



BANDAI NAMCO Studios

プレイヤーに反応するだけのAIはもう古い！ ゲームAIへのプランニング技術の導入

株式会社バンダイナムコスタジオ
長谷 洋平

アジェンダ

- プランニングの概要
- STRIPS
 - アルゴリズム
 - 使用例
- HTNプランニング
 - アルゴリズム
 - 使用例
- ゲームでの利用
- まとめ

アジェンダ

- プランニングの概要
- STRIPS
 - アルゴリズム
 - 使用例
- HTNプランニング
 - アルゴリズム
 - 使用例
- ゲームでの利用
- まとめ

プランニングの歴史

- ダートマス会議、1956年
- 第1次AIブーム、1956年～1960年代
- STRIPS、1971年
- HTNプランニング、1977年

ちなみに・・・

- コンピュータースペース(世界初の業務用ゲーム)、1971年
- パックマン、1980年
- ファミリーコンピュータ、1983年

プランニングが使われたタイトル(一部)

STRIPS

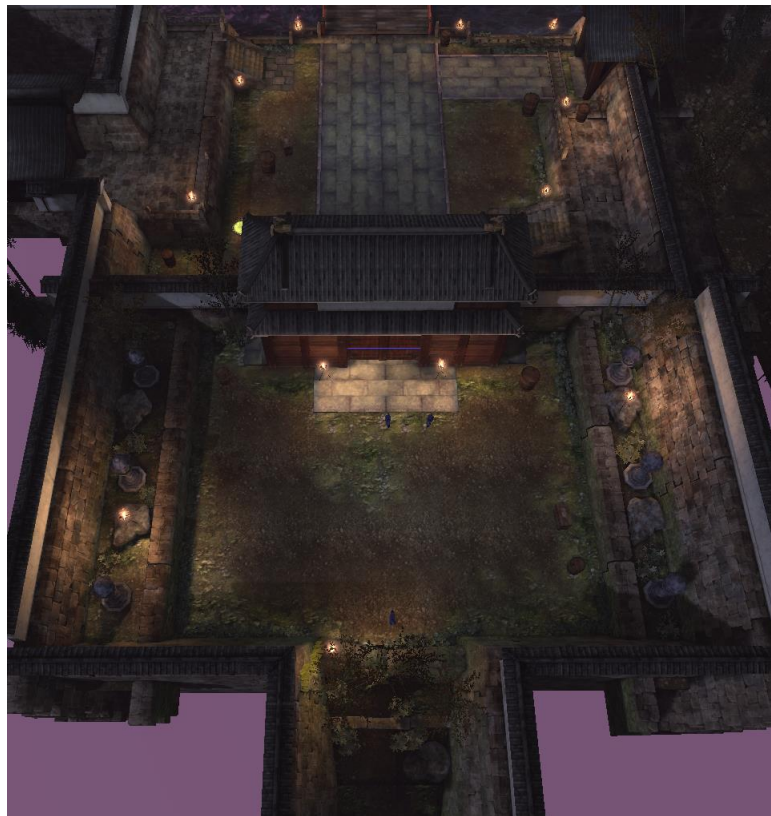
- F.E.A.R.
- Deus Ex: Human Revolution
- Middle-Earth: Shadow of Mordor
- Tomb Raider

HTNプランニング

- KILLZONE 2 ~
- Transformers: Fall of Cybertron
- Max Payne 3
- Dying Light

なぜプランニングが使われているのか

従来ของเกม

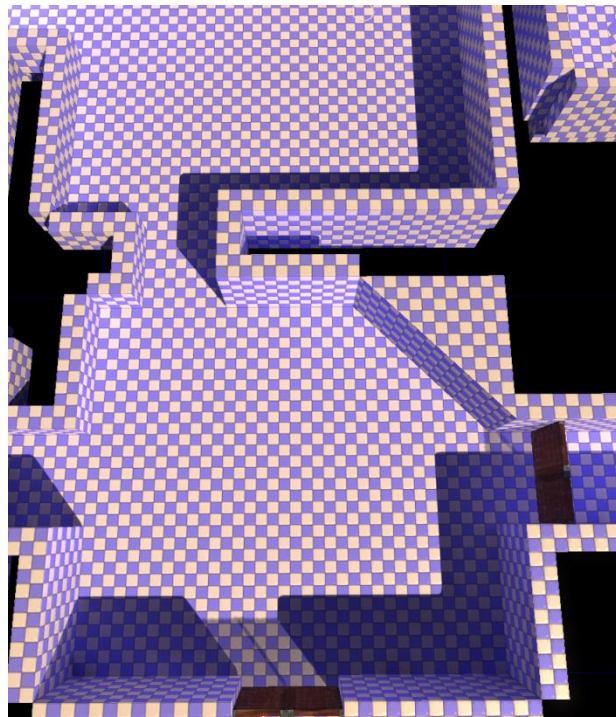


なぜプランニングが使われているのか

従来のゲーム

(ゲームプレイに関連する要素のみ)

- 部屋が通路でつながっている
- 部屋の形状はシンプルで
特殊なギミックや障害物も少ない
- 敵キャラクターはプレイヤーと
1対1で対峙してれば OK



なぜプランニングが使われているのか

最近のゲーム

- 複雑で広大なマップ
- 高低差やギミックが用意されている
- キャラクター同士で連携する必要がある
- グラフィックがリッチになって
シンプルなAIでは不自然さが目立ってきた

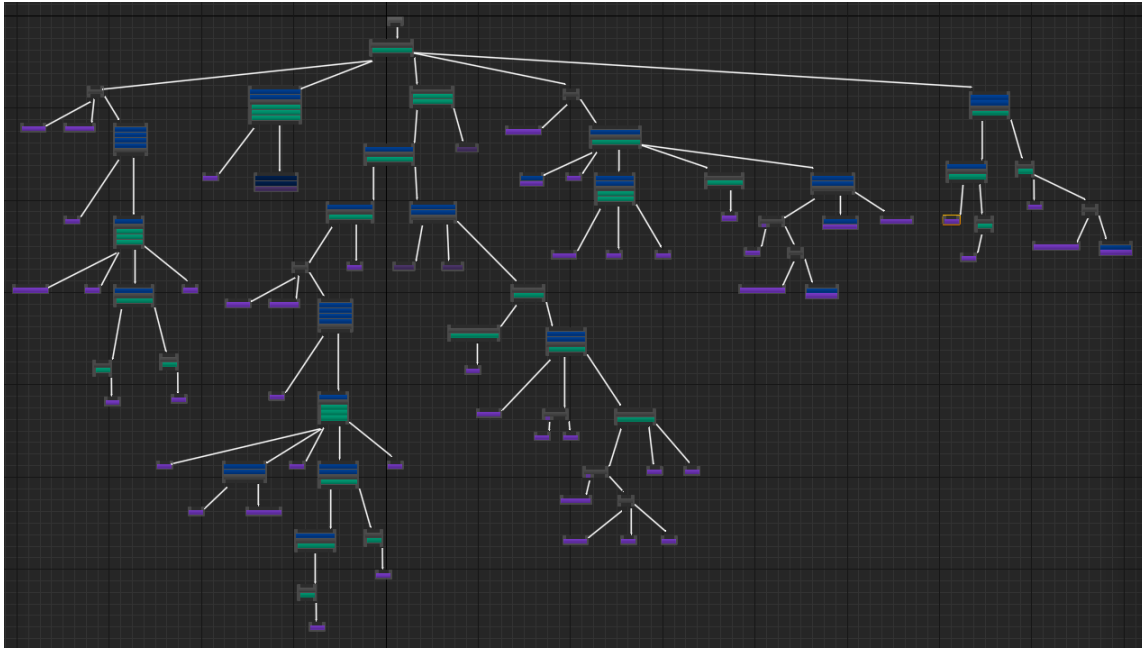


多くの条件を考慮して行動を決定しないといけない



既存手法の欠点

- 複雑で巨大な思考ルーチンを構築しないといけない



(イメージ)

既存手法の欠点

- 膨大な数のアセットが出来上がる



(イメージ)

既存手法の欠点

- 複雑でリッチな環境で動く AI をこれまでと同じように作ろうとすると多くの課題にぶち当たる
 - 全体像が把握できないほど巨大な思考ルーチンになり、試行錯誤をしにくい
 - いろいろなものが複雑に絡み合っていて
仕様のには少しの変更でも多くの箇所に変更を加えないといけない
 - 少しの違いであっても別アセットとして作成する必要が出てきて
管理コストが増える
- 開発コストがかかる割にクオリティは低い AI を量産する原因の一つに

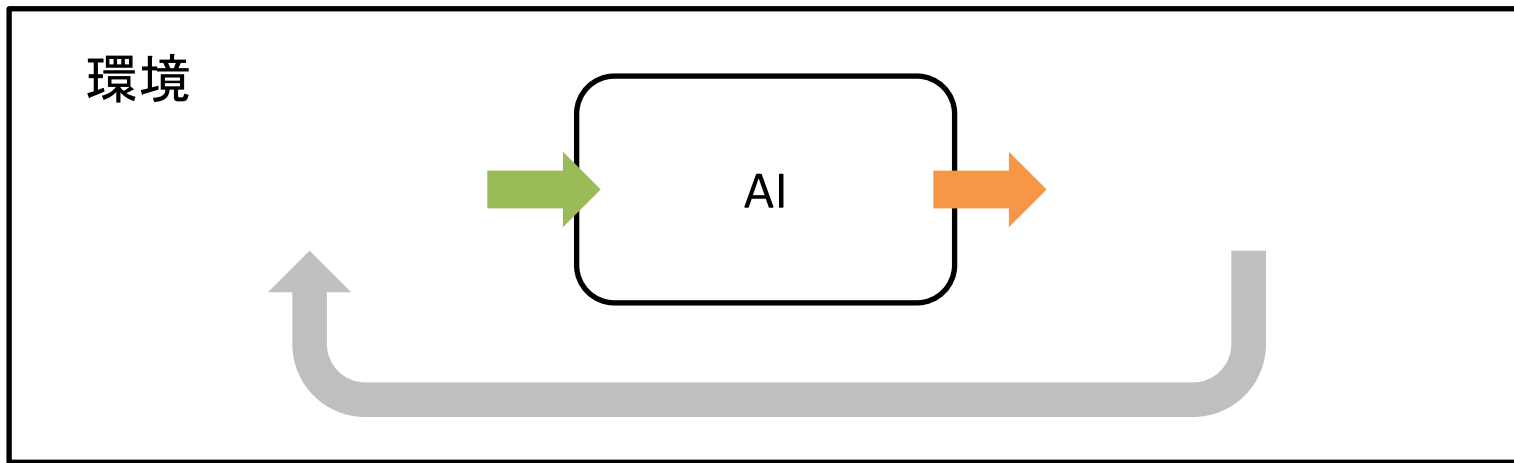


プランニングを使うことで、合理的な AI をよりシンプルに構築できる

AIの仕組み

- 環境から何らかの情報を受け取って何らかの情報を出力する
- 自分の行動やほかの要因で変化した環境の情報を再度入力として受け取って出力を出す

⋮



ボードゲーム

入力

- 盤面の状態
- 進行度(序盤、終盤)
- 持ち駒

出力

- 次の手



入力

- 味方の位置、状態
- 敵の位置、状態
- レベルの構造

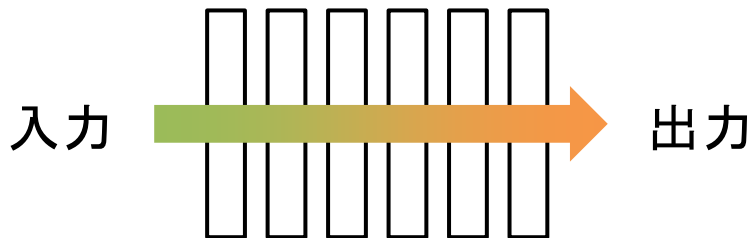
出力

- アニメーション
- ボイス
- 武器の発射

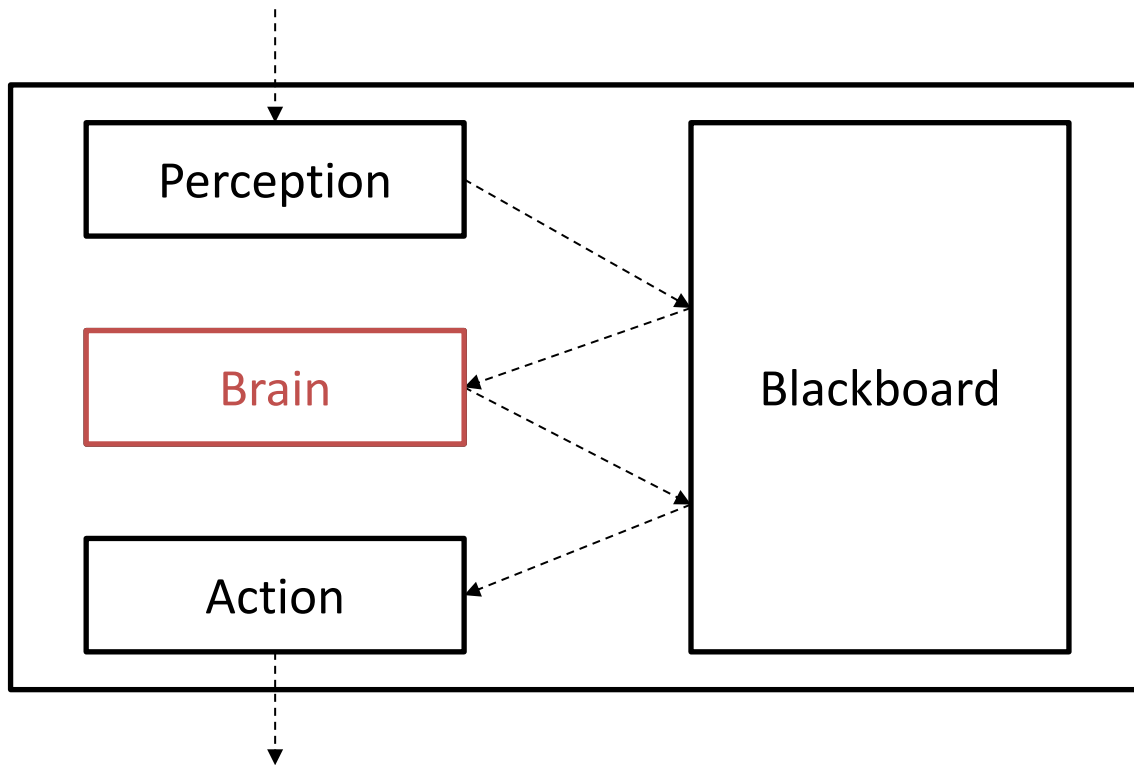


AIの仕組み

- 与えられる入力と期待される出力はゲームや AI ごとに変わってくる
- 多くのモジュールを介して入力から出力を求める
 - 機能単位でモジュールに分割することで追加・変更が容易になる
 - 異なる AI 間での再利用が容易になる
- 目的とする AI に合わせていろいろなモジュールを組み合わせてデザインする必要がある



AIの仕組み



意思決定に関する技術

- FSM
- BehaviorTree
- UtilitySystem

→ 現在の状態をもとに行動を決定する

- プランニング (STRIPS、HTNプランニング)

→ その行動をとった結果、状態がどう変化するか (未来) を考慮したうえで行動を決定する

FPSのようなゲームで・・・

- 敵を倒すのが目的
- 銃で敵を攻撃できる
- 敵を倒すには何発か弾を当てる必要がある
- 銃弾はフィールド上に落ちている

所持弾数が少ない状態で敵を見つけた

1. とりあえず持ち弾を全て使って攻撃してから弾を補充して再度攻撃する
 2. 敵を倒すのに十分な量の弾を補充してから攻撃しに行く
- ・・・ 多くの場合、後者のほうが自然で合理的

従来のAI

思考ルーチン

- もし弾がなければ
 - 補給する
- もし敵がいれば
 - 攻撃する

現在の状態

- 所持弾数: 1、敵を倒すのに必要な弾数: 2

AIの振る舞い : 敵を攻撃する → 弾を補給する → 敵を攻撃する

思考ルーチン

- もし敵を倒すのに十分な弾がない and 所持弾数がMAXじゃない
 - 補給する
- もし敵がいれば
 - 攻撃する

現在の状態

- 所持弾数: 1、敵を倒すに必要な弾数: 2

AIの振る舞い : 弾を補給する → 敵を攻撃する

思考ルーチン

- もし敵を倒すのに十分な弾がない and 所持弾数がMAXじゃない
 - 補給する
 - もし敵がいれば
 - 攻撃する
- 弾1発で与えられるダメージ量を計算する
 - 敵の体力を0にするために必要な弾数を計算する
 - 求まった弾数と所持弾数を比較する

現在の状態

- 所持弾数: 1、敵を倒すのに必要な弾数: 2

自然な振る舞いにするには
複雑な条件判定が必要になる

AIの振る舞い : 弾を補給する → 敵を攻撃する

ほかの要素も加味すると...

- ほかの攻撃手段がある
- 味方を回復できる
- etc...



思考ルーチン

- もし味方の体力が少なければ
 - 回復薬を持っている
 - 味方を回復する
 - 回復薬を持っていない
 - 回復薬を取りに行く
- もし敵がいれば
 - 弾が少ない
 - 弾を補給する
 - 倒せる
 - 攻撃する
 - ⋮

どんどん複雑になって
実装、管理コストの増大や
不具合が頻出する原因に！

プランニングを用いたAI

アクション

- 攻撃: 所持弾数が減る
- 補給: 所持弾数が増える

現在の状態

- 所持弾数: 1、敵を倒すのに必要な弾数: 2

行動の候補

- 敵を攻撃する → 弾を補給する → 敵を攻撃する
- 弾を補給する → 敵を攻撃する

← 選択

プランニングを用いたAI

アクション

- 攻撃: 所持弾数が減る
 - 補給: 所持弾数が増える
 - 回復: 回復薬を消費する
 - 薬の取得: 回復薬を獲得する
- } AIが行える行動が増えたときは
アクションを追加するだけ
- 行動した時に状態がどう変化するかを定義しておけば
状態の変化を考慮した振る舞いを組み立ててくれるので
事前の条件判定であらゆる可能性を考えなくていい

プランニング概要



BANDAI NAMCO Studios

- 目標を達成するために行う必要のある行動の順序(プラン)を自動で組み立てるためのアルゴリズム
- 現在の状態と行動をとったときの状態の変化から、どの行動をどういう順番で行えば目標を達成できるかを見つけ出す
- ロボットの行動計画などで使われている
- ゲームAIでよく用いられるのは、STRIPSとHTNプランニングの2つ

プランニング概要

- ドメイン、ステート、ゴールの3つをプランナに渡すことで最適なプランを計算する
 - ドメイン: 問題領域で取りうるアクションの集合
 - 作成するAIに合わせて事前に定義しておくもの
 - ステート: 現在の環境や自分の状態
 - ゴール: 達成すべき目標
- } → その時の状況に合わせてリアルタイムに設定される



アジェンダ



BANDAI NAMCO Studios

- プランニングの概要
- STRIPS
 - アルゴリズム
 - 使用例
- HTNプランニング
 - アルゴリズム
 - 使用例
- ゲームでの利用
- まとめ

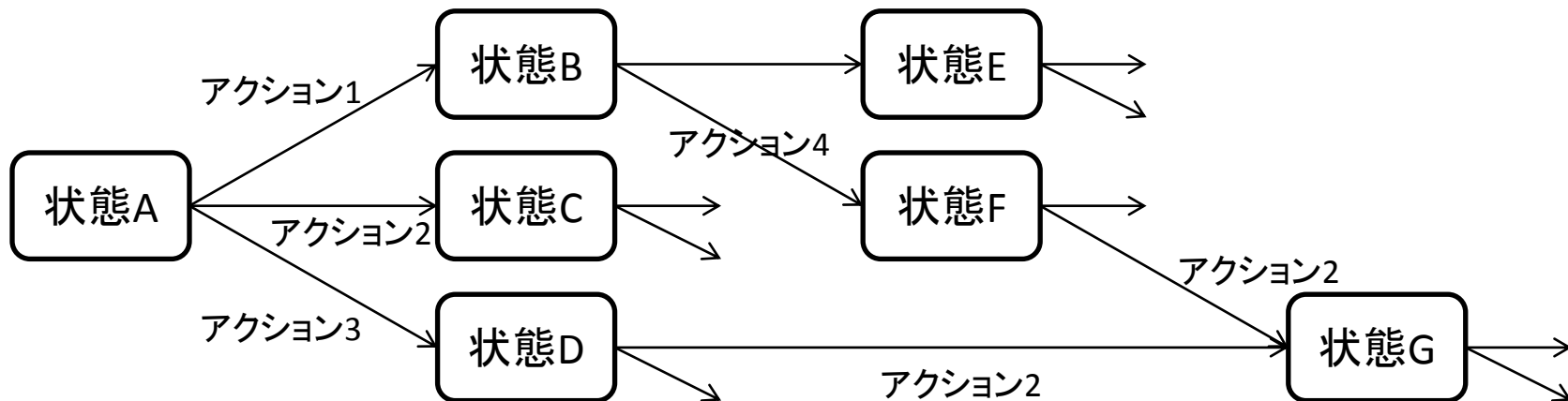
- 現在の状態から目標の状態に達するまでの行動の順序を探索するアルゴリズム
- ゲーム業界では GOAP (Goal Oriented Action Planning) とも呼ばれる

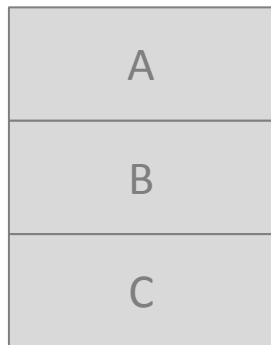
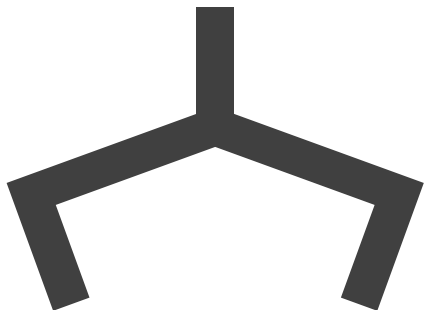
STRIPSに与える情報

- **アクション**: 事前条件と事後条件のセット
 - **事前条件**: アクションを行うのに満たしておく必要のある条件
 - **事後条件**: アクションを行った後に満たされる条件 (状態の変化)
- **ドメイン**: アクションの集合
- **ステート**: 現在の状態
- **ゴール**: 目標の状態

STRIPS

- 状態をノードとすると状態Aから状態Bに変化するには何かしらのアクションをとっているはずなのでアクションをノード間の接続と考えることができる
- 現在の状態から目標の状態までの行動の列は通常のグラフ探索と同様の方法で求めることができる



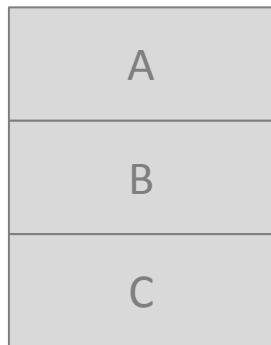
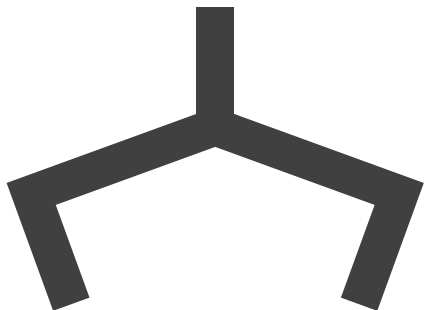


ルール

- テーブルの上に複数のブロックが置かれている
- ブロックの上にはブロックを1つ積むことができる
- ロボットアームはブロックを1つだけ持ち上げることができる

ブロックをあらゆる状態に積みなおすことができる
プログラムはどう作ればいいでしょうか？

STRIPSであればアクションを4つ定義するだけでいい



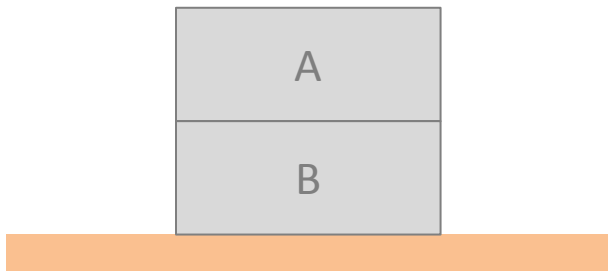
アクション

- pickup(a) :
テーブルの上にある a を持つ
- putdown(a) :
持っている a をテーブルの上に置く
- stack(a, b) :
持っている a を b の上に置く
- unstack(a, b) :
b の上にある a を持つ

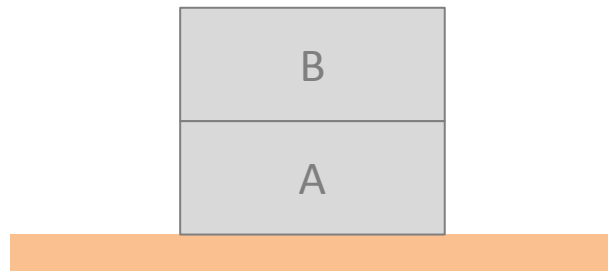
Blocks World

ブロックの積まれた順番を変える場合

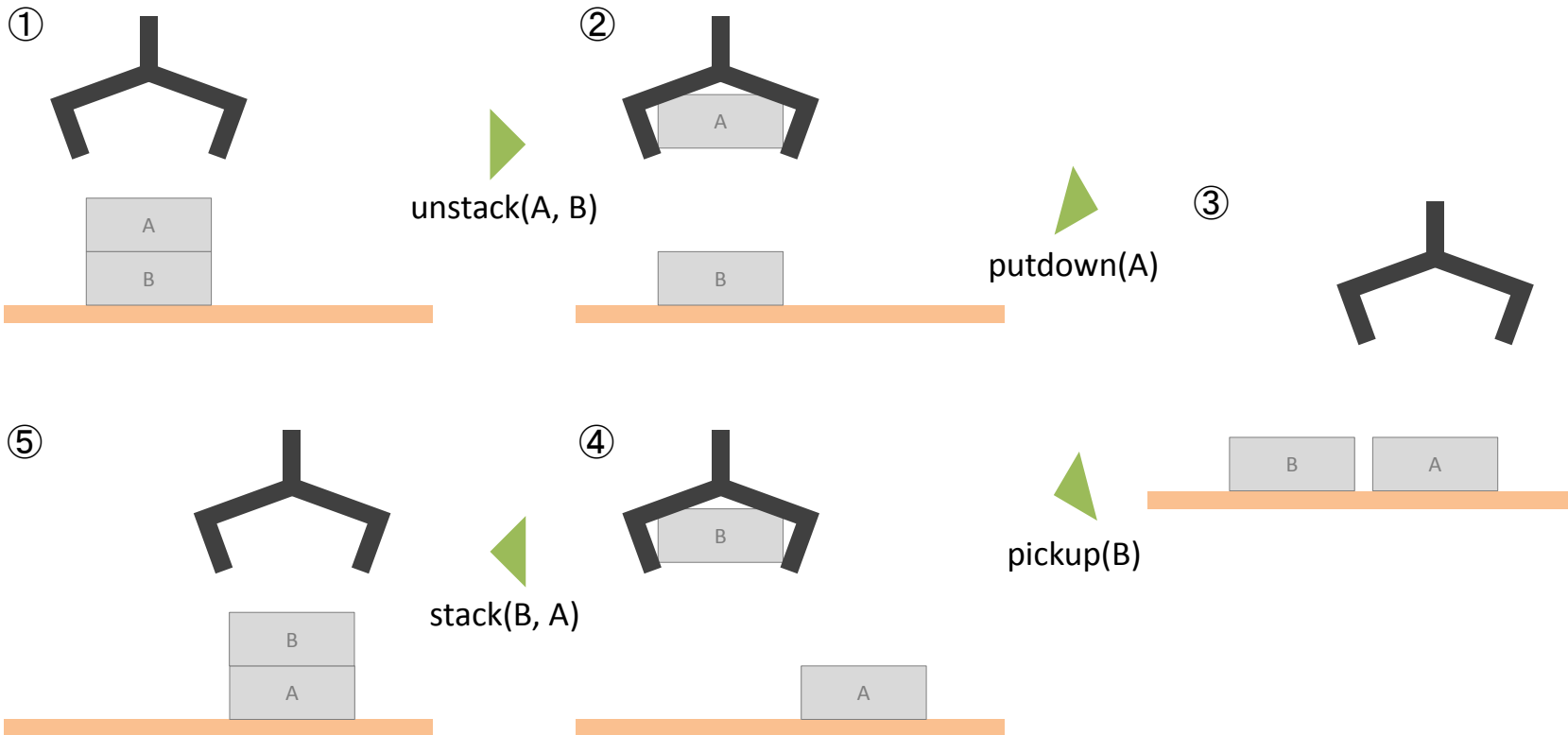
初期状態



目標状態



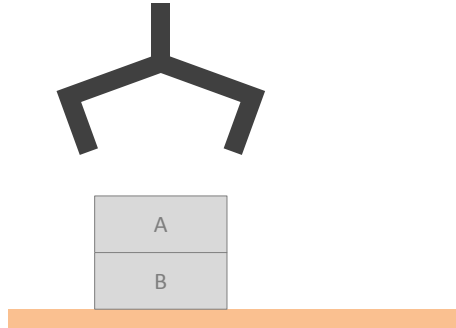
Blocks World



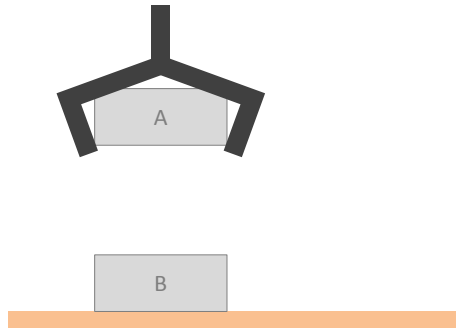
状態

- $\text{OnTable}(a)$: a はテーブルの上にある
- $\text{On}(a, b)$: a は b の上にある
- $\text{Clear}(a)$: a の上には何も積まれていない
- $\text{Holding}(a)$: ロボットアームは a を持っている
- HandEmpty : ロボットアームは何も持っていない

Blocks World



- $\text{On}(A, B)$
- $\text{OnTable}(B)$
- $\text{Clear}(A)$
- HandEmpty



- $\text{OnTable}(B)$
- $\text{Clear}(B)$
- $\text{Holding}(A)$

Blocks World

- pickup(a) : テーブルの上にある a を持つ
 - 事前条件 : OnTable(a), Clear(a), HandEmpty
 - 事後条件 : + Holding(a), – OnTable(a), Clear(a), HandEmpty
- putdown(a) : 持っている a をテーブルの上に置く
 - 事前条件 : Holding(a)
 - 事後条件 : + OnTable(a), Clear(a), HandEmpty, – Holding(a)
- stack(a, b) : 持っている a を b の上に置く
 - 事前条件 : Holding(a), Clear(b)
 - 事後条件 : + On(a, b), Clear(a), HandEmpty, – Holding(a), Clear(b)
- unstack(a, b) : b の上にある a を持つ
 - 事前条件 : On(a, b), Clear(a), HandEmpty
 - 事後条件 : + Holding(a), Clear(b), – On(a, b), Clear(a), HandEmpty

STRIPSのアルゴリズム

- 定義された状態とアクションを使って探索していく
 - 前向き探索 : 初期状態から目標状態へ向かって探索する
 - 現在の状態で実行可能な(事前条件をクリアしている)アクションを選んでいって、目標の状態に到達したら終了
 - 後ろ向き探索 : 目標状態から初期状態へ向かって探索する
 - 目標の状態を満たすことができる(事後条件をクリアしている)アクションを選んでいって、初期状態に到達したら終了
- 一般的に後ろ向き探索を用いることが多い
 - 前向き探索では探索空間が大きくなりがちで後ろ向き探索のほうが効率がいい
 - 複雑な目標に対しても柔軟な動作が可能

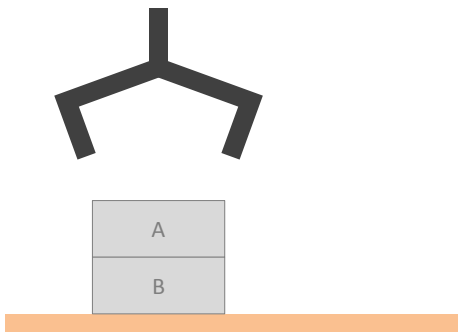
STRIPSのアルゴリズム

後ろ向き探索の場合

- 目標状態をリストに追加する
- リストがなくなるまでループ
 - リストから状態を取得
 - 取得した状態を初期状態が満たしている(=初期状態に達した)
 - プランが見つかったので終了
 - その他
 - 取得した状態を満たすことのできるアクションを選択
 - アクションの事前条件、事後条件から新たな状態を作成してリストに追加
- プランが見つからなかった

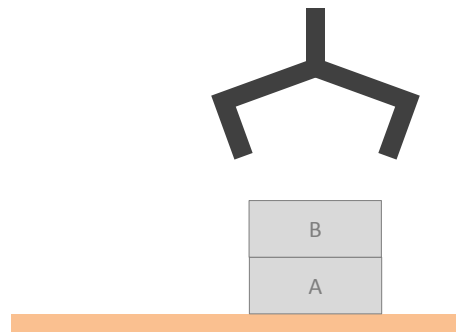
動作例

初期状態



- $\text{On}(A, B)$
- $\text{OnTable}(B)$
- $\text{Clear}(A)$
- HandEmpty

目標状態



- $\text{On}(B, A)$

動作例

1. $\text{On}(B, A)$ から探索を始める
2. $\text{On}(B, A)$ を満たすことができるのは $\text{stack}(B, A)$ のみ
3. $\text{stack}(B, A)$ で満たされる条件は削除して、事前条件を追加する
 - $\text{Holding}(B), \text{Clear}(A)$
4. まだ初期状態には達していないのでアクションを選択する
 - $\text{pickup}(B), \text{unstack}(B, A)$
5. それぞれのアクションを適用する前の状態を作成する
 - $\text{pickup}(B) : \text{OnTable}(B), \text{Clear}(B), \text{HandEmpty}, \text{Clear}(A)$
 - $\text{unstack}(B, A) : \text{On}(B, A), \text{Clear}(A), \text{HandEmpty}$
 -
 -
 -

- ゲームではプランの探索に通常 A* を用いる
 - アクションごとに重みを設定したい（例：通常攻撃と必殺技）
 - 多数の AI をリアルタイムに制御するゲームでは処理の効率化が求められる
 - あと、単純になじみ深い

コスト

- アクションごとに定義
- プランのコストはそのプランに含まれるアクションのコストの合計

ヒューリスティック

- その状態のうち、初期状態で満たされていない条件の数

- A* によるプランの探索①

```
function searchPlan(domain, state, goal)
  -- Run the A* algorithm to find a valid plan
  local openList = {}
  local closeList = {}

  -- Search from the goal state
  local goalNode = { state=goal, act=nil, next=nil, cost=0 }
  goalNode.diff, goalNode.diffCount = getDiffState(goalNode.state, state)
  goalNode.score = goalNode.diffCount
  table.insert(openList, goalNode)

  while #openList > 0 do
    local node = table.remove(openList, 1)

    if node.diffCount == 0 then
      return node
    end

    table.insert(closeList, node)

    local neighbors = getNeighborActions(domain, node.state)
    for key, var in next, neighbors, nil do
```

引数:ドメイン、ステート、ゴール

Openリスト、Closeリストの作成

目標状態から始める

初期状態に達したか
(ヒューリスティックの計算結果を流用)

• A* によるプランの探索②

状態を満たすアクションの取得

```
local neighbors = getNeighborActions(domain, node.state)
for key, var in next, neighbors, nil do
    local newState = table_merge(getDiffState(node.state, var.effect), var.precondition)
    local diff, diffCount = getDiffState(newState, state)
    local newCost = node.cost + var.cost
    local newScore = newCost + diffCount
    local newNode = { state=newState, act=var, next=node, cost=newCost, score=newScore, diff=diff, diffCount=diffCount }

    local indexOpen = getIndex(openList, newNode)
    if indexOpen > 0 then
        if openList[indexOpen].score > newNode.score then
            openList[indexOpen] = newNode;
        end
    else
        local indexClose = getIndex(closeList, newNode)
        if indexClose == 0 then
            table.insert(openList, newNode)
        end
    end
end

table.sort(openList, compareNode)
end

return false
end
```

- 新しい状態を作成
(アクションで満たされる条件の削除、事前条件の追加)
- 初期状態との差を取得
- コストの計算
- コスト+ヒューリスティック

- アクションの取得
 - ドメインにあるすべてのアクションが調査対象
 - 事後条件と状態を比べて、そのアクションを行った時に状態を満たすことができるかを調べる

```
function getNeighborActions(domain, state)
  local actions = {}
  for key, var in next, domain, nil do
    if isInState(var.effect, state) then
      table.insert(actions, var)
    end
  end
  return actions
end
```

- ステート
 - Luaのテーブルで表現するために少し変更

```
state = {}  
state.hand = ""  
state.on = { a="table", b="a", c="b" }  
state.isTop = { a=false, b=false, c=true }
```

- OnTable(a) → on[a] = "table"
- On(a, b) → on[a] = b
- Clear(a) → isTop[a] = true
- Holding(a) → hand = a
- HandEmpty → hand = ""

TIPS

- STRIPSではステートは可能な限り少なくしたほうが探索時間を抑えられる
- 真偽値で表現できる場合はそちらを使う

• アクション

-- テーブル上の a を手に持つ

```
function pickup(a)
    local pre = { hand = "", isTop = {}, on = {} }
    pre.isTop[a] = true
    pre.on[a] = "table"

    local effect = { hand = a, isTop = {}, on = {} }
    effect.isTop[a] = false
    effect.on[a] = "hand"

    return makeAction(pre, effect)
end
```

-- 手に持った a をテーブルの上に置く

```
function putdown(a)
    local pre = { hand = a, isTop = {}, on = {} }
    pre.isTop[a] = false
    pre.on[a] = "hand"

    local effect = { hand = "", isTop = {}, on = {} }
    effect.isTop[a] = true
    effect.on[a] = "table"

    return makeAction(pre, effect)
end
```

-- 手に持った a を b の上に置く

```
function stack(a, b)
    local pre = { hand = a, isTop = {}, on = {} }
    pre.isTop[a] = false
    pre.isTop[b] = true
    pre.on[a] = "hand"

    local effect = { hand = "", isTop = {}, on = {} }
    effect.isTop[a] = true
    effect.isTop[b] = false
    effect.on[a] = b

    return makeAction(pre, effect)
end
```

-- b の上の a を手に持つ

```
function unstack(a, b)
    local pre = { hand = "", isTop = {}, on = {} }
    pre.isTop[a] = true
    pre.isTop[b] = false
    pre.on[a] = b

    local effect = { hand = a, isTop = {}, on = {} }
    effect.isTop[a] = false
    effect.isTop[b] = true
    effect.on[a] = "hand"

    return makeAction(pre, effect)
end
```

実行例

- AB → BA

```
state = {}  
state.hand = ""  
state.on = { a="b", b="table" }  
state.isTop = { a=true, b=false }
```

```
goal = {}  
goal.on = { b="a" }
```

```
plan = strips(domain, state, goal)
```



```
1: unstack, a, b  
2: putdown, a  
3: pickup, b  
4: stack, b, a
```

- CBA → AC

```
state = {}  
state.hand = ""  
state.on = { a="table", b="a", c="b" }  
state.isTop = { a=false, b=false, c=true }
```

```
goal = {}  
goal.on = { a="c" }
```

```
plan = strips(domain, state, goal)
```



```
1: unstack, c, b  
2: putdown, c  
3: unstack, b, a  
4: putdown, b  
5: pickup, a  
6: stack, a, c
```

FPSのキャラクタAI

- 近接格闘と射撃での攻撃ができるキャラクタでターゲットとなる敵を倒す
- 敵に近づくと危ないので可能な限り射撃で攻撃してほしい

可能な行動

- 近接攻撃
- 射撃
- リロード
- 移動



• アクション

```
-- 近接攻撃
function melee()
    local pre = { hasTarget = true, atTarget = true }
    local effect = { hasTarget = false, atTarget = false }

    return makeAction(pre, effect)
end

-- 射撃
function shot()
    local pre = { hasTarget = true, hasAmmo = true, canSeeTarget = true }
    local effect = { hasTarget = false, canSeeTarget = false }

    return makeAction(pre, effect)
end

-- リロード
function reload()
    local pre = { hasAmmo = false, hasMagazine = true }
    local effect = { hasAmmo = true, hasMagazine = false }

    return makeAction(pre, effect)
end

-- ターゲットまで移動する
function moveToTarget()
    local pre = { atTarget = false }
    local effect = { atTarget = true }

    return makeAction(pre, effect)
end

-- 射撃可能な場所へ移動する
function moveToShootingPoint()
    local pre = { canSeeTarget = false }
    local effect = { canSeeTarget = true }

    return makeAction(pre, effect)
end
```

振る舞いの変更

- コストでの調整（近接攻撃をしにくくする）

```
-- 近接攻撃
function melee()
    local pre = { hasTarget = true, atTarget = true }
    local effect = { hasTarget = false, atTarget = false }

    return makeAction(pre, effect)
end
```



```
-- 近接攻撃
function melee()
    local pre = { hasTarget = true, atTarget = true }
    local effect = { hasTarget = false, atTarget = false }

    return makeAction(pre, effect, 5)
end
```

- 事前条件を追加（事前にリロードさせる）

```
-- 射撃可能な場所へ移動する
function moveToShootingPoint()
    local pre = { canSeeTarget = false }
    local effect = { canSeeTarget = true }

    return makeAction(pre, effect)
end
```



```
-- 射撃可能な場所へ移動する
function moveToShootingPoint()
    local pre = { hasAmmo = true, canSeeTarget = false }
    local effect = { canSeeTarget = true }

    return makeAction(pre, effect)
end
```

欠点

- アクションが多くなると管理が難しくなることがある
- あまり直感的でない

アジェンダ

- プランニングの概要
- STRIPS
 - アルゴリズム
 - 使用例
- HTNプランニング
 - アルゴリズム
 - 使用例
- ゲームでの利用
- まとめ

HTNプランニング

- 抽象的なタスクをより具体的なタスクへ分割していくことで必要な行動とその順序を見つけるアルゴリズム
- タスクをどのように分割するかを事前に定義しておく
プランナはそのタスクネットワーク(Hierarchical Task Networks)を使って問題を解く
- 人が行動を計画するときの思考と似ているので
実世界の問題を扱う場合、構築しやすいことが多い
- 分割していきだけでプランを見つけられるので非常に効率がいい

HTNプランナに与える情報

- **プリミティブタスク**: それ以上分割のしようがないタスク
 - 事前条件と事後条件のセット、STRIPSにおけるアクション
- **複合タスク**: タスクをどのように分割するかを定義した抽象的なタスク
 - どの条件のときにこういったサブタスクに分割するかを必要なだけ定義する
- **ドメイン**: プリミティブタスクと複合タスクの集合
- **ステート**: 現在の状態
- **ゴール**: 達成してほしいタスク

旅行問題

- ある目的地に行くにはいくつもの手段(乗り物)とルートがある
- すべての候補のうち、現在の状況で利用可能で、かつ最適なものをしらみつぶしに調べるには探索空間が大きすぎる
- 多くの場合、人はこういう時にはこういう方法をとるべきという知識を持っていてそれを活用することで現実的な時間で計画ができる

複合タスクとして記述する



東京の家から福岡駅に行く

複合タスク

AからBに行く

- もし距離が遠くてお金がある
 - 飛行機で行く
 - (航空券を買う、空港Aに行く、飛行機に乗る、Bに行く)
- もし距離がそこそこ遠くてお金がある
 - 電車で行く
 - (駅Aに行く、切符を買う、電車に乗る、Bに行く)
- その場所にいない
 - 歩いていく
 - (歩く)

プリミティブタスク

- 歩く
- 切符を買う
- 電車に乗る
- 航空券を買う
- 飛行機に乗る

HTNプランニングのアルゴリズム

- ゴールとなるタスクをリストへ追加する
- リストがなくなるまでループ
 - リストからタスクを取得
 - もし複合タスクなら
 - 現在の状態に適したサブタスクに分割する
 - サブタスクをリストへ追加する
 - もしプリミティブタスクなら
 - タスクの事後条件で現在の状態を更新する
 - プランに追加する
- プランを返す

動作例

東京の家から福岡駅に行く



ステート

所持金 : 100,000円

航空券を買う、羽田空港に行く、飛行機に乗る、福岡駅に行く

↓ プランに追加、所持金 : $100,000\text{円} - 50,000\text{円} = 50,000\text{円}$

航空券を買う、羽田空港に行く、飛行機に乗る、福岡駅に行く



航空券を買う、東京駅に行く、切符を買う、電車に乗る、羽田空港に行く、
飛行機に乗る、福岡駅に行く

⋮

航空券を買う、東京駅まで歩く、切符を買う、電車に乗る、飛行機に乗る、切符を買う、電車に乗る

動作例

東京の家から福岡駅に行く



福岡駅まで歩く



福岡駅まで歩く

ステート

所持金 : 100円

- 実装も比較的シンプル

(この実装は再帰を使っているのでスタックオーバーフローだけ注意！)

```
-- Primitive Task
if domain.primitive[taskName] ~= nil then
    local newState = deepcopy(state)
    local res = domain.primitive[taskName](newState, table.unpack(task))
    if res == true then
        return htn_internal(domain, newState, tasks, array_merge(plan, {{taskName, table.unpack(task)}}))
    else
        return false
    end
end

-- Compound Task
elseif domain.compound[taskName] ~= nil then
    for i, func in ipairs(domain.compound[taskName]) do
        local res = func(state, table.unpack(task))
        if res ~= false then
            res = htn_internal(domain, state, array_merge(res, tasks), plan)
            if res ~= false then
                return res
            end
        end
    end
end
end
```

状態を更新

プランに追加して残りのタスクを処理

サブタスクに分割

タスクのリストに追加して続ける

- プリミティブタスク

-- 航空券を買う

```
domain.primitive.buyAirTicket = function(state, a, b)
  local price = getAirPrice(a, b)
  if state.money >= price then
    state.money = state.money - price
    return true
  end
  return false
end
```

-- 飛行機に乗る

```
domain.primitive.rideAirplane = function(state, a)
  state.at = a
  return true
end
```

-- 歩く

```
domain.primitive.walk = function(state, a)
  state.at = a
  return true
end
```

-- 切符を買う

```
domain.primitive.buyTrainTicket = function(state, a, b)
  local price = getTrainPrice(a, b)
  if state.money >= price then
    state.money = state.money - price
    return true
  end
  return false
end
```

-- 電車に乗る

```
domain.primitive.rideTrain = function(state, a)
  state.at = a
  return true
end
```

- 複合タスク

```
-- すでにいる
if from == to then
    return {}
end

local dist = getDistance(from, to)

-- 飛行機
if dist > 300 then
    local airA = getAirport(from)
    local airB = getAirport(to)
    local airPrice = getAirPrice(airA, airB)
    if airPrice ~= nil and airPrice <= state.money then
        return {"buyAirTicket", airA, airB}, {"move", airA}, {"rideAirplane", airB}, {"move", to}
    end
end

-- 電車
if dist > 2 then
    local stationA = getStation(from)
    local stationB = getStation(to)
    local trainPrice = getTrainPrice(stationA, stationB)
    if trainPrice ~= nil and trainPrice <= state.money then
        return {"move", stationA}, {"buyTrainTicket", stationA, stationB}, {"rideTrain", stationB}, {"move", to}
    end
end

-- 徒歩
return {"walk", to}
```

実行例

- 所持金: 100,000円

```
state = {}  
state.at = "home"  
state.money = 100000  
  
plan = htn(domain, state, {"move", "fukuoka"})
```



```
1: buyAirTicket, haneda_airport, fukuoka_airport  
2: walk, tokyo  
3: buyTrainTicket, tokyo, haneda_airport  
4: rideTrain, haneda_airport  
5: rideAirplane, fukuoka_airport  
6: buyTrainTicket, fukuoka_airport, fukuoka  
7: rideTrain, fukuoka
```

- 所持金: 100円

```
state = {}  
state.at = "home"  
state.money = 100  
  
plan = htn(domain, state, {"move", "fukuoka"})
```



```
1: walk, fukuoka
```

STRIPSとの比較

メリット

- 高速にプランを見つけることができる
- 分割方法を制御できるため知識表現力が高く、AI の振る舞いの変更や調整がしやすい
- ドメインの定義が人にとって自然な方法で行われているのでAI の振る舞いを把握しやすい

デメリット

- 問題によっては複合タスクの記述が複雑になってしまう

- プリミティブタスク
 - STRIPSのアクションとほぼ同等

```
-- テーブル上の a を手に持つ
domain.primitive.pickup = function(state, a)
    if state.hand == "" and state.isTop[a] == true and state.on[a] == "table" then
        state.hand = a
        state.isTop[a] = false
        state.on[a] = "hand"
        return true
    end
    return false
end

-- 手に持った a をテーブルの上に置く
domain.primitive.putdown = function(state, a)
    if state.hand == a then
        state.hand = ""
        state.isTop[a] = true
        state.on[a] = "table"
        return true
    end
    return false
end
```

```
-- 手に持った a を b の上に置く
domain.primitive.stack = function(state, a, b)
    if state.hand == a and state.isTop[b] == true then
        state.hand = ""
        state.isTop[a] = true
        state.isTop[b] = false
        state.on[a] = b
        return true
    end
    return false
end

-- b の上の a を手に持つ
domain.primitive.unstack = function(state, a, b)
    if state.hand == "" and state.isTop[a] == true and state.on[a] == b then
        state.hand = a
        state.isTop[a] = false
        state.isTop[b] = true
        state.on[a] = "hand"
        return true
    end
    return false
end
```


- ブロックを持つ

```
domain.compound.get = {  
  function(state, a)  
    if state.hand == "" and state.isTop[a] == true then  
      if state.on[a] == "table" then  
        return {"pickup", a}  
      else  
        return {"unstack", a, state.on[a]}  
      end  
    end  
    return false  
  end  
}
```

- ブロックを置く

```
domain.compound.put = {  
  function(state, a, b)  
    if state.hand == a then  
      if b == "table" then  
        return {"putdown", a}  
      elseif state.isTop[b] == true then  
        return {"stack", a, b}  
      end  
    end  
    return false  
  end  
}
```

テーブルが関係しているかで
使用するプリミティブタスクを切り替える

- ブロックの移動

```
domain.compound.move = {  
  function(state, a, b)  
    if state.hand == a then  
      if state.isTop[b] == true then  
        return {"put", a, b}  
      else  
        return {"clear", b}, {"put", a, b}  
      end  
    elseif state.isTop[a] == false then  
      return {"clear", a}, {"move", a, b}  
    elseif b ~= "table" and state.isTop[b] == false then  
      return {"clear", b}, {"move", a, b}  
    else  
      return {"get", a}, {"put", a, b}  
    end  
    return false  
  end  
end  
}
```

もしAをすでに持っていれば
Bに置く、もしくは
Bの上のものをどかしてからBに置く

もしAの上に何かが乗っていれば
それをどかしてから移動させる

もしBの上に何かが乗っていれば
それをどかしてから移動させる

AをとってBに置く

- 目的の状態へブロックを積みなおす
 - テーブルに近いところから確定させていけばいいというアプローチ
 - 結構複雑・・・ (AIデザイナーに任せるには難しい)

```
domain.compound.achieve = {  
  function(state, goal)  
    -- テーブルから何段目かを取得する  
    local getStepNum = function(block, goal)  
      local b = block  
      local i = 0  
      while goal.on[b] ~= "table" do  
        i = i + 1  
        b = goal.on[b]  
      end  
      return i  
    end  
  end
```



```
    local block, minStep = "", 10000  
    for key, var in next, goal.on, nil do  
      if state.on[key] ~= goal.on[key] then  
        -- 目標の状態がテーブル上か  
        if goal.on[key] == "table" then  
          return {{ "move", key, "table" }, { "achieve", goal }}  
        end  
        -- テーブルに近いか  
        local step = getStepNum(key, goal)  
        if step < minStep then  
          block = key  
          minStep = step  
        end  
      end  
    end  
    -- 目標の状態が一番下段に近いブロックを移動  
    if minStep < 10000 then  
      return {{ "move", block, goal.on[block] }, { "achieve", goal }}  
    end  
    -- 目標を達成した  
    return {}  
  end  
end
```

FPSのキャラクタAI

- 近接格闘と射撃での攻撃ができるキャラクタでターゲットとなる敵を倒す
- 敵に近づくと危ないので可能な限り射撃で攻撃してほしい

可能な行動

- 近接攻撃
- 射撃
- リロード
- 移動



• プリミティブタスク

-- 近接攻撃

```
domain.primitive.melee = function(state)
  if state.hasTarget == true and state.atTarget == true then
    state.hasTarget = false
    state.atTarget = false
    return true
  end
  return false
end
```

-- 射撃

```
domain.primitive.shot = function(state)
  if state.hasTarget == true and state.hasAmmo == true and state.canSeeTarget == true then
    state.hasTarget = false
    state.canSeeTarget = false
    return true
  end
  return false
end
```

-- リロード

```
domain.primitive.reload = function(state)
  if state.hasAmmo == false and state.hasMagazine == true then
    state.hasAmmo = true
    state.hasMagazine = false
    return true
  end
  return false
end
```

-- ターゲットまで移動する

```
domain.primitive.moveToTarget = function(state)
  if state.atTarget == false then
    state.atTarget = true
    return true
  end
  return false
end
```

-- 射撃可能な場所へ移動する

```
domain.primitive.moveToShootingPoint = function(state)
  if state.hasAmmo == true and state.canSeeTarget == false then
    state.canSeeTarget = true
    return true
  end
  return false
end
```

- 複合タスク

```
-- ターゲットを倒す
domain.compound.killTarget = {
  -- 射撃
  function(state)
    return {"prepareShooting"}, {"shot"}}
  end,
  -- 近接攻撃
  function(state)
    if state.atTarget == true then
      return {"melee"}}
    end
    return {"moveToTarget"}, {"melee"}}
  end
end
}
```

先に射撃を試して
タスクの分割に失敗したら
近接攻撃

```
-- 射撃準備をする
domain.compound.prepareShooting = {
  function(state)
    if state.hasAmmo == false then
      if state.hasMagazine == true then
        return {"reload"}, {"prepareShooting"}}
      else
        return false
      end
    elseif state.canSeeTarget == false then
      return {"moveToShootingPoint"}, {"prepareShooting"}}
    end

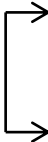
    -- 準備完了
    return {}
  end
}
```

残りの準備を完了させる

振る舞いの変更

- 条件判定順の変更

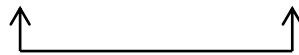
```
-- ターゲットを倒す
domain.compound.killTarget = {
  -- 射撃
  function(state)
    return {"prepareShooting"}, {"shot"}
  end,
  -- 近接攻撃
  function(state)
    if state.atTarget == true then
      return {"melee"}
    end
    return {"moveToTarget"}, {"melee"}
  end
end
}
```



- サブタスクの順番の変更

```
-- 射撃準備をする
domain.compound.prepareShooting = {
  function(state)
    if state.hasAmmo == false then
      if state.hasMagazine == true then
        return {"reload"}, {"prepareShooting"}
      else
        return false
      end
    elseif state.canSeeTarget == false then
      return {"moveToShootingPoint"}, {"prepareShooting"}
    end

    -- 準備完了
    return {}
  end
}
```



利点

- 直感的に調整しやすい

アジェンダ

- プランニングの概要
- STRIPS
 - アルゴリズム
 - 使用例
- HTNプランニング
 - アルゴリズム
 - 使用例
- ゲームでの利用
- まとめ

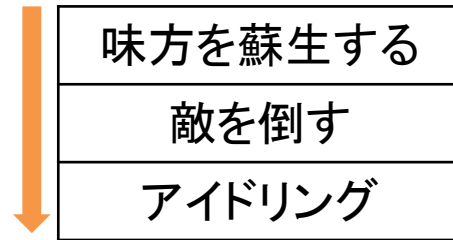
エージェントへの組み込み

ゲームのエージェントAIへプランニングを組み込むには
考慮しないといけないことがある

- ゴールの選択
 - プランナに与えるゴールをどのように決定するか
 - 上位に何かしらのゴール選択アルゴリズムを用意しないといけない
- プラン実行中の環境の変化への対応
 - ゲーム内の環境は毎フレーム変化しており
プラン実行中にもそれらに動的に対応しないといけない
 - 例: 倒そうと思っていた敵が先に味方に倒された

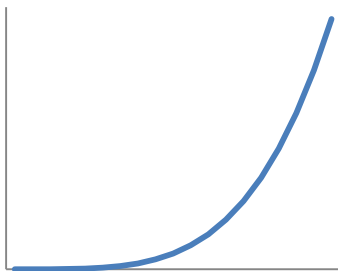
ゴールの選択

- 優先度付きのゴールリスト
 - それぞれのゴールに事前に優先度をつけておき優先度順にリストで管理する
 - 優先度の高いものからプランニングし、プランニングに失敗したら次に優先度の高いゴールを試す
 - プラン実行中でも定期的にプランニングを行い現在のプランよりも優先度の高いゴールが実行可能になれば、プランの実行を中止してそちらに移る

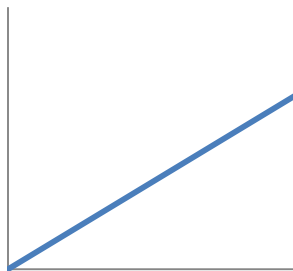


ゴールの選択

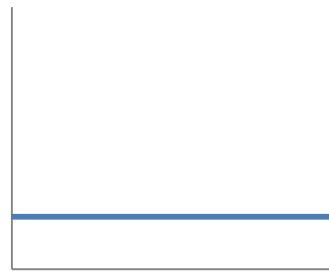
- ユーティリティシステム
 - 固定されたプライオリティだと柔軟な対処ができなかったり無駄なプランニングが起動されたりする
 - ゴールごとにそのゴールをどれだけ重視すべきか(達成したいか)をあらわすユーティリティ値を計算する
 - ユーティリティ値の高いものからプランニングしていく



蘇生



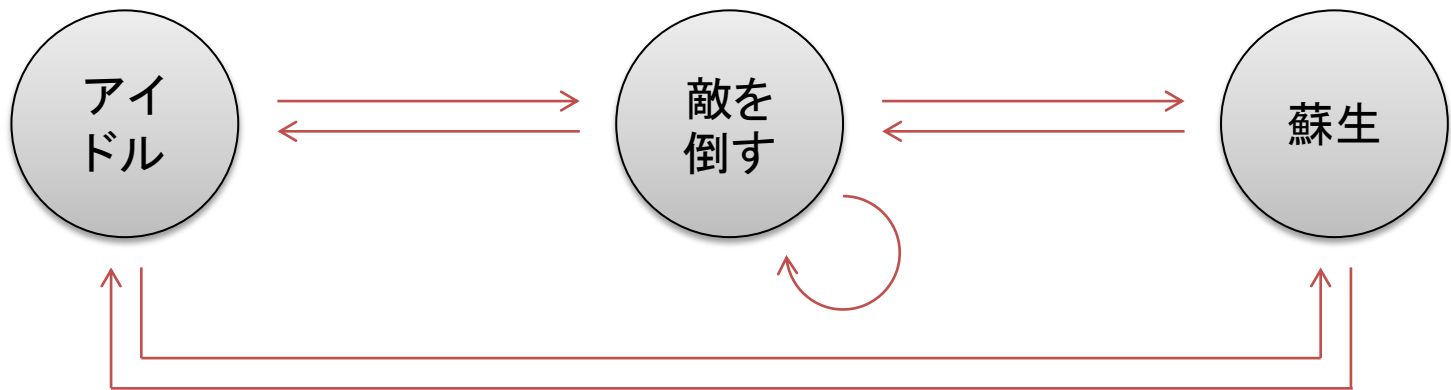
敵を倒す



アイドリング

ゴールの選択

- 簡易なステートマシン
 - ステート数が数個程度の簡易なステートマシンを上位に持つてくる
 - ステートの遷移時にそのステートにあったゴールでプランニングする
 - プランニングの頻度が減ることによって処理負荷への影響を抑えることができる



環境の変化への対応

- プランニング時に使ったアクションの事前条件をアクションの実行時にもう一度判定する
- プランニング時には考慮できない・しなくていいこともあるのでアクション実行時にだけ判定する条件を別途追加する
 - プランニング時には味方の行動まで考慮しないので自分がターゲットを倒すまでに他から倒されることは想定しない
- 失敗したらもう一度リプランニングする

環境の変化への対応

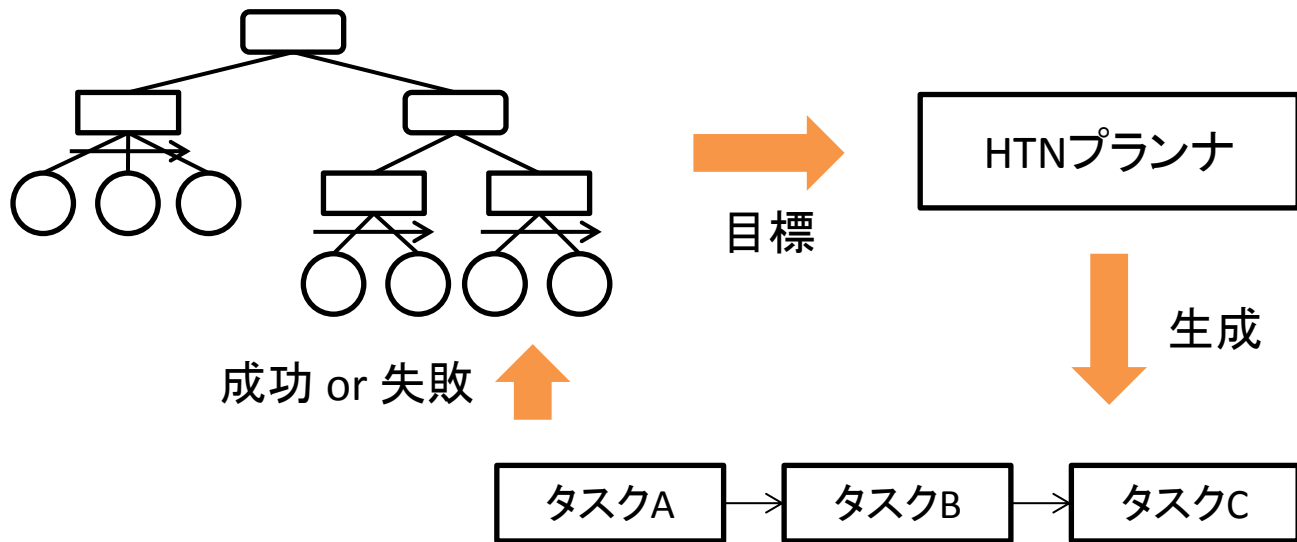
- HTNプランニング限定
 - プランニングしてからアクションの実行までに
間が空くから環境の変化にさらされる
 - 実際のタスクの実行に近づいてから分割するようにすれば
その時の最新の状態が反映され環境の変化に左右されにくい
 1. 直近で実行するプリミティブタスクが見つかったら
いったんプランニングを保留する
 2. 実行できるプリミティブタスクがなくなったら
保留していたプランニングを再開する

BNSでの事例

- タイムクライシス5、LOST REAVERS の AI エンジンでは BehaviorTree と HTN プランニングを使用
(複数タイトルで使われた柔軟性の高いAIエンジン, CEDEC2015)
- デザイナフレンドリーなエディタ環境としてはBehaviorTreeは抜群
- ハイブリッドなシステムにすることで
エディットのしやすさとAI開発の効率化を両立

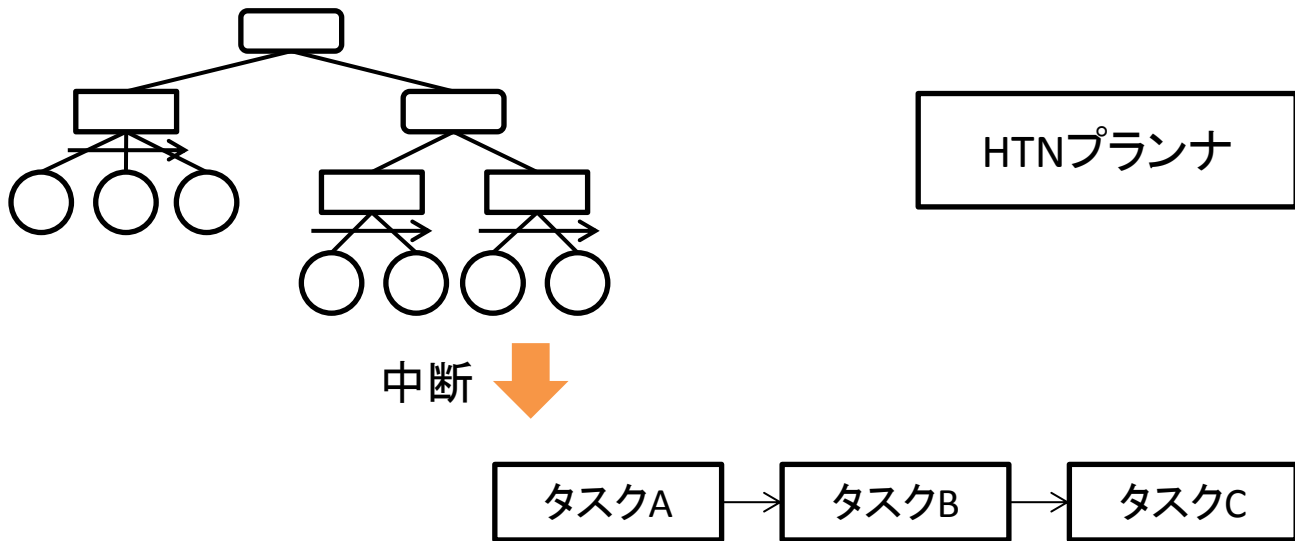
ハイブリッドシステム

- BehaviorTreeで決めた目標を達成するプランをHTNプランニングにより求める



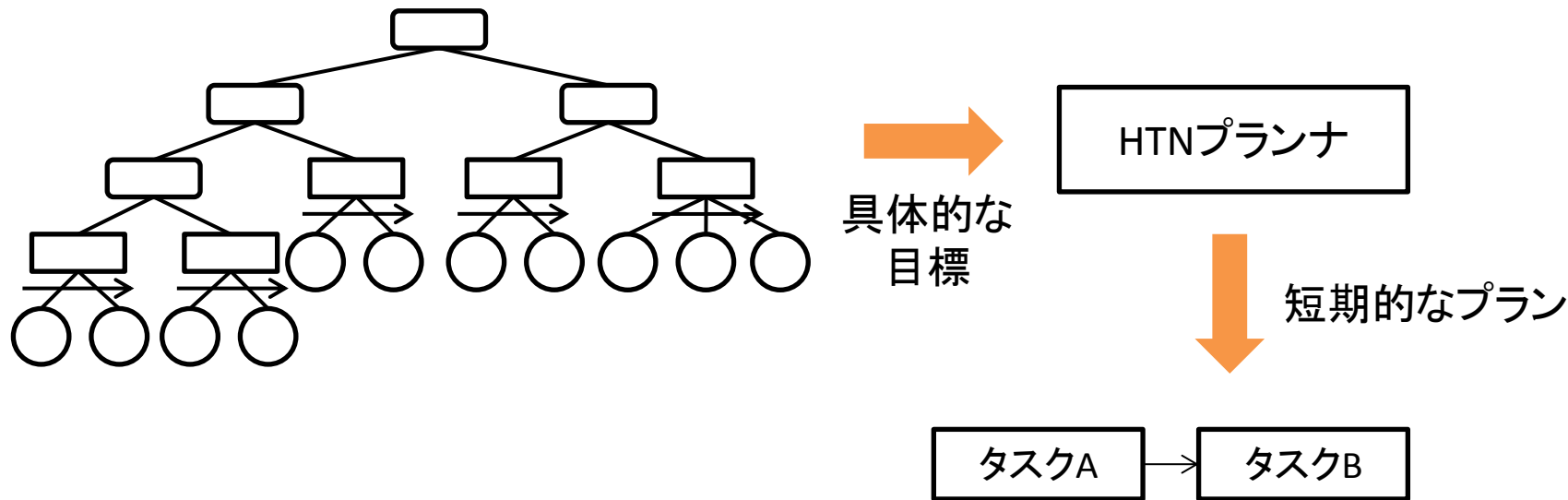
ハイブリッドシステム

- ほかに優先度の高い目標ができた場合などは BehaviorTree側からプランの中断を行う



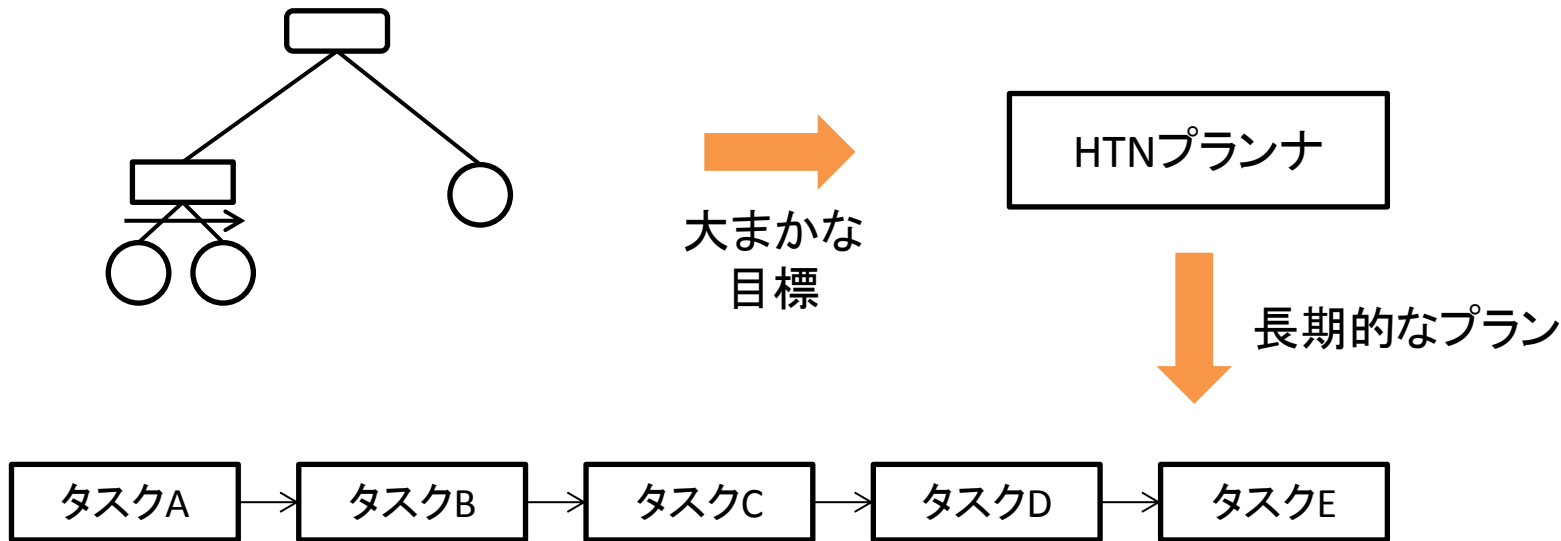
ハイブリッドシステム

- プランナに与える目標を具体的(=BehaviorTreeをメイン)にすれば
細かな制御が可能



ハイブリッドシステム

- プランナに与える目標を大まか(=HTNプランニングをメイン)にすれば
AIによる自律的な行動が可能



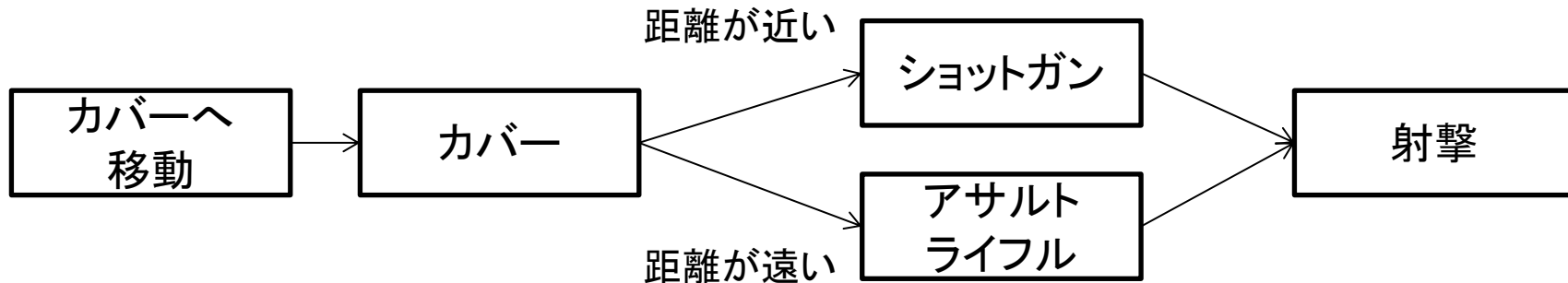
- HTNプランナにより生成されるプランも BehaviorTree として出力
- タスクのシーケンスとしてしか生成されなかったプランが BehaviorTreeの表現力を生かすことで柔軟に表現することができる
 - 遅延評価
 - 並列タスク
 - 一部プランニングの保留
- BehaviorTreeのデバッグ環境を流用できる利点も

ゲームでの例①

遅延評価

- プレイヤーの行動などプランニング時には予測できないことをその時になってから評価する

(例) カバーポイントから攻撃したい

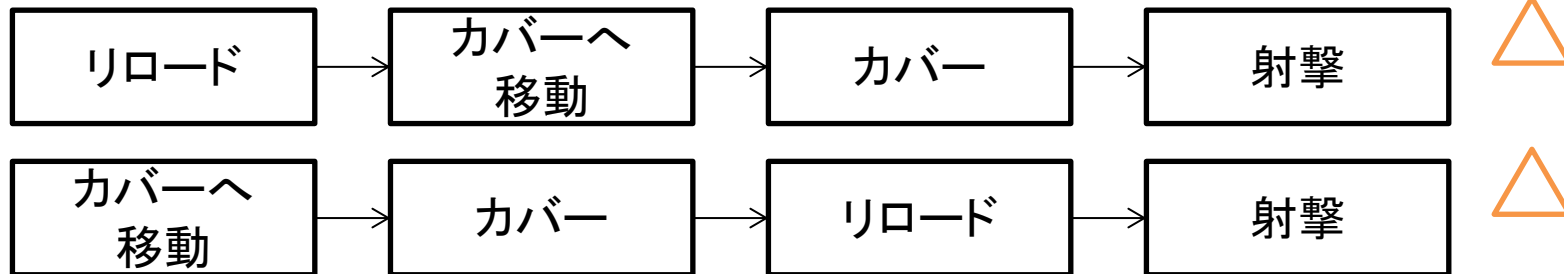


ゲームでの例②

並列タスク

- 同時に行える(そのほうが自然な)タスクは同時に行いたい

(例)カバーポイントから攻撃したい(マガジンの弾が空)

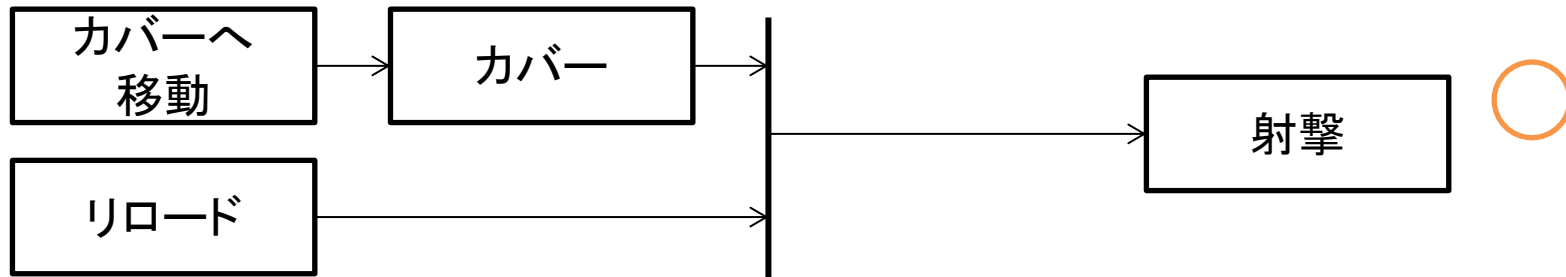


ゲームでの例②

並列タスク

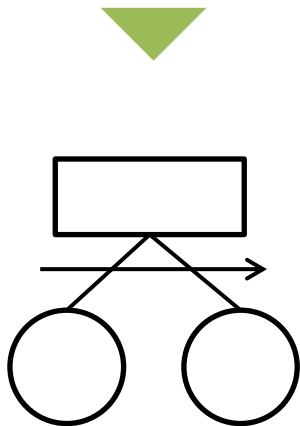
- 同時に行える(そのほうが自然な)タスクは同時に行いたい

(例)カバーポイントから攻撃したい(マガジンの弾が空)

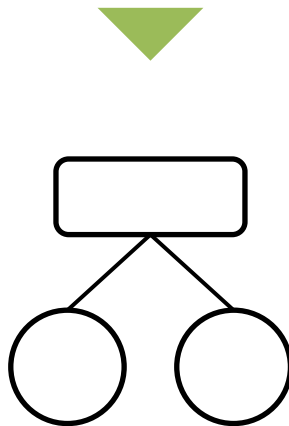


BehaviorTreeへの展開

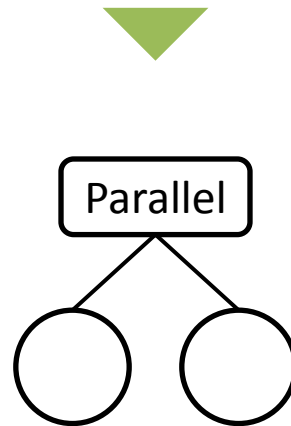
(タスク1, タスク2)



priority(タスク1, タスク2)

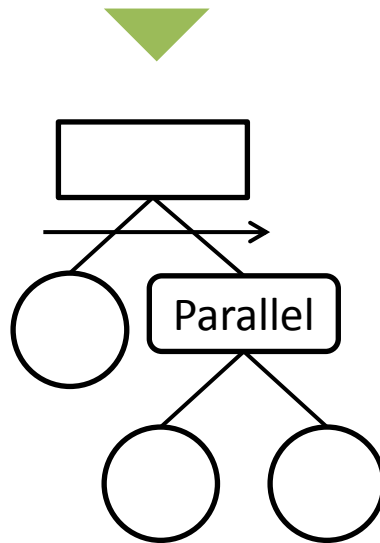


parallel(タスク1, タスク2)



BehaviorTreeへの展開

(タスク1, parallel(タスク2, タスク3))



もちろんBehaviorTreeのほかのノードも使える

戦闘

if ターゲットがない
(敵を探す)

if カバーがある
(カバーから戦う)

if true
(距離を取って戦う)

距離を取って戦う

if true
(戦闘準備, parallel(距離を取る, 射撃))

カバーから戦う

if カバーがある
(parallel(カバーする, 戦闘準備), priority(射線 ×, 射撃))

カバーする

if すでにカバーにいる
(カバー)

if カバーにいない
(カバーへ移動, カバー)

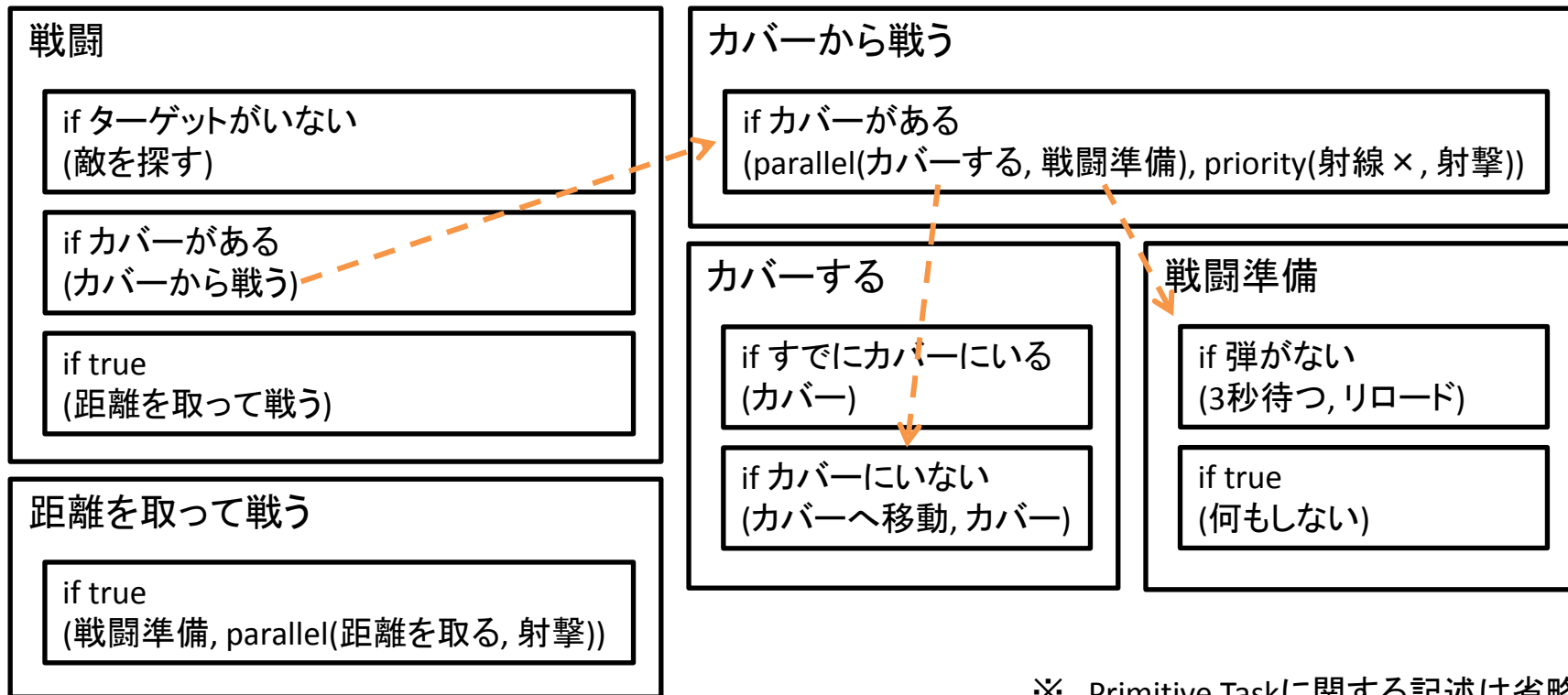
戦闘準備

if 弾がない
(3秒待つ, リロード)

if true
(何もしない)

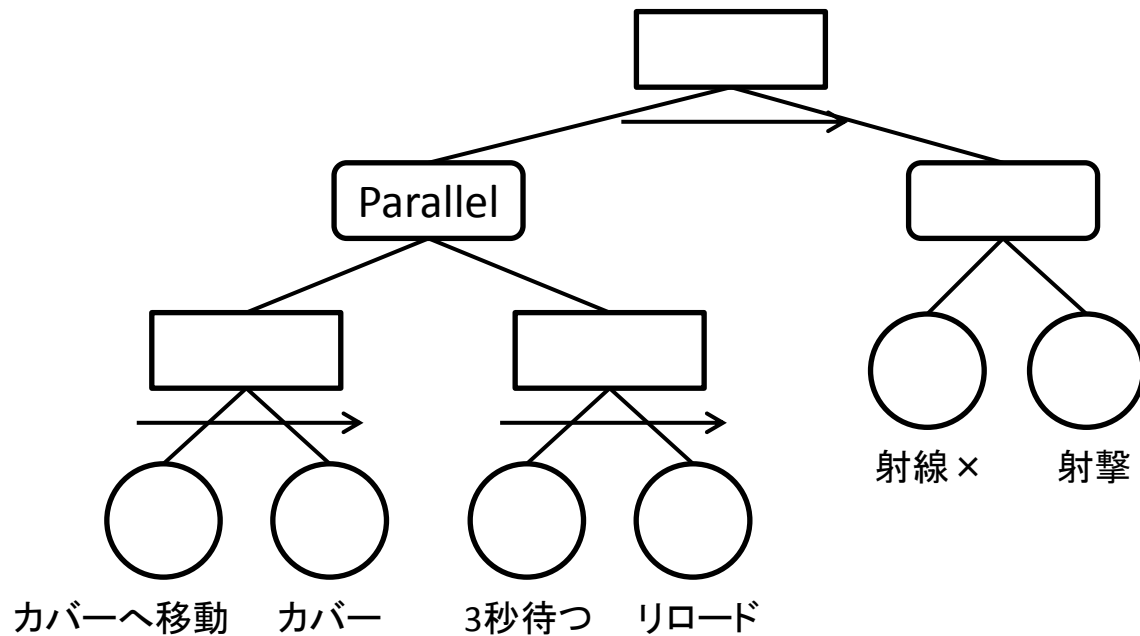
※ Primitive Taskに関する記述は省略

例：戦闘（カバーあり、弾なし）



※ Primitive Taskに関する記述は省略

例：戦闘（カバーあり、弾なし）



例：戦闘（カバーなし、弾あり）

戦闘

if ターゲットがない
(敵を探す)

if カバーがある
(カバーから戦う)

if true
(距離を取って戦う)

距離を取って戦う

if true
(戦闘準備, parallel(距離を取る, 射撃))

カバーから戦う

if カバーがある
(parallel(カバーする, 戦闘準備), priority(射線 ×, 射撃))

カバーする

if すでにカバーにいる
(カバー)

if カバーにいない
(カバーへ移動, ~~カバー~~)

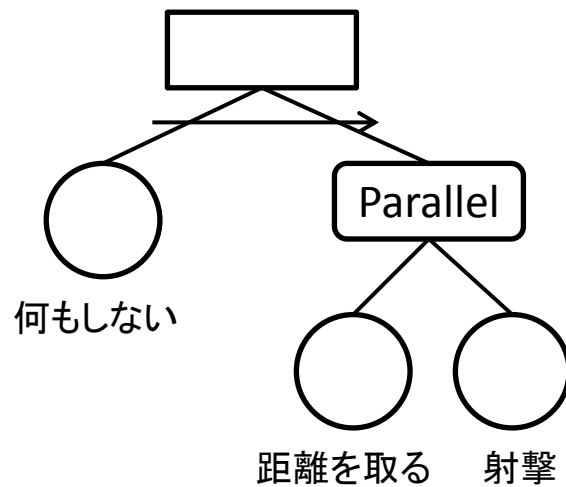
戦闘準備

if 弾がない
(3秒待つ, リロード)

if true
(何もしない)

※ Primitive Taskに関する記述は省略

例：戦闘（カバーなし、弾あり）



アジェンダ

- プランニングの概要
- STRIPS
 - アルゴリズム
 - 使用例
- HTNプランニング
 - アルゴリズム
 - 使用例
- ゲームでの利用
- まとめ

まとめ

- ゲームが複雑になり、グラフィックがリッチになっていく中で、従来の AI の作り方は通用しにくくなっている
- 見た目におかしくない自然な振る舞いをする
合理的なキャラクタを作るには将来も見据えた
中長期的なプランニングが必要になる
- ゲームでは STRIPS と HTNプランニングがよく用いられる
- STRIPSが適した場面とHTNプランニングが適した場面があるので
問題に合わせてどちらを使うかはよく考えないといけない

STRIPS

メリット

- 少数のアクションの定義で AI を構築できる
- ゲームAIへの採用事例が多い
- ゲーム開発者にとってなじみ深いアルゴリズム(A*)を使用しているため実装が理解しやすい

デメリット

- 意図しないプランの生成
- 条件の定義のし忘れ等による不具合
- デバッグが困難

HTNプランニング

メリット

- 高速な動作
- タスクを分割していくという人にとって自然な方法のため
AIの構築がしやすく、振る舞いの調整も簡単
- ゲームのような動的な環境に対応しやすい

デメリット

- STRIPSに比べると定義しないといけないものが多い

サンプルコード

https://github.com/yhase7/lua_planner

- すべてのコードを公開中！
- MITライセンス
- コード、ライセンスは変更される可能性があります