Efficient Programs

Group 3 - Magic Hexagon Optimization

Steps taken

1. Algorithmic optimization of solve() function
2. Adapt exploration order
3. Bisecting search space
4. Optimization of sum() function
5. Compiler flags

Algorithmic optimization of solve() function

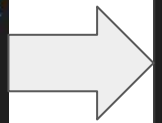
Challenges:

- Understand the program code.
- Function solve() contains many goto jumps.
- Figure out which goto jumps are necessary.
- Reduce the number of goto jumps.
- Preserve program correctness.

Algorithmic optimization of solve() function

Delete unnecessary goto jumps.

```
restart:
 for (i=0; i<r*r; i++) {
   Var *v = &vs[i];
   if (v->lo == v->hi && occupation[v->lo-o] != i) {
     if (occupation[v->lo-o] < r*r)
       return 0;
     occupation[v->lo-o] = i;
     goto restart;
   }
 }
```



```
restart:
 for (i=0; i<r*r; i++) {
   Var *v = &vs[i];
   if (v->lo == v->hi && occupation[v->lo-o] != i) {
     if (occupation[v->lo-o] < r*r)
       return 0;
     occupation[v->lo-o] = i;

   }
 }
```

Algorithmic optimization of solve() function

Jump only when boundaries for all points are updated.

Same solution applied for sum constraints.

```
for (i=0; i<r*r; i++) {
  Var *v = &vs[i];
  if (v->lo < v->hi) {
    if (occupation[v->lo-o] < r*r) {
      v->lo++;
      goto restart;
    }
    if (occupation[v->hi-o] < r*r) {
      v->hi--;
      goto restart;
    }
  }
}
```

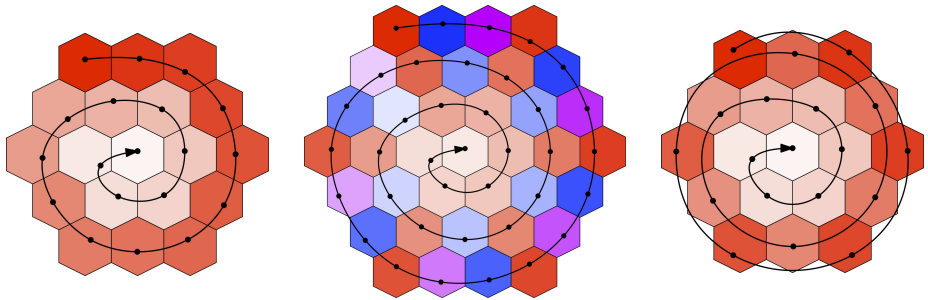
```
bool update_boundaries = false;
for (i=0; i<r*r; i++) {
  Var *v = &vs[i];
  if (v->lo < v->hi) {
    if (occupation[v->lo-o] < r*r) {
      v->lo++;
      update_boundaries = true;
    }
    if (occupation[v->hi-o] < r*r) {
      v->hi--;
      update_boundaries = true;
    }
  }
}
if(update_boundaries){
  goto restart;
}
```

Adapt exploration order

- Goal:

  By exploring the variables in a different order inside labeling() we aim to check more constraints earlier and consequently eliminate parts of the search tree

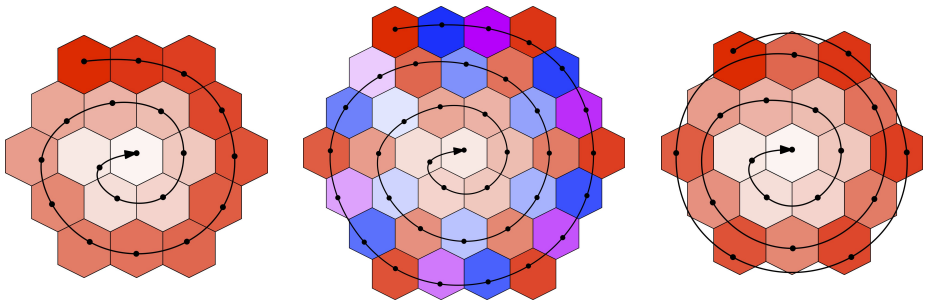
- Attempted strategies:
  - Variable with fewest values first
  - Spiral order
  - Wheel order
  - Spiral order with corners first



Spiral | Wheel | Spiral + corners

Adapt exploration order

	Left to right, Top to bottom	**Variable with fewest values first**	**Spiral order**	**Wheel order**	**Spiral order with corners first**
Precomputed	No	No	Yes	Yes	Yes
Leafs visited	15,808,871	14,286,898	2,720,000	40,307,012	2,270,926
Running time	51.17 sec	40.74 sec	6.83 sec	121.51 sec	6.21 sec
Improvement	-	20.4%	86.7 %	-137.5%	87.9%



Bisecting search space

- Goal:
  Eliminate branches of the search / recursion tree earlier
  by calling solve() on already stronger constrained fields
- Strategy: Divide & Conquer
  Divide the value range in the labeling() step into
    lower and
    upper range
  and solve separately

⇒ Classic bisection of search space (binary search, > git bisect, etc.)

Bisecting search space

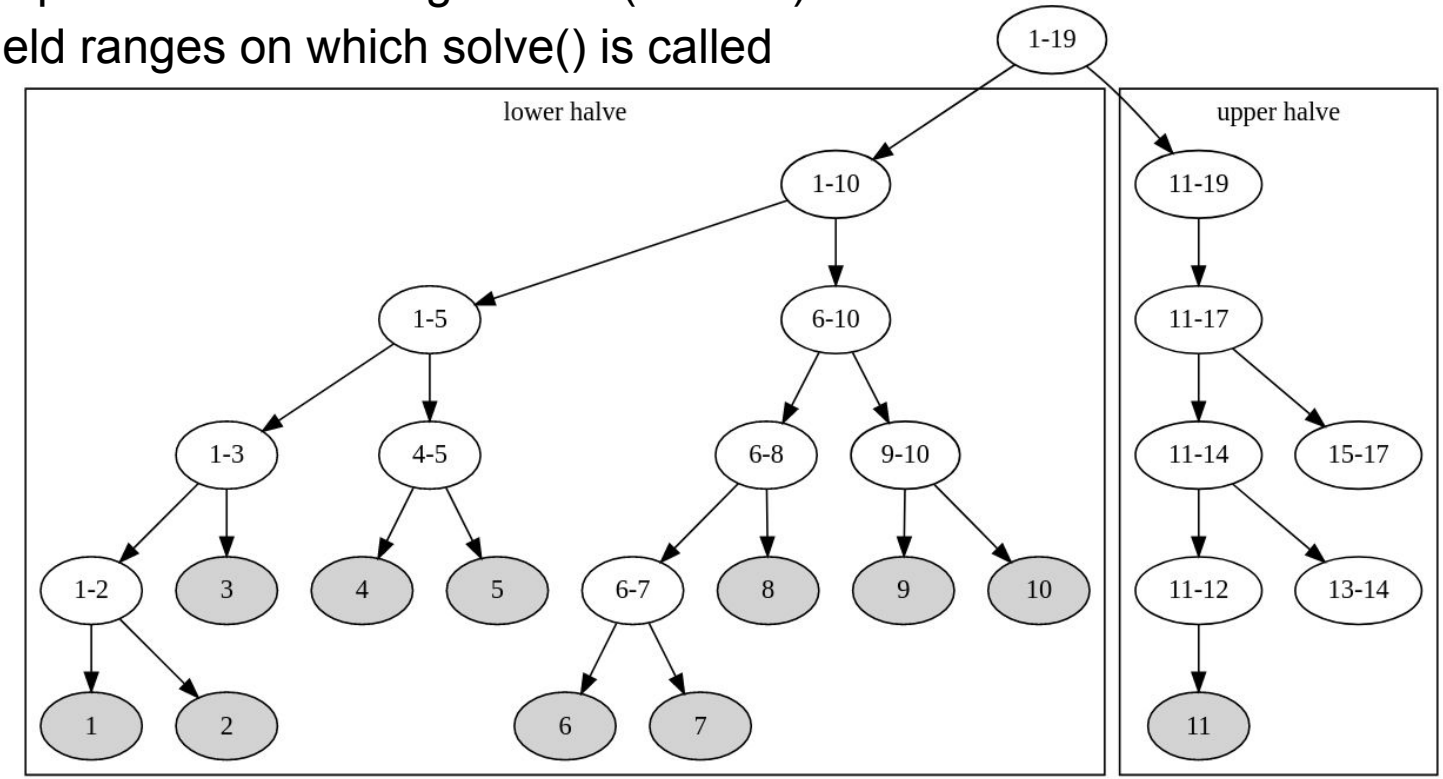
```
void labeling (...) {
 // ...
 if(hexagonEntry->lo == hexagonEntry->hi)
   // hexagonEntry already fully constrained, continue to next hexagon index.
   return labeling(hexagon,index+1,order);
 // ...
 long mid = (oldHexagon[i]->lo + oldHexagon[i]->hi) / 2;
 // ...
 // initialize new hexagon and set lower-halve bounds for entry i
 newHexagon[i].lo = oldHexagon[i]->lo;
 newHexagon[i].hi = mid;
 if (solve(newHexagon))
   labeling(newHexagon,index,order);
 // ...
 // re-initialize new hexagon and set upper-halve bounds for entry i
 newHexagon[i].lo = mid+1;
 newHexagon[i].hi = oldHexagon[i]->hi;
 if (solve(newHexagon))
   labeling(newHexagon,index,order);
 // ...
}
```

Bisecting search space

Resulting search space for first Hexagon field (index 0)

Nodes are new field ranges on which solve() is called

> ./magichex 3 2



Bisecting search space

- Improvements for > ./magichex 4 3 14 33 30 34 39 6 24 20

original:	15808871 leafs visited	
original+bisect:	4930863 leafs visited	69 % reduction
spiral access:	2270926 leafs visited	
spiral access+bisect:	397928 leafs visited	82 % reduction

Optimization of sum() function

- Context of sum()
  - Calculates and sets constraints for upper and lower boundary
  - Ensures that the sum of variables holds certain constraints
- Early termination
  - Utilizing early termination conditions
- Optimizing conditional checks
- Reduced cycles and run time

Optimization of sum() function

- Early termination

```
for (i=0, hexagonEntry_p=hexagon; i<nv; i++, hexagonEntry_p+=stride) {
  assert(hexagonEntry_p>=hexagonStart);
  assert(hexagonEntry_p<hexagonEnd);
  assert(hexagonEntry_p->id != PLACEHOLDER_ENTRY_ID);
  hi -= hexagonEntry_p->lo;
  lo -= hexagonEntry_p->hi;
}

if(hi < lo){
  return NOSOLUTION;
}
```

Optimization of sum() function

- Optimizing conditional checks

```
CHANGE_IDENTIFIER sethi(HexagonEntry *hexagonEntry, long x) {
  assert(hexagonEntry->id != PLACEHOLDER_ENTRY_ID);
  if (x < hexagonEntry->hi) {
    hexagonEntry->hi = x;
    if (hexagonEntry->lo > hexagonEntry->hi)
      return NOSOLUTION;
    else
      return CHANGED;
  }
  return NOCHANGE;
}
```

Compiler flags

- Tried various flags mentioned in the gcc docs
- Using the -O flag
  - 0 - 3, no optimization to highest optimization
  - s for size optimization
  - g for optimizing the debugging experience
  - fast for optimizing speed only
- funroll-loops, funroll-all-loops
- march=native
- fwhole-program
- fsplit-loops
- AutoFDO

Compiler flags

- -O0: Baseline
- -O1: ~60% faster
- -O2: ~70% faster
- -O3: ~71% faster
- -Ofast: ~70% faster
- -Os: ~47% faster
- -Og: ~62% faster
- -O3 -funroll-loops: ~76% faster
- -O3 -funroll-all-loops: ~75% faster
- -O3 -fsplit-loops -funroll-loops: ~76.1% faster

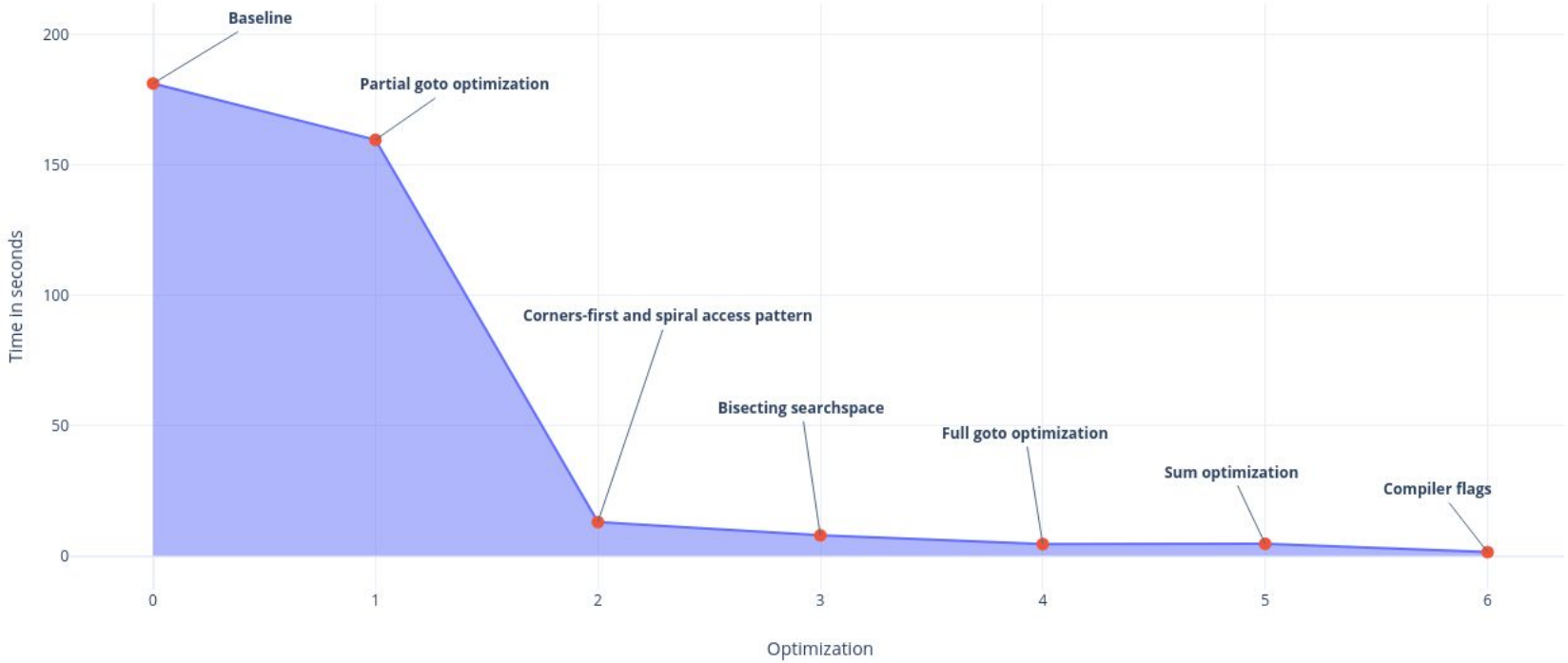
Compiler flags

- **-Wall -O3 -DNDEBUG -funroll-loops -fsplit-loops -march=native -fwhole-program:** ~79% faster
  - -O3 most of optimization flags specified by gcc
  - -DNDEBUG
  - -funroll-loops
    - unroll loops whose number of iterations can be determined at compile time or upon entry to the loop, loop peeling, more code
  - -fsplit-loops
    - split a loop into two if it contains a condition that’s always true for one side of the iteration space and false for the other.
  - -march=native
    - tune generated code for the micro-architecture and ISA extensions of the host CPU
  - -fwhole-program
    - all public functions and variables (except main) become static and in effect are optimized more aggressively

Benchmarks

- **Baseline: 181.1950 +- 0.0531 seconds**
  - 15809528 leafs visited, 849851023565 cycles, 2372629024670 instructions
- **Partial goto optimization: 159.582 +- 0.140 seconds**
  - 15808871 leafs visited, 748731599056 cycles, 2096155150602 instructions
- **Corners-first and spiral access pattern: 12.9534 +- 0.0245 seconds**
  - 2270926 leafs visited, 60759414081 cycles, 221531374766 instructions
- **Bisecting search space: 7.8680 +- 0.0132 seconds**
  - 397928 leafs visited, 21159230657 cycles, 132925675516 instructions
- **Full goto optimization: 4.50446 +- 0.00202 seconds**
  - 397929 leafs visited, 21159230657 cycles, 73292614871 instructions
- **Sum optimization: 4.62492 +- 0.00896 seconds**
  - 397929 leafs visited, 21855899006 cycles, 73105231178 instructions
- **Compiler flags: 1.425913 +- 0.000960 seconds**
  - 397929 leafs visited, 6962204289 cycles, 21246335112 instructions

Benchmarks


Baseline
Partial goto optimization
Corners-first and spiral access pattern
Bisecting searchspace
Full goto optimization
Sum optimization
Compiler flags
Time in seconds
200
150
100
50
0
0
1
2
3
4
5
6
Optimization

Project online at:

http://www.complang.tuwien.ac.at/anton/lvas/effizienz-abgaben/2023w/group03/