Decision Making: Planning

Matthew Guzdial

Announcements

- Practice Quiz today!
- Monday in-class discussion of “Future of Game AI” topics (more on this later)
- Monday I’ll release a helper video on Assignment 3
- Lab next Thursday! (in-person!?)
- Quiz 3 next Friday
- Assignment 3 due Oct 25th

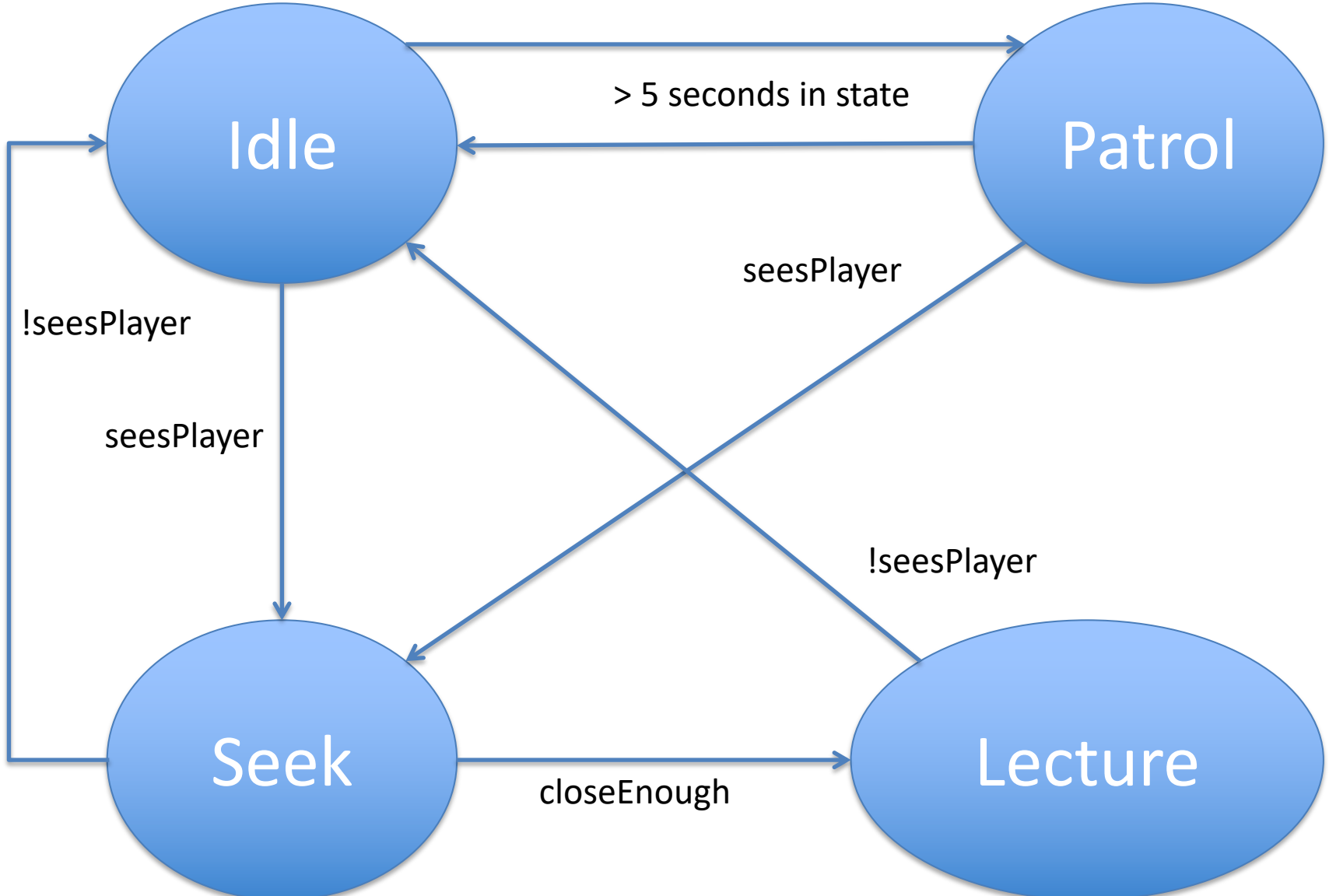
Review: Decision Making

- How an AI picks the next action to take
  - Move, shoot, farm, sleep, etc

- Different techniques
  - Finite state machines
  - Behavior trees
  - Rule systems

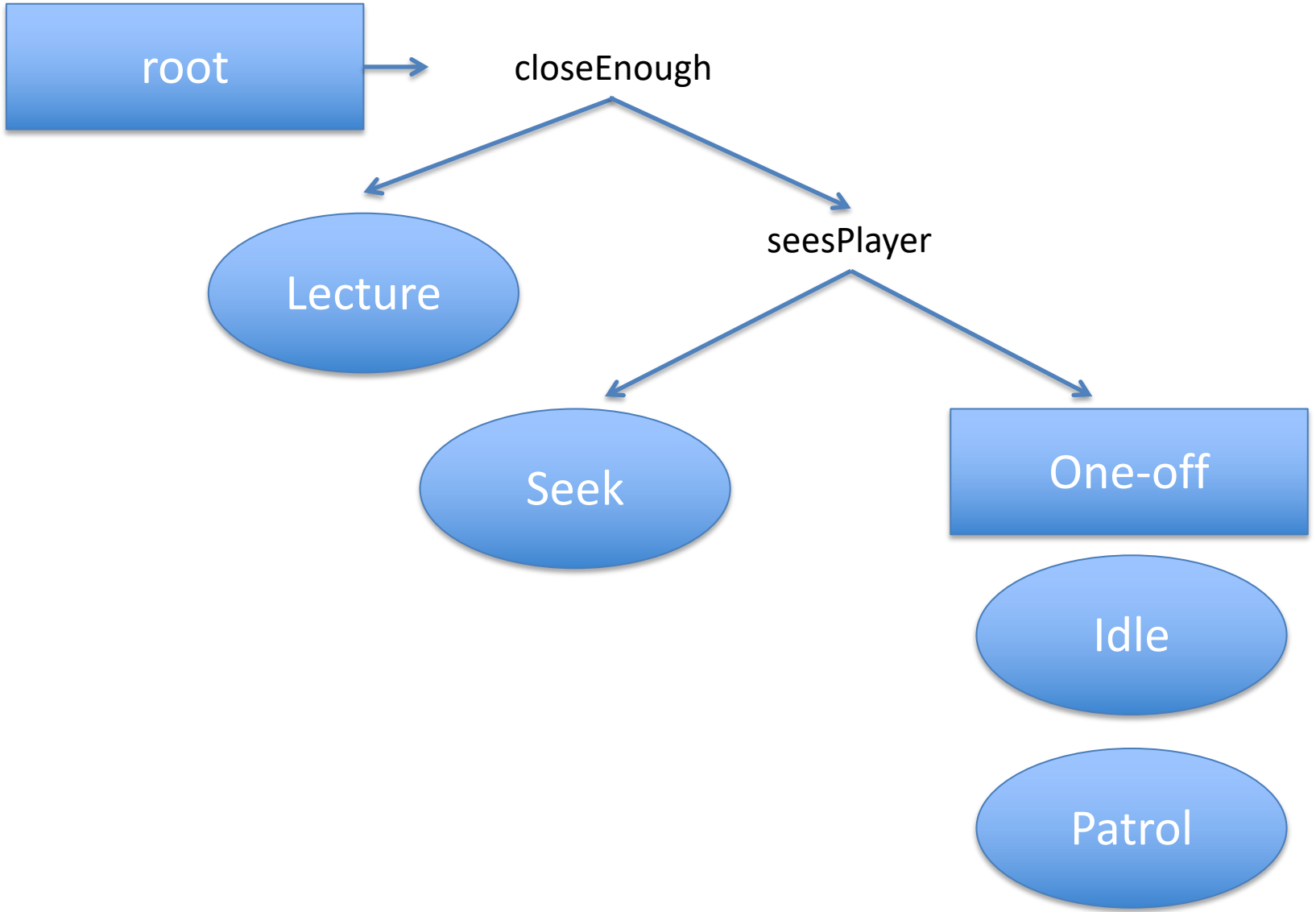
Finite State Machines:

based on the current game state, move to different action or keep at this one



Behavior Trees

based on the current game state find the appropriate action



Rule Systems

based on the current game state, activate a set of rules, pick from those based on some heuristic (e.g. best matches conditions)

Rule Matching

Query

```
{ who: nick, concept: onHit, curMap:circus, health: 0.66, nearAllies: 2, hitBy: zombieclown }
```

PASS Rule1: { who = nick, concept = onHit } → *"ouch!"*

FAIL Rule2: { who = nick, concept = onReload } → *"changing clips!"*

FAIL Rule3: { who = nick, concept = onHit, health < 0.3 } → *"aaargh I'm dying!"*

PASS Rule4: { who = nick, concept = onHit, nearAllies > 1 } → *"ow help!"*

PASS Rule5: { who = nick, concept = onHit, curMap = circus } → *"This circus sucks!"*

PASS Rule6: { who = nick, concept = onHit, hitBy = zombieclown } → *"Stupid clown!"*

PASS Rule7: { who = nick, concept = onHit, hitBy = zombieclown, curMap = circus}
→ *"I hate circus clowns!"*

Current Problems

- Shallowness
- Adaptability
- Heavy Design Burden

Rules: Reminder

- Condition (if): The set of facts that must be true for the rule to be used
- Effect (then): The changes made to the world
- A rule is *activated* if its conditions are met in this state (it can, but doesn't have to be used)

- Examples:
  - **Unlock Door:** *if* hasKey *then* doorLocked = false
  - **Shoot Enemy:** *if* hasGun *&&* hasAmmo *&&* raycast(self, enemy) *then* enemy.health -= gun.damage

Chain multiple rules together?

- Rules:
  - **Unlock Door**
  - **Shoot Enemy**
- What if we want to get through the door, but the enemy has the key?
- Chaining multiple actions together in a sequence is called a **plan**

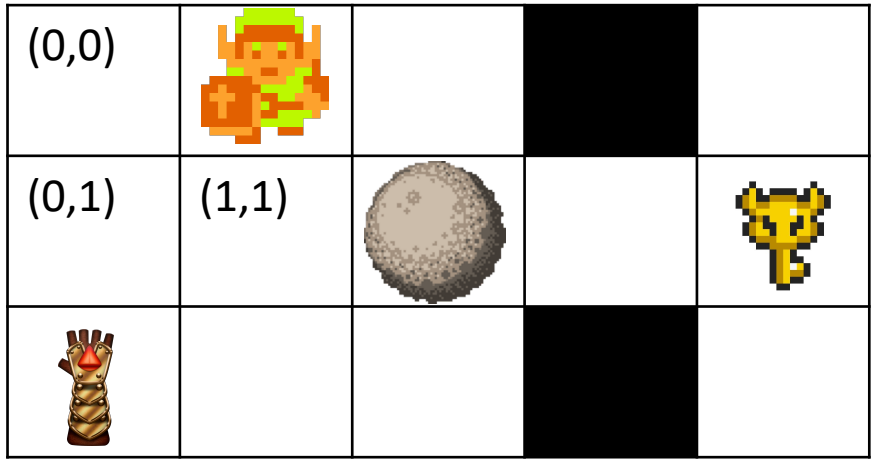
Solution: Planning

- Give each agent a **goal** or set of goals
  - Condition(s) to reach to stop planning
  - E.g. “kill the player”, “make money”, etc.
- Give the agent access to a pool of actions with causes and effects (**rules**)
- Sequence of actions to reach said goal (**plan**)

Examples of Plans

- Route search: Find a route from dining hall to library during construction
  - *See all path planning lectures*
- Military operations
- Outlining a new coding project
- Coming up with dance choreography
- Composing a piece of music
- Writing a script

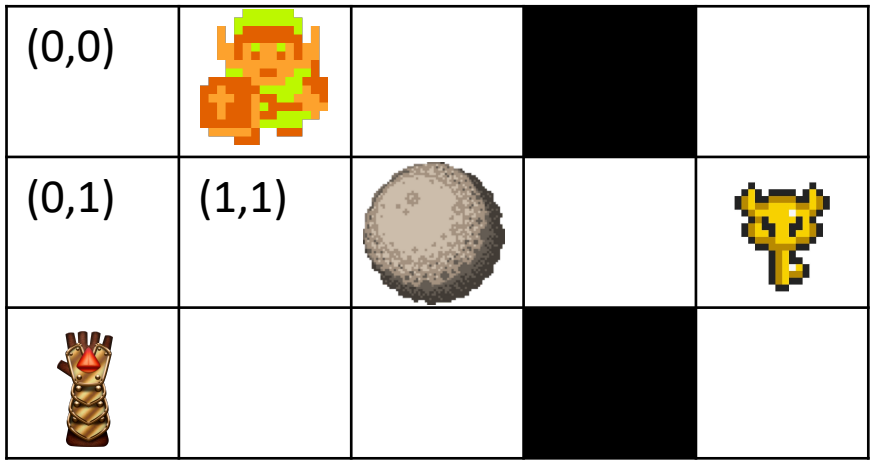
- Anything that requires thought ahead of time and a sequence of actions



Goal: inventory.has(key)

Rules:

- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player

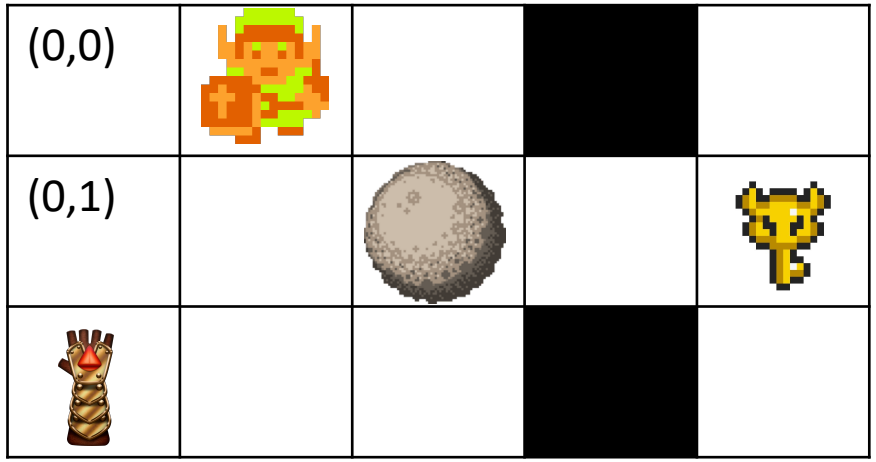


Goal: inventory.has(key)

PQ1: Give me a plan (sequence of rules firing) to reach the goal
https://forms.gle/DEZijGS9734g36wc9
https://tinyurl.com/guz-pq18

Rules:

- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player



My Answer

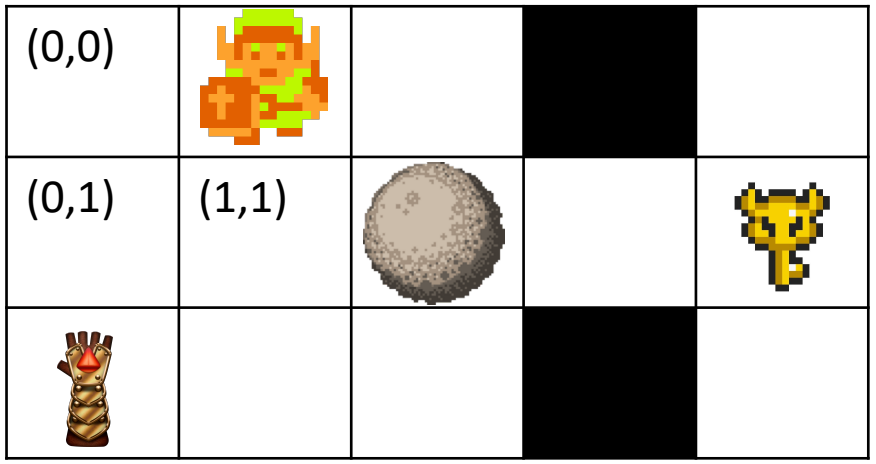
Move (1,1); Move(0,1); Move(0,2); //move gauntlet

Pickup(gauntlet); //get gauntlet

Move(1,2); Move(2,2); //move to boulder

Push(boulder); //push boulder

Move(2,1); Move(3,1); Move(4,1); Pickup(key); //goal

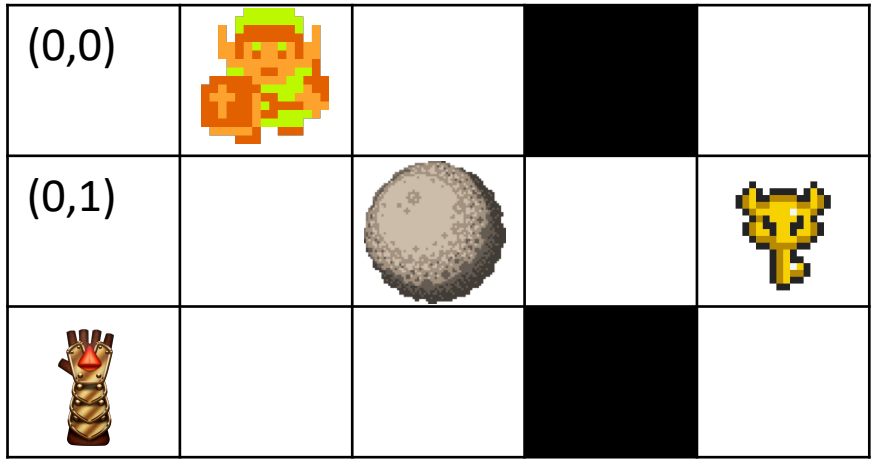


Goal: inventory.has(key)

Give me a plan (sequence of rules firing) to reach the goal

Rules:

- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player



My Answer

Move (1,1); Move(0,1); Move(0,2); //move gauntlet

Pickup(gauntlet); //get gauntlet

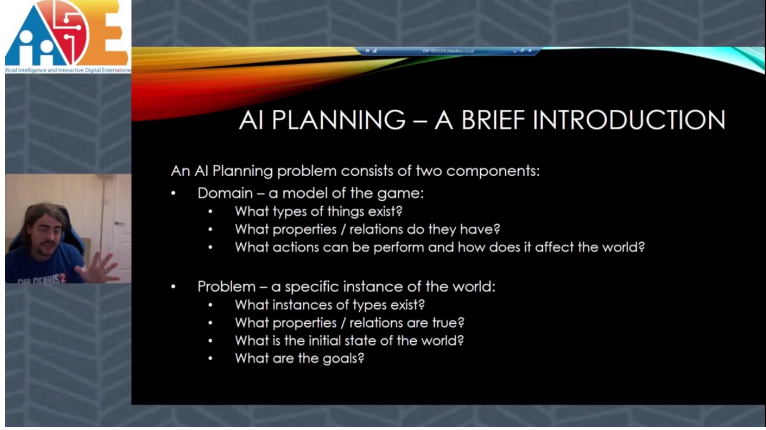
Move(1,2); Move(2,2); //move to boulder

Push(boulder); //push boulder

Move(2,1); Move(3,1); Move(4,1); Pickup(key); //goal

How do we define a planning problem?

- Rules: What can we do in the world?



Bram Ridder, Rebellion Developments (2021)

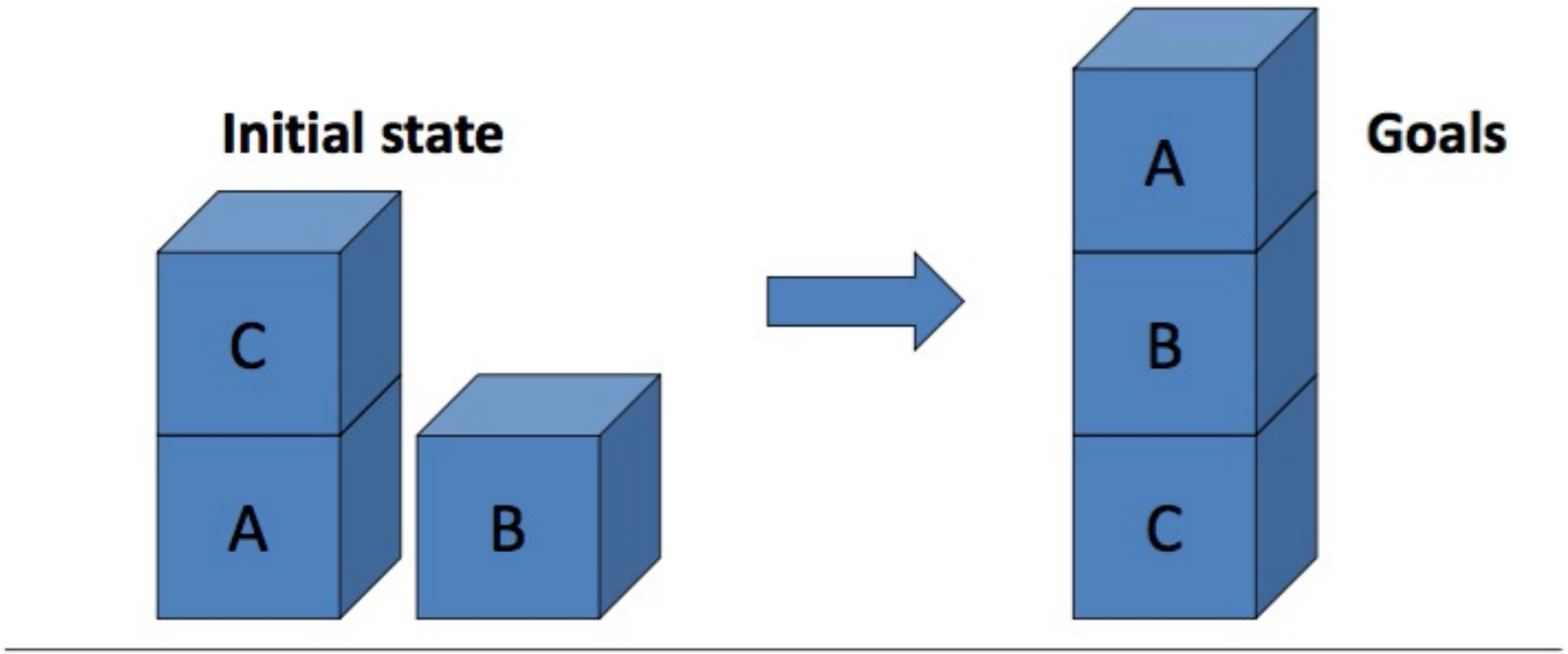
- Environment/Domain/State: How do we represent the world? What can exist within it?

- Goal: What are we trying to achieve in the world?

How Do We Construct a Plan?

- Forward planning/chaining (STRIPS, Fikes & Nilsson 1971)
  - Given some current state
  - Pick out the activate rules
  - Each potential activated rule generates a new potential neighboring state
  - Pick the next current state from our set of neighbors according to some heuristic
  - When hit goal, backtrack to construct sequence of actions
- (This should sound pretty familiar)

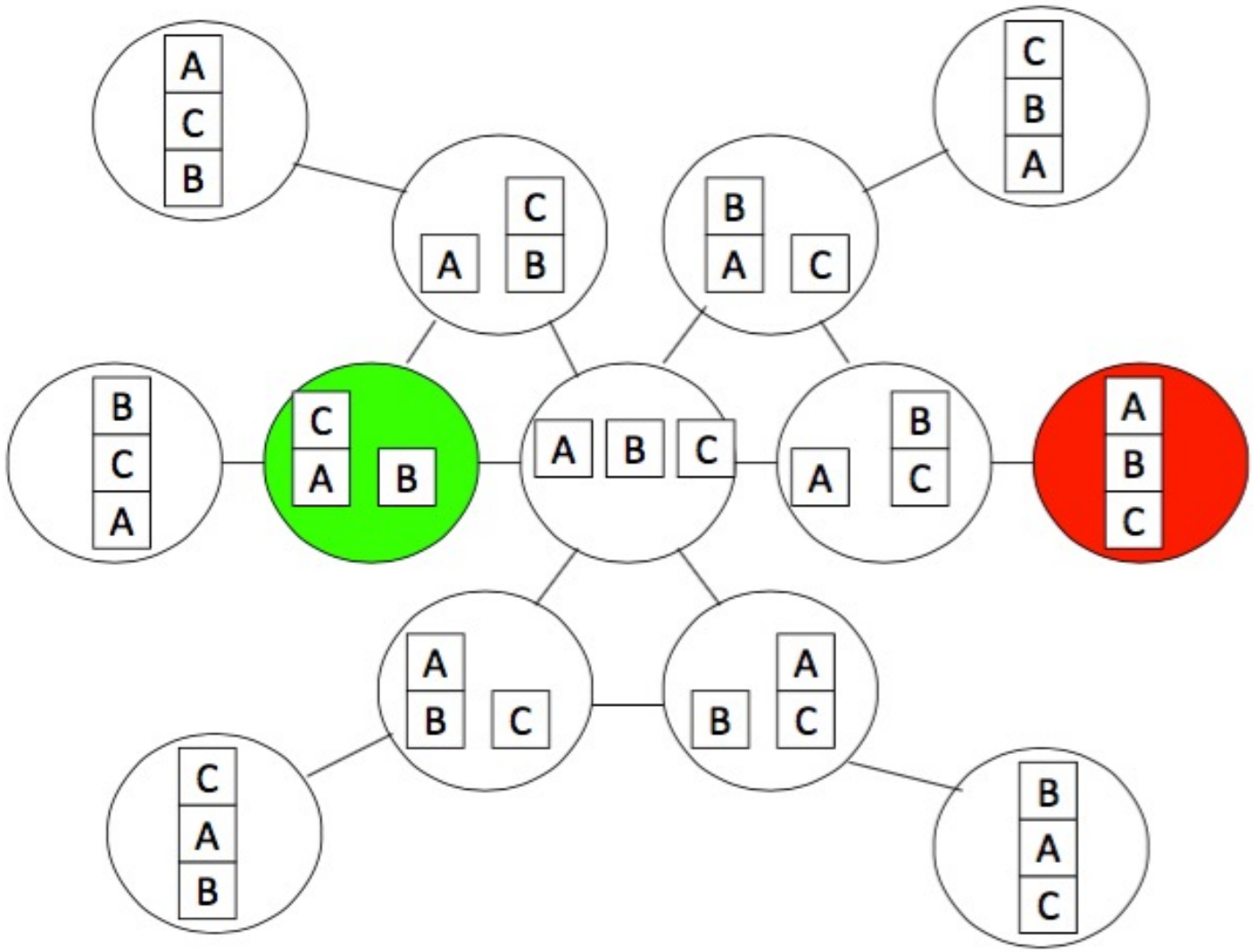
Given some problem…



- **Initial state**: (on A Table) (on C A) (on B Table) (clear B) (clear C)
- **Goals**: (on C Table) (on B C) (on A B) (clear A)

(Ke Xu)

Search Space (World States)



(Michael Moll)

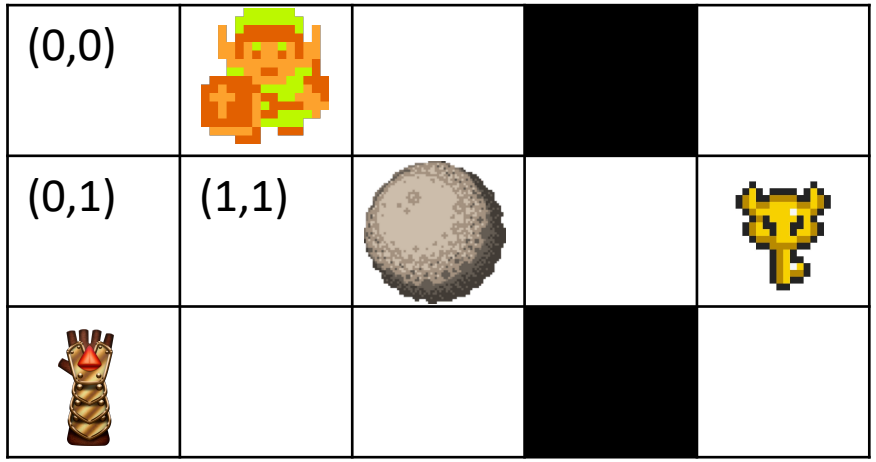
A* Review

add **start** to **openSet**
while **openSet** is not empty:
    current = **openSet**.pop()
    if *current* == **goal**:
        return reconstruct_path(current)
    closedSet.Add(*current*)
    for each *neighbor* of *current*:
        if *neighbor* in **closedSet**:
            continue
        gScore = *current*.gScore + dist(*current*, *neighbor*)
        if *neighbor* not in **openSet**:
            openSet.add(*neighbor*)
        else if *gScore*< **openSet**.get(*neighbor*).gScore
            openSet.replace(**openSet**.get(*neighbor*), *neighbor*)

A* Review

```
add start to openSet
while openSet is not empty:
    current = openSet.pop()
    if current == goal:
        return reconstruct_path(current)
    closedSet.Add(current)
    for each neighbor of current:
        if neighbor in closedSet:
            continue
        gScore = current.gScore + dist(current, neighbor)
        if neighbor not in openSet:
            openSet.add(neighbor)
        else if gScore< openSet.get(neighbor).gScore
            openSet.replace(openSet.get(neighbor), neighbor)
```

Now this represents activated actions firing (neighboring world states), not just movement

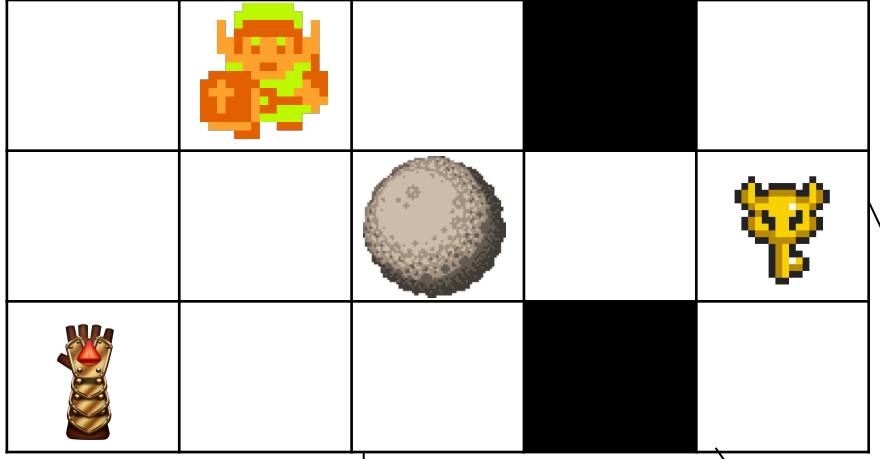


Goal: inventory.has(key)
Heuristic=distance to goal item

Rules:

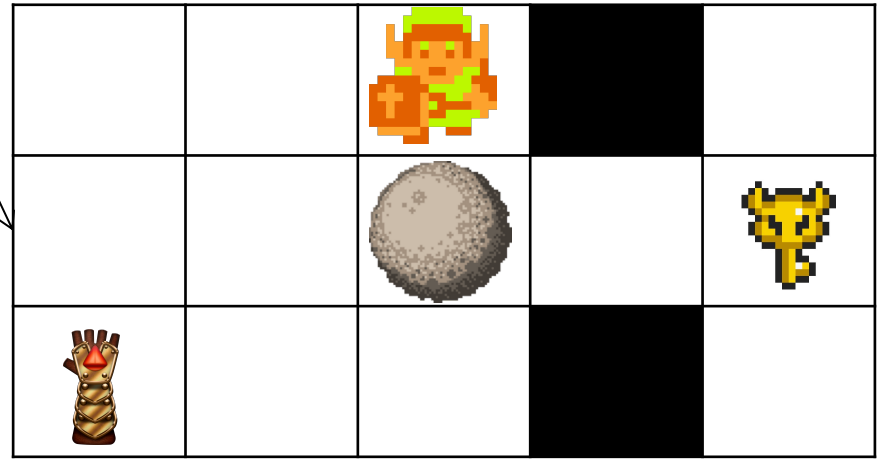
- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player

Goal: inventory.has(key)



Move(1,1)

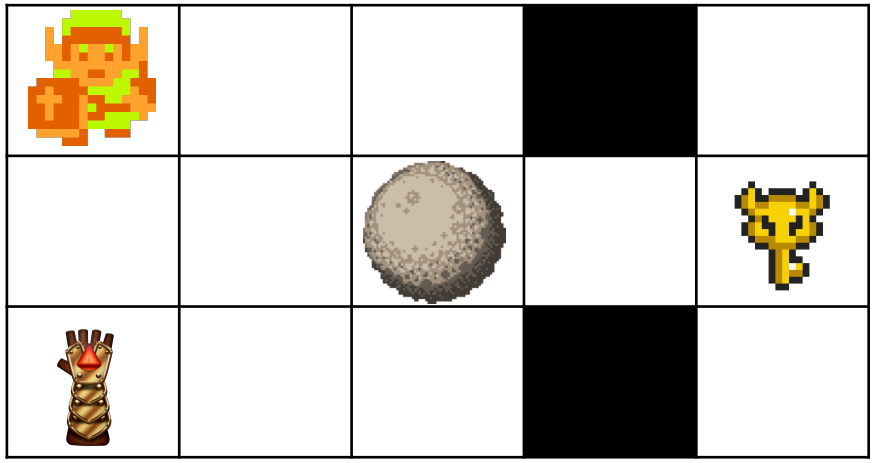
H=(2), G=(1), F=(3)



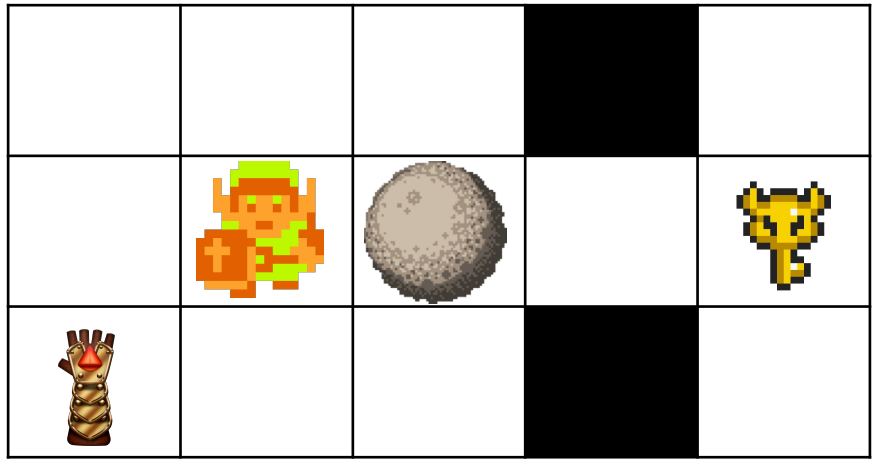
Move(0,0)

Move(2,0)

H=(5), G=(1), F=(6)

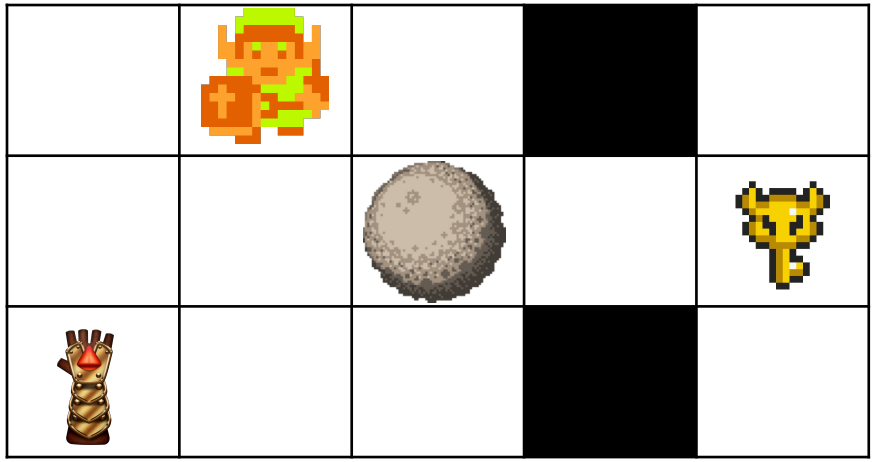


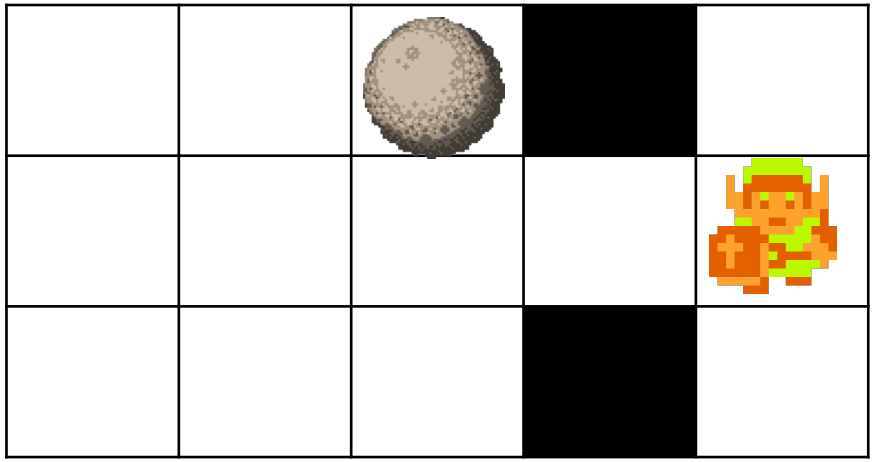
H=(4), G=(1), F=(5)



How do we fix this?

Change our goal representation?

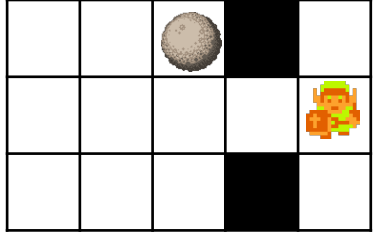


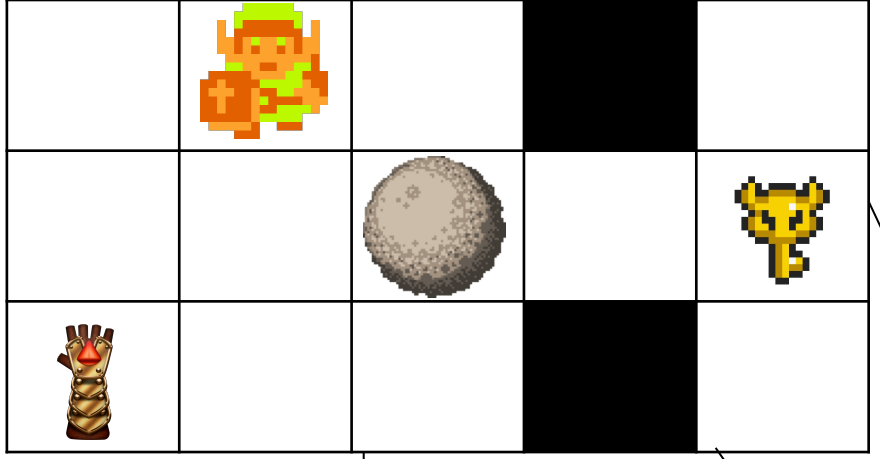


Then our heuristic can be the number of differences between current and goal state

- Example: boulder position, player position, gauntlet picked up, key picked up (Max value of 4 with each of these checks being 0 or 1)

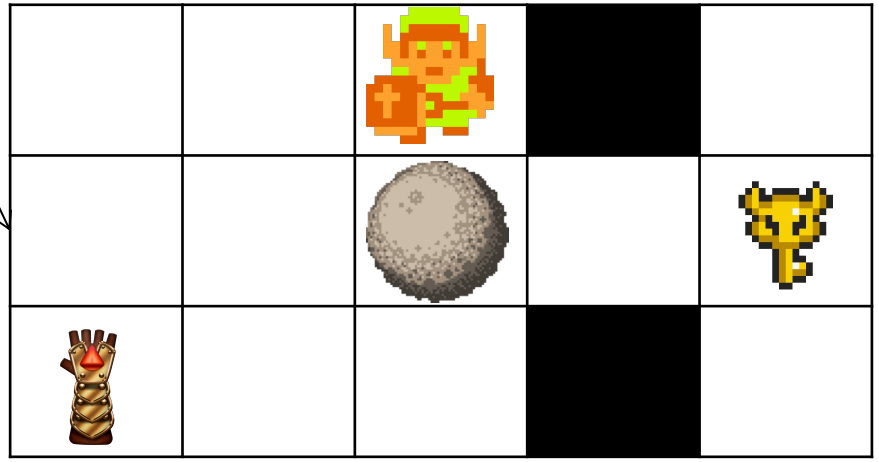
Goal:





Move(1,1)

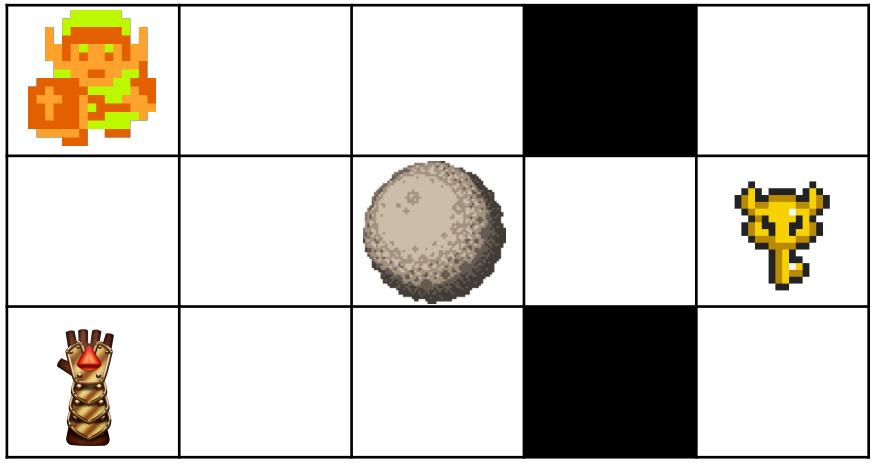
H=(4), G=(1), F=(5)



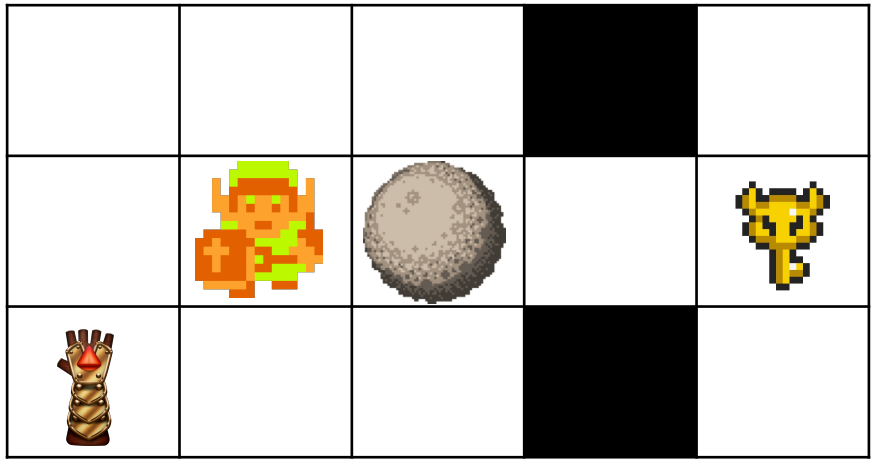
Move(0,0)

Move(2,0)

H=(4), G=(1), F=(5)



H=(4), G=(1), F=(5)

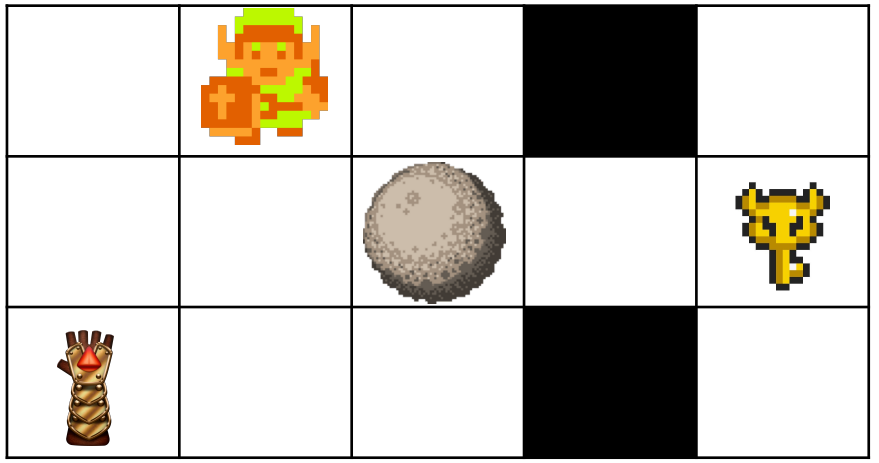


What if we searched from the goal backwards to get a plan?

- Backward planning/chaining
  - Can cut down on search time
  - Allows for more general goal specifications
- From goal search backward to initial state
  - Use rules in reverse
  - Pick the rule whose effects fulfill the conditions of the current state
  - Can use the same sorts of heuristics

Backwards planning

Goal: inventory.has(key)



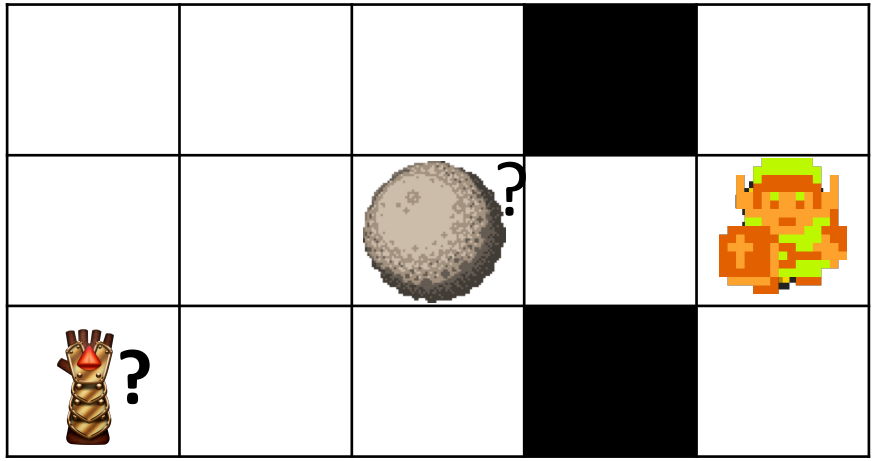
Rules:

- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player

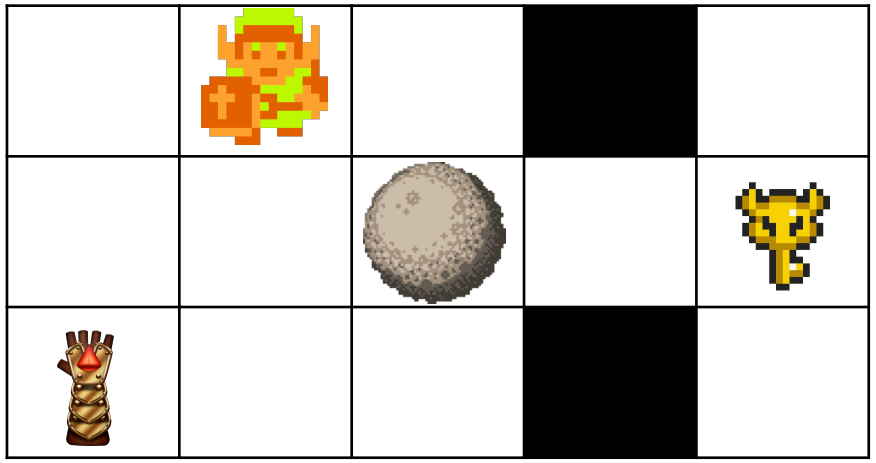
Backwards planning

Goal: inventory.has(key)

Pickup(key)



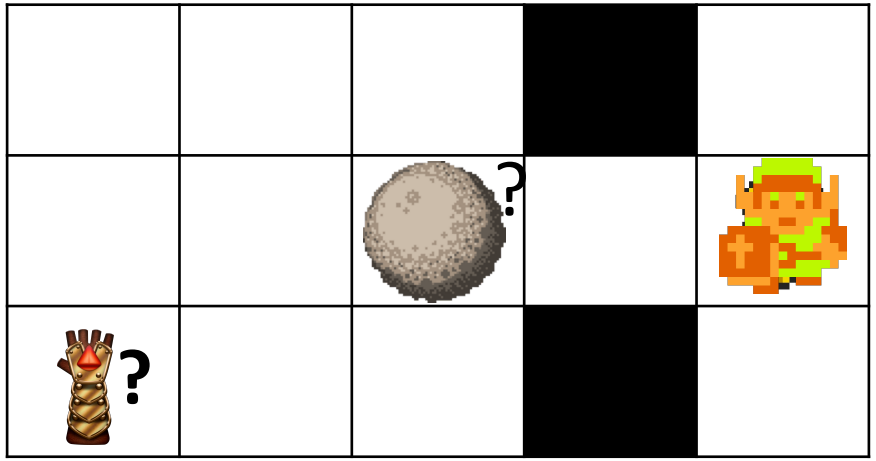
Player.x=key.x, player.y = key.y



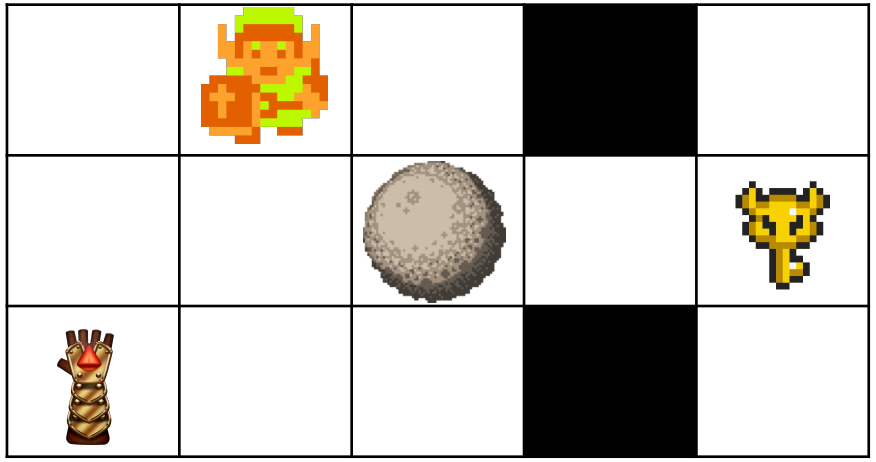
Rules:

- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player

Backwards planning



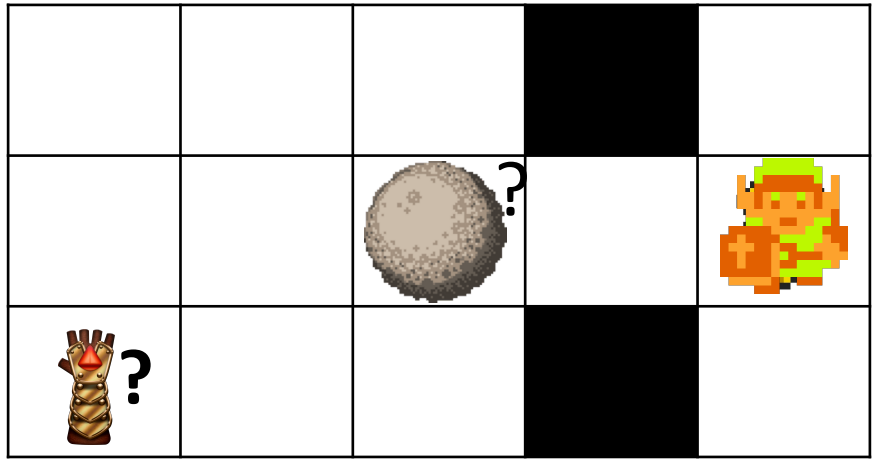
Player.x=key.x, player.y = key.y



Rules:

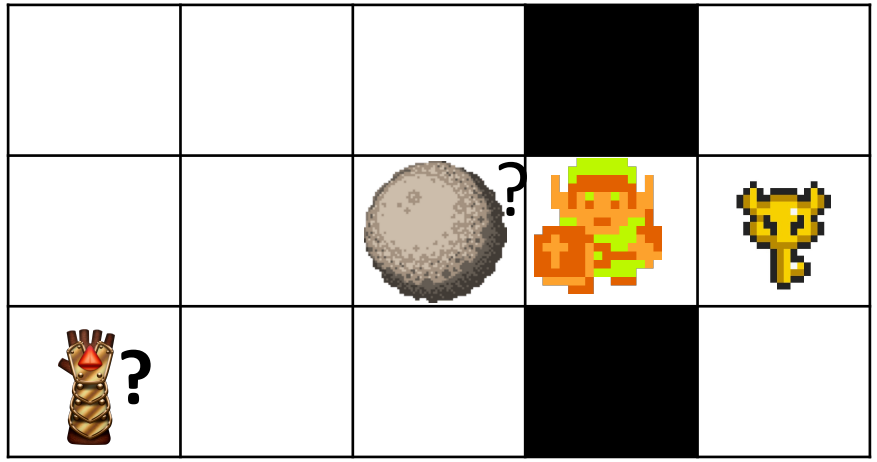
- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player

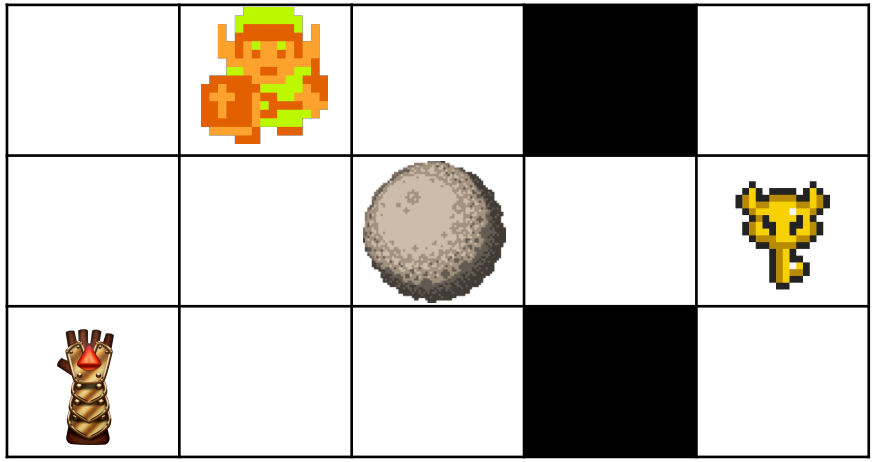
Backwards planning



Player.x=key.x, player.y = key.y

Move(4,1)





Rules:

- **Pickup**(item) *if* item.x==player.x *and* item.y==player.y *then* add item to inventory
- **Move**(x2,y2) *if* open(x2,y2) *&&* neighbors(player.x,player.y,x2,y2) *then* player.x=x2, player.y = y2
- **Push**(boulder) *if* inventory.has(gauntlet) *&&* neighbors(boulder, player) *then* move boulder away from player

Forward Planning

- Pros:
  - Intuitive
  - Natural fit to typical graph search approaches
  - Can handle inconsistent actions
- Cons:
  - Irrelevant actions that can be activated in most states cause high branching factors (lots of children for each node)

Backwards Planning

- Pros:
  - Only considers necessary actions
- Cons:
  - Susceptible to long backtracks when effects negate decisions
  - Works poorly with inconsistent actions or actions with overlapping effects

Planning Problems Generally: Agent Autonomy

Bethesda “Radiant AI” Failure:
https://www.youtube.com/watch?v=pjbx6-KQoRg

How can we ensure a plan matches designer vision?

- Instead of searching through states and constructing a plan -> search through possible plans

- Partial-order planning (POP)
  - Using same structure as before
  - Adding **ordering constraints** to actions (some actions must occur before/after others)

Partial-Order Planning Vocab

- Conflict
  - An action C conflicts with a partial plan A->B if
    - A has an effect p
    - B has a precondition p
    - C has an effect !p
    - C can occur between A and B
- Consistent Plan
  - No cycles
  - No conflicts
- Solution
  - A consistent plan with no open preconditions

POP Algorithm

- Start with initial plan [Start, Finish] where Start can be ordered before Finish
- Pick one **flaw** (open precondition or conflict)
  - Arbitrarily or based on some heuristic
- Generate successor plans for every possible consistent way to resolve flaw
  - Pick plans based on some heuristic
- Stop when solution is reached

Example

- A simple agent that wants two things in life:
  - To kill a possum that has been tormenting it
  - To be rich

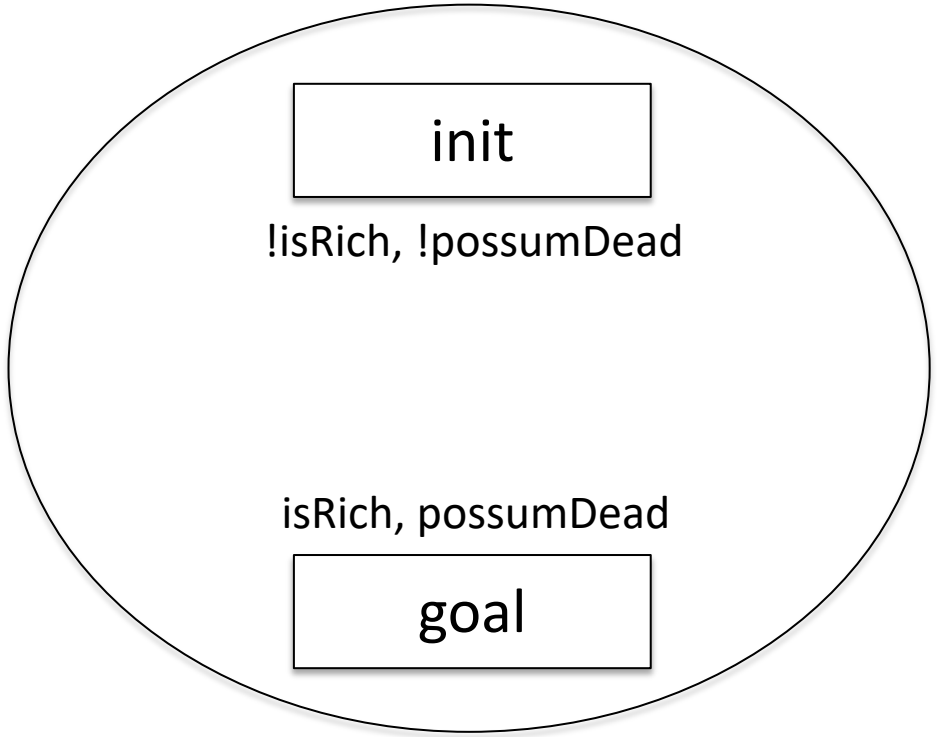
Actions

- **BuyGun** (if _ then hasGun)
- **BuyAmmo** (if _ then hasAmmo)
- **LoadGun** (if hasGun and hasAmmo and !loadedGun then loadedGun and !hasAmmo)
- **RobBank** (if loadedGun and hasGun then isRich)
- **ShootPossum** (if loadedGun and !deadPossum then deadPossum &and!loadedGun)

Ordering Constraints

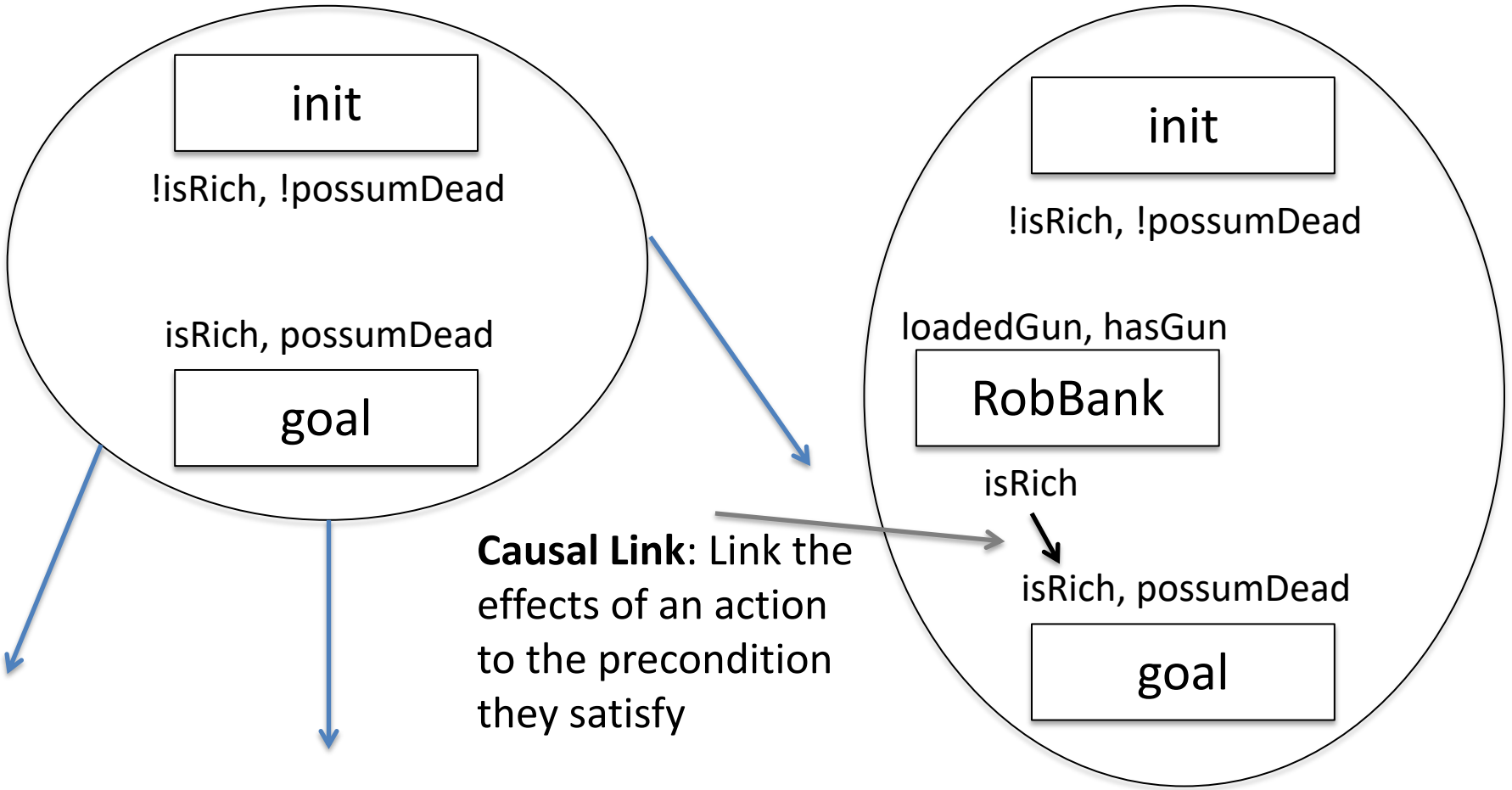
- BuyGun > LoadGun
- BuyAmmo > LoadGun
- LoadGun>ShootPossum
- LoadGun>RobBank

Start with empty plan

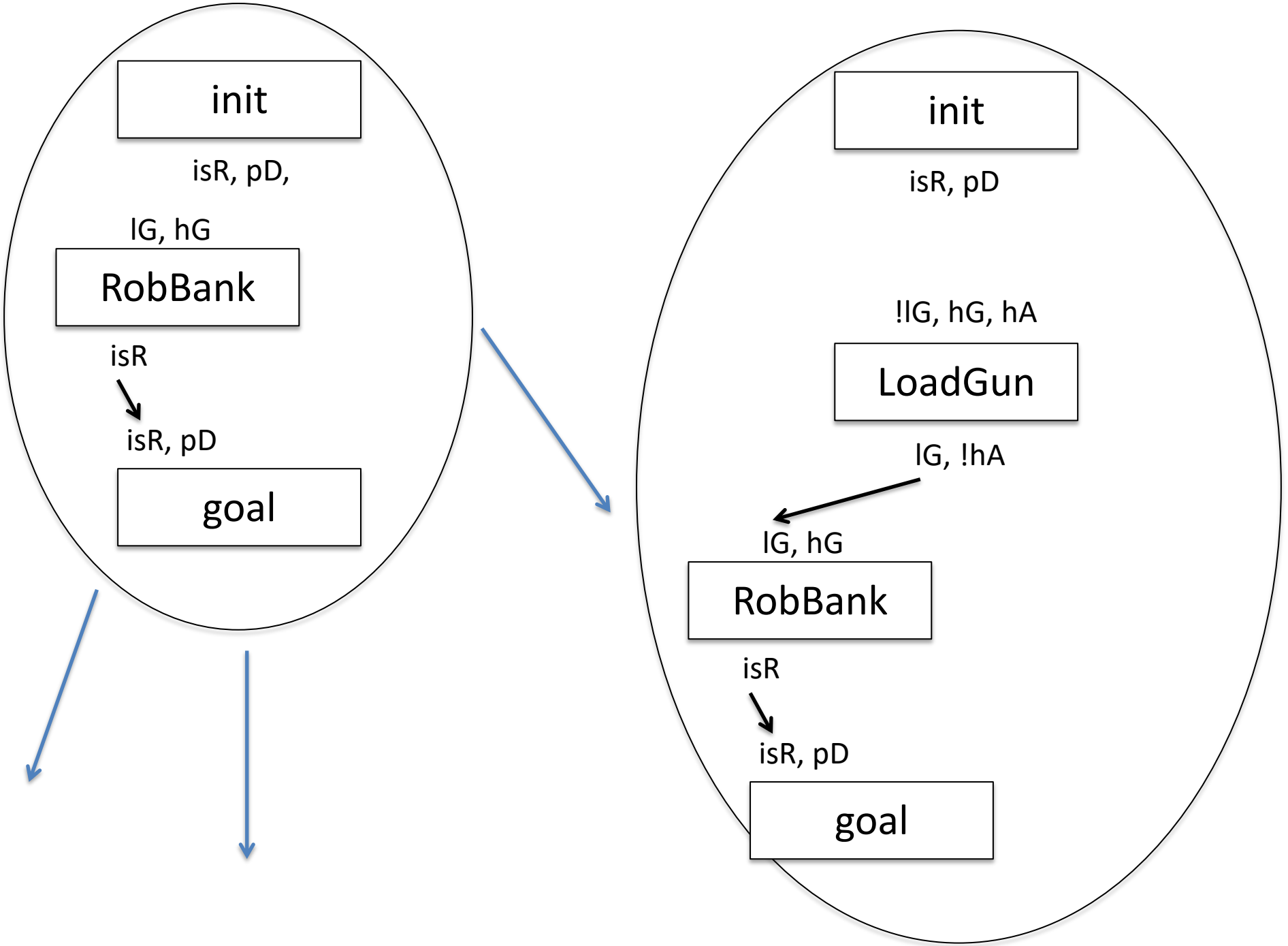


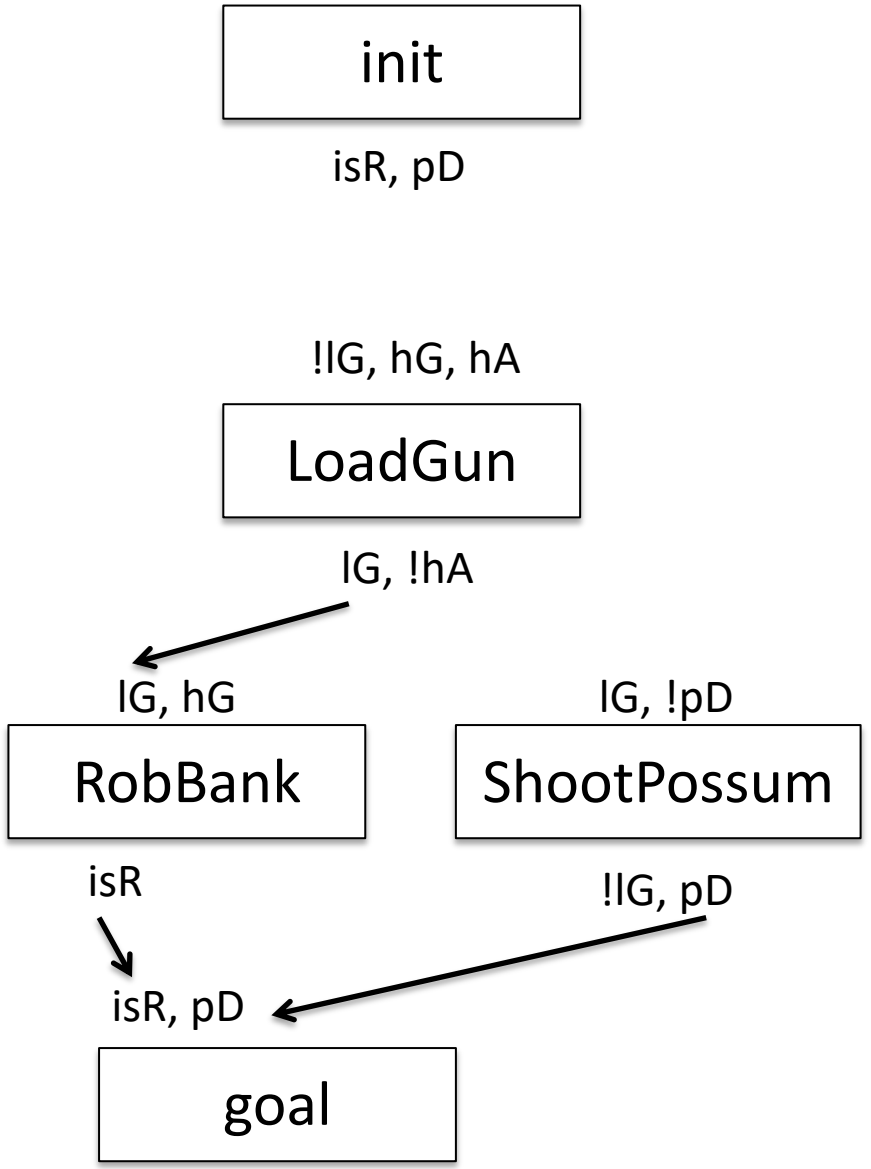
This plan has two **flaws**

Pick a flaw: isRich



If there were other actions to achieve isRich

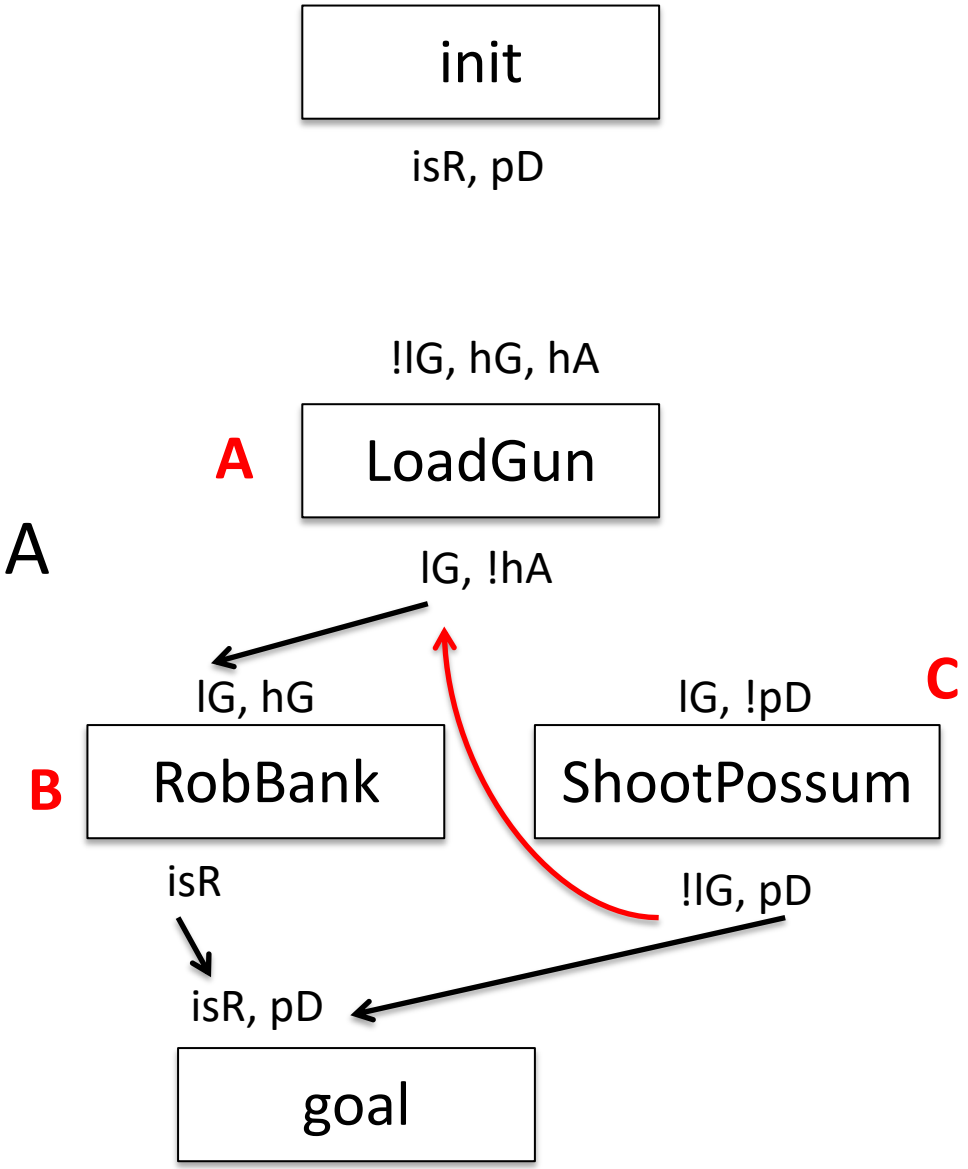

init
isR, pD,
lG, hG
RobBank
isR
isR, pD
goal
init
isR, pD
!lG, hG, hA
LoadGun
lG, !hA
lG, hG
RobBank
isR
isR, pD
goal


init
isR, pD
!lG, hG, hA
LoadGun
lG, !hA
lG, hG
RobBank
lG, !pD
ShootPossum
isR
!lG, pD
isR, pD
goal

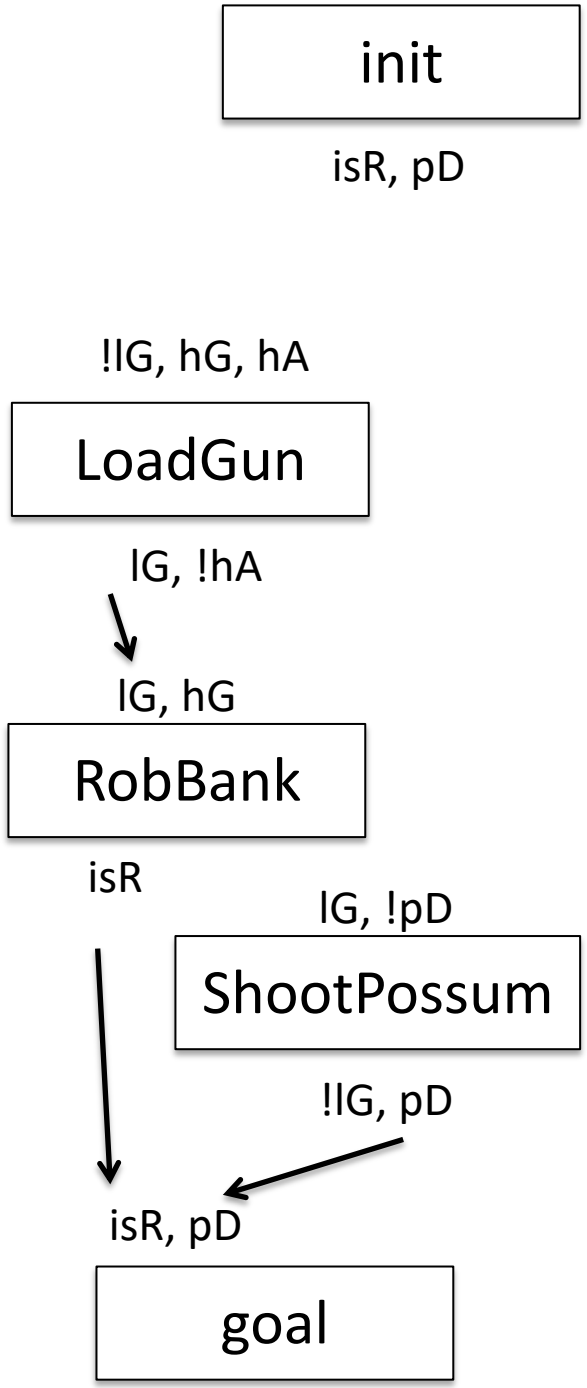
Conflict!

Causal Threat

- Effect of action *could* negate causal link
- **Promote:** Move C before A
- **Demote:** Move C after B



Resolve Conflict



```
agenda = { make_empty_plan(init, goal) }
current = pop(agenda)

WHILE agenda not empty and current has flaws DO:
        flaw = pick_flaw(current)
        IF flaw isa open condition flaw DO:
                FOREACH op in library that has an effect that unifies with o.c. DO:
                        successors += make_new_plan_from_new(...)
                FOREACH op in current that is before and has an effect that uifies with o.c.
DO:
                        successors += make_new_plan_reuse(...)
                IF a condition in init unifies with o.c. DO:
                        successors += make_new_plan_from_init(...)
                IF a condition is negative and CWA applies DO:
                        successors += make_new_plan_from_cwa(...)
        ELSE IF flaw isa causal threat flaw DO:
                successors += make_new_plan_promote(...)
                successors += make_new_plan_demote(...)
        agenda = agenda + successors
        current = pop(agenda)
END WHILE

RETURN current or nil
```

POP Heuristic

- Domain independent heuristic
  - # of flaws
  - Length of plan
- Domain dependent heuristic
  - Designer intent! (e.g. Don’t rob banks)

Benefits of Planning?

- Agents can make decisions/exhibit behaviours that were not pre-authored.
- Reduces authoring burden.
- Increases flexibility.

Problems with Planning?

- Long runtime (replanning from scratch)
- Designers surrender complete control
- Difficult to debug

Applications?

REBELLION

In-development: Playtesting (limited)

Indie Game AI (limited)

Monday:
The One* AAA Example of Planning in Games

Fear (2005)

*Also the Killzone series and Square Enix's "Left Alive" (Front Mission series)
http://www.jp.square-enix.com/tech/library/pdf/GDC2021_ART_HTN.pdf

Quiz Time

https://forms.gle/nYoXnWt13Qw5H8ec8

https://tinyurl.com/guz-quiz3

1. Which of the following requires the least runtime computation?
   A.) Planning
   B.) FSMs
   C.) B trees
   D.) Rule Systems
2. Imagine a planning approach with the following rules:
   (1) if A then B=false and C+=1,
   (2) If B then A=false and C+=1,
   (3) if C is odd then A=true and B=true.
   Suppose we start in a state where C=0, A=true, and B=true, with a goal of C=6. Give me the best plan to get to the goal given a heuristic that prefers **smaller** plans with the most **variety** (more rules used).
3. What planning approach would you use to generate this plan?
4. What topic(s) would you be interested in for the Future of Game AI section/quiz?

Answers

1. (B) FSMs
2. The important thing to note is that the shortest path would be to repeat (1) or (2) 6 times. But the best path given the heuristic (that values variety) is instead to swap back and forth from (1) and (2) after using (3).
3. Forward or POP.
4. Any answer