Real World FunLog

Kendall Stewart
CS510 Logic & Prog. Lang
Fall 2014

Original Goal

- Solver for a “courier problem”
    - Input:
        - A set of deliveries, where each delivery is a triple (source, destination, parcel). The source and destination are physical locations, and a parcel is a pair (weight, value).
        - A vehicle, which is a pair (daily range, weight capacity).
        - A starting location.
- Output:
    - A sequence of locations, marked as either start, pickup or dropoff, that minimizes distance travelled while maximizing value delivered, subject to range and capacity constraints. No location should be repeated, except that the start and end locations should be the same.

“Courier Problem”

- Originally I thought this was my own formulation, but it turns out this is also known as the “Vehicle Routing Problem with Pickup and Delivery”
- Yes, it’s NP-Complete.
    - This makes sense: intuitively, it draws on the Traveling Salesperson Problem and the Knapsack Problem

Project

- My first step was to get to know FunLog's affinity for NP-Complete optimization problems by first trying to implement as solver for the Traveling Salesperson Problem

- Turns out this took a long time to get right, and led to a few side projects…

TSP in FunLog

- Best solver? SAT? IP? SMT?
  - SAT doesn't know about arithmetic (could implement partial adders in boolean logic?)
  - IP is fairly restrictive (can't do inequalities or disjunctions easily)
  - SMT is the best choice … but SMT is for decision problems, so let's start with a decision version of TSP:
    - `∃? path . hamiltonian(path) && len(path) ≤ k`

TSP in FunLog

- I used an adjacency matrix representation for the graph:

```
numLocations = 9
dim cityInts#Int = [1 .. numLocations]

cities = array #(cityInts)
  [
      "Seattle",
      "San Francisco",
      "Los Angeles",
      "Denver",
      "Austin",
      "Chicago",
      "St. Louis",
      "New Orleans",
      "New York"
  ]

-- Retrieved with Google's "Distance Matrix" API. All data in kilometers.
edges = array #(cityInts,cityInts)
  [
    0,    1299, 1827, 2114, 3425, 3320, 3357, 4171, 4598,
    1299, 0,    613,  2011, 2828, 3430, 3308, 3657, 4676,
    1826, 614,  0,    1635, 2216, 3244, 2937, 3045, 4468,
    2115, 2015, 1636, 0,    1473, 1616, 1365, 2081, 2862,
    3423, 2830, 2217, 1474, 0,    1803, 1328, 817,  2805,
    3321, 3429, 3243, 1612, 1803, 0,    477,  1490, 1271,
    3353, 3304, 2939, 1366, 1327, 477,  0,    1088, 1533,
    4198, 3657, 3045, 2083, 818,  1490, 1088, 0,    2099,
    4589, 4679, 4493, 2862, 2803, 1271, 1529, 2097, 0
  ]
```

TSP in FunLog

- Hamiltonicity is relatively easy to express:

```
dim pathIndex = [ 1 .. numLocations + 1 ]

∃ (path : array #(pathIndex) Int) . (
     ∀x . (valid(path[x]))
  && path[1] == path[numLocations + 1]
  && ∀x . ∀y . (x /= y && x /= 1 -> path[x] /= path[y])
  && ∀x . (edges.(path[x], path[x+1]) /= 0)
)
```

- This doesn't work in FunLog, because we can't index into a matrix using SMT variables.

- Need to express connectedness as a proposition.

TSP in FunLog

```
dim pathIndex = [ 1 .. numLocations + 1 ]

∃ (path : array #(pathIndex) Int) . (
     ∀x . (valid(path[x]))
  && path[1] == path[numLocations + 1]
  && ∀x . ∀y . (x /= y && x /= 1 -> path[x] /= path[y])
  && ∀x . (connected(path[x], path[x+1]))
)

connected(p,q) =
  ∃ a . ∃ b . (p == a && q == b && edges.(a,b) /= 0)
```

- Two variables represent a pair of connected vertices if there exist two vertices with a non-zero edge that match the value of the variables.

TSP in FunLog

- Adding distances poses an additional problem. As with connectedness, we can't use SMT variables to lookup in the adjacency list to find the distance between two points. We need to encode the distance lookup as some kind of "widget" in the SMT constraints using propositions.

- This was a tough problem to crack.

- The solution: add more SMT variables to represent the distance between two points on the path.

TSP in FunLog

```
dim cityInts#Int = [1 .. numLocations]
dim pathIndex = [ 1 .. numLocations + 1 ]

∃ path  : array #(pathIndex) Int .
∃ dists : array #(cityInts)  Int . (
     ∀x . (valid(path[x]))
  && path[1] == path[numLocations + 1]
  && ∀x . ∀y . (x /= y && x /= 1 -> path[x] /= path[y])
  && ∀x . (connected(path[x], path[x+1]))
  && ∀x . (weight(path[x], path[x+1], dists[x]))
  && ∑x . dists[x] <= k
)


connected(p,q) =
  ∃ a . ∃ b . (p == a && q == b && edges.(a,b) /= 0)

weight(p,q,w) =
  ∃ a . ∃ b . (p == a && q == b && w == edges.(a,b))
```

SMT Minimization

- Could've stopped here and moved on to the VRP problem … but the inability to do minimization with SMT bothered me.

- Recall the trick for converting decision problems into optimization problems: do a binary search over the decision space.

- Need to implement this binary search as a new "strategy" for SMT in the FunLog interpreter.

SMT Minimization

- Basic idea: start at 1 and expand the search space exponentially. Use the first satisfying result as an upper bound, and the previous result as a lower bound, and conduct a binary search in that space to find the minimum satisfying result.

- Problem: as we've seen (in tableau solvers etc), finding a satisfying result might be easy, while deciding unsatisfiability might require an exhaustive search and take a long time.

- If we want to achieve anything resembling reasonable efficiency, we'll have to make our minimization partially decidable by using timeouts.

Timeouts

- Yices2 has a built-in timeout feature, but its precision is fixed at one-second intervals.
- We can use GHC's System.Timeout package for microsecond precision.

Timeouts

```
import System.Timeout

yicesTimeout' :: String -> [String] -> [CmdY] -> Int -> IO ResY
yicesTimeout' yPath yOpts cmds tout =
  do (Just hIn, Just hOut, Just hErr, ph) <-
       createProcess (proc yPath yOpts)
         { std_in  = CreatePipe
         , std_out = CreatePipe
         , std_err = CreatePipe
         , create_group = True
         }

     let input = (unlines $ map show (cmds ++ [CHECK, MODEL, EXIT]))
     hPutStr hIn input >> hFlush hIn
     attempt <- timeout tout (poll ph)
     terminateProcess ph >> waitForProcess ph
     case attempt of
       Nothing -> do
         return (InCon ["timeout"])
       Just _  -> do
         _   <- hGetContents hErr
         out <- hGetContents hOut
         return $
           case lines out of
              "sat"     : ss -> Sat     (parseExpYs $ unlines $ filter (not.null) ss)
              "unknown" : ss -> Unknown (unlines ss)
              "unsat"   : ss -> UnSat   (unlines ss)
              other          -> InCon   other

poll processHandle  =
  do attempt <- getProcessExitCode processHandle
     case attempt of
       Nothing -> poll processHandle
       Just c  -> return c
```

Expansion Phase

```
{- Expansion -}

-- Initial timeout (usec) for expansion phase
initialExpandTimeout = 5000

-- Bound growth factor for expansion phase
expandBoundGrowth   = 8

-- create the initial search space
expand bound minExpY cmds timeout =
  do let cmds'           = cmds
                         -- add in assertion for minimization expression
                         ++ [ASSERT (minExpY :<= LitI bound)]

                         -- call yices with timeout
     result              <- yicesTimeout [] cmds' timeout
     case result of
                         -- if fails, expand bound
       InCon ["timeout"] -> do print  (bound, timeout, "timeout")
                               expand (bound * expandBoundGrowth)
                                       minExpY cmds timeout

       UnSat s           -> do print  (bound, timeout, "unsat")
                               expand (bound * expandBoundGrowth)
                                       minExpY cmds timeout

                         -- if succeeds, return previous bound and current bound
       Sat   sol         -> do print  (bound, timeout, "sat")
                               return (bound `div` expandBoundGrowth, bound)
```

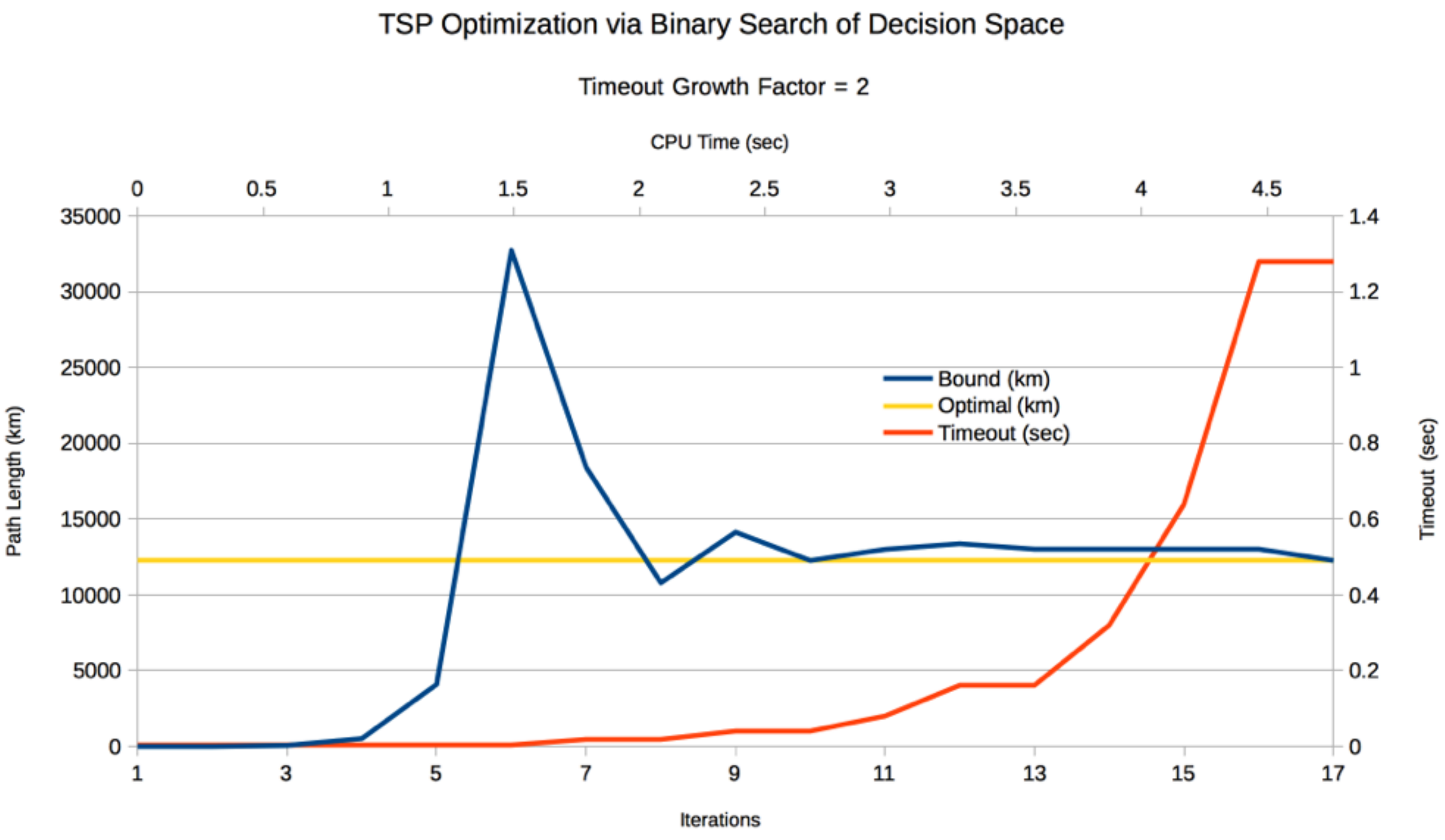
Search Phase

```
{- Search -}

-- Initial timeout (usec) for search phase
initialSearchTimeout = 20000

-- Timeout growth rate on failure (timeout)
searchTimeoutGrowth = 2

-- binary search through the search space
search low high minExpY cmds timeout

    -- base case: minimal difference between success and failure
  | (high-low) <= 1 = do print (high, timeout, "sat")
                         runYices [] (cmds ++ [ASSERT (minExpY :<= LitI high)])

    -- recursive case: search
  | otherwise       =
    do let bound    = (high - low) `div` 2 + low
       let cmds'    = cmds
                    ++ [ASSERT (minExpY :<= LitI bound)]
       result      <- yicesTimeout [] cmds' timeout
       case result of
         -- if failure due to timeout, increase bound and increase timeout
         InCon ["timeout"] -> do print (bound, timeout, "timeout")
                                 search bound high minExpY cmds (grow timeout)

         -- if failure due to unsat, increase bound but do not change timeout
         UnSat s           -> do print (bound, timeout, "unsat")
                                 search bound high minExpY cmds timeout

         -- if success, check result and replace upper bound with result
         Sat sol           -> do
           print (bound, timeout, "sat")
           let solMap = (DM.fromList(map oneExp sol))
           let VBase (LInt min) = solMap DM.! (show minExpY)
           search low min minExpY cmds timeout

   where grow t = floor (fromIntegral t * searchTimeoutGrowth)
```

Putting it Together

```
solveCon env deltaEnv ts ss (Min m) SMT (v@(VNS (NSYices e))) =
  do    -- evaluate the minimization expression
        VNS(NSYices minExpY) <- evalC m deltaEnv return

        -- expand bound exponentially to find a search space
        (low,high)           <- expand 1 minExpY cmds initialExpandTimeout

        -- find the minimum in the search space
        Sat solution         <- search low high minExpY cmds initialSearchTimeout

        -- return the solution substitution
        let subst            = (DM.fromList (map oneExp solution))
        return                 (map (\(nm,v) -> (nm, subVal subst v)) ts)

where define(nm,i,v)         = DEFINE (name nm, i) Nothing
      cmds                   = map define ss ++ [ASSERT e]
```

Performance

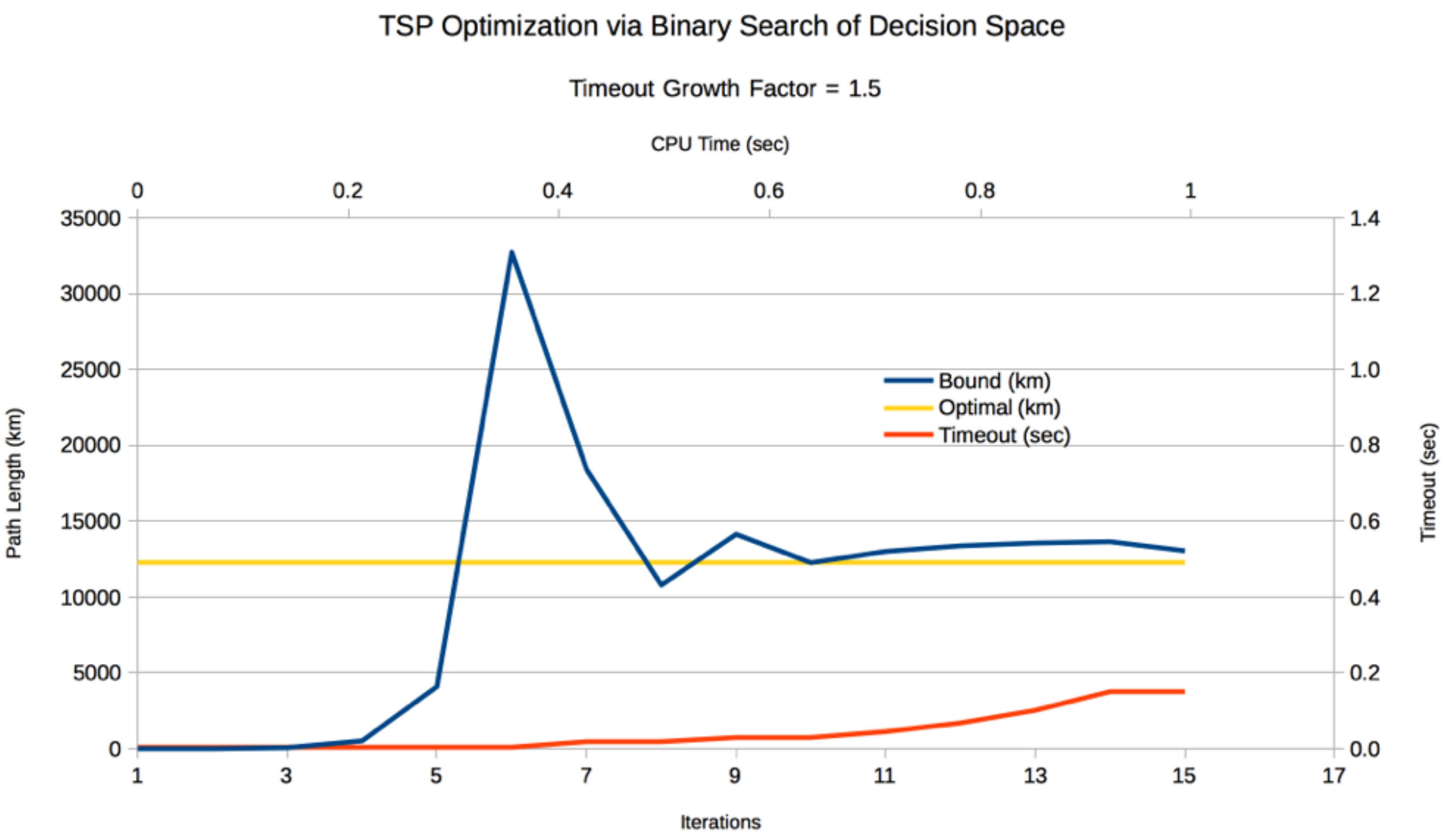


Drawbacks

Accuracy and performance are very sensitive to the timeout parameters (and to the state of machine that the solver is running on):

```
-- Initial timeout (usec) for expansion phase
initialExpandTimeout = 5000

-- Bound growth factor for expansion phase
expandBoundGrowth    = 8

-- Initial timeout (usec) for search phase
initialSearchTimeout = 20000

-- Timeout growth factor for search phase
searchTimeoutGrowth  = 2
```

Drawbacks


TSP Optimization via Binary Search of Decision Space
Timeout Growth Factor = 1.5
CPU Time (sec)
0
0.2
0.4
0.6
0.8
1
35000
30000
25000
20000
15000
10000
5000
0
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Path Length (km)
Timeout (sec)
Bound (km)
Optimal (km)
Timeout (sec)
1
3
5
7
9
11
13
15
17
Iterations

The All-Seeing Oracle

- Where'd that "optimal" line come from, anyway?

- In addition to their "Distance Matrix" API (which was used to get the instance for my examples), Google also has a "Directions" API that employs a proprietary TSP solver to allow you to find the optimal circuit between a set of locations.

- FunLog did pretty well vs. Google on 9 cities (the maximum you can ask Google to figure out)

- Demonstration: generating instances on the fly

Possible Extensions

- Attempting to heuristically compute the timeout parameters based on properties of the problem
- Utilizing parallelism to try different parameters simultaneously
- Allowing the programmer to set a “patience” level in the FunLog code when specifying an SMT optimization problem