Traversal Strategies

Specification and Efficient Implementation

(Graph Theory of OOP/OOD)
Introduction

- Define subgraphs succinctly
- Define path sets succinctly
- Applications
  - writing adaptive programs
  - marshaling objects
  - storing objects, persistent objects
Applications of Traversal Strategies

• Defining high-level artifact in terms of a low-level artifact without committing to details of low-level artifact in definition of high-level artifact. Low-level artifact is parameter to definition of high-level artifact.

• Exploit structure of low-level artifact.
Applications of Traversal Strategies

• Application 1
  – High-level: Adaptive program
  – Low-level: Class graph

• Application 2 (see paper with Dean Allemang)
  – High-level: High-level API
  – Low-level: Low-level API
Similar to a function definition accessing parameter generically

• **High-level**(Low-level)
  – *High-level* does not refer to all information in *Low-level* but *High-level*(Low-level) contains details of *Low-level*.
  – *High-level* uses generic operations to extract information from *Low-level*.
  – Traversal strategies are the generic operations.
Applications of traversal strategies

• Specify mapping between graphs (adaptors)
  – Advantage: mapping does not have to refer to
    details of lower level graph → robustness

•Specify traversals through graphs
  – Specification does not have to refer to details of
    traversed graph → robustness

• Specify function compositions
  – without referring to detail of API → robustness
Applications of traversal strategies

• Specify range of generic operations such as comparing, copying, printing, etc.
  – without referring to details of class graph → robustness. Used in Demeter/Java. Used in distributed computing: marshalling, D, AspectJ, Xerox PARC
Summary of lecture

• Concept of traversal strategies
• How to write traversal strategies
• Detailed meaning of strategies
• Complexity of compilation: polynomial in the size of strategy and class graph
• How to implement traversals manually
• Define concepts of class and object graph.
Summary of lecture

• Previous approaches: less general and their compilation algorithms were of exponential complexity.

• Show need for parameters in traversal methods.
Overview

• Use structure in graphs to express subgraphs and path sets in those graphs.
• Gain: writing programs in terms of strategies yields shorter and more flexible programs.
• Does not work well on dense graphs and graphs with self loops.
Connections

- strategy graphs, class graphs, object graphs
- simple class graphs, flat class graphs
- natural correspondence between paths in class graphs and object graphs
- compilation algorithm has some similarity with simulation of a non-deterministic automaton
Graphs used

• object graphs
• class graphs
• strategy graphs
• traversal graphs
• propagation graphs = folded traversal graphs

Therefore, introduce graph machinery for multiple use.
Simplified form of theory

• Focus on class graphs with one kind of nodes and one kind of edges.
• Roles graphs play in OOD.
• Define concept of path expansion.
• Define concept of path set.
• Introduce graph relationships and connections between them.
Underlying ideas

• Graph1 refinement Graph2
• Graphs can play the following roles:
  – interface class graph
  – (application) class graph
  – positive strategy graph
    • have a source and a target
$G_1$ compatible $G_2$

Compatible: connectivity of $G_2$ is in $G_1$
$G_1$ refinement $G_2$

refinement: connectivity of $G_2$ is in $G_1$ and $G_1$ contains no new connections in terms of nodes of $G_2$
Roles graphs play in OOD under refinement relations

- **Small graph**
  - G
  - PSG
  - PSG

- **Big graph**
  - G
  - PSG
  - G

G: class graph (CG) or interface class graph (ICG). ICG is a view on a class graph. PSG: positive strategy graph.
Roles graphs play in OOD under refinement relations

<table>
<thead>
<tr>
<th>SMALL</th>
<th>BIG</th>
<th>INTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>CG</td>
<td>evolution</td>
</tr>
<tr>
<td>ICG</td>
<td>CG</td>
<td>view (abstr.)</td>
</tr>
<tr>
<td>CG</td>
<td>ICG</td>
<td>view (roles)</td>
</tr>
<tr>
<td>ICG</td>
<td>ICG</td>
<td>layering</td>
</tr>
<tr>
<td>PSG</td>
<td>CG</td>
<td>AP</td>
</tr>
<tr>
<td>PSG</td>
<td>ICG</td>
<td>improved AP</td>
</tr>
<tr>
<td>PSG</td>
<td>PSG</td>
<td>subtraversal</td>
</tr>
</tbody>
</table>
Roles graphs play in OOD under refinement relations

<table>
<thead>
<tr>
<th>application</th>
<th>$PSG()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CG$</td>
<td>AP</td>
</tr>
<tr>
<td>$ICG$</td>
<td>Better AP, class graph views</td>
</tr>
<tr>
<td>$PSG$</td>
<td>Meta strategy</td>
</tr>
</tbody>
</table>
Theory of Strategy Graphs

• Palsberg/Xiao/Lieberherr: TOPLAS ‘95
• Palsberg/Patt-Shamir/Lieberherr: Science of Computer Programming 1997
• Lieberherr/Patt-Shamir: Strategy graphs, 1997 NU TR
• Lieberherr/Patt-Shamir: Dagstuhl ‘98 Workshop on Generic Programming (LNCS)
Strategy graph and base graph are directed graphs

Key concepts

• Strategy graph $S$ with source $s$ and target $t$ of a base graph $G$. $Nodes(S)$ subset $Nodes(G)$ (Embedded strategy graph).

• A path $p$ is an expansion of path $p'$ if $p'$ can be obtained by deleting some elements from $p$.

• $S$ defines path set in $G$ as follows: $PathSet_{st}(G,S)$ is the set of all $s$-$t$ paths in $G$ that are expansions of any $s$-$t$ path in $S$. 
Key concepts

• A path $p$ in $G$ is an *expansion* of path $p'$ in $S$ if $\text{OrderedNodes}(p')$ can be obtained by deleting some elements from $\text{OrderedNodes}(p)$.

• $\text{OrderedNodes}(p)$ is the ordered sequence of nodes in $p$ in the order the nodes appear in $p$.

• *Recall*: $\text{Nodes}(S) \subseteq \text{Nodes}(G)$
In other words ...

• Let $S$ be a strategy graph, let $G$ be a base graph with $\text{Nodes}(S) \subseteq \text{Nodes}(G)$. Given a strategy-graph path $p = \langle a_0 \ a_1 \ \ldots \ a_n \rangle$, we say that a path $p'$ in $G$ is a expansion of $p$ if there exist paths $p_1, \ldots, p_n$ in $G$ such that $p' = p_1 \cdot p_2 \cdot \ldots \cdot p_n$ and: For all $0<i<n+1$, $\text{Source}(p_i)=a_{i-1}$ and $\text{Target}(p_i)=a_i$. 
$PathSet(G, S)$

$A = s$

$D \rightarrow E \rightarrow F = t$

$B \rightarrow C$

$S$

$G$

$F \rightarrow E \rightarrow C$

$D \rightarrow B$

$A$
Strategy graph and base graph are directed graphs

Key concepts

• A strategy graph $G_1$ is a *path-set-refinement* of a strategy graph $G_2$ if for all base graphs $G_3$: $\text{PathSet}(G_3, G_1) \subseteq \text{PathSet}(G_3, G_2)$.

• Surprise?: co-NP-complete

• See recent paper with Boaz Patt-Shamir.
$G_1$ path-set-refinement $G_2$
Strategy graph and base graph are directed graphs

Key concepts

• A strategy graph $G_1$ is an *expansion* of a strategy graph $G_2$ if for any path $p_1$ (from $s$ to $t$) in $G_1$ there exists a path $p_2$ (from $s$ to $t$) in $G_2$ such that $p_1$ is an expansion of $p_2$.

• Surprise? Co-NP-complete. Equivalent to path-set-refinement.

• See recent paper with Boaz Patt-Shamir.
$G_1$ path-set-refinement $G_2$

$G_1$ expansion $G_2$
Key concepts

• Let $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ be directed graphs with $V_2$ a subset of $V_1$. Graph $G_1$ is a refinement of $G_2$ if for all $u,v$ in $V_2$ we have that $(u,v)$ in $E_2$ if and only if there exists a path in $G_1$ between $u$ and $v$ which does not use in its interior a node in $V_2$.

• Polynomial. Implies path-set-refinement and expansion
Refinement means: no surprises

not $G_1$ refinement $G_2$

$G_1$ expansion $G_2$
Refinement means: no surprises

$G_1$ refinement $G_2$
Reﬁnement means: no surprises

\[
\begin{align*}
G_1 \text{ expansion } G_2 \\
\text{not } G_1 \text{ reﬁnement } G_2
\end{align*}
\]
Connection to Demeter/Java

• How to enforce a refinement relationship between class graph and strategy graph?
Surprise paths

- \{A \rightarrow B \ B \rightarrow C\}
- surprise path: A P C Q A B R A S C
- eliminate surprise paths:
  \{A\rightarrow B \text{ bypassing } \{A,B,C\}\}
  \{B\rightarrow C \text{ bypassing } \{A,B,C\}\}
- bypass edges into A and bypass edges out of B
- \{A\rightarrow A \text{ bypassing } A\}\}
Wysiwig strategies

• Avoid surprise paths
• Bypass all classes mentioned in strategy on all edges of the strategy graph
• Some users think that wysiwig strategies are easier to work with
• For wysiwig strategies, if class graph has a loop, strategy must have a loop.
Example: In-laws

Person = Brothers Sisters Status.
Status : Single | Married.
Single = .
Married = <marriedTo> Person.
Brothers ~ {Person}.
Sisters ~ {Person}.
Example: In-laws

{Person -> Married bypassing Person
  Married -> spouse:Person bypassing Person
  spouse:Person -> Brothers bypassing Person
  spouse:Person -> Sisters bypassing Person
  Brothers -> brothers_in_law:Person bypassing Person
  Sisters -> sisters_in_law:Person bypassing Person
}

Note: not yet implemented
Traversals and naming roles (not implemented)

- Can use strategy graphs to name roles which objects play depending on when we get to them during traversal.
Traversal dependent roles

Class graph with super-imposed strategy graph

Strategy graph

10/28/98 AOO / Demeter
Learning map

numbers: order of coverage

1. graph paths labeled
3. object graph
4. object traversal defined by concrete path set
7. Algorithm 1
   in: strategy + class graph
   out: traversal graph

2. class graph
5. strategy graph
6. name map constraint map
9. traversal graph
10. propagation graph
11. zig-zags short-cuts
12. Algorithm 2
   in: traversal + object graph
   out: object traversal

correspondences
X: class path - concrete path
Y: object path - concrete path
traversal path - class path

FROM-TO computation
Graphs and paths

• Directed graph: \((V,E)\), \(V\) is a set of nodes, \(E \subseteq V \times V\) is a set of edges.

• Directed labeled graph: \((V,E,L)\), \(V\) is a set of nodes, \(L\) is a set of labels, \(E \subseteq V \times L \times V\) is a set of edges.

• If \(e = (u,l,v)\), \(u\) is source of \(e\), \(l\) is the label of \(e\) and \(v\) is the target of \(e\).
Graphs and paths

• Given a directed labeled graph: $(V,E,L)$, a node-path is a sequence $p = <v_0v_1…v_n>$ where $v_i \in V$ and $(v_{i-1}, l_i, v_i) \in E$ for some $l_i \in L$.

• A path is a sequence $<v_0 l_1 v_1 l_2 … l_n v_n>$, where $<v_0 … v_n>$ is a node-path and $(v_{i-1}, l_i, v_i) \in E$. 
Graphs and paths

• In addition, we allow node-paths and paths of the form $\langle v_0 \rangle$ (called trivial).

• Unlabeled graphs have only node paths.

• First node of a path or node-path $p$ is called the source of $p$, and the last node is called the target of $p$, denoted $Source(p)$ and $Target(p)$, respectively. Other nodes: interior.
Graphs and paths

• \( P_G(u,v) \): set of all paths in \( G \) with source \( u \) and target \( v \).

• concatenation of paths: If \( p_1 = <v_0 \ldots l_i v_i> \) and \( p_2 = <v_i l_{i+1} \ldots v_n> \) are paths, we define the concatenation \( p_1 . p_2 = <v_0 \ldots l_i v_i l_{i+1} v_{i+1} \ldots v_n> \). Only one copy of meeting point \( v_i \). Similar for node-paths.
Class graphs / object graphs

- Set of class names $CC$. Each class name is either abstract or concrete.
- Set of field names $LL$.
- Distinguished symbol ♦ not in $LL$ for labeling subclass edges.
Class graphs / object graphs

• Class graphs are graphs $G = (V,E,L)$ such that
  
  – $V$ is a subset of $CC$ (nodes are class names).
  
  – $L$ is a subset of $LL \cup \{\diamond\}$.
  
  – For all $v$, field names of edges outgoing from $v$ are distinct.
  
  – The set of edges labeled by $\diamond$ is acyclic.
Class graphs / object graphs

- Edges labeled by field names are called reference edges, edges labeled by $\diamondsuit$ are called subclass edges.
- Reflexive notion of a superclass: Given a class graph $G = (V, E, L)$, $v \in V$ is a superclass of $u \in V$ if there is a possibly empty path of subclass edges from $v$ to $u$.
- Ancestry of $v$: set of all superclasses of $v$. 
Class graphs / object graphs

- Multiple inheritance conflicts are disallowed: we require that for all nodes $v$, if $v$ has two superclasses $u$ and $w$ with outgoing edges labeled by the same field name, then either $u$ is in the ancestry of $w$ or $w$ in the ancestry of $u$. 

\[ \text{disallowed} \]
Class graphs / object graphs

- Induced references of a class: the set of all reference edges outgoing from its ancestry.
- Usual overriding rule: for each field name $f$ used in edges outgoing from the ancestry of $v$, only the edge labeled $f$ closest to $v$ is in the induced references.
- Note: Induced references = direct references $\cup$ inherited references
Object graphs

• Model instantiations of class graphs.
• An object graph is a labeled directed graph $O = (V', E', L')$ where nodes are called objects and $L'$ is a subset of $LL$.
• An object graph $O = (V', E', L')$ is an instance of a class graph $G = (V, E, L)$ under a function $Class$ mapping objects to classes, if the following conditions hold:
Object graphs

– For all objects $o \in V'$, $\text{Class}(o)$ is concrete.
– For each object $o \in V'$, the set of field names outgoing from $o$ is exactly the set of field names of the induced references of $\text{Class}(o)$.
– For each edge $(o,f,o') \in E'$, $\text{Class}(o)$ has an induced reference edge $(v,f,u)$ such that $v$ is a superclass of $\text{Class}(o)$ and $u$ is a superclass of $\text{Class}(o')$. 
Object graphs

For each edge \((o,f,o') \in E'\), \(\text{Class}(o)\) has an induced reference edge \((v,f,u)\) such that \(v\) is a superclass of \(\text{Class}(o)\) and \(u\) is a superclass of \(\text{Class}(o')\).
Natural correspondence

- So far, class graphs are very general: multiple inheritance is allowed, superclasses are not forced to be abstract.
- Without loss of generality: consider only a limited set of class graphs: simple class graphs. In simple class graphs: Easy mapping between class graph paths and object graph paths.
Simple class graphs

• We assume that class graphs are simple.

• A class graph $G = (V,E,L)$ is simple, if
  – for all edges $(u,f,v) \in E$, we have that $f = \diamond$ if and only if $u$ is abstract, and
  – for all edges $(u,\diamond,v) \in E$, we have that $v$ is concrete.
Simple class graphs

– for all edges \((u,f,v) \in E\), we have that \(f = \Diamond\) if and only if \(u\) is abstract, and

• Says that (1) all edges outgoing from abstract classes are subclass edges and (2) all edges outgoing from concrete classes are reference edges.

• (1) flatness

• (2) concrete classes have no subclasses
Simple class graphs

– for all edges \((u, \hat{\ell}, v) \in E\), we have that \(v\) is concrete.

• All subclass edges are incoming into concrete classes.
Simple class graphs

• Three situations are forbidden:
  – concrete superclasses
    • introduce abstract classes and rearrange
  – common parts
    • flatten
  – inheritance chains
    • for abstract $v$: find all concrete classes $u$ reachable from $v$ using subclass edges only. Add a subclass edge $(v <> u)$ if one does not exist. Delete subclass edges leading to abstract classes.
Proposition: nothing lost with simple class graphs

• Let $G = (V, E, L)$ be an arbitrary class graph. Then there exists a class graph $Simplify(G) = (V', E', L)$ such that an object graph $O$ is an instance of $G$ if and only if $O$ is an instance of $Simplify(G)$. Moreover, $|V'| = O(|V|)$, $|E'| = O(|E|^2)$.

• Introduces multiple inheritance.
Natural correspondence $X$

- A concrete path is an alternating sequence of concrete class names and field names (excluding ◊).
- Natural correspondence $X$: map path $p$ in class graph to concrete path $X(p)$ by omitting abstract classes and subclass edges.
Natural correspondence $Y$

• Map an object-graph path $p'$ to a concrete path $Y(p')$ by taking the sequence of class names (under the $Class$ function) and field names.

• Motivation: If $p$ is a path in class graph $G$, then there is some object graph $O$ which is an instance of $G$, and a path $p'$ in $O$, such that $X(p) = Y(p')$. 
Natural correspondence

- Simple class graphs have a simple relationship between class graph paths and object graph paths.

v is concrete, v’ is abstract
Learning map

1. Graph paths labeled
2. Class graph
3. Object graph
4. Object traversal defined by concrete path set
5. Strategy graph
6. Name map
7. Algorithm 1
   in: strategy + class graph
   out: traversal graph
8. FROM-TO computation
9. Traversal graph
10. Propagation graph
11. Zig-zags short-cuts
12. Algorithm 2
   in: traversal + object graph
   out: object traversal

Correspondences:
- X: class path - concrete path
- Y: object path - concrete path
- Traversal path - class path

Generalization and other relationships

Numbers: order of coverage

10/28/98
Definition of traversals

• For a set of sequences $R$:
  – $head(R) = \{x \mid \text{there exists } p: x.p \in R\}$
  – $tail(R,x) = \{p \mid x.p \in R\}$

• Assume a total order $<$ on set of field names $LL$. 
Definition of traversals

• *traversing* \(O\) *from* \(o\) *guided by* \(R\) *produces* \(H = \)
  traversing the object graph \(O\) starting with \(o\), and
  guided by a concrete path set \(R\), yields traversal
  history \(H\).

• Most of the time we can think of \(R\) as being
  defined by a subgraph of the original class graph.

• When object is visited, a method may be invoked.
Definition of traversals

If for $i$ from 1 to $n$

- traversing $O$ from $o_i$ guided by $\text{tail(tail}(R, \text{Class}(o)),l_i)$ produces $H_i$ then
- traversing $O$ from $o$ guided by $R$ produces $H_1, \ldots, H_n$ provided $\text{head(tail}(R, \text{Class}(o))) = \{l_i \mid i = 1 \ldots n\}$, $(o,l_i,o_i) \in O$ and $l_j < l_k$ for $0 < j < k < n+1$. 
Definition of traversals

• If \( \text{tail}(R, \text{Class}(o)) = \text{empty set} \), then \textit{traversing} \( O \) \textit{from} \( o \) \textit{guided by} \( R \) \textit{produces} \( \varepsilon \), where \( \varepsilon \) denotes the empty history.
Remarks about traversals

• If object graph is cyclic, traversal is not well defined.
• Traversals are opportunistic: As long as there is a possibility for success (i.e., getting to the target), the branch is taken.
• Traversals do not look ahead. Visitors must delay action appropriately.
Strategies: traversal specification

• Strategies select class-graph paths and then derive concrete paths by applying the natural correspondence.

• Traversals are defined in terms of sets of concrete paths.

• A strategy selects class graph paths by specifying a high-level topology which spans all selected paths.
Strategies

• A strategy $SS$ is a triple $SS = (S, s, t)$, where $S = (C, D)$ is a directed unlabeled graph called the strategy graph, where $C$ is the set of strategy-graph nodes and $D$ is the set of strategy-graph edges, and $s, t \in C$ are the source and target of $SS$, respectively.
Strategies, name map

• Let $SS = (C,D)$ be a strategy graph and let $G = (V,E,L)$ be a class graph. A name map for $SS$ and $G$ is a function $N:C \to V$. If $p$ is a sequence of strategy graph nodes, then $N(p)$ is the sequence of class nodes obtained by applying $N$ to each element of $p$.

• Intuitively, strategy graph edge “a to b” represents paths from $N(a)$ to $N(b)$.
Strategies, expansion

- Given a sequence $p$, a sequence $p'$ is an expansion of $p$ if $p'$ can be obtained by inserting elements between the elements of $p$. 
Strategies, path sets

Let $SS = (S,s,t)$ be a strategy, let $G = (V,E,L)$ be a class graph, and let $N$ be a name map for $SS$ and $G$. The set of concrete paths $PathSet[SS,G,N]$ is $\{X(p') \mid p' \in P_G(N(s),N(t)) \text{ and there exists } p \in P_S(s,t) \text{ such that } p' \text{ is an expansion of } N(p)\}$. 
Strategies, constraint map

• Need negative constraints

• Given a class graph $G = (V,E,L)$, an element predicate $EP$ for $G$ is a predicate over $V \cup E$. Given a strategy $SS$, a function $B$ mapping each edge of $SS$ to an element predicate is called a constraint map for $SS$ and $G$. 
Strategies, constraint map

• Let $S$ be a strategy graph, let $G$ be a class graph, let $N$ be a name map and let $B$ be a constraint map for $S$ and $G$. Given a strategy-graph path $p = <a_0 \ a_1 \ \ldots \ a_n>$, we say that a class graph path $p'$ is a satisfying expansion of $p$ with respect to $B$ under $N$ if there exist paths $p_1, \ \ldots, p_n$ such that $p' = p_1 \cdot p_2 \ \ldots \ p_n$ and:
Strategies, constraint map

– For all $0 < i < n+1$, $Source(p_i) = N(a_{i-1})$ and $Target(p_i) = N(a_i)$.

– For all $0 < i < n+1$, the interior elements of $p_i$ satisfy the element predicate $B(a_{i-1}, a_i)$. 
Strategies

• Many ways to decompose a path.
• Element constraints never apply to the ends of the subpaths.
• from A bypassing \{A, B\} to B
Strategies, path sets

- Let $SS = (S,s,t)$ be a strategy, let $G = (V,E,L)$ be a class graph, and let $N$ be a name map for $SS$ and $G$ and let $B$ be a constraint map for $S$ and $G$. The set of concrete paths $PathSet[SS,G,N,B]$ is $\{X(p') \mid p' \in P_G(N(s),N(t)) \text{ and there exists } p \in P_S(s,t) \text{ such that } p' \text{ is an expansion of } N(p) \text{ w.r.t. } B\}$. 
Strategies

- \( \text{PathSet}[SS, G, N] = \text{PathSet}[SS, G, N, B_{TRUE}] \) for the constraint map \( B_{TRUE} \) which maps all strategy graph edges to the trivial element predicate that is always TRUE.

- Encapsulated strategies: want a clean separation between strategy graphs and class graphs.
Strategies

A \quad \text{bypassing B} \quad C

n1 \quad \text{bypassing n3} \quad n2

Name map:

n1 \quad A
n2 \quad C
n3 \quad B
A \quad \text{Company}
B \quad \text{Retirement}
C \quad \text{Salary}

In Demeter/Java:

name map is identity
Strategies

• Are used in adaptive programs.
• Adaptive programs are expressed in terms of class-valued and relation-valued variables. Class graph not known when program is written.
• Wildcard notation in predicate specification: bypassing \((*, f, *)\).
End of fourth session
Learning map

correspondences
X: class path - concrete path
Y: object path - concrete path
traversal path - class path

FROM-TO computation

numbers: order of coverage

generalization
other relationships

Algorithm 1
in: strategy + class graph
out: traversal graph

Algorithm 2
in: traversal + object graph
out: object traversal

object traversal defined by concrete path set
Compilation of strategies

• Compilation problem
  – INPUT: A strategy $SS = (S, s, t)$, a simple class graph $G = (V, E, L)$, a name map $N$ for $S$ and $G$, and a constraint map $B$ for $S$ and $G$.
  – OUTPUT: A set of methods such that for any object graph $O$, invoking the traversal method at an object $o \in O$ yields a traversal history $H$ satisfying $\text{traversing } O \text{ from } o \text{ guided by } PathSet[SS, G, N, B]$ produces $H$. 
What we tried.

- Path set is represented by subgraph of class graph, called propagation graph. Propagation graph is translated into a set of methods. Works in many cases. Two important cases which do not work:
  - short-cuts
    - zig-zags
Short-cut

class graph

strategy graph with name map

strategy:
{A -> B
B -> C}

A

B

C

x

A

B

C

x

b

c

0..1

0..1

x

x

C

x

b

c

A

B

C

x

b

c

0..1

x

C

propagation graph
1 + 1 = 3

Short-cut

Incorrect traversal code:
class A { void t() { x.t(); } }
class X { void t() { if (b!==null) b.t(); c.t(); } }
class B { void t() { x.t(); } }
class C { void t(){} }

Correct traversal code:
class A { void t() { x.t(); } }
class X { void t() { if (b!==null) b.t2();
    void t2() { if (b!==null) b.t2(); c.t2(); } }
} class B { void t2() { x.t2(); } }
class C { void t2(){} }

strategy: { A → B
             B → C }

strategy graph with name map

propagation graph

A  B  C
x  c
0..1
Short-cut

strategy: 
\{A \rightarrow B \\
B \rightarrow C\}

class graph

traversal method \(t\)

traversal method \(t2\)

abstract representation of traversal code

thick edges with incident nodes: traversal graph
Zig-zags

strategy graph
with name map

< A C D E G > is excluded

At a D-object need to remember how we got there. Need argument for traversal methods. Represent traversal by tokens in traversal graph.
Compilation of strategies

• Two parts
  – construct graph which expresses the traversal $\text{PathSet}[SS,G,N,B]$ in a more convenient way: traversal graph $TG(SS,G,N,B)$. Represents allowed traversals as a “big” graph.
  – code for traversal methods which uses $TG(SS,G,N,B)$. 
Compilation of strategies

• Idea of traversal graph:
  – Paths defined by \texttt{from A to B} can be represented by a subgraph of the class graph. Compute all edges reachable from A and from which B can be reached. Edges in intersection form graph which represents traversal.
  – Generalize to any strategies: Need to use \textit{big} graph but above \texttt{from A to B} approach will work.
Compilation of strategies

• Idea of traversal graph:
  – traversal graph is “big brother” of propagation graph
  – is used to control traversal
  – FROM-TO computation: Find subgraph consisting of all paths from A to B in a directed graph: Fundamental algorithm for traversals
  – Traversal graph computation is FROM-TO computation.
Strategy behind Strategy

• Instead of developing a specialized algorithm to solve a specific problem, modify the data until a standard algorithm can do the work. May have implications on efficiency.

• In our case: use FROM-TO computation.
FROM-TO computation

• Problem: Find subgraph consisting of all paths from A to B in a directed graph.
  – Forward depth-first traversal from A
    • colored in red
  – Backward depth-first traversal from B
    • colored in blue
  – Select nodes and edges which are colored in both red and blue.
Variations

• reverse edges during first traversal in copy of graph
• could also use breadth-first traversal
Depth-first traversal

- Topological sorting
- Cycle checking
- Compute strongly-connected components
- Shortest paths
Traversal graph computation

Algorithm 1

• Let the strategy graph $S = (C,D)$ and let the strategy graph edges be $D = \{e_1, e_2, \ldots, e_k\}$.

• 1. Create a graph $G'=(V',E')$ by taking $k$ copies of $G$, one for each strategy graph edge. Denote the $i$th copy as $G^i = (V^i, E^i)$.

• The nodes in $V^i$ and edges in $E^i$ are denoted with superscript $i$, as in $v^i$, $e^i$, etc.
Why $k$ copies?

- Mimics using $k$ distinct traversal method names.
- Run-time traversals need enough state information.
Traversel graph computation

- Each class-graph node $v$ corresponds to $k$ nodes in $V'$, denoted $v^1, \ldots, v^k$.
- Extend $Class$ mapping to apply to nodes of $G'$ by setting $Class(v^i) = v$, where $v^i \in V$ and $v \in V$. 
Preview of step 2

• Link the copied class graphs through temporary use of intercopy edges.
• Each strategy graph node is responsible for additional edges in the traversal graph.
• If strategy graph node has one incoming and one outgoing edge, one edge is added.
Preview of step 2

- Redirection of edges from one copy to the next:

\[
\begin{array}{c}
A \\
\downarrow f \\
C \\
\end{array} 
\]

intercopy edge

\[
\begin{array}{c}
f \\
C \\
\end{array} 
\]

f may be ◊
Traversal graph computation

• 2.a For each strategy-graph node \( a \in C \): Let \( I = \{ei_1, \ldots, ei_n\} \) be the strategy-graph edges incoming into \( a \), and let \( O=\{eo_1, \ldots, eo_m\} \) be the set of strategy graph edges outgoing from \( a \). Let \( N(a)=v \in V \). Add \( n \) times \( m \) edges \( v^j \) to \( v^l \) for \( j=1, \ldots,n \) and \( l = 1, \ldots,m \). Call these edges intercopy edges.
Traversal graph computation

- 2.b For each node $v^i \in G'$ with an outgoing intercopy edge: Add edges $(u^i, f, v^j)$ for all $u^i$ such that $(u^i, f, v^i) \in E^i$, and for all $v^j$ which are reachable from $v^i$ through intercopy edges only.

- 2.c Remove all intercopy edges added in step 2.a.
Preview of step 3

- Delete edges and nodes which we do not want to traverse.
Traversal graph computation

3. For each strategy-graph edge $e_i = \text{from } a \text{ to } b$: Let $N(a) = u$ and $N(b) = v$. Remove from the subgraph $G^i$ all elements which do not satisfy the predicate $B(e_i)$, with the exception of $u^i$ and $v^i$.

- $V^i = \{v^i, u^i\} \cup \{w^i \mid B(e_i)(w) = \text{TRUE}\}$, and
- $E^i = \{(w^i, l, y^i) \mid B(e_i)(w, l, y) = B(e_i)(w) = B(e_i)(y) = \text{TRUE}\}$. 
Preview of step 4

• Get ready for the FROM-TO computation in the traversal graph: need a single source and target.
Traversing graph computation

• 4.a Add a node $s^*$ and an edge $(s^*, N(s)^i)$ for each edge $e_i$ outgoing from $s$ in the strategy graph, where $s$ is the source of the strategy.

• 4.b Add a node $t^*$ and an edge $(N(t)^i, t^*)$ for each edge $e_i$ incoming into $t$ in the strategy graph, where $t$ is the target of the strategy.
Traversal graph computation

• 4.c Mark all nodes and edges in $G'$ which are both reachable from $s^*$ and from which $t^*$ is reachable, and remove unmarked nodes and edges from $G'$. Call the resulting graph $G''=(V'',E'')$.

• The above is an application of the FROM-TO computation.
Traversals graph computation

5. Return the following objects:
   - The graph obtained from $G''$ after removing $s^*$ and $t^*$ and all their incident edges. This is the traversal graph $TG(SS,G,N,B)$.
   - The set of all nodes $v$ such that $(s^*,v)$ is an edge in $G''$. This is the start set, denotes $T_s$.
   - The set of all nodes $v$ such that $(v,t^*)$ is an edge in $G''$. This is the finish set, denoted $T_f$. 
Traversal graph properties

• If $p$ is a path in the traversal graph, then under the extended $Class$ mapping, $p$ is a path in the class graph. (Roughly: traversal graph paths are class graph paths.)
Short-cut

strategy:
\{A \rightarrow B
B \rightarrow C\}

abstract representation of traversal code

class graph

traversal method t

traversal method t2

thick edges with incident nodes: traversal graph
Can now think in terms of a graph and need no longer path sets. But graph may be bigger.

**Traversal graph properties**

- Let $SS$ be a strategy, $G$ a class graph, $N$ a name map, and let $B$ be a constraint map. Let $TG = TG(\text{SS}, \text{G}, \text{N}, \text{B})$ be the traversal graph and let $T_s$ be the start set and $T_f$ the finish set generated by algorithm 1. Then $X(\text{Class}(P_{TG}(T_s, T_f))) = \text{PathSet}[\text{SS}, \text{G}, \text{N}, \text{B}]$. (Roughly: Paths from start to finish in traversal graph are the paths selected by strategy.)
Short-cut

abstract representation of traversal code

strategy:
\{ A \rightarrow B \\
   B \rightarrow C \}
Learning map

generalization
other relationships
numbers: order of coverage

Algorithm 1
in: strategy + class graph
out: traversal graph

Algorithm 2
in: traversal + object graph
out: object traversal

correspondences
X: class path - concrete path
Y: object path - concrete path
traversal path - class path

FROM-TO computation

object graph
strategy graph
class graph
traversal graph
name map
constraint map
propagation graph
zig-zags short-cuts

object traversal defined by concrete path set

1. graph paths labeled
8. FROM-TO computation
9. traversal graph
10. propagation graph

4. object traversal defined by concrete path set
6. name map
7. Algorithm 1
   in: strategy + class graph
   out: traversal graph
11. zig-zags short-cuts

3. object graph
5. strategy graph
Traversal methods algorithm
Algorithm 2

- Idea is to traverse an object graph while using the traversal graph as a road map.
- Maintain set of “tokens” placed on the traversal graph.
- May have several tokens: path leading to an object may be (under $Y$) a prefix of several distinct paths in $PathSet[SS,G,N,B]$. 
Traversals method algorithm

- Traversal method $\text{Traverse}(T)$, where $T$ a set of tokens, i.e., a set of nodes in the traversal graph.
- When $\text{Traverse}(T)$ invokes visit at an object, that object is added to traversal history.
Traversals method algorithm

- $\text{Traversal}(T)$ is generic: same method for all classes.
- $\text{Traversal}(T)$ is initially called with the start set $T_s$ computed by algorithm 1.
Traversals methods algorithm

- \textit{Traverse}(T), guided by traversal graph TG.
  - 1. define a set of traversal graph nodes \( T' \) by \( T' = \{ v \mid \text{Class}(v) = \text{Class}(\text{this}) \text{ and there exists } u \in T \text{ such that } u = v \text{ or } (u, \hat{\prec}, v) \text{ is an edge in } TG \} \).
  - 2. If \( T' \) is empty, return.
  - 3. Call \texttt{this.visit}().
Traversal methods algorithm

– 4. Let $Q$ be the set of labels which appear both on edges outgoing from a node in $T' \in TG$ and on edges outgoing from this in the object graph. For each field name $l \in Q$, let

$$T_l = \{ v | (u,l,v) \in TG \text{ for some } u \in T' \}.$$

– 5. Call $this.l.Traverse(T_l)$ for all $l \in Q$, ordered by “<“, the field ordering.
Short-cut

strategy:
{A -> B
 B -> C}

Object graph

A(
  <x> X(
    <b> B(
      <x> X(
        <c> C())))
    <c> C()))

Traversal graph

start set

C

finish set
Short-cut

Object graph

A(  
<x> X(  
<b> B(  
<x> X(  
<c> C()()  
<c> C()()  

Used for token set and currently active object

Traversal graph

strategy:  
{A -> B  
B -> C}
Short-cut

Object graph

A(
  <x> X( ⬤ ⬤
  <b> B(
    <x> X(
      <c> C()()
    ))
  ))

Used for token set and currently active object

Traversal graph

strategy:
{A -> B
  B -> C}

Start set

A

B

X

0..1

X

B

B

X

C

Finish set

C

AOO/Demeter

10/28/98
Short-cut

Strategy:
\{ A \rightarrow B \\ B \rightarrow C \}

Object graph

A(
  \langle x \rangle \ X( \\
    \langle b \rangle \ B( \circ ) \\
    \langle x \rangle \ X( \\
      \langle c \rangle \ C()) ) \\
  \langle c \rangle \ C())

Used for token set and currently active object
Short-cut

Object graph:

A( <x> X( <b> B( <x> X( <c> C()) ) ) )

Traversal graph:

strategy:
{A -> B
B -> C}

Used for token set and currently active object
Short-cut

strategy:
{A -> B
B -> C}

Object graph

A(
<x> X(
<b> B(
<x> X(
<c> C()))))
<c> C()))

Traversal graph

A start set

X
b
0..1
b
X
B
C

Used for token set and currently active object
Short-cut

Object graph

A(
  <x> X(ző)
  <b> B(
    <x> X(lovakus)
    <c> C(lovakus))
  <c> C(lovakus))

Used for token set and currently active object

After going back to X

Strategy:
{A → B
  B → C}
Traversal algorithm property

• Let $O$ be an object tree and let $o$ be an object in $O$. Suppose that the $\text{Traverse}$ methods are guided by a traversal graph $TG$ with finish set $T_f$. Let $H(o,T)$ be the sequence of objects which invoke visit while $o.\text{Traverse}(T)$ is active, where $T$ is a set of nodes in $TG$. Then traversing $O$ from $o$ guided by $X(P_{TG}(T,T_f))$ produces $H(o,T)$. 
Example

• Using multiple tokens.
• Reuse zig-zag example.
Zig-zags

strategy graph
with name map

traversal graph = strategy graph (essentially)
strategy graph with name map

traversal graph = strategy graph (essentially)

<A C D E G> is excluded

A OO/Demeter 129
Zig-zags

strategy graph
with name map

traversal graph = strategy graph
(essentially)

object tree

class graph
strategy graph with name map

Zig-zags

class graph

\[ \langle A, C, D, E, G \rangle \text{ is excluded} \]

traversal graph = strategy graph (essentially)
strategy graph
with name map

Zig-zags

A

traversal graph = strategy graph
(essentially)

<A C D E G> is excluded

class graph

A

object tree

A

B

D

E

G

F

C

D

E

G()

F

G()}

G

D

E

F

G()}}

10/28/98

AOO/Demeter
strategy graph with name map

\[ \text{A(} \text{B(} \text{D(} \text{E(} \text{G()) G()} \text{F()) F()) C()) C()) A()) } \]

traversal graph = strategy graph (essentially)

\(<\text{A C D E G}> \text{ is excluded}\)

Zig-zags

class graph
strategy graph with name map

Zig-zags

traversal graph = strategy graph (essentially)

\(<A \ C \ D \ E \ G> \text{ is excluded}\)
Zig-zags

strategy graph with name map

 traversal graph = strategy graph (essentially)

class graph

< A C D E G > is excluded
Zig-zags

strategy graph
with name map

A B D E
C D F

G

< A C D E G > is excluded

traversal graph = strategy graph
(essentially)

A( object tree
B( D( E( G())
C( D( E( G()))
D( E( G())

class graph

A

B

C

D

E

F

G
strategy graph with name map

traversal graph = strategy graph (essentially)

Zig-zags

class graph

object tree

10/28/98
Main Theorem

• Let $SS$ be a strategy, let $G$ be a class graph, let $N$ be a name map, and let $B$ be a constraint map. Let $TG$ be the traversal graph generated by Algorithm 1, and let $T_s$ and $T_f$ be the start and finish sets, respectively.
Main Theorem (cont.)

- Let $O$ be an object tree and let $o$ be an object in $O$. Let $H$ be the sequence of nodes visited when $o.Traverse$ is called with argument $T_s$, guided by $TG$. Then traversing $O$ from $o$ guided by $PathSet[SS,G,N,B]$ produces $H$. 
Complexity of algorithm

- **Algorithm 1**: All steps run in time linear in the size of their input and output. Size of traversal graph: $O(|S|^2 |G| d_0)$ where $d_0$ is the maximal number of edges outgoing from a node in the class graph.

- **Algorithm 2**: How many tokens? Size of argument $T$ is bounded by the number of edges in strategy graph.
Simplifications of algorithm

• If no short-cuts and zig-zags, can use propagation graph. No need for traversal graph. Faster traversal at run-time.
• Presence of short-cuts and zig-zags can be checked efficiently (compositional consistency).
• See chapter 15 of AP book.
Extensions

- Multiple sources
- Multiple targets
- Intersection of traversals
Summary

• Abstract model behind strategy graphs.
• How to implement strategy graphs.
• How to apply: Precise meaning of strategies; how to write traversals manually (watch for short-cuts and zig-zags).
Where to get more information

• Paper with Boaz-Patt Shamir (strategies.ps in my FTP directory)
• Implementation of Demeter/Java shows you how algorithms are implemented in Demeter/Java (and Java). See Demeter/Java resources page.
• Chapter 15 of AP book.
Feedback

• Send email to dem@ccs.neu.edu.