Spice it up! Enriching Open World NPC Simulation Using Constraint Satisfaction

Martin Černý and **Cyril Brom** and **Roman Barták**
Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, Prague 1, Czech Republic
cerny.m@gmail.com, {brom,bartak}@mff.cuni.cz

Martin Antoš
Warhorse Studios
Pernerova 53, Prague, Czech Republic
martin.antos@warhorsestudios.cz

Abstract

With more computing power available, video games may spare increasing amounts of processing time for AI. One prospective application of the newly available resources is the simulation of large amounts of non-player characters (NPCs) in open world games. While it is relatively easy to simulate simple behaviours of individual NPCs it is much more difficult to create meaningful interactions between the NPCs. However, without interaction, the world cannot look very alive. In this paper we present a technique that enriches the NPC simulation with pre-scripted *situations* — short sketches involving coordinated interaction between several NPCs that do not substantially alter the state of the game world but increase the appeal of the world to the player. We use constraint satisfaction techniques to find NPCs suitable to enact the situations at runtime. We have implemented situations on top of the AI system for an upcoming AAA open-world game and show that this approach satisfies functional and computational requirements for practical deployment in the final version of the game.

Crowd simulations are a well-studied subject from the point of view of academic AI and its applications as an aid to make decisions about complex system in urban planning, computational biology and other areas. It is also an interesting inspiration for creating believable worlds for video games with many non-player characters (NPCs). Examples of recent games with large crowds are the Dead Rising and Assasin's Creed series (Capcom 2013; Ubisoft 2013).

The two aforementioned games, along with most others, build on simple crowd simulation approaches: they feature large amounts of NPCs with very basic decision making — NPCs simply select one of few prescripted behaviours (walk, approach player, flee from player, etc.). Importantly, the NPCs are autonomous and independent, although coordination may emerge from the individual behaviours.

Coordinated behaviour on a low level emerges easily from simple individual behaviours (e.g. flocking), but this is not the case for higher lever coordination which usually requires complex individual decision making. In addition, emergent phenomena are hard to debug and control. However games, unlike many other crowd simulation applications, are not required to be plausible models of the real world and thus may replace autonomy by centralized control.

Recently we had the opportunity to work on an AI system for an AAA open-world game currently under development. Since the game should fully simulate hundreds of NPCs with their daily routines and needs, the game essentially features an interactive crowd simulation. In our previous work, we have created *Smart Areas* (Cerny et al. 2014) — parts of the virtual space that carry behavioural information for NPCs that enter them. For example, instead of NPCs knowing how to behave in a pub, it is the pub that knows how NPCs should behave inside it. This promotes easy extensibility of the environment with new behaviours and it eases development: the communication protocol between the innkeeper and the guests, the management of free seats and other data structures is kept at one place in the code. This allowed us to distribute and divide the complexity of the behaviours into clearly separated units.

Still, the NPCs spent a lot of time "in transit" between the various Smart Areas and while the streets and roads of our virtual world were not empty, they lacked interaction between the NPCs who simply walked past each other. We aimed to increase the appeal of the simulated world and make it appear more "alive". Literature on crowd simulation provides inspiration for good implementation of low-level behaviours and interaction of the NPCs (gaze control, collision avoidance, . . .), but there are fewer guidelines how to introduce higher level interaction (e.g., small talk, petty crime). At the same time, design and computing power restrictions severely limit the acceptable complexity of NPCs.

To enrich the simulation of our world without the need to increase complexity of the individual NPCs, we have created what we call *situations*. A situation is a short scripted scene that involves multiple interacting NPCs taking individual *roles* in the situation. Each role imposes requirements for an NPC to be allowed to take it. For example a situation called "Beggar", where a rich man shows contempt for a poor beggar and gives him some money has two roles: the beggar and the benefactor. Only poor NPCs may take the role of the beggar and only rich ones the role of the benefactor. Moreover the two NPCs must be close to each other to enact the situation promptly and safely.

In our scenario the goal is to select 2-5 participants meeting various constraints from a pool of several dozen

NPCs, preferably choosing a different solution every time the search is performed. Due to combinatorial explosion, a brute-force approach cannot satisfy runtime requirements. Instead we describe the requirements as a constraint satisfaction problem (CSP) (Dechter 2003) and use CSP solving algorithms to find suitable NPCs quickly.

In this paper we describe a prototype of our situation system. We report on a case study of several situations we have implemented to demonstrate the viability of our system for the game. We discuss qualitative feedback we got from game designers and scripters and quantitative measurements of computation requirements of the system.

The rest of the paper is organized as follows: First, related work is reviewed then requirements we had to fulfil are given, followed by an overview of our system. Then CSPs in general and our CSP implementation are discussed and we introduce a case study and quantitative evaluation.

Related Work

(Olivier, Dickinson, and Duckett 2011) discuss the role of crowds in games, especially in open-world games. They outline a group-based approach to introduce more complex social dynamics to a crowd but they have not implemented it.

The simulation of (Shao and Terzopoulos 2007) features autonomous pedestrians in a virtual railway station. Several social behaviours (e.g., buying tickets, spectating an art show) with coordination (e.g., queue at the ticket booth) mediated by specialized environment objects are introduced. However, characters do not directly interact with each other and every character must be explicitly prepared for all the social behaviours it may perform, limiting scalability.

(Pedica and Vilhjálmsson 2010) pioneered the area of social interactions in virtual crowds. However, their work focuses on relatively low-level social behaviour (attention, gaze and positioning) rather than on higher-level behaviours.

In the CAROSA framework (Li and Allbeck 2011) interactions between characters may emerge from agent-centric decision making. Although little detail is provided, the main mechanism seems to involve characters switching behaviours in response to their needs and in response to another character with a given need nearby. This way the interaction code has to be split among several parts of the system — the code that triggers the need for interaction, the code that ensures proper role switching in the other character once the initiator approaches and the code for the interacting roles.

Similarly to our approach, (Stocker et al. 2010) introduced "Smart Events" — specific objects providing NPCs with ready-made responses to external events. An important difference is that Smart Events influence area of the virtual world and provide behaviours for all types of characters without any explicit coordination among the characters while our situations provide behaviours for carefully selected NPCs and focus on coordination of the NPCs.

Also in other contexts, it has been noted that complex short-time events are hard to generate and work better when scripted (Shoulson et al. 2013).

In The Sims: Medieval, there is a central entity that selects individual NPCs to perform a specific role in the game based on simple constraints (Graham 2011). However, tuples of NPCs are not searched for.

In Hitman: Absolution, AI uses "situations" in a different meaning than we do (Vehkala 2012). Whenever an NPC deals with an event that requires coordination with other NPCs (e.g. the player is trespassing and should be stopped), it subscribes to a corresponding *situation object*. The situation object maintains a list of subscribed NPCs and internally assigns roles to them (in most cases just "leader" and "other"). The situation object receives updates from the game world and alters the knowledge of the subscribed NPCs (e.g. tells the NPC that it is in a trespassing situation, who is the leader of the situation and how aggressively should the NPC react). The NPCs than take that knowledge into account in their own decision making. This way, every NPC needs to include specific code for every situation it may participate in. Furthermore, the code for the situation is scattered among multiple NPCs and all variants of the situation code have to be considered if an NPC needs to coordinate its behaviour with actions of the other participants. Our approach has the advantage of decoupling situations from the rest of the AI and it allows for tighter coordination among NPCs at the cost of fixing the number of NPCs for a situation.

The only CSP application in computer games we are aware of is in procedural generation of game content. In (Horswill and Foged 2012) a map in a game is populated with items and enemies to fulfil various constraints. The paper also reviews other applications of CSP for procedural content generation.

Requirements and Problem Analysis

Since the goal of our situation system is "only" to increase appeal of the world to the player, it must not interfere with gameplay-critical mechanics and designers must retain tight control over it. Further, the situation code should be well encapsulated so that it is possible to develop situations almost independently of other AI code. The situation code must also be able to make some assumptions about the NPCs taking the roles — this is why constraints were introduced.

The downside of this central approach is the necessity to search for suitable sets of NPCs. Computing time is scarce in video games, so although situations are scheduled at relatively long intervals (several situations per minute at maximum), an individual search must not take more than 1 ms in the worst case and the average should be lower than 100 μs.

While our world should feature hundreds of NPCs, there are multiple considerations that will reduce the number of NPCs involved in search and thus let us meet the runtime requirements. Most importantly, situations are, in our case, only "eye candy" so there is no need to consider NPCs that are too far from the player and cannot become visible to the player for the duration of the situation. Next, most of our situations make sense only in certain areas (e.g., the "Beggar" situation is tied to a city) and thus only NPCs that are present in the area may be considered. Finally, some of the NPCs will not be subscribed for situations because they perform a more important behaviour. This way we will never search more than a few dozen candidate NPCs.

System Overview

The situation system is built over our AI system as described in (Plch et al. 2014) which is deployed in an upcoming AAA RPG game. The AI system uses an advanced variant of behaviour trees (Champandard 2007) and supports inter-NPC messaging and seamless behaviour switching.

The central component of the situation system is the situation manager, which decides what situations should be enacted, when they should start and which NPCs should participate. Situation scheduling is driven by designer-specified minimal and maximal intervals between successive executions. Once a situation is scheduled for execution, the manager searches for suitable NPCs based on constraints associated with the situation. As in the "Beggar" example, there are unary constraints restricting the holders of individual roles by various traits of the NPCs (abilities, occupation, social status etc.) and n-ary constraints such as maximal distance between the NPCs or requirement for visual contact between the NPCs. Furthermore, each NPC keeps track of situations it has been involved in recently and thus designer-specified intervals between successive executions of situations with the same NPC can be enforced.

To promote designer control and to make sure gameplay-critical behaviours are not interrupted, NPCs must explicitly subscribe to the situation manager. Once NPCs are given a role in the situation, they suspend their current behaviour and are assigned a behaviour specified by the role. This eases the coordination of the NPCs, because all role holders are fully controlled by the situation. E.g., in a brawl scenario participants do not need to negotiate the fighting animations they should perform and may use explicitly defined shared semaphore to synchronize the animations.

Transitions between behaviours, including situations, are managed by the AI system and are detailed in (Plch et al. 2014). If any of the NPCs switches to a higher-priority behaviour (e.g. combat), the behaviour given by the situation is terminated and after an optional quick cleanup behaviour the NPC starts the higher priority behaviour. If this happens, all other role holders in the situation are notified and terminate their behaviours as well. The AI system is responsible for selecting the behaviour that should be run once the NPC leaves the situation. In most cases the NPC simply resumes the behaviour it performed before joining the situation.

The search for NPCs suitable for the situation is modelled as a constraint satisfaction problem (CSP). Using CSPs has let us to take benefit of the large amount of previous work on the topic and quickly implement a solver that meets the strict runtime requirements of a computer game.

CSP Overview

A Constraint Satisfaction Problem (CSP) (Dechter 2003) consists of a finite set of variables, roles in our case, where each variable has a finite set of possible values, NPCs in our case, called a domain. The values that can be assigned to variables are restricted by constraints, where each constraint is defined over a subset of variables and implicitly defines a subset of the Cartesian product of variables' domains (allowed tuples). A binary constraint can for example express the maximal distance between two roles and a unary constraint may express required NPC properties for a given role. A solution to a CSP is an instantiation of all the variables satisfying all the constraints.

CSP is NP-complete so the search is exponential in the worst case, but we expect the search to be relatively "easy" — situations should be designed in such a way, that most of the time there are multiple combinations of NPCs that satisfy the constraints.

The mainstream approach to solve CSPs is based on a combination of *backtracking search* and *inference*. Backtracking search repeatedly selects a variable for instantiation and then selects a value to be assigned to that variable. After each variable instantiation, constraints are tested against the partial solution. If the constraints fail, the search *backtracks*: tries another value for the current variable. If there are no more values for the current variable, the backtrack returns to the variable assigned previously. The search terminates once all variables have been instantiated (success) or when there are no more values to try for the first variable (failure).

Inference techniques reduce search space by propagating the partial instantiation to other variables. One of the basic inference techniques is *forward checking*. It means that values violating any constraint (with the currently instantiated variables) are removed from domains of not-yet instantiated variables. For example, if we assign an NPC to a given role and there is a maximal-distance constraint for another role then all NPCs that are too far are removed from that role's domain. If any domain becomes empty then we backtrack immediately. Upon backtrack the domains of unassigned variables are restored to the state prior to the instantiation.

Since unary constraints are independent of assignment of other variables, values that violate them may be removed from the domains prior to search. This is called *node consistency* and it is actually a weaker form of forward checking, where only unary constraints are considered. There are more advanced inference techniques such as *arc consistency* where all the constraints between unassigned variables are taken into account. Such techniques are useful for hard combinatorial problems, which is not our case.

Our CSP Solver

We have implemented two solvers: a *backtracking solver* for general use based on the techniques described in previous section and a *local simplified solver* for situations featuring various forms of short interactions between NPCs passing each other (greeting, waving, etc.). All of those situations have exactly two roles and are basically reactions to external stimuli rather than scheduled events. It would be impractical and wasteful to use a backtracking CSP solver considering all NPC pairs in this scenario. The solver used for a particular situation is chosen by the designer.

In addition to explicit constraints given by the designers, an implicit all-different constraint (the same NPC cannot take more than one role) is encoded in the solver algorithms. The solvers are implemented in C++.

Local Simplified Solver

The local solver is run within the update of an NPC's AI. It asks the situation manager for a locally searched situation that is scheduled for execution. If there is such a situation and if the NPC meets the unary conditions for the first role, it searches the NPCs with direct visual contact for a suitable candidate for the second role (the AI system already maintains a visibility list for each NPC for other purposes, so this is computationally cheap). If no situation is available or no solution is found a minimal interval is enforced before another search is attempted.

The greetings and similar situations could be as well implemented with a preprogrammed reactive layer in the AI architecture. But involving the situation system gives more control to the scripters and thus reduces the workload of programmers. More importantly, describing all of the situations in a unified way fosters fast development as there is only one learning curve the scripters need to overcome and programmers have written only single editor support code.

Backtracking Solver

For most situations we use a CSP solver based on backtracking. The solver is invoked directly by the situation manager whenever a situation is scheduled and is updated independently of the individual NPCs. Since our CSPs are small, it did not make sense to implement a sophisticated CSP solver with complex inference — the overhead would easily cancel out the slight gain in search performance. Instead, we were adding inference techniques one by one to see, when the performance stops improving.

We started with the basic backtracking algorithm. Since variety is important, the solver should not return the same solution upon repeated execution with the same data (if there are multiple solutions). Thus the instantiation at each level starts at a random element of the domain.

Then we added node consistency (NC), i.e. all NPCs that did not match the unary conditions were removed from the domains before the search. Surprisingly initial results showed this approach to be often worse than plain backtracking. Closer analysis revealed that on the smaller domains the time spent in evaluating all the unary constraints dominated the time in the actual search by factor of two to ten. Thus we introduced lazy NC evaluation: the unary constraints are tested only on the values that are actually searched, but if the condition is not met, the value is removed from the domain permanently, i.e. it is not reinserted upon backtrack. While lazy approaches to consistency are known in the literature (Schiex et al. 1996) we are not aware of any applications to node consistency.

Last we added forward checking (FC). With a straightforward implementation where contents of domains (arrays of NPCs) were simply copied prior to instantiation of variables and restored upon backtrack, the algorithm fared worse than lazy NC only. An implementation trick was needed to make the algorithm competitive: Pruned values were not removed from the domains, but kept at the beginning of the domain and an internal pointer to the first domain element that was not pruned was kept. This way, only the pointer needed to be changed on backtrack and still permanent removal of values from the domain due to lazy NC was possible in constant time (copying the last element over the removed element).

We did not implement arc consistency (AC), because it requires keeping explicit track of pairs of values in unassigned variables. Since NC was already slow and allocating and updating linear amount of space was troublesome for FC, it was not likely that working with quadratic amounts of data could improve performance.

To summarize, we tested five variants of backtracking solver — plain, with eager NC, with lazy NC, with eager NC and FC, with lazy NC and FC.

Case Study

The development team includes 6 scripters and 6 designers. All of the scripters have limited programming experience, while designers have close to none. Except for one scripter with mathematical modelling background, both designers and scripters had no experience in CSP or similar formalisms. Nevertheless, the idea of situations was relatively clear to both designers and scripters.

The designers found it natural to think in terms of situations, although there was some confusion on the capabilities of the system and its intended use. Most often, designers would propose situations that could break if a participant leaves it due to a higher priority event. To give an example: *"NPC falls into a trapping pit and someone comes to help"* — if the rescuer is disturbed, the NPC is stuck in the pit.

Another frequently mentioned issue was the impossibility to choose and constrain non-NPC game entities as a part of the search (e.g. find an NPC and a nearby water source where he could drink). This was left out as future work, once the system is proven suitable in practice.

To field test the situation system, our design team has proposed several situations that we have implemented:

- **Greetings**. Various forms of greetings and waving.
- **Beggar**. A rich NPC passes a beggar and gives him money. The beggar displays gratitude.
- **Payment**. A peasant meets a rich man. The rich one demands money to settle a loan. The peasant is reluctant, but finally pays the requested sum.
- **Small talk**. Two peasants meet on a corner of the street and discuss the weather briefly.
- **Lively argument**. Three peasants meet to argue about trade, taxes and the heir of the throne.
- **Dance**. A musician starts playing music on the street. Four peasants gather around and perform a group dance.
- **Difficult**. A non-natural representation of the "Dance" situation to test a slightly more extreme scenario.

Formally, all of the situations have constraints on the occupation/social class of the participants, and distance/visibility constraints among all pairs of the participants. The "Difficult" situation is an exception as only 4 pairs of NPCs are affected by binary distance constraints. Still the constraint graph is connected and thus there are implicit "transitive" distance constraints over all NPC pairs, but those implicit constraints are inaccessible to the search algorithm and violations of those constraints are discovered only after assigning NPCs to the “intermediary” roles.

The “Greetings” situation is the only one using the local solver. The “Small talk”, “Lively argument” and “Dance” also require all NPCs to be close to one of designer-marked areas suitable for situation enactment. “Payment” has a special condition for the peasant to check whether it makes sense for him to pay a loan. This condition is expressed as a Lua (Ierusalimschy, De Figueiredo, and Celes Filho 1996) script and we expect it to be very costly to call. Except for “Greetings” which is available everywhere, all other situations are tied to a city.

Describing situations in terms of conditions on role holders was a straightforward idea for the scripters, but they had some difficulty to decide what should be modelled by unary constraints and what should be a n-ary constraint. For example, in the “Beggar” situation, the first idea was to not constrain the richness of the benefactor in a unary condition, but create a binary condition “benefactor richer than beggar”, which would likely result in longer search times.

Quantitative Results

We created two scenarios. Scenario 1 models the expected workload and involves 50 NPCs: 2 musicians, 6 rich, 8 beggars and 34 peasants. The NPCs moved around the game world, sometimes performing a non-interruptible behaviour; in effect 28 NPCs were registered for situations in an average search task (standard deviation: 4.7, max: 49, min: 14) Scenario 2 tests a heavier, unrealistic load to check the scalability of the system. It involves a larger world and 300 NPCs, divided in the same proportions as in Scenario 1, with 193 registered for situations on average (standard deviation: 24, max: 270, min: 149). Tests were run on an Intel i5-3470 quad core processor @3.2 GHz, with 8GB RAM. To reduce measurement noise, the backtracking solver was run synchronously with the game engine and the engine was forced into single-threaded execution for the purpose of the test.

Both scenarios were run until at least 200 instances of all globally scheduled situations were executed. During the experiment all situations were run with the same frequency. All five backtracking solver variants solved exactly the same CSP instances[1].

Local Solver

The local solver (used for the “Greetings” situation) is hard to measure precisely as the individual executions are very quick and thus the measurement itself skews the results significantly. It is however safe to say that even in Scenario 2 the individual executions were faster than 1 µs. Since the solver does not need to be run frequently (a 1 Hz tick is still acceptable) it means that it is safe to execute the local solver within the AI updates. Thanks to the distributed nature of the local solver the AI system has tight control over the time allocated to solving — if time is running out, local solving may be skipped or postponed to next frame for some NPCs and the system will degrade gracefully.

[1]The full dataset is available at `http://bit.ly/1mmGBks`

Backtracking Solver

Table 1 shows the ratio of successful searches in both scenarios. Since all the search was never interrupted prematurely, failed searches represent problem instances that had no solution. The table shows that we tested both situations that always had a solution and those where solutions were rare and those in between, although the number of solvable assignments grew with more NPCs available.

	Scenario 1			Scenario 2		
Situation	**Suc.**	**F.**	**%**	**Suc.**	**F.**	**%**
Beggar	215	2	99%	222	0	100%
Payment	101	116	47%	205	25	89%
S. Talk	98	123	44%	157	69	69%
Argument	56	161	26%	172	52	77%
Dance	24	187	11%	93	134	41%
Complicated	45	166	21%	168	60	74%

Table 1: No. of successful (Suc.) and failed (F.) searches.

The timing results are given in Table 2. The most important finding is that LazyNC performed the best for all situations in Scenario 1 and safely fulfilled the required time limits (recall the requirements: 100 µs on average and 1 ms in the worst case). The problem instances are just too small for FC to make a difference, although the LazyNC-FC variant fulfils the time limits as well. Solving “Payment” takes a remarkably long time, because it involves execution of a script to evaluate a unary constraint which is costly. Script constraints are discouraged for production use and should serve as a prototyping tool only. Still LazyNC and LazyNC-FC stayed within performance bounds.

In Scenario 2 LazyNC still performs very well for all the 2-NPC situations. This is not surprising, as FC needs to scan the whole domain of the second NPC with every assignment of the first NPC during inference and then scans the remaining values again during search while LazyNC visits all the domain values at most once. In this regard it is unexpected that LazyNC-FC is better in the “Small Talk” situation. The reason is probably that the binary distance constraint is faster to evaluate than the unary constraint that checks whether the NPC is close to a suitable area, which is worst-case linear in the number of nearby areas (up to 5 in our scenarios). Some optimization of this condition might be useful.

The FC technique performs better also at the 3-NPC “Argument” and 5-NPC “Dance” where finally the domains are large enough for pruning to have notable effect. In contrast with Scenario 1, “Complicated” became much faster to solve than the “Dance” situation on average. This is most likely due to the fact that in Scenario 2, “Complicated” had solutions much more often than “Dance” and successful searches were generally much faster than failed ones.

Combination of LazyNC for 2-NPC situations and LazyNC-FC for multiple NPCs situations satisfies the runtime constraints even upon heavy load, except for the “Dance” situation and “Payment” (which takes long due to the scripted constraint). If the “Dance” situation is not

Scenario 1 - 50 NPCs, ∼28 registered

	Beggar		S. Talk		Payment		Argument		Dance		Complicated	
	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max
Back	5 (10)	125	18 (10)	60	95 (52)	331	30 (22)	123	22 (25)	154	245 (618)	4869
NC	16 (04)	34	19 (05)	43	117 (27)	258	27 (09)	**55**	22 (19)	79	38 (11)	82
LazyNC	**3 (02)**	**12**	**14 (07)**	39	**87 (42)**	**205**	**20 (09)**	60	**17 (11)**	**55**	**21 (08)**	**53**
NC-FC	27 (05)	47	29 (07)	61	161 (37)	329	38 (10)	81	35 (21)	110	51 (12)	92
LazyNC-FC	15 (02)	33	25 (05)	**36**	92 (41)	248	36 (08)	60	42 (08)	80	55 (12)	93

Scenario 2 - 300 NPCs, ∼193 registered

	Beggar		S. Talk		Payment		Argument		Dance		Complicated	
	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max	Avg (sd)	Max
Back	6 (03)	19	95 (92)	594	539 (396)	1816	213 (242)	1239	435 (700)	5557	14ms (38ms)	273ms
NC	130 (18)	208	158 (20)	232	875 (173)	1702	235 (37)	415	306 (149)	584	366 (66)	584
LazyNC	**4 (03)**	**16**	69 (45)	183	**466 (335)**	1586	119 (73)	366	179 (124)	600	**89 (38)**	**246**
NC-FC	164 (26)	408	196 (27)	286	1104 (208)	1958	271 (41)	423	345 (155)	686	408 (66)	577
LazyNC-FC	31 (04)	54	**52 (29)**	**171**	482 (359)	**1579**	**96 (59)**	**305**	**149 (067)**	**462**	136 (66)	326

Table 2: Aggregate results for search time for individual situations and algorithms. The average (Avg) times, standard deviation (sd) and maximum (Max) are given. Best results in each column are in boldface. Unless explicitly stated, times are in µs.

scheduled frequently, the system as a whole would be fast enough.

Notably, the search for NPCs is on the same time scale as pathfinding — instances in Scenario 1 are comparable to finding shorter paths and those in Scenario 2 are comparable to finding paths across large parts of the game world.

Further analysing the successful and failed runs separately shows some deviations from the average case, but the overall picture remains the same. In general, plain backtracking ranks better in successful runs than on average, but is much worse in failed runs, while eager NC and NC-FC ranks better in unsuccessful ones than on average. This is most likely an intrinsic property of the algorithms: If there are many solutions, backtracking is likely to find one and it avoids the overhead of the other algorithms. Eager NC on the other hand is a pessimistic technique which fails quickly if there are no or very few values satisfying the unary constraints.

Discussion and Future Work

We have introduced a system for enriching a simulated world with short pre-scripted situations. Our algorithm uses CSP to search for appropriate NPCs and to decouple the situations from the rest of the AI code. We have shown that the system meets functional and runtime requirements of a commercial game. The system is currently being evaluated for production use in an upcoming AAA RPG game and the initial response from scripters and designers was positive.

In recent years the advances in hardware allowed classical AI techniques such as planning to become part of mainstream game AI. We show that CSP may as well follow. In our view application areas of CSPs in games are open: CSP makes it possible to decouple various parts of the game and find appropriate connections (NPCs to situation role in our case) at runtime. It also has numerous applications in both offline and online procedural content generation. Instantiating side quest templates or detecting important abstract game events might be among the interesting future applications. CSP also maintains a high level of designer control — undesirable solutions are easy to remove by adding new constraints. We hope that this example will promote the use of CSPs in games, as even simple and fast-to-implement algorithms have satisfactory performance.

Future work includes extending the situation system to search for items, locations and other non-NPC entities in the game world as part of the situation search. Another possibility is to introduce "passive" and "optional" roles. Passive NPCs are constrained in the search, but do not receive behaviour; they serve as a target of actions only (e.g. a situation where an NPC comments on another NPCs work). Optional roles on the other hand receive behaviours, but the search may be successful, even when no NPCs are found for the role. The situation system should also be evaluated by humans to make sure that situations actually increase the appeal of the game world to players.

Acknowledgements

This research is supported by the Czech Science Foundation under the contract P103/10/1287 (GACR), by student grant GA UK No. 559813/2013/A-INF/MFF and partially supported by SVV project number 260 104.

Special thanks belong to Warhorse studios and its director Martin Klíma for making this research possible by their openness to novel approaches and by letting researchers work in close cooperation with the company. Thanks also to Tomáš Plch, Petr Ondráček and Matěj Marko for their work on the underlying AI system and ongoing support.

References

Capcom. 2013. Dead Rising 3. `http://www.xbox.com/en-US/xbox-one/games/dead-rising-3`. Last checked: 2014-05-11.

Cerny, M.; Plch, T.; Marko, M.; Ondracek, P.; and Brom, C. 2014. Smart areas: A modular approach to simulation of daily life in an open world video game. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence*, 703–708.

Champandard, A. 2007. Understanding behavior trees. *AIGameDev.com*. `http://aigamedev.com/open/article/bt-overview/` Last checked 2014-01-05.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Graham, D. 2011. AI development postmortems: Inside The Sims: Medieval. In *Game Developers Conference 2011*. `http://bit.ly/1qMNsYL` Last checked: 2014-07-18.

Horswill, I. D., and Foged, L. 2012. Fast procedural level population with playability constraints. In *Proceedings of The Eighth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 20–25.

Ierusalimschy, R.; De Figueiredo, L. H.; and Celes Filho, W. 1996. Lua - an extensible extension language. *Software: Practice & Experience* 26(6):635–652.

Li, W., and Allbeck, J. M. 2011. Populations with purpose. In *Proceedings of Motion in Games*, 133–144. Springer. LNCS 7060.

Olivier, S.; Dickinson, P.; and Duckett, T. 2011. From individual characters to large crowds: Augmenting the believability of open-world games through exploring social emotion in pedestrian groups. In *Proceedings of DiGRA 2011 Conference: Think Design Play*.

Pedica, C., and Vilhjálmsson, H. H. 2010. Spontaneous avatar behavior for human territoriality. *Applied Artificial Intelligence* 24(6):575–593.

Plch, T.; Marko, M.; Ondracek, P.; Cerny, M.; Gemrot, J.; and Brom, C. 2014. An AI system for large open virtual world. In *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. This proceedings.

Schiex, T.; Régin, J.-C.; Gaspin, C.; and Verfaillie, G. 1996. Lazy arc consistency. In *AAAI/IAAI, Vol. 1*, 216–221.

Shao, W., and Terzopoulos, D. 2007. Autonomous pedestrians. *Graphical models* 69(5):246–274.

Shoulson, A.; Gilbert, M. L.; Kapadia, M.; and Badler, N. I. 2013. An event-centric planning approach for dynamic realtime narrative. In *Proceedings of the 6th International Conference on Motion in Games*, 99–108. ACM.

Stocker, C.; Sun, L.; Huang, P.; Qin, W.; Allbeck, J. M.; and Badler, N. I. 2010. Smart events and primed agents. In *10th International Conference on Intelligent Virtual Agents*, 15–27. Springer. LNCS 6356.

Ubisoft. 2013. Assasin's Creed IV: Black Flag. `http://assassinscreed.ubi.com`. Last checked: 2014-08-07.

Vehkala, M. 2012. Crowds in hitman: Absolution. *AIGameDev.com*. `http://aigamedev.com/ultimate/video/hitmancrowds/` Last checked 2014-08-07.