Proceedings of the Twenty-Fourth Innovative Appications of Artificial Intelligence Conference

QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem

Erich C. Teppan
Universität Klagenfurt
Universitätsstr. 65-67
9020 Klagenfurt, Austria

Gerhard Friedrich
Universität Klagenfurt
Universitätsstr. 65-67
9020 Klagenfurt, Austria

Andreas A. Falkner
Siemens Austria
Siemensstraße 90
1210 Wien, Austria

Abstract

The Partner Units Problem (PUP) constitutes a challenging real-world configuration problem with diverse application domains such as railway safety, security monitoring, electrical engineering, or distributed systems. Although using the latest problem-solving methods including Constraint Programming, SAT Solving, Integer Programming, and Answer Set Programming, current methods fail to generate solutions for mid-sized real-world problems in acceptable time. This paper presents the QuickPup algorithm based on backtrack search combined with smart variable orderings and restarts. QuickPup outperforms the available methods by orders of magnitude and thus makes it possible to automatically solve problems which couldn't be solved without human expertise before. Furthermore, the runtimes of QuickPup are typically below one second for real-world problem instances.

1 Introduction

Knowledge-based configuration systems (Stumptner 1997; Felfernig, Friedrich, and Jannach 2001) are among the most successful applications of Artificial Intelligence (AI).

In the area of configuration problems, the recently defined Partner Units Problem (PUP) represents a new challenge for industrial configuration systems in particular and for AI in general. The PUP was first described in 2010 (Falkner et al. 2011) and has gained more and more attention. Since 2011 the PUP has been taking part in the ASP competition[1] and hence is recognized as an important benchmark problem for logic programming.

The problem originates in the railway domain. In order to ensure safety of train movements, various equipment is responsible for detecting whether a track section (also called zone) is occupied by a train or a wagon. Figure 1 shows a small railway station with a through track consisting of three track sections (A, B, and C) and another two sections (D and E) for the sidings. The sections are separated by wheel sensors (1 - 8) which communicate the number of wheels entering and exiting the adjacent sections to occupancy indicators responsible for those sections. Thus, each indicator knows how many wheels are in its sections, i.e. whether the section/zone is occupied or not.

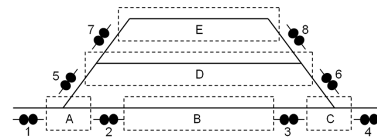

Figure 1: Example of a railway station layout

Due to stringent temporal and fail-safe requirements, special hardware is needed. Wheel sensors and occupancy indicators are connected via communication units. The communication units are limited therein that only a certain number (called *unit capacity* and abbreviated to UCAP) of elements of each type can be installed on a communication unit. For example only two sensors and only two occupancy indicators can be placed on a communication unit, hence UCAP=2[2]. Every element must be placed on exactly one unit. Furthermore, a unit can be connected to only a limited number (called *inter-unit capacity* and abbreviated to IUCAP) of other communication units. These connected units are called the partner units of the unit. Communication between a certain wheel sensor and a certain occupancy indicator (responsible for a certain track zone) is ensured either by installing them on the same unit or by connecting the corresponding communication units. The input is given in the form of a relation (i.e. the input relation, also called the input graph) defining which sensors are relevant for which zones. The PUP is about placing the elements on communication units and connecting the corresponding communication units so that all communication requirements given by the input relation are fulfilled.

Figure 2 shows the input relation and a corresponding solution for the example in Figure 1 when employing units with a UCAP = 2 and IUCAP = 2 (i.e. only two partner unit connections allowed per unit). Placing the indicator of zone A on unit 1 instead of unit 4 does not result in a valid solution. In this case, unit 1 and unit 3 should be connected partner units which would exceed the IUCAP of unit 3. An

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] https://www.mat.unical.it/aspcomp2011

[2] Without any loss of generality and for the ease of presentation, we will keep unit capacity equal for both the sensors and the indicators.

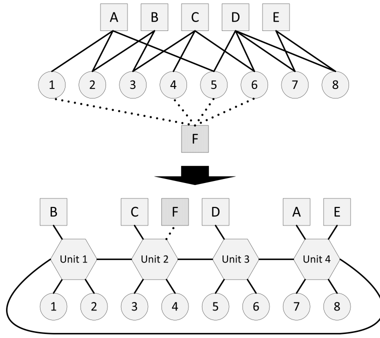

Figure 2: Input relation and solution for the example depicted in Figure 1

additional indicator for the whole through track (F), which subsumes zones A, B, C and needs information from sensors 1, 4, 5, 6, could be placed on unit 2 without exceeding the limits.

Apart from railway safety, the PUP has a broad range of further application domains. In the context of security monitoring, it is possible to think of zones which have to be monitored by a certain number of swiveling CCTV cameras. In the context of electrical engineering, any problems where two distinct types of elements (e.g. INs and OUTs) have to be placed on electrical units are in fact instances of the PUP. An application domain for the PUP in the area of distributed computing is bootstrapping in peer-to-peer networks with server and client processes. The challenge here is to controll the communication and computing load for each host by the number of client and server processes (UCAP) and the number of connected peers (IUCAP).

Based on the railway domain, we refer to the two types of elements to be installed on communication units as $zones$ and $sensors$ and to the communication units as $units$ for the rest of the paper.

Formally, the PUP constitutes a partitioning problem on a bipartite input graph $G = (Z, S, IR)$[3] into a set U of bags, with Z and S being disjunct sets of nodes connected by a set of edges $IR = \{(z, s) | z \in Z, s \in S\}$[4], such that:

- Every $z \in Z$ and $s \in S$ is contained in exactly one bag $u \in U$.
- Every $u \in U$ contains at most $UCAP$ $z \in Z$ and $s \in S$ each.
- Every bag $u \in U$ is connected to at most $IUCAP$ other bags.
- Let u_1 and u_2 be bags, whenever $z \in u_1$, $s \in u_2$ and $(z, s) \in IR$ then either u_1 and u_2 are connected or $u_1 = u_2$.

The PUP is indeed a hard problem. In the general case, that is when UCAP and IUCAP are part of the input and thus are not fixed, the PUP is NP-complete (Aschinger et al. 2011b; Garey and Johnson 1990). With $IUCAP = 0$ or $IUCAP = 1$ the PUP even mutates to classical bin-packing (Koiliaris 2011). When UCAP and IUCAP are not part of the input and thus are limited by some fixed natural number, the situation is two-sided. Whereas for the special case where $IUCAP = 2$ a polynomial-time algorithm could be found, algorithmic complexity remains unclear for the problem classes where $IUCAP \geq 3$ (Aschinger et al. 2011b) and as a consequence no polynomial-time algorithm is known. Industrial applications of our project partner from Siemens Austria employ units with a $IUCAP = 4$ and a $UCAP = 2$ (Falkner et al. 2011). As we will show in our test cases even the current polynomial time algorithm does not scale for mid-sized problems where $IUCAP = 2$. In order to deal with $IUCAP = 4$, the latest general problem solving methods have been explored such as Integer Programming, Constraint Programming, SAT Solving, and Answer Set Programming (Aschinger et al. 2011a). However, none of these methods have been able to compute solutions for all problem instances, given a time out after 600 seconds. Consequently, interactive configurations or large-sized problems are clearly out of reach for current methods. In this paper, we propose the novel algorithm *QuickPup* for solving PUPs. We present results which clearly show that QuickPup outperforms all state-of-the-art approaches by orders of magnitude. In particular, all the test cases published in (Aschinger et al. 2011b) are solved within at most a few seconds (usually a fraction of a second is required). The low runtimes of QuickPup even allow for interactive settings, which facilitate immediate system response when adapting input graphs in order to produce alternative solutions. Moreover, we present a first prototype web-application together with all employed test cases. These test cases were developed by our industrial partner representing realistic scenarios.

The remainder of this paper is structured as follows: In the next section we present the QuickPup algorithm. In Section 3 we discuss runtime results of QuickPup and compare them with state-of-the-art results produced by DecidePup, SAT Solving, Integer-, Constraint-, and Answer Set Programming. Furthermore, we present the *Simple Pup Solver*, which constitutes the very first web application dedicated to the PUP. The *Simple Pup Solver* uses *QuickPup* as low-runtime back-end. We conclude by summarizing the most important aspects and by addressing future work.

2 QuickPup Algorithm

QuickPup (QP) is a novel algorithm for tackling the PUP. QP basically follows a backtracking search approach but combines it with a static heuristic ordering of the zones and sensors (elements). Based on this fixed ordering, QP tries to assign each element to a unit and backtracks in case of unsatisfiability.

Listing 1 depicts the main procedure of QP. The input consists of a set of zones, a set of sensors, the input relation specifying which zones have to communicate with which sensors, and a maximal time limit (maxTime) for solution

[3] Without limiting generality, we consider only connected input graphs. Non-connected input graphs can be partitioned into independent problems.

[4] *Z* represents the Zones, *S* the sensors, *U* the units and *IR* is the input relation.

Listing 1 QuickPup: Main

```
INPUT: zones, sensors, inputRelation, maxTime

timeslice ← maxTime DIV numberOf(zones)

for all startZone IN zones do
    elements ← GetBreadthFirstOrder(startZone, zones, sensors, inputRelation)
    index ← firstIndexOf(elements)
    model ← {}
    stopTime ← SystemTime + timeslice
    status ← Assign(elements, inputRelation, model, index, stopTime)

    if status = TRUE then
        Minimize(model)
        return model
    else if status = FALSE then
        return FALSE
    else if status = TIMEOUT then
        Continue
    end if
end for
return TIMEOUT
```

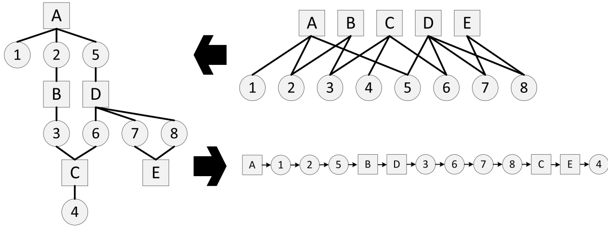

Figure 3: Reordered input relation and corresponding ordering for startZone 'A' for example given in Figures 1 and 2

calculation. The first important extension to simple backtracking is to restart the backtracking process from a different entry point if no solution can be found within a certain time slice. If unsatisfiability is proven, no further enrtry points are investigated. In QP each zone constitutes a possible entry point (startZone). For each entry point there is a maximal timeslice of *maxTime DIV number of zones*. Furthermore, the algorithm produces a different breadth-first ordering of the zones and sensors (elements) for each entry point.

For ordering the elements, QP uses a breadth-first strategy (see Figure 3). Starting from a certain zone (startZone) the next elements to be considered are all connected sensors based on the given input relation. Then, all zones connected to these sensors are considered, and so forth, until there are no more elements. This way of traversing a graph (i.e. the input relation) is known as breadth-first or also as topological order, as the graph is traversed from level to level. Listing 2 shows how this is realized. The $\oplus_l$ operation stands for inserting an element into a vector of elements at the last position.

Once an element ordering is fixed, QP creates an empty model and calls a recursive sub-procedure ($Assign$, see Listing 4) creating and connecting the units and trying to assign the elements. Thus, the model (also called the *solution graph* or simply *solution*) consists of the units, their partner unit connections, and the element assignments to the units. If $Assign$ runs into a timeout (SystemTime $>$ stopTime), QP continues with the next entry point (i.e. startZone is reassigned). If all iterations produced timeouts maxTime

Listing 2 QuickPup: GetBreadthFirstOrder

```
INPUT: startZone, zones, sensors, inputRelation

elements ← {startZone}
while sizeOf(zones) + sizeOf(sensors) < sizeOf(elements) do
    connectedElems ← getConnectedElems(elements, zones, sensors, inputRelation)
    for all elem IN connectedElems do
        if elem NOT IN elements then
            elements ← elements ⊕_l elem
        end if
    end for
end while
return elements
```

is reached and QP stops with no decision. If $Assign$ can prove the unsatisfiability of the given input relation QP returns FALSE. Please note that the combination of multiple start zones and breadth-first ordering focuses on the early detection of unsatisfiable instances. The idea is that if an instance is unsatisfiable then there is also at least one zone which is part of the conflict. Iterating through all zones guarantees that the subsequent backtracking procedure encounters the conflict in the beginning at least once.

Listing 3 QuickPup: Minimize

```
INPUT: model

for all unitA IN model AND unitB IN model AND unitA ≠ unitB do
    if NOT tooManyZones(unitA, unitB) then
        if NOT tooManySensors(unitA ⊕_m unitB) then
            if NOT tooManyPartners(unitA ⊕_m unitB) then
                unitA ← unitA ⊕_m unitB
                remove(unitB, model)
            end if
        end if
    end if
end for
```

If $Assign$ is successful, i.e. a consistent assignment for all elements has been found, such that the input relation is fulfilled, QP minimizes the model and returns it. Minimizing the model in this context means merging units when possible. This step is important for reducing the number of units in the model. Listing 3 depicts the idea. For pairs of units in the (consistent) model, merging is executed if possible. The $\oplus_m$ operator stands for unit merging. Units are limited in their number of possible zone/sensor assignments (UCAP) and their maximal number of partner unit connections (IUCAP). If merging is successful, the obsolete unit will be removed from the model.

Actual model checking by backtracking is done by $Assign$. Listing 4 shows the procedure. The input consists of the (ordered) elements, the input relation, the intermediate model, an index pointing to the next element to be assigned, and a time limit (stopTime). First, $Assign$ checks whether the index is greater than the last possible index. In this case all elements have already been assigned successfully and $Assign$ returns TRUE. If this is not the case, $Assign$ checks whether there is still some time left for further calculations, otherwise $Assign$ returns TIMEOUT.

If there is at least one element and some time left $Assign$ proceeds with the assignment of the next element (currElem). To this end QP first creates a new unit of the model and checks whether the assignment to the new unit leads to a consistent intermediate model, i.e. all relevant partner unit connections can be established. Please note that

Listing 4 QuickPup: Assign

```
INPUT: elements[], inputRelation, model, index, stopTime

if index > lastIndexOf(elements) then
    return TRUE
else if SystemTime > stopTime then
    return TIMEOUT
end if

currElem ← elements[index]

unit ← createNewUnit(model)
if AssignAndConnect(currElem, unit, model) = TRUE then
    consistent ← Assign(elements, model, index + 1, stopTime)

    if consistent = TRUE then
        return TRUE
    else if consistent = FALSE then
        UndoAssignAndConnect(currElem, unit, model)
        remove(unit,model)
    else if consistent = TIMEOUT then
        return TIMEOUT
    end if
end if

for all oldUnit IN model do
    if AssignAndConnect(currElem, oldUnit, model) = TRUE then
        consistent← Assign(elements, model, index + 1, stopTime)
        if consistent = TRUE then
            return TRUE
        else if consistent = FALSE then
            UndoAssignAndConnect(currElem, oldUnit, model)
        else if consistent = TIMEOUT then
            return TIMEOUT
        end if
    end if
end for
return FALSE
```

a unit is limited in its maximal number of zones/sensors (UCAP) and its maximal number of partner unit connections (IUCAP).

Consistency checking, the establishment of new partner unit connections and element assignment are carried out in $AssignAndConnect$, see Listing 5. Basically, $AssignAndConnect$ checks two preconditions before an element is assigned to a unit. First, there must be at least one free place left on the unit for picking up a further zone or sensor, respectively. In the case of a new unit, this precondition is always given. Second, $AssignAndConnect$ verifies that all additional partner unit connections can be established, this being limited by means of IUCAP[5].

Listing 5 QuickPup: AssignAndConnect

```
INPUT: currElem, unit, model, inputRelation

if hasFreePlace(unit, currElem) = FALSE then
    return FALSE
else if relevantUnitsCanBeConnected(currElem, unit, model, inputRelation) = FALSE then
    return FALSE
else
    add(currElem, unit)
    establishConnections(currElem, unit, model)
    return TRUE
end if
```

If the assignment is successful, $Assign$ calls itself recursively with the updated intermediate model and incremented index pointing to the next element. In case the subsequent $Assign$ returns TRUE, all remaining elements have been assigned consistently, and the current instance of $Assign$ also returns TRUE. If a timeout has been triggered, and hence the return value of the called $Assign$ instance is TIMEOUT, the current $Assign$ back-propagates TIMEOUT.

If the called $Assign$ instance returns FALSE, this means that no assignment for the remaining elements could be found which is consistent with assignment of the current element (currElem) to the newly created unit. In this case, all changes which have been done by $AssignAndConnect$ are revoked and the new unit is removed from the model.

In a second step, QP tries to assign currElem to one of the old units already existing in the model. The procedure for any old unit is similar to the case where new units are exploited, except that it is well possible that the unit could be 'full', i.e. there is no free place for the current element on that unit. If no consistent assignment could be found both for both the old units and a newly generated unit, $Assign$ returns FALSE (i.e. backtracks).

It is obvious, that preferring the creation of new units typically results in non-minimal models, regarding the number of units. For optimization of the model, the $Minimize$ procedure is necessary. If the problem is only to decide whether for a given input relation a configuration exists, the optimization step can be skipped. Optimization (i.e. minimizing the model) can also be skipped when using a variant of $Assign$. By changing $Assign$ such that assigning the current element (currElem) to an old unit is preferred and new units are only generated if the assignment to an old unit is impossible or does not lead to a consistent model, the resulting algorithm already produces optimized models by construction. We will designate the optimizing version of QuickPup as QuickPup* (QP*).

Both variants of QuickPup, i.e. QP and QP*, have different strengths and weaknesses. On the one hand, QP* produces models which are typically smaller than those produced by QP, also when QP uses the $Minimize$ procedure. On the other hand, producing a non-optimal model with QP may be much easier for many problems than producing an optimized model with QP*. Note, that neither QP* nor applying a minimization of the model guarantees minimality in the number of units. However, our experimental results show that the optimum is found in many cases. As a matter of fact, QP* has always produced optimal models so far.

3 Experimental Results

In order to test the performance of QP (QP*) we refer to the results and the benchmark instances presented in (Aschinger et al. 2011a). All experiments in (Aschinger et al. 2011a) were carried out on a 3 GHz dual-core system with 4 GByte of RAM, running Fedora Linux. The new results for QP (with model minimizing) and QP* were produced by an Intel Centrino dual-core notebook with 2 GHz and 2 GByte of RAM, running Windows XP. QP and QP* are implemented in Java 1.5.

In (Aschinger et al. 2011a) five different implementations were tested[6]:

- DecidePup (DP, polynomial time algorithm only for IUCAP = 2 (Aschinger et al. 2011b))

[5] Note, that the partner unit connections are uniquely determined, i.e. no search needed.

[6] Detailed information about the implementations can be found in (Aschinger et al. 2011a)

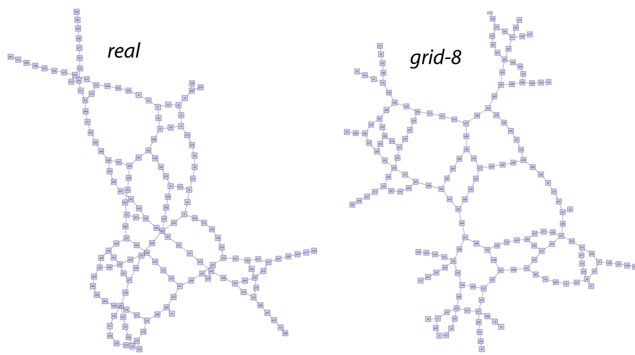

Figure 4: Real problem instance of Siemens Austria in the railway station domain and benchmark instance *grid8*

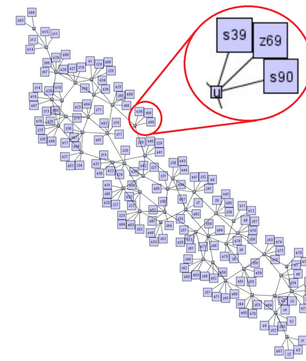

Figure 5: Solution graph for the real problem instance shown in Figure 4 produced by Simple Pup Solver

- Constraint Programming (CP, implementation with Eclipse-Prolog (www.eclipseclp.org))
- SAT Solving (SAT, MiniSat (www.minisat.se))
- Integer Programming (IP, two different systems were tested: CBC from the COIN-OR project (www.coin-or.org) and IBM's Cplex (www.ibm.com))
- Answer Set Programming (ASP, Clingo from the Potsdam Answer Set Solving Collection (potassco.sourceforge.net))

The benchmark[7] consists of two parts. In part one the corresponding instances are to be solved with a unit capacity of 2 (UCAP=2) and an inter unit capacity of 2 (IUCAP=2). The instances of part two are to be solved with the same UCAP but with an IUCAP = 4. There are four different types of instances: double (dbl) double-variant (dblv), triple (tri), and grid[8]. The instances differ in their number of zones and sensors and the number of sensors per zone. Furthermore, the instances have different structural characteristics, as they are patterned on real problem instances which have already been successfully configured by our project partners from Siemens Austria[9].

For example, the *grid* instances are based on real problem instances in the domain of railway stations. Figure 4 shows two input graphs of partner unit problem instances. On the left, a real railway station problem instance provided by our project partners is shown. As opposed to this, the right hand side depicts the graph of the *grid8* benchmark instance. As it can be recognized easily, the structures of the instances are similar. Figure 5 shows the corresponding solution graph for the real instance shown in Figure 4.

Runtimes of QP/QP* for real cases provided by Siemens Austria are significantly below one second, whereas ASP (which performs best among the other approaches) needs up to 17 minutes. Although, the runtimes for real cases already show the potential of QP, more insights can be gained when looking at the scaled benchmark instances.

Table 1 and Table 2 summarize the experimental runtimes (seconds) on the described benchmark instances[10]. The *units* column lists the minimal number of units required for a consistent solution. A minimal number of '0' means that no solution exists (instances tri-34 and tri-64 for IUCAP = 2). In these cases, the results refer to the time needed in order to prove unsatisfiability. A '/' means that the corresponding approach could not solve an instance within a certain time frame. For the experiments in (Aschinger et al. 2011a) the time frame was limited to 600 seconds. Although using a weaker hardware, a time limit of 100 seconds was more than enough for QP and QP*. Except for QP, all approaches produced only minimal solutions. The number of additional units needed by QP is listed in the *+units* column.

Only QP and QP* were able to solve all instances. Even DP, which is a polynomial time algorithm capable of only solving problem instances with IUCAP = 2, was not able to solve all instances (with IUCAP = 2). In the cases where the other approaches were able to calculate a solution, QP was always much faster. In fact, the time needed for the calculation of most solutions was significantly below one second. In terms of runtime, only QP* was better than QP in a few cases. The overhead of additional units used by QP is very small in most cases. As a matter of fact, for many instances QP produced minimal solutions, i.e. *+units = 0*. QP* always produced minimal solutions.

Based on the algorithms presented in this paper, we implemented the first online web application for solving the partner units problem. Figure 6 shows a screen shot of the interface of the *Simple Pup Solver* (SPS). The SPS uses units where UCAP = 2 (i.e. max. two sensors and two zones per unit). The timeout can be set to 1 second or 10 seconds. The IUCAP can be set to 2 and to 4, whereby the default is 4 as this reflects the realistic settings of our industrial partner. Furthermore, an optimizing option is given such that it is

[7]Benchmark instances can be downloaded at http://demo2-iwas.uni-klu.ac.at/pupsolver/

[8]Instances *grid-90, ..., grid-99* were removed from the benchmark as the corresponding input graphs were not connected such that those instances can be seen as a collection of trivial non-relevant instances.

[9]More details about the instance structure can be found in (Aschinger et al. 2011a).

[10]For IP the better result produced by the two different approaches is listed.

INSTANCE	UNITS	DP	SAT	CP	ASP	IP	QP*	QP	+UNITS
dbl-20	14	0.01	0.48	0.02	0.16	1.53	0.00	0.02	1
dbl-40	29	0.05	2.36	0.28	3.93	13.58	0.00	0.03	1
dbl-60	44	0.08	29.74	0.42	/	213.58	0.00	0.03	1
dbl-80	59	0.16	/	1.14	/	522.5	0.01	0.04	1
dbl-100	74	0.41	/	1.89	/	/	0.03	0.08	1
dbl-120	89	0.39	/	3.21	/	/	0.02	0.08	1
dbl-140	104	0.59	/	5.01	/	/	0.02	0.09	1
dbl-160	119	0.71	/	13.94	/	/	0.03	0.10	1
dbl-180	134	0.87	/	20.07	/	/	0.04	0.13	1
dbl-200	149	1.08	/	14.40	/	/	0.04	0.15	1
dblv-30	15	65.49	0.42	0.09	0.26	2.93	0.00	0.00	0
dblv-60	30	/	3.15	0.26	1.94	/	0.01	0.00	0
dblv-90	45	/	12.54	0.82	27.35	/	0.01	0.01	0
dblv-120	60	/	41.65	1.85	13.92	/	0.02	0.01	0
dblv-150	75	/	20.97	3.48	29.54	/	0.02	0.02	0
dblv-180	90	/	44.28	6.20	54.50	/	0.03	0.03	0
tri-30	21	0.50	0.79	1.07	0.41	45.17	2.33	0.08	0
tri-32	20	/	0.74	0.64	0.26	4.66	0.02	0.04	1
tri-34	0	/	22.77	21.10	0.89	5.06	0.03	0.03	0
tri-60	40	114.08	315.42	158.49	4.40	108.01	2.08	1.61	0
tri-64	0	/	379.36	/	43.88	76.26	0.20	0.21	0

Table 1: Results UCAP=2

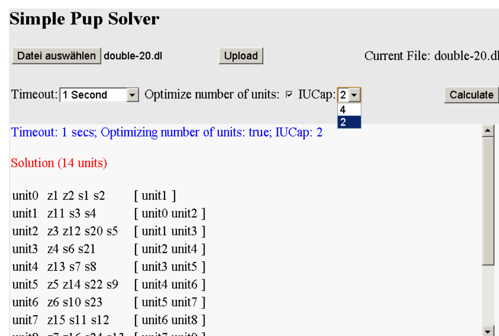

Figure 6: The web interface of the Simple Pup Solver, http://demo2-iwas.uni-klu.ac.at/pupsolver

possible to use either QP or QP*.

In order to make the structure of a problem and its solution visible and, as a consequence, more comprehensible, we have also included the feature of a smart graph representation of the input relation and the solution graph[11].

4 Conclusions

In this paper, we have presented the QuickPup algorithm (QP) for tackling the Partner Units Problem (PUP). The PUP constitutes a configuration problem with many industrial application domains such as railway safety, electrical engineering, or distributed computing. Beside its practical relevance, the PUP is employed as a benchmark problem for the ASP competition. By comparing the runtimes of QP with previous results for the state-of-the-art approaches SAT Solving, Constraint Programming, Answer Set Programming , Integer Programming, and DecidePup, we could clearly show the superiority of the proposed algorithm. The high performance of QP also enables automatic problem solving for real instances where other AI approaches fail to find a solution within reasonable time. Future work will mainly consist of two things. First, we will create a more challenging benchmark in order to further investigate the strengths, weaknesses and limits of QP. Second, we will apply QP to reconfiguration tasks, where the algorithm starts with a partial solution which must not be altered any more.

[11] In fact, Figure 4 and Figure 5 were produced by this feature.

INSTANCE	UNITS	SAT	CP	ASP	IP	QP*	QP	+UNITS
tri-30	20	2.40	0.12	0.40	24.79	0.00	0.00	0
tri-32	20	1.91	0.14	0.66	20.84	0.00	0.00	2
tri-34	20	1.98	/	0.60	/	0.00	0.00	5
tri-60	40	/	0.52	11.07	/	0.00	0.01	0
tri-64	40	/	/	7.61	/	0.01	0.01	6
tri-90	60	401.44	1.50	332.34	/	2.33	0.01	0
tri-120	79	/	3.37	/	/	8.23	0.02	0
grid1	50	78.19	/	31.45	/	0.18	0.02	0
grid2	50	90.89	/	18.91	/	0.69	0.01	0
grid3	50	88.87	/	25.72	/	0.10	0.01	1
grid4	50	95.12	/	24.66	/	0.00	0.01	0
grid5	50	454.42	/	48.88	/	0.01	0.01	2
grid6	50	204.85	/	9.15	/	0.01	0.01	1
grid7	50	112.36	/	12.89	/	0.05	0.01	2
grid8	50	/	/	11.89	/	1.54	0.01	0
grid9	50	91.62	/	19.71	/	0.01	0.01	0
grid10	50	545.16	/	13.54	/	4.15	0.02	0

Table 2: Results UCAP=4

5 Acknowledgments

Work has been performed in the scope of FFG-FIT-IT Grant 825071 in cooperation between Universität Klagenfurt, Siemens Austria, and the University of Oxford.

References

Aschinger, M.; Drescher, C.; Friedrich, G.; Gottlob, G.; Jeavons, P.; Ryabokon, A.; and Thorstensen, E. 2011a. Optimization methods for the partner units problem. In *CPAIOR'11*, 4–19. Berlin, Heidelberg: Springer-Verlag.

Aschinger, M.; Drescher, C.; Gottlob, G.; Jeavons, P.; and Thorstensen, E. 2011b. Tackling the partner units configuration problem. In *IJCAI'11*, 497–503.

Falkner, A.; Haselboeck, A.; Schenner, G.; and Schreiner, H. 2011. Modeling and solving technical product configuration problems. *AI EDAM* 115–129.

Felfernig, A.; Friedrich, G.; and Jannach, D. 2001. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering* 15(2):165 – 176.

Garey, M. R., and Johnson, D. S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Koiliaris, K. 2011. *Complexity of constraint satisfaction and automated configuration*. United Kingdom: University of Oxford.

Stumptner, M. 1997. An overview of knowledgebased configuration. *AI Commun.* 10:111–125.