Complexity in Simplicity: Flexible Agent-Based State Space Exploration

Jacob I. Rasmussen, Gerd Behrmann, and Kim G. Larsen

Department of Computer Science, Aalborg University, Denmark
{illum,behrmann,kgl}@cs.aau.dk

Abstract. In this paper, we describe a new flexible framework for state space exploration based on cooperating agents. The idea is to let various agents with different search patterns explore the state space individually and communicate information about fruitful subpaths of the search tree to each other. That way very complex global search behavior is achieved with very simple local behavior. As an example agent behavior, we propose a novel anytime randomized search strategy called frustration search. The effectiveness of the framework is illustrated in the setting of priced timed automata on a number of case studies.

1 Introduction

Efficient exploration of large state spaces given as graphs is highly relevant in a number of areas, e.g. verification, model checking, planning, scheduling, etc.

For many applications we are interested in placing guarantees on systems. For verification this could be guaranteeing deadlock freedom or guaranteeing optimality in scheduling and planning. Such guarantees often require exhaustive search of the state space and algorithms for doing this are expensive in terms of time and memory usage. The high memory usage is required to keep track of all states that have been explored. Algorithms in this category often have breadth-first characteristics, such as e.g. A*, [12].

However, covering the entire state space is sometimes unnecessary or even infeasible. Many real application domains prefer algorithms to find solutions fast and then gradually improve the solution instead of guaranteeing optimality. Algorithms with such characteristics are called anytime algorithms and include genetic algorithms, [14], simulated annealing, [16], beam-stack search (a complete variant of beam search), [20], tabu search, [9,10], and others.

Other algorithms rely on heuristic information for states such as estimated distance to the goal. Such heuristic algorithms include beam search and best-first search (e.g. A*). Alternately, randomized or stochastic algorithms like Monte Carlo methods can be used when optimality does not have to be guaranteed. For a good introduction to many type of algorithms for optimization purposes, we refer the reader to [15]. For an interesting approach to LTL model checking using Monte Carlo methods, we refer the reader to [11].

When searching for solutions to optimization problems, the famous "no-free-lunch" theorem, [19], states that all optimization algorithms perform

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 231–245, 2007.

indistinguishably when averaged over all optimization problems. The theorem implies the importance of tailoring solutions to different problems as there is no single best algorithm for all problems. However, there is also the important implication that general purpose exploration engines cannot rely on a single search strategy, but need to offer a wide variety algorithms.

One way to let an exploration engine use multiple search algorithms is to run the search algorithms as co-routines in simulated parallelism. This can be a very efficient approach to searching because of the wide range of strong search algorithms that have been published. However, a weak point with this approach is that no algorithm utilizes the strength of the other algorithms, e.g. a depth-first approach could be searching a fruitful part of the state space, but might not find a good solution in reasonable time due to the search strategy, whereas a beam search performed in the same part of the state space might find good solutions immediately. In turn, beam search might never get to explore that part of the state space due to poor heuristics and/or very expensive transitions to reach that part of the state space.

Alternatively, if the algorithms are able to detect fruitful parts[1] of the state space and employ other algorithms to assist in the exploration, very complex search behaviors can be achieved using just simple and well-known algorithms.

This is exactly the approach we advocate in this paper. We propose an agent framework where individual agents use basic search algorithms (e.g. (random) depth-first search, beam search) and execute as co-routines using a given exploration engine. The agents are connected to a pool of tasks where each agent can put new tasks and get tasks. Tasks in this setting are sub-paths of the state space indicating interesting areas to search. This way an agent with a fixed search strategy that detects a potentially interesting part of the state space can put a number of tasks to the task pool and let other agents with different search strategies pick the tasks and aid in the search of the given part of the state space.

We apply our framework in the setting of priced timed automata (PTA), [17,3], an extension of timed automata, [2], that address the optimal reachability problem. PTA have proven useful for a wide range of different search problems such as model checking, [7], and scheduling, [1,5,18]. The diversity of applications and generality of the modelling language of PTA suggests that no single search algorithm is superior and, thus, could benefit from an agent-based approach to search. The framework has been applied to a number of case studies from PTA with promising results.

The rest of the paper is structured as follows: Section 2 describes the agent framework and the constituents hereof. Section 3 proposes a novel search algorithm termed frustration search and describes its incorporation in the agent framework. In Section 4, we describe the instantiation of our framework for priced timed automata and its application to a number of case studies. We finally conclude in Section 5 and indicate directions of future work.

[1] E.g., estimated cost of finding a solution is lower than the current best solution, or parts of the search space where deadlocks a rarely encountered.

2 Agent Framework

In this section, we propose a highly flexible agent-based framework for state space exploration. The framework has been implemented in the setting of priced timed automata, which is evaluated in Section 4.

An overview of the framework is depicted in Fig. 1. Subsequently, we describe the three components of our framework - exploration engine, agents and task store.

Exploration Engine. The framework we propose is constructed to be independent of the type of state space that is being explored. We require a front-end to the state space in terms of an exploration engine that can take states and return their successors plus meta information such as traditional heuristics (e.g. remaining costs) if the state space supports it. We assume that the given exploration engine offers a single interface function `getSuccessors(s)` that, as input, takes a state and, as output, returns a collection of successors.

Task Store. The task store is a pool of tasks available to the agents. The task store offers an interface to access the pool of tasks by means of putting (adding) new tasks to the store and getting (removing) existing tasks from the store. A task is an entry into the state space either in terms of a sub-path from the initial state or simply a reachable state. The task store is considered to have an infinite number of the initial state.

There are several choices involved in managing the store with respect to states that are added and removed. Different design considerations include:

- When agents perform a `get` call, tasks can be removed in e.g. FIFO, LIFO, most promising first, random, or some other order.
- The initial state can be returned only when there are no other tasks in the store or by some probability.
- The task store can be implemented with a fixed size where the oldest tasks are removed when the size limit is reached. This makes sense when searching for optimality since old tasks might relate to parts of the search space that are no longer promising as new better solutions might have been found since the tasks was added to the store.

Agents. The framework defines a fixed number of agents which are co-routines running in simulated parallel interacting with the exploration engine and, indirectly, with each other through the task store. The agents are simply search algorithms employing some search strategy, e.g. variants of depth-first search, beam search, etc. The configuration of agents can be either static (the collection of agents remains unchanged throughout the search) or dynamic (agents might be replaced by other types of agents depending on how they perform). Each agent A_i has a personal configuration C_i necessary to perform the given search strategy. The configuration holds information about which state to explore next and possibly a list of states waiting to be explored.

There are two aspects to bounding the overall memory consumption of the agent framework. First, each agent should have a reasonably bounded memory consumption such as some constant times the largest depth of the state space.

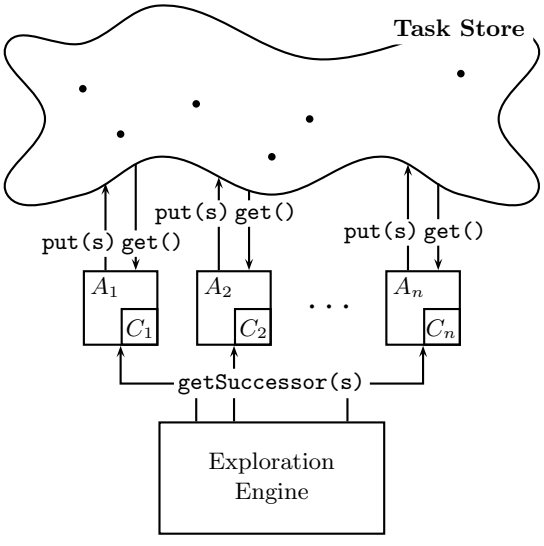

Fig. 1. The three part agent framework consisting of an exploration engine, a set of agents and a task store

This is very applicable for a large number of search algorithms such a different variants of depth-first search (e.g. random, best) and beam search.

Second, there can be no central store of states that have already been explored. This is obviously a trade-off as agents in the framework might explore states that have already been explored by itself or other agents[2]. However, together with the memory limit requirement of the agents, the main benefit is that the search framework can search indefinitely, constantly improving solutions for optimization problems.

Obviously, the behavior of a given agent setup can be described in terms of a single anytime search algorithm, however, that behavior would be inherently complex to describe and very inflexible if changes needed to be made. On the other hand, the agent framework is highly flexible to changes and agents can be added or removed to fit a certain application area.

Furthermore, the agent framework is easily distributed to a multiple processor architecture (in either a cluster, grid, or single PC) by having a number of agents running on each processor sharing a single or multiple task stores.

Note that the agent framework generalizes all of search algorithms, as any one algorithm can be implemented in a framework using a single agent.

3 Frustration Search

In this section we introduce a novel search strategy termed frustration search. Frustration search is an incomplete, randomized anytime algorithm build around

[2] Obviously, for cycle detection, an agent will not explore state already found on its current search path.

random depth-first search and will be discussed in detail and analyzed for time and memory usage. Furthermore, incorporation of the algorithm into the agent framework will be discussed.

Prior to describing the frustration search algorithm, we need to establish some notation to be used in this and the following section.

Preliminaries. We consider state spaces given as a weighted, potentially cyclic, digraph $\langle V, s_0, G, E, \mathsf{Cost}\rangle$, where V is a finite set of vertices, s_0 the root vertex, $G \subseteq V$ the set of goal vertices, $E \subseteq V \times V$ the set of directed edges, and $\mathsf{Cost} : E \to \mathbb{N}$ a weight assignment to edges. As a shorthand notation, we write $s \to s'$ to denote $(s, s') \in E$. The set of all vertices reachable from a vertex s by means of a single transition is denoted by $\mathsf{Succ}(s)$, i.e.,

$$\mathsf{Succ}(s) = \{s' \mid s \to s'\}.$$

A path in a search space $\langle V, s_0, G, E, \mathsf{Cost}\rangle$ is a sequence of states:

$$\sigma = s_1, s_2, ..., s_n$$

such that $s_1 = s_0$ and for $1 \leq i < n$, $s_i \to s_{i+1}$. If $s_i \in V$ appears in σ, we write $s_i \in \sigma$; otherwise $s_i \notin \sigma$. $\mathsf{tail}(\sigma)$ and $\mathsf{head}(\sigma)$ denote the first (leftmost) and last (rightmost) vertex in a path, respectively. The empty path is denoted by ϵ. The binary operator '$\cdot$' denotes concatenation of paths. I.e., for two paths $\sigma_1 = s_i, ...s_j$ and $\sigma_2 = s_k, ..., s_l$, the concatenation $\sigma_1 \cdot \sigma_2$ is given by:

$$\sigma_1 \cdot \sigma_2 = s_i, ..., s_j, s_k, ..., s_l,$$

and is only valid when $(s_j, s_k) \in E$. Furthermore, for any path σ, $\sigma \cdot \epsilon = \epsilon \cdot \sigma = \sigma$.

The cost of the path σ is the sum of costs of the edges, i.e.,

$$\mathsf{Cost}(\sigma) = \sum_{i=1}^{n-1} \mathsf{Cost}(s_i, s_{i+1}).$$

A path $\sigma = s_1, ..., s_n$ is a solution if $s_1 = s_0$ and $s_n \in G$. The optimization problem associated with a state space is to find the solution with minimal cost.

Some heuristic search algorithms assume for every state an under-approximated cost of reaching the goal. We define such a heuristic as a function $rem : V \to \mathbb{N}$ satisfying,

$$\forall s \in V \,.\, \forall \sigma = s, ..., s' \,.\, s' \in G \implies rem(s) \leq \mathsf{Cost}(\sigma). \tag{1}$$

In other words, *rem* is a valid admissible heuristic. Having no such a priori information of the state space corresponds to $rem(s) = 0$ for all states $s \in V$.

Now, we progress to describing the frustration search algorithm that is depicted in Algorithm 1. Frustration search explores different areas of the state space in a randomized fashion identically to random depth-first search, however, the extent to which a given part of the state space is explored depends on the

"attractiveness" of the given part. Unattractive parts of the state space are areas with many deadlocks or states that have *rem* values that cannot improve the current best solution.

Intuitively, frustration search explores the state space while keeping track of its own frustration level. The frustration level increases when encountering deadlocked states or states that can only reach the goal with a cost much higher than current best solution. The frustration level decreases when finding goals with costs close to or better than the current best solution. Whenever the frustration level exceeds a given threshold the current part of the search space is discarded and another part is explored. How much the frustration level decreases depends on how much of the path to the current state is maintained.

The formal structure of frustration search is given in Algorithm 1, which we describe in detail below.

Lines 1-4 initialize the algorithm stating that the best found cost is infinite, the best path is not found, the frustration level is zero (empty path), and the waiting list contains only the path consisting of the initial state. The main (while) loop of the algorithm terminates when the WAITING stack is empty.

At each iteration, a path is selected from WAITING (line 6). How we proceed depends on whether the state at the head of the path is a goal state or not. In case it is, the path can fall in one of three categories compared to the current best solution: Better than, reasonably close to (say, within 10 percent), or far from (say, by more than 10 percent). In the former case, we update the current best solution to the current path, update the best cost and reset the frustration level to zero. Resetting the frustration level guarantees that the current part of the search space is searched more thoroughly. In the middle case, the frustration level is decreased slightly to search the current part of the state space more thoroughly. In the latter case, the frustration level is increased.

In case the head of the path is not a goal state, we need to compute the successors of the state at the head of the path. At line 18, we select only those successors that are neither on the path already nor unable to reach the goal within an acceptable margin of the current best solution. If the computed set of successors is empty, we increase the frustion level accordingly. Otherwise, we add the successors to the WAITING list in random order.

Lines 27 through 31 are executed regardsless of whether the head of the current path is a goal state or not. Here, the frustration level is tested against a predefined frustration threshold. If the threshold has not been exceeded, we do nothing. Otherwise, we compute a random number between zero and the size of WAITING plus one. This value determines the number of states that should be removed from the top of the WAITING list (line 30). Furthermore, the frustration level is decreased proportionally to the number of states that has been removed from WAITING.

Finally, when the WAITING list is empty the algorithm returns to line 4 and reinserts the initial state into WAITING to re-search the state space. The randomization of the algorithm guarantees a diminishingly small chance of two runs being identical.

Algorithm 1. Frustration search

proc FrustSearch ≡

1: COST ← ∞
2: BEST ← ϵ
3: FRUST ← 0
4: WAITING ← {ϵ · s_0}
5: **while** WAITING ≠ ∅ **do**
6: $\sigma \leftarrow$ pop(WAITING)
7: **if** head(σ) ∈ G **then**
8: **if** Cost(σ) < COST **then**
9: COST ← Cost(σ)
10: BEST ← σ
11: FRUST ← 0
12: **else if** Cost(σ) ≤ COST×1.10 **then**
13: FRUST← dec(FRUST)
14: **else**
15: FRUST← inc(FRUST)
16: **end if**
17: **else**
18: SUCC ← $\{s' \mid s' \in \mathsf{Succ}(\mathsf{head}(\sigma)), s' \notin \sigma, \mathsf{Cost}(\sigma \cdot s') + rem(s') \leq \text{COST}\times 1.10\}$
19: **if** SUCC ≠ ∅ **then**
20: **for all** $s' \in$ SUCC **do**
21: push(WAITING, $\sigma \cdot s'$)
22: **end for**
23: **else**
24: FRUST ← Inc(FRUST)
25: **end if**
26: **end if**
27: **if** FRUST ≥ MAXFRUST **then**
28: RAND ← (rand mod |WAITING|+1)
29: FRUST ← FRUST $\times \frac{|\text{WAITING}|-\text{RAND}}{|\text{WAITING}|}$
30: pop(WAITING, RAND)
31: **end if**
32: **end while**
33: **goto** 4

MAXFRUST and how much the frustration level is incremented or decremented can be adjusted to specific applications, e.g., how often are deadlocked states expected or how often should the algorithm start over etc.

Since frustration search restarts after each termination, it only makes sense to talk about time and memory usage for each iteration (the while-loop). The worst case time behavior of frustration search ocurs when no states are ever removed from the WAITING list due to frustration. In this case, the behavior is identical to random depth-first search. Thus, the worst-case execution time is $\mathcal{O}(|V|)$ as with random depth-first search. The memory usage depends on the size of the WAITING list and will - like depth-first search - never exceed $\mathcal{O}(MaxDepth \cdot$

MaxOut), where *MaxDepth* is the maximum depth of any path of the state space[3] and *MaxOut* is the maximum number of successors of any state.

Variants of Frustration Search. The behavior of the frustration search algorithm is easily changed into different variants by changing the order in which states are added to the WAITING list. Using either the *rem* heuristic or other user defined guides, frustration search can be tailored to different kinds of problems. E.g., in Section 4, we use a variant of frustration search that sorts the successors according to the current cost plus the remaining estimate. That way, the states are searched in a best first manner with randomization as a tiebreaker. Variants of this type have no impact on the worst-case time or memory usage.

Relating to the Agent Framework. To implement frustration search in the agent framework defined in Section 2, the state inserted into the WAITING list can be taken from the task store instead of the initial state. Furthermore, tasks can be added to the task store whenever a promising goal location is found. Subpaths of this solution can be added to the task store for further investigation by other agents.

4 Framework Instantiation

In this section, we discuss ways of utilizing the agent-based architecture in scheduling using priced timed automata. We explore only static agent setups and leave dynamic setups as future work.

Timed Automata. Timed automata were first introduced by Alur and Dill in [2] as a model for describing reactive real-time systems. The benefit of using timed automata over the more expressive hybrid automata is that the model checking problem for timed automata - unlike for hybrid automata - is decidable. Decidability is achieved because the infinite state space for timed automata has a finite quotient. The finite quotient allows algorithms to reason about sets of states (a symbolic state) with equivalent behavior.

Because timed automata analysis requires representing sets of states, a symbolic state of most timed automata exploration engines has the form (l, Z) where l is discrete state information and Z is a convex polyhedron representing dense timing information. For priced timed automata (PTA), the state representation is similar except that we associate an affine cost function with the polyhedron. In minimum-cost reachability analysis we need to compute the minimum of the cost function using linear programming.

A symbolic A* algorithm for minimum-cost reachability of priced timed automata has been implemented as an extension to the symbolic state space exploration engine of the real-time verification tool UPPAAL. The result is the optimization tool UPPAAL CORA, which has been successfully applied to a number of benchmarks and industrial scheduling problems, [6]. This A* algorithm is depicted in Algorithm 2.

[3] A path ends whenever it encounters a state already on the path or has no successors.

Algorithm 2. Reachability algorithms used by UPPAAL CORA

```
1: WAITING = {ε · s_0}
2: PASSED = ∅
3: COST = ∞
4: while WAITING ≠ ∅ do
5:    σ ← pop(WAITING)
6:    if head(σ) ∈ G then
7:       if Cost (σ) < COST then
8:          COST = Cost (σ)
9:       end if
10:   else
11:      SUCC ← {s' | s' ∈ Succ(head(σ)), s' ∉ PASSED, s' ∉ WAITING, Cost (σ · s')
                  +rem(s') < COST}
12:      for all s ∈ SUCC do
13:         push(WAITING, σ · s)
14:         sort(MinCostRem, WAITING)
15:      end for
16:      add(PASSED, σ)
17:   end if
18: end while
19: return COST
```

The algorithm is a classical A^* search algorithm with a WAITING list sorted according to the cost plus remaining estimate for each state. The algorithm further keeps a PASSED list of states that have been explored. The successor computation on line 11 involves manipulation of symbolic states that are up to cubic in the number of variables used in Z. Furthermore, inclusion checking and computation of symbolic state costs involve solving linear programs.

Given the computational complexity of manipulating symbolic states, most of the work done for optimizations of (priced) timed automata involves finding better representations of Z assuming that the algorithm is fixed. In this section, we focus solely on the algorithm to explore whether an incomplete search framework like that agent framework is competitive to existing methods.

4.1 Applications

The implementation of the agent framework applied in this section uses the following search agents, referred to as 'Mix':

- *Depth-first search* (DFS): Deterministic search where states are added to the waiting list in the order provided by the exploration engine.
- *Best depth-first search* (BDFS): A variant of DFS that sorts the successors according to their expected cost (current cost plus remaining estimate). Randomization is used for tie-breaking states with equal cost.
- *Random depth-first search* (RDFS): Successors are shuffled before adding them to the waiting list.
- *Beam search* (BS): Classical beam search with a fixed beam width of 100.

- *Frustration search* (Frust): Successors are shuffled before added to the waiting list. Frustration decrement is set to 0.5 and increments to 1 with a MaxFrust set to 1000.
- *Best frustration search* (BestFrust): Like Frust, but successors are sorted according to their cost plus remaining estimate.

The framework uses one agent of every type and creates ten copies of every job in the task store to increase the chance that every agent has a chance to explore every job.

Lacquer Scheduling. This case studies has been provided by Axxom[4] - a German company that provides scheduling software for the lacquer production industry. The scheduling problem involves fulfilling a number of orders (with deadlines) for lacquer of different colors. The lacquer production process requires the use of a number of different machines, which have to be cleaned in-between usage for lacquer of different colors. Each type of lacquer has a special recipe describing which machines to use. Costs are incurred during machine use, cleaning and storage (if recipes are fulfilled before the deadline). The Axxom case has been studied for schedulability with timed automata in [5] and for optimization in [13]. For a thorough introduction to the Axxom case study, we refer the reader to either of the two papers.

The lacquer production scheduling problem is of a size that inhibits exploration of the entire state space, and schedules used rely on suboptimal solutions. Axxom uses custom-made software for scheduling, but [13] reports that the solution found with Uppaal Cora are comparable to those of the custom scheduling software. Thus, every advancement of the solutions found by Uppaal Cora only make the PTA approach more competitive.

The purpose of the following experiments is twofold: First, to compare cooperating agents in a scheduling framework where guaranteeing optimality is unrealistic to other single search algorithm methods. Second, to determine how well the heuristic used in frustration search for skipping parts of the state space is to an uninformed approach. The algorithm used in [13] is ideal for such a comparison as the algorithm is a best depth-first algorithm with (random) restarting (BDFS-RR) based on a stochastic choice. We will test two agent setups with a single agent using the default variant of frustration search and a single agent using the best frustration search approach. The test setup we have used is the following:

- *Hardware:* 3.2GHz PC with 4GB RAM running Linux/Debian 3.1.
- *Models:* We have chosen the two models used in [13]. Both models have 29 orders, but the first model (6) has no costs associate with storage of lacquer whereas the second model (7) has. Model 6 has two variants depending on how machine availability is modelled. Model 6a models the availability as a fraction of the overall times where model 6b models availability according

[4] The case study was provided as a test bed for algorithms in the European Community Project IST-2001-35304 Ametist.

to the work shifts provided by Axxom. The models use the guiding *rem* estimates defined in [13].

- *Instances:* For each variant of model 6, we vary the number of orders that can be active simultaneously. This heuristic was used in [13] where the limit was set to five orders, but we include 15 and 29 as well. Model 7 is the most accurate model of the case study, and we use no variant hereof even though the model is reported to be very difficult in [13].
- *Algorithms:* We use four different algorithms in our test setup: BDFS-RR, Frust, BestFrust, and Mix.
- *Duration:* For model 6, we run experiments for 10 minutes each in accordance with the tests of [13]. In that paper, problems were reported for executing model 7 for longer than 10 seconds, however, we have not experienced such problems and have performed test for 10 minutes and 2 minutes to investigate how fast solutions are found, and how much they are improved over time.
- *Repetition:* Each test is executed 10 times for every algorithm.

For the experiments with model 6(a and b) in Table 1, Mix is clearly the best algorithm for finding schedules, it is only outperformed once by BestFrust on model 6b with 15 active orders. For all other cases Mix is superior both for the best solution, worst solution and average solution. BestFrust is clearly better than BDFS-RR for all instances supporting the idea of using a more informed heuristics for skipping parts of the state space. For all instances, but one, BestFrust also outperforms Frust showing that some guiding is important for model 6. This is further supported by the fact that Frust only outperforms BDFS-RR on two instances.

Even though Mix is the superior approach, the experiments suggests that the benefits of having a mix of agents to a single BestFrust agent are negligible, but the following experiments show that this is not the case.

Table 1. Results for two versions of Axxom model 6 from [13] showing the costs of the best solutions found within 10 minutes of search

Axxom model 6a												
Agent	Mix			BestFrust			Frust			BDFS-RR		
Active orders:	29	15	5	29	15	5	29	15	5	29	15	5
Best (10^6)	2.08	1.98	1.73	2.28	2.36	1.81	3.16	2.10	2.46	-	2.61	2.03
Worst (10^6)	4.91	2.59	2.07	6.09	2.69	2.12	11.77	2.47	6.02	-	3.89	11.1
Average (10^6)	**2.89**	**2.18**	**1.90**	**3.76**	**2.54**	**1.97**	**8.61**	**2.26**	**4.59**	-	**2.91**	**4.33**

Axxom model 6b												
Agent	Mix			BestFrust			Frust			BDFS-RR		
Active orders:	29	15	5	29	15	5	29	15	5	29	15	5
Best (10^6)	6.97	7.03	6.46	7.25	6.87	6.98	7.61	8.02	7.52	7.18	7.44	7.21
Worst (10^6)	7.88	8.56	7.82	7.75	7.73	7.59	8.91	8.85	8.82	8.32	8.55	9.8
Average (10^6)	**7.41**	**7.58**	**7.34**	**7.50**	**7.37**	**7.46**	**8.33**	**8.31**	**8.11**	**7.86**	**7.93**	**8.2**

Table 2. Results for Axxom model 7 from [13] showing the best costs after 10 and 2 minutes of search

Axxom model 7								
Agent Setup	Mix		BestFrust		Frust		BDFS-RR	
Time	10min	2min	10min	2min	10min	2min	10min	2min
Best (10^6)	2.09	2.11	10.21	11.6	4.21	2.44	69.85	64.87
Worst (10^6)	6.32	14.93	17.50	25.21	10.01	15.37	87.83	94.49
Average (10^6)	**3.12**	**6.55**	**13.36**	**18.77**	**8.15**	**8.60**	**79.60**	**88.10**
Found solution	100%	100%	100%	100%	100%	100%	50%	30%

The experiments for model 7 in Table 2 show that the algorithm used in [13] was unable to find even a single solution for a significant fraction of the instances, whereas all agent setups found solution for all experiments. The solutions BDFS-RR actually found are significantly inferior to any solution found by the agent setup. For this model, Frust clearly outperforms BestFrust for both best, worst and average solutions. However, neither algorithm alone is competitive to the cooperating agent framework, which consequently finds the best solutions.

All of the experiments above support that using the agent framework for search has significant benefits for general purpose search. Furthermore, variying the search intensity in different areas of the search space with frustration search seems very fruitful.

Aircraft Landing Problem. The aircraft landing problem involves scheduling landing times for a number of aircraft onto a fixed number of runways. Each aircraft has an earliest, target, and latest landing time given by physical constraints on aircraft speed and fuel capacity. Costs are incurred for each plane deviating from the target landing time. For more information see [4,17,8].

The aircraft landing problem was first discussed in [4] where a mixed integer linear programming solution was given. In [17], the problem was solved using priced timed automata and the results obtained were highly competitive to those of [4]. The case was reused in [18] with optimizations of the algorithm used for minimum-cost reachability for PTA, and with the optimized algorithm PTA were faster than [4] at solving the case for almost every instance.

The purpose of the following experiment is to determine how efficient the agent framework (and frustration search) is at finding optimal schedules in state spaces that are small enough to be searched exhaustively. And further, how well the performance competes with a powerful complete search strategy, A*. The test setup we have used is the following:

- *Hardware:* 3.2GHz PC with 4GB RAM running Linux/Debian 3.1.
- *Models:* We have chosen the seven models used in [4,17,8]. The model uses no form of guiding with *rem* estimates.
- *Instances:* The instances for the models involves varying the number of runways until all planes can land with a total cost overhead of zero.
- *Algorithms:* We use mix of agents described in the beginning of this section together with the default A* search algorithm used in Uppaal Cora.

Table 3. Aircraft landing problem. *: 50% of the tests completed within the time limit of 20 minutes, and the average is computed among these.

Instance			1	2	3	4	5	6	7
Runways	Algorithm								
1	Cora		0.04s	0.17s	0.11s	0.50s	1.10s	0.05s	0.07s
	Agents	Best	0.16s	2.24s	3.77s	6.01s	4.77s	0.05s	6.68s
		Worst	0.22s	3.15s	4.75s	>1200s	8.97s	0.06s	7.70s
		Average	0.18s	2.64s	4.34s	58.01s*	5.88s	0.06s	7.34s
2	Cora		0.15s	0.29s	0.25s	3.56s	4.98s	0.14s	0.35s
	Agents	Best	0.09s	0.28s	5.65s	6.14s	9.69s	2.96s	2.25s
		Worst	0.59s	4.23s	6.12s	7.23s	162.37s	13.87s	13.19s
		Average	0.19s	3.00s	5.83s	6.90s	37.33s	8.86s	4.85s
3	Cora		0.16s	0.22s	0.33s	91.43s	71.95s	0.15s	
	Agents	Best	0.67s	1.66s	7.01s	6.71s	7.12s	0.06s	
		Worst	3.06s	5.97s	7.70s	19.94s	18.43s	0.07s	
		Average	1.94s	5.27s	7.45s	9.44s	9.76s	0.07s	
4	Cora					7.60s	3.14s		
	Agents	Best				9.13s	8.48s		
		Worst				26.48s	20.93s		
		Average				12.59s	11.79s		

– *Duration:* A maximum of 20 minutes were allowed for each instance.
– *Repetition:* The agent setup was executed 10 times and the Uppaal Cora algorithm only once as it is deterministic.

The results in Table 3 clearly indicate that for most instances the A* algorithm outperforms the agent framework. However, it is interesting to note that for the most difficult instances - 4 and 5 - the A* algorithm shows exponential growth in running time until all aircraft can be scheduled with zero cost (4 runways). However, the agent framework does not have this issue, as there appears to be no correlation between the number of runways and the time to find the optimal solution. On instances 4 and 5 with 3 runways, the agent framework clearly outperforms the A* algorithm in finding the optimal solution. On instance 5 with 2 runways, the agent framework performs significantly worse on average, but reasonably for the best case. Only for one instance - 4 with 1 runway - was the agent framework unable to find the optimal solution with the time limit. The optimum was found for 50 percent of the executions, however, all executions - even the ones that never found the optimum within the time limit - found a solution deviating only 1 percent from the optimal within 6 seconds!

Another interesting observation of the agent framework is that for the majority (∼80%) of the executions, the optimum was found by agents searching subpaths created by other agents that found reasonable solutions. In many cases, the beam search agent was able to find a close to optimal solution, but did not find the optimum, and some frustration agent found the optimal solution using

subpaths of the suboptimal solution. This supports the use of interacting agents to achieve complex global search behavior through simple local behavior.

5 Conclusions and Future Work

In this paper, we have investigated using sets of cooperating agents to explore large state spaces. We have tested the agent framework against complete and incomplete single algorithm methods. The results show that for state spaces that are too large to be searched exhaustively, the agent framework consistently finds good solutions that are superior to any single algorithm tested. For smaller search spaces where exhaustive search is possible, the A* algorithm performs better, however, unlike A*, the agent framework does not perform exponentially worse as the state space grows. For the most difficult problems, the agent framework performed significantly better than A*.

We also introduced frustration search as an anytime algorithm for large state spaces. The heuristic to skip parts of the state space based on frustration was shown to be superior to an uninformed stochastic heuristic. Furthermore, for the Axxom case study, frustration search alone proved competitive to the agent framework for a number of instances. Thus, frustration search seems a promising algorithm for general purpose search when exhaustive search is infeasible.

As future work, we need to explore the agent setup for a larger number of cases and compare to other frameworks for general purpose search. Furthermore, a distributed version of the agent framework needs to be implemented to take advantage of multi-processor architectures. Also, we need to investigate a more dynamic strategy for assigning agents to search problem, e.g., by adjusting the number of agents of different kinds by keeping track of how well the agents perform in the given search space.

References

1. Yasmina Abdeddaim, Abdelkarim Kerbaa, and Oded Maler. Task graph scheduling using timed automata. *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.
2. R. Alur and D. Dill. Automata for modelling real-time systems. In *Proc. of Int. Colloquium on Algorithms, Languages and Programming*, volume 443 of *Lectur Notes in Computer Science*, pages 322–335. Springer-Verlag, July 1990.
3. Rajeev Alur, Salvatore La Torre, and George J. Pappas. Optimal paths in weighted timed automata. In *Proc. of Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 49–62. Springer-Verlag, 2001.
4. J. E. Beasley, M. Krishnamoorthy, Y. M. Sharaiha, and D. Abramson. Scheduling aircraft landings - the static case. *Transportation Science*, 34(2):pp. 180–197, 2000.
5. G. Behrmann, E. Brinksma, M. Hendriks, and A. Mader. Scheduling lacquer production by reachability analysis – a case study. In *Workshop on Parallel and Distributed Real-Time Systems 2005*, pages 140–. IEEE Computer Society, 2005.
6. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

7. Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. Model-checking weighted timed automata. In *Proc. of Formal Modelling and Analysis of Timed Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 277–292. Springer-Verlag, 2004.
8. Ansgar Fehnker. *Citius, Vilius, Melius - Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* IPA Dissertation Series, University of Nijmegen, 2002.
9. Fred Glover. Tabu search-part I. *ORSA Jour. on Computing*, 1(3):190–206, 1989.
10. Fred Glover. Tabu search-part II. *ORSA Jour. on Computing*, 2(1):4–32, 1990.
11. R. Grosu and S. A. Smolka. Monte carlo model checking. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 271–286. Springer-Verlag, 2005.
12. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
13. Martijn Hendriks. *Model Checking Timed Automata - Techniques and Applications.* IPA Dissertation Series, University of Nijmegen, 2006.
14. John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.
15. Juraj Hromkovic and Waldyr M. Oliva. *Algorithmics for Hard Problems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
16. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220(4598):671–680, 1983.
17. Kim Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *Proc. of Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 493+. Springer-Verlag, 2001.
18. J. Rasmussen, K. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages pages 220–235. Springer Verlag, 2004.
19. David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM, 1995.
20. Rong Zhou and Eric A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proc. of International Conference on Automated Planning and Scheduling*, pages 90–98. AAAI, 2005.