

Kangaroo: An Efficient Constraint-Based Local Search System Using Lazy Propagation

M.A.Hakim Newton^{1,2}, Duc Nghia Pham^{1,2}, Abdul Sattar^{1,2}, and Michael Maher^{1,3,4}

¹ National ICT Australia (NICTA) Ltd.

² Institute for Integrated and Intelligent Systems, Griffith University

³ School of Computer Science and Engineering, University of New South Wales

⁴ Reasoning Research Institute, Sydney

Abstract. In this paper, we introduce Kangaroo, a constraint-based local search system. While existing systems such as Comet maintain invariants after every move, Kangaroo adopts a lazy strategy, updating invariants only when they are needed. Our empirical evaluation shows that Kangaroo consistently has a smaller memory footprint than Comet, and is usually significantly faster.

1 Introduction

Constraint-based local search (CBLs) has been quite successful in solving problems that prove difficult for constraint solvers based on constructive search. Unfortunately, there is little published work on existing implementations that led to this success – most work has addressed language features [12, 7, 8] and applications [5, 2]. In this paper we present Kangaroo, a new constraint-based local search system and expose key details of its implementation. We compare it with Comet, which has been the state-of-the-art in CBLs for several years.

Kangaroo differs from Comet in several aspects: it currently provides a C++ library, not a separate language; it employs a lazy strategy for updating invariants; it uses well-supported simulation to explore neighbourhoods, rather than directly using invariants as Comet seems to; and data structures are encapsulated at the system level instead of at the object level as Comet does. Nevertheless, Kangaroo provides many of the capabilities of Comet and is also very efficient. On a benchmark of well-known problems Kangaroo more frequently solves problems than Comet, and usually solves them faster. It also consistently uses about half the memory footprint of Comet.

The rest of the paper is organised as follows: Section 2 outlines constraint-based local search; Section 3 introduces Kangaroo terminology; Section 4 describes the Kangaroo system in detail; Section 5 discusses the experimental results and analyses; and finally, Section 6 presents conclusions and future work.

2 Constraint-based Local Search

Constraint-based local search is based on a view of constraint satisfaction as an optimization problem. With every constraint c , there is a violation metric

function μ_c that maps variable assignments¹ θ to non-negative numbers. We require of a violation metric only that $\mu_c(\theta) = 0$ iff c is satisfied by θ . For a set of constraints \mathcal{C} , constraint satisfaction is reformulated as the minimisation problem: $\text{minimize}_{\theta} \mu_{\mathcal{C}}(\theta)$ where $\mu_{\mathcal{C}}(\theta) = \sum_{c \in \mathcal{C}} w_c \mu_c(\theta)$ and the weights w_c (often $w_c = 1$) are to guide the search. If the minimum value of $\mu_{\mathcal{C}}$ is 0 then \mathcal{C} is satisfiable by the assignment that produces this value. If the minimum is non-zero then \mathcal{C} is unsatisfiable, but the assignment that produces the minimum achieves a “best” partial solution of \mathcal{C} .

CBLS uses local search to try to solve this minimization problem. There are many variants of local search, but common to them all is a behaviour of moving from one variable assignment θ to another, in search of a better assignment, and the exploration of a neighbourhood \mathcal{N}_{θ} before selecting the next assignment. Since this behaviour is repeated continually, key to the performance of local search is the ability to sufficiently explore the neighbourhood and perform the move to the next variable assignment quickly.

Comet [6] provides two progressively higher level linguistic concepts to support the specification of these operations. *Invariants* are essentially equations $y = f(x_1, \dots, x_n)$ which are guaranteed to hold after each move; as the value of the expression $f(x_1, \dots, x_n)$ is changed by a move, the value of y is revised. Note that such an invariant induces a *dependency* of y on x_1, \dots, x_n and, transitively, y depends on the terms that any x_i depends on. *Differentiable constraints* associate, with each constraint c , several methods that are often implemented with many invariants; they provide a way to inspect the effect of neighbouring assignments on c and they support an abbreviated exploration of the neighbourhood, by providing an estimate of *gradients*: the amount that the violation measure might change by changing the value of a given variable. Objective functions, and expressions in general, can be differentiable.

The invariants and differentiable objects as discussed above provide certain guarantees, but the job of actually keeping the guarantees is performed by a CBLS system. Invariants are implemented in the Comet system by a two-phase algorithm based on a relatively standard approach for one-way constraints [1]: in a planning phase the invariants are topologically ordered according to the dependencies between them, and then the execution phase propagates the one-way constraints, respecting this order. Planning ensures that each invariant is propagated at most once in each move. In terms of Fig. 1, this is a bottom-up execution, where a change in a problem variable such as *Queens*[k] is propagated through all invariants possibly affected by the change. This approach clearly keeps the guarantees.

An alternative approach is to perform a propagation only when it is needed. When a top-level term’s value is needed (for example, the value of the objective function), those invariants that it depends on and are out-of-date are visited by a (top-down) depth-first search and recursively, when all children of a node have been visited, and are up-to-date, the propagation for the invariant at that node is executed. If an invariant’s value is unchanged by its execution, we can

¹ A variable assignment maps each variable to a value in its domain.

avoid useless propagations. This approach requires that when a move is made all invariants that might be affected are marked as out-of-date before beginning evaluation. This approach, which is called mark-sweep, is also well-established [9]. It also keeps the guarantees. Kangaroo employs a variant of this approach.

There are several languages, toolkits and libraries supporting local search, including constraint-based local search. Local search frameworks or libraries such as HOTFRAME [3] and EasyLocal++ [4] focus on flexibility, rather than efficiency. They do not seem to support the light-weight inspection/exploration of neighbourhoods. Nareyek’s system DragonBreath [11] is not described directly; it appears to require explicit treatment of incrementality (no invariants) and a devolved search strategy where, at each step, a constraint is delegated to choose a move. Localizer [10] is a precursor to Comet that incorporated invariants but not differentiability. The invariant library of the iOpt toolkit [14] is similar to Localizer in many ways, except it is a Java toolkit, rather than a language, and it employs a mark-sweep approach to invariants, rather than topological ordering. Among SAT solvers employing local search only [13] employs a dependency structure similar to CBLs systems, and it needs only a simple update strategy because each move is a single flip. Unfortunately, most of the published literature focuses on language features, APIs, etc. and their use in applications, rather than describe details of their implementation. We believe this is the first paper on CBLs with an implementation focus.

3 Kangaroo Terminology

We introduce some terminology and notation for discussing Kangaroo. Given a CBLs problem, there is a set \mathcal{K} of constants and a set \mathcal{V} of problem variables. Every variable v is assumed to have a finite domain of values it can take. The basic element of computation in Kangaroo is a *term*, denoted by τ . If a term τ has the form $f(p_1, \dots, p_n)$, we refer to each p_i as a *parameter* and define $P(\tau) = \{p_1, \dots, p_n\}$. Terms other than variables and constants are referred to as *dependent* terms. Some dependent terms may be known to have values that are independent of the values of any variables and hence will not need to be repeatedly evaluated; the remaining dependent terms are *computable*. A term is *updatable* if it is a variable or is computable. A *root* term is a term on which no term depends. Γ is the set of root terms. For each term τ , $D(\tau)$ denotes the set of terms that are *directly dependent* on τ .

After a move, the values of some terms may be explicitly required; these are called *requisite* terms. The set of requisite terms is denoted by \mathcal{R} . A computable term τ is called an *enforced* term if τ is a requisite term, or some requisite term depends on τ . All other computable terms are called *deferred* terms; their recomputations are deferred until they become enforced terms, or an explicit *one-shot* request is made (e.g. by the search algorithm).

An assignment θ maps each variable v to a value in its domain. A *partial assignment* maps only some variables to values. We write $v \in \theta$ if θ maps v to a value. In the running of Kangaroo, a committed assignment is an assignment θ that describes the current node of the local search. Any neighbouring assignment

θ' of the committed assignment θ can be described by θ and a partial assignment ϑ specifying the new values that some variables take. That is, a partial assignment specifies a possible move from the committed assignment θ to θ' where $\theta'(v) = \text{if } v \in \vartheta \text{ then } \vartheta(v) \text{ else } \theta(v)$. Thus, a local search process that runs for N moves can be viewed as a sequence $\langle \theta_0, \vartheta_1, \dots, \vartheta_N \rangle$ of an assignment followed by a number of partial assignments.

Given a partial assignment ϑ , an updatable term τ is called a *candidate* term if τ is a variable in ϑ or depends on such a variable. These are the terms that *might* be recomputed if the move described by ϑ is made. Given ϑ and a computable term τ , the *candidate* parameters are the parameters of τ that are candidate terms for ϑ ; we denote the set of candidate parameters by $P_c(\tau, \vartheta)$.

Assume that a deferred term σ is last recomputed at iteration i and the latest iteration completed is j where $0 \leq i < j \leq N$. The *postponed* parameters $P_p(\sigma)$ are those parameters of σ that cause σ to recompute. We define them recursively as follows. A postponed parameter for σ wrt iterations $i \dots j$ is a parameter of σ that either (i) has actually changed value in an iteration between i and j (inclusive), or (ii) is a deferred term that has one or more postponed parameters wrt iterations $i \dots j$. A deferred term σ is said to be *out-of-date* at iteration j , if $P_p(\sigma)$ is non-empty; otherwise, the term is *up-to-date*².

4 Kangaroo System

The Kangaroo architecture as depicted in Fig. 1 has two components: Representation Component (RC) and Exploration Component (EC). The RC allows description of a given CBLs problem in a declarative way while the EC allows specification of the search algorithms and the heuristics/meta-heuristics. The RC consists of two units: Assignment Unit (AU) and Propagation Unit (PU). The AU holds all variables and constants, supports definition of the constants, and allows run-time assignments of new values to the variables by the EC. The PU holds all the dependent terms and provides the EC with the up-to-date values of the updatable terms under assignment of any new values to the variables. Overall, the RC is responsible for running an assignment-propagation cycle for each iteration of the search algorithm in the EC. On the other hand, the EC is responsible for any run-time decision during search, including selections of best or least-bad assignments and restarting the search at plateaus.

The Kangaroo architecture has a number of notable features: **i)** on-demand recomputation of the updatable terms using lazy propagation of the given assignments, **ii)** specialised incremental execution to compute aggregate formula, **iii)** specialised incremental simulation boosted by caching when ranges of values are tried but the variables remain the same or different sets of variables are tried but the sets differ by just one variable, **iv)** type-independent linearly ordered scalar view of domain values to allow unified selection over variables or values, **v)** low-level memory management to obtain fast and compact data structures, **vi)** data encapsulation at the system level and unencapsulation at

² Note that ‘out-of-date’ refers to terms that are *potentially* out-of-date; it may be that a term has the correct value but it is still regarded as out-of-date.

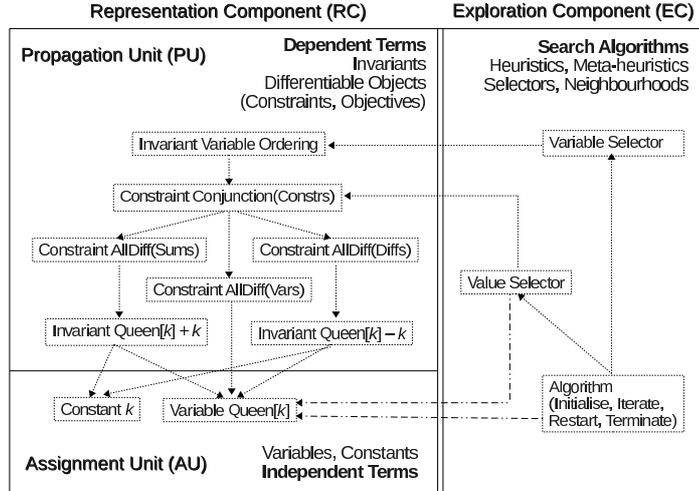


Fig. 1. The Kangaroo architecture and the view of queens problem.

the object level to allow array-based data storage and faster access through non-virtual function calls, and **vii)** implementation as a C++ library to allow easier integration with large systems and also to get the complete support of a very well-known programming language and its powerful compiler.

Execution vs Simulation

In Kangaroo, *computations* are performed in two different modes: execution and simulation. In the *execution mode*, variables can be assigned new values by accessing the AU. The PU then propagates the effect by recomputing the requisite terms in a top-down fashion. In the *simulation mode*, the AU and PU work in the same way, but the updates are temporary and the effects are not committed. The simulation mode thus allows a neighbourhood-exploration algorithm to investigate the effect of potential assignments and then to elect an assignment for the next execution. Kangaroo has two separate sets of data structures throughout the entire system to run these two modes in an interleaving fashion. The interleaving facility allows execution of some deferred terms (with the last committed assignment), if required by the EC, in between performing two simulation runs.

Note that the simulation in Kangaroo and that in Comet have significantly different purposes. Comet uses simulation (i.e. the *lookahead* method) only in rare cases where moves are very complex and cannot be evaluated incrementally or are not supported by the differentiable API. The implementation of simulation in Comet performs execution for a given assignment, obtains the result, and then reverses the effect of the execution; the simulation is thus typically significantly slow [6]. In Kangaroo, simulation is at the centre of neighbourhood investigation. Our simulation efficiently performs light-weight incremental computation for a given assignment and does not require reverting to a previous state.

Incremental Computation

For some invariants, it is possible to perform a computation without accessing all the parameters. This is particularly attractive when there are many parameters and only a few are modified. Consider the invariant $S = \sum_{x \in X} x$. When all parameters in X are assigned values for the first time, S is computed *anew*, that is, from scratch. In later computation, when only a subset $X' \subset X$ of all parameters are changed, then the new value of S can be obtained as $S.\text{oldval} + \sum_{x \in X'} (x.\text{newval} - x.\text{oldval})$. This type of computation is referred to as *incremental computation* and involves only the parameters that are changed. We formulate this as an *undo* operation with the old value of x and a *redo* operation with the new value of x , for each modified parameter x . Incremental computations are useful in both execution and simulation modes.

Top-Down Computation

Kangaroo performs computation of a given term in a top-down fashion. In the execution mode, when the effect of a partial assignment ϑ is to be propagated, the PU performs a depth-first recursive exploration, starting from each requisite term. During this exploration, each visited computable term τ first determines the set of candidate parameters $P_c(\tau, \vartheta)$. If any candidate parameter p requires recomputation, the depth-first exploration moves to that p . Recomputation of a variable $v \in P_c(\tau, \vartheta)$ involves assigning it the value $\vartheta(v)$. When all candidate parameters are explored and recomputed, τ is recalculated from only those candidate parameters that are actually modified.

This process is complicated when the given term τ is a deferred term. This is tackled by maintaining a list of postponed parameters for each deferred term. Whenever a term τ' has *actual* modification, it notifies each of its deferred dependents σ to add τ' to $P_p(\sigma)$. This denotes σ might have a change because of the modification of τ' . Consequently, each deferred dependent σ recursively notifies its deferred dependents of its *potential* change, resulting in it being added in their list of postponed parameters. During recomputation of any deferred term, its postponed parameters are explored and recomputed.

The recomputation of τ is further complicated when an incremental computation is to deal with the actual modification of a parameter $p \in P_p(\tau)$ during the iterations when τ was not recomputed. This is because an incremental computation, to ensure correctness, requires undoing the effect of p 's value that was used in τ 's last recomputation. During next recomputation of τ , of course we would need to redo for the latest value of p . Nevertheless, the undo operation is performed when p is modified for the first time between these two recomputations. An undo operation also adds p to τ 's list of *undone* parameters. This list is used in recomputation of τ to determine which parameters need only redo.

In the simulation mode, recomputations are done in the same top-down recursive way as is done in the execution mode. In local search, each execution normally follows a range of simulations that are closely related. For example, during exploration of values for a given variable, the variable remains the same for all simulation runs; or during exploration of pair-wise swapping between

variables, many potential pairs have one shared variable. Kangaroo exploits this close relationship by caching once and reusing in all the simulations: the candidate parameters, computations performed for postponed parameters, and even undo operations for the candidate parameters.

It is worth emphasizing here that our lazy top-down approach is different from the mark-sweep approach taken in [9] for incremental attribute evaluation. For a given assignment, the mark-sweep approach marks all candidate terms. In our case, we need not mark the enforced terms; these terms will be explored by the top-down traversal algorithm starting from the requisite terms. We therefore mark only the deferred terms. An actually modified term marks only its deferred dependents, and consequently these marked deferred dependents recursively mark their deferred dependents.

Data Structures

While implementing Kangaroo, we found that the choice of certain representations and data structures are key to performance. Although the differential benefits are not accounted for each individual choices made, their combination appears to have significant impact on both speed and memory.

1. **System Clocks:** At the system level, there are two separate assignments ϑ_e and ϑ_s respectively for execution and simulation; to denote the time of change in the assignments, there are two clocks T'_e and T'_s . Also, there are two more clocks T_e and T_s to denote the current propagation cycle in the execution and simulation modes. Each term tracks when it was last computed, using timestamps T_e and T_s . Clocks T_e and T_s help avoid recomputations within the same cycle while clocks T'_e and T'_s help detect need for recomputation of the intra-term caches.
2. **Data Buffers:** Each term has three sets of data buffers – B_c , B_p , and B_n to hold respectively the result of currently completed execution, that of immediately previous execution, and that of the currently completed simulation (i.e. potentially the next execution). Simulations are always subject to the currently completed execution; if there is no such simulation, then $B_n = B_c$. The $\langle B_p, B_c \rangle$ pair is used in incremental execution while the $\langle B_c, B_n \rangle$ pair in incremental simulation. Note that each term also has two more boolean buffers to hold the values of $(B_c \neq B_p)$ and $(B_c \neq B_n)$, which saves repeated checking for actual change.
3. **Data Tables:** The data buffers described above are stored in array-based tables within the system and accessed using term indexes. This enables efficient access to the data without making costlier virtual function calls. However, this is a violation of data encapsulation at the term level; in the object-oriented paradigm, data is normally stored within the object. We deal with this by assigning such responsibilities to the system level.
4. **Term Lists:** Kangaroo keeps a list of root terms Γ and another list of requisite terms \mathcal{R} . The root terms are executed during initialisation or restarting of the system while the requisite terms are executed in incremental execution

phase. The simulation mode does not require these lists as recomputations in this mode are performed only based on instant demand.

5. **Term Representations:** Each updatable term τ stores the dependents $D(\tau)$ and the deferred dependents $D_d(\tau)$. The key records for each dependent term δ are parameters $P(\delta)$, involved variables³ $\mathcal{V}(\delta)$, postponed parameters $P_p(\delta)$, undone parameters $P_u(\delta)$, candidate parameters $P_c(\delta, \vartheta_e)$ for ϑ_e , candidate parameters $P_c(\delta, \vartheta_s)$ for ϑ_s , and, for each $v \in \mathcal{V}(\delta)$, the set $P_d(\delta, v)$ of updatable parameters of δ that depend on v . $P_d(\delta, v)$ is recomputed statically and is used to compute candidate parameters of δ using a simple set-union.
6. **Value Representations:** To facilitate implementation of type-independent selectors and search algorithms, Kangaroo takes the unified approach of using linear indexes to denote domain values. This eliminates costlier navigation on the value space through virtual function calls. For discrete variables, such indexes are a natural choice. For continuous variables, the assumption is to have a step size that allows discretisation of the value space.
7. **Customised Data Structures:** To obtain efficiency and better memory usage, we implemented customised data structures for arrays, heaps, hash-sets, and hash-maps. We also have timestamp-based arrays of flags to efficiently perform set-unions, marking and unmarking operations.

Recomputations

Both in execution and simulation modes, recomputation of a computable term could be done anew or incrementally. During anew recomputation, each computable term first invokes anew recomputation of all its updatable parameters, and then recalculates itself from scratch without requiring results of the previous iterations. Anew recomputations are needed when all the variables are initialised, and also when a complete restart is required. In this paper, we mainly describe the procedures required in incremental and deferred recomputations. Refer to Fig. 2 for pseudocode of the procedures.

1. **Potential Recomputations:** Initially all computable terms are deferred. When such a term is marked as a requisite term, its descendant computable terms recursively become enforced. For conditional terms such as *if-then-else*, only the conditional term becomes enforced; depending on the condition, the *then* or *else* term, if not enforced w.r.t. other terms, is executed only on one-shot request mode. Nevertheless, when a term is no longer a requisite term, its descendant enforced terms are notified. In this process, a computable term, that is neither a requisite term itself nor an enforced term w.r.t. another requisite term, becomes deferred again. Computation of deferred terms is allowed only after the first iteration. At the end of first iteration, each deferred term executes `notifyRecomp` to notify each of its parameters that it

³ For each dependent term δ , the set $\mathcal{V}(\delta)$ of *involved* variables of δ is the set of variables that δ depends on.

```

proc  $\tau$ .execIncr
  //  $\tau$  is computable.
  if  $\tau.T_e = T_e$  then return;
  if  $|P_u(\tau)| = 0$  then  $\tau.B_p = \tau.B_c$ ;
  if  $\tau.T'_e \neq T'_e$  then compute  $P_c(\tau, \vartheta_e)$ ;
  foreach  $p \in P_p(\tau) \cup P_c(\tau, \vartheta_e)$  do
    if  $p \notin \mathcal{V}$  then  $p$ .execIncr();
    if  $p \notin P_u(\tau) \wedge p.B_p \neq p.B_c$  then
      undo( $p.B_p$ ), redo( $p.B_c$ );
      //  $\text{sum}.B_c += p.B_c - p.B_p$ 
  foreach  $p \in P_u(\tau)$  do
    redo( $p.B_c$ ) //  $\text{sum}.B_c += p.B_c$ ;
  if  $\tau$ .Deferred then
    foreach  $p \in P_p(\tau) \cup P_u(\tau)$  do
       $p$ .notifyRecomp( $\tau$ );
  clear  $P_p(\tau)$  and  $P_u(\tau)$ ;
  if  $\tau.B_c \neq \tau.B_p$  then
    foreach  $\sigma \in D_d(\tau)$  do
       $\sigma$ .notifyChange( $\tau, \text{true}$ );
    clear  $D_d(\tau)$ ,
   $\tau.T_e = T_e$ ;

proc  $\sigma$ .notifyChange( $p$ , isActualChange)
  //  $\sigma$  is deferred.
  if isActualChange then
    if  $p \in P_u(\sigma)$  then return;
     $P_u(\sigma) = P_u(\sigma) \cup \{p\}$ 
    if  $|P_u(\sigma)| = 1$  then
       $\sigma.B_p = \sigma.B_c$ ;
      undo( $p.B_p$ ) //  $\text{sum}.B_c -= p.B_p$ ;
    else
      if  $p \in P_p(\sigma)$  then return;
       $P_p(\sigma) = P_p(\sigma) \cup \{p\}$ ;
  foreach  $\sigma' \in D_d(\sigma)$  do
     $\sigma'$ .notifyChange( $\sigma$ , false);

proc  $\tau$ .notifyRecomp( $\sigma$ )
  //  $\tau$  is updatable.
   $D_d(\tau) = D_d(\tau) \cup \{\sigma\}$ 

```

```

proc  $\tau$ .simulIncr
  //  $\tau$  is computable.
  if  $\tau.T_s = T_s$  then return;
  if  $\tau.T'_s \neq T'_s$  then
    compute  $P_c(\tau, \vartheta_s)$ ;
    // Cache =  $\tau.B_c$ 
    foreach  $p \in P_p(\tau) \setminus P_c(\tau, \vartheta_s)$  do
      if  $p \notin \mathcal{V}$  then  $p$ .simulIncr();
      if  $p \notin P_u(\tau) \wedge p.B_n \neq p.B_c$ 
        then
          undo( $p.B_c$ ), redo( $p.B_n$ );
          // Cache +=  $p.B_n - p.B_c$ 
    foreach  $p \in P_c(\tau, \vartheta_s) \setminus P_u(\tau)$  do
      undo( $p.B_c$ )
      // Cache -=  $p.B_c$ 
    foreach  $p \in P_u(\tau) \setminus P_c(\tau, \vartheta_s)$  do
      redo( $p.B_c$ ); // Cache +=  $p.B_c$ 
  //  $\tau.B_n = \text{Cache}$ 
  foreach  $p \in P_c(\tau, \vartheta_s)$  do
    if  $p \notin \mathcal{V}$  then  $p$ .simulIncr();
    redo( $p.B_n$ ) //  $\tau.B_n += p.B_n$ 
   $\tau.T_s = T_s$ ;

```

```

proc  $v$ .execIncr
  //  $v$  is a variable.
  if  $v.T_e = T_e$  then return;
   $v.B_p = v.B_c$ ,  $v.B_c = \vartheta_e(v)$ ;
  if  $v.B_c \neq v.B_p$  then
    foreach  $\sigma \in D_d(v)$  do
       $\sigma$ .notifyChange( $v$ , true);
    clear  $D_d(v)$ ;
   $v.T_e = T_e$ ;

```

```

proc  $v$ .simulIncr
  //  $v$  is a variable.
  if  $v.T_s = T_s$  then return;
   $v.B_n = \vartheta_s(v)$ ,  $v.T_s = T_s$ ;

```

Fig. 2. The Kangaroo algorithms using summation as an example.

has been computed. Each parameter p adds the deferred term to its $D_d(p)$ so that it can later notify its actual or potential changes to the deferred term.

2. **Incremental Execution:** During incremental execution, Procedure `execIncr` first checks the execution clock to determine whether execution has already been performed in the current cycle. During incremental execution of a variable v , B_c is saved in B_p and the new value $\vartheta_e(v)$ is assigned to B_c . For computable terms, B_c is saved in B_p , if there is no undone parameter; if there is any, `notifyChange` has already done this. Next, the candidate parameters are recomputed, if T'_e has changed. All postponed and candidate parameters are then executed incrementally, but calculation is performed for the given term only using those parameters that are not modified; the calculations involve undoing with B_p and redoing with B_c . Next, for undone parameters, only redo operations are performed; undo operations have already been performed in `notifyChange`. If the computable term τ is a deferred term, it then executes `notifyRecomp` to notify each parameter in $P_p(\tau) \cup P_u(\tau)$ that τ has been recomputed. The computable term then clears the lists of undone and postponed parameters. For both computable terms and variables, if the term

being executed has a new value, it notifies this to its deferred dependents by executing their `notifyChange` and then clears the list. During execution of `notifyChange`, the deferred dependents need to perform undo operations and then recursively notify each of their deferred dependents about their potential modification. The execution of a term ends by updating its execution timestamp T_e .

3. **Incremental Simulation:** Procedure `simullncr` performs incremental simulation. For variables, incremental simulation involves only assigning new values. For computable terms, candidate parameters are to be determined first, if T'_s has changed. For a new T'_s , the cache inside the term also needs recomputation. Cache recomputation involves incremental simulation of all non-candidate postponed parameters and both undoing and redoing for the unmodified parameters. It also requires undoing for the candidate parameters (excluding undone parameters) and redoing for the undone parameters (excluding candidate parameters). Given the same assignment variables, each simulation run reuses the cached result and then incrementally simulates the candidate terms and performs redo for them. Notice that the cache mentioned above could be split into two parts: one for the postponed parameters, and the other for the candidate parameters. The cache for candidate parameters cannot be reused in the simulations with different assignment variables, but the cache for postponed parameters could still be reused. The use of cache could further be extended over assignments that have overlapping variables (e.g. a range of variable-value swappings with one variable in common).

Search Controls

Kangaroo currently implements a number of variable selectors, value selectors, and swap selectors. To support these selectors, especially the variable selectors, Kangaroo implements a specific type of invariant, called *variable ordering*. Such invariant utilises priority queues to maintain candidate variables for selection. In addition, the taboo heuristic is integrated into the *variable ordering* invariant, enabling Kangaroo to ignore calculation of tabooed variables. This is different from the way taboo variables are handled in Comet.

5 Experiments

We compared Kangaroo and Comet on a set of benchmarks from CSPLib: a problem library for constraints (www.csplib.org). We selected benchmarks that use representative elements of key features such as selectors, invariants, constraints, and taboo and restart heuristics. The benchmarks are: all interval series, golomb ruler, graph coloring, magic square, social golfer, and vessel loading. We also included the well-known n -queens problem. It is worth mentioning here that we only used satisfaction versions of the benchmarks, although some of them in CSPLib are optimisation problems. There was insufficient time to run optimisation benchmarks; we expect similar performance for those benchmarks.

We briefly describe the benchmarks, and also the constraint model and search algorithm to solve each benchmark. For detailed description of these benchmarks, please refer to CSPLib.

1. **all interval series:** The problem is to find a permutation $s = (s_1, \dots, s_n)$ of $Z_n = \{0, 1, \dots, n-1\}$ such that $V = (|s_2 - s_1|, |s_3 - s_2|, \dots, |s_n - s_{n-1}|)$ is a permutation of $Z_n - \{0\} = \{1, 2, \dots, n-1\}$. Problem instances are generated for $n = 10, 15, 20$, and 25 . The problem model includes an **AllDifferent** constraint on V . The search algorithm is based on swapping a pair of variables that leads to the minimum violation. The length of taboo on variables is 5.
2. **golomb ruler:** Given M and m , a Golomb ruler is defined as a set of m integers $0 = a_1 < a_2 < \dots < a_m \leq M$ such that the $m(m-1)/2$ differences $a_j - a_i$ ($1 \leq i < j \leq m$) are distinct. In the problem model, the domain of a variable is dynamically restricted using the values of its neighbours. Problem instances are generated for $4 \leq m \leq 11$ and M to be equal or close to the minimum M known for m . The problem model includes an **AllDifferent** constraint on the differences. The search algorithm is based on assigning a variable causing maximum violation with a value that results in the minimum violation (max/min search). Taboo length is 5.
3. **graph coloring:** Given a k colorable graph generated with n vertices and the probability of having any edge being p , the problem is to assign k colors to the vertices such that no two adjacent vertices get the same color. Problems instances were generated using the graph generator programs written by Joseph Culberson⁴ with parameter values $p = 0.5$, $k \in [3, 10]$, $n \in \{25, 50, 75, 100\}$ and 5 instances per setting.⁵ The problem model uses a not-equal constraint for each pair of adjacent vertexes. It uses max/min search. Taboo length is 5.
4. **magic square:** Given a square S of dimension $n \times n$, the problem is to assign values $\{1, \dots, n^2\}$ to the n^2 cells such that the sum of each row, column, and diagonal is equal to $C = n(n^2 + 1)/2$. The problem instances are generated for $n \in [10, 50]$ in step of 5. The search algorithm is based on swapping between a pair of variables that leads to the minimum violation. Taboo length is 5.
5. **n -queens:** Given a chess-board of dimension $n \times n$, put n queens on the board such that no two queens attack each other. Problems instances were generated for $n \in [1000, 50000]$. The problem model uses three **AllDifferent** constraints and an invariant on most violated queens (this is automatically maintained by the variable ordering invariant in Kangaroo – see Fig. 1). It uses max/min search.
6. **social golfer:** Given w weeks, g groups, and p persons per group, the problem is to find a golf playing schedule such that each person plays every week in a group with other persons, but no two persons fall in the same group more than once. Problems instances were generated for $w \in [3, 9]$, $g \in [3, 5]$, and $p \in [2, 5]$. The problem model uses **atmost** and **exactly** constraints. It uses max/min search that also incorporates random restarts in every given $r \in \{1000, 5000, 15000\}$ iterations. Taboo length is 5.

⁴ The generator can be downloaded at <http://webdocs.cs.ualberta.ca/~joe/Coloring/>

⁵ For the graph coloring problem, generating good benchmark instances was difficult: the generated instances were either too difficult or too easy for both systems. We included instances that are roughly at the capability horizon of Comet.

7. **vessel loading:** Given a vessel of size $L \times W$, a number of containers are to be placed on the vessel such that there is no overlapping among the containers. The size of each container k is $l_k \times w_k$ where $l_k, w_k \in [D_{\min}, D_{\max}]$. Problems are generated with $10 \leq W \leq L \leq 20$, $D_{\min} = 2$, $D_{\max} = 5$ and the total area covered by the containers is varied between 25%, 50%, and 75% of the vessel area. The problem model uses *if-then-else* to determine the length and width of the containers based on their orientations, and the constraints required to ensure non-overlappings of the containers use invariants that maintain disjunctions. It uses max/min search. Taboo length is 5.

We ran experiments on the NICTA (www.nicta.com.au) cluster machine with a maximum limit on the iteration-count. The cluster has a number of machines each equipped with 2 quad-core CPUs (Intel Xeon @2.0GHz, 6MB L2 Cache) and 16GB RAM (2GB per core), running Rocks OS (a Linux variant for cluster). For each benchmark, a number of solvable instances were created, varying the complexity by specifying the parameters. Both Comet and Kangaroo were run 100 times for each problem instance and the instance distribution was considered to be non-parametric. When specifying problems, we tried our best to keep the specifications for Comet and Kangaroo as close as possible both in the problem models and in the search algorithms (for example, see Fig. 3). The experimental results of Kangaroo and Comet are summarised in Table 1 and presented in details in Fig. 4.

For each benchmark, Table 1 reports the number of instances, the percentage success rate, the mean-of-median iteration-counts, the mean-of-median solution-time in seconds, and the mean-of-median memory usage in megabytes. The percentage success rate of a benchmark is calculated as the percentage of solved instances in that benchmark set. Here an instance is considered solved if its suc-

Comet.AllIntervalSeries	Kangaroo.AllIntervalSeries
<pre> range Size = 0..(n - 1); range Diff = 1..(n - 1); // Create n vars with a domain Size var{int} s[Size](m, Size); // Post an alldiff constraint on intervals S.post (alldifferent (all (i in Diff) abs(s[i] - s[i - 1])))); // Assign random permutation to s RandomPermutation distr (Size); forall(i in Size) s[i] := distr.get (); int tabu[Size] = 0; while(S.violations() > 0){ if it ≥ MaxIt then return; // Select two vars that, if swapped, // lead to the min violation delta selectMin(i in Size: tabu[i] ≤ it, j in Size: i < j ^ tabu[j] ≤ it) (S.getSwapDelta(s[i], s[j])) { s[i] := s[j]; // Swap the values tabu[i] = it + TabuLength; tabu[j] = it + TabuLength; } it = it + 1; } </pre>	<pre> defineSolver (Solver); // Set the tabu tenure: effectively tell Kangaroo // to automatically maintain the tabu heuristic setTabuLength (Solver, TabuLength); // Create n vars with domain [0, n - 1] forall(i in 0..n - 1) defineVar (Solver, s[i], 0, n - 1); // Create n - 1 intervals abs(s[i] - s[i - 1]) forall(i in 1..n - 1) defAbsDiff (Solver, v[i], s[i], s[i - 1]); // Post an alldiff(v[i], i in 1..n-1) constraint defAllDifferent (Solver, alldiffConstr, v); // Create a Selector: selects two non-tabu variables // leading to the min violation metric, if swapped defTabuMinSwapSel (Solver, Selector, alldiffConstr); assign a random permutation of [0..n-1] to s; while(alldiffConstr.violations() > 0){ if it ≥ MaxIt then return; run Selector to select a pair of vars (s[i], s[j]); swap the value of s[i] and s[j]; it = it + 1; } </pre>

Fig. 3. Problem specifications for Comet(left) and Kangaroo(right).

Instances	Kangaroo				Comet			
	success rate (%)	#iteration	CPU time (in secs)	memory (in MBs)	success rate (%)	#iteration	CPU time (in secs)	memory (in MBs)
all interval series (4)	75%	21,734	1.6	20	50%	504,465	65.1	42
golomb ruler (11)	91%	681,452	8.0	21	45%	not computed		42
graph coloring (20)	100%	774	0.0	22	50%	590	0.3	44
magic square (9)	100%	212	172.6	22	100%	213	103.3	43
n-queens (18)	100%	8,532	104.7	111	100%	8,597	140.9	293
social golfer (16)	88%	987,822	21.7	22	19%	not computed		47
vessel loading (45)	100%	212	0.0	24	62%	3,741,397	96.4	43

Table 1. Kangaroo and Comet comparison summary.

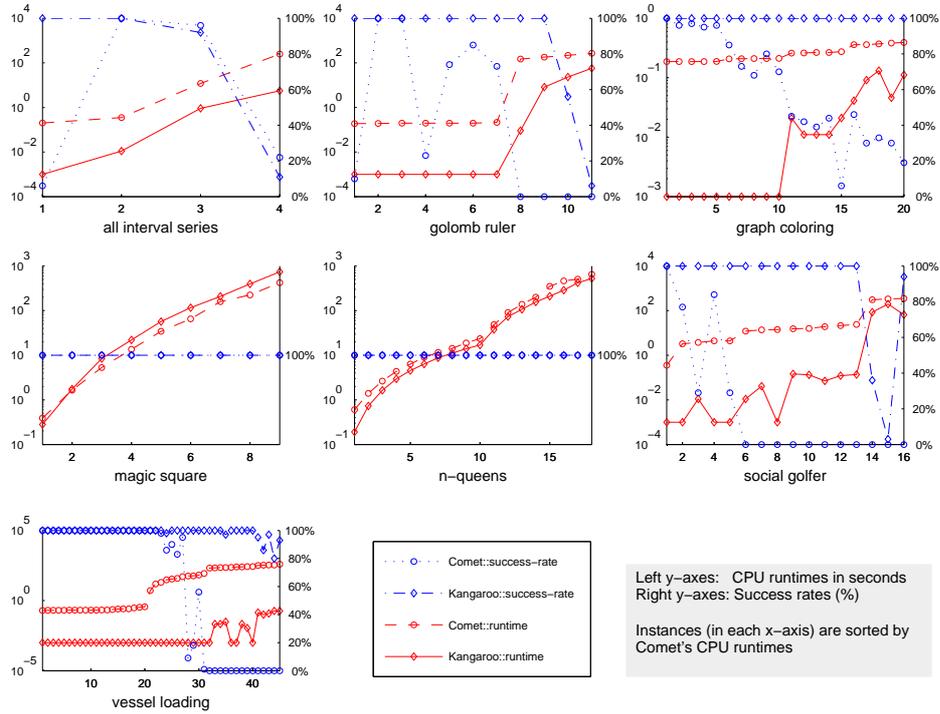


Fig. 4. Kangaroo vs Comet: a detailed comparison on success rate and runtime.

cess rate is at least 50%. Medians are taken over all runs of each instance while the means are taken over the medians of the instances. Memory statistics are based on both successful and unsuccessful runs. However, statistics on iteration-count and solution-time are only on successful runs and when success rate is at least 50%. For each benchmark, Fig. 4 graphically presents the % success rate and the median solution-times over the successful runs. Note, when the success rate for an instance is below 50%, the solution time plotted is not completely meaningful.

Overall, we found that in all of the above benchmarks, Kangaroo consistently uses less than 50% of the memory required by Comet. The success rate of Kangaroo is significantly higher in all of the benchmarks except magic square and n -queens, where both systems could solve all problem instances. In terms of solution times, Kangaroo significantly outperforms Comet, except in magic square where Comet performs better than Kangaroo.

The models and the search algorithms for Comet and Kangaroo are best matched in magic square and n -queens. This is reflected by the similar number of iterations required by the two systems to solve instances in these two benchmarks. The performance of Kangaroo is better in n -queens than that of Comet. This clearly shows the performance advantage of the Kangaroo architecture.

In magic square Comet performs better than Kangaroo because it uses a specialised `getSwapDelta` method to compute the effect of a swap. This specialisation helps in magic square where constraints are based on summation of variables. When two variables belonging to the same summation are swapped, the summation result remains unchanged. Thus, there is no need to compute the effect of such a swap. In general, this specialisation can be exploited for all invariants where the changes in swapping variables would nullify each other. By contrast, Kangaroo currently implements a generic `getSwapDelta` method that emulates swaps as assignments of two variables simultaneously. In other words, Kangaroo has to simulate the changes in both variables and then adds them to obtain the new summation result. In future, we plan to eliminate such redundant calculation from Kangaroo.

However, there are many cases where parameters of an invariant consists of complex functions and a swap between two variables is not necessarily a no-op for that invariant although it may involve both variables. In such cases, it is a must to use the generic `getSwapDelta` method to compute the effect of a swap. There are also cases where the benefit of a specialised swap is not much different compared to a generic swap. A strong evidence for such situations is demonstrated in the all interval series benchmark. Here, a fresh computation of $|s_k - s_{k-1}|$ is cheaper than its incremental computation. Nevertheless, the success rates of Comet and Kangaroo are significantly different in this benchmark. From the chart, it appears that Comet search does not progress on the problem instances with ($n = 10$). For the time being, we cannot speculate on reasons for that.

In graph coloring, golomb ruler, vessel loading, and social golfer, we found Comet to be performing very poorly while Kangaroo showed notable performance. We reiterate that the models and search algorithms for those benchmarks are semantically matched. This variation must be due to differences in implementation but, unfortunately, the implementation details of Comet are not available to us.

6 Conclusion and Future Work

We have presented Kangaroo and key details of its implementation. Empirical results show it to improve on Comet in both time and memory usage. In the future, we hope to release Kangaroo under an open source license. We plan to

provide a technical report with more details than could be presented in this paper, and hope to provide a FlatZinc interface, so that Zinc can be used as a frontend to the system.

Acknowledgements: NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *SODA*, pages 32–42, 1990.
2. Pham Q. Dung, Yves Deville, and Pascal van Hentenryck. Constraint-based local search for constrained optimum paths problems. In *CPAIOR*, pages 267–281, 2010.
3. Andreas Fink and Stefan Voss. Hotframe: A heuristic optimization framework. In D.L. Woodruff S. Voss, editor, *Optimization Software Class Libraries*, pages 81–154. Kluwer, 2002.
4. Luca Di Gaspero and Andrea Schaerf. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software — Practice & Experience*, 33(8):733–765, July 2003.
5. Pascal Van Hentenryck, Carleton Coffrin, and Boris Gutkovich. Constraint-based local search for the automatic generation of architectural tests. In *CP*, pages 787–801, 2009.
6. Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.
7. Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. *Constraints*, 10(2):137–157, 2005.
8. Pascal Van Hentenryck and Laurent Michel. Differentiable invariants. In *CP*, pages 604–619, 2006.
9. Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Trans. Program. Lang. Syst.*, 13(3):315–341, 1991.
10. Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1/2):43–84, 2000.
11. Alexander Nareyek. *Constraint-Based Agents*, volume 2062 of *Lecture Notes in Computer Science*. Springer, 2001.
12. Alexander Nareyek. Using global constraints for local search. In E. C. Freuder and R. J. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization*, pages 9–28. American Mathematical Society Publications, 2001.
13. Duc Nghia Pham, John Thornton, and Abdul Sattar. Building structure into local search for SAT. In *IJCAI*, pages 2359–2364, 2007.
14. Christos Voudouris, Raphaël Dorne, David Lesaint, and Anne Liret. iOpt: A software toolkit for heuristic search methods. In *CP*, pages 716–719, 2001.