Efficient Memoization for Dynamic Programming with Ad-Hoc Constraints

Joxan Jaffar **Andrew E. Santosa** **Răzvan Voicu**
School of Computing, National University of Singapore
Republic of Singapore 117590
{joxan,andrews,razvan}@comp.nus.edu.sg

Abstract

We address the problem of effective reuse of subproblem solutions in dynamic programming. In dynamic programming, a memoed solution of a subproblem can be reused for another if the latter's context is a special case of the former. Our objective is to generalize the context of the memoed subproblem such that more subproblems can be considered subcases and hence enhance reuse. Toward this goal we propose a generalization of context that 1) does not add better solutions than the subproblem's optimal, yet 2) requires that subsumed subproblems preserve the optimal solution. In addition, we also present a general technique to search for at most $k \geq 1$ optimal solutions. We provide experimental results on resource-constrained shortest path (RCSP) benchmarks and program's exact worst-case execution time (WCET) analysis.

Introduction

Dynamic programming is a widely used technique to solve combinatorial optimization problems exhibiting certain characteristics. One major characteristic of a dynamic program is the existence of overlapping subproblems. This allows for the reuse of a solution of a subproblem to solve another subproblem. Therefore, essential to a dynamic programming technique is the notion of memoization of subproblem solutions already solved. Essential to memoization is the *identification* of whether a solution can be reused or not. Each subproblem is executed in a *context*, which is an abstraction of the computation history so far. Whenever the same subproblem is encountered in a *similar* context, the previous solution can be reused.

A simple example of a problem that can be modeled as a dynamic program is the shortest path problem. Assuming an edge-weighted graph $(\mathcal{V}, \mathcal{E})$ of vertices $\mathcal{V}$ and edges $\mathcal{E}$, we distinguish two elements of $\mathcal{V}$ as the *source* and *destination*. A shortest path problem is finding a path from the source to destination such that the sum of the edge weights is minimal. As is usual in the literature this can be modeled using the following dynamic program, where $c_{v,w}$ denotes the weight of the edge (v,w).

$$f(X) = \textbf{if } \mathit{destination}(X) \textbf{ then } 0 \textbf{ else } \mathit{min}\{ f(Y) + c_{X,Y} | (X,Y) \in \mathcal{E} \}$$

Dynamic programming very efficiently executes this program using memoization, which results in a search tree size that is linear to $|\mathcal{V}|$. In the above formulation, the solution to a subproblem $f(v)$ is computed when the subproblem is encountered for the first time, and will be reused whenever the same subproblem $f(v)$ is subsequently encountered.

As an extension to our example, let us consider the *resource-constrained shortest path* (*RCSP*) problem, which is NP-complete. Here, we assume that there are a number of resources with maximum values represented as the vector $\tilde{u}$. We also assume that each edge (v,w) of a path consumes the amount $\tilde{r}_{v,w}$ of resources. The aim of this problem is to find the shortest path whose traversal does not exceed the maximum usage $\tilde{u}$. The resource usage requirement is an instance of an *ad-hoc constraint* added to the original shortest path problem. An RCSP solution can be modeled by the following dynamic program:

$$f(X,\tilde{R}) = \textbf{if } \tilde{R} > \tilde{u} \textbf{ then } \infty \textbf{ else if } \mathit{destination}(X) \textbf{ then } 0 \textbf{ else } \mathit{min}\{ f(Y, \tilde{R} + \tilde{R}_{X,Y}) + c_{X,Y} | (X,Y) \in \mathcal{E} \} \quad (1)$$

We note that there are various ways of modeling this problem as a dynamic program, the earliest of which is provided in (Joksch 1966). In the above program, we note the test $\tilde{R} > \tilde{u}$ which stops the program if the ad-hoc constraint is violated. Similar to the previous example, we would like to reuse the solution to a subproblem whenever that subproblem (or a more specific one) is encountered again. An obvious way to identify the opportunity for reuse is to check that the current subproblem $f(v,\tilde{r})$ is the same as some previously encountered subproblem $f(w,\tilde{s})$. This can be achieved by checking that $v = w$ and $\tilde{r} = \tilde{s}$.

A less obvious, but more general way of finding opportunities for reuse is to check that $v = w$ and $\tilde{r} \geq \tilde{s}$. Indeed, since $\tilde{r}$ represents the amount of resources consumed before the node v has been reached, we can surely reuse the solution to $f(v,\tilde{r})$ whenever we encounter the subproblem $f(v,\tilde{s})$, where the amount of resources consumed before reaching v is $\tilde{s} \leq \tilde{r}$. In other words, there is opportunity for reusing a solution whenever there is a certain relationship between the *context* $\tilde{s}$ of the current subproblem, and the context $\tilde{r}$ of a previously encountered subproblem. As we shall see later, we can use *constraints* to express the context of a subproblem, and *constraint subsumption* to express the relationship that allows sub-solution reuse.

In this paper we propose a more efficient way to detect subproblem similarity based on the notion of *interpolants* (Craig 1955) which generalize the context of a

solved subproblem such that more subproblems can be considered similar and can simply reuse the existing solution. As an example, suppose that $f(v,\tilde{r})$ returns no solution because $\tilde{r} \leq \tilde{u}$ is violated. Here we can generalize v and $\tilde{r}$ to the constraint Ψ on the variables X and $\tilde{R}$ such that $\Psi(X,\tilde{R}) \equiv X = v \wedge \tilde{R} > \tilde{u}$. Thus, we obtain a constraint that indicates the *lack of a solution* to a subproblem. This result can be reused when, say, $f(w,\tilde{s})$ is subsequently encountered, and $\Psi(w,\tilde{s})$ holds. Then, we can infer immediately that $f(w,\tilde{s})$ has no solution. In other words, once we have found the solution, or possibly the lack thereof, to a sub-problem in a given context, we compute a *more general context* in which the solution is still valid. Since a more general context may subsume a larger number of specific subproblem contexts, this approach may lead to an increased number of new subproblems that would reuse the solution to $f(v,\tilde{r})$, thus reducing the overall execution time of our method.

In general, when a solution to a subproblem $f(X,\tilde{R})$, with the context $\Psi(X,\tilde{R})$ has already been computed, we want another subproblem $f(Y,\tilde{S})$ with context $\Psi'(Y,\tilde{S})$ to reuse the solution. The solution is reusable when the following hold:

1. $\Psi'(Y,\tilde{S}) \Rightarrow \Psi(Y,\tilde{S})$. Here we want to generalize the context $\Psi(X,\tilde{R})$ to some $\overline{\Psi}(X,\tilde{R})$ because it is then more likely that $\Psi'(Y,\tilde{S}) \Rightarrow \overline{\Psi}(Y,\tilde{S})$. We require that this generalization *does not add better solutions* than the optimal solutions of the subproblem $f(X,\tilde{R})$ with the original context $\Psi(X,\tilde{R})$. This holds when the generalization does not add more solutions, which means that it has to preserve all paths that violate ad-hoc constraints.
2. The optimal solution for $f(X,\tilde{R})$, with context $\overline{\Psi}(X,\tilde{R})$ also has to be applicable for $f(Y,\tilde{S})$, with context $\Psi'(Y,\tilde{S})$, otherwise, an infeasible path may be taken to be the overall optimal.

In this paper we provide a dynamic programming algorithm with a more general subproblem similarity test based on the above two conditions. For this algorithm we also provide a mechanism to return not just one, but up to $k \geq 1$ solutions.

Our method is in fact designed to *augment* other search algorithms. To demonstrate this, we have implemented two well known algorithms for the RCSP problem (Beasley and Christofides 1989): one that uses simple tabling, and one that employs the branch-and-bound method. Then, we have augmented these two algorithms with our opportunistic solution reuse method. In our experimental evaluation section, we report improvements in the running times of the augmented algorithms, as compared to their corresponding non-augmented version.

Our technique is related to various *formula caching* techniques in CSP and SAT solving. The main difference being the use of interpolants to strengthen cached formulas to achieve better reuse. Our use of interpolant can be considered as an on-the-fly method to better approximate the *dominance relation* (Kohler and Steiglitz 1974) for better reuse of memoed subproblems.

Formula caching also encompasses techniques known as no-good learning in CSP (Frost and Dechter 1994) and conflict-driven and clause learning in SAT solving (Bayardo and Schrag 1997; Moskewicz et al. 2001; Silva and Sakallah 1996). In these techniques, learning occurs when an algorithm encounters a *dead-end*, where further search on a subproblem is impossible due to the given problem constraints. For many optimization problems, the occurrence of dead-ends are usually small or none at all, hence these techniques are not generally effective for optimization problems. Our work is related to these in the sense that we extract information from any node where an ad-hoc constraint is violated. A significant departure is that our algorithm still learns, even when there are no dead-ends. This is important for the purpose of efficient search for all solutions in the presence of search paths that are infeasible (due to ad-hoc constraints) but not forming dead-ends.

Instead of formula caching, (Kitching and Bacchus 2007) employs *component caching* technique commonly used in CSP solving to solve optimization problems. Although the technique also employs memoization, it is only effective for problems with largely independent constraints, thereby excluding the RCSP problem considered here.

The use of interpolants has been widely known in the area of computer program reasoning, notably for *abstract interpretation* (Cousot and Cousot 1977), which is an efficient approximate execution of programs to find errors, but often results in many false positives. Interpolants are used to refine the approximation upon encountering a false positive (Henzinger et al. 2004; Jhala and McMillan 2005). Interpolation is also employed to improve the completeness of SAT-based bounded model checking (McMillan 2003).

We will start by explaining the basic idea of our algorithm using an example in the next section. We will then formalize our algorithm and provide experimental results on RCSP benchmarks from the OR-library (Beasley and Christofides 1989) and also on WCET analysis of various programs from Mälardalen benchmark suite (Mälardalen 2006).

The Basic Idea

This section provides an informal introduction to our approach. Consider finding the shortest path between nodes a and z for the graph in Figure 1. In this graph, each edge is labeled with $c[r]$, where c is the cost and r is the resource consumed when the edge is included in a path. We require that the total resource consumed in a path from a to z should not exceed 10.

An execution of a depth-first backtracking algorithm on the dynamic program is illustrated in Figure 2. In this tree, the symbols R and C annotate nodes and paths, respectively. The symbol R denotes the partial sum of the resource consumed in reaching a node from a, whereas C denotes the optimal value of the respective path. Note that node d is visited in three different contexts: (α), (β), and (γ) (in that order), where the partial sums R of the resource consumed are different. In traditional dynamic programming, (β) cannot reuse the solutions of (α) since its context ($R = 5$) is different from that of (α), which is $R = 4$. Also note that in the subtree (α), the maximum resource usage is violated by the edge (d,z), which consumes 7 resources, since at (α), the resource consumed is already 4. At (α) any partial sum greater than 3 would still violate this constraint. Instead of memoing (α) with partial sum 4, we memo the generalized partial sum $R > 3$. This is the most general "interpolant" that keeps the edge (d,z) infeasible. When (β) satisfies this context, this implies that the solutions found under (β) are a

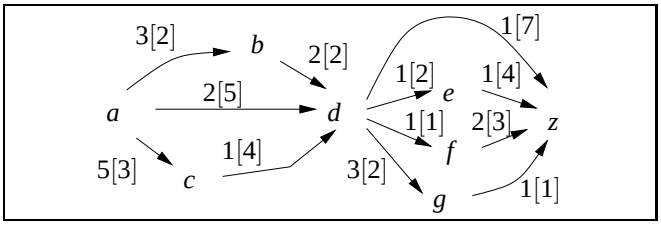

Figure 1: Graph Example

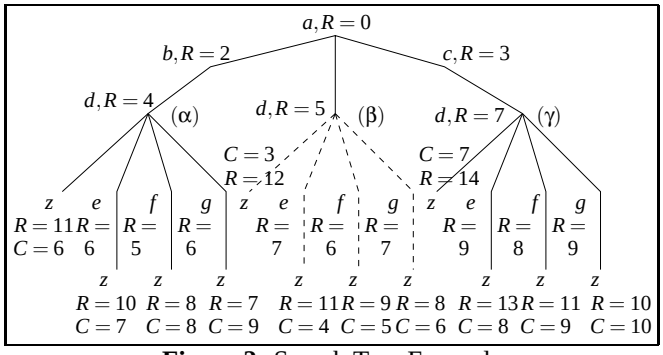

Figure 2: Search Tree Example

subset of the solutions of (α). However, this does not mean that every solution found under (α) is also a solution under (β). Indeed, (α) represents the situation where node d has been reached under the context $R = 4$, which means that solutions under (α) have a potential "slack" of 10-4=6 resources. In contrast, by similar reasoning, solutions under (β) have a potential slack of only 10-5=5 resources. Thus, by taking a path segment that represents a solution under (α), and transferring it under (β), we might in fact violate the maximum usage requirement. If we simply assume that (β) includes the optimal solution of (α), that is, $\langle d,e,z\rangle$, the result returned by (β) becomes approximate, since the actual optimal path for (β) in the above example is $\langle d,f,z\rangle$.

In order to reuse the solution of (α) at (β), we also require that the context upon reaching (β) be subsumed by the generalized context inferred for (α). Otherwise, the algorithm may return an infeasible "solution." To allow more reuse, our algorithm may also record, or *memo*, up to $k \geq 1$ most optimal solutions in the *memo* table, from where they can be retrieved whenever a similarity check is performed. Now suppose that $k = 2$, and for (α) we store both the first optimal $\langle d,e,z\rangle$ and the second optimal $\langle d,f,z\rangle$. Here, the solution $\langle d,e,z\rangle$ cannot be reused for (β), but $\langle d,f,z\rangle$ can, and this becomes the optimal solution for (β). Note that by checking that 1) (β) satisfies the generalized context of (α), and ensuring that 2) some of the stored solutions for (α) are feasible for (β), we avoid traversing the subtree of (β) resulting in an optimized search space.

When the algorithm visits the node (γ), we test whether we can reuse the solutions of (α). Remember that for (α), we have memoed only two solutions: $\langle d,e,z\rangle$ and $\langle d,f,z\rangle$. The context of (γ) is $R = 7$, which keeps the edge (d,z) infeasible, and this partial sum satisfies the generalized context of (α), which is $R > 3$. However, both the stored solutions $\langle d,e,z\rangle$ and $\langle d,f,z\rangle$ of (α) are not feasible at (γ). The algorithm therefore expands the subtree of (γ) to search for solutions. At (γ), the solutions $\langle d,z\rangle$, $\langle d,e,z\rangle$, and $\langle d,f,z\rangle$ are all infeasible. A generalization of the context of (γ) should keep all these solutions infeasible. Notice that in order to keep each of $\langle d,z\rangle$, $\langle d,e,z\rangle$, and $\langle d,f,z\rangle$ infeasible, we can generalize the context to $R > 3$, $R > 4$, and $R > 6$, respectively. Our algorithm combines all these interpolants into their conjunction, which is $R > 6$.

Now consider the problem of returning the best $k = 2$ solutions at the root node. Here we should not simply pick the best k solutions among all solutions returned by the children. To see why, consider the smaller problem of returning k solutions of only nodes (α) and (β). Recall that at (α) and (β), respectively there are only two solutions, of which one was memoed. Note that the best overall solution is $\langle a,d,f,z\rangle$, which is one that visits (β). Here we cannot use the best solution $\langle a,b,d,e,z\rangle$ of (α) as the second best overall because the second best overall is actually $\langle a,d,g,z\rangle$, which is not memoed. Here, when comparing the solutions of two children, we may not pick a solution of a child as the next best (such as $\langle a,b,d,e,z\rangle$ of (α) for the second best) when it is not comparatively better than the solutions returned by the other child. As a result, the algorithm may return fewer than k solutions. In our example, we only return $\langle a,d,f,z\rangle$ as the optimal solution without a second optimal.

The Algorithm

A dynamic program induces a *state transition system* that defines transformations from one context (state) to another. For instance, a transition system for the RCSP dynamic program would define its transitions as moves from a graph vertex to one of its neighbors, while consuming certain resources.

Important to the notion of context is the notion of a *state variable*. We denote a context by a constraint on a set $\tilde{X}$ of state variables. A *transition* in a state transition system defines restriction and update of state variables. We would denote a transition by a constraint called *transition relation* between one version $\tilde{X}$ of the state variables and another version $\tilde{X}'$ with possibly different values. A state transition system consists of a context called the *initial state* and a number of transitions.

Given our RCSP problem in Figure 1 and our dynamic program formulation (1), we have as state variables X, whose value is any of the graph vertices a, b, c, ..., and R, whose value is the amount of resources consumed so far. We have as initial state $X = a \wedge R = 0$, and one of the transition relations induced by the graph is $X = a \wedge X' = b \wedge R \leq 10 \wedge R' = R+5$. Here $R \leq 10$ manifests the ad-hoc constraint.

Every node in the search tree corresponds to a *trace*, which is the sequence of constraints along the path from the root to the node. Formally, a *symbolic trace of length m* (or simply *trace*) is the formula

$$\Theta(\tilde{X}_0) \wedge \bigwedge_{i=0}^{m-1} \rho_{i+1}(\tilde{X}_i, \tilde{X}_{i+1}),$$

where $\Theta(\tilde{X}_0)$ is the initial state, and each $\rho_{i+1}(\tilde{X}_i, \tilde{X}_{i+1})$ is a transition relation. We call the variables $\tilde{X}_i$ as the *level-i state variables*. We use Π to denote a trace. We say that a trace is *infeasible* if it is unsatisfiable (note that a trace is a formula). Otherwise, we call it *feasible*. A *solution* is a feasible trace Π of length m which entails a *final condition* $\varphi(\tilde{X}_m)$, that is,

$\Pi \Rightarrow \varphi(\tilde{X}_m)$. For example, the final condition of the RCSP problem is that the destination vertex is reached. A *suffix* trace is a trace without the initial state, that is, the formula $\bigwedge_{i=l}^{m-1} \rho_{i+1}(\tilde{X}_i, \tilde{X}_{i+1})$ for some l and m. We use variants of π to denote a suffix trace. Given a suffix trace π with state variables $\tilde{X}_l, \ldots, \tilde{X}_m$, we denote by $\pi(t, f)$ the renamed formula $\pi[\tilde{X}_t/\tilde{X}_l, \ldots, \tilde{X}_{t+m-l}/\tilde{X}_m]$, where $f = t + m - l$. We also denote $\pi(t, f)$ as simply $\pi(t)$ when we are not concerned about its highest-level state variables. A suffix trace $\pi(m, f)$ is a *subsolution* when there is a trace Π of length m such that $\Pi \wedge \pi(m, f)$ is a solution.

A dynamic programming problem exhibits the property of *optimal substructure*, where the optimal subsolutions of subproblems can be used to construct the optimal solutions of the overall problem. Therefore in a dynamic programming setting we can speak about not just optimal solutions, but also about optimal subsolutions. We assume that every subsolution π has a *cost* attached to it, denoted $cost(\pi)$, which we assume to belong to a total order $(O, \preceq)$ (e.g., $(\mathbb{N}, \leq)$ in case of RCSP). When $cost(\pi_1) \preceq cost(\pi_2)$ we say that π_1 is more optimal than π_2. It is therefore unambiguous to mention the first (or most), second, or in general the t-th *optimal* subsolution of a finite set S of subsolutions.

The search tree traversal constructs traces. Each trace is possibly extended into solutions and after these have been found, the algorithm constructs a *summarization* for the trace, which contains the k-most optimal solutions, as well as the *most general context* Ψ w.r.t. the constraint language at hand, under which these solutions are valid. This general context Ψ is a generalization of the specific context under which the current sub-problem was encountered. Moreover, the solutions to the current sub-problem may be reused whenever we encounter a new subproblem whose context is subsumed by Ψ. Summarizations are stored in a *memo table*, from which they will be retrieved whenever a similarity check is attempted. When a new trace is encountered during the search, we want to avoid further traversal by using summarizations already stored in the memo table whenever possible.

Definition 1 (Summarization) *Given a bound $k \geq 1$, a* summarization *is a triple (Ψ, S, C), where Ψ is a constraint on some state variables $\tilde{X}_i$, S a set of subsolutions (where $0 \leq |S| \leq k$), and C is a boolean flag.*

Given a formula Ψ on state variables $\tilde{X}_i$, we denote by $\Psi(m)$ the formula $\Psi[\tilde{X}_m/\tilde{X}_i]$. Suppose that T is the set of all subsolutions that extend Π into solutions and suppose that U is the set of all subsolutions π such that $\Psi(m) \wedge \pi(m)$ is satisfiable. We say that (Ψ, S, C) is a summarization *of a trace* Π of length m when:

1. Ψ is a generalization of Π that does not add more solutions, that is, $\Pi \Rightarrow \Psi(m)$ and $T = U$.
2. $S \subseteq T$, T is nonempty iff S is nonempty, and if π is t-th optimal in S, it is also t-th optimal in T.
3. C denotes whether all subsolutions of Π is included in S or not, formally, $C = true$ iff $S = T$.

During search, the algorithm would encounter four kinds of traces: infeasible traces, solutions, traces that are *subsumed* by a memoed summarization, and traces that are extended to other traces by transition relation. In Propositions 1 to 4 below, we consider how we may produce a summarization for each of them.

We start with infeasible traces:

Proposition 1 *$(false, \emptyset, true)$ is a summarization of an infeasible trace Π.*

The construction of summarizations of other traces require the concept of *interpolants* (Craig 1955).

Definition 2 (Interpolant) *If F and G are formulas such that F entails G, then there exists an* interpolant *H which is a formula such that F entails H and H entails G, and each parameter of H is a parameter of both F and G.*

When $F \Rightarrow G$, we denote any interpolant H such that $F \Rightarrow H$ and $H \Rightarrow G$ as $int(F, G)$.

We now state the summarization of a solution:

Proposition 2 *$(int(\Pi, \varphi(\tilde{X}_m)), \{true\}, true)$ is a summarization of a solution Π of length m.*

In Proposition 2, the interpolant $int(\Pi, \varphi(\tilde{X}_m))$ is a formula $\Psi(\tilde{X}_m)$ such that $\Pi \Rightarrow \Psi(\tilde{X}_m)$ and $\Psi(\tilde{X}_m) \Rightarrow \varphi(\tilde{X}_m)$. That is $int(\Pi, \varphi(\tilde{X}_m))$ is a generalization of Π that still implies the final condition.

When *subsumption* holds, we guarantee that instead of searching for all subsolutions extending a trace, we could use the results memoed in a summarization.

Definition 3 (Subsumption) *A trace Π of length m is subsumed by a summarization (Ψ, S, C) when $\Pi \Rightarrow \Psi(m)$ and either $C = true$, or, $C = false$, S is not empty, and there is $\pi \in S$ such that $\Pi \wedge \pi(m)$ is a solution.*

When a trace Π of length m is subsumed by a summarization (Ψ, S, C), we want to generate another summarization (Ψ', S', C') of Π without having to search for solutions among the extensions of Π.

Proposition 3 *Suppose that Π is subsumed by (Ψ, S, C). Define a set $F = \{\pi | \pi \in S$ and $\Pi \wedge \pi(m) \Rightarrow false\}$. When $\Psi' \equiv \Psi \wedge \bigwedge_{\pi \in F} int(\Pi, \pi(m) \Rightarrow false)$, $S' = S - F$, and $C' = C$, then (Ψ', S', C') is a summarization of Π provided when $C' = false$, then $S' \neq \emptyset$.*

In Proposition 3, $int(\Pi, \pi(m) \Rightarrow false)$ is a formula $\Psi(\tilde{X}_m)$ such that $\Pi \Rightarrow \Psi(\tilde{X}_m)$ and $\Psi(\tilde{X}_m) \Rightarrow (\pi(m) \Rightarrow false)$, that is, a generalization of Π that preserves the infeasibility of the subsolution π.

We now consider the last kind of traces: those that are extended into other traces.

Proposition 4 *Suppose that there are l transition relations and that a trace Π of length m is extended to traces $\Pi^1, \ldots, \Pi^l$ of length $m+1$ by all transition relations where $\Pi^i \equiv \Pi \wedge \rho^i_{m+1}(\tilde{X}_m, \tilde{X}_{m+1})$. (Ψ', S', C') is a summarization of Π when the following conditions hold:*

- *(Ψ^i, S^i, C^i) is a summarization of Π^i, for all i, $1 \leq i \leq l$.*
- *$\Psi'(m) \equiv \bigwedge_{i=1}^{l} int(\Pi, \rho^i_{m+1}(\tilde{X}_m, \tilde{X}_{m+1}) \Rightarrow \Psi^i(m+1))$.*
- *For every set S^i where $1 \leq i \leq l$, construct a set T^i such that $\pi \in S^i$ iff $(\rho^i_{m+1}(\tilde{X}_m, \tilde{X}_{m+1}) \wedge \pi(m)) \in T^i$. Now construct a set T such that $\pi \in T$ if $\pi \in T^i$ for some i such that*

```
func summarize(Π of length m) do
  if (Π ⇒ false) then return (false, ∅, true)
  else if (Π ⇒ φ(X̃_m)) then
    return (int(Π, φ(X̃_m)), {true}, true)
  else if (There is (Ψ,S,C) ∈ Table s.t. Π ⇒ Ψ(m)) then
    Ψ'(X̃_m), S', C' := Ψ(m), ∅, C
    foreach (π ∈ S) do
      if (Π ∧ π(m) ⇒ false) then
        Ψ'(X̃_m) := Ψ'(X̃_m) ∧ int(Π, π(m) ⇒ false)
      else
        S' := S' ∪ {π}
      endif
    endfor
    if (S ≠ ∅ and ¬C and S' = ∅) goto Recurse
    return (Ψ'(X̃_m), S', C')
  else
Recurse:
    (Ψ', S', C') := (true, ∅, true)
    foreach (Transition relation ρ(X̃_m, X̃_{m+1})) do
      (Ψ(m+1), S, C) := summarize(Π ∧ ρ(X̃_m, X̃_{m+1}))
      Ψ' := Ψ' ∧ int(Π, ρ(X̃_m, X̃_{m+1}) ⇒ Ψ(m+1))
      Replace all π ∈ S with ρ(X̃_m, X̃_{m+1}) ∧ π(m+1)
      S', C' := combine_subsolutions(S, C, S', C')
    endfor
    Table := Table ∪ {(Ψ', S', C')}
    return (Ψ', S', C')
  endif
endfunc
```

Figure 3: The Algorithm

$1 \le i \le l$ *and for all* T^j *such that* $1 \le j \le l$, *if* $C^j = false$ *then there is* $\pi' \in T^j$ *such that* $cost(\pi) \preceq cost(\pi')$. *Now* S' *is a subset of* T *(where* $|S'| \le k$ *for some bound* $k \ge 1$*) such that* T *is nonempty iff* S' *is nonempty, and* π *is* t*-th optimal in* S' *only if* π *is* t*-th optimal in* T.

- $C = true$ *when* $S' = T$ *and for all* i *such that* $1 \le i \le l$, $C^i = true$.

Note that in all summarizations (Ψ', S', C') of Π mentioned in Propositions 1 to 4, we maintain the property that if $\pi \in S'$ is the first optimal subsolution such that $\Psi'(m) \wedge \pi(m)$ is satisfiable, then it is also the first optimal subsolution of Π. This ensures that the generalization $\Psi'(m)$ does not add a better optimal. This is easy to see in Propositions 1 and 2. For Proposition 3, this follows from the fact that $\Psi'(m) \wedge \pi(m)$ is also unsatisfiable for the optimal solutions π where $\Pi \wedge \pi(m)$ is unsatisfiable. For Proposition 4, this follows from the fact that $S' \subseteq T$, where T is the set of the most optimal subsolutions of Π.

The pseudocode of our algorithm is shown in Figure 3. The procedure *summarize* receives as an input a trace Π and returns a summarization of Π. We assume the existence of a global variable *Table* representing the memo table, which is initialized to $\emptyset$. The body of the function *summarize* includes as the first statement an **if** conditional which has four cases. The first case is when the input Π is an infeasible trace. Here we generate the summarization of Π according to Proposition 1. The second case is when Π is a solution, where

```
func combine_subsolutions(S, C, S', C') do
  S'', i := ∅, 1
  while (((S ≠ ∅ and S' ≠ ∅) or
          (S ≠ ∅ and S' = ∅ ∧ C') or
          (S' ≠ ∅ and S = ∅ ∧ C)) and i ≤ k) do
    Let π be the first optimal in S
    Let π' be the first optimal in S'
    if (S ≠ ∅ and S' ≠ ∅) then
      if (cost(π) ⪯ cost(π'') for all π'' ∈ S') then
        S, S'' := S − {π}, S'' ∪ {π}
      else if (cost(π') ⪯ cost(π'') for all π'' ∈ S) then
        S', S'' := S' − {π'}, S'' ∪ {π'}
      endif
    else if (S ≠ ∅ and S' = ∅ and C') then
      S, S'' := S − {π}, S'' ∪ {π}
    else if (S' ≠ ∅ and S = ∅ and C) then
      S', S'' := S' − {π'}, S'' ∪ {π'}
    endif
    i := i + 1
  endwhile
  if (S = ∅ and C and S' = ∅ and C') then C'' := true
  else C'' := false endif
  return S'', C''
endfunc
```

Figure 4: Combining Two Sets of Subsolutions

we produce the summarization according to Proposition 2. The third case is when Π is subsumed by a summarization in *Table*. Here the algorithm performs the test of subsumption where it iterates over all subsolutions π contained in the summarization, and test the satisfiability of $\Pi \wedge \pi(m)$. A summarization is produced according to Proposition 3. When subsumption does not hold, we need to recurse deeper and hence the **goto** jump to the fourth case, which is the case when Π has to be extended by transition relations into other traces. Here we extend Π with all the transition relations, and recursively call *summarize* with the extended trace as the argument. The recursive calls then return summarizations, which are then combined to construct the summarization (Ψ', S', C') of Π which satisfies Proposition 4. This is then stored in *Table* and returned by the procedure. In computing Ψ' we incrementally construct a conjunction of all interpolants returned by the recursive call and for computing S' and C' we employ another procedure *combine_subsolutions*, which processes the combining of two up-to-k most optimal subsolution sets.

The procedure *combine_subsolutions* (see Figure 4) produces the components S' and C' of Proposition 4. Whereas in Proposition 4 we are considering l sets $T^1, \ldots, T^l$, here we consider only two sets S and S', given as inputs to the procedure together with their respective completeness flags C and C'. The procedure produces an output S'' which satisfies the condition for S' in Proposition 4, and its flag C'', which satisfies the condition for C' in Proposition 4.

We conclude this section with a statement of correctness which follows from the fact that *summarize*$(\Theta(\tilde{X}_0))$ produces a summarization of the initial state $\Theta(\tilde{X}_0)$.

Theorem 1 *Suppose P is a dynamic program which induces*

Prob.	Memoization Only								Branch-and-Bound				Convex Hull (Custom)	
	DP		DP+I ($k=1$)		DP+I ($k=2$)		DP+I ($k=10$)		DP+BB		DP+BB+I ($k=1$)		Relaxation	Closing Gap
#	Nodes	T	Nodes	T	Nodes	T	Nodes	T	Nodes	T	Nodes	T	T	T
1	55044	94	25021	78	19765	297	19765	281	17950	63	13030	15	16	16
2	47484	46	22648	31	17724	110	17724	125	13075	16	9655	15	16	16
3	11448	16	7980	15	8308	32	7659	47	6195	0	5429	0	16	0
4	9777	16	7237	16	7497	31	7055	47	5982	0	5613	0	32	0
5	∞		1239214	111623	878879	55804	878879	55640	10393	16	8480	15		
6	∞		556583	20922	379342	10508	379342	10516	9751	32	8047	0		
7	178372	687	43888	172	103722	423	103722	454	9978	31	7565	15		
8	49193	172	18390	94	31759	94	31759	108	39560	94	17380	31		
9	492	15	459	0	459	0	459	0	498	15	432	0	16	0
10	347	0	321	16	321	0	321	0	345	0	307	0	16	0
11	40346	78	18020	47	26251	93	16941	110	15525	31	14309	0	46	0
12	35432	62	18973	47	25465	109	17535	110	15439	16	14425	16	31	0
13	9699	47	7708	16	7266	16	7266	32	4678	16	4094	0		
14	3678	16	3208	0	3187	32	3187	16	3678	32	3208	0		
15	111890	375	56273	188	86036	266	86036	281	20162	31	15482	31		
16	28881	94	20073	62	23698	78	23698	78	26416	62	19094	31		
17	858743	2234	281101	1031	247821	1110	247821	1234	1218781	1312	562562	500	46	16
18	752241	1988	255952	922	226414	954	226414	1106	1034608	984	512734	422	47	48
19	76215	180	42221	93	56806	250	42072	219	36789	31	29735	16	78	0
20	63592	130	40255	94	47860	204	39684	172	44402	31	37463	47	62	0
21	590257	2204	60619	266	64461	266	64461	250	21730	78	16677	15		
22	131905	485	33176	125	35779	110	35779	157	17381	47	12943	15		
23	3744373	15516	991298	10610	2670921	10258	2670921	10203	100440	610	71837	110		
24	603421	2234	283042	1203	486856	1673	486856	1625	145546	859	103502	157		

Table 1: RCSP Experimental Results

a state transition system with initial state $\Theta(\tilde{X}_0)$. *Suppose that summarize*$(\Theta(\tilde{X}_0))$ *returns a summarization* (Ψ,S,C). *Then if* $S=\emptyset$, *P has no solution. Otherwise, let* π *be the i-th optimal subsolution in S. Then* $\Theta(\tilde{X}_0)\wedge\pi(0)$ *is also the i-th optimal solution of P.*

Experimental Evaluation

We now demonstrate the interpolation method in augmenting three classes of algorithms. The first is a basic depth-first search with memoization (DP). The second is a refinement of this, using branch-and-bound (DP+BB). Both of these are applied to an underlying RCSP problem benchmark. The third class is from WCET, and we demonstrate the interpolation method augmenting a symbolic trace traversal of straight-line programs, in search for the "longest" path.

Table 1 shows the experimental results using the 24 OR-library RCSP benchmarks (Beasley and Christofides 1989). In the table "DP" denotes runs with basic memoization, "I" denotes runs augmented with the interpolation method, "BB" denotes runs with branch-and-bound, and $k=\alpha$ denotes the limit α on the number of stored subsolutions. The "Nodes" column displays the number of nodes visited in the search tree while the "T" column displays the running time of the experiments in milliseconds. The algorithms were implemented in Java (JDK 1.5.0) and ran on 1.86 GHz Pentium 4 machine with 1.5GB RAM.

As seen, interpolation (DP+I) improves DP significantly. When comparing runs with $k=1$, $k=2$, and $k=10$, the performance seems to degrade from $k=1$ to $k=2$, yet $k=10$ has the smallest search space. The runs with $k=2$ and $k=10$ seems to reduce the search space of benchmarks 5 and 6 significantly. Similarly, interpolation (DP+BB+I) improves DP+BB significantly.

In other words, while interpolation results in a uniformly better search space, our experiments show that the time penalty is not significant.

It is interesting, though *orthogonal*, to compare the algorithms DP+I and DP+BB, that is, interpolation vs branch-and-bound. As can be seen, the results are mixed. On problems 5 and 6 for example, interpolation performs much worse because the trees contain deep regions that are pruned early using branch-and-bound. On problems 8, 17, and 18 on the other hand, interpolation performs much better.

Finally for Table 1, we include some timing results for two state-of-the-art *custom* algorithms. Based on convex hull construction (Ziegelmann 2001), the "Relaxation" column concerns the time solving relaxation method, and the "Closing Gap" column concerns the time closing gap method. We note that our general-purpose algorithm is in fact sometimes competitive with these custom RCSP algorithms.

Next we consider a completely different class of problems, that of worst-case execution time (WCET) analysis of finite-looping programs. We seek the maximum number of statements among all of the paths of the program. Obviously, some execution paths are infeasible due to the interplay of assignments and conditional statements. These would correspond to the ad-hoc constraints of our search problem. In this experiment, we *delete constraints* induced by program statements that do not contribute to the infeasibility of a path. This then is the basic step which implements our interpolant. Note that we are considering *exact* WCET here, whereas typical WCET analyzers such as the integer linear programming approach described in (Li and Malik 1995) obtain only an upper bound.

Our prototype is implemented in CLP($\mathcal{R}$) (Jaffar et al. 1992) programming language and run on Pentium 4 2.8 GHz CPU with 512 Mb RAM. The advantage of using constraint logic programming technology is easy formulation of dy-

Program	DP		DP+I	
	Nodes	Time (s)	Nodes	Time (s)
bsort(5)	2233	11.22	58	0.05
bsort(10)	∞		218	0.96
bsort(15)	∞		478	7.04
binary(4)	381	0.70	169	0.30
binary(6)	2825	27.47	873	6.54
decoder	344	0.31	132	0.19
sqrt	923	4.25	253	1.43
qurt	1104	14.47	290	2.60
janne_complex	1517	17.93	683	4.36

Table 2: WCET Analysis Results

namic programs with ad-hoc constraints and efficient algorithm for projection of formulas onto a limited set of variables which simplifies the generation of interpolants.

We show the results in Table 2. The "DP" denotes runs with standard dynamic programming, "DP+I" denotes the runs that employs our algorithm. The "Nodes" columns in the table contain the numbers of nodes in the search tree, and "Time" is the running time in seconds. ∞ denotes cases when the execution ran out of resources. We are considering several array sizes for bubble sort program "bsort" with unrestricted array elements, as well as the "binary" case when the array elements are binary (i.e. restricted to two values only). As is known, the complexity of bubble sort is quadratic to the array size, hence also the search tree depth. In the case when the array elements are unrestricted, the search tree would have very few infeasible traces. For these problems, our algorithm has a linear spatial performance to the tree depth. When the array elements are restricted to binary values, the number of infeasible traces is far larger. The more the infeasible traces, the less general the summarizations since we need to keep the component Ψ of a summarization restrictive enough to preserve the infeasibility, therefore here memoization is less effective. Other programs are obtainable from the Mälardalen benchmark suite (Mälardalen 2006), where the experiments also show a significant amount of reduction. The amount of reduction lessens in some cases when the control flow increases in complexity thus increasing the number of infeasible traces.

Conclusion

We addressed the problem of effective reuse of subproblem solutions in dynamic programming. We proposed a generalization of the context of subproblems using interpolants, and store these generalizations and the optimal solutions of the subproblems as summarizations in the memo table for later reuse when a subproblem with similar context is encountered in the search tree. We also presented a general technique to search for at most $k \geq 1$ optimal solutions. We provide experimental results on RCSP benchmarks, where we showed that our general algorithm is competitive to state-of-the-art custom algorithms, and also experimental results on program's exact WCET analysis.

Acknowledgment

We thank Chu Duc Hiep for running the experiments. This work is supported by Singapore Ministry of Education Academic Research Fund No. R-252-000-234-112.

References

Bayardo, Jr., R. J., and Schrag, R. 1997. Using csp look-back techniques to solve real-world sat instances. In *14th AAAI/9th IAAI*, 203–208. AAAI Press.

Beasley, J. E., and Christofides, N. 1989. An algorithm for the resource-constrained shortest path problem. *Networks* 19(3):379–394.

Cousot, P., and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis. In *4th POPL*, 238–252. ACM Press.

Craig, W. 1955. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation* 22.

Frost, D., and Dechter, R. 1994. Dead-end driven learning. In *12th AAAI*, 294–300. AAAI Press.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and McMillan, K. L. 2004. Abstractions from proofs. In *31st POPL*, 232–244. ACM Press.

Jaffar, J.; Michaylov, S.; Stuckey, P. J.; and Yap, R. H. C. 1992. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS* 14(3):339–395.

Jhala, R., and McMillan, K. L. 2005. Interpolant-based transition relation approximation. In Etessami, K., and Rajamani, S. K., eds., *17th CAV*, volume 3576 of *LNCS*, 39–51. Springer.

Joksch, H. C. 1966. The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications* 14(2):191–197.

Kitching, M., and Bacchus, F. 2007. Symmetric component caching. In Veloso, M. M., ed., *20th IJCAI*, 118–124.

Kohler, W. H., and Steiglitz, K. 1974. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM* 21(1):140–156.

Li, Y.-T. S., and Malik, S. 1995. Performance analysis of embedded software using implicit path enumeration. In *2nd LCT-RTS*, 88–98. ACM Press. SIGPLAN Notices 30(11).

2006. Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

McMillan, K. L. 2003. Interpolation and SAT-based model checking. In W. A. Hunt, J., and Somenzi, F., eds., *15th CAV*, volume 2725 of *LNCS*, 1–13. Springer.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *38th DAC*, 530–535. ACM Press.

Silva, J. P. M., and Sakallah, K. A. 1996. GRASP—a new search algorithm for satisfiability. In *ICCAD 1996*, 220–227. ACM and IEEE Computer Society.

Ziegelmann, M. 2001. *Constrained Shortest Paths and Related Problems*. Ph.D. Dissertation, University of Saarlandes, Saarbrücken.