goals, and plan execution could rely on behavior trees, scripts, or event-driven systems.

Previous Work

Because this article is based on Jeff Orkin's **GOAP** in "*Applying Goal-Oriented Action Planning to Games*", I would like to briefly review it first. It is an AI planning technique where the problem domain is comprised of **goals** and **actions** and solving the problems is to find sequences of **actions** to achieve the given **goals**.

- **World Property**: Relevant information for an AI agent to make decisions, e.g., locations of agents and types of current weapons.
- **World State**: Collection of **world properties** that the agent holds.
- **Goal**: Desired **world state** for the agent to achieve. The agent may have a number of **goals** at the same time with a procedure of goal selection.
- **Action**: Something the agent can do to modify its **world state**. The changes an **action** makes can also be described as **world state** and are called its **effects**. An **action** may have optional **preconditions** to be met. For instance, a melee attack requires the target to be at close range. **Preconditions** may be expressed as **world states** as well. For complex conditions, **GOAP** systems usually provides a way for developers to customize extra procedural **preconditions**.
- **Plan**: Sequence of **actions** to execute for a certain **goal**.

Finding a solution to reach a given goal **state** from a certain starting **state** is like searching in a state space tree where the edges represent the **actions** taken and the nodes represent the **world states** after those **actions** on the path to the root are carried out. We can use **A*** to find the shortest path like pathfinding, but in this case, the path is actually a sequence of **actions** to produce the goal **state**.

Laser Example

To show you what I mean, I'm going to use the laser example from Orkin's paper to illustrate the search process as many readers may already know it and I can save time from explaining it. But I'll omit some of the **actions** for economy of space.

The goal for the agent is to kill the target and there is a laser weapon that requires a generator to be switched on. The agent has four possible **actions** as follows:

actions		At	Switch	Target
GotoLaser	**preconditions**			
	effects	laser		
GotoGenerator	**preconditions**			
	effects	generator		
ActivateGenerator	**preconditions**	generator		
	effects		on	
FireLaser	**preconditions**	laser	on	
	effects			dead
start		other	off	alive
goal				dead

Table 1. The **actions** and **world states** of laser example

We can apply A* to search from the starting **state** for a path to the goal **state** like the following state space tree:

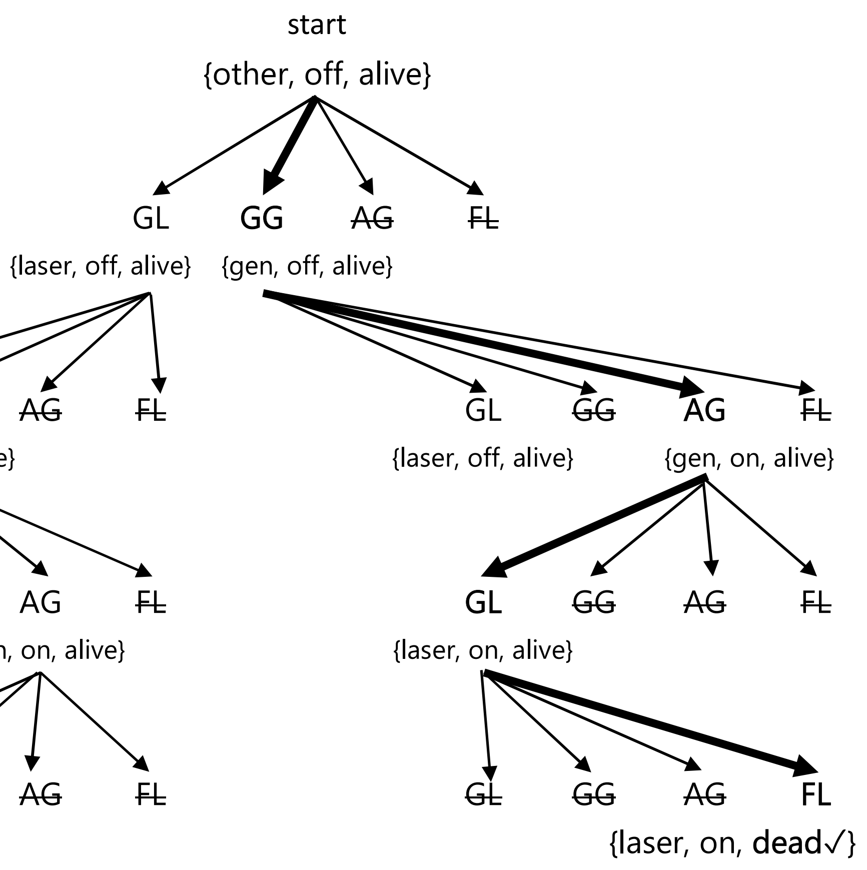
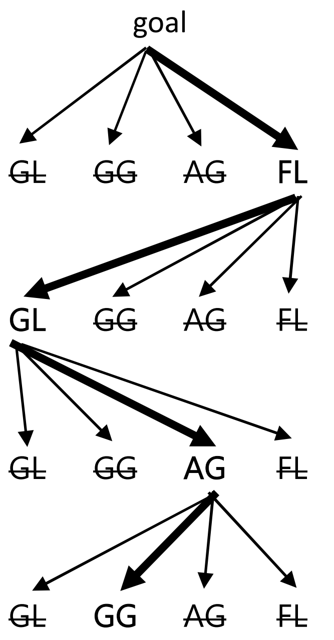


Figure 1. Laser example in forward search. To simplify the figure, each **action** is allowed only once.

Navigational pathfinding can be done from start to goal, or reversely from goal to start, and so can **GOAP**. In fact, it's usually more efficient to do it backwards because previous possible **actions** in backward search are usually less than next possible **actions** in forward search. Consequently, backward search space is often smaller. When we regress from goal, we keep track of current and desired **states** and find the previous possible **actions** whose **effects** satisfy at least one of the unsatisfied desired **properties** and whose **effects** and **preconditions** have no conflicts with the unsatisfied desired **properties**. The **preconditions** of the previous possible **actions** are considered as further desired **properties** to satisfy. When all desired **world properties** are satisfied, the solution is found to be the **actions** on the path to the root. You can see in the laser example there is only one valid regressive path from the goal in the following figure:



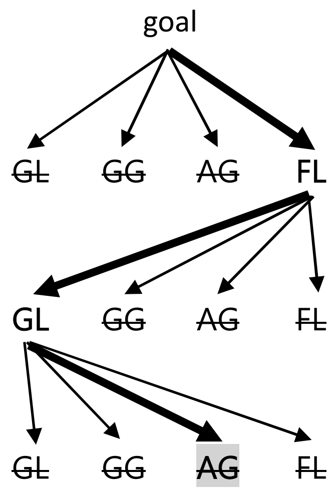
	At	Switch	Target
current			alive
goal			dead
?→**FL**	At	Switch	Target
current	other	off	dead✓
desired	laser	on	dead✓
?→**GL**→FL	At	Switch	Target
current	laser✓	off	dead✓
desired	laser✓	on	dead✓
?→**AG**→GL→FL	At	Switch	Target
current	laser	on✓	dead✓
desired	generator	on✓	dead✓
GG→AG→GL→FL	At	Switch	Target
current	generator✓	on✓	dead✓
desired	generator✓	on✓	dead✓

Figure 2. Laser example in backward search

Improved Regressive Algorithm

In the preceding backward search, for each previous possible **actions**, their **effects** are copied to the current **states** if the **effects** match the desired **properties**, and then their **preconditions** are copied to the desired **states**. However, this approach has a limitation that there shall not be inconsistency between the **preconditions** and **effects** of the same **actions**. To show that, let's add a **precondition** of Switch off to ActivateGenerator in the laser example and see what will happen:

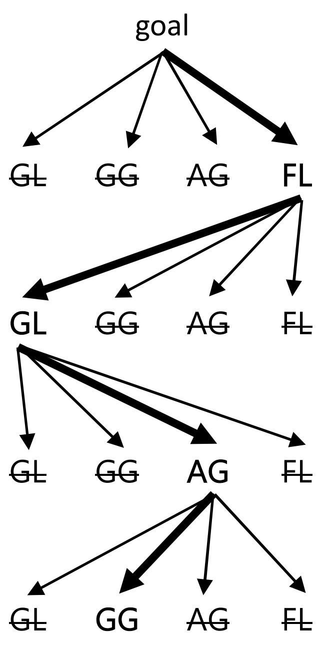
ActivateGenerator	**precondition**s	generator	off✗	
	effects		on	



	At	Switch	Target
current			alive
goal			dead
?→**FL**	At	Switch	Target
current	other	off	dead✓
desired	laser	on	dead✓
?→**GL**→FL	At	Switch	Target
current	laser✓	off	dead✓
desired	laser✓	on✗	dead✓
?→~~**AG**~~→GL→FL	At	Switch	Target

Figure 3. Laser example with an extra **precondition** of Switch off in backward search

The **action** of ActivateGenerator would become infeasible before GotoLaser and FireLaser with the seemingly innocent modification due to the discrepancy between its **precondition** of Switch off and the desired **property** of Switch on. Even if we allow the action, the current value of Switch would be on and the desired value would be off, which is still a problem for the planner. On the contrary, the forward search works just fine under the additional **precondition**. Ideally the backward search should be a more performant replacement for the forward search without extra limitation. To fix that issue, we can simply remove satisfied **world properties** from the desired **states** and check whether the starting **state** satisfies the desired **states** instead to see if a solution is found. Moreover, the current **states** on nodes are no longer needed and can be removed, which is beneficial for space demand of the improved algorithm.



	At	Switch	Target
start	other	off✓	alive
goal			dead
?→**FL**	At	Switch	Target
desired	laser	on	~~dead~~

?→GL→FL	At	Switch	Target
desired	~~laser~~	on	

?→AG→GL→FL	At	Switch	Target
desired	generator	~~on~~→off	

GG→AG→GL→FL	At	Switch	Target
desired	~~generator~~	off✓	

Figure 4. Laser example with an extra **precondition** of Switch off in improved regressive search

In forward search, it's easy to see that the **world states** of nodes are the **world states** modified by the **actions** on the path to the root. On the other hand, what do the **world states** of nodes mean in backward search? To understand it, the key idea is that one step in regressive **GOAP** means to **undo** the **action** in question. For instance, when we regress from the goal in the laser example, the only feasible **action** for dead Target is FireLaser, and being at laser and Switch on would be the previous desired **state** to satisfy as FireLaser is undone. Keep this idea of undoing **actions** in mind as I'll extend the regressive algorithm for arithmetic operations with it.

Workarounds without Arithmetic Operations

In the laser example, you may notice that the planning algorithm is not limited to only Boolean since the **world property** of At is actually an enumeration of three possible values: other, laser, or generator. In fact, you can use **world properties** of any cloneable and comparable types such as floating-point numbers, strings and game object handles as you wish. However, in original **GOAP** you can only have **actions** with **effects** that copy values but not manipulate values even if it's just simply plus one. Conceptually this means you can't formulate a goal that requires an **action** to carry out more than once to achieve it without some tricks.

Let's say in a turn-based simulation game an AI agent wants to buy a house that costs $1000. It can make money by working for others (+$100), running a shop (+$300), or taking out a loan (+$700). Moreover, running a shop and taking out a loan both require non-negative credit score and a loan will decrease its score a lot so it's only possible once for a while. A small amount of credit score can be gained by working (+10). The agent can either work ten times or run a shop four times to get enough money for a house. The fastest solution obviously is to run a shop and then take out a loan, but not in reverse order due to the credit limit of loan.

One way to adapt the preceding algorithm for this instance is to define multiple Boolean **world properties** that represent different thresholds, e.g. whether its money is more than 100, 200, 300, …, and 1000, respectively, so that the **preconditions** and **effects** can be formulated. Nonetheless, this workaround requires some bookkeeping and does not scale very well and may not be suitable in some cases.

An alternative could be to define a Boolean **world property** to tell if the money is enough. Then we subclass the **world states** for extra fields of balance and credit score and subclass the **actions** for procedural **preconditions** of balance and credit check. Although the balance value is hidden to the planner, we can override the cost functions to make **A*** prefer getting a loan. Nevertheless, none of the three **actions** can earn $1000 at once, so we have to fake and set the Boolean **property** to true in their **effects** for the planner to be able to find a solution. But when the plan is executed, it will fail to buy a house at the first time and trigger replanning. Eventually it will get enough money in the successive retries.

actions		EnoughMoney	HasHouse	(Balance)	(Credit)
BuyHouse	**preconditions**	true		≥1000	
	effects		true	-1000	
Work	**preconditions**				
	effects	true		+100	+10
RunShop	**preconditions**				≥0
	effects	true		+300	
GetLoan	**preconditions**				≥0
	effects	true		+700	-70
start		false	false	10	0
goal			true		

Table 2. The **actions** and **world states** of house example with additional hidden numeric fields

Notwithstanding, since the planning algorithm still can't see all the information with this trick, it may result in local optimal solutions instead of global optimal ones. Imagine that the first plan could be to get a loan and then the second plan could be to work three times, which is not the best solution. Hence, it would be better if we can support numeric **world properties** with arithmetic operations such as addition and subtraction in **GOAP** for this kind of needs.

Forward Arithmetic GOAP

Regarding the operations on **world properties**, the original **GOAP** only requires the **equality** comparator for **preconditions** and the **assignment** operator for **effects**. For numeric **world properties**, we can consider also supporting the **four arithmetic operations** for **effects** and **less than or equal to** and **greater than or equal to** for **preconditions**, which can be used to express ranges of possible values instead of just single values. Since the results from arithmetic operations could surpass the goal values, they are more useful than the equality comparator.

	comparators for **preconditions**	operators for **effects**	
general **properties**	==	=	
numeric **properties**	≤	+	-
	≥	×	÷

Table 3. The comparators and operators on general and numeric **world properties**

House Example

With the arithmetic operations, we can redefine the house example as follows:

actions		House	Balance	Credit
BuyHouse	**preconditions**		≥1000	
	effects	+1	-1000	
Work	**preconditions**			
	effects		+100	+10
RunShop	**preconditions**			≥0
	effects		+300	
GetLoan	**preconditions**			≥0
	effects		+700	-70
start		0	10	0
goal		≥1		

Table 4. The **actions** and **world states** of house example with arithmetic operations

Since all the conditions can be exposed to the algorithm now, the planner knows that RunShop has to be done before GetLoan and can find the best solution.

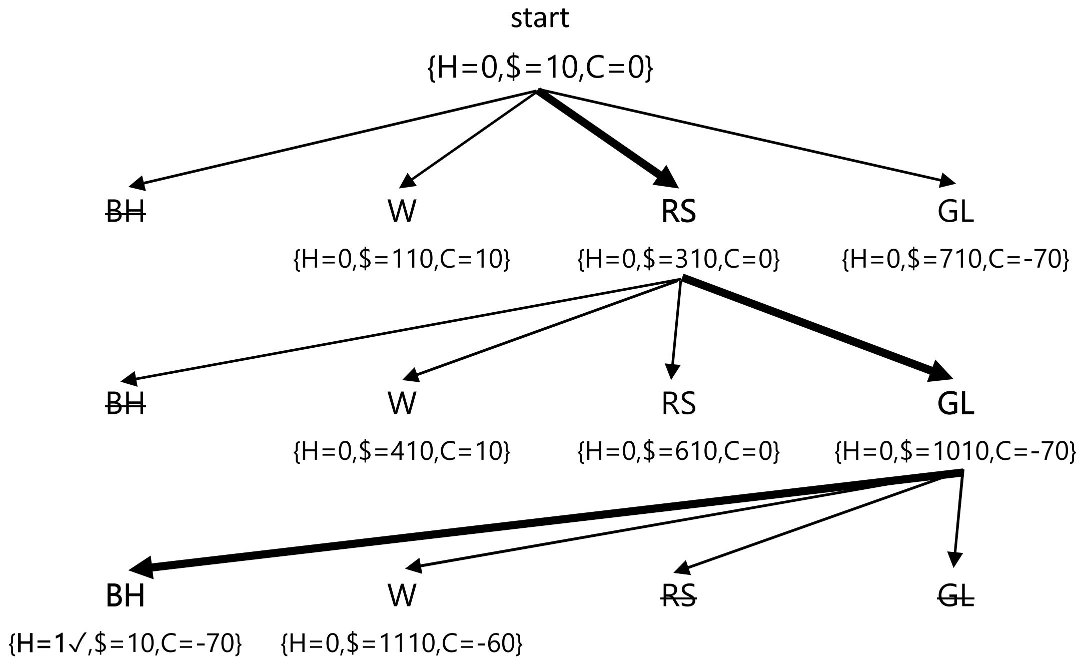


Figure 5. House example with arithmetic operations in forward search. Because of limited horizontal space, only the sibling branches of the shortest path in the state space tree is shown.

Regressive Arithmetic GOAP

Regressive arithmetic **GOAP** is similar to the aforementioned backward search where we regress from goal and undo previous possible **actions** and retrodict previous desired **world states**. In addition, we need to support the two additional comparators and the four arithmetic operators as well.

Addition

To undo the **effects** of addition, we can generalize the problem in the following inequality and solve it:

$a_1 \le x+n \le a_2 \Rightarrow a_1-n \le x \le a_2-n$

, where x is the **world property** in question, +n is the **effect**, and a_1 & a_2 are the lower & upper bounds of current desired value and could be $-\infty/\infty$ if unbounded.

For instance, to undo BuyHouse from the goal **state** and deduce the previous desired value in the house example:

$1 \le \text{House}+1 \le \infty \Rightarrow 0 \le \text{House} \le \infty$

, which can be translated as no House is needed before the **action** of BuyHouse.

Subtraction

Subtraction can be converted to addition by the following equation and solved like addition:

$x-n = x+(-n)$

Multiplication

Similar to addition, we can formulate the problem of multiplication and solve it:

$a_1 \le x \times n \le a_2$

$$\Rightarrow \begin{cases} a_1/n \le x \le a_2/n & \text{, if } n > 0 \\ a_1/n \ge x \ge a_2/n & \text{, if } n < 0 \\ x=\varnothing & \text{, if } n = 0 \text{ \& } 0 \notin [a_1, a_2] \\ -\infty \le x \le \infty & \text{, if } n = 0 \text{ \& } 0 \in [a_1, a_2] \end{cases}$$

In the last case, because anything times zero equals zero, x could be any number if the multiplicand is zero and the desired value could be zero.

Division

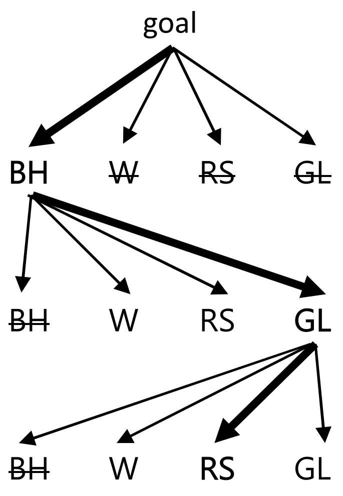
Division can be converted to multiplication as follows:

$x \div n = x \times (1/n)$

Be aware of zero divisor though, the **effect** is invalid since divide-by-zero is undefined.

House Example using Regressive Arithmetic GOAP

Let's go back to the previous house example and see how the regressive state space tree would look like:



	House	Balance	Credit
start	✓0	✓10	✓0
goal	≥1		
?→BH	House	Balance	Credit
desired	≥0=1-1	≥1000	

?→GL→BH	House	Balance	Credit
desired	≥0	≥300=1000-700	≥0

RS→GL→BH	House	Balance	Credit
desired	✓≥0	✓≥0=300-300	✓≥0

Figure 6. House example with arithmetic operations in regressive search

You may notice that unlike the original **GOAP**, the results from arithmetic operations are ranges of possible values instead of single values and they are not removed from the desired **states**. In addition, when copying the **preconditions** to the desired **states**, for existing values we use their common solutions instead of just overwriting, which are the intersections of their ranges (e.g. Table 7).

Range Limits of Numeric World Properties

In practice, it may be useful to put range limits on **world properties** in some cases. Suppose that an agent has two **actions** of Work and Rest and wants to earn money in a game, we can apply forward **GOAP** as the following table:

forward search		Money	Fatigue
Work	**preconditions**		≤0
	effects	+5	+2
Rest	**preconditions**		≥1
	effects		-3
start		0	4
goal		✓≥8	
R		0	1=4-3
R→R		0	-2=1-3
R→R→W		5=0+5	0=-2+2
R→R→W→W		✓10=5+5	2=0+2

Table 5. The **actions** and **world states** of range example without range limits in forward search

However, Fatigue is disallowed to be negative in the game. Even though there's already a **precondition** put on Fatigue for Rest, that is a **pre**-condition instead of **post**-condition so it doesn't restrain the outcome of the **action**. If we put range limits on the numeric **world properties** and clamp the results after the **effects** are applied, we can avoid out-of-range and find a correct plan as follows:

forward search		Money	Fatigue
ranges		*≥0*	*≥0*
Work	**preconditions**		≤0
	effects	+5	+2
Rest	**preconditions**		≥1
	effects		-3
start		0	4
goal		✓≥8	
R		0	1=clamp(4-3*, 0, ∞*)
R→**R**		0	0=clamp(1-3*, 0, ∞*)
R→R→**W**		5=clamp(0+5*, 0, ∞*)	2=clamp(0+2*, 0, ∞*)
R→R→W→**R**		5	0=clamp(2-3*, 0, ∞*)
R→R→W→R→**W**		✓10=clamp(5+5*, 0, ∞*)	2=clamp(0+2*, 0, ∞*)

Table 6. The **actions** and **world states** of range example with range limits in forward search

Regressive Algorithm with Range Limits

Implementing range limits for forward search is pretty straightforward. Nevertheless, to support it for backward search, we have to undo the additional clamping of range limits before arithmetic operations since it's the post-operation after them in forward search and we have to do it reversely in regressive search.

If a range limit doesn't intersect the interval of current desired value, then there is no valid solution to previous desired value. Otherwise, there could be a solution within their intersection that can be deduced with the aforementioned process of undoing arithmetic operations. In addition, if the minimum of a range limit lays in the current desired interval, it will unbind the lower bound because that means even negative infinity will be clamped to the minimum and thus it still can have a solution. It's similar for the maximum too. The unclamping can be formulated as follow:

$a_1 \leq \text{clamp}(x*n, r_1, r_2) \leq a_2$, where $* \in \{+,-,\times,\div\}$ and r_1 & r_2 are the range limit.

$$\Rightarrow \begin{cases} x=\varnothing & \text{, if } [r_1,r_2]\cap[a_1,a_2]=\varnothing \\ a_1` \leq x*n \leq a_2` & \text{, if } [r_1,r_2]\cap[a_1,a_2]\neq\varnothing \end{cases}$$

$$\text{, where } \begin{array}{l} a_1`=(r_1\in[a_1,a_2])\ ?\ -\infty : a_1 \\ a_2`=(r_2\in[a_1,a_2])\ ?\ \infty : a_2 \end{array}$$

After the unclamping and undoing operations, the resulting **world states** should be checked against their range limits owing to the fact that numeric **world properties** are always constrained within their ranges by definition. If there is no intersection between the result and the range, the **effect** is invalid and the **action** is not feasible.

Let's check how backward search works in the previous example:

backward search		Money	Fatigue
ranges		*[0,∞)*	*[0,∞)*
Work	**preconditions**		(-∞,0]
	effects	+5	+2
Rest	**preconditions**		[1,∞)
	effects		-3
start		0✓	4✓
goal		[8,∞)	
?→W		unclamp([8,∞), *[0,∞)*)=[8,∞) [8,∞)-5∩*[0,∞)*=[3,∞)	(-∞,0]∩*[0,∞)*=0
?→~~W~~→W		unclamp([3,∞), *[0,∞)*)=[3,∞) [3,∞)-5*∩[0,∞)*=[0,∞)	unclamp([0,0], *[0,∞)*)=(-∞,0] (-∞,0]-2∩(-∞,0]*∩[0,∞)*=∅✗
?→R→W		[3,∞)	unclamp([0,0], *[0,∞)*)=(-∞,0] (-∞,0]+3∩[1,∞)∩*[0,∞)*=[1,3]
?→W→R→W		unclamp([3,∞), *[0,∞)*)=[3,∞) [3,∞)-5*∩[0,∞)*=[0,∞)	unclamp([1,3], *[0,∞)*)=[1,3] [1,3]-2∩(-∞,0]∩*[0,∞)*=0
?→~~W~~→W→R→W		unclamp([0,∞), *[0,∞)*)=(-∞,∞) (-∞,∞)-5*∩[0,∞)*=[0,∞)	unclamp([0,0], *[0,∞)*)=(-∞,0] (-∞,0]-2∩(-∞,0]*∩[0,∞)*=∅✗
?→R→W→R→W		[0,∞)	unclamp([0,0], *[0,∞)*)=(-∞,0] (-∞,0]+3∩[1,∞)*∩[0,∞)*=[1,3]
R→R→W→R→W		[0,∞)✓	unclamp([1,3], *[0,∞)*)=[1,3] [1,3]+3∩[1,∞)*∩[0,∞)*=[4,6]✓

Table 7. The **actions** and **world states** of range example with range limits in regressive search. The values of **world properties** are formulated in interval here for ease of comprehension.

Implementation Considerations

As you can see in the previous example, numeric **world properties** can be formulated in interval for undoing the four arithmetic operations, so they can actually be coded as interval type. Boolean and enumerative values can also be expressed as degenerate intervals. However, not all the results of undoing numerical operations and functions can be formulated just in interval.

The heuristic function of original **GOAP** is the number of unsatisfied **world properties**. For numeric **world properties**, since they are generalized in interval, we can adopt interval gaps instead and sum them up so that **A*** would prefer **effects** closer to the desired **world states**. The gap between two intervals can be calculated as the following equation:

$$= \begin{cases} 0 & \text{, if } [a_1, a_2] \cap [b_1, b_2] \neq \varnothing \\ \min(|a_1 - b_2|, |a_2 - b_1|) & \text{, if } [a_1, a_2] \cap [b_1, b_2] = \varnothing \end{cases}$$

It's noteworthy for numeric **world properties** with wide range, it could be a good idea to scale down the results of interval gaps with weights less than one. Otherwise, the heuristic cost would be overestimated too much, and **A*** may not find the best solution in some edge cases.

In **A***, for each iteration we need to find out adjacent nodes, which represent feasible **actions** in **GOAP**. In backward search, we can loop through all the **actions** and check their **effects**, or alternatively we can build lookup tables before the iterations and look up desired **world properties** for candidate **actions** in the tables. In addition, for Boolean and enumerative **properties** with only constant **effects**, we can build **effect** tables out of all **actions** and rule out impossible desired values if they are neither in the tables nor in the starting **state**. For numeric **properties**, we can build **effect** direction tables instead to tell whether the **properties** could be increased or decreased or both by the **effects** of all **actions**, and then rule out impossible desired values if the starting **state** is unachievable due to the only **effect** directions of the desired **properties** opposite to the required ones.

Although pathfinding algorithms based on **A*** usually build **closed sets** to prevent double-back, it may not be as useful for **GOAP** since exploring adjacent nodes for already used **actions** on the path doesn't mean revisiting the same nodes, but just redoing the **actions**. The results could be different, and it's necessary in case of partial satisfaction, where the **actions** require to perform several times to satisfy the desired **world states**. On the other hand, it could be helpful to put limit on the execution times in cases where the **actions** should not be permitted more than once or certain times. This can reduce the search space, which is good for performance.

It could be advantageous to allow customization of **world states**, **goals**, and **actions**. Imagine that an agent using **GOAP** to get guns and ammunition can store its planning positions in the subclasses of **world states** and override **A*** cost functions of its **actions** for the additional cost from distance between the planning positions to find the shortest route among several nearby candidates.

Since subtraction is the inverse of addition and division is the inverse of multiplication, we can implement only one for each pair and convert another to the counterpart. It may be useful to allow numerical operation composition in **effects**.

Starting **states** are necessary input as goal **states** to deduce correct plans, so it's important to make sure that agents have the latest **world states** before planning.

Conclusion

GOAP is a great AI technique for agents to devise action plans at runtime to deal with dynamic game environments and for developers to add diversity among AI without exponential complexity. By introducing arithmetic operations and range limits of numeric **world properties**, we can enhance the applicability and competency of **GOAP** and still keep it performant with the proposed regressive search and lookup tables.

Moreover, the regressive algorithm is extensible as we can add more mathematical operations by finding their solutions. A simple case such as the solution to Boolean negation can be deduced as follows:
$\neg x=b \Rightarrow x=\neg b$

Reference

Orkin, J. (2003) *Applying Goal-Oriented Action Planning to Games*

Links of Example Code

You can find C++ code of the mentioned examples in this article via the following links:

- Laser example
  https://github.com/Isatin/GOAPPrototypes/blob/main/GOAP/LaserExample/LaserExample.cpp
- House example
  https://github.com/Isatin/GOAPPrototypes/blob/main/ArithGOAP/HouseExample/HouseExample.cpp
- Range example
  https://github.com/Isatin/GOAPPrototypes/blob/main/ArithGOAP/RangeExample/RangeExample.cpp