Finding Good Compilation Sequences

Keith D. Cooper

Department of Computer Science
Rice University
Houston, Texas

The Team:
Devika Subramanian, Linda Torczon, Tim Harvey
Steve Reeves
Alex Grosul, Todd Waterman, L. Alamagor

Funding:
DARPA (early work), DOE LACSI, NSF ITR
Overview

This talk will examine a hard problem in compiler design

• Choosing optimizations to improve a program
• Problem has been recognized for decades
• So what’s new?
  ♦ We will compute the answers we seek

This talk will require little background in compilation

• I can teach you what you need to know in five minutes
Overview

Rough Outline

• The problem
  ♦ Finding good program-specific sequences of optimizations

• Early work

• Our work on the problem
  • Learning about the search spaces
  • Designing good search techniques

• Applications to other problems in compilation

Experimental results represent ≥ 100,000,000 experiments
Compilers are well understood

Simple Design Questions

• What optimizations should the compiler run?
• In what order should they run?

Classic answer (1957 to 1997)

• Compiler writer chooses them at design time
The First Complex Compiler

• Uses the “modern” structure (Front end, optimizer, back end)
• Series of filter-style passes
• Passes execute in fixed order, for all programs
Modern Open Source Compiler

- 3 front ends, 1 back end, five-level IR
- Fixed set of passes (> 20)
- Fixed orders (-g, -o1, -o2, ...)

Finding Good Compilation Sequences, February 2005 (Pitt)
Fixed-sequence compilers leave 20% or more on the table
- Achieved performance is fraction of promised performance
- Window where compiler does well is quite narrow

Research community’s response
- Blame the architects
- Consider new paradigms, such as dynamic reoptimization
- Rethink the structure of our compilers
What Is So Hard About This Problem?

Overlap among effects of the various optimizations

- Several ways to eliminate most inefficiencies
- Algorithms to implement an optimization differ in scope & coverage
Choosing Optimizations

Consider redundancy elimination

• **Dominator Value Numbering (DVNT)**
  - Extends local value numbering to dominator regions

• **Alpern-Wegman-Zadeck’s partitioning method (AWZ)**
  - Uses Hopcroft’s DFA minimizer to prove equivalence
  - Second pass performs replacement

• **GCSE using Available Expressions**
  - Global data-flow analysis to find redundant expressions
  - Second pass performs replacement

• **Lazy Code Motion (LCM)**
  - Extends GCSE to handle partial redundancy
  - Computes careful placement for inserted expressions
Finding “Good” Sequences

Two major approaches

• Analytical (Pitt)
  ♦ Model the interactions between optimizations & the impact of applying optimizations to specific code
  ♦ Predict good sequences
  ♦ Capturing & modeling the interactions is hard
  Have made major progress in modeling

• Computational (Rice, U Va & FSU, Princeton)
  ♦ Use feedback-driven empirical search
  ♦ Measure the impact of the sequences
  ♦ Costs are high & engineering is hard
  Ride Moore’s law
  10x, 100x, 1000x
Computing Sequences

The idea is simple, but the implementation is difficult

• How can we find good sequences?

We knew almost nothing about the search spaces (ca. 1998)

• Assumed the problem was hard & space was fractured
• In absence of knowledge, a genetic algorithm is attractive

Problem fits nicely into a genetic algorithm framework

• Assign a letter to each optimization
• Optimization sequence becomes a string of letters
• We were able to compute good sequences!
Steering algorithm tries to minimize an objective function

- Chooses a sequence of optimizations from the pool
- Compiles, evaluates, & provides feedback for next choice
- Several steering algorithms, several objective functions
**Schielke’s Early Experiments**

- **Steering was genetic algorithm** (12 passes from pool of 10)
  - 20 sequences per generation, 1000 generations
  - Replace worst + 3 bad ones at each generation
  - Generate replacement strings with crossover
  - Apply mutation to rest, exempt the best

- **Optimizing for space then speed**
  - 13% smaller code than fixed sequence
  - Code was often faster

- **Optimizing for speed then space**
  - 20% faster code than fixed sequence
  - Code was generally smaller

- Typically found “best” sequence in 200 to 300 generations

This GA turns out to be fairly poor. Even so, it took many fewer probes to find “best” sequence than did random sampling. We need to do better.

6,000 evaluations
How Can We Do Better?

Work has two major thrusts

• Characterizing the search spaces
  ♦ Large-scale enumerations of small spaces to develop insights
  ♦ Small-scale experiments in large spaces to confirm insights

• Designing effective search algorithms
  ♦ Rapid offline experiments in enumerated spaces
  ♦ Confirming online experiments in large spaces

• Question: can we understand the space analytically?
  ♦ Models of optimizations & their combinations (Pitt Group)
  ♦ I don’t yet know enough about interactions & effects

Is it convex or differentiable?
Current Prototype Adaptive Compiler

- 16 transformations
  - Run in any order
    ⇒ Exposes subtle bugs
  - Many options & variants

- Backends for PowerPC, Sparc & instrumented, simulated RISC

- Search-based steering algorithms
  - Hill climber (valley descender?)
  - Greedy construction
  - Genetic algorithm + variations

- Objective functions
  - Speed, space, bit transitions

Experimental tool

- Exploring applications
- Learning about search space
- Designing better searches
Characterizing the Search Space

Enumeration Experiments

- Full search space is huge: $1,099,511,627,776$ points in $16^{10}$
- Work with tractable subspaces: $5^{10}$ has $9,765,625$ points
  - Work with small programs, of necessity
  - Enumerate full $5^{10}$ subspaces & analyze data offline
  - First enumeration, FMIN, took 14 CPU-months
  - Today, takes about 2.5 CPU-months
  - We’ve done 8 full enumerations in $5^{10}$
    - ≥ 60,000,000 compilations & evaluations

- Use large-scale studies to gain insight into the space
  - Rely on randomization & restart to explore

Farm of Apple XServes
and a couple of Suns

Finding Good Compilation Sequences, February 2005 (Pitt)
What Have We Learned About Search Spaces?

We confirmed some obvious points

These spaces are:
- not smooth, convex, or differentiable
- littered with local minima at different fitness values
- program dependent

p: peeling
l: PRE
o: logical peephole
s: reg. coalescing
n: useless CF elimination

adpcm-coder, 5^4 space, plosn

Characterizing the Spaces
What Have We Learned About Search Spaces?

We confirmed some obvious points

These spaces are:

• not smooth, convex, or differentiable
• littered with local minima at different fitness values
• program dependent

p: peeling
l: PRE
o: logical peephole
s: reg. coalescing
n: useless CF elimination
What Have We Learned About Search Spaces?

Both programs and optimizations shape the space.

Characterizing the Spaces

\[ fmin \ \& \ \text{zeroin} \rightarrow \text{plosn} \]

- \( p \): peeling
- \( l \): PRE
- \( o \): logical peephole
- \( s \): reg. coalescing
- \( n \): useless CF elimination

\[ \Rightarrow \ \text{Range is 0 to 70\%} \]

\[ \Rightarrow \ \text{Can approximate distribution} \]

\[ \text{with 1,000 probes} \]
What Have We Learned About Search Spaces?

Both programs and optimizations shape the space

- fmin & zeroin — pdnxt
- p: peeling
- d: dead code elimination
- n: useless CF elimination
- x: dominator value num’g
- t: strength reduction

⇒ Range is compressed (0-40%)
⇒ Best is 20% worse than best in “plosn”
What Have We Learned About Search Spaces?

Many local minima are “good”

Many local minima
258 strict
27,315 non-strict

Lots of chances for a search to get stuck in a local minima
What Have We Learned About Search Spaces?

Distance to a local minimum is small

Downhill walk halts quickly

Best-of-\(k\) walks should find a good minimum, for big enough \(k\)
Search Algorithms

- Knowledge does not make the code run faster
- Must use that knowledge to build better search techniques
  ⇒ Genetic algorithms, hillclimbers, greedy algorithms

For each search technique, we must answer two questions:
- Does it find good sequences?
- How quickly does it find those sequences?

Answering these has been the bulk of our work on this problem for the last two years

Moving from curiosity-driven research to practical techniques
Search Algorithms: Genetic Algorithms

- Original work used a genetic algorithm
- Experimented with many variations on GA
- Current favorite is “GA-50x50”
  - Population of 50 sequences
  - 50 evolutionary steps
- At each step
  - Best 10% survive
  - Remaining 90% of new generation created by crossover
    - Fitness-weighted reproductive selection
    - Single-point, random crossover
  - Hash test for redundancy with mutate until unique for hits

Difference between 50 steps & 100 steps is typically < 0.1%
Search Algorithms: Genetic Algorithms

Progress over time

adpcm-decoder
50 sequences
Best 10 survive
40 from crossover
— fitness-weighted selection
— 2% mutation rate
— mutate until unique
2,000 evaluations
Search Algorithms: Genetic Algorithms

Progress over time

GA runs for tomcatv
GA-50x100, 20% elitism

tomcatv
same GA
4,000 probes
Search Algorithms: Hill climbers

Many nearby local minima suggests descent algorithm

- Neighbor $\Rightarrow$ Hamming-1 string (differs in 1 position)
- Evaluate neighbors and move downhill
- Repeat from multiple starting points

- Steepest descent $\Rightarrow$ take best neighbor
- Random descent $\Rightarrow$ take 1st downhill neighbor (random order)
- Impatient descent $\Rightarrow$ random descent, limited local search
  - HC algorithms examine at most 10% of neighbors
  - HC-10 uses 10 random starting points, HC-50 uses 50
HC-10: Why 4 probes are enough?

Patience vs. portion of HC descents finishing within 5% of the best value for \( f_{\text{min+plosn}} \)

At 10% patience, 40% of runs find a “good” sequence

Random restart is a better way to spend your time
HC-10 Behavior: Patient Version

Patient HC-10 performance for fmin+plosn

Distance from the best value

Total # of evaluations

start points  end points
HC-10 Behavior: Impatient Version

Impatient HC-10 performance for fmin+plosn

- Distance from the best value:
  - Start points
  - End points

Graph showing the performance of HC-10 with distance from the best value and total number of evaluations.
Search Algorithms: Greedy Constructive

Greedy algorithms work well on many complex problems

How do we do a greedy search?

1. start with empty string
2. pick best optimization as 1st element
3. for i = 2 to k
   try each pass as prefix and as suffix
   keep the best result

Algorithm takes $k \cdot (2n - 1)$ evaluations for a string of length $k$

Takes locally optimal steps

Early exit for strings with no improvement

Local minimum under a different notion of neighbor

95 evaluations for 10-of-5 space
### Search Algorithms: Greedy Constructive

Successive evaluations refine the string

<table>
<thead>
<tr>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sp</td>
<td>snp</td>
</tr>
<tr>
<td></td>
<td>ps</td>
<td>psn</td>
</tr>
<tr>
<td>p</td>
<td>sl</td>
<td>snl</td>
</tr>
<tr>
<td>l</td>
<td>ls</td>
<td>lsn</td>
</tr>
<tr>
<td>o</td>
<td>s</td>
<td>so</td>
</tr>
<tr>
<td>s</td>
<td>os</td>
<td>sno</td>
</tr>
<tr>
<td>n</td>
<td>winner</td>
<td>no</td>
</tr>
<tr>
<td>s</td>
<td>winner</td>
<td>sn</td>
</tr>
<tr>
<td>n</td>
<td>winner</td>
<td>snn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nsn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Search Algorithms: Greedy Constructive

Unfortunately, ties (equal-valued choices) pose a major problem

- Ties can take GC to wildly different places
- Have experimented with three GC algorithms
  - GC-exh explores pursues all equal-valued options
  - GC-bre does a breadth-first rather than depth-first search
  - GC-10 & GC-50 break ties randomly and use restart to explore

<table>
<thead>
<tr>
<th></th>
<th>adpcm-d</th>
<th>GC-exh</th>
<th>GC-bre</th>
<th>GC-50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequences checked</td>
<td>91,633</td>
<td>325</td>
<td>2,200</td>
<td></td>
</tr>
<tr>
<td>Code speed</td>
<td>1.0</td>
<td>+ 0.003%</td>
<td>+ 2%</td>
<td></td>
</tr>
</tbody>
</table>

Experiments use GC-10 & GC-50
Search Results vs “rvzcodtvzcod” ("standard sequence")

Instructions executed
Simulated RISC execution

% of "standard"

Searches run to completion

Finding Good Compilation Sequences, February 2005 (Pitt)
Why “instructions executed”?

1. experiments done over several years
2. correlates loosely to speed & power
3. have experiments with wall time, code space, bit transitions, …
4. “close enough” for exploring the search spaces

Searches run to completion
Search Results vs “rvzcodtvzcod” ("standard sequence")

Instructions executed
Simulated RISC execution

% of "standard"

0%  20%  40%  60%  80%  100%  120%  140%

fmin  zeron  adpcm-c  adpcm-d  g721-e  g721-d  fppp  nsieve  tomcatv  svd

Searches run to completion

GC-rand 10  GA 50×100  HC 10
Search Results vs “rvzcodtvzcod” ("standard sequence")

Instructions executed
Simulated RISC execution

% of "standard"

<table>
<thead>
<tr>
<th></th>
<th>GC-rand 10</th>
<th>GA 50x100</th>
<th>HC 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>zeroin</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>adpcm-c</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>adpcm-d</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>g721-e</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>g721-d</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>fpppp</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>nsieve</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>svd</td>
<td>120%</td>
<td>120%</td>
<td>140%</td>
</tr>
</tbody>
</table>

Searches run to completion

GC-10 has problems with fmin & tomcatv
Your mileage may vary: fmin

Finding Good Compilation Sequences, February 2005 (Pitt)
GA 50x50 does almost as well as GA 50x100.
Search Effectiveness

Solution Quality at 250 Trials

% of Batch Sequence

Programs

Finding Good Compilation Sequences, February 2005 (Pitt)
Search Effectiveness

Solution Quality at 1,000 Trials

% of Batch Sequence

Finding Good Compilation Sequences, February 2005 (Pitt)
Search Algorithms

So, where are we?

• Reliably find good sequences in 100 to 600 evaluations
  ♦ Good means “competitive with GA-50 for 100 generations”
  ♦ Old standard was 6,000 to 20,000 evaluations

• How close are we to global minimum?
  ♦ Cannot know without enumeration
  ♦ Enumeration is hard to justify in \(16^{10}\) (& harder to perform)
  ♦ Next big experiment: HC-100,000 on several codes
    (use 264-processor IA-64 machine)

• Next major step is bringing code features into the picture ...
  ♦ Students now working on this problem
Adaptation in Other Contexts

Complex optimizations may need adaptive strategies

- Blocksize selection has fairly simple search spaces

ATLAS finds a good point in an 8 dimensional parameter space, so it looks at much more detail than this experiment considered...

Using the MIPSPro heuristic

Blocksize given on command line

DGEMM from ATLAS Library

Compiler parameters do not allow adequate control

One blocksize fits all loops & all arrays

DGEMM, 1500x1500

LACSI 2004 paper
Adaptation in Other Contexts

Complex optimizations may need adaptive strategies

- Inline substitution
  - Find the right set of call sites to inline
  - Program-specific strategies beat one-size fits all
  - Prototype tool uses CNF expressions & adaptive controller
    - Experiment in parameter-set design as well as optimization
Adaptative Choice in Inline Substitution

Characterizing the search spaces with 2d parameter sweeps

Running times for inlined variants of Vortex — as a function of inliner parameters

Space looks much easier to search than the sequence finding spaces

Finding Good Compilation Sequences, February 2005 (Pitt)
Adaptative Choice in Inline Substitution

Characterizing the search spaces with 2d parameter sweeps

Running times for inlined variants of bzip — as a function of inliner parameters

Space looks much easier to search than the sequence finding spaces

SC is inlined statement count

Finding Good Compilation Sequences, February 2005 (Pitt)
Adaptation in Other Contexts

Complex optimizations may need adaptive strategies

• Algebraic reassociation
  ♦ Reorder expressions to improve optimization results
    ⇒ Associative, distributive, & commutative laws
  ♦ Right strategy depends on optimization context
  ♦ Adaptive choice of detailed strategy

• We have begun a project to look at adaptive reassociation
  ♦ Driving reassociation strategy by program-specific needs of other optimizations
  ♦ Two goals: reduce operation count & change instruction mix
Adaptation in Other Contexts

Command-line parameter options can change code quality

• Several folks have looked at adaptive selection of program-specific command-line parameter sets
  ♦ Good work by Granston & Holler, Chow & Wu, Knijnenburg et al
  ♦ All show publishable improvements

• Current compilers are not built to facilitate adaptive control
  ♦ Parameters are poorly selected (-o1, -o2, -o3, ...)
  ♦ Hard to specify locality, names, and distinct treatment
Designing Practical Adaptive Compilers

User may not want to pay 300x for compilation
⇒ Moore’s law will help ...

Engineering approaches

• Make the search incremental across many compilations
  ♦ Or incremental across a large time-step calculation

• Develop faster techniques to evaluate sequences on codes

• Use parallelism

• And, of course, make the underlying compiler passes fast

We are pursuing all of these approaches
Speeding up Evaluation

Compile-evaluate cycle takes most of the time

- Avoid duplicate evaluations (Schielke, VISTA folks)
- Faster evaluation methods
  - Low-level performance models (Mellor-Crummey et al.)
  - Analytical models (Soffa et al.)
  - Synthetic execution (Harvey, Reeves, Grosul)
- Machine-learning to predict sequences
  - Want to relate sequences to source-code features
  - Preliminary attempts have been disappointing
  - Not clear what source-code features matter

Success in any or all of these approaches would reduce the cost of evaluating each sequence
Where Does This Research Lead?

Practical systems within ten years

How will they work? (Frankly, we don’t yet know)

• Efficient searches that capitalize on properties of the space
• Incremental searches distributed over program development
• Predictive techniques that use program properties to choose good starting points
• Compiler structures & parameterizations that fit adaptation

In the meantime, we have a lot of work to do

And machines will keep getting faster ...
And that’s the end of my story ...
But, All These Experiments Used Our Compiler

We have tried a number of other compilers, with no success

- Try to reorder a pass in GCC
- Hit problems in GCC, SUIF-1, ORC, ...
- Look forward to using LLVM & Phoenix

Our platform is reconfigurable by accident of design

- Have run > 100,000,000 configurations in our system
- One unavoidable phase-order bug

We have used MIPSPro in another series of experiments
Characterizing FMIN

• One application, FMIN
  ♦ 150 lines of Fortran, 44 basic blocks
  ♦ Exhibited complex behavior in other experiments

• Ran hill-climber from 100 random starting points
  ♦ Picked the 5 most used passes from winners

• Evaluated all strings of length 10 from those 5
  ♦ 9,765,625 distinct combinations (for speed)
  ♦ 34,000 to 37,000 experiments/machine/day
  ♦ Produces sequence & number of cycles to execute
  ♦ First subspace took 14 CPU-months to complete

• We’re learning from the results
  ♦ Can run experiments against the data offline

Have now done several subspaces, at 250,000 experiments/machine/day
Results from FMIN

In the initial subspace - plosn (9,765,625 sequences)

- Best sequence is 986 cycles
- Seven at 987
- Worst sequence is 1716
- Unoptimized code takes 1765
- Range is 3% to 43% improvement

Local minima

- 258 strict local minima \((x < \text{all Hamming-1 points})\)
- 27,315 non-strict local minima \((x \leq \text{all Hamming-1 points})\)
- All local minima are improvements
- They come in many shapes

With full set of transformations:

- GA: 822 in 184 generations of 100
  825 in 152 generations of 100
- Hill: 833 at round 2232
  830 at round 750
Cross Training Results

Effect of best strings found on benchmarks $f_{min}$ and $zeroin$
Cross Training Results

Effect of best strings found on benchmarks *adpcm-coder* and *ft*
Training-Testing Results

Operations executed relative to rvzcodtvzcod
Simulated RISC machine

Take best string & run on another data set ...

Training/testing data shows small variation
- no systematic bias from training data

Operations executed relative to rvzcodtvzcod
Simulated RISC machine

Finding Good Compilation Sequences, February 2005 (Pitt)
### Results from Exhaustive Enumeration

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>#</th>
<th>Worst</th>
<th>Distance from best</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmin+plosn</td>
<td>1,002</td>
<td>1</td>
<td>1,716</td>
<td>71%</td>
<td>1</td>
</tr>
<tr>
<td>fmin+pdxnt</td>
<td>1,216</td>
<td>8</td>
<td>1,716</td>
<td>41%</td>
<td>1,024</td>
</tr>
<tr>
<td>zeroin+plosn</td>
<td>832</td>
<td>3,099</td>
<td>1,446</td>
<td>74%</td>
<td>1</td>
</tr>
<tr>
<td>zeroin+pdxnt</td>
<td>1,020</td>
<td>8</td>
<td>1,446</td>
<td>42%</td>
<td>1,024</td>
</tr>
<tr>
<td>adpcm-c+plosn</td>
<td>9,341,852</td>
<td>291</td>
<td>22,929,174</td>
<td>145%</td>
<td>1</td>
</tr>
<tr>
<td>svd+plosn</td>
<td>7,340</td>
<td>44,515</td>
<td>14,283</td>
<td>94%</td>
<td>1</td>
</tr>
</tbody>
</table>
Diversity of Sequences

Best sequences found (to date) in $16^{10}$ space

<table>
<thead>
<tr>
<th>fmin</th>
<th>pppxocdldsn</th>
</tr>
</thead>
<tbody>
<tr>
<td>zeroin</td>
<td>oplvscdzsn</td>
</tr>
<tr>
<td>adpcm-c</td>
<td>prppocvdsn</td>
</tr>
<tr>
<td>adpcm-d</td>
<td>prppopcdlsn</td>
</tr>
<tr>
<td>g721-e</td>
<td>pnpppcdzsn</td>
</tr>
<tr>
<td>g721-d</td>
<td>vpppppcdpsn</td>
</tr>
<tr>
<td>fpppp</td>
<td>nopclvdspn</td>
</tr>
<tr>
<td>nsieve</td>
<td>ppppppdczsn</td>
</tr>
<tr>
<td>tomcatv</td>
<td>crxpotvdzsn</td>
</tr>
<tr>
<td>svd</td>
<td>cztpvodvsn</td>
</tr>
</tbody>
</table>
What About Presentation Order?

Clearly, order might affect the picture...

Still, some bad local minima

Reality

Fiction
Road Map for our Project

Our goals

• Short term (now)
  ♦ Characterize the problems, the potential, & the search space
  ♦ Learn to find good sequences quickly (search)

• Medium term (3 to 5 years)
  ♦ Develop proxies and estimators for performance (speed)
  ♦ Demonstrate practical applications for adaptive scalar optimization
  ♦ Understanding interface between adaptive controller & compiler

• Long term (5 to 10 years)
  ♦ Apply these techniques to harder problems
    • Data distribution, parallelization schemes on real codes
    • Compiling for complex environments, like the Grid
  ♦ Develop a set of design & engineering principles for adaptive compilers
Experience from Early Experiments

Improving the Genetic Algorithm

• Experiments aimed at understanding & improving convergence
• Larger population helps
  ♦ Tried pools from 50 to 1000, 100 to 300 is about right
• Use weighted selection in reproductive choice
  ♦ Fitness scaling to exaggerate late, small improvements
• Crossover
  ♦ Tried single point and two point random crossover
  ♦ Apply “mutate until unique” to each new string
• Variable length chromosomes
  ♦ With fixed string, GA discovers NOPs
  ♦ Varying string rarely has useless pass

GA now dominates both the hill climber and random probing of the search space, in results/work.
Effectiveness in the GA

Work versus solution quality

Sometimes, more effort pays off