LGOAP

Adaptive layered planning for real-time video games

Dr. Giuseppe Maggiore

NHTV University of Applied Sciences
Breda, Netherlands

Agenda

NPCs in games

Realism

- Non-playing characters (NPCs) in modern games often exhibit behavior that is strategically and tactically inferior
- In general, NPC behaviour is clearly artificial and unrealistic

NPCs in games

Realism

- We wish NPCs who are part of the game world as much as the player is
- They should *play* the game rather than *be a passive part* of it
- Anything that the player can do, NPCs should be able to do as well

NPCs in games

Realism

- NPCs should be able to recognize the actions available to them
- NPCs should choose the best course of action because it makes sense according to their internal logic...
- ...either to fulfill their needs or to reach some goal

NPCs in games

Problem statement

To what extent can planning be used to create adaptive behaviour of NPCs that allows them to achieve long and short term goals in a real-time game or simulation?

NPCs in games

Idea

- *create and execute, in real-time, plans that consist of sequences of actions*
- Actions are defined as transitions in the game world state
- Multiple actions are available to an NPC in a given state, but only one can be chosen for the final plan
- We define the game world as a distribution of resources over entities and agents
- Actions cause only changes to that distribution
- This allows us to consider actions as transitions in a graph, across which the planner finds an optimal path in order to reach the desired state

NPCs in games

Idea, part 2

- how do we model an abstract game world so that the approach works on any concrete game world without modification?
- what search algorithm can we use that can be executed sufficiently fast?
- how can we deal with NPC personalization?
- how can we deal with the fact that an NPC can only plan his own actions, while the future world state depends on the actions of all NPCs?

Ontology of actions

Entities, resources, actions

- We define the game world as a series of *entities* (such as a table, a chair, a gun, a gem, etc.)
- A series of *agents* are the playing and non-playing characters
- Entities may change dynamically: their locations may vary and they may be added or removed from the game world
- All entities contain *resources* (or *stats*)
- All agents contain resources as well
- When an agent interacts with an entity, it does so by activating an *action*
  - An action exchanges resources between entities and agents
  - Actions may change dynamically and may be added or removed

Simple example

```
L=life         Sword      Agent      Ogre
A=attack        L=1        L=5        L=5
*=entity        A=2        A=0        A=1       Goal
-=tile           *----------*----------*----------*
```

Ontology of actions

Entities, resources, **actions**

- `Sword.pickup`, which increases the attack rating of the agent who picks it up and decreases the life of the sword;
- `Tile.walk` that changes the current location of an agent;
- `Ogre.fight` that subtracts the attack resource of the agent from the life resource of the ogre and vice-versa.

Ontology of actions

Example run

Example run

Ontology of actions

Entities, resources, actions

- Extensions without much effort
- For example, we could add a new resource, `key`
- We can now add a room to the dungeon which requires a key in order to obtain the sword

Extended example

```
L=life                   Agent      Ogre
A=attack     Key          L=5        L=5
K=key        K=1          A=0        A=1       Goal
*=entity      *----------*----------*----------*
_=tile                    |
                          | Door
                          *  K=1
                          |
                          | Sword
                          *  A=2
```

Ontology of actions

New actions

- `Key.pickup`, which adds to an agent a key resource
- `Door.open`, which subtracts from the door key resource the agent's key resource

Ontology of actions

Example run

Example run

Ontology of actions

Constraint logic programming

- Framework connected to *constraint logic programming* [2]
- Actions change the resources of the various entities of the game world if some preconditions are met

Sword.pickup

```
Agent.L > 0 ∧ Sword.L > 0 ∧ Agent.P = Sword.P → Sword
    .L := 0 ∧ Agent.A := Agent.A + 2
```

Planning

Event horizon

- A game state is a distribution of resources over entities and agents
- Actions move the game world from one state to another
- Current NPCs blindly pick one action at a time, without considering the medium and long term consequences of their decisions
- Planning solves this problem by giving the AI the ability to select *sequences* of actions

Planning

Planning as path-fiding

- Such a planning algorithm is a (multi-dimensional) path-finder
- Explore a large graph where:
  - Nodes are *all* the valid states of the game world
  - Actions represent transitions from one of the valid states into another one
  - The new state of the game world will be forward in time
  - As a result of the actions used it will have different resources, entities, and available actions
- Graph exploration techniques are connected to planning [8]

Planning

Naïve planning

- Planning in a game can be done naïvely with backtracking on all the possible sequences of actions
- Backtracking ends after a satisfactory plan is found
- Appropriate failure conditions must be taken into account (upper bound on number of actions, timeout, etc.)

Naïve planning

```
find_plan(world,agent,steps) =
  if length(steps) > MAX_STEPS then
    return null
  if goal_reached(world,agent) then
    return steps
  else
    for action in available_actions(world,agent) do
      world',agent' = simulate_action(world,agent,
           action)
      result = find_plan(world',action',action::steps)
      if result <> null then
        return result // this plan reaches the goal
    return null // failure
```

Planning

Cost of naïve planning

- Back-tracking is effective but inefficient
- Assume that at every step there are at least A actions available
- Assume that plans with more than N actions are discarded

Planning

Cost of naïve planning

- Back-tracking is effective but inefficient
- Assume that at every step there are at least A actions available
- Assume that plans with more than N actions are discarded
- Every possible sequence of actions of length N may be explored
- The complexity of the algorithm is $O(A^N)$
- This number will quickly become too large to be feasible

Planning

Optimization

- Heuristic search [1]: a less wasteful variation of backtracking
- Some *inferior* plans are not explored at all Take into account the resource constraints of the possible actions [3]
- Steer the planner towards plans that maintain desirably high levels of certain important resources
- Steer the planner towards a given goal
- For example, a plan that reduces the `health` of an NPC to zero can safely be ignored

Planning

Path-finding

- A graph containing all reachable game worlds is too large to warrant straightforward exploration
- Cannot use Dijkstra's algorithm
- We can use *iterative deepening depth-first search* (IDDFS) [9]
- Each iteration increases the depth of exploration until the shallowest goal state is reached
- IDDFS is a form of breadth-first search
- In particular, we use IDA* [7]

Planning

IDA*

- IDA* is an informed search, because it expands the reached nodes according to some heuristic
- A node represents a possible state of the game world
- We pick and expand the *most desirable* state
- The desirability of a game world is a heuristic which may vary depending on the specific scenario
- For example, an ordering of the NPC resources

Planning

IDA*

- This heuristic drive has two side-effects: *(i)* finds plans that satisfy the goal while optimizing some resources; and *(ii)* speed up the search process by focusing on promising states
- Further speed up: trim the set of working plans to a maximum size
- Warning: trade-off between search thoroughness and speed
- Smaller working set means faster
- Too small working set may cull (temporarily bad) plans

Heuristic search

```
find_plan_fast(world,agent) =
  Q = {(world,agent)}
  C = 0
  do
    if ∃ p ∈ Q : reaches_goal(p) then
      return p
    Q ← { simulate_action(p,a) : p ∈ Q ∧ a ∈
        available_actions(world,agent) }
    Q ← take_best_M()
  while Q ≠ ∅ ∧ Steps < N
  return null
```

Planning

Heuristic search complexity

- Resulting complexity becomes $O(N \times M \times A)$, where M is the maximum size of the queue
- Optimizations can partially mitigate slowness of the algorithm
- Increasing the target length of plans steeply increases the computation time required
- Unfortunately, we need long plans

Planning

Hierarchy of planners

- Further reduce complexity
- Hierarchical system similar to [10]
- Multiple planners nested inside one another
- Highest level planners actions span a long time
- Each action involves large changes in resources
- Lower level planners find plans that respect the constraints given by the higher layers
- Lower level actions are more concrete, take less time, and generally involve smaller resource exchanges

Planning

Hierarchy of planners

- Consdier three layers in the RPG example

Highest level `complete quest X`, `acquire sword`, `fight ogre`

Middle layer `use item X`

Lowest layer `move to X`

Planning

Hierarchy of planners

- Final planning system invokes the fast planning function multiple times
  - Different actions
  - Different goal

Layered planner

```
find_plan_layered(world,agent) =
  plan = find_plan_fast_L(world,agent)
  for l = L-1 downto 0 do
    new_plan ← []
    for (world,agent,action) in plan
      new_plan ← find_plan_fast_l,action(world,agent) ::
          new_plan
    plan ← new_plan
  return plan
```

Planning

Layering complexity

- New algorithm has significantly decreased computational load
- Our planner can now plan for much longer periods of time
- Algorithm invokes the fast planning algorithm once for every layer (L times), and for each action found by each layer (N times)
- $O(L \times N^2 \times M \times A)$
- Final plan, entirely composed of lowest level actions is now long N^L actions

Planning

Complexities

Algorithm	Complexity	Steps
Naïve	$O(A^{T_{tgt}/T})$	10^{100}
Fast	$O(\frac{T_{tgt}}{T} \times M \times A)$	100000
Layered	$O(L \times \sqrt[L]{\frac{T_{tgt}}{T}}^2 \times M \times A)$	15000

Table: Steps per algorithm for $M = 100$, $A = 10$, $T_{tgt}/T = 100$, $L = 3$

Planning

Memory

- Small memory of old plans
- Useful past plans are stored for each layer [5] (*memoization* or *tabling* [11])
- Each plan contains a sequence of actions and some *circumstances* (time, location, NPC resources)

Planning

Memory

- In case of matching circumstances, the plan is considered for re-activation
- Results of the plan are then estimated from the current configuration
- If the result achieves the current goal then the plan is used
- When a plan is used, then its score is increased
- Score determines removal/permanence of plan in memory

Memory

```
recall_plan(world,agent) =
  for plan in agent.memory do
    if similar(plan.world,world) ∧
       reaches_goal(plan) then
       increase_score(plan)
       return plan
  return null
```

Planning

Personalized NPCs

- Our framework allows the differentiation of NPCs
- Different NPCs make plan based on their preferences

Planning

Personalized NPCs

- NPCs now favour certain plans
- Still, restrict to valid plans
- Also, priority given by heuristic selection depends on preferences
- For example, an NPC that favors magic will expand magic fighting actions

Execution

Plan execution

- After the planning phase plans are actually put into action
- While planning, NPC's read and modify temporary copies of the game world state
- The game world itself is never affected by planning

Execution

Sequentialization

- The simplest scheme just alternates planning and execution
- Planning happens in a single tick of the simulation
- After planning, straightforward execution executes the plan actions in order

Sequentialization

```
while true do
  plan = find_plan(world,self)
  execute_plan(world,self,plan)
```

Sequentialization

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    execute_action(world,self,action)
```

Execution

Plan execution

- We are executing plans in real-time
- We must ensure that an executed plan remains sensible
- Cannot predict how lower level plans will achieve goals, other agents, etc.

Execution

Non-determinism

- Long-term planning, non-determinism, or layering cause wrong expectations
- Planning errors may be benign (a plan is yielding higher benefits than expected)
- Planning errors may be malign (a plan is costing more than anticipated)
- Negative effects may add up to the point of an NPC death
- Checks to determine if an action can be performed safely

Non-determinism

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    if action.cost < agent.resources then
      execute_action(world,self,action)
    else
      return
```

Execution

Tracking expectations

- We can keep track inside a plan of the expected results of each action
- We can then check so that an NPC keeps running a plan only under the original assumptions

Tracking expectations

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    if action.cost < agent.resources ∧
       matched(action.expectations,world,agent) then
      execute_action(world,self,action)
    else
      return
```

Execution

Tracking expectations

- NPCs can safely try and plan ahead for even multiple days or weeks of simulation time
- Replanning will be triggered before an execution is completed if needed

Execution

Concurrent planning

- Up until now we have assumed that planning is virtually instantaneous
- Many NPCs make planning becomes too slow for a single tick
- We must split the computation of plans across multiple ticks

Execution

Concurrent planning

- NPCs will then factor in the time required for completing the current action into the plan itself
- Suppose that the amount of time that NPCs take for planning is T_{plan}, then the planning algorithm will simulate that T_{plan} seconds have elapsed, and then will try to plan across multiple ticks of the simulation
- If the planner cannot find a plan in T_{plan} seconds, then T_{plan} is increased by a factor K so that the next planning phase will take longer

Concurrent planning

```
while true do
  world',self' = simulate_after(world,self,T_plan)
  plan = find_plan(world',self') | wait(T_plan)
  if plan ≠ null then
    T_plan ← T_plan / K
    execute_plan(world,self,plan)
  else
    T_plan ← T_plan × K
```

Execution

Concurrent planning

- The planner will now suspend periodically
- It will let the rest of the simulation tick and run interactive code

Concurrent planning

```
find_plan_fast(world,agent) =
  Q = {(world,agent)}
  C = 0
  do
    ... (* update Q of explored plans *)
    suspend()
  while Q ≠ ∅ ∧ Steps < N
  return null
```

Execution

Concurrent planning

- Plans may be computed during execution
- Doing so allows us to take even more than the allotted time of T_{plan}
- Use long actions (for example sleep)
- Plan is formulated from the end of the action

SLIDE

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    if is_last(action) then
      world',self' = simulate_after(world,self,action)
      return find_plan(world',self') & execute_action(
          world,self,action)
    else
      execute_action(world,self,action)
```

Results

Virtual city

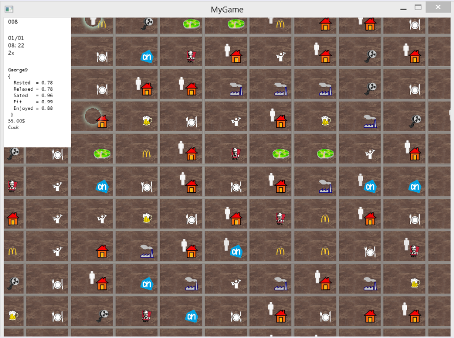

Figure: The first virtual city

Results

Virtual city demo

Results

Preferences

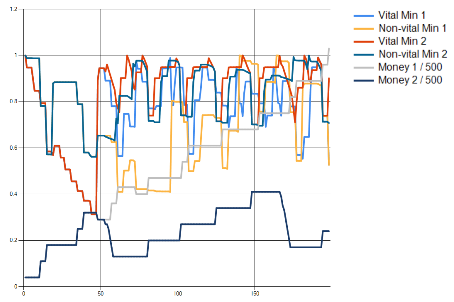

Figure: The resources of two agents: first with the goal of survival plus earning 500 units of money, second with just the goal of survival

Results

Preferences

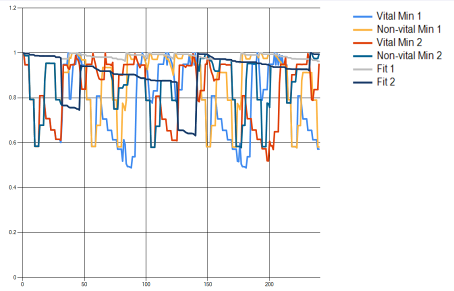

Figure: The resources of two agents: first is a fitness lover, second is a regular NPC

Results

Long-term survival

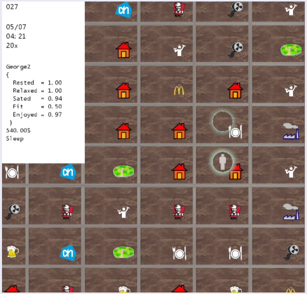

Figure: Six months survival

Results

Scalability

Num agents	City size (sq. km)	FPS	Speedup (min. / sec.)
100	6×6	30	10
150	6×6	30	5
200	6×6	30	5
250	7.5×7.5	30	5
300	9×9	15	2

Table: Planner performance

Conclusions

Problem

- Use of a layered planning technique in order to create non-playing-characters (NPCs) that *live* in a game world
- Greatly increases the believability of the game world, and gives additional depth to the game

Conclusions

Idea

- Planner inspired from forward-chaining constraint logic programming
- Aggressive heuristics for steering
- Aggressive pruning of less promising partial plans
- Memoization to recycle old but effective plans
- Using such a system in real-time poses additional challenges of reliability and concurrency

Conclusions

Results

- Our technique offers multiple positive features
- Our layered planner is *fast* and **effective**
- It guides hundreds of NPCs to survival in challenging, real-time scenarios for long periods of time
- NPC survive indefinitely

Conclusions

Results

- **General-purpose** planner
  - Parametrized set of actions
  - Parametrized environment
- The behaviors of our NPCs reach short-, medium-, and long term *goals*
- NPCs are *customizable*

That's it

Thank you!

References I

Blai Bonet and Hèctor Geffner.
Planning as heuristic search.
Artificial Intelligence, 129:5–33, 2001.

T. Frühwirth and S. Abdennadher.
Essentials of Constraint Programming.
Cognitive Technologies. Springer, 2003.

Patrik Haslum and Hèctor Geffner.
Heuristic planning with time and resources, 2001.

References II

Paul Hudak.
Conception, evolution, and application of functional programming languages.
ACM Comput. Surv., 21(3):359–411, September 1989.

Mark Johnson.
Memoization in constraint logic programming.
In *Proc. Intl. Workshop on Principles and Practice of Constraint Programming*, page manuscript., 1993.

J.Orkin.
Three states and a plan: The ai of f.e.a.r.
Proceedings of the Game Developer's Conference (GDC), 2006.

References III

Richard E. Korf.
Depth-first iterative-deepening: An optimal admissible tree search.
Artificial Intelligence, 27:97–109, 1985.

S.J. Russell and P. Norvig.
Artificial Intelligence: A Modern Approach.
Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010.

Stuart J. Russell and Peter Norvig.
Artificial Intelligence: A Modern Approach.
Pearson Education, 2003.

References IV

Earl D. Sacerdott.
Planning in a hierarchy of abstraction spaces.
In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 412–422, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

Neng-Fa Zhou and Taisuke Sato.
Efficient fixpoint computation in linear tabling.
In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, PPDP '03, pages 275–283, New York, NY, USA, 2003. ACM.