Basic Performance Engineering

Saman Amarasinghe

Fall 2010
Basic Performance Engineering

- **Maximum use of the compiler/processor/system**
- **Modifying data structures**
- **Modifying code structures**
- **Using the right algorithm**

### Matrix Multiply Example

- **Today**

- **Most Bithacks**

© Saman Amarasinghe 2009
Bentley’s Rules

There is no “theory” of performance programming

Performance Programming is:

- Knowledge of all the layers involved
- Experience in knowing when and how performance can be a problem
- Skill in detecting and zooming in on the problems
- A good dose of common sense

A set of rules

- Patterns that occur regularly
- Mistakes many make
- Possibility of substantial performance impact
- Similar to “Design Patterns” you learned in 6.005

© Saman Amarasinghe 2009
Bentley’s Rules

A. Modifying Data
B. Modifying Code
Bentley’s Rules

A. Modifying Data
   1. Space for Time
   2. Time for Space
   3. Space and Time

B. Modifying Code
Bentley’s Rules

A. Modifying Data
   1. Space for Time
      a. Data Structure Augmentation
      b. Storing Precomputed Results
      c. Caching
      d. Lazy Evaluation
   2. Time for Space
   3. Space and Time

B. Modifying Code
Data Structure Augmentation

Add some more info to the data structures to make common operations quicker

When is this viable?

- Additional information offers a clear benefit
- Calculating the information is cheap/easy
- Keeping the information current is not too difficult

Examples?

- Faster Navigation
  - Doubly linked list and Delete Operation
- Reduced Computation
  - Reference Counting
Storing Precomputed Results

Store the results of a previous calculation. Reuse the precomputed results than redoing the calculation.

When is this viable?

- Function is expensive
- Function is heavily used
- Argument space is small
- Results only depend on the arguments
- Function has no side effects
- Function is deterministic

Examples:
Precomputing Template Code

result precompute[MAXARG];

result func_initialize(int arg)
{
    for(i=0; i < MAXARG; i++)
    {
        precompute[arg] = func(arg);
    }
}

result func_apply(int arg)
{
    return precompute[arg];
}
Pascal’s Triangle

```c
int pascal(int y, int x)
{
    if (x == 0) return 1;
    if (x == y) return 1;
    return pascal(y-1, x-1) + pascal(y-1, x);
}
```

```c
int pt[MAXPT][MAXPT];

main()
{
    ...
    for(i=0; i < PTMAX; i++) {
        pt[i][0] = 1;
        pt[i][i] = 1;
        for(j=1; j < i; j++)
            pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
    }

    int pascal(int y, int x) {
        return pt[y][x];
    }
}
```

© Saman Amarasinghe 2009
Another example of precomputing

```c
unsigned long fib(int n)
{
    if(n==1) return 1;
    if(n==2) return 1;
    return fib(n-1) + fib(n-2);
}
```

```c
int i;
unsigned long prev, curr, tmp;
if(n==1) return 1;
if(n==2) return 1;
prev = 1;
curr = 1;
for(i=3; i <=n; i++) {
    tmp = prev + curr;
    prev = curr;
    curr = tmp;
}
return curr;
```
Caching

Store some of the heavily used/recently used results so they don’t need to be computed

When is this viable?

- Function is expensive
- Function is heavily used
- Argument space is large
- There is temporal locality in accessing the arguments
- A single hash value can be calculated from the arguments
- There exists a “good” hash function
- Results only depend on the arguments
- Function has no side effects
- Coherence
  - Is required:
    - Ability to invalidate the cache when the results change
      - Function is deterministic
  - Or stale data can be tolerated for a little while
Caching Template Code

typedef struct cacheval {
    argtype1 arg1;
    ...
    argtypen argn;
    resulttype result;
}
struct cacheval cache[MAXHASH];

resulttype func_driver(argtype1 a1, ..., argtypen an) {
    resulttype res;
    int bucket;
    bucket = get_hash(a1, a2, ..., an);
    if((cache[bucket].arg1 == a1) && ...(cache[bucket].argn == an))
        return cache[bucket].result;
    res = func(a1, ..., an);
    cache[bucket].arg1 = a1;
    ...
    cache[bucket].argn = an;
    cache[bucket].result = res;
    return res;
}
Lazy Evaluation

Differ the computation until the results are really needed

When is this viable?
- Only a few results of a large computation is ever used
- Accessing the result can be done by a function call
- The result values can be calculated incrementally
- All the data needed to calculate the results will remain unchanged or can be packaged-up
Lazy Template Code

```c
resulttype precompute[MAXARG];

resulttype func_apply(int arg)
{
    resulttype res;
    if(precompute[arg] != EMPTY)
        return precompute[arg];

    res = func(arg);
    precompute[arg] = res;
    return res;
}
```

© Saman Amarasinghe 2009
int pascal(int y, int x)
{
    if(x == 0) return 1;
    if(x == y) return 1;
    return pascal(y-1, x-1) + pascal(y-1, x);
}

Precomputation

int pt[MAXPT][MAXPT];

main()
{
    ...
    for(i=0; i < PTMAX; i++) {
        pt[i][0] = 1;
        pt[i][i] = 1;
        for(j=1; j < i; j++)
            pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
    }

    int pascal(int y, int x) {
        return pt[y][x];
    }
}

Lazy Evaluation

int pt[MAXPT][MAXPT];

int pascal(int y, int x)
{
    if(x == 0) return 1;
    if(x == y) return 1;
    if(pt[y][x] > 0) return pt[y][x];
    val = pascal(y-1, x-1) + pascal(y-1, x);
    pt[y][x] = val;
    return val;
}
Bentley’s Rules

A. Modifying Data
   1. Space for Time
   2. Time for Space
      1. Packing/Compression
      2. Interpreters
   3. Space and Time

B. Modifying Code
Packing/Compression

Reduce the space of the data by storing them as “processed” which will require additional computation to get the data.

When is it viable?

- Storage is at a premium
  - Old days → most of the time!
  - Now
    - Embedded devices with very little memory/storage
    - Very large data sets
- Ability to drastically reduce the data size in storage
- Extraction process is amenable to the required access pattern
  - Batch – expand it all
  - Steam
  - Random access

© Saman Amarasinghe 2009
Packing / Compression

**Packing Level**
- Packing in memory
- Packing out of core storage

**Packing Methods**
- Use smaller data size
- Eliminate leading zeros
- Eliminate repetitions (LZ77)
- Heavyweight compression

© Saman Amarasinghe 2009
LZ77 Basics

<1, 3> O <2, 4>
LZ77 Basics

input stream → decompress → output stream

<1, 3> O <2, 4> L A
LZ77 Basics

<1, 3> O

<2, 4>

input stream ➔ decompress ➔ output stream
LZ77 Basics

input stream → decompress → output stream

<1, 3> O L A L A L A

<2, 4>
LZ77 Basics

input stream → decompress → output stream

<1, 3>  O L A L A L A L A
LZ77 Basics

input stream  ➔ decompress  ➔ output stream

<1, 3>  O L A L A L A L A
LZ77 Basics

input stream → decompress → output stream

<1, 3>

O O O O O L A L A L A L A
Interpreters

Instead of writing a program to do a computation, use a language to describe the computation at a high level and write an interpreter for that language

Benefits:

- Nice and clean abstraction of the language
- Easy to add/change operations by changing the HLL program
- Much more compact representation

Examples:

- String processing
- Bytecodes

© Saman Amarasinghe 2009
Bentley’s Rules

A. Modifying Data
   1. Space for Time
   2. Time for Space
   3. Space and Time
      1. SIMD

B. Modifying Code
SIMD

Store short width data packed into the machine word (our Intel machines are 64 bit)

- 64 Booleans (unsigned long long)
- 2 32-bit floats
- 2 32-bit integers
- 4 16-bit integers
- 8 8-bit integers

Intel SSE instructions operate on these multi granularities

Single operation on all the data items
Win-win situation both faster and less storage

When viable?

- If the same operation is performed on all the data items
- Items can be stored contiguous in memory
- Common case don’t have to pick or operate on each item separately

© Saman Amarasinghe 2009
Example: overlap in battleship boards

```c
#define BSZ 64

int overlap(int board1[BSZ][BSZ],
            int board2[BSZ][BSZ])
{
    int i, j;
    for(i=0; i<BSZ; i++)
        for(j=0; j<BSZ; j++)
            if((board1[i][j] == 1) &&
               (board2[i][j] == 1))
                return 1;
    return 0;
}
```

```c
#define BSZ 64

int overlap(
    unit64_t board1[BSZ],
    unit64_t board2[BSZ])
{
    int i;
    for(i=0; i<BSZ; i++)
        if((board1[i] & board2[i]) != 0)
            return 1;
    return 0;
}
```

© Saman Amarasinghe 2009
Bentley’s Rules

A. Modifying Data

B. Modifying Code
   1. Loop Rules
   2. Logic Rules
   3. Procedure Rules
   4. Expression Rules
   5. Parallelism Rules
Bentley’s Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
   a. Loop Invariant Code Motion
   b. Sentinel Loop Exit Test
   c. Loop Elimination by Unrolling
   d. Partial Loop Unrolling
   e. Loop fusion
   f. Eliminate wasted iterations

2. Logic Rules

3. Procedure Rules

4. Expression Rules

5. Parallelism Rules

© Saman Amarasinghe 2009
Loops

If each program instruction is only executed once
- 3 GHz machine
- Requires 12 Gbytes of instructions a second! (1 CPI, 32 bit instructions)
- A 100 GB disk full of programs done in 8 seconds

Each program instruction has to run millions of times
→ Loops

90% of the program execution time in 10% of the code
- All in inner loops

© Saman Amarasinghe 2009
Loop Invariant Code Motion

Move as much code as possible out of the loops
Compilers do a good job today
- Analyzable code
- Provably results are the same in every iteration
- Provably no side effects

Viability?
- Loop invariant computation that compiler cannot find
- The cost of keeping the value in a register is amortized by the savings
double factor;
factor = exp(sqrt(PI/2));
for(i=0; i < N; i++)
    X[i] = X[i] * factor;

for(i=0; i < N; i++)
    X[i] = X[i] * exp(sqrt(PI/2));
Sentinel Loop Exit Test

When we iterate over a data to find a value, we have to check the end of the data as well as for the value.

Add an extra data item at the end that matches the test

Viability?

- Early loop exit condition that can be harnessed as the loop test
- When an extra data item can be added at the end
- Data array is modifiable
# Example of Sentinel Loop Exit Test

```c
#define DSZ 1024
datatype array[DSZ];

int find(datatype val) {
    int i;
    for(i=0; i<DSZ; i++)
        if(array[i] == val)
            return i;
    return -1;

    OR
    i = 0;
    while(((i<DSZ) && (array[i] != val))
        i++;
    if(i==DSZ) return -1;
    return i;
}
```

```c
#define DSZ 1024
datatype array[DSZ+1];

int find(datatype val) {
    int i;
    array[DSZ] = val;
    i = 0;
    while(array[i] != val)
        i++;
    if(i == DSZ)
        return -1;
    return i;
}
```
Loop Elimination by Unrolling

Known loop bounds $\rightarrow$ can fully unroll the loop

Viability?

- Small number of iterations (code blow-up is manageable)
- Small loop body (code blow-up is manageable)
- Little work in the loop body (loop test cost is non trivial)

Can get the compiler to do this.

Example:

```c
sum = 0;
for(i=0; i<10; i++)
    sum = sum + A[i];
```

```c
```
Partial Loop Unrolling

Make a few copies of the loop body.

Viability?
- Work in the loop body is minimal (viable impact of running the loop test fewer number of times)
- Or the ability to perform optimizations on combine loop bodies

Can get the compiler to do this

Example:

```c
sum = 0;
for (i=0; i<n; i++)
    sum = sum + A[i];
```

```c
sum = 0;
for (i=0; i<n-3; i += 4)
for (; i<n; i++)
    sum = sum + A[i];
```
Loop Fusion

When multiple loops iterate over the same set of data put the computation in one loop body.

Viability?

➢ No aggregate from one loop is needed in the next
➢ Loop bodies are manageable

Example

```c
amin = INTMAX;
for(i=0; i<n; i++)
    if(A[i] < amin) amin = A[i];

amax = INTMIN;
for(i=0; i<n; i++)
    if(A[i] > amax) amax = A[i];
```

```c
amin = INTMIN;
for(i=0; i<n; i++)
    if(A[i] < amin) amin = A[i];

amax = INTMAX;
```

```c
int atmp = A[i];
if(atmp < amin) amin = atmp;
if(atmp > amax) amax = atmp;
```
Eliminate Wasted Iterations

Change the loop bounds so that it will not iterate over an empty loop body

Viability?
- For a lot of iterations the loop body is empty
- Can change the loop bounds to make the loop tighter
- Or ability to change the data structures around (efficiently and correctly)

Example 1

```c
for(i=0; i<n; i++)
    for(j = i; j < n - i; j++)
        ...
```

```c
for(i=0; i<n/2; i++)
    for(j = i; j < n - i; j++)
        ...
```
Example II

```c
int val[N];
int visited[N];

for(i=0; i < N; i++)
    visited[i] = 0;

for(i=0; i < N; i++) {
    int minloc;
    int minval = MAXINT;
    for(j=0; j < N; j++)
        if(!visited[j])
            if(val[j] < minval) {
                minval = val[j];
                minloc = j;
            }
    visited[minloc] = 1;
    // process val[minloc]
}
```
Bentley’s Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
   a. Exploit Algebraic Identities
   b. Short Circuit Monotone functions
   c. Reordering tests
   d. Precompute Logic Functions
   e. Boolean Variable Elimination
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules
Exploit Algebraic Identities

If the evaluation of a logical expression is costly, replace algebraically equivalent

Examples

- \( \text{sqr}(x) > 0 \) →
- \( \sqrt{x^2 + y^2} < \sqrt{a^2 + b^2} \) →
- \( \ln(A) + \ln(B) \) →
- \( \sin(X)\sin(X) + \cos(X)\cos(X) \) →
Short Circuit Monotone Functions

In checking if a monotonically increasing function is over a threshold, don’t evaluate beyond the threshold

Example:

```
sum = 0;
for(i=0; i < N; i++)
    sum = sum + X[i];
if(sum > cutoff) return 0;
return 1;
```

```
sum = 0;
i = 0;
while((i < N) && (sum < cutoff))
    sum = sum + X[i++];
if(sum > cutoff) return 0;
return 1;
```

```
sum = 0;
i = 0;
X[N] = cutoff;
while (sum < cutoff)
    sum = sum + X[i++];
if(i == N) return 1;
return 0;
```
Reordering tests

Logical tests should be arranged such that inexpensive and often successful tests precede expensive and rarely successful ones.

Add inexpensive and often successful tests before expensive ones.

Example:

```python
if (sqrt(sqr(x1 - x2) +
    sqr(y1 - y2)) < rad1+rad2)
    return COLLITION;
return OK;
```

```python
if (abs(x1-x2) > rad1+rad2)
    return OK;
if (abs(y1-y2) > rad1+rad2)
    return OK;
if (sqrt(sqr(x1 - x2) +
    sqr(y1 - y2)) < rad1+rad2)
    return COLLITION;
return OK;
```
Precompute Logic Functions

A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.

Example:

```c
int palindrome(unsigned char val)
{
    unsigned char l, r;
    int i;
    l = 0x01;
    r = 0x80;
    for(i=0; i < 4; i++) {
        if(((val & l) == 0) ^ ((val & r) == 0)) // initialize array
            return 0;
        l = l << 1;
        r = r >> 1;
    }
    return l;
}

int isPalindrome[256];
int palindrome(unsigned char val)
{
    return isPalindrome[val];
}
```

© Saman Amarasinghe 2009
Boolean Variable Elimination

Replace the assignment to a Boolean variable by replacing it by an IF-THEN-ELSE

Example:

```c
int v;
if(Boolean expression) {
    v = Boolean expression;
    S1;
    S2;
    S4;
    S5;
} else {
    S1;
    S3;
    S4;
    S5;
}
```
Bentley’s Rules

A. Modifying Data

B. Modifying Code
   1. Loop Rules
   2. Logic Rules
   3. Procedure Rules
      a. Collapse Procedure Hierarchies
      b. Coroutines
      c. Tail Recursion Elimination
   4. Expression Rules
   5. Parallelism Rules
Collapse Procedure Hierarchies

Inline small functions into the main body

- Eliminates the call overhead
- Provide further opportunities for compiler optimization

```c
#include <limits.h>

#define max(a, b) (((a) > (b))?(a):(b))

int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}

inline int max(int a, int b) {
    if(a > b) return a;
    return b;
}

int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

© Saman Amarasinghe 2009
Coroutines

Multiple passes over data should be combined
➢ Similar to loop fusion (B.1.f), but at a scale and complexity beyond a single loop

Example pattern

```plaintext
Loop {
    Loop {
        Read from I
        ProcessA
        Write to II
    }
    Loop {
        Read from II
        ProcessB
        Write to III
    }
}
```

```plaintext
Loop {
    Loop {
        Read from I
        ProcessA
        Write to buffer
    }
    Loop {
        Read from buffer
        Process B
        Write to III
    }
}
```
Tail Recursion Elimination

In a self recursive function, if the last action is calling itself, eliminate the recursion.

Example pattern

```c
int fact(int n, int res) {
    if(n == 1) return res;
    return fact(n - 1, res*n);
}
```

```c
int fact(int n, int res) {
    while(1) {
        if(n == 1) return res;
        res = res*n;
        n = n - 1;
    }
}
```
Bentley’s Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
   a. Compile-time Initialization
   b. Common Subexpression Elimination
   c. Pairing Computation
5. Parallelism Rules
Compile-Time Initialization

If a value is a constant, make it a compile-time constant.

- Save the effort of calculation
- Allow value inlining
- More optimization opportunities

Example

```c
#define PI 3.14159265358979
#define R 12

......
vol = 2 * pi() * r * r;
{ ...
    ......
    vol = 2 * PI * R * R;
```

© Saman Amarasinghe 2009
Common Subexpression Elimination

If the same expression is evaluated twice, do it only once

Viability?

- Expression has no side effects
- The expression value does not change between the evaluations
- The cost of keeping a copy is amortized by the complexity of the expression
- Too complicated for the compiler to do it automatically

Example

```c
double tmp;
...
tmp = sin(a);
x = tmp * tmp;
```
Pairing Computation

If two similar functions are called with the same arguments close to each other in many occasions, combine them.

- Reduce call overhead
- Possibility of sharing the computation cost
- More optimization possibilities

Example

```c
typedef struct twodouble {
    double d1;
    double d2;
} twodouble;
...

twodouble dd;
dd = sincos(a);
x = r * dd.d1;
y = r * dd.d2;
```
Bentley’s Rules

A. Modifying Data

B. Modifying Code
1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules
   a. Exploit Implicit Parallelism
   b. Exploit Inner Loop Parallelism
   c. Exploit Coarse Grain Parallelism
   d. Extra computation to create parallelism
Implicit Parallelism

Reduce the loop carried dependences so that “software pipelining” can execute a compact schedule without stalls.

Example:

```
xmax1 = MININT;
xmax2 = MININT;
for(i=0; i < N -1; i += 2) {
    if(X[i] > xmax1) xmax1 = X[i];
    if(X[i+1] > xmax2) xmax2 = X[i+1];
}
if((i < N) & & (X[i] > xmax1)) xmax1 = X[i];
xmax = (xmax1 > xmax2)?xmax1:xmax2;
```
Example 2

```c
curr = head;
tot = 0;
while(curr != NULL) {
    tot = tot + curr->val;
    curr = curr->next;
}
return tot;
```
Example 2

```
curr = head;
tot = 0;
while(curr != NULL) {
    tot = tot + curr->val;
    curr = curr->next;
}
return tot;
```

Also see Rule A.1.a Data Structure Augmentation

```
curr = head;
if(curr == NULL) return 0;
tot1 = 0;
tot2 = 0;
while(curr->next) {
    tot1 = tot1 + curr->val;
    tot2 = tot2 + curr->next->val;
    curr = curr->nextnext;
}
if(curr)
    tot1 = tot1 + curr->val;
return tot1 + tot2;
```
Exploit Inner Loop Parallelism

Facilitate inner loop vectorization (for SSE type instructions)

How? → by gingerly guiding the compiler to do so

- Iterative process by looking at why the loop is not vectorized and fixing those issues
- Most of the rules above can be used to simplify the loop so that the compiler can vectorize it
Exploit Coarse Grain Parallelism

Outer loop parallelism (doall and doacross loops)
Task parallelism
Ideal for multicores

You need to do the parallelism yourself → later lectures
Extra Computation to Create Parallelism

In many cases doing a little more work (or a slower algorithm) can make a sequential program a parallel one. Parallel execution may amortize the cost

Example:

```c
double tot;
tot = 0;
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        tot = tot + A[i][j];

tot = 0;
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        tot = tot + A[i][j];
```

```c
double tot;
double tottmp[N];
for(i = 0; i < N; i++)
    tottmp[i] = 0;
for(i = 0; i < N; i++) {  //parallelizable
    double tmp;
    for(j = 0; j < N; j++)
        tmp = tmp + A[i][j];
    tottmp[i]= tottmp[i]+ tmp;
}
tot = 0;
for(i = 0; i < N; i++)
    tot = tot + tottmp[i];
```
Bentley’s Rules

A Modifying Data
1. Space for Time
   a. Data Structure Augmentation
   b. Storing Precomputed Results
   c. Caching
   d. Lazy Evaluation
2. Time for Space
   a. Packing/Compression
   b. Interpreters
3. Space and Time
   a. SIMD

B Modifying Code
1. Loop Rules
   a. Loop Invariant Code Motion
   b. Sentinel Loop Exit Test
   c. Loop Elimination by Unrolling
   d. Partial Loop Unrolling
   e. Loop fusion
   f. Eliminate wasted iterations
2. Logic Rules
   a. Exploit Algebraic Identities
   b. Short Circuit Monotone functions
   c. Reordering tests
   d. Precompute Logic Functions
   e. Boolean Variable Elimination
3. Procedure Rules
   a. Collapse Procedure Hierarchies
   b. Coroutines
   c. Tail Recursion Elimination
4. Expression Rules
   a. Compile-time Initialization
   b. Common Subexpression Elimination
   c. Pairing Computation
5. Parallelism Rules
   a. Exploit Implicit Parallelism
   b. Exploit Inner Loop Parallelism
   c. Exploit Coarse Grain Parallelism
   d. Extra computation to create parallelism

© Saman Amarasinghe 2009
Traveling Salesman Problem

Definition

- List of cities
- Location of each city (x,y coordinates on a 2-D map)
- Need to visit all the cities
- What order to visit the cities so that the distance traveled is shortest

Exact Shortest Distance Algorithm → Exponential

A Good Greedy Heuristic

- Start with any city
- Find the closest city that haven’t been visited
- Visit that city next
- Iterate until all the cities are visited

© Saman Amarasinghe 2009
### TSP Example

```c
void get_path_1(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist;
    double totdist;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0)
                if(dist(curr, j) < cdist) {
                    cdist = dist(curr, j);
                    closest = j;
                }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}

double dist(int i, int j) {
    return sqrt((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x) +
                 (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

© Saman Amarasinghe 2009
TSP Example

original

```
void get_path_1(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist;
    double totdist;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0)
                if(distsq(curr, j) < cdist) {
                    cdist = distsq(curr, j);
                    closest = j;
                }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}

double dist(int i, int j) {
    return sqrt((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x) +
                 (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

algebraic identities

```
void get_path_2(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist;
    double totdist;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0)
                if(distsq(curr, j) < cdist) {
                    cdist = distsq(curr, j);
                    closest = j;
                }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}

double distsq(int i, int j) {
    return ((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x) +
            (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```
TSP Example

```c
void get_path_2(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist;
    double totdist;
    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;
    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0)
                if(distsq(curr, j) < cdist) {
                    cdist = distsq(curr, j);
                    closest = j;
                }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}

double distsq(int i, int j) {
    return ((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x) +
            (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

B.2.a Exploit

Algebraic Identities

B.4.b Common

Subexpression Elimination

© Saman Amarasinghe 2009
### TSP Example

```c
void get_path_3(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist, tdist;
    double totdist;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;

    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;

        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0) {
                tdist = distsq(curr, j);

                if(tdist < cdist) {
                    cdist = tdist;
                    closest = j;
                }
            }

        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}
```

### B.4.b Common Subexpression Elimination

### B.1.a Loop Invariant Code Motion

```c
void get_path_4(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist, tdist;

    double cx, cy;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;

    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;

        cx = Cities[curr].x;
        cy = Cities[curr].y;

        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0) {
                tdist = (Cities[j].x - cx)*(Cities[j].x - cx) +
                        (Cities[j].y - cy)*(Cities[j].y - cy);

                if(tdist < cdist) {
                    cdist = tdist;
                    closest = j;
                }
            }

        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}
```

© Saman Amarasinghe 2009
TSP Example

```c
void get_path_4(int path[])
{
    int visited[MAXCITIES];
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++)
    {
        tdist = (Cities[curr].x - cx)*(Cities[curr].x - cx) +
                (Cities[curr].y - cy)*(Cities[curr].y - cy);
        if(tdist < cdist) {
            cdist = tdist;
            closest = j;
        }
    }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
}

void get_path_5(int path[])
{
    int visited[MAXCITIES];
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;

    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++)
    {
        tdist = (Cities[curr].x - cx)*(Cities[curr].x - cx) +
                (Cities[curr].y - cy)*(Cities[curr].y - cy);
        if(tdist < cdist) {
            cdist = tdist;
            closest = j;
        }
    }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
}
```

© Saman Amarasinghe 2009
void get_path_5(int path[]) {
    int visited[MAXCITIES];
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;
    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;
    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        cx = Cities[curr].x;
        cy = Cities[curr].y;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0) {
                tdist = (Cities[j].x - cx)*(Cities[j].x - cx);
                if(tdist < cdist) {
                    tdist += (Cities[j].y - cy)*(Cities[j].y - cy);
                    cdist = tdist;
                    closest = j;
                }
            }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}

void get_path_6(int path[]) {
    int visited[MAXCITIES];
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;
    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;
    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        cx = Cities[curr].x;
        cy = Cities[curr].y;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0) {
                tdist = (Cities[j].x - cx)*(Cities[j].x - cx);
                if(tdist < cdist) {
                    tdist += (Cities[j].y - cy)*(Cities[j].y - cy);
                    cdist = tdist;
                    closest = j;
                }
            }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}
TSP Example

void get_path_6(int path[]) {
    int visited[MAXCITIES];
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;
    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;

    curr = MAXCITIES - 1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        cx = Cities[curr].x;
        cy = Cities[curr].y;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0) {
                double tx = (Cities[j].x - cx);
                tdist += tx*tx;
                if(tdist < cdist)
                    double ty = (Cities[j].y - cy);
                    tdist += ty*ty;
                    cdist = tdist;
                    closest = j;
            }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
        curr = closest;
    }
}

void get_path_7(int path[]) {
    int unvisited[MAXCITIES];
    int i, j, k, curr, closek, closej;
    double cdist, tdist, cx, cy, totdist;
    for(i=0; i < MAXCITIES; i++)
        unvisited[i] = i;

    curr = MAXCITIES - 1;
    unvisited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        cx = Cities[curr].x;
        cy = Cities[curr].y;
        for(j=0; j < MAXCITIES-i; j++)
            if(unvisited[j] == 1) {
                double tx = (Cities[j].x - cx);
                tdist += tx*tx;
                if(tdist < cdist)
                    double ty = (Cities[j].y - cy);
                    tdist += ty*ty;
                    cdist = tdist;
                    closek = k;
                    closej = j;
            }
        path[i] = closest;
        unvisited[closest] = unvisited[MAXCITIES - i - 1];
        totdist += dist(curr, closek);
        curr = closek;
    }
}

B.4.b Common Subexpression Elimination

B.1.f Eliminate Wasted Iterations

© Saman Amarasinghe 2009
void get_path_6(int path[]) {
    int visited[MAXCITIES];
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;
    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;
    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        cx = Cities[curr].x;
        cy = Cities[curr].y;
        for(j=0; j < MAXCITIES; j++)
            if(visited[j] == 0) {
                double tx = (Cities[j].x - cx);
                tdist = tx*tx;
                if(tdist < cdist) {
                    double ty = (Cities[j].y - cy);
                    tdist += ty*ty;
                    if(tdist < cdist) {
                        cdist = tdist;
                        closest = j;
                    }
                }
            }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}

void get_path_8(int path[]) {
    int i, j, curr, closest;
    double cdist, tdist, cx, cy, totdist;
    for(i=0; i < MAXCITIES; i++)
        visited[i] = 0;
    curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
        cdist = MAXDOUBLE;
        cx = Cities[curr].x;
        cy = Cities[curr].y;
        for(j=0; j < MAXCITIES-i; j++)
            if(visited[j] == 0) {
                double tx = (Cities[j].x - cx);
                tdist = tx*tx;
                if(tdist < cdist) {
                    double ty = (Cities[j].y - cy);
                    tdist += ty*ty;
                    if(tdist < cdist) {
                        cdist = tdist;
                        closest = j;
                    }
                }
            }
        path[i] = closest;
        visited[closest] = 1;
        totdist += dist(curr, closest);
        curr = closest;
    }
}
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Result</th>
<th>Time</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Original</td>
<td>96.344605</td>
<td>519.0697 ms</td>
<td>1.52</td>
</tr>
<tr>
<td>2</td>
<td>B.2.a Exploit Algebraic Identities</td>
<td>96.344605</td>
<td>341.0185 ms</td>
<td>1.08</td>
</tr>
<tr>
<td>3</td>
<td>B.4.b Common Subexpression Elimination</td>
<td>96.344605</td>
<td>314.5534 ms</td>
<td>1.02</td>
</tr>
<tr>
<td>4</td>
<td>B.1.a Loop Invariant code Motion</td>
<td>96.344605</td>
<td>308.9298 ms</td>
<td>1.02</td>
</tr>
<tr>
<td>5</td>
<td>B.1.c Reordering Tests</td>
<td>96.344605</td>
<td>302.6724 ms</td>
<td>0.91</td>
</tr>
<tr>
<td>6</td>
<td>B.4.b Common Subexpression Elimination</td>
<td>96.344605</td>
<td>331.6027 ms</td>
<td>2.7</td>
</tr>
<tr>
<td>7</td>
<td>B.1.f Eliminate Waded Iterations</td>
<td>97.738769</td>
<td>123.7693 ms</td>
<td>1.23</td>
</tr>
<tr>
<td>8</td>
<td>B.1.f Eliminate Waded Iterations</td>
<td>99.003897</td>
<td>100.6096 ms</td>
<td>5.16</td>
</tr>
</tbody>
</table>