Divide-and-Conquer
(Recurrence)

• Textbook:
  - *Algorithm Design and Applications*, Goodrich, Michael T. and Roberto Tamassia (chapter 8)
  - *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein (chapters 2 and 4)
Intended Learning Outcomes

• Understand the divide-and-conquer paradigm and how recurrence can be obtained

• Solve recurrences using substitution method

• Describe various examples to analyse divide-and-conquer algorithms and how to solve their recurrences
Divide-and-Conquer

- The **divide-and-conquer** paradigm involves three steps at each level of the recursion:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem \( D(n) \)
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner \( aT(n/b) \)
  - **Combine** the solutions to the subproblems into the solution for the original problem \( C(n) \)

\[
T(n) = \Theta(1) \text{ if } n \leq c, \\
T(n) = D(n) + aT(n/b) + C(n) \text{ otherwise.}
\]
Illustrative Example (Merge Sort)

• The merge sort algorithm closely follows the **divide-and-conquer** paradigm
  
  ▪ **Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n=2$ elements each ($\Theta(1)$)
  
  ▪ **Conquer:** Sort the two subsequences recursively using merge sort ($2T(n/2)$)
  
  ▪ **Combine:** Merge the two sorted subsequences to produce the sorted answer ($\Theta(n)$)

\[
T(n) = \Theta(1) \text{ if } n = 1,
T(n) = \Theta(1) + 2T(n/2) + \Theta(n) \text{ otherwise.}
\]
How Does Merge Sort Work?

• **Idea:** suppose we have two piles of cards face up on a table; *each pile is sorted*, with the smallest cards on top

• We wish to merge the two piles into a single sorted output pile, which is to be face down on the table

  1. choose the smaller of the two cards on top of the face-up piles
  2. remove it from its pile (which exposes a new top card)
  3. place this card face down onto the output pile
  4. repeat this until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile
Merge Sort Algorithm

MergeSort(A, left, right) {
    if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
    }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
// (how long should this take?)
```
Merge(A, left, mid, right)
    n₁ = mid-left+1
    n₂ = right-mid
    create two sorted subarrays L[0..n₁] and R[0..n₂]
    for i=0 to n₁-1
do    L[i]=A[left+i]
    for j=0 to n₂-1
do    R[j]=A[mid+j+1]
L[n₁]=∞
R[n₂]=∞
i=0
j=0
    for k=left to right
    do if L[i] ≤ R[j]
then    A[k]=L[i]
    i=i+1
else    A[K]=R[j]
    j=j+1
```
Merge()

Merge(A, left, mid, right)

\[
\begin{align*}
    n_1 &= \text{mid-left} + 1 \\
    n_2 &= \text{right-mid} \\
    \text{create two sorted subarrays } & L[0..n_1] \text{ and } R[0..n_2] \\
\end{align*}
\]

\[\Theta(1)\]

\[
\begin{align*}
    \text{for } i = 0 \text{ to } n_1 - 1 & \quad \Theta(n_1) \\
        \quad \text{do } L[i] &= A[\text{left}+i] \\
    \text{for } j = 0 \text{ to } n_2 - 1 & \quad \Theta(n_2) \\
        \quad \text{do } R[j] &= A[\text{mid}+j+1] \\
\end{align*}
\]

\[\Theta(1)\]

\[
\begin{align*}
    L[n_1] &= \infty \\
    R[n_2] &= \infty \\
    i &= 0 \\
    j &= 0 \\
\end{align*}
\]

\[\Theta(1)\]

\[
\begin{align*}
    \text{for } k = \text{left} \text{ to } \text{right} & \quad \Theta(n) \\
        \text{do } \text{if } L[i] & \leq R[j] \\
            \quad \text{then } A[k] &= L[i] \\
                \quad i &= i+1 \\
            \quad \text{else } A[K] &= R[j] \\
                \quad j &= j+1
\end{align*}
\]
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on an array
MergeSort() running on a array
MergeSort() running on a array

24  85

63  45

17  31  96  50
MergeSort() running on an array.
MergeSort() running on an array
MergeSort() running on a array

Diagram showing the process of MergeSort on an array, with elements 17, 31, 96, and 50.
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on an array

24 45 64 85

17 31 50 96
MergeSort() running on an array

24  45  64  85  17  31  50  96
MergeSort() running on a array
Complexity Analysis of Merge Sort

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MergeSort(A, left, right) {</code></td>
<td><code>T(n)</code></td>
</tr>
<tr>
<td>if (left &lt; right) {</td>
<td><code>Θ(1)</code></td>
</tr>
<tr>
<td>mid = floor((left + right) / 2);</td>
<td><code>Θ(1)</code></td>
</tr>
<tr>
<td>MergeSort(A, left, mid);</td>
<td><code>T(n/2)</code></td>
</tr>
<tr>
<td>MergeSort(A, mid+1, right);</td>
<td><code>T(n/2)</code></td>
</tr>
<tr>
<td>Merge(A, left, mid, right);</td>
<td><code>Θ(n)</code></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>• So <code>T(n) = Θ(1)</code> when <code>n = 1</code>, and</td>
<td></td>
</tr>
<tr>
<td><code>2T(n/2) + Θ(n)</code> when <code>n &gt; 1</code></td>
<td></td>
</tr>
<tr>
<td>• So what (more succinctly) is <code>T(n)</code>?</td>
<td></td>
</tr>
</tbody>
</table>
Recurrences

• The expression that represents the **merge sort**: 

\[
T(n) = \begin{cases} 
  c & n = 1 \\
  2T\left( \frac{n}{2} \right) + cn & n > 1 
\end{cases}
\]

• is a **recurrence**.
  - Recurrence: an **equation** or **inequality** that describes a function in terms of its **value on smaller functions**
Recurrences: Factorial

• What is the recurrence equation for this algorithm?

```latex
\text{fac}(n) \text{ is }
\begin{align*}
\text{if } n = 1 & \text{ then return } 1 \\
\text{else return } & \text{ fac}(n-1) \ast 1 \\
\end{align*}
```

• A recurrence defines $T(n)$ in terms of $T$ for smaller values

```latex
T(n) = c \quad \text{ if } n=1 \\
T(n) = T(n-1) + c \quad \text{ if } n>1
```
Recurrences: Binary Search

• What is the recurrence equation for this algorithm?

\[ \text{BinSearch}(A[1...n], q) \]
\[ \begin{align*}
\text{if } n &= 1 \\
& \quad \text{then if } A[n] = q \text{ then return } n \\
& \quad \text{else return } 0
\end{align*} \]

\[ k \leftarrow (n + 1) / 2 \]

\[ \begin{align*}
\text{if } q &< A[k] \\
& \quad \text{then BinSearch}(A[1...k-1], q) \\
& \quad \text{else BinSearch}(A[k...n], q)
\end{align*} \]

\[ T(n) = c \quad \text{if } n = 1 \]
\[ T(n) = c + T(n/2) \quad \text{if } n > 1 \]
Other Recurrence Examples

\[
s(n) = \begin{cases} 
0 & n = 0 \\
c + s(n-1) & n > 0 
\end{cases}
\]

\[
s(n) = \begin{cases} 
0 & n = 0 \\
n + s(n-1) & n > 0 
\end{cases}
\]

\[
T(n) = \begin{cases} 
c & n = 1 \\
2T\left(\frac{n}{2}\right) + c & n > 1 
\end{cases}
\]

\[
T(n) = \begin{cases} 
c & n = 1 \\
aT\left(\frac{n}{b}\right) + cn & n > 1 
\end{cases}
\]
Solving Recurrences

- Substitution method
- Iteration method
- Master method
Proof by Induction
(COMP11120)

• Why is it important to learn **proof by induction**?

```
unsigned int N=0;
unsigned int i = 0;
long double x=2;
while( i < N ){
    x = ((2*x) - 1);
    ++i;
}
assert( i == N );
assert(x>0);
```

```
unsigned int N=0;
unsigned int i = 0;
long double x=2;
if( i < N ){
    x = ((2*x) - 1);
    ++i;
}  \{ k copies \}
...
assert( !(i < N) );
assert( i == N );
assert(x>0);
```
Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy More1, Lucas 2
1 Electronic Co and Information Technology
2 Electronic and Information Technology
3 Department of Computer Science
4 Division of Computer Science

Abstract—We extended ESBMC to model check embedded C software with unbounded loops. The k-induction algorithm consists of three cases: in the base case, we aim to find a counterexample with up to k loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property holds in all states reachable within k unwindings; and in the inductive step, we check that whenever k holds for k unwindings, it also holds after the next unwinding of the system. For each step of the k-induction algorithm, we infer invariants using affine constraints (i.e., polyhedrons) to specify pre- and post-conditions. Experimental results show that our approach can handle infinite safety properties in typical embedded software applications from telecommunications, control systems, and medical devices, demonstrating an improvement of the induction algorithm effectiveness compared to other approaches.

1 Overview

ESBMC is a context-bounded single- and multi-threaded C C program that can be used to verify programs, unless we know that it is generally not the case. The ESBMC is described in terms of the differences between the version 2 and the version 3 of the k-induction algorithm.

2 Differences to ESBM

Except for the loop handling version. The main changes are the new checks for each loop unwindings.

I. INTRODUCTION

The Explicit State Model Checking (BSMC) techniques based on Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) have been applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [1], [2], [3]. The idea behind the BSMC techniques is to check whether a given property is satisfied at a given depth, i.e., given a transition system M, a property φ, and a limit of iterations k, BSMC unfolds the system k times and converts it into a Verification Condition (VC) such that φ is satisfiable if and only if it has a counterexample of depth less than or equal to k.

Typically, BSMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BSMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BSMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BSMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BSMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BSMC techniques are only able to falsify properties up to a given depth k; they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth.

DepthK: A k-Induction Verifier Based on Invariant

Inference for C Programs

(Competition Contribution)

William Roche1, Heri Lucas Coelho2, 3
1 Electronic and Information Technology
2 Department of Computer Science
3 Division of Computer Science

Abstract. DepthK is a software verification tool that combines k-induction with a counterexample-driven pseudo-proof procedure. It leverages polyhedral constraints to represent and reason about the program's state space, allowing for the verification of complex properties. DepthK has been successfully applied to verify real-world programs, demonstrating its effectiveness in handling challenging verification tasks.

1 Overview

DepthK is a software verification tool that combines k-induction with a counterexample-driven pseudo-proof procedure. It leverages polyhedral constraints to represent and reason about the program's state space, allowing for the verification of complex properties. DepthK has been successfully applied to verify real-world programs, demonstrating its effectiveness in handling challenging verification tasks.

Handling Loops in Bounded Model Checking of C Programs via k-Induction

Mikhail Y. B. Godelha, Hussana I. Ismaili, and Lucas C. Cordeiro
Electronic and Information Research Center, Federal University of Amazonas, Brazil

Received: date / Revised version date

Abstract. The first attempts to apply the k-induction method to software verification are recent. In this paper, we present a novel k-induction algorithm, which is built on top of a symbolic model checker and uses an iterative deepening approach to verify, for each step k up to a given maximum, whether a given safety property holds in the program. The proposed k-induction algorithm consists of three different cases: base cases, forward conditions, and inductive steps. Initially, in the base case, we aim to find a counterexample with up to k loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that k holds in all states reachable within k unwindings; and in the inductive step, we check that whenever k holds for k unwindings, it also holds after the next unwinding of the system. The algorithm is implemented in two different ways: a sequential and a parallel version, and the results are compared. Experimental results show that both forms of the algorithm can handle a wide variety of safety properties extracted from standard benchmarks, ranging from reachability to time constraints. And by comparison, the parallel algorithm advances more verification tasks in less time. This paper marks the first application of the k-induction algorithm.
Review: Proof by Induction

• Show that a property $P$ is true for $n \geq k$
  ▪ Base case
  ▪ Step case or inductive step

• Suppose that
  ▪ $P(k)$ is true for a fixed constant $k$
    o Often $k = 0$
  ▪ $P(n) \iff P(n+1)$ for all $n \geq k$

• Then $P(n)$ is true for all $n \geq k$
Induction Example: Gaussian Closed Form

- Prove $0+1 + 2 + 3 + \ldots + n = n(n+1) / 2$

  - **Basis:** If $n = 0$, then $0 = 0(0+1) / 2$

  - **Inductive hypothesis:** Assume that $0 + 1 + 2 + 3 + \ldots + k = k(k+1) / 2$

  - **Inductive step:** show that if $P(k)$ holds, then also $P(k+1)$ holds

    $(0+1+2+\ldots+k)+(k+1) = (k+1)((k+1)+1) / 2$

      - Using the induction hypothesis that $P(k)$ holds:

        $k(k+1)/2 + (k+1)$.

        $k(k+1)/2 + (k+1) = [k(k+1)+2(k+1)]/2 = (k^2 +3k+2)/2 = (k+1)(k+2)/2$

        $= (k+1)((k+1)+1)/2$  hereby showing that indeed $P(k+1)$ holds
Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$

  - **Basis:** show that $a^0 = (a^{0+1} - 1)/(a - 1)$
    
    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  
  - **Inductive hypothesis:**
    
    - Assume $a^0 + a^1 + \ldots + a^k = (a^{k+1} - 1)/(a - 1)$
  
  - **Inductive step:** show that if $P(k)$ holds, then also $P(k+1)$ holds:
    
    $a^0 + a^1 + \ldots + a^{k+1} = a^0 + a^1 + \ldots + a^k + a^{k+1}$
    
    $= (a^{k+1} - 1)/(a - 1) + a^{k+1} = (a^{k+1+1} - 1)/(a - 1)$
The Substitution Method for Solving Recurrences

• The substitution method
  ▪ A.k.a. the “making a good guess method”
  ▪ Guess the form of the answer, then use induction to find the constants and show that solution works
  ▪ Examples:

Recurrence: $T(n) = 2T\left(\left\lfloor n/2 \right\rfloor \right) + n$

Guess the solution is: $T(n) = O(n \log n)$

Prove that $T(n) \leq cn \log n$ for some $c > 0$ using induction and for all $n \geq n_0$
The Substitution Method

• Induction requires us to show that the solution remains valid for the limit conditions

• **Base case**: show that the inequality holds for some $n$ sufficiently small
  - If $n = 1 \rightarrow T(1) \leq c \times 1 \times \log 1 = 0$
  - However, $T(n) = 2T(\lfloor n/2 \rfloor) + n \therefore T(1) = 1$
  - But

\[ \begin{align*} 
  n=2 & \rightarrow T(2) = 2T(1) + 2 = 4 \quad \text{and} \quad cn \log n = c \times 2 \times \log 2 = 2c \\
  n=3 & \rightarrow T(3) = 2T(1) + 3 = 5 \quad \text{and} \quad cn \log n = c \times 3 \times \log 3
\]
The Substitution Method

• We can start from $T(2)=4$ or $T(3)=5$ using some $c \geq 2$, given that

- $T(n) \leq cn \log n$
- $T(2) \leq c2 \log 2$
- $T(3) \leq c3 \log 3$

• Base case holds w.r.t. the asymptotic notation:
  - $T(n) \leq cn \log n$ for $n \geq n_0$

• Hint: extend the boundary conditions to make the inductive hypothesis count for small values of $n$
The Substitution Method

- **Inductive hypothesis:**
  - Assume that $T(n) \leq c \ n \log n$ for $c > 0$ holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding

  $$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$
The Substitution Method

• Induction: Inequality holds for $n$

$T(n) = 2T\left(\lfloor n/2 \rfloor \right) + n$

$\leq 2(c \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor) + n$

$\leq cn \log (n/2) + n$

$= cn \log n - cn + n$

$\leq cn \log n$
The Substitution Method

• Induction: Inequality holds for $n$

\[
T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n \\
\leq 2(c \left\lfloor \frac{n}{2} \right\rfloor \log\left\lfloor \frac{n}{2} \right\rfloor) + n \\
\leq cn \log(n/2) + n \\
= cn(\log n - \log 2) + n \\
= cn \log n - cn + n \\
\leq cn \log n \quad (\text{holds for } c \geq 1, \text{ upper bound analysis})
\]
Making a good guess

• Guessing a solution takes experience / creativity
  ▪ Use heuristics to help you become a good guesser
  ▪ Use recursion trees

• Consider this example: $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$
  ▪ What is your guess here? $T(n) = O(n \log n)$
  ▪ The term “17” cannot substantially affect the solution

• Prove loose upper and lower bounds, e.g.:
  ▪ Start with $T(n) = \Omega(n)$, then $T(n) = O(n^2)$ until we converge to $T(n) = O(n \log n)$
Changing Variables

- A little algebraic manipulation can make unknown recurrence similar to one you have seen before

\[
T(n) = 2T(\sqrt{n}) + \log n \quad m = \log n \quad : \quad n = 2^m
\]
\[
T(2^m) = 2T(2^{m/2}) + m \quad S(m) = T(2^m)
\]
\[
S(m) = 2S(m/2) + m \quad \text{similar to} \quad T(n) = 2T(n/2) + n
\]
\[
T(n) = T(2^m) = S(m) \quad S(m) = O(m \log m)
\]
\[
= O(m \log m) = O(\log n \log \log \log n)
\]

Note that \(\log 2^m = m\) and \(\log 2^{m/2} = m/2\)
Exercise: Recursive Binary Search

• Given the recurrence equation of the binary search:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1; \\
1 + T(n/2) & \text{if } n > 1;
\end{cases} \]

• Guess the solution is: \( T(n) = O(\log n) \)

• Prove that \( T(n) \leq c \log n \) for some \( c > 0 \) using induction and for all \( n \geq n_0 \)
Summary

• Many useful algorithms are recursive in structure:
  ▪ to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems
• We also analysed recurrences using the substitution method
• We have demonstrated that $T(n)$ of merge sort is $\Theta(n \log n)$, where $\log n$ stands for $\log_2 n$