A Constraint Model for the Settlers Planning Domain

Peter Gregory
University of Strathclyde
Glasgow, UK
pg@cis.strath.ac.uk

Andrea Rendl
University of St. Andrews
St. Andrews, UK
andrea@cs.st-and.ac.uk

Abstract

The Settlers planning domain has proved a challenging problem for planning technology. We present a preliminary model of Settlers in the Essence' specification language. We generate a constraint model for the CSP solver Minion using the automated modelling tool Tailor. We show this model to be competitive with state-of-the-art planning technology.

Introduction

Settlers is an challenging planning domain that was been introduced by (Long & Fox 2003). It relates to the German board game 'Settlers of Catan' and includes city development involving good production, building construction and transportation. It is a complex problem that shares features with supply chain problems.

Constraint Programming is a successful technique to tackle hard combinatorial problems and has successfully handled Planning Problems in the past, such as Peg Solitaire (Jefferson *et al.* 2006) and various others (Pralet & Verfaillie 2008). Hence Settlers is an interesting domain for Constraint Programming. Constraint Programming has a two-phase approach: first a problem is *modelled* as Constraint Satisfaction Problem (CSP) and then it is *solved* by applying propagation and search. Modelling, however, has proven to be difficult, especially for non-experts and an efficient model is crucial to solve the problem in a reasonable amount of time. Therefore, the challenge lies in modelling Settlers efficiently.

In this paper, we propose a constraint model for the Settlers domain and compare its solving performance and its solutions to those of an equivalent planning model. Our observations are noteworthy. The constraint model showed to be more scalable with respect to the number of goals and number of cities. However, Settlers is very difficult to represent as as a constraint model while the planning model is an intuitive representation of the problem. Both problem models are competitive.

Numeric Planning Problems

A numeric planning problem consists of three parts: a set of operators, an initial state and a partial goal state. The initial state contains both propositions and assignments to numeric variables. The goal state is a list of facts that have to be achieved and a list of numeric conditions that have to be satisfied in order to solve the problem.

An operator is defined in three parts: its precondition, its propositional effects and its numeric effects. The precondition consists of a set of propositional and numeric conditions that are required to be true in order to apply the action in a given state. The propositional effects state that some facts are added to the resulting state, and some are removed. The numeric effects of an action are that the values of given variables are either assigned to, increased, decreased or scaled.

Numeric planning problems in this form can be represented in PDDL 2.1 (Fox & Long 2003). A solution to a numeric planning problem is a set of ordered actions that transform an initial state into a state that satisfies the goal condition.

Constraint Satisfaction Problems

In Constraint Programming, problems are modelled as constraint satisfaction problems (CSPs). A CSP is defined as a set of decision variables and a finite set of constraints on those variables. Decision variables typically range over a finite set of integer values that is denoted *domain* in the constraint community. A domain represents the values that a particular decision variable may take. Constraints on decision variables restrict those domains.

For illustration, consider the following sample CSP that is formulated in the solver-independent modelling language ESSENCE$'$ (Gent, Miguel, & Rendl 2007): we define two decision variables x and y, both ranging over the domain $\{1..4\}$, and the constraints $x > y$ and $x + y = 4$ (for which $x = 3, y = 1$ is the only solution).

find x, y : int(1..4)
such that
$x > y$,
$x + y = 4$

However, modelling problems as CSPs has proven to be difficult and requires expertise. A lot of problems have no intuitive CSP representation and especially novices struggle to find reasonable representations. Furthermore, most problems can be specified as several different CSPs (of different quality) and it requires a lot of expertise to determine the most effective representation (with respect to solving time).

This bottleneck prevents the widespread use of constraint programming techniques.

The Settlers Problem

The Settlers Problem is taken from the third International Planning Competition (Long & Fox 2003) in 2002 and proves as challenging to planners today as it did at that time. It is based loosely on the German board game 'Settlers of Catan[1]' and involves city development. Both raw and refined materials are required in order to construct buildings. Some buildings are prerequisites for the extraction and production of different materials. For example, a mine is required to extract ore and a saw-mill is required to produce wood.

Each instance of the Settlers Problem has a goal of constructing various buildings across a set of cities. Different cities have access to different raw-materials and so goods usually have to be transported between cities in order to construct the required buildings. There are three ways of transporting goods: by cart, train and ship. There are different costs associated with creating these forms of transport, and there are different costs in terms of operating them.

In the IPC3 version of the problem, an instance consists of:

- A set of cities connected by an initial road network. The road network is unalterable.
- A set of raw materials that each location can potentially exploit (not all locations have access to all of the materials). The possible materials are timber, stone and ore.

  Timber Timber costs one unit of labour to fell.

  Stone Quarrying stone costs a unit of labour and a unit of resource.

  Ore Mining ore costs a unit of labour and two units of resource.
- A set of costs, a combination of which need to be optimised. These are labour, pollution and resource. Most actions have a labour cost. The extraction of ore and stone has a resource cost. Production of some derivative materials and some transport methods carry a pollution cost.
- A set of building requirements for each city, taken from the following buildings:

  Cabin Required to log timber. No material requirements, costs one unit of labour.

  Quarry Required to extract stone. No material requirements, costs two units of labour.

  Mine Required to extract ore. Requires two units of wood to construct, costs three units of labour.

  Sawmill Produces wood. Requires two units of timber to construct, costs two units of labour.

  Coal Stack Produces coal. Requires one unit timber to construct, costs three labour units.

  Iron Works Produces iron. Requires two units of both wood and stone, costs two units of labour.

  Dock Allows ships to dock at coastal cities. Requires two units of both wood and stone to construct, costs two units of labour.

  Wharf Allows ships to be built at a coastal city. Requires a dock and two units of both iron and stone, costs two units of labour.

  Houses Each location has a requirement of a certain number of houses. Each house requires one unit of wood and one unit of stone. A single labour unit is required in order to construct one house.
- Production of derivative materials consume raw materials in the following way:

  Wood Consumes one unit of timber, produces one unit of wood.

  Coal Consumes one unit of timber, produces one unit of coal. Produces a unit of pollution.

  Iron Consumes one unit of coal and one unit of ore to produce one unit of iron. Produces two units of pollution.
- There are three different ways in which goods can be transported, by road, by rail and by sea. Loading and unloading one item of goods onto any form of transport costs one labour unit.

  Road Goods transported by road are transported on carts. Each cart can transport one unit of any type of material. Constructing a cart requires only one unit of timber. There is no material cost associated with transporting by road except construction of the cart. Constructing a cart costs one unit of labour. Moving a cart costs two units of labour.

  Rail To transport goods by rail requires rail links and at least one train to be created. To construct a rail link between two locations, a unit of both iron and wood is required. To construct a train, two units of iron are required; moving a train requires a unit of coal which must be stored on the train. Having a train allows much greater transport capacity than using carts. A train has capacity 20, although that must include the coal needed to complete the journey. Constructing a train costs two units of labour, constructing track also costs two units of labour. Moving a train produces a unit of pollution.

  Sea To transport goods by sea, a dock is required at two or more coastal cities. A ship must be constructed at a city with a wharf and has capacity 50 when constructed. Four units of iron are required in order to construct a ship and two units of coal are consumed in each journey (as with trains, this is taken from the ship's capacity). Constructing a ship costs three units of labour. Moving a ship produces two units of pollution.

The Planning Model

In this section we describe the planning model used in IPC3. The model has three types: cities, vehicles and resources. There are numeric values that represent the amount of each resource at each location and in each truck. There are numeric values to represent labour, pollution and resource

[1] First published in Germany as "Die Siedler von Catan" by Franckh-Kosmos Verlags-GmbH & Co.

costs. Facts represent location of vehicles, which buildings exist at a location, how the locations are connected and what properties each location has. In the model, actions model basic functions in the problem (extracting resource, producing derivative materials, constructing buildings and transportation of goods). Templates for each of these type of action are given below:

Extracting Resource

Parameters	?c - City ?r - Resource
Precondition	?c produces ?r ?c has prerequisite buildings for ?r
Effect	Increase amount of ?r at ?c by one Increase costs as appropriate

Derivative Production

Parameters	?c - City ?r - Resource
Precondition	?c has resources to make ?r ?c has prerequisite buildings for ?r
Effect	Increase amount of ?r at ?c by one Decrease resources required to make ?r Increase costs as appropriate

Building Construction

Parameters	?c - City ?b - Building
Precondition	?c has resources to construct ?b ?c has prerequisite buildings for ?b
Effect	?c has building ?b Decrease resources required to build ?b Increase costs as appropriate

Transportation

Load	
Parameters	?c - City ?v - Vehicle ?r - Resource
Precondition	?r is at ?c ?v has remaining capacity > 0
Effect	Increase amount of ?r in ?v by one Decrease amount of ?r at ?c by one Increase costs as appropriate

Move	
Parameters	?c - City ?v - Vehicle ?c2 - City
Precondition	?c and ?c2 are connected ?v has sufficient fuel ?v is at ?c
Effect	Decrease amount of fuel in ?v ?v is at ?c2 Increase costs as appropriate

The Constraint Model

In this section, we describe our constraint model of the Settlers problem. We formulate Settlers in the constraint modelling language ESSENCE′ (Gent, Miguel, & Rendl 2007). ESSENCE′ is a solver-independent modelling language that provides a wide range of facilities, such as quantifications, to allow abstract modelling. An ESSENCE′ instance is obtained by pairing a problem class model (*problem description*) with a parameter specification (*data*). Its syntax is related to mathematical description languages and is therefore self-explanatory. All the constraint examples are given in ESSENCE′.

Restrictions

In our model we restrict transportation to carts only. We also specify that houses are built in the last step. Additionally, we only produce goods if we need them in the current step, i.e. we do not store goods.

Modelling Settlers

We formulate Settlers as a satisfaction problem i.e. we try to satisfy the problem in t steps. The model consists of constants, parameters, variables and constraints on those variables. We describe our model by considering particular parts of the problem and demonstrate how we can express them in a CSP.

Basics Each raw material (e.g. iron) and each building type (e.g. ironworks) are represented by unique integer constants. We define a parameter *horizon* that specifies the number of steps to build required buildings. In order to distinguish between timesteps, we introduce the integer domain TIME, as shown below.

letting TIME **be domain** int(0..*horizon*)

Representing cities The number of cities is specified by the parameter *noCities*. The distances between cities are given in the parameter-array *driveTimes*, encoded as an adjacency matrix. Each city has certain resources, such as woodland or access to the coast, that are also given by a parameter. The range of cities is defined as illustrated below.

letting CITIES **be domain** int(0..*noCities*)

Building Facilities The aim of Settlers is to construct certain buildings in each city. In our model, these building requirements are given by the parameter-array *buildingRequirements* that contains 0/1 variables. As an example, the following two constraints

buildingRequirements[*city*, MINE] = 1
buildingRequirements[*city*, CABIN] = 0

state that *city* is required to build a mine but no cabin. The variable-array *buildingBuiltInCity* holds the step at which a building has been built, for instance

buildingBuiltInCity[*city*, CABIN] = 2

states that the cabin in *city* has been built at step 2. For the case that a building is never built in a city, we introduce the constant NEVER that we define as the step following our last step (which we will never reach).

letting NEVER **be** *horizon*+1

Hence we can state the constraint that if we are required to build a particular building, we need to build it at some time:

```
forall city : CITIES .
  forall b : BUILDINGS .
    (buildingRequirements[city,b]=1) =>
      (buildingBuiltInCity[city,b] != NEVER)
```

Producing Goods The production of goods in a particular city at a certain step t is represented by the variable-array *production* where

$$production[t, city, good]$$

represents the amount of *good* (for instance timber) that was produced at step t in *city*. Producing raw materials requires local resources, e.g. producing timber requires woodland. Each resource requirement is expressed by a constraint. As an example, we give the constraint stating that if there is no woodland in *city*, it cannot produce timber at any time.

```
forall city : CITIES) .
  (cityResources[city,WOOD] = 0) =>
    (forall t : TIME .
      (production[t, city, TIMBER] = 0)
    )
```

The production of goods also requires production facilities, e.g. timber production requires a cabin. The appropriate facility has to be built before production, which we state in another constraint. As an example, the constraint below denotes that if *city* produces timber at step t, a cabin must have been built there before t.

```
forall city : CITIES .
  forall t : TIME .
    (production[t, city, TIMBER] > 0) =>
      (buildingBuiltInCity[city, CABIN] < t)
```

Goods Requirement Both material production and building construction require goods. For instance, producing a unit of wood requires a unit of timber. The parameter-array *requirementTable* gives the amount of each good that is required to produce a particular good or build a certain building. The amount of goods a city requires for production and construction is represented by the variable-arrays *materialRequirement* and *buildingRequirement* respectively. For illustration, the following constraint

$$materialRequirement[t, city, \text{TIMBER}] = 3$$

states that *city* requires 3 units of timber for producing material at step t. In the following constraint, we state that the *materialRequirement* of a particular *good* equals the amount of *good* that is required to produce other goods.

```
forall city : CITIES .
  forall good : GOODS .
    forall t : TIME .
      materialRequirement[t, city, good] =
        ( sum prodGood : GOODS .
            production[t, city, prodGood]*
            requirementTable[prodGood, good]
        )
```

Similarly, we constraint the building requirements. The total amount of goods a city requires at step t is denoted by the decision variable-array *totalRequirement*[t, *city*, *good*]. We state that the total requirement is the sum of material and building requirement:

```
forall city : CITIES .
  forall good : GOODS .
    forall t : TIME .
      totalRequirement[t, city, good] =
        materialRequirement[t, city, good]
       + buildingRequirement[t, city, good]
```

Transportation In case a city requires a raw material that it cannot produce itself, it has to import it from another city. Hence transportation is necessary. We represent transportation by a variable-array for each good. For instance

$$transportIron[t, fromCity, toCity]$$

represents the amount of iron that is transported from *fromCity* to *toCity*. The variable-array *totalExport* contains the the amount of a particular good a city exports. Hence the constraint

$$totalExport[t, city, \text{IRON}] = 3$$

states that *city* exports 3 units of iron at step t. Similarly, we define the variable-array *totalImport* to denote the amount of a particular good a city imports at a certain step. Since transportation between cities takes time, a good is imported after it has been exported. The exact offset is computed from the distances between cities that are given in the adjacency matrix *driveTimes*. As an example, consider the following constraint stating that the total iron import of *city* is the sum of all iron that has been transported to *city* from all other cities.

```
forall city : CITIES .
  forall t : TIME .
    totalImport[t, city, IRON] =
      sum fromCity : CITIES .
        (t - driveTimes[fromCity, ,city] > 0)*
        transportIron[t - driveTimes[fromCity, city],
                fromCity, city]
```

The multiplication of (t - *driveTimes*[*fromCity*, ,*city*] > 0) with *transportIron* serves the elimination of invalid (negative) indices of the array *transportIron*: invalid index references are multiplied by zero. However, there is no consensus among ESSENCE′ developers if invalid array references should be declared undefined and constraints like this be invalid. Nevertheless, his has no effect on the validity of the constraint in the current version of ESSENCE′.

In our model, we restrict transportation to carts only. We consider carts as an additional good that we import from other cities. We express cart-transportation by setting the amount of carts that *city* imports (i.e. the carts heading to *city*) is greater or equal to the sum of all imported goods:

```
forall city : CITIES .
  forall t : TIME .
    totalImport[t, city, CART] >=
      totalImport[t, city, IRON]
     + totalImport[t, city, WOOD]
     + totalImport[t, city, STONE]
     + totalImport[t, city, COAL]
     + totalImport[t, city, ORE]
     + totalImport[t, city, TIMBER]
```

Finally, we ensure that production, import and export satisfy the total requirement, as we state in the constraint below:

```
forall city : CITIES .
  forall t : TIME .
    forall good : GOODS .
      (good != CART) =>
        (totalRequirement[t, city, good] =
              production[t, city, good]
            + totalImport[t, city, good]
            - totalExport[t, city, good]
        )
```

Labour Production, construction and transport create labour. We summarise all labour in the variable *finalLabour* which is the sum of the total labour of all cities over all time and the labour accumulated by loading and driving carts, as the constraint below illustrates.

```
finalLabour = loadLabour + cartLabour +
        (sum t : TIME .
          sum city : CITIES .
            totalLabour[t, city]
        )
```

Comparing the Models

It is interesting to notice that neither of the two models are capable of providing optimality guarantees for the original problem specification. In this section we examine the differences between the models, and also their limitations.

Planning Model

The planning model is restricted in the number of possible vehicles that can be created. Obviously, a very large problem might be best solved using many vehicles. Perhaps more vehicles are required in order to find the optimal solution. Unfortunately, adding more vehicles dramatically increases the number of ground actions in the problem, and so a balance has to be found between number of vehicles and size of ground instance.

PDDL also does not allow numeric variables in operators, and so each resource production action only generates one unit of resource. This is in contrast to the constraint model, where many units of resource can be produced at the same timestep.

Constraint Model

The first restriction of the constraint model is that a finite planning horizon has to be decided upon before search. This possibly restricts the solver from finding optimal solutions (even worse, not finding a solution at all). This problem is reduced somewhat, since the model we have created is very compressed as many actions are performed at each layer.

Another problem relates to the storage and production of resources. As all of the model's domains are finite, we have to decide maximum values for the amount of stored, transported and produced resources at each timestep. In practice, we can experiment to find suitably high values which offer good results.

Experiments

In order to evaluate the comparative performance of a constraint solver on our model and a planner on the IPC3 planning model, we use a combination of Tailor and Minion (Gent, Jefferson, & Miguel 2006) to first translate and then to solve the constraint instances, and the planner LPRPG (Coles *et al.* 2008) to solve the planning instances. LPRPG was chosen as its linear programming based heuristic provides best guidance for domains that involve flow of numeric values compared with other numeric planners. As we include both the time taken to translate our Essence' model and the time taken to solve the resulting Minion instance, we use the name Tailor + Minion to refer to the constraint solver in the discussion of the results.

To generate the instances, a modified version of the IPC3 generator is used. This modified generator has two important parameters: number of cities in the problem and the number of individual goals. Goals are either building goals or housing goals. A building goal means that a location is required to build a sawmill, iron-works, coal-stack, docks or wharf. A dock and a wharf can only be considered at coastal locations. A housing goal specifies how many houses have to be built at a certain location (this number ranges from one to ten). In order to maintain consistency between the two models, no goals mention railways. Railway construction is not prohibited in the planning model, however, and so any leverage gained from railway construction can be exploited in plans.

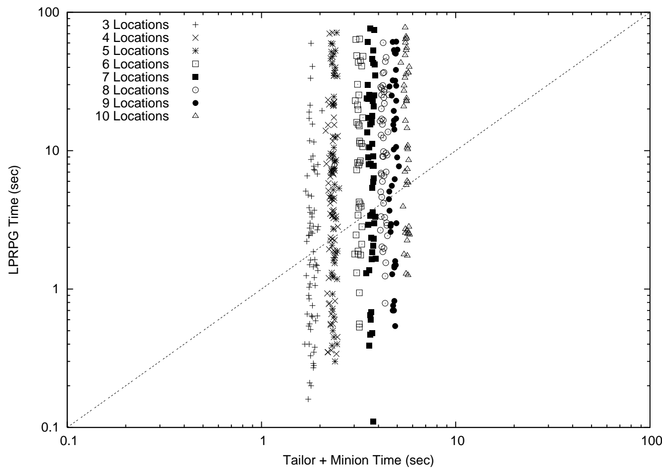
We have generated a problem set with varied numbers of cities and goals. We generate 10 random instances for each combination of both cities and goals between 3 and 10 inclusive. This provides 640 instances of varied difficulty to provide the basis of comparison for the solvers. An important final remark is that in the planning model, it is required to explicitly state the potential number of vehicles. We have chosen five potential vehicles after some preliminary tests suggested this provided good LPRPG performance.

Our experiments are all run on Intel Dual-Core 3.40GHz machines equipped with 2GB RAM and running Ubuntu Linux (kernel 2.6.24). Each run was limited to 90 seconds CPU time. This was decided after some preliminary tests revealed that very few instances that were unsolved after 90 seconds were solved at all within a much larger time-frame.

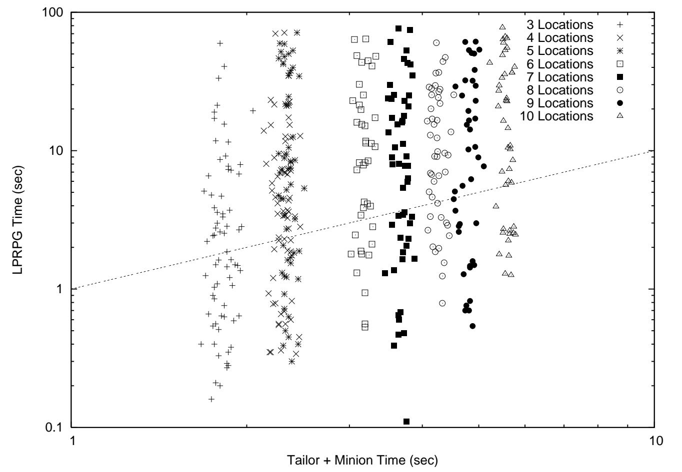
Results

We have found that both of the approaches compared provide good performance on the test instances. Graphs of the performance of LPRPG and Tailor + Minion are shown in Figures 1 and 2. Figure 1 displays the data partitioned by the number of cities in each instance. Figure 2 displays the data partitioned by the number of goals in each instance.

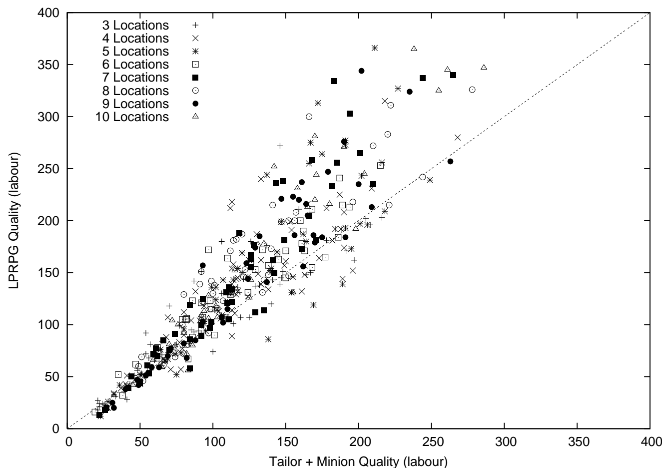
Time Performance Figure 1(a) and Figure 1(b) show the comparative performance of each solver directly, in terms of time taken to solve. Each of these graphs are log plots of the time taken to solve. On each graph the line $y = x$ is drawn. Any points above this line indicate stronger performance by Tailor + Minion, any points below the line indicate stronger performance by LPRPG.



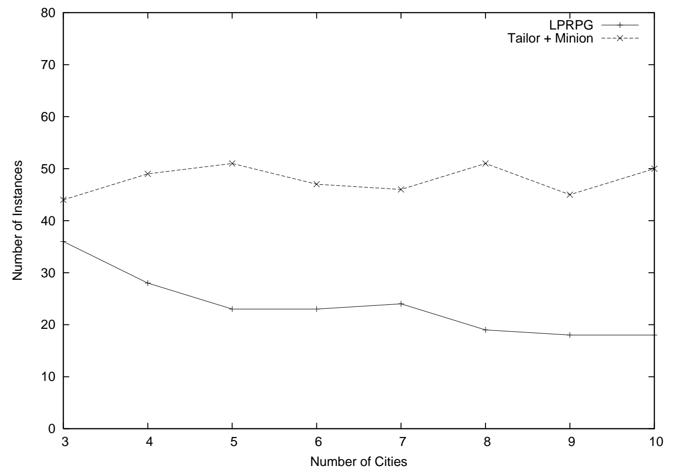
(a) Tailor + Minion vs. LPRPG: Time (sec). Results partitioned by number of cities.



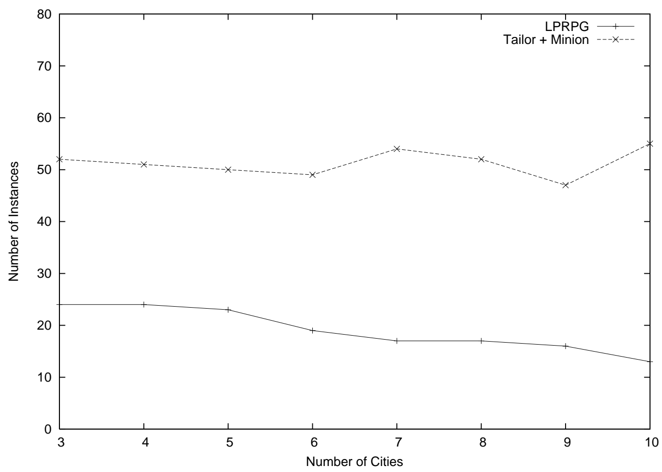
(b) Tailor + Minion vs. LPRPG: Time (sec). Results partitioned by number of cities. (more detailed view)



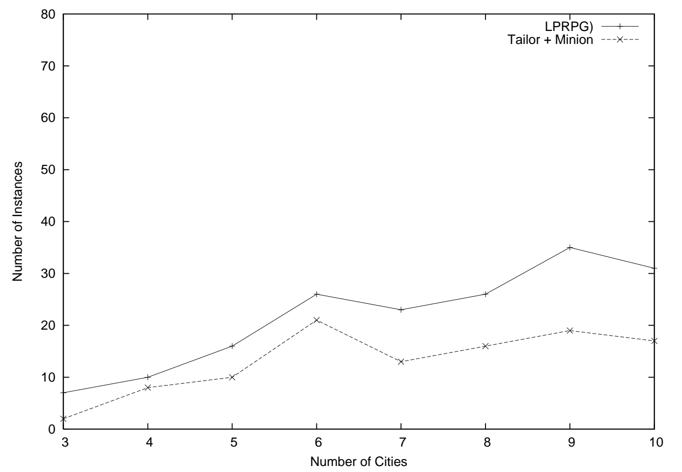
(c) Tailor + Minion vs. LPRPG: Quality (labour). Results partitioned by number of cities.



(d) Number of instances solved faster by LPRPG and Tailor + Minion

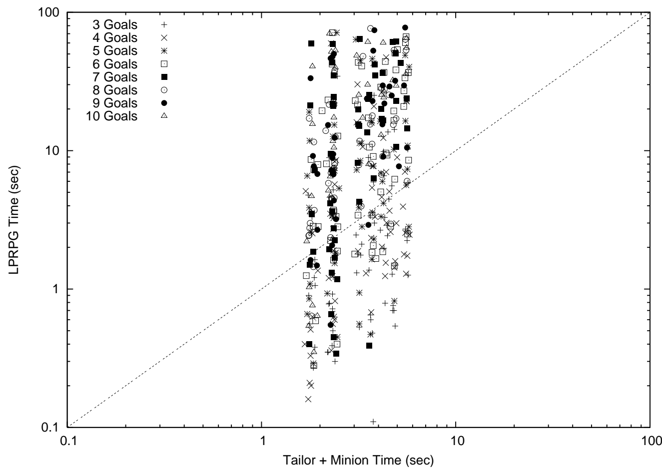


(e) Number of instances solved to a higher quality by LPRPG and Tailor + Minion

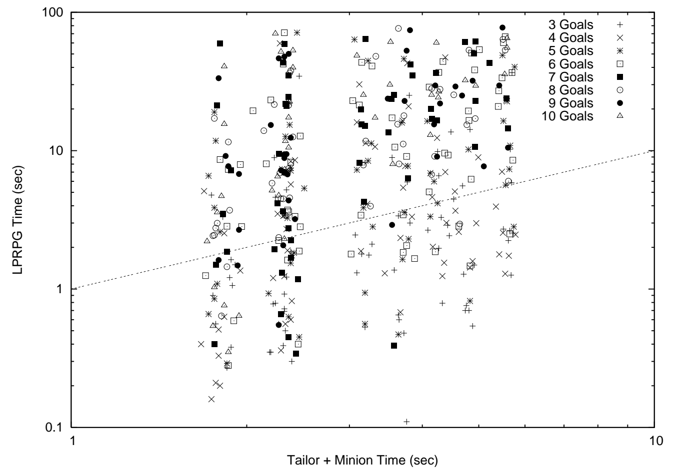


(f) Number of instances that failed to solve by LPRPG and Tailor + Minion

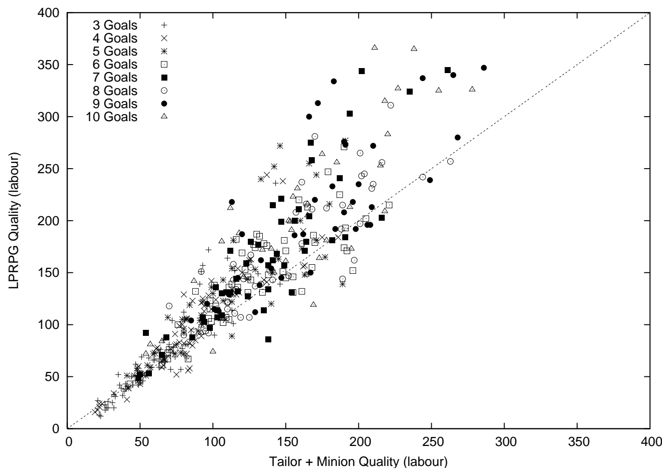
Figure 1: Comparisons between LPRPG and our constraint model translated by Tailor and running on the Minion constraint solver. All graphs are partitioned by the number of cities in the problem instance. For each size of city, there are 80 instances.



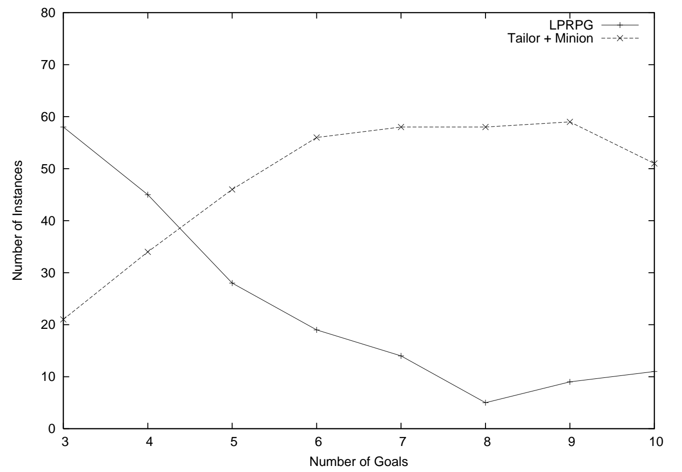
(a) Tailor + Minion vs. LPRPG: Time (sec). Results partitioned by number of goals.



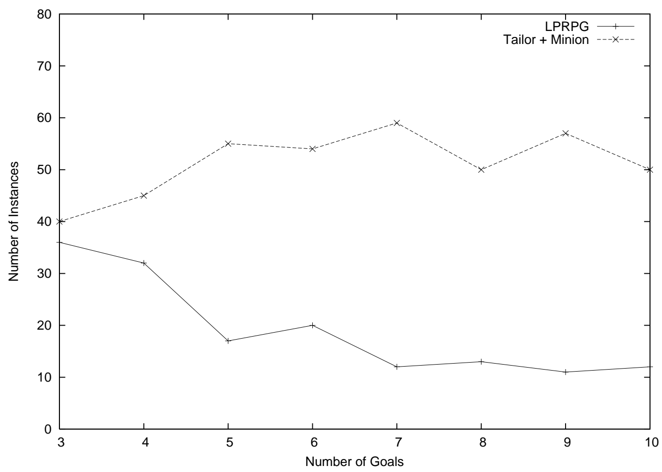
(b) Tailor + Minion vs. LPRPG: Time (sec). Results partitioned by number of goals. (more detailed view)



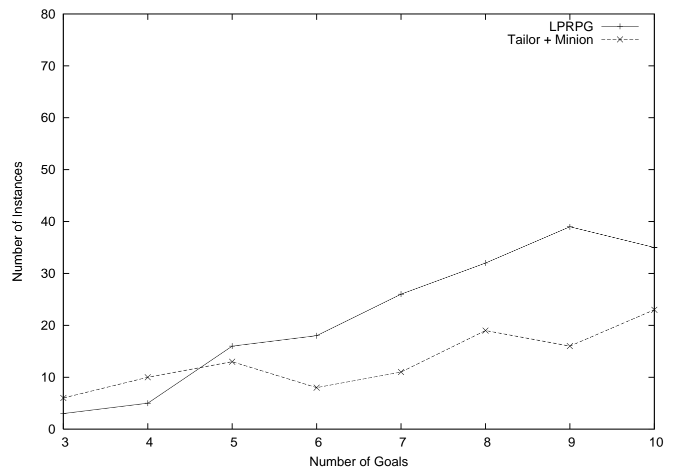
(c) Tailor + Minion vs. LPRPG: Quality (labour). Results partitioned by number of goals.



(d) Number of instances solved faster by LPRPG and Tailor + Minion



(e) Number of instances solved to a higher quality by LPRPG and Tailor + Minion



(f) Number of instances that failed to solve by LPRPG and Tailor + Minion

Figure 2: Comparisons between LPRPG and our constraint model translated by Tailor and running on the Minion constraint solver. All graphs are partitioned by the number of goals in the problem instance. For each number of goals, there are 80 instances.

There is a clear stratification in the performance of Tailor + Minion. The more detailed view, given in Figure 1(b), shows that the solve time for Tailor + Minion is heavily dependent on the number of locations in a given instance. This is clearly not the case for LPRPG, which displays much greater variance in the solve time for a given number of locations. Translation time dominates the time taken for Tailor + Minion to solve. Adding a new city to an instance increases the size of the model, since most of our constraints are quantified over all cities. It is this increase in model size that causes the stratification of the Minion + Tailor times.

Neither number of location nor number of goals seems to singularly dominate the time performance of LPRPG. Figure 2(b) shows the time performances with respect to the number of goals. There does appear to be a loose relationship between a combination of the number of goals and number of locations (for example, the only instances with 10 goals that are solved faster by LPRPG have just three locations).

Figure 1(d) shows that with respect to the number of cities, Tailor + Minion consistently solves faster than LPRPG. The performance of LPRPG deteriorates as the number of cities is increased. This suggests that the the constraint model provides more scalability, at least in terms of number of cities. Figure 2(d) shows that the constraint encoding is also more scalable with respect to the number of goals. However, for three and four goal problems, LPRPG typically performs better than the constraint encoding. For large numbers of goals, the performance of LPRPG degrades significantly compared to the constraint encoding.

Quality Performance Figure 1(c) and Figure 2(c) show the comparative metric performance of each solver. It should be emphasised that neither of the two planners optimises plans; both simply return the first plan found. Tailor+Minion solves many more instances to a higher quality than LPRPG. However, many plans are of similar quality and since neither planner is explicitly optimising, the results are encouraging for both technologies.

Figure 1(e) and Figure 2(e) show that as both number of cities and number of goals are varied, the constraint encoding provides higher quality solutions than LPRPG. In the case of the number of goals (Figure 2(e)) the number of times Tailor + Minion provides better metric performance than LPRPG increases starkly at five goals, and is then maintained at a high level.

Failures Figure 1(f) and Figure 2(f) show how many times each solver failed to solve instances in the 90 second limit. Typically, LPRPG was solving less instances than Tailor + Minion. An interesting, unexplained, fact is that when Minion solves an instance, it does so very quickly ($<$ 1 second) as Tailor dominates the time performance. However, there are instances with only three locations that are unsolved by Minion. These instances remain unsolved at a 10 minute time limit. These instance are clearly worthy of further study in order to ascertain if there are particular structural qualities that degrade the usual strong performance of Minion.

For both number of goals and number of locations Tailor + Minion is shown to be more scalable than LPRPG. As either variable is increased, the difference between the the number of failed instances of the two approaches tends to diverge.

Future Work

For future work, we would like to extend transportation to trains and ships in the constraint model. We would also like to enhance our problem representation by adding implied constraints to improve propagation. Furthermore, adding symmetry breaking could gain a more competitive model when minimising the total labour (and hence searching over the whole search space). We would also like to examine the cases in which our model fails to solve instances.

Conclusions

We have presented a constraint model for the Settlers planning domain. This model is preliminary as it does not model the entire problem: only transportation by cart is possible. However, the empirical analysis demonstrates that the constraint-based approach has potential and is worthy of both extension and further analysis.

We feel that the empirical analysis provides cheer for both the constraint programming and planning communities. Constraint programming has been shown to provide excellent performance in solving a complex planning problem. The planning model is arguably more intuitive than the constraint model, and LPRPG also performs strongly.

Acknowledgements. Andrea Rendl is supported by a DOC fFORTE scholarship from the Austrian Academy of Science and UK EPSRC grant EP/D030145/1.

References

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. A hybrid relaxed planning graph-lp heuristic for numeric planning domains. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 08)*.

Fox, M., and Long, D. 2003. Pddl2.1: An extension of pddl for expressing temporal planning domains. *Journal of AI Research* 20:61–124.

Gent, I. P.; Jefferson, C.; and Miguel, I. 2006. Minion: A fast scalable constraint solver. In Brewka, G.; Coradeschi, S.; Perini, A.; and Traverso, P., eds., *ECAI*, 98–102. IOS Press.

Gent, I. P.; Miguel, I.; and Rendl, A. 2007. Tailoring solver-independent constraint models: A case study with essence' and minion. In Miguel, I., and Ruml, W., eds., *SARA*, volume 4612 of *Lecture Notes in Computer Science*, 184–199. Springer.

Jefferson, C.; Miguel, A.; Miguel, I.; and Tarim, A. 2006. Modelling and solving english peg solitaire. *Computers and Operations Research* 33(10):2935–2959.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of AI Research* 20:1–59.

Pralet, C., and Verfaillie, G. 2008. Using constraint networks on timelines to model and solve planning and scheduling problems. In *International Conference on Automated Planning and Scheduling (ICAPS 2008)*.