Heuristic Search In The Cloud

IndyCloud 2019

Who am I? – Jordan T. Thayer

- B.S. CS, RHIT, 2006
- PhD Artificial Intelligence, U. of New Hampshire, 2012
  - Advisor: Wheeler Ruml
  - Thesis: Heuristic Search Under Time and Quality Bounds
    - This stuff isn’t my thesis area, but it’s closely related
- Since then
  - Logistics, Planning, Scheduling
  - Formal Verification
  - Static Analysis
- Currently Sr. Software Engineer for SEP

Talk Outline

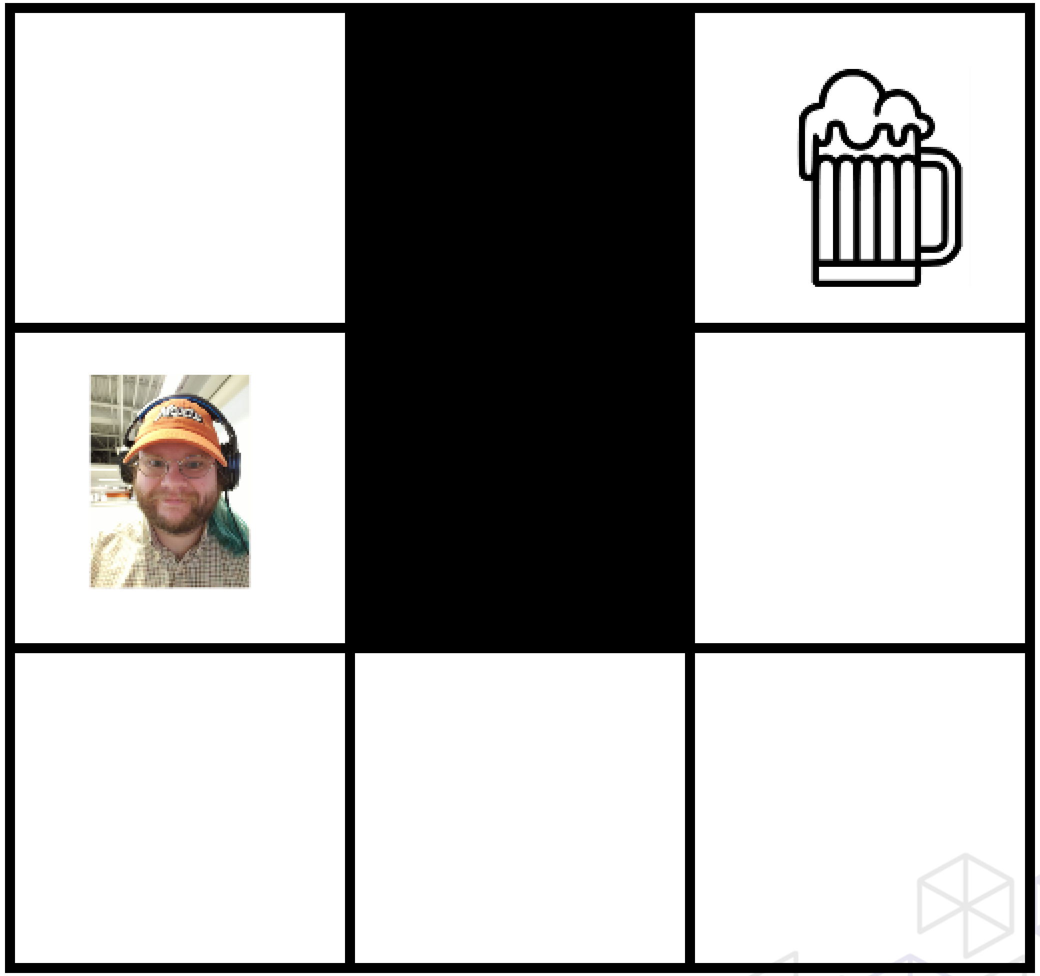
- What is heuristic search and why should I care?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
    - Pancake Flipping – a toy domain
    - The Travelling Salesman Problem
- Distributed Depth First Search
- Distributed Depth First Search In The Cloud

What is Search, Why do I care?

- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics

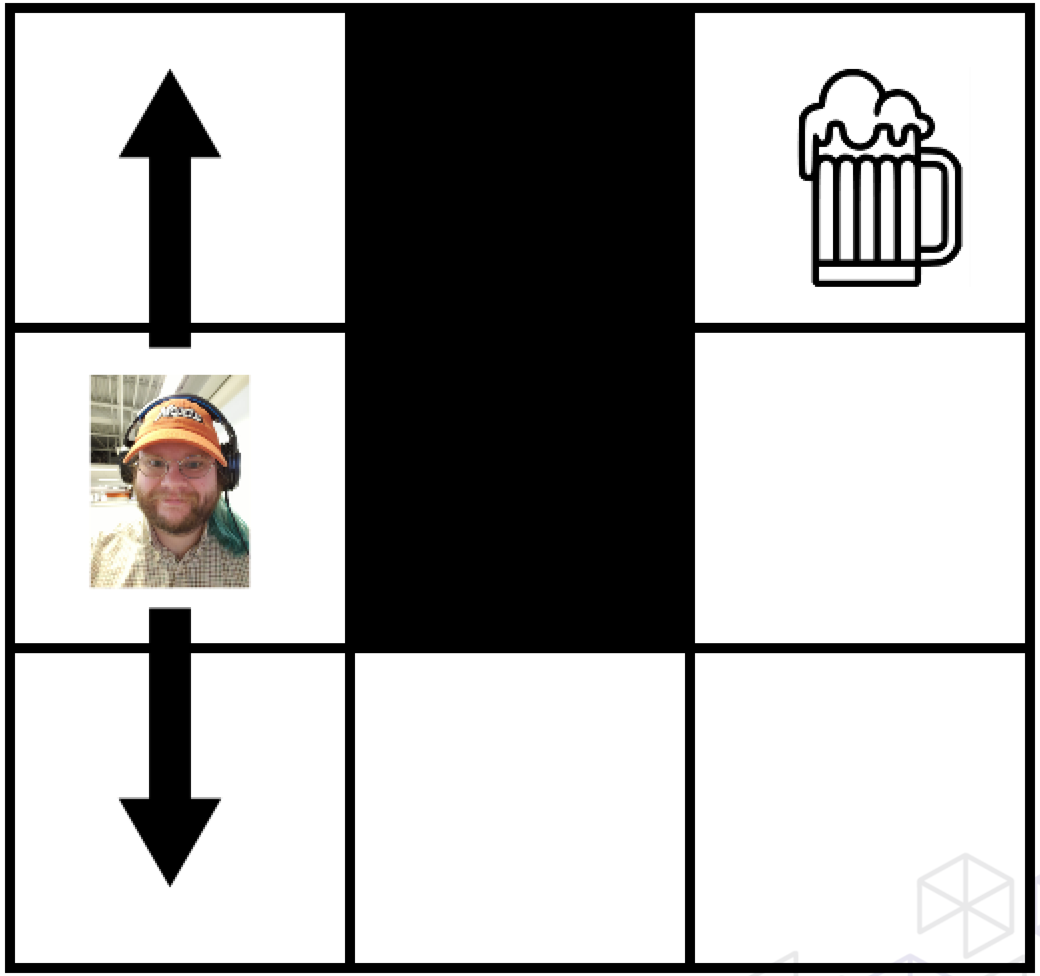
What is Search, Why do I care?

- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics



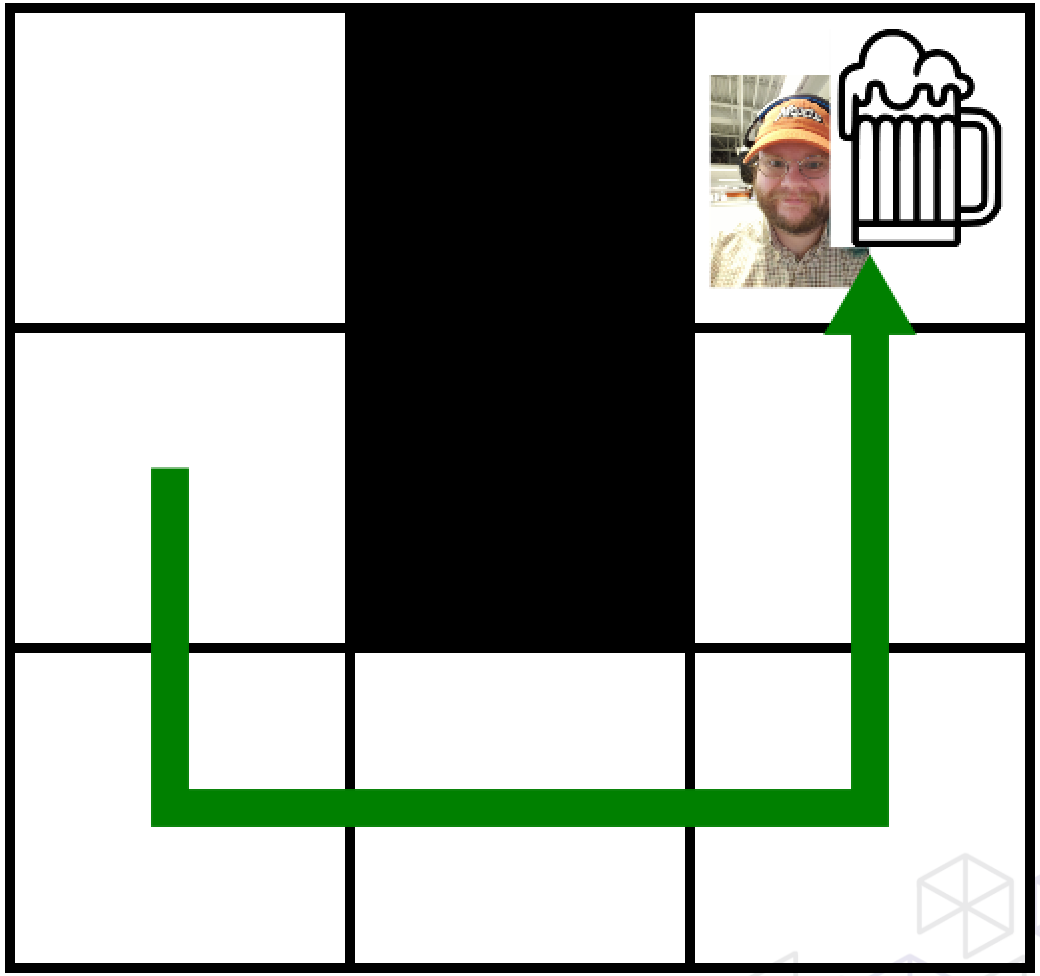
What is Search, Why do I care?

- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics



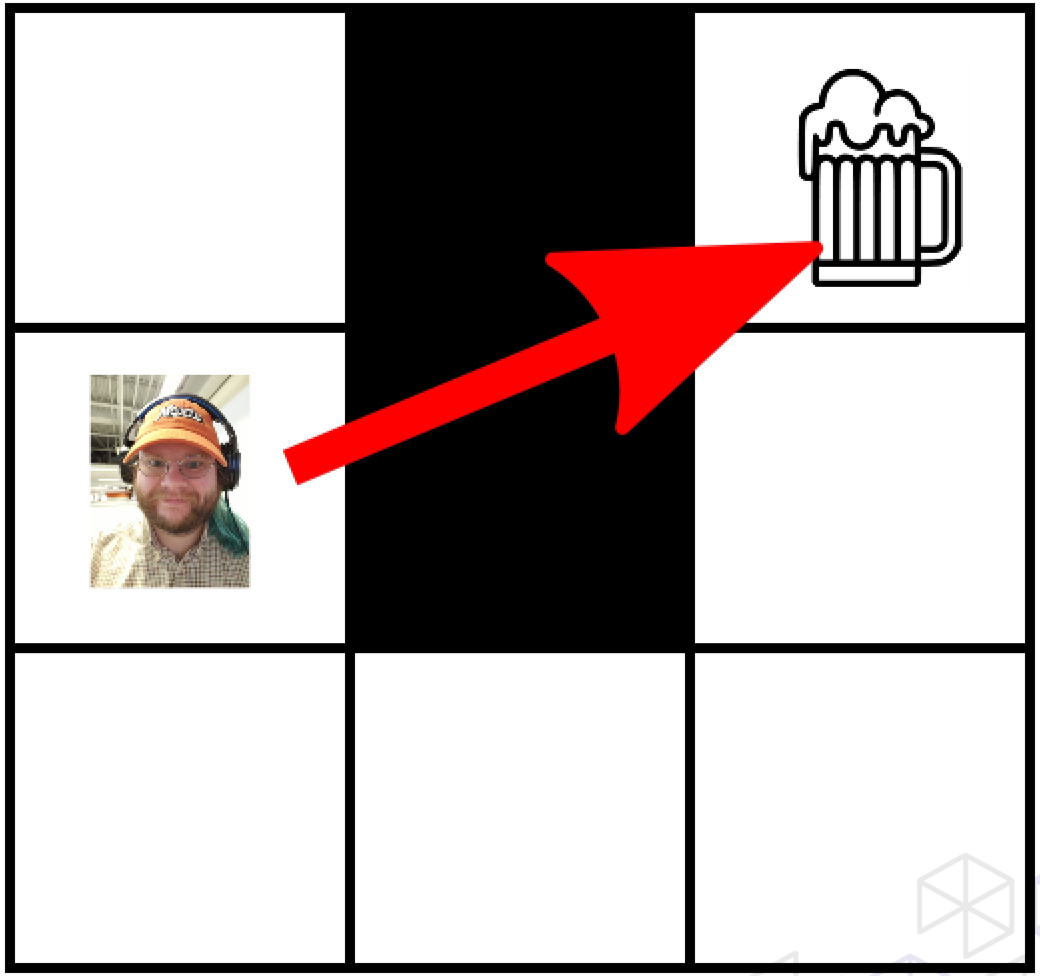
What is Search, Why do I care?

- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics



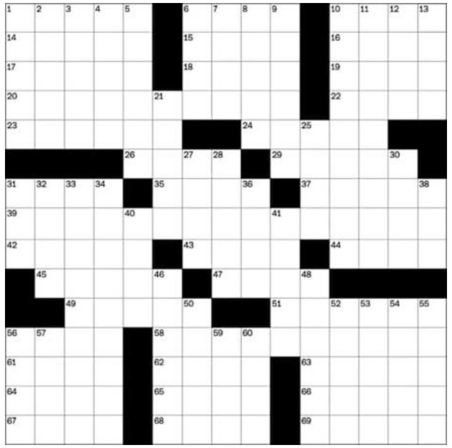
What is Search, Why do I care?

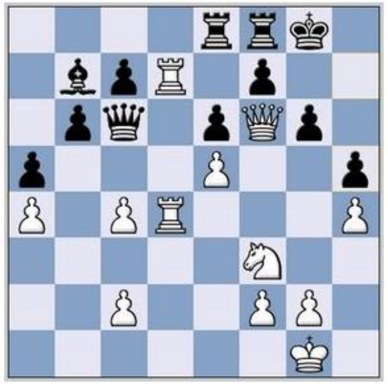
- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics

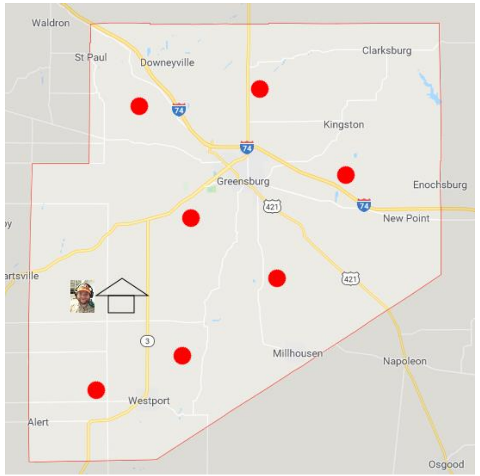


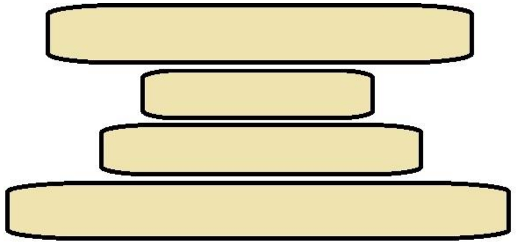
What is Search, Why do I care?

- Search is a *general* technique for solving problems
- The search cares about your problem *using this abstraction:*
  - States
  - Actions
  - Goals
  - Heuristics



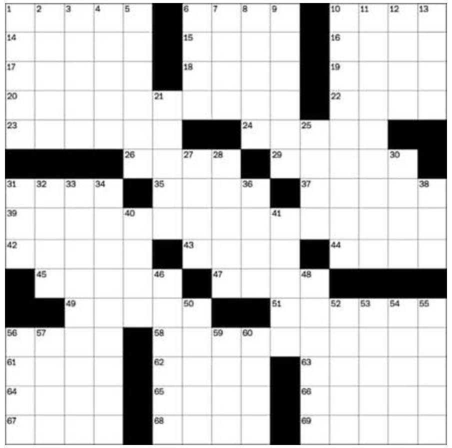


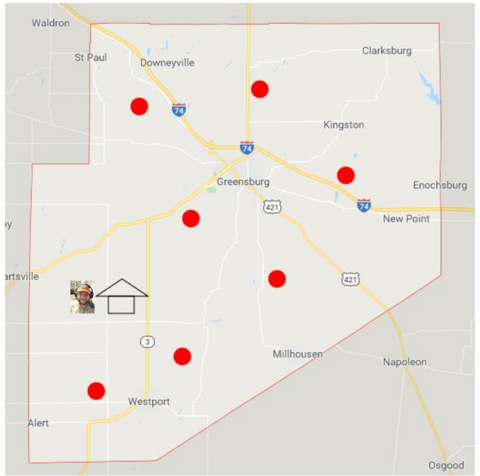


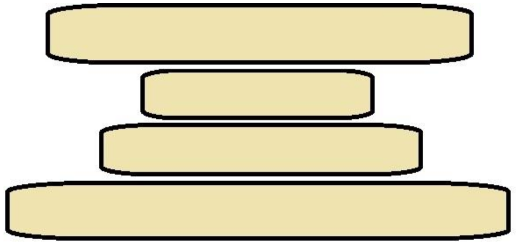


What is Search, Why do I care?

- Search is a technique for solving *hard* problems
- These problems look like this:
  - NP hard or worse
  - Domain specific solution doesn't yet exist



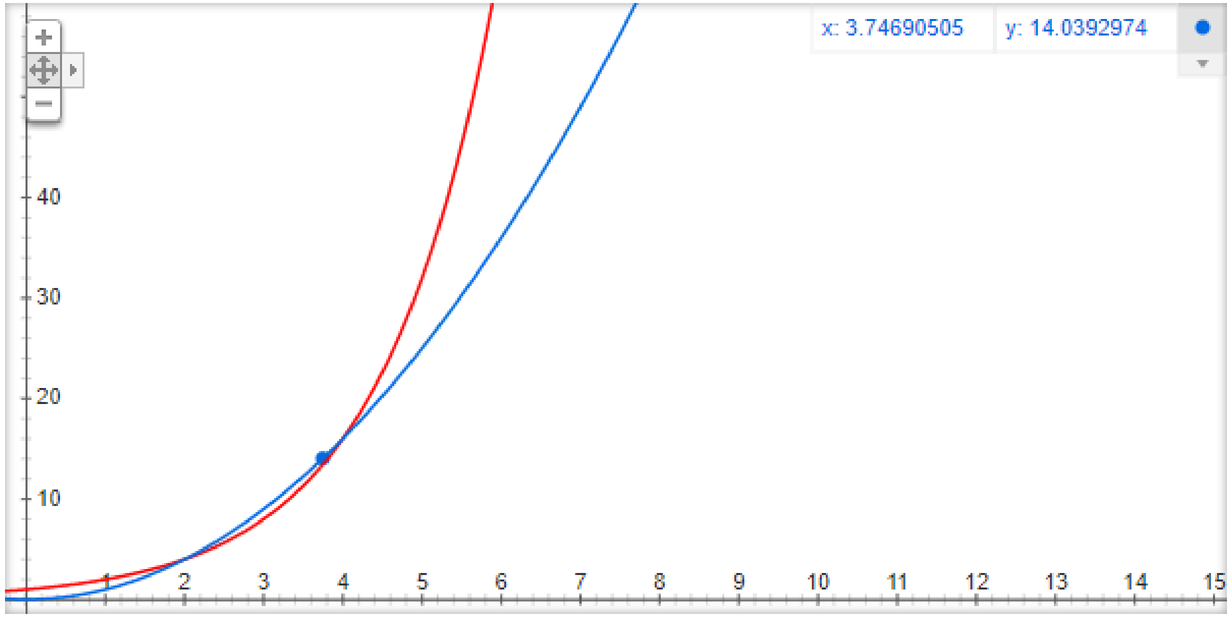




What's NP Hard? Why do I care?

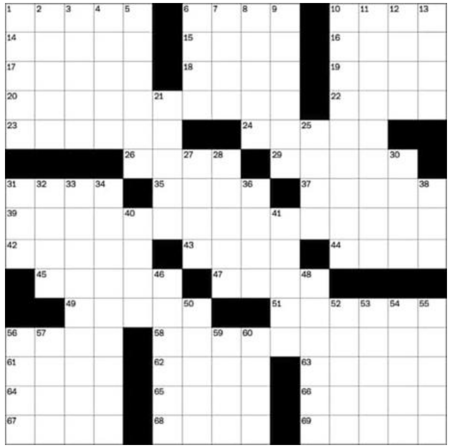
- Informally " … real expensive, and there are no known cheap solutions
- Formally-ish "… at least as hard as the hardest problems in NP"
  - A class of problems that have similar costs to solve
  - No known polynomial time algorithms for solving
    - Selection sort is poly-time for input size n n^2
    - Depth first search is not for solution depth n b^n

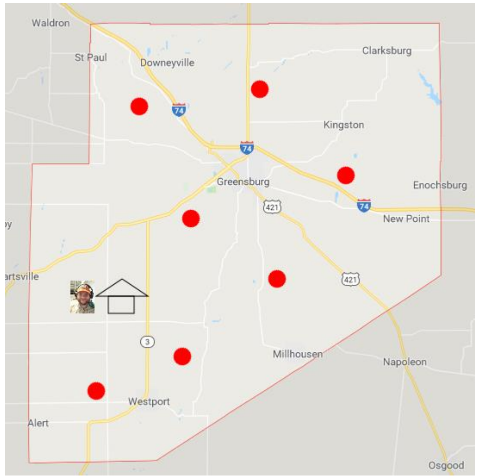
Graph for x^2, 2^x

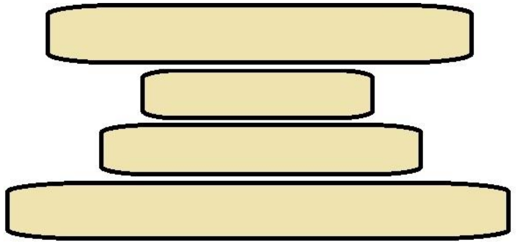


What is Search, Why do I care?

- Search is a technique for solving *intractable* problems
- These problems look like this:
  - NP hard or worse
  - Domain specific solution doesn't yet exist







Talk Outline

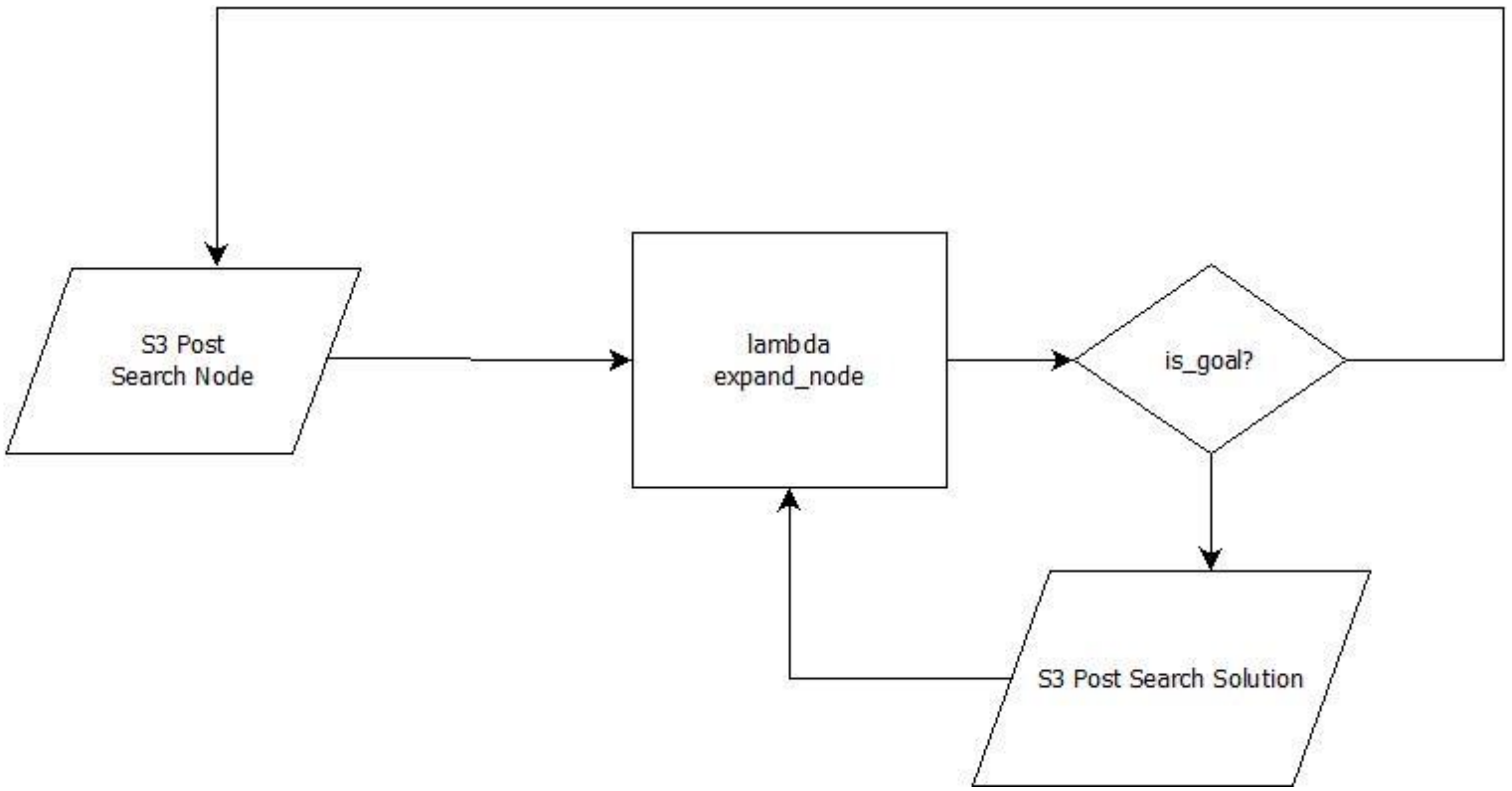
- What is heuristic search and why should I care?
    - Where are we going with this talk?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
    - Pancake Flipping – a toy domain
    - The Travelling Salesman Problem
- Distributed Depth First Search
- Distributed Depth First Search In The Cloud

Heuristic Search Can Be Costly

- Checkers, the extreme case
  - Constant computation from 1989 to 2007 involving around 200 processors
- VLSI & TSP, the hard case
  - Hours to days of compute time for moderate instances (2500-3000)
- Scheduling
  - Minutes to days depending on problem size, constrainedness

- Mercifully, CPU Time is not Wall Clock Time!

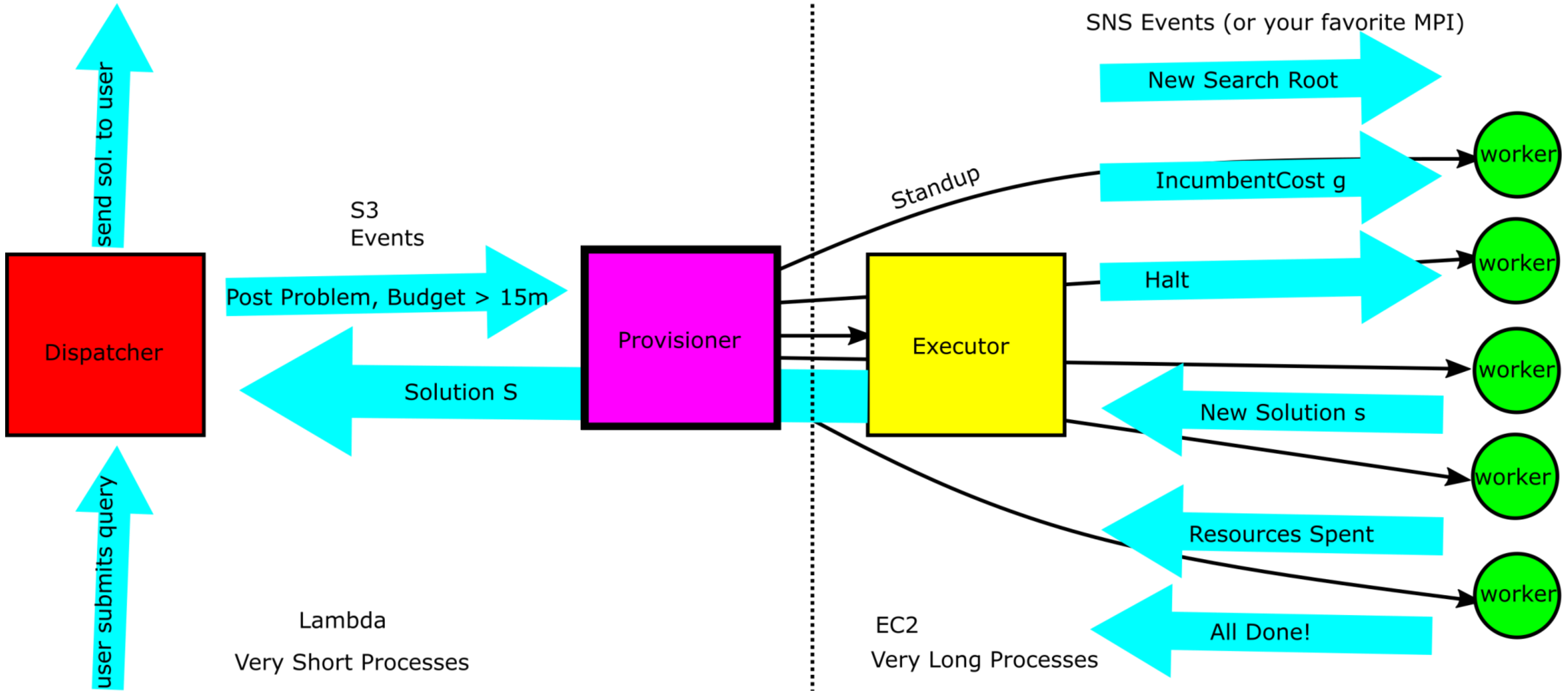
The Simplest Approach



Why you can't do that

- A problem of interest was a 115,000 city tsp
  - 115,000! Potential solutions
  - At the outside, maybe we prune 75% of those
  - Still ~ 1.5×10^{532039} nodes / expansions
- How much do 10^{532039} lambda calls cost?
  - First million are free, 20 cents per million after that.
  - So, about \$ 10^{532032}
- Current Worldwide GDP for 100,000 years is $\sim\$10^{17}$

What you can do



Talk Outline

- What is heuristic search and why should I care?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
  - Pancake Flipping – a toy domain
  - The Travelling Salesman Problem
- Distributed Depth First Search
- Distributed Depth First Search in the cloud

Depth First Search from AI:AMA

```
def depth_first_tree_search(problem):
    """Search the deepest nodes in the search tree first.
        Search through the successors of a problem to find a goal.
        The argument frontier should be an empty queue.
        Repeats infinitely in case of loops. [Figure 3.7]"""

    frontier = [Node(problem.initial)]  # Stack

    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

Depth First Search from AI:AMA

```
def depth_first_tree_search(problem):
    """Search the deepest nodes in the search tree first.
        Search through the successors of a problem to find a goal.
        The argument frontier should be an empty queue.
        Repeats infinitely in case of loops. [Figure 3.7]"""

    frontier = [Node(problem.initial)]  # Stack

    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
```

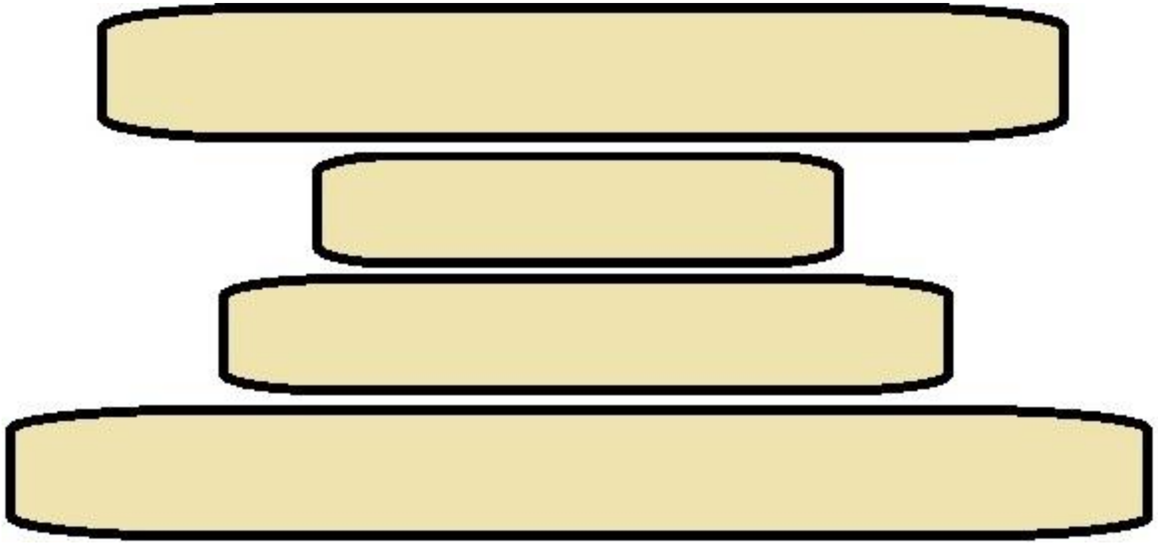
Depth First Search

```
def depth_first_tree_search(problem):

    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
            continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        frontier.extend(node.expand(problem))
    return solution
```

Talk Outline

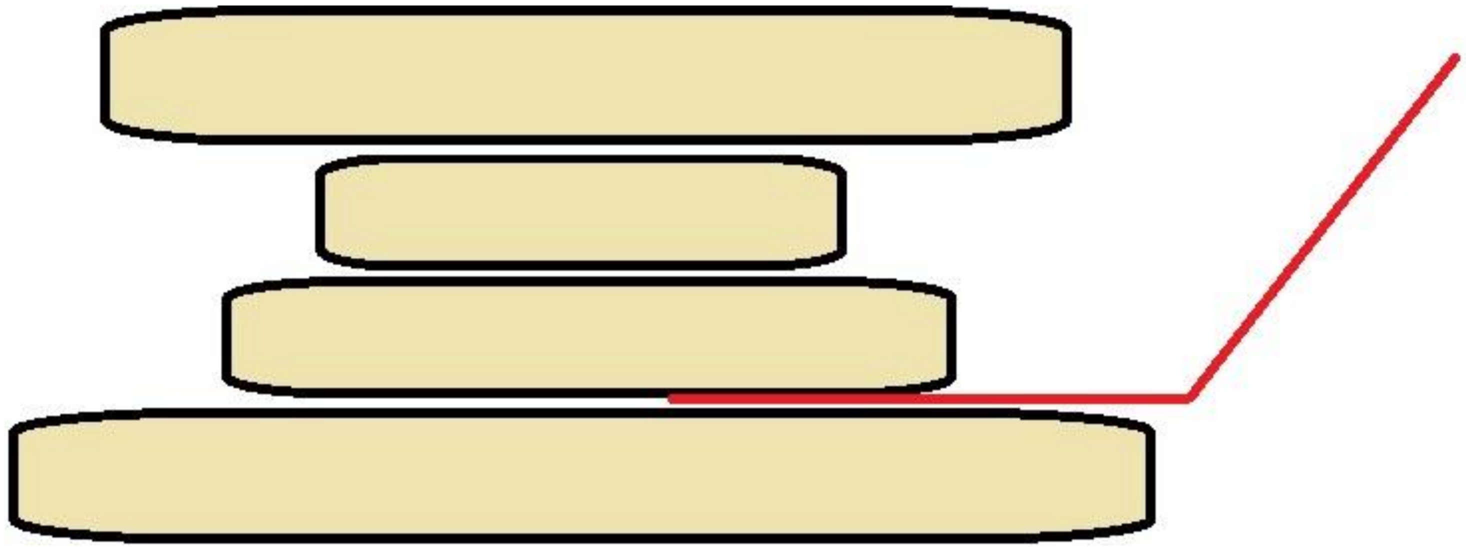
- What is heuristic search and why should I care?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
    - Pancake Flipping – a toy domain
    - The Travelling Salesman Problem
- Distributed DFS
- Distributed DFS in the cloud

The Pancake Domain

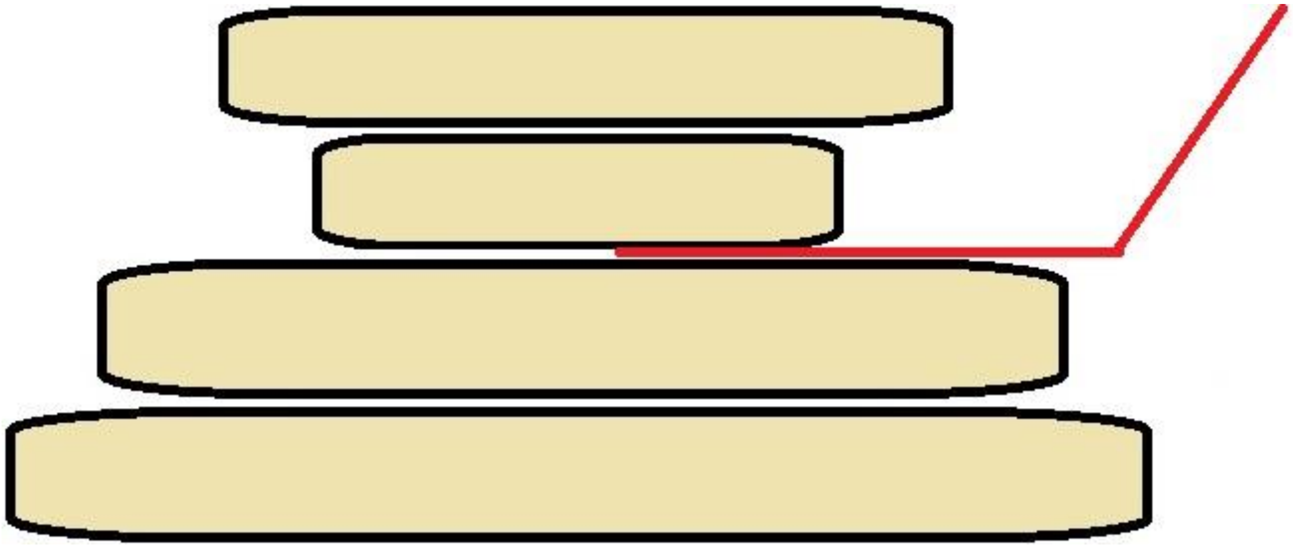


Given an unordered stack of pancakes,
Order them using only a spatula and the ability to flip the stack

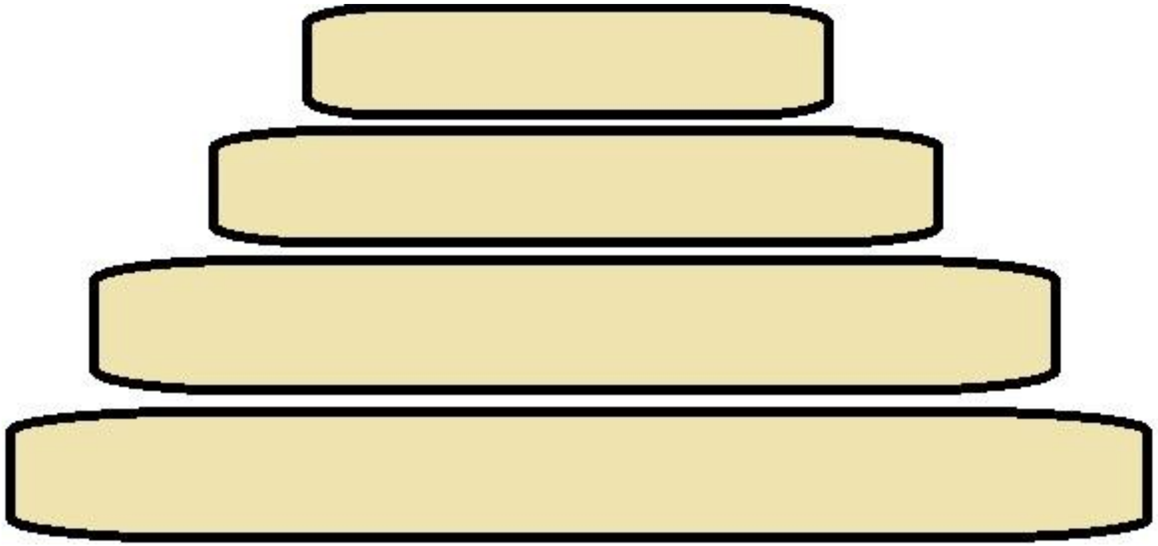
Step 1



Step 2

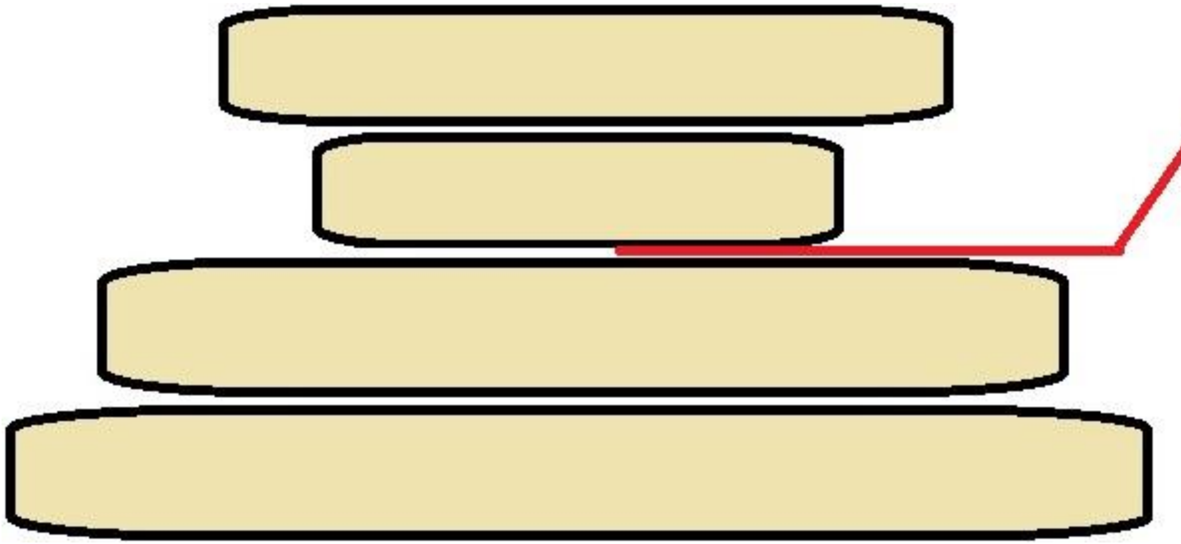


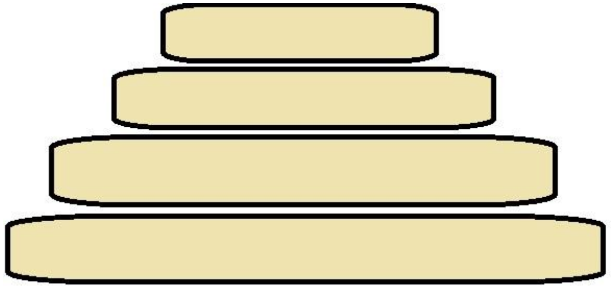
Step 3



What is Search, Why do I care?

- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics





What is Search, Why do I care?

- Search is a technique for solving *hard* problems
- These problems look like this:
  - NP hard or worse
  - Domain specific solution doesn't yet exist
- Solving the pancake problem optimally is equivalent to known NP hard problems
  - Rubik's Cube Optimally
  - 15 Puzzle Optimally
- There's a new-ish reduction from 3-SAT
  - Pancake Flipping is Hard by Bulteau, Fertin, and Rusu
  - https://arxiv.org/pdf/1111.0434.pdf

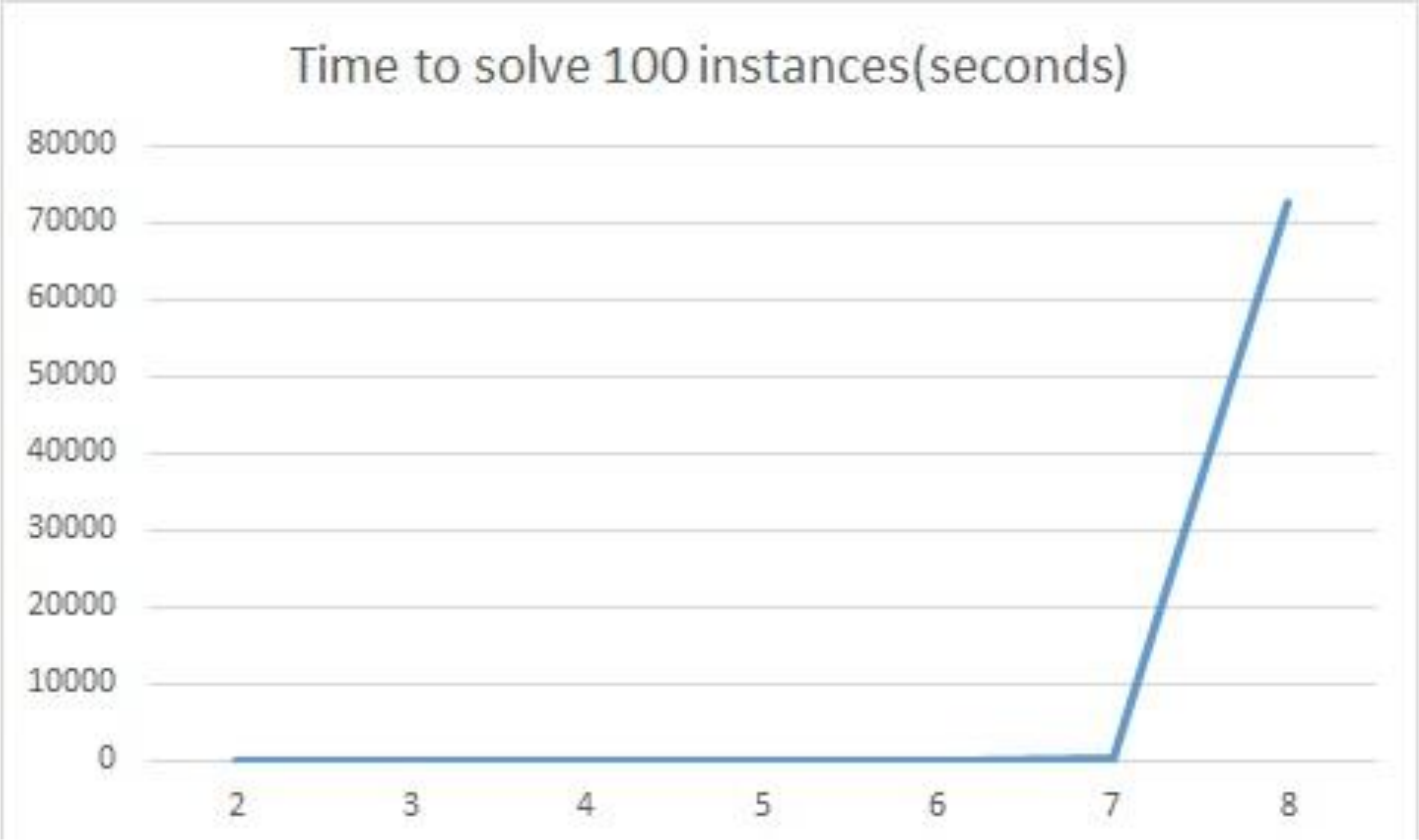
Depth First Search for Pancakes

```
def depth_first_tree_search(problem):

    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
              continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        frontier.extend(node.expand(problem))
    return solution
```

It can (only) solve small instances

```
--------------------------------------------------------------------
Problem: {numCakes = 4;
 initState = [4; 2; 3; 1];}
{generated = 0;
 expanded = 0;
 goalsFound = 0;
 duplicatesFound = 0;
 pruned = 0;}
[[4; 2; 3; 1]; [2; 4; 3; 1]; [3; 4; 2; 1]; [4; 3; 2; 1]; [1; 2; 3; 4]]
The target process exited without raising a CoreCLR started event. Ensure that
The program '[8360] treeSearch.exe' has exited with code 0 (0x0).
```

But how does it scale? (Real Bad)

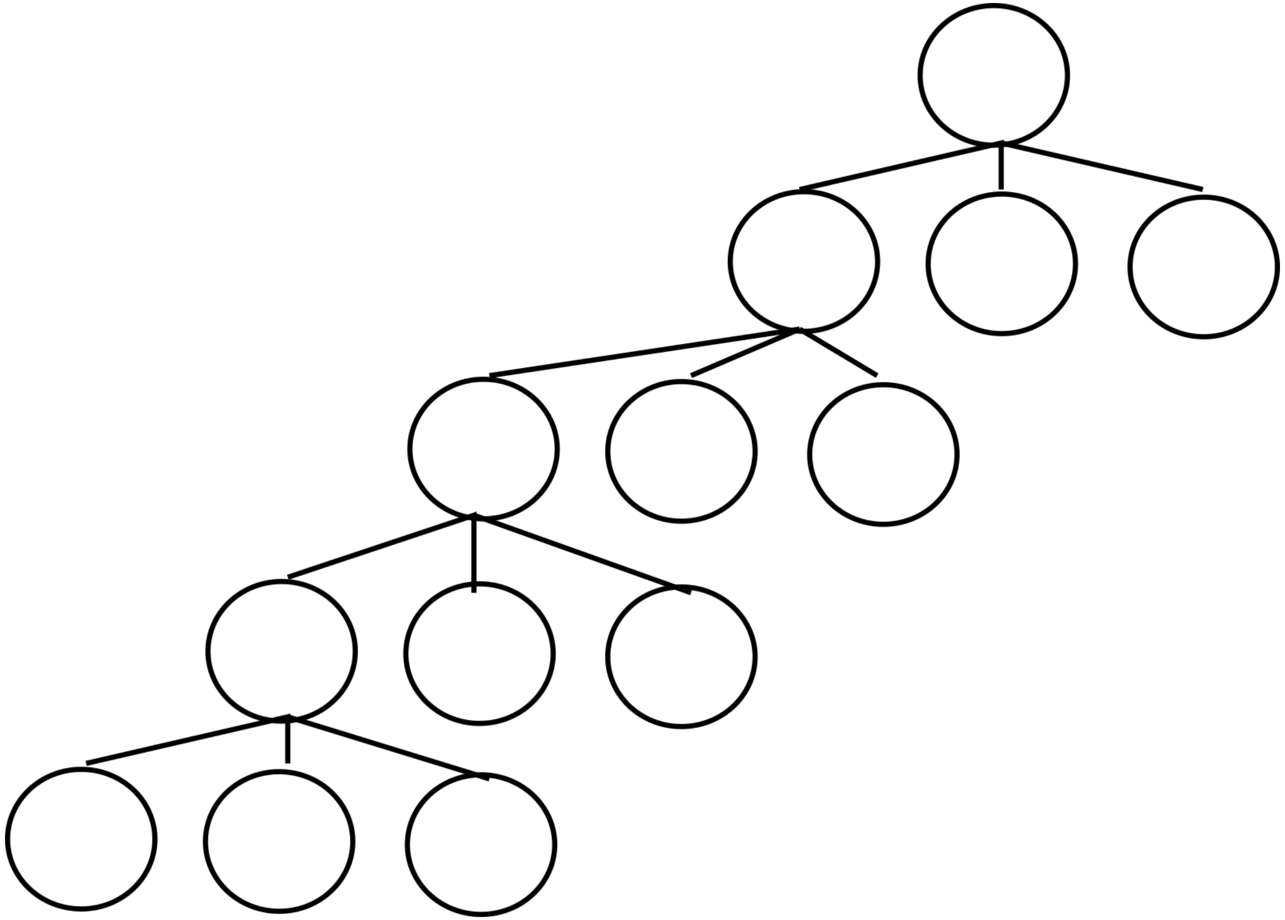


Why?

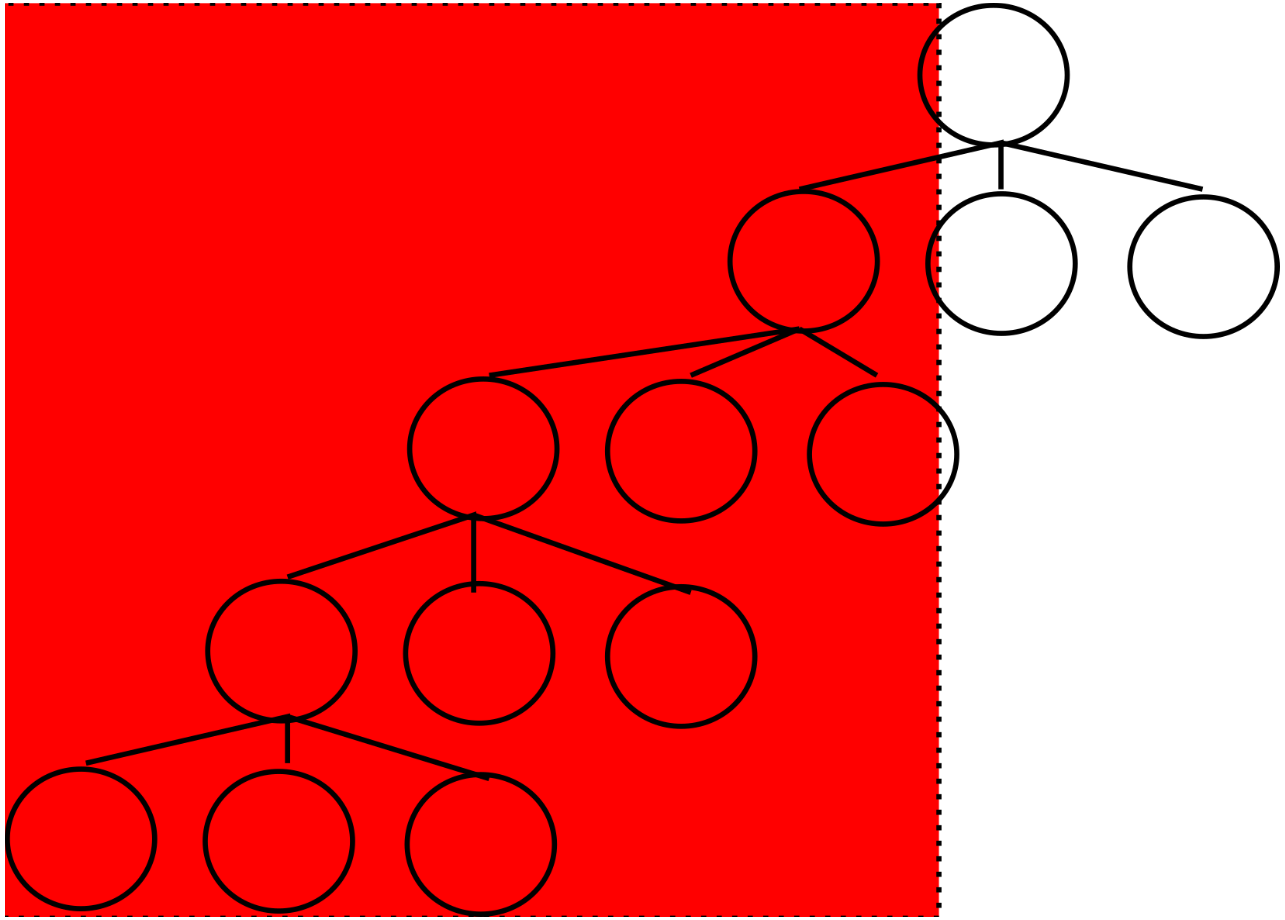
```
def depth_first_tree_search(problem):

    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
            continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        frontier.extend(node.expand(problem))
    return solution
```

Children Are Unsorted!

Child Ordering is Critical



Child Ordering is Critical



Depth First Search: Child Ordering

```
def depth_first_tree_search(problem):
    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
              continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        children = node.expand(problem)
        children.sort()
        frontier.extend(children)
    return solution
```

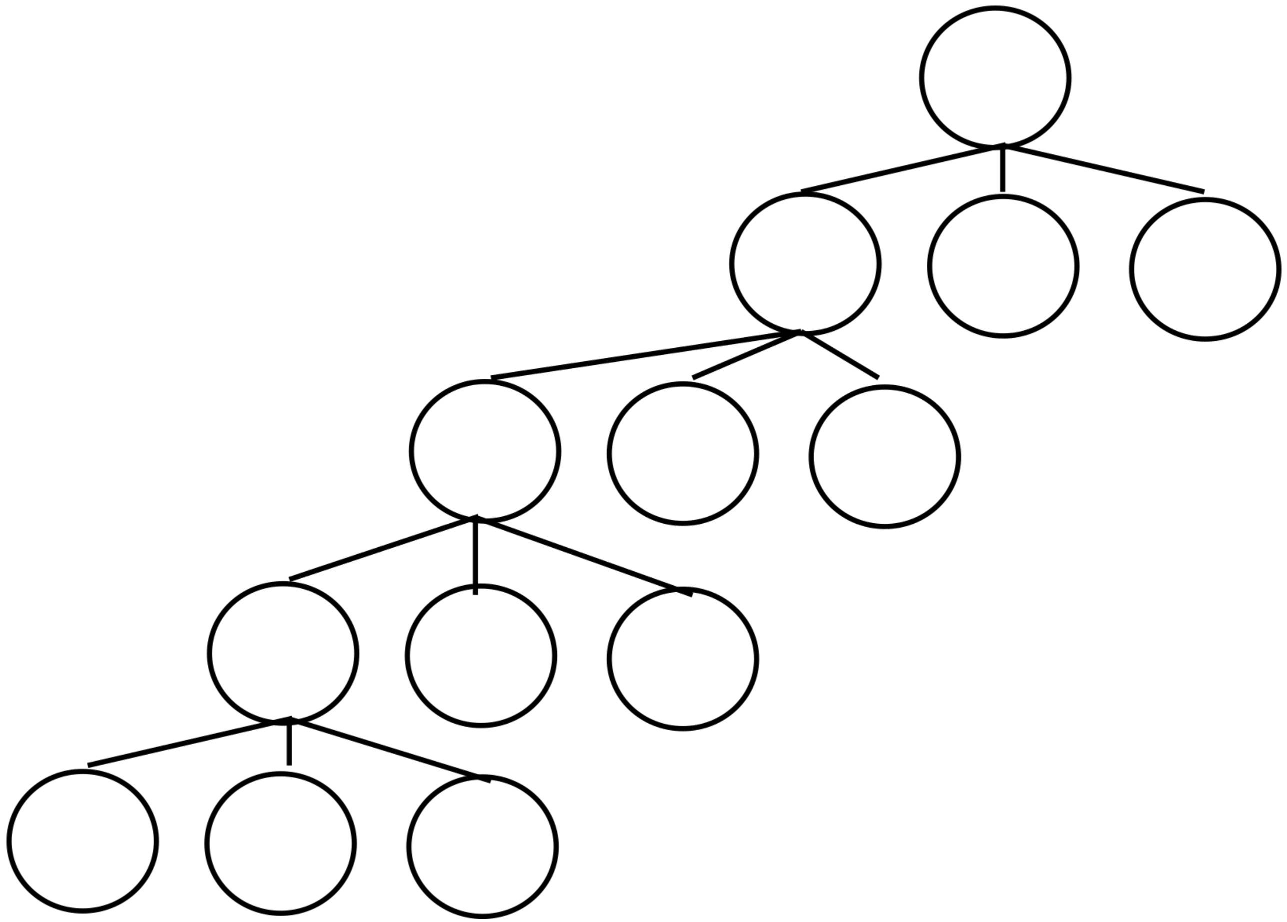
Children are sorted (Heuristics go here!)

Depth First Search: Child Ordering

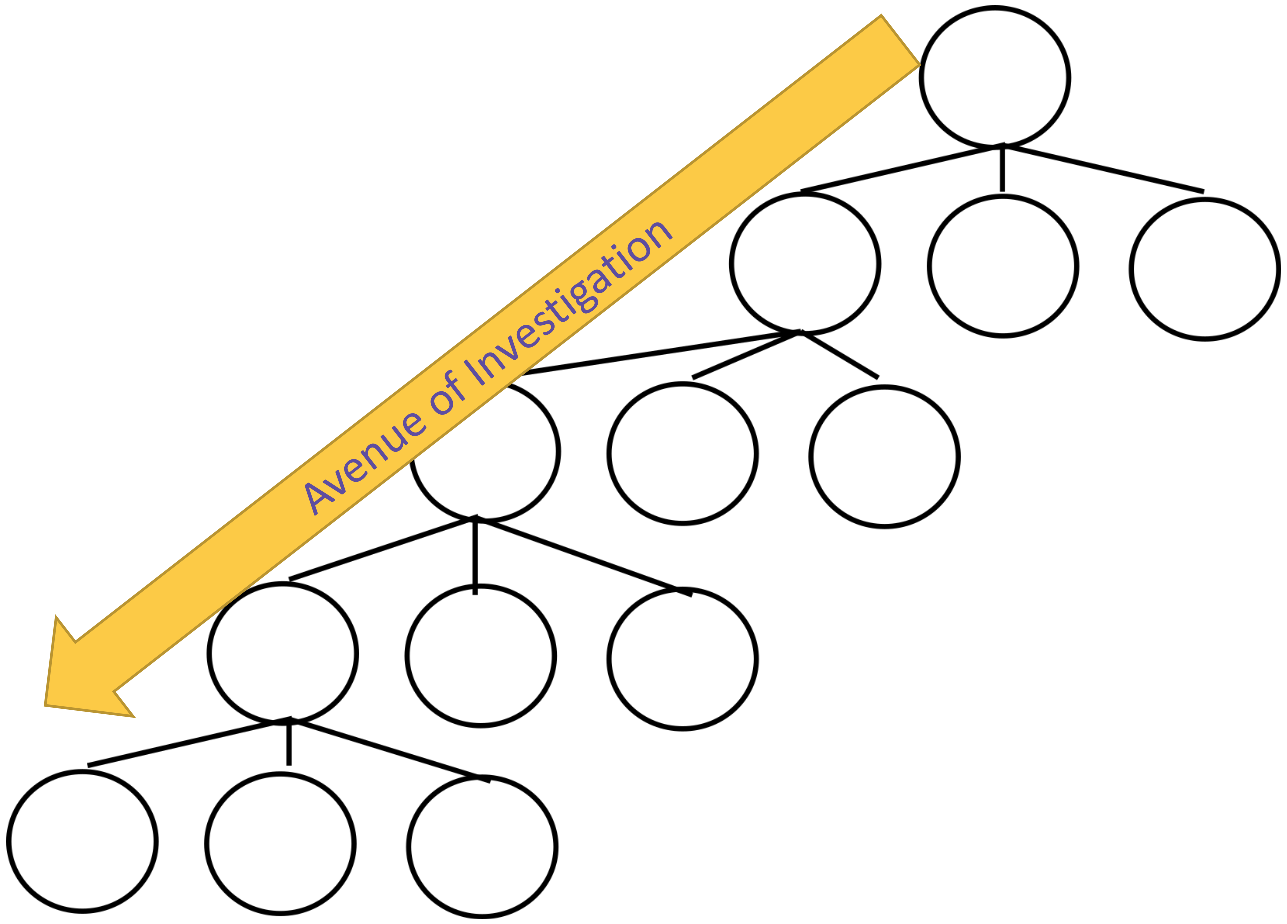
```
def depth_first_tree_search(problem):
    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
              continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        children = node.expand(problem)
        children.sort()
        frontier.extend(children)
    return solution
```

Children are all generated at once

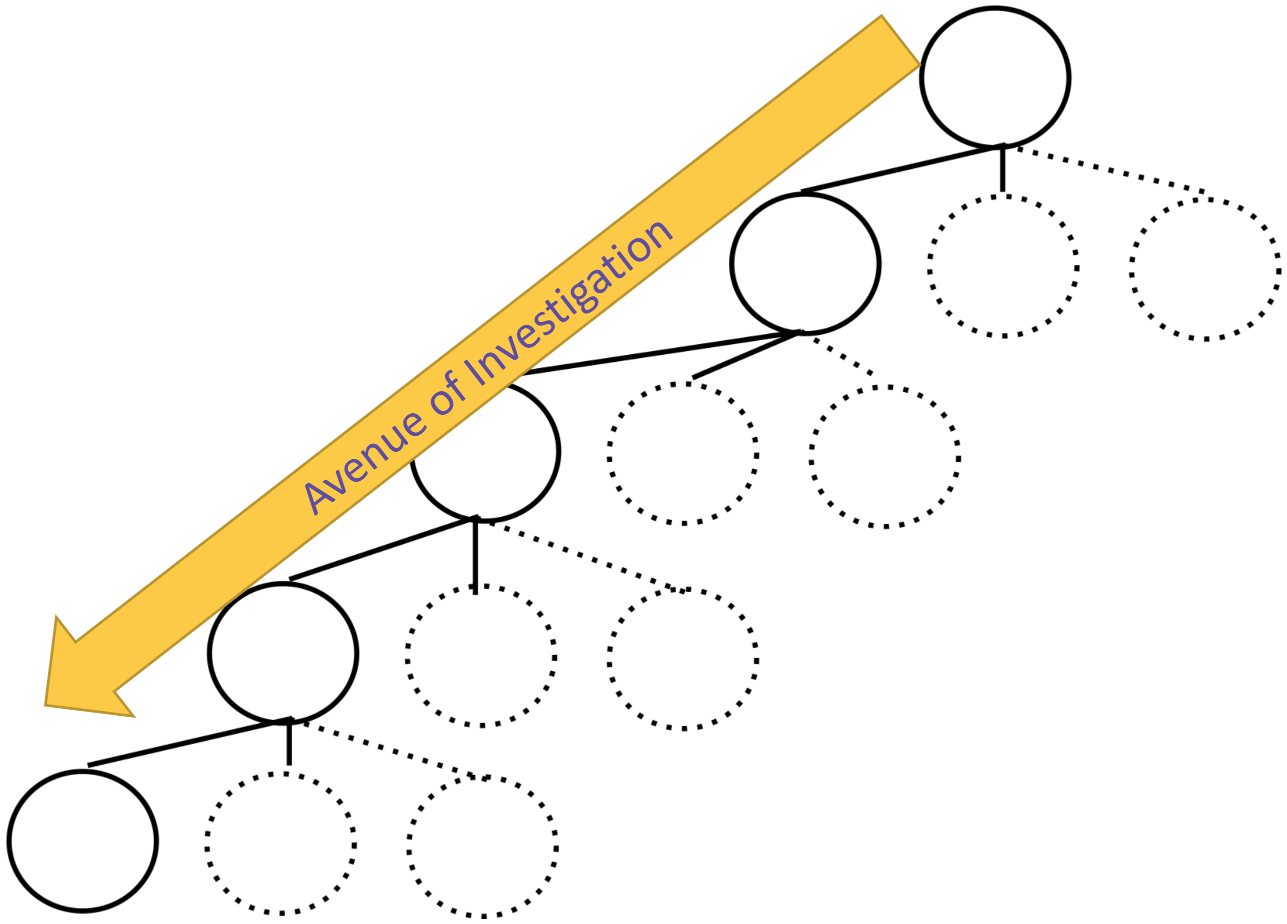
Making All Kids At Once Is Bad!



Making All Kids At Once Is Bad!



Making All Kids At Once Is Bad!

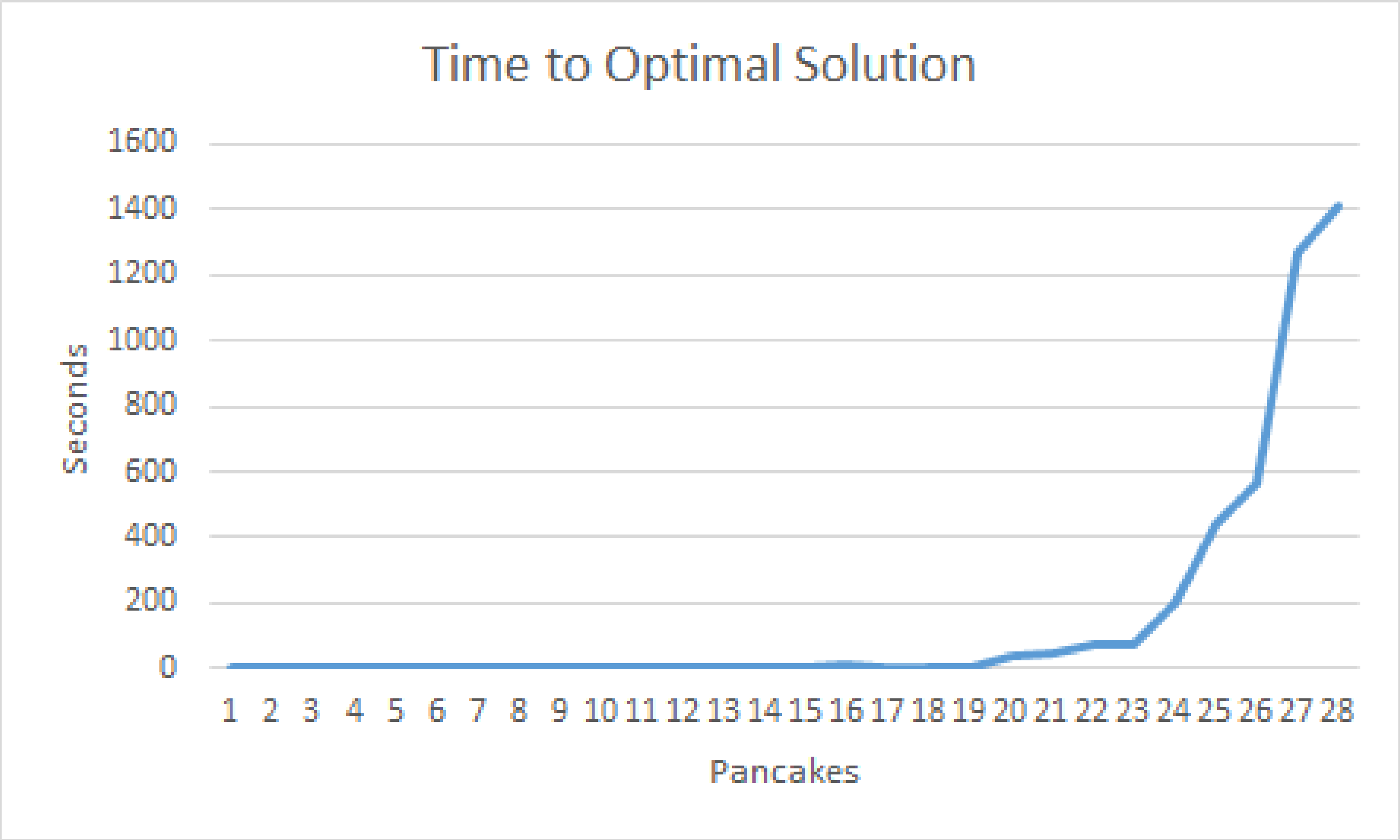


Depth First Search: Child Ordering

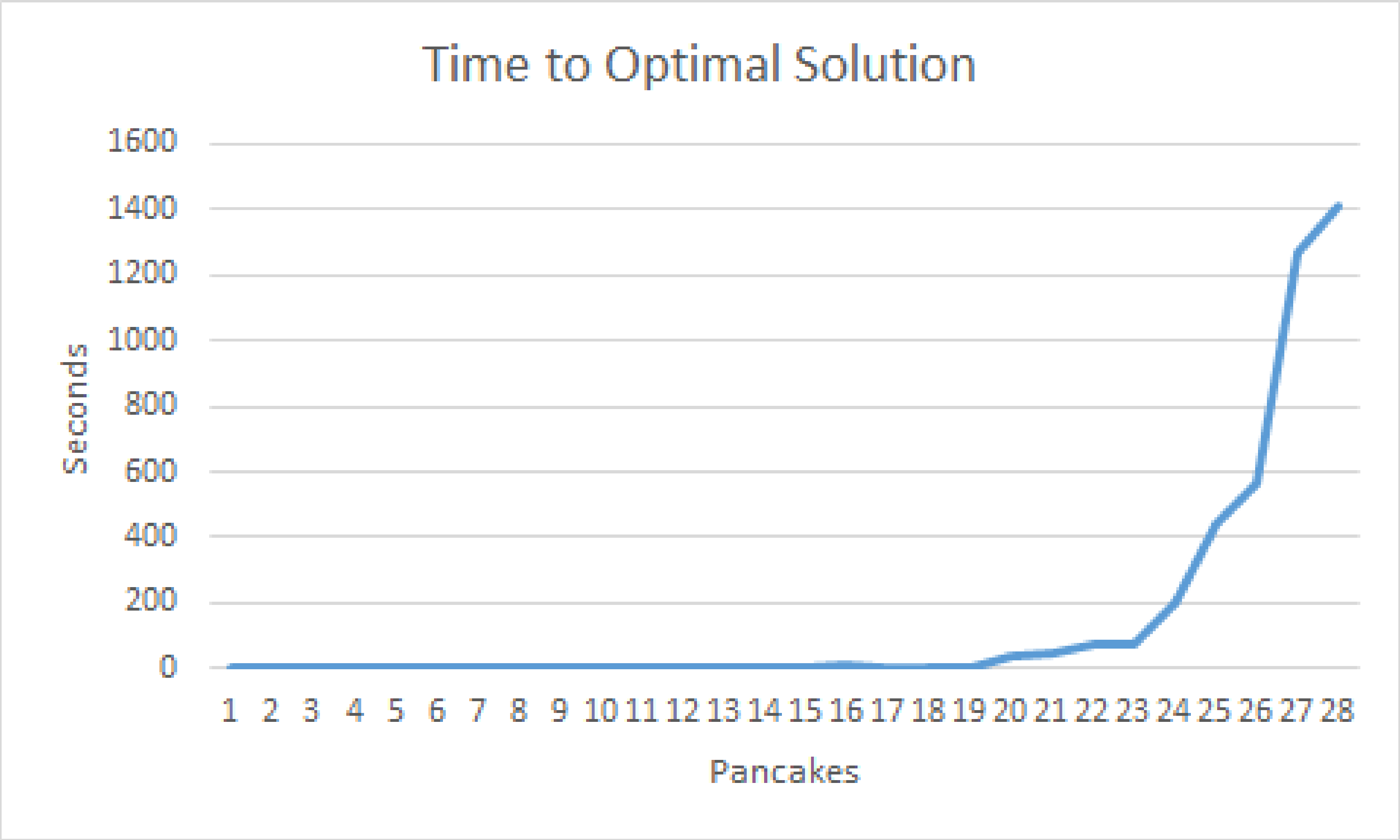
```
def depth_first_tree_search(problem):
    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
              continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        next = node.get_next_child(problem)  # child ordering is now baked into next_child
        if not next is None:
           frontier.extend([next, node])
    return solution
```

One child at a time

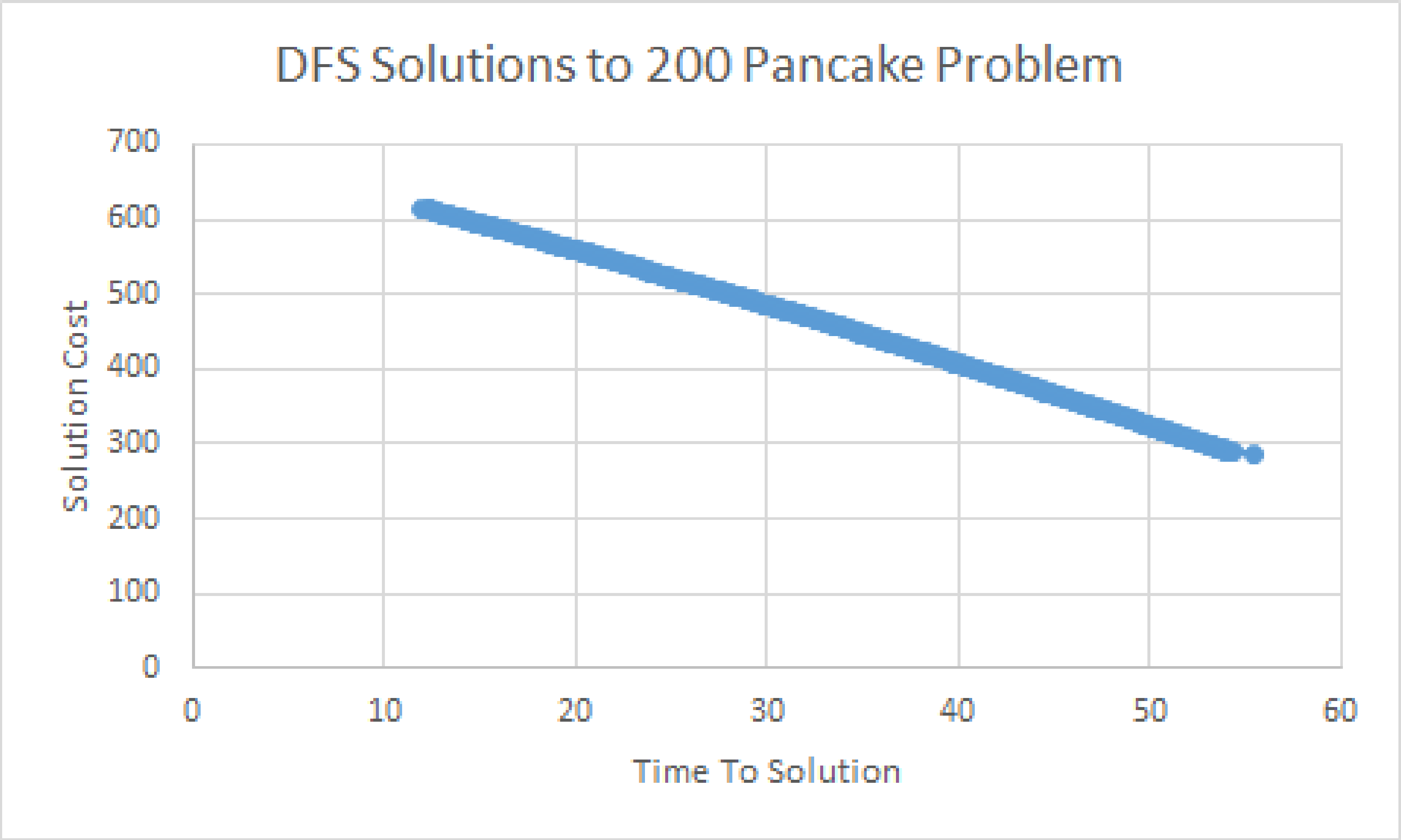
How’s It Perform Now?



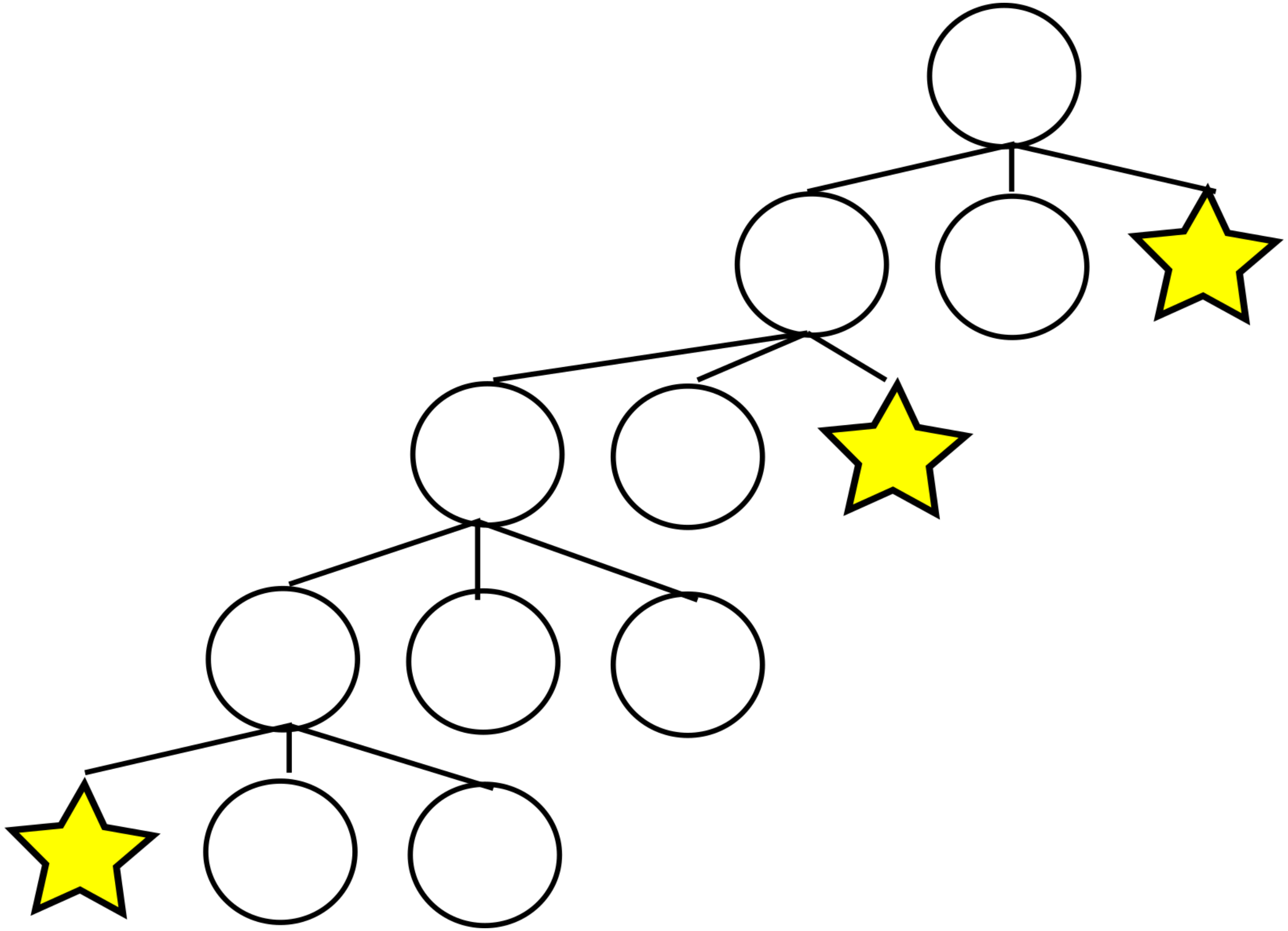
Actually, the performance is complicated…



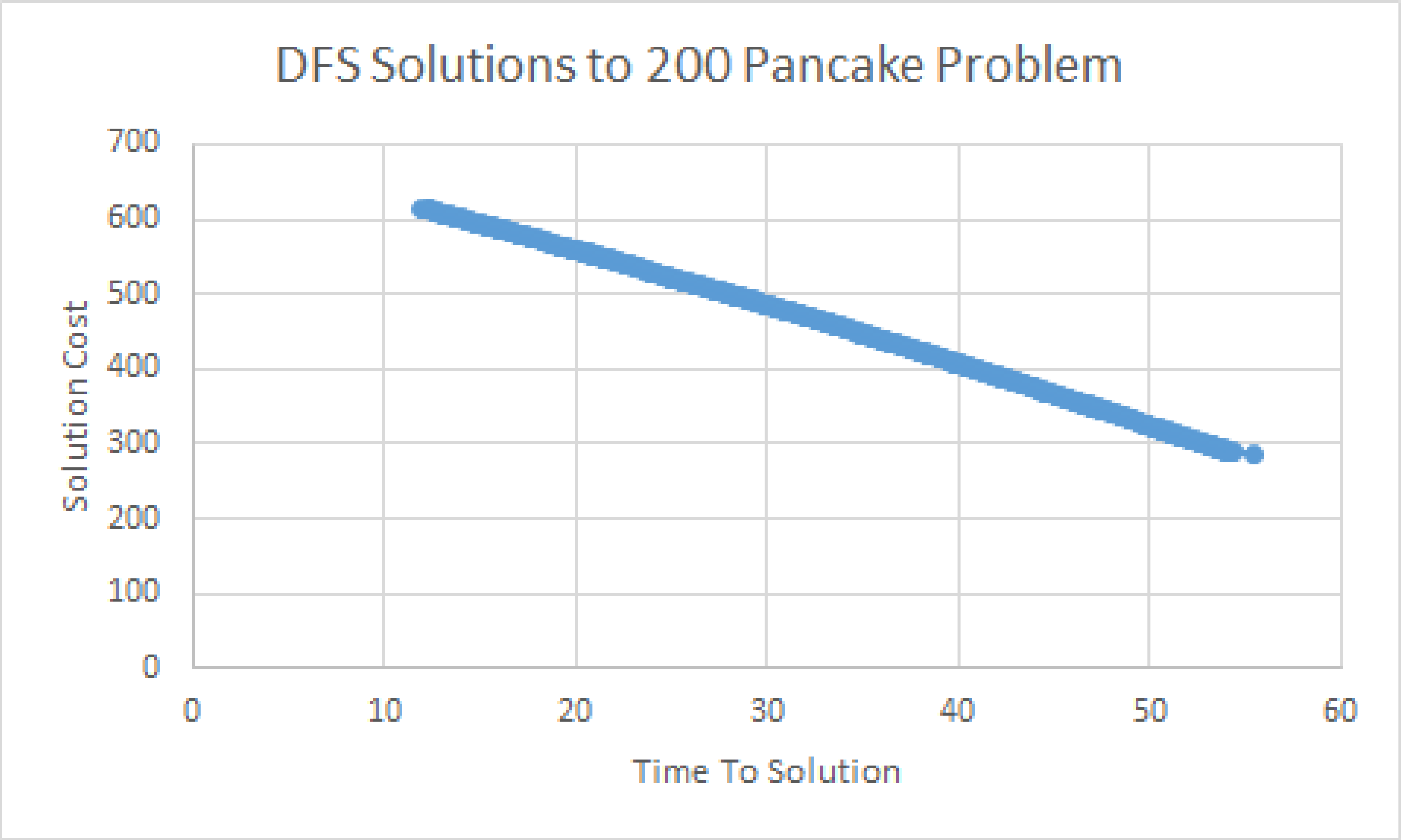
Actually, the performance is complicated…



DFS is an Anytime Search



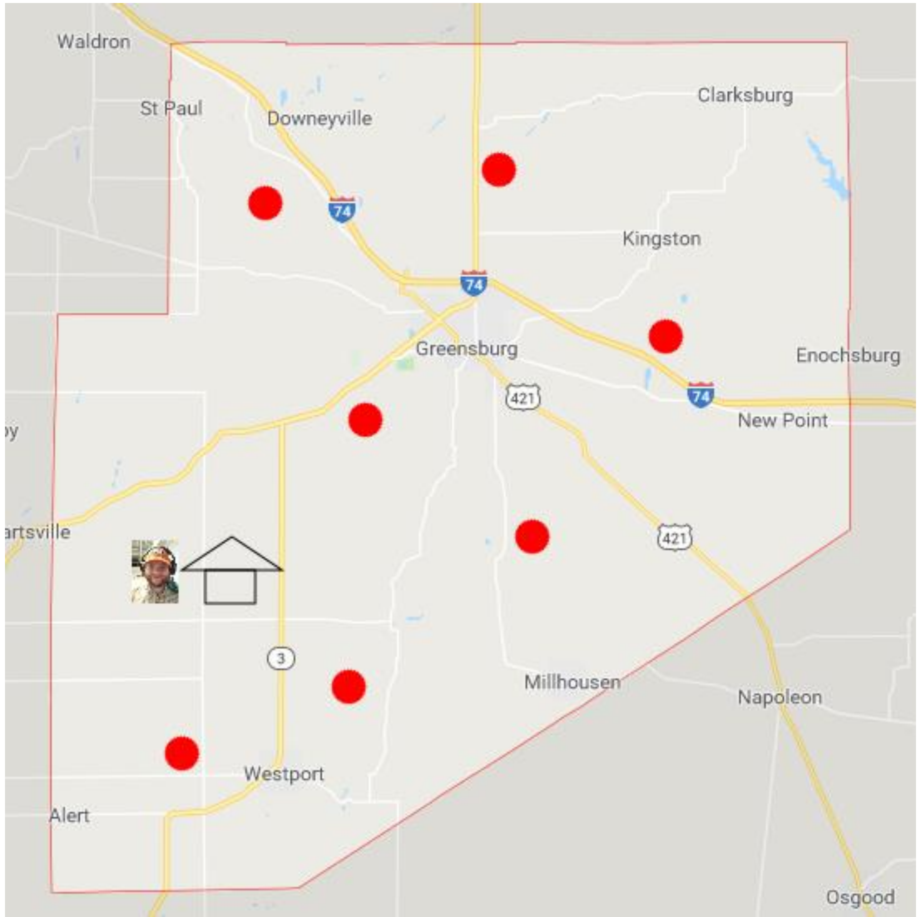
Actually, the performance is Complicated…



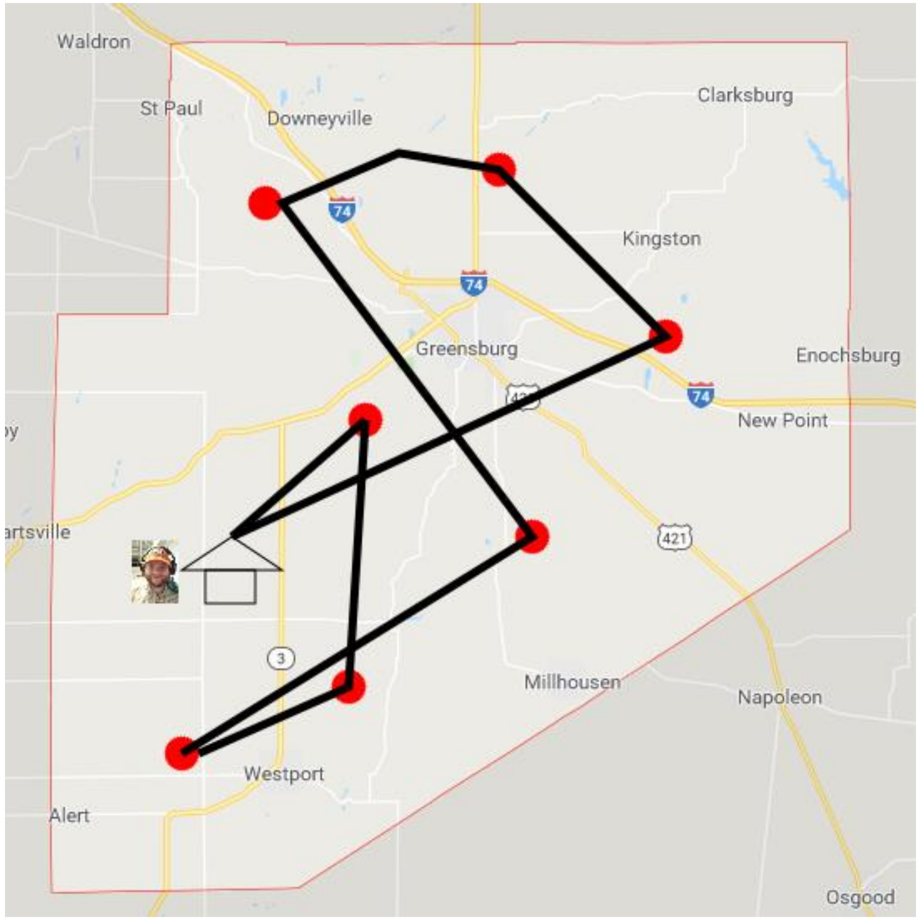
Talk Outline

- What is heuristic search and why should I care?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
  - Pancake Flipping – a toy domain
  - The Travelling Salesman Problem
- Distributed Depth First Search
- Distributed DFS in the Cloud

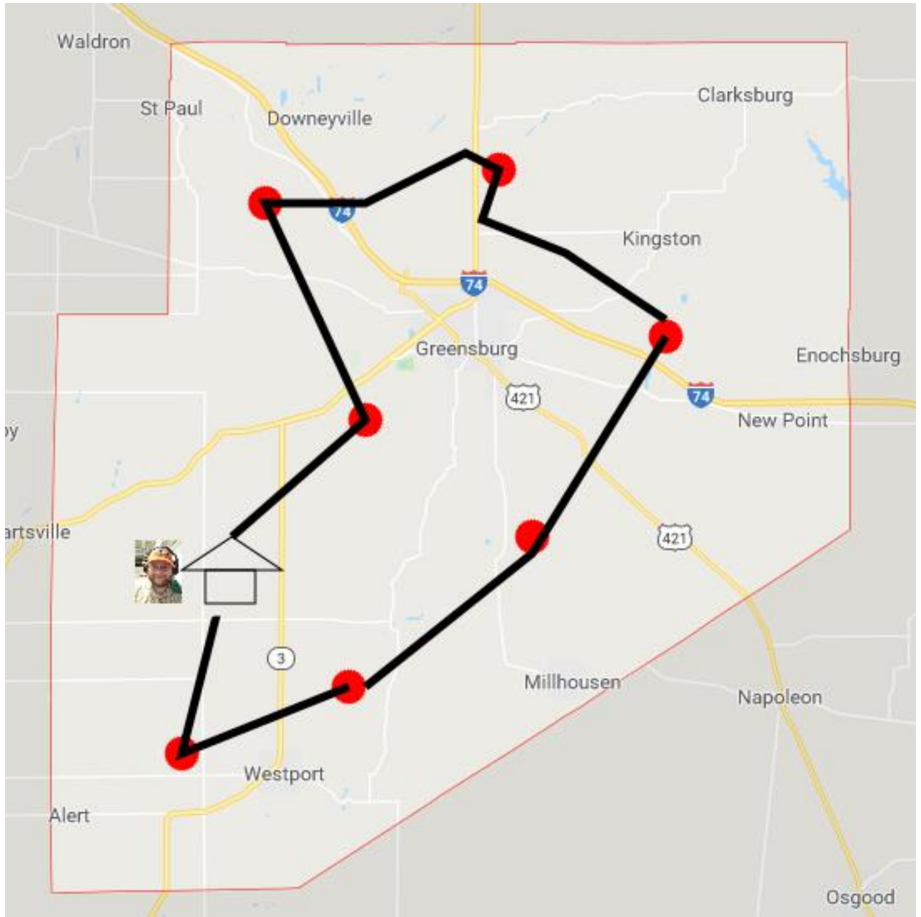
Travelling Salesman Problem



Travelling Salesman Problem



Travelling Salesman Problem



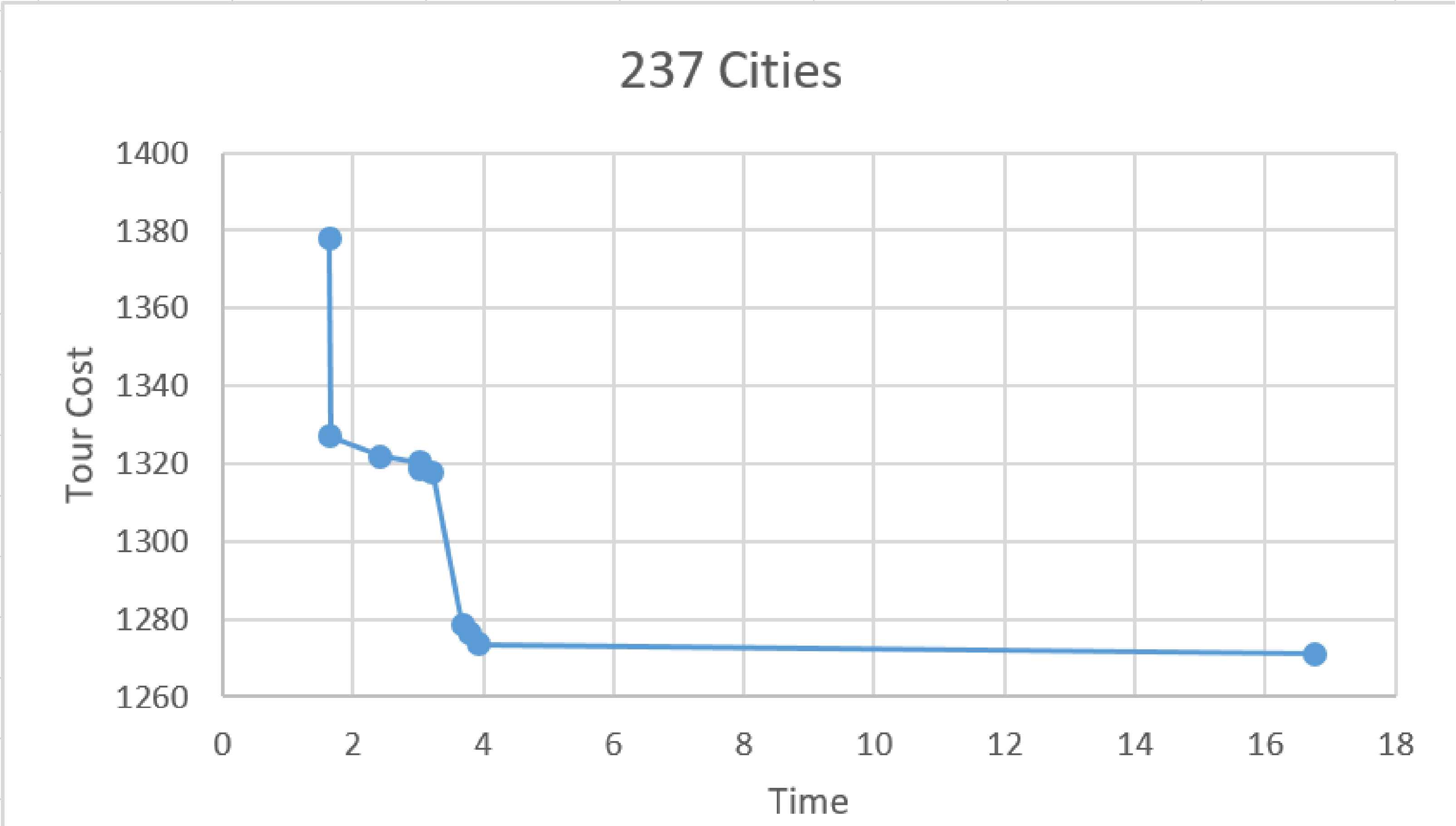
This is what makes heuristic search so cool:

I can solve a new problem,
But I don’t have to change my approach!

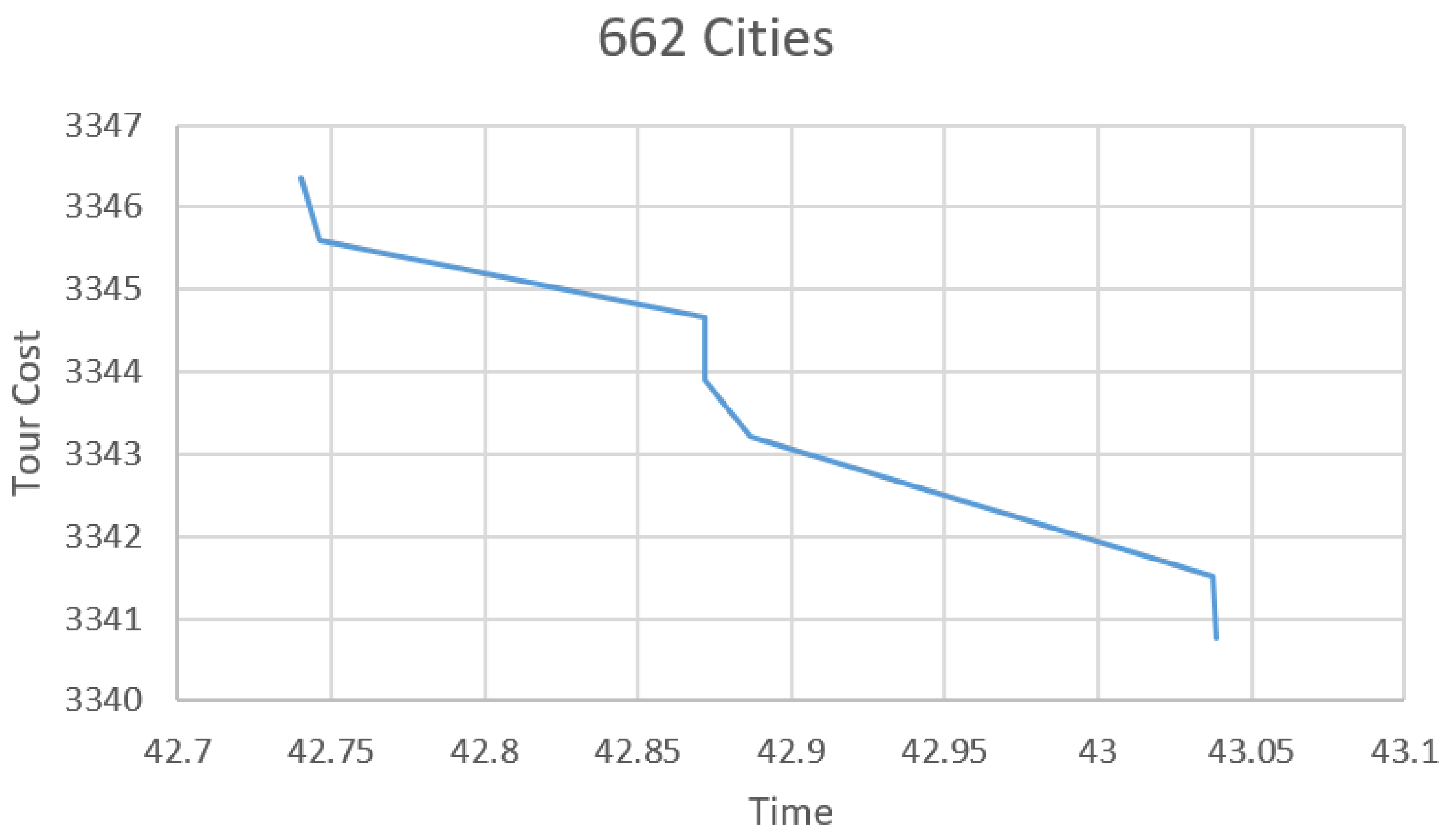
```
type TreeSearchInterface(problem) =
    interface TreeSearch.TreeSearch<State, float> with
        member this.InitialState = deterministicInit problem // State
        member this.Goal state = goalTest problem state // State -> bool
        member this.H state = euclidH problem state  // State -> float
        member this.D state = d problem state // State -> int
        member this.Equal s1 s2 = equal s1 s2 // State -> State -> bool
        member this.NumChildren state = numChildren problem state // State -> int
        member this.NthChild state n = nthChild problem state n // State -> int -> float * Stat
        member this.InitialCost = 0. // float
        member this.ChildOrder = None // (float * State -> float * State -> int) option

type InPlaceModificationTreeSearch(problem) =···

let solve problem = // Problem -> unit
    let iface = TreeSearchInterface(problem) in
    let solNode, metrics = DFSv2.floatCycles iface in
    //let solNode, metrics = ILDS.ildsCycles iface in
        printfn "%A" metrics;
        match solNode with
         | None -> printfn "No solution"
         | Some s -> ImperativeTreeSearch.getSolution s |> printfn "%A"
```

```
type TreeSearchInterface(problem) =
    interface TreeSearch.TreeSearch<State, int> with
        member this.InitialState = problem.initState // State
        member this.Goal state = goalTest state // State -> bool
        member this.H state = gapHeuristic state  // State -> int
        member this.D state = gapHeuristic state // State -> int
        member this.Equal s1 s2 = s1 = s2 // State -> State -> bool
        member this.NumChildren _ = problem.numCakes - 1 // State -> int
        member this.NthChild state n = flipStack state (n + 2) // State -> int -> int * State
        member this.InitialCost = 0 // int
        member this.ChildOrder = None // (int * State -> int * State -> int) option

let solveNaive problem = // Problem -> unit

let solve problem = // Problem -> unit
    let iface = TreeSearchInterface(problem) in
    let solNode, metrics = DFSv2.intCycles iface in
    //let solNode, metrics = ILDS.ildsCycles iface in
        printfn "%A" metrics;
        match solNode with
         | None -> printfn "No solution"
         | Some s -> ImperativeTreeSearch.getSolution s |> printfn "%A"
```

TSP Anytime Performance



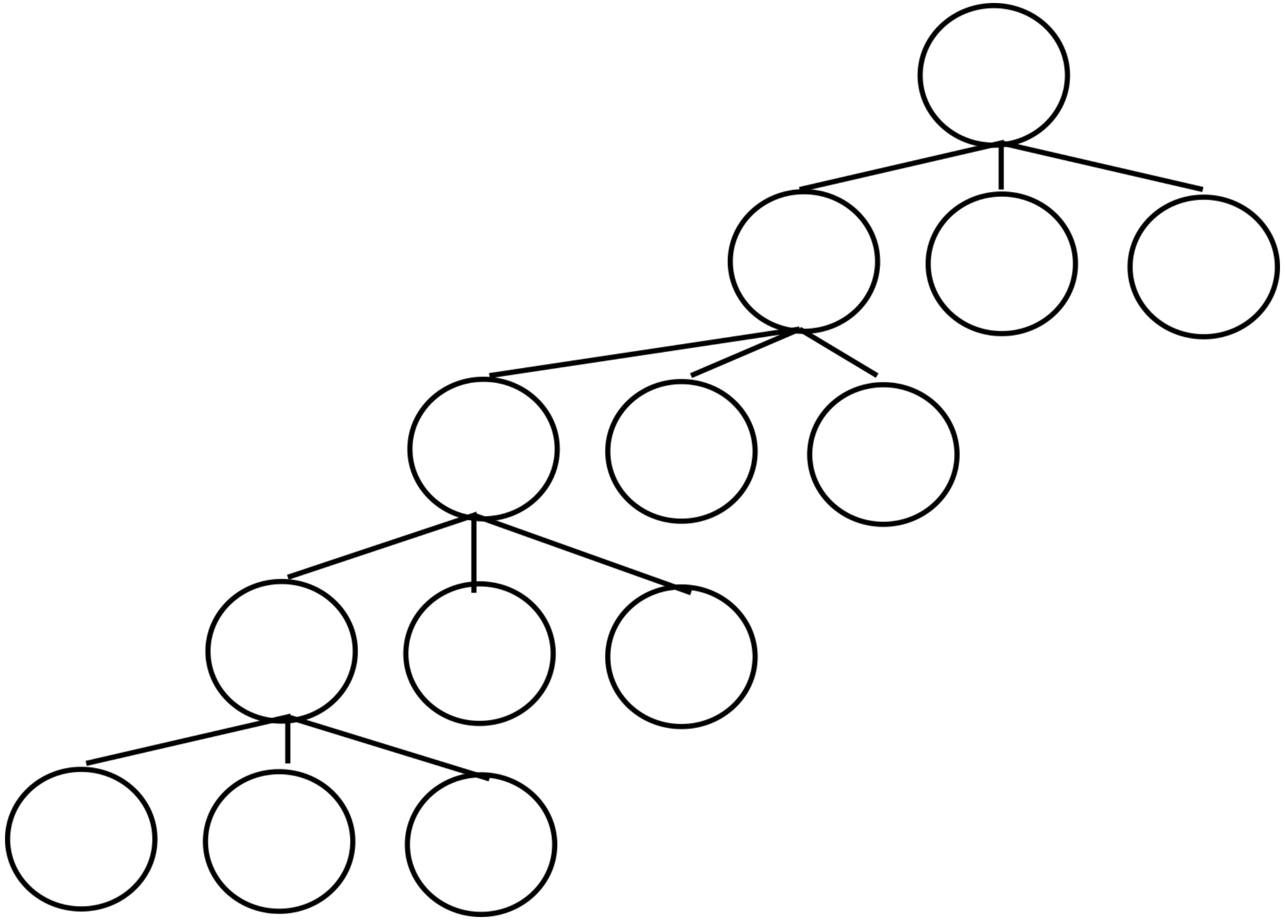
TSP Anytime Performance



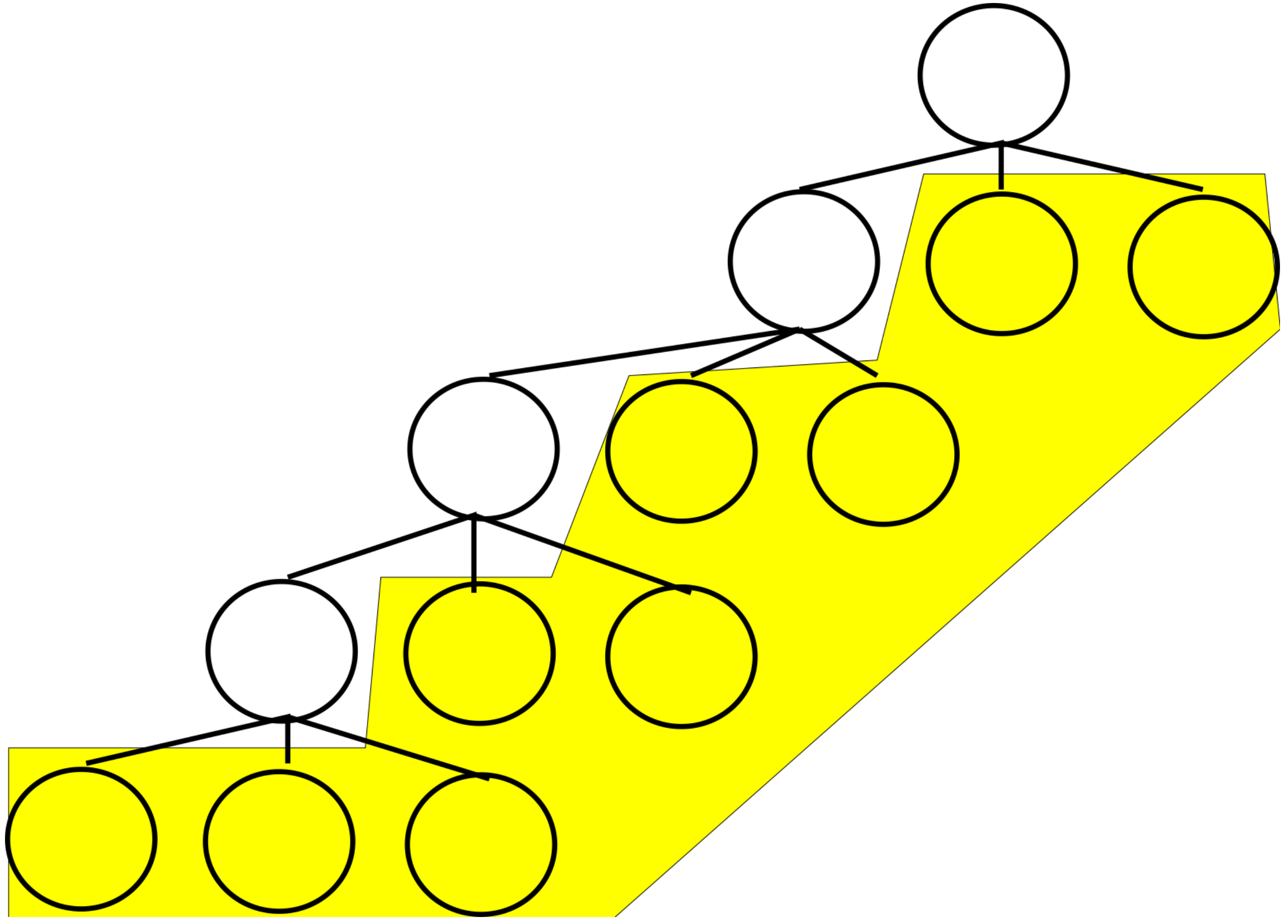
Talk Outline

- What is heuristic search and why should I care?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
  - Pancake Flipping – a toy domain
  - The Travelling Salesman Problem
- Distributed Depth First Search
- Distributed DFS in the cloud

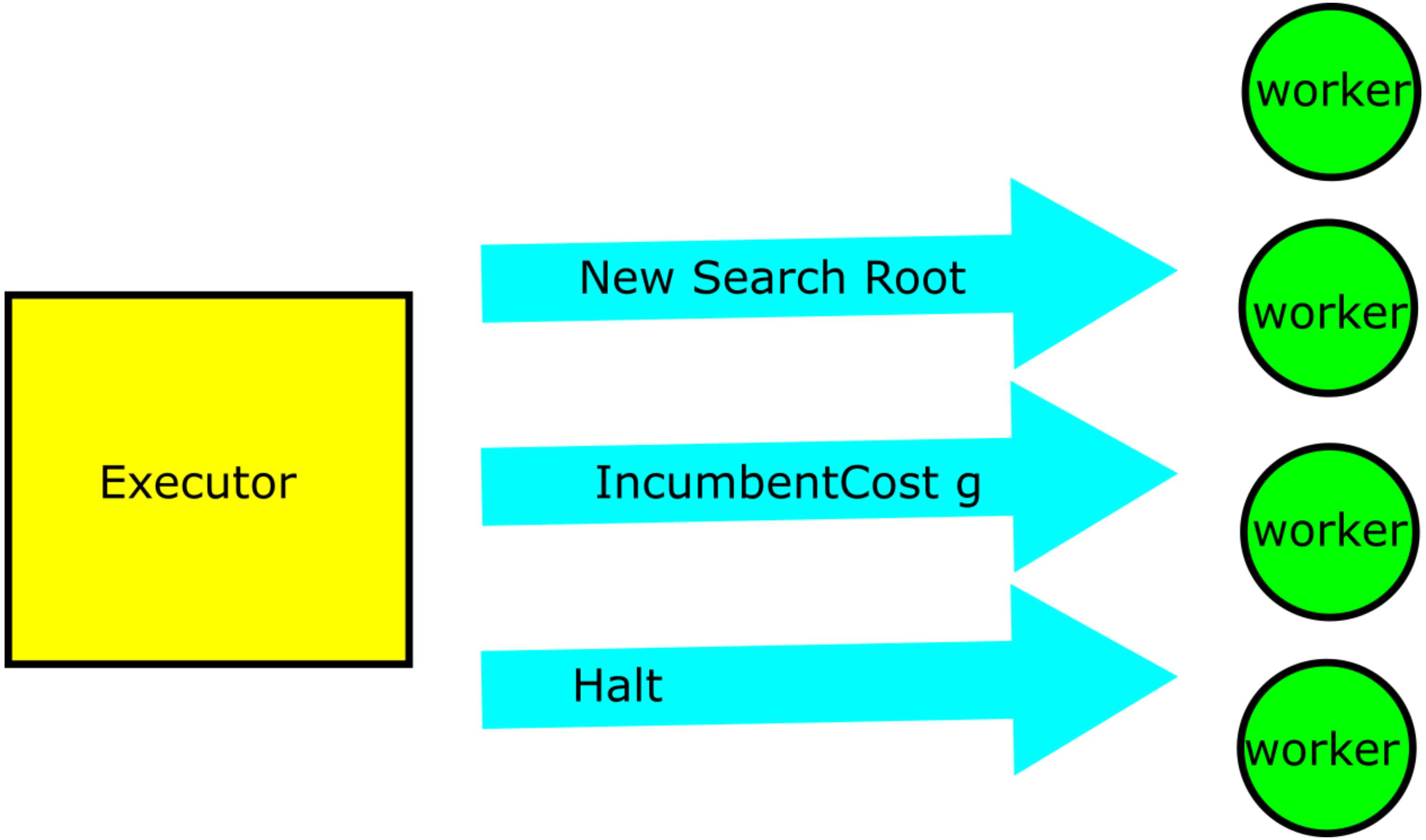
Distributed Depth First Search



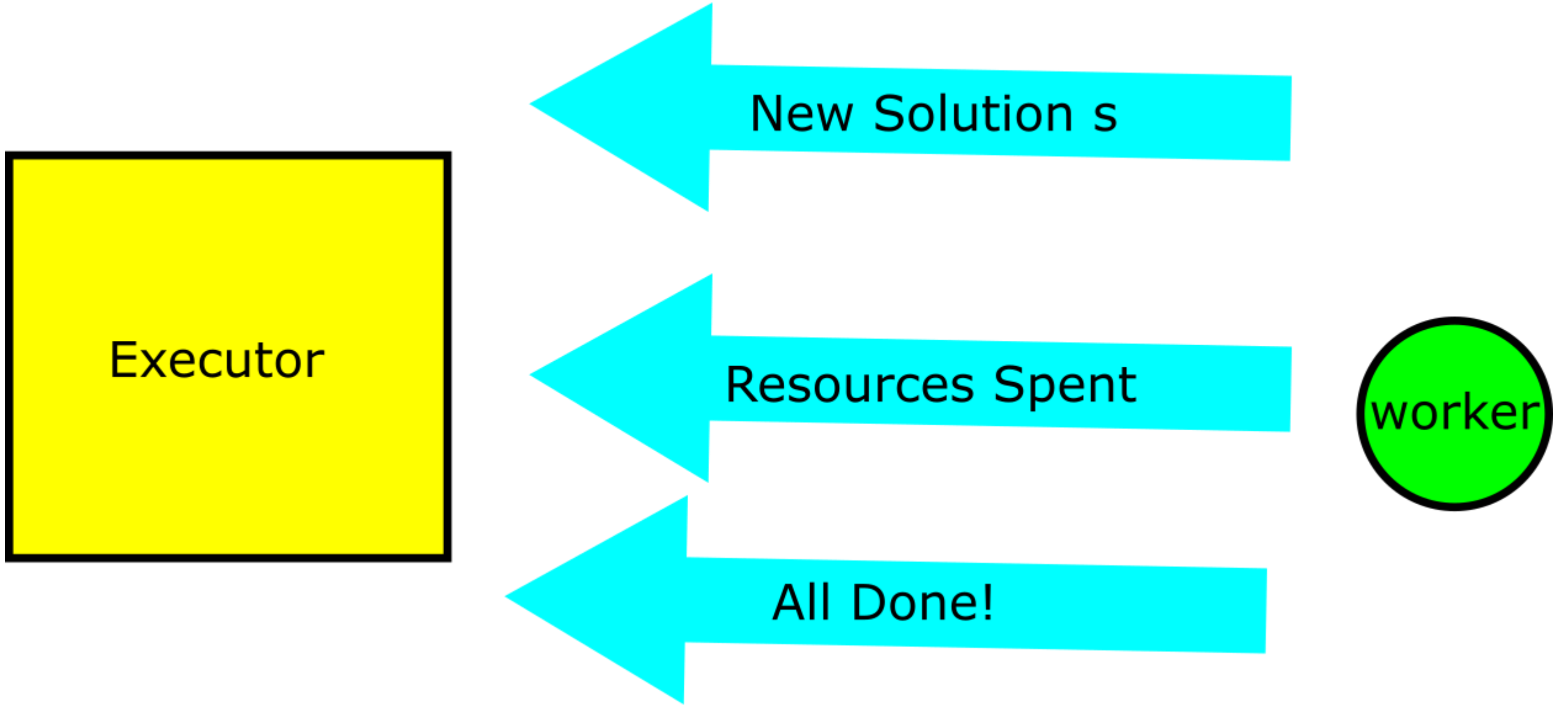
Distributed Depth First Search



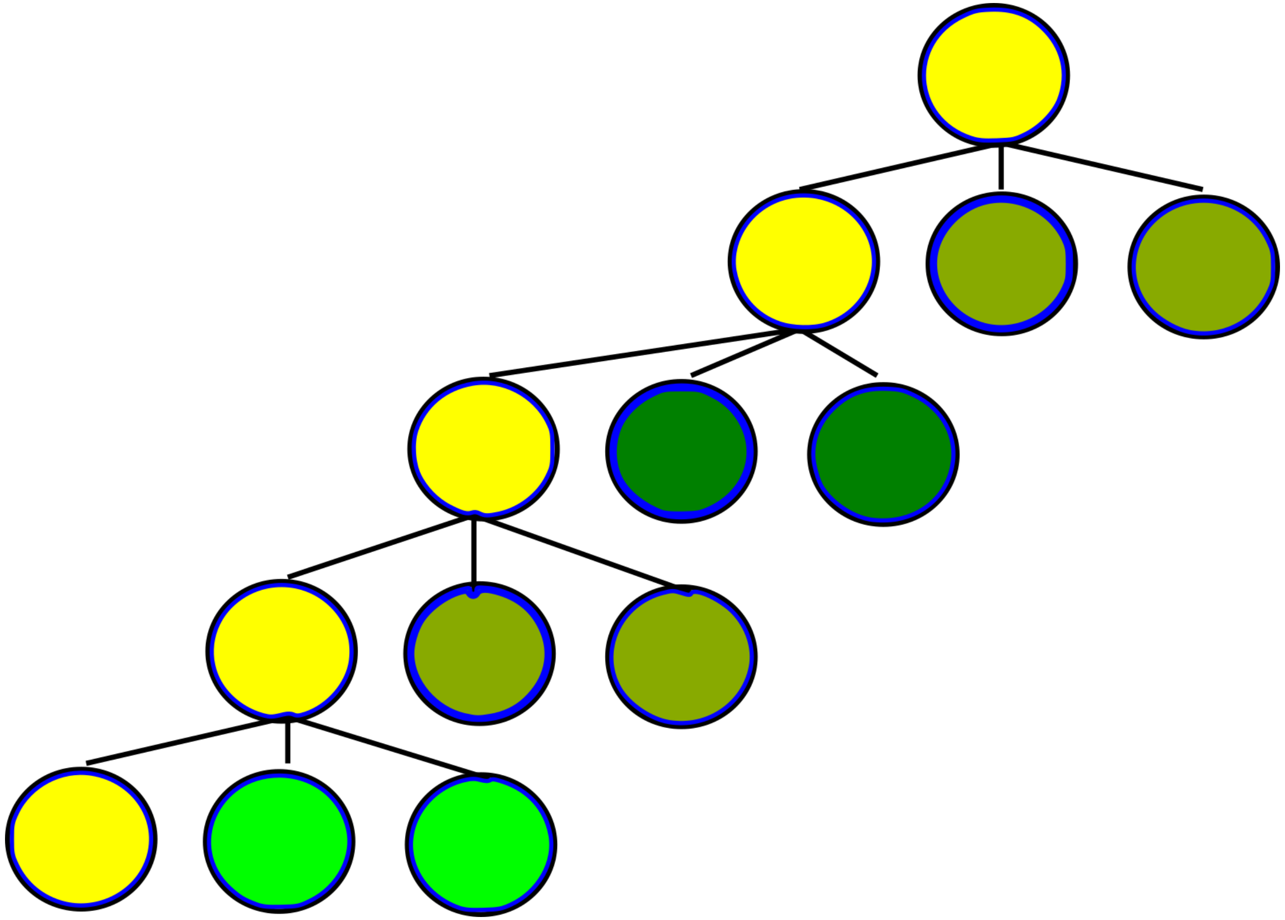
Distributed Depth First Search



Distributed Depth First Search



Distributed Depth First Search



DDFS Implementation

```
    openList := [ root ];
    let worker =
        MailboxProcessor.Start (fun inbox ->
            let rec loop () = async{
                let! msg = inbox.TryReceive(10)
                let halt = ref false
                match msg with
                    | Some (IncumbentCost g) -> incCost := Some g
                    | Some Halt -> halt:= true
                    | Some (Root r) -> openList := [ r ]
                    | None -> ()
                if !halt || search iterationsBetweenPolls then
                    printfn "Working %A is done or was told to halt" index;
                    executorInbox.Post (Metrics metrics);
                    executorInbox.Post (Done index) else
                    return! loop()
            } loop()
        ) in worker
```

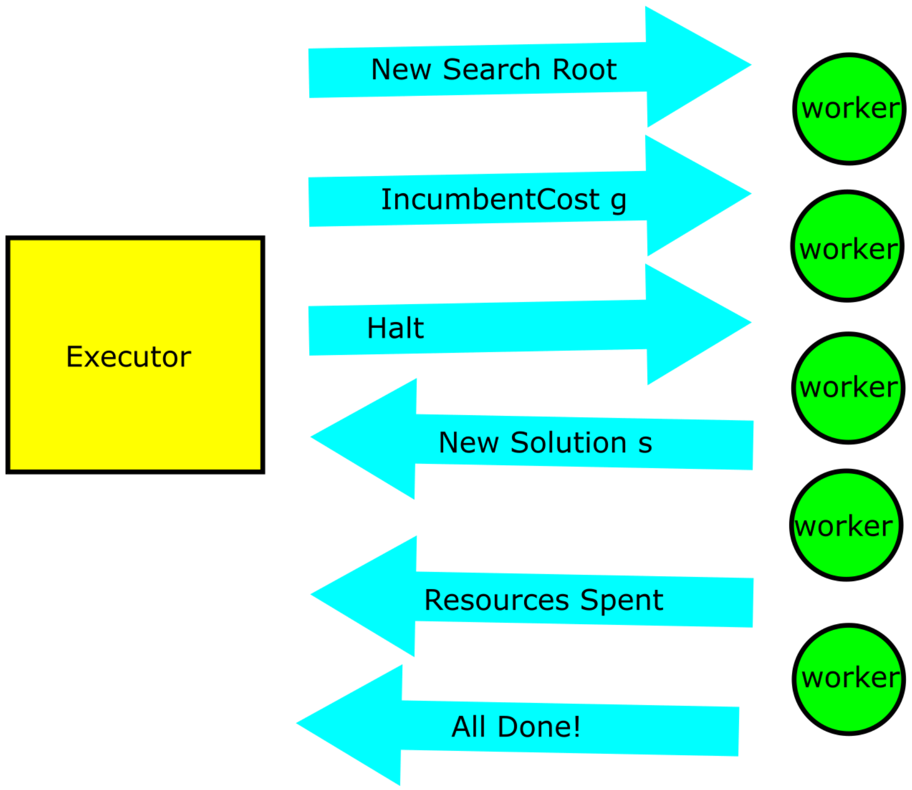
DDFS Implementation

```
let distributedIntCycles (iface : TreeSearch.TreeSearch<'state, int>) (numWorkers : int) =  // TreeSearch.TreeSearch<'state,int> -> int -> unit
    let totalMetrics = initMetrics() in
    let incumbent = ref None in
    let (workers : MailboxProcessor<MessageToWorker<'state,int>> Option array) = Array.init numWorkers (fun i -> None) in
    let betterSol sol =
        match !incumbent with
            | Some p -> p.cost < sol.cost
            | None -> true in
    let postInc (g : int) (w : MailboxProcessor<MessageToWorker<'state,'int>> Option) =
        match w with
        | None -> ()
        | Some w -> w.Post (IncumbentCost g) in
    let getSome = function
    | None -> false
    | Some _ -> true in
    let executer =
        MailboxProcessor.Start (fun inbox ->
            let rec loop () = async{
                let! msg = inbox.Receive()
                match msg with
                | IncumbentSolution sol ->
                    if betterSol sol then
                        incumbent := Some sol;
                        Array.iter (postInc sol.cost) workers
                | Metrics m -> mergeMetrics totalMetrics m
                | Done index -> printfn "%A is done with search" index; workers.[index] <- None
                return! loop()
            } loop()) in
    let makeWorker index =
        let urRoot = makeRoot iface.InitialCost iface.InitialState in
        let root = { urRoot with considering = index } in
            Some (makeIntCycles iface executer 100 root index) in
    for index in 0 .. (numWorkers - 1) do
        workers.[index] <- makeWorker index
    while (Array.exists getSome workers) do
        () //printfn "%A" incumbent
```

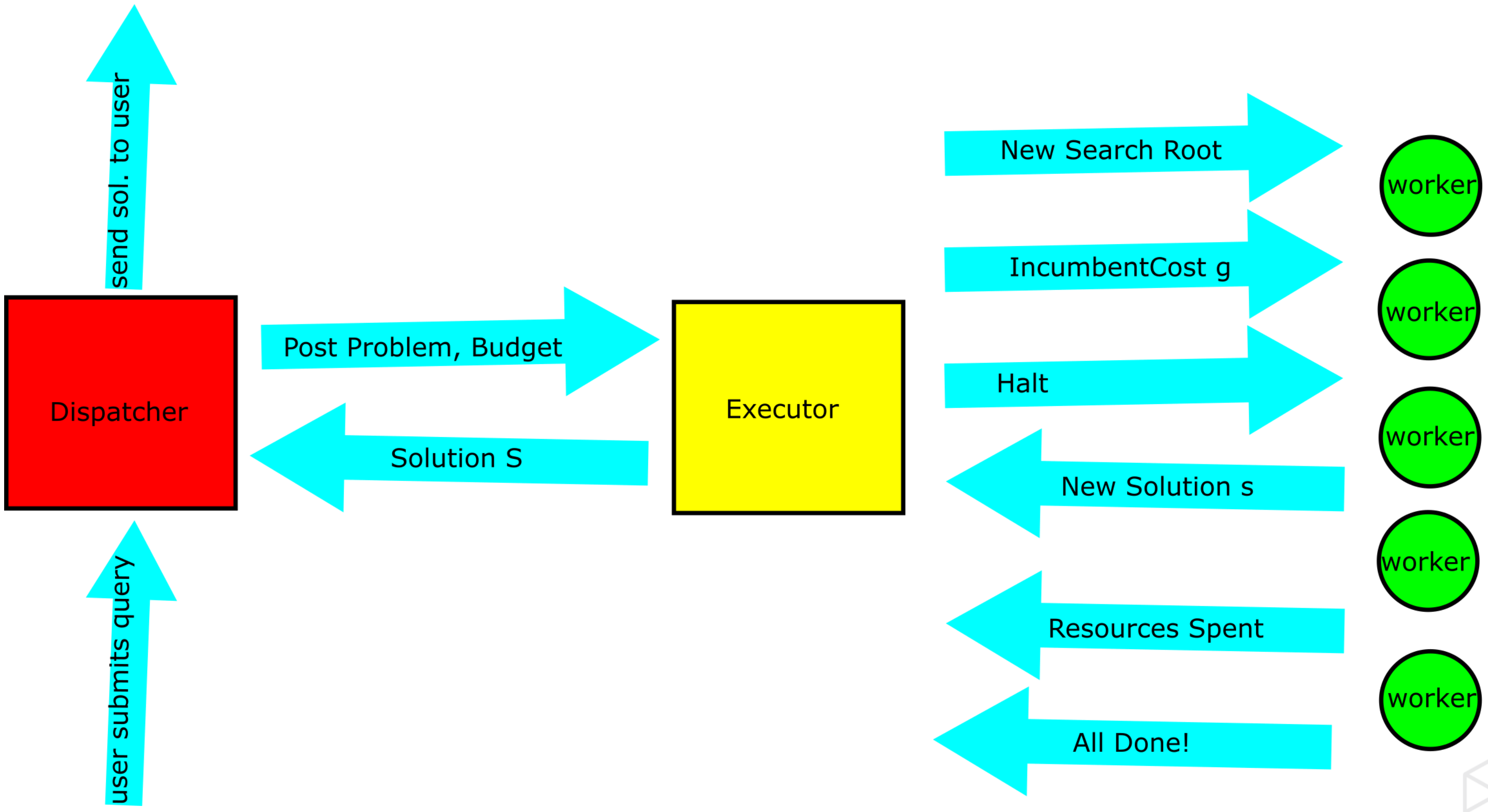
Talk Outline

- What is heuristic search and why should I care?
- Depth First Search: The Textbook Definition
- Depth First Search in Action
  - Pancake Flipping – a toy domain
  - The Travelling Salesman Problem
- Distributed Depth First Search
- Distributed DFS in the Cloud

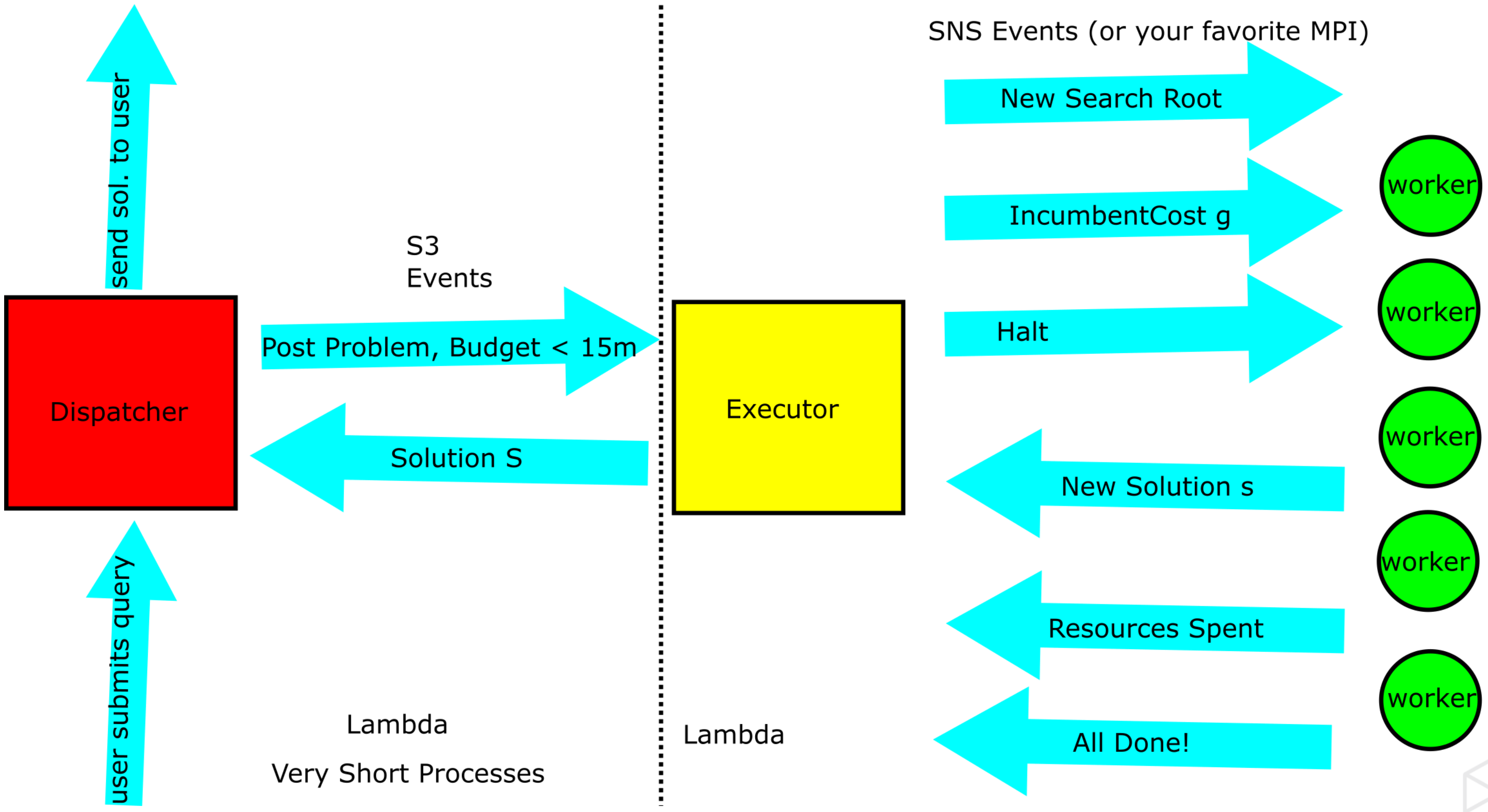
Distributed Depth First Search



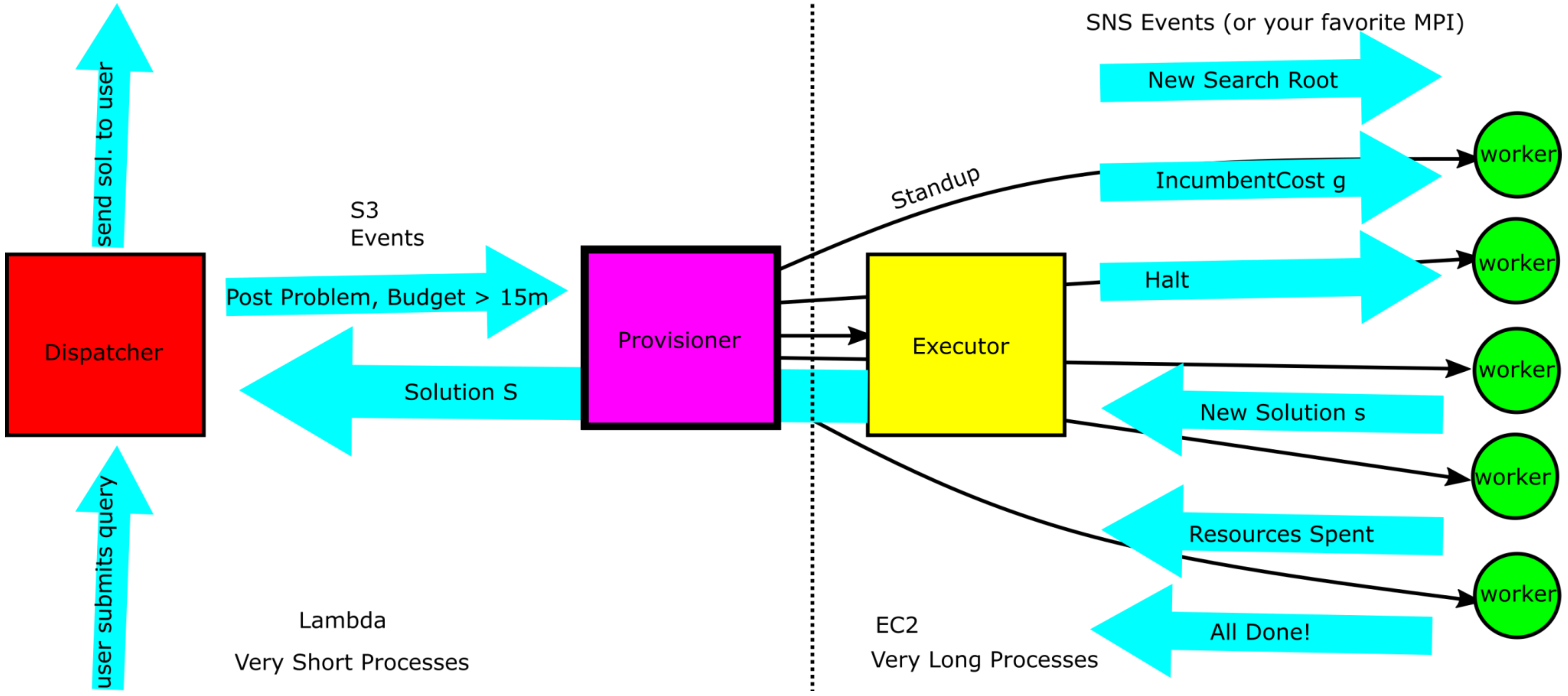
Distributed Depth First Search - Concept



Distributed Depth First Search – Low Budget



Distributed Depth First Search – Big Budget



- Thanks for your attention
- What questions do you have?

BACKUP SLIDES

- Here be dragons, proofs, F#

Wait, What's Optimal?

- Informally, it's the best solution to the problem
- Formally
  - Let goal(n) be the goal test applied to some node n
  - Let g(n) be the cost of arriving at some node n
  - Let G be the (potentially) infinite graph induced by the tree search
  - Then $Goals = \{ n \in G : goal(n)\}$
  - Then $Optimal = \{ n \in Goals : \forall m \in Goals : g(n) \leq g(m)\}$
  - Which is just "its cost is no more than that of any other goal"

Depth First Search: Convergence on Optimal

```
def depth_first_tree_search(problem):
   frontier = [Node(problem.initial)]  # Stack
   solution = None
   while frontier:
      node = frontier.pop()
      if is_cycle(node, problem.are_equal):
           continue
      if is_better(solution, node):
         continue
      if problem.goal_test(node.state):
         solution = node
      next = node.get_next_child(problem)  # child ordering is now baked into next_child
      if not next is None:
        frontier.extend([next, node])
   return solution
```

Pruning on incumbent solution

Depth First Search: Convergence on Optimal

```
def depth_first_tree_search(problem):
    frontier = [Node(problem.initial)]  # Stack
    solution = None
    while frontier:
        node = frontier.pop()
        if is_cycle(node, problem.are_equal):
              continue
        if is_better(solution, node):
            continue
        if problem.goal_test(node.state):
            solution = node
        next = node.get_next_child(problem)  # child ordering is now baked into next_child
        if not next is None:
           frontier.extend([next, node])
    return solution
```

We exhaust the space of all solutions

All nodes must improve

Solutions must improve

```
let makeIntCycles (iface : TreeSearch.TreeSearch<'state, int>) (executorInbox : MailboxProcessor<MessageToExecutor<'state,'int>>)  //
    (iterationsBetweenPolls : int) (root : Node<'state,'int>) (index : int) =
    let startTime = DateTime.Now in
    let better (a : Node<'state, 'cost>) (b : Node<'state, 'cost>) = a.g < b.g in
    let f (a : Node<'state, 'cost>) = a.g + (iface.H a.state) in
    let (openList : Node<'state, 'cost> list ref) = ref [] in
    let (incumbent : Node<'state, 'cost> option ref) = ref None in
    let (incCost : 'cost option ref) = ref None in
    let metrics = initMetrics() in
    let childOrder = makeIntChildOrder iface in
    let newGoal = makeIntNewGoal metrics startTime better incumbent in
    let cycleTest = makeIntCycleTest iface metrics in
    let counter = ref iterationsBetweenPolls in
    let betterSol node =
         match !incCost with
            | Some g -> g > (f node)
            | None -> true in
    let addNextChild node =
        let node' = match node.childOrder with
                    | [] -> { node with childOrder = computeOrder iface childOrder node}
                    | _ -> node in
        let childIndex = List.item node'.considering node'.childOrder in
        let (costDelta, childState) = iface.NthChild node'.state childIndex in
        let node'' = { node' with considering = node'.considering + 1 } in
        let child = { parent = Some node''; childOrder = []; state = childState; considering = 0; g = node''.g + costDelta } in
            metrics.gen <- metrics.gen + 1;
            push openList node'';
            push openList child in
    let searchStep node =
        if not (cycleTest node) then
            if betterSol node then begin
                if iface.Goal node.state then
                    incCost := Some node.g;
                    printfn "Worker %A is posting new solution" index;
                    executorInbox.Post (IncumbentSolution (getSolution node));
                    newGoal node else
                    if node.considering >= (iface.NumChildren node.state) then
                        metrics.exp <- metrics.exp + 1 else
                        addNextChild node end else
                metrics.pruned <- metrics.pruned + 1 in
    let rec search counter =
        if (!openList).Length = 0 then
            true else
            if counter <= 0 then
                false else
                let node = pop openList in
                    searchStep node;
                    search (counter - 1) in
```

nplementation

A More Exact Definition of Pancakes

treeSearch.fs | depthFirstSearch.fs | pancakes.fs

```fsharp
module Pancakes

type State = int list

type Problem = {
    numCakes : int;
    initState : State;
}

let newProblem seed size = // int option -> int -> Problem

let goalTest state = // 'a list -> bool

let cakesOutOfOrder state = // 'a list -> int

let flipStack state index = // 'a list -> int -> int * 'a list

type TreeSearchInterface(problem) =···

let solve problem = // Problem -> unit

[<EntryPoint>]
let main argv = // string [] -> int
```

State, Instance Definition

A More Exact Definition of Pancakes

treeSearch.fs | depthFirstSearch.fs | pancakes.fs ●

```fsharp
module Pancakes

type State = int list

type Problem = {···
}

let newProblem seed size = // int option -> int -> Problem

let goalTest state = // 'a list -> bool

let cakesOutOfOrder state = // 'a list -> int

let flipStack state index = // 'a list -> int -> int * 'a list
    let aboveSpatula, belowSpatula = List.splitAt index state in
    1, (List.rev aboveSpatula) @ belowSpatula

type TreeSearchInterface(problem) =···

let solve problem = // Problem -> unit

[<EntryPoint>]
let main argv = // string [] -> int
```

Action Definition

A More Exact Definition of Pancakes

treeSearch.fs | depthFirstSearch.fs | pancakes.fs

```
module Pancakes

type State = int list

type Problem = {···
}

let newProblem seed size = // int option -> int -> Problem

let goalTest state = // 'a list -> bool
    let rec test = function
    | [] -> true
    | [_] -> true
    | a::b::tl -> a < b && test (b::tl) in
    test state

let cakesOutOfOrder state = // 'a list -> int

let flipStack state index = // 'a list -> int -> int * 'a list

type TreeSearchInterface(problem) =···

let solve problem = // Problem -> unit

[<EntryPoint>]
let main argv = // string [] -> int
```

Goal Definition

A More Exact Definition of Pancakes

```
//Inadmissible Heuristics
let cakesOutOfOrder state = // int list -> int

let longestRunHeuristic state = // int list -> int
    let ret = ref 0 in
    let rec count cur = function
    | []
    | [_] -> if cur > !ret then ret := cur
    | a::b::tl ->
        let delta = abs(a - b) in
        if delta > 1 then
            (if cur > !ret then ret := cur);
            count 1 (b::tl)  else
            count (cur + 1) (b::tl) in
    count 1 state;
    !ret

//Admissible Heuristics
let gapHeuristic state = // int list -> int
    let rec count = function
    | []
    | [_] -> 0
    | a::b::tl ->
        let delta = abs(a - b ) in
        let thisVal = if delta > 1 then 1 else 0 in
        thisVal + (count (b::tl)) in
    count state

let notGoalHeuristic state = // 'a list -> int
```

Heuristics

Domain Meets Search

```
type NaiveTreeSearchInterface(problem) =
    interface TreeSearch.NaiveTreeSearch<State, int> with
        member this.InitialState = problem.initState // State
        member this.Goal state = goalTest state // State -> bool
        member this.H state = if goalTest state then 0 else 1 // State -> int
        member this.D state = if goalTest state then 0 else 1 // State -> int
        member this.Equal s1 s2 = s1 = s2 // State -> State -> bool
        member this.InitialCost = 0 // int
        member this.Expand state = // State -> (int * State) list
            List.mapi (fun ind el -> flipStack state (ind+1)) state

type TreeSearchInterface(problem) =···

let solve problem = // Problem -> unit
    let iface = NaiveTreeSearchInterface(problem) in
    let solNode, metrics = DFS.naiveDFS iface in
        printfn "%A" metrics;
        match solNode with
        | None -> printfn "No solution"
        | Some s -> DFS.getSolution s |> printfn "%A"

[<EntryPoint>]
let main argv = // string [] -> int
    for i in 1 .. 1 do
        let problem = newProblem None 4 in
            printfn "Problem: %A" problem;
            solve problem
    0
```

Here’s how Pancakes fulfils that interface.

Here’s us telling DFS to solve the abstracted problem.

What’s f, why is it special?

- $g(n)$ is the cost of reaching a node n
- $h(n)$ is a lower bound on the cost of an optimal solution starting at n
- $h^*(n)$ is the true cost of an optimal solution starting at n
- $h^*(n) = h(n) = 0 \; if \; goal(n)$

- $f(n) = g(n) + h(n)$
- $f^*(n) = g(n) + h^*(n)$ is the true cost of an optimal solution

- $f(n) < f^*(n) < f^*(sol) = g(sol)$ and additionally,
- $f^*(n) \geq f(n) \geq f^*(sol) = g(sol)$

TSP Problem Representation

```
type State = {
    mutable current : int;
    mutable visitedSoFar : int;
    mutable visited : bool array;
}

type Delta = {⋯
}

let applyDelta (s : State) (d : Delta) =   // State -> Delta -> float

let undoDelta (s : State) (d: Delta) = // State -> Delta -> float

type Problem = {
    mutable origin : int;
    mutable numberCities : int;
    mutable cityLocations : float [,];
    mutable distanceMatrix : float array array;
}

let deterministicInit problem = // Problem -> State

let emptyProblem phLabel num = { // 'a -> int -> Problem
}

let initProblemSquare num problem = // int -> Problem -> unit
    problem.numberCities <- num;
    problem.cityLocations <- Array2D.create num 2 0.;
    problem.distanceMatrix <-
        Array.init num (fun index -> Array.create num 0.);
```

TSP Heuristics

```
let nextNearest (p : Problem) (s : State) = // Problem -> State -> int list
    let getDist = getDistance p s.current in
    let accum = ref [] in
    let i = ref 0 in
    for ind in 0 .. p.numberCities do begin
        if not s.visited.[ind] then
            accum := (getDist ind, !i) :: !accum;
            i := !i + 1;
    end;
    List.sort !accum |> List.map (fun (_,b) -> b)

let euclidH (p : Problem) (s : State) = // Problem -> State -> float
    let (minX : float ref) = ref p.cityLocations.[p.origin,0] in
    let (minY : float ref) = ref p.cityLocations.[p.origin,1] in
    let (maxX : float ref) = ref p.cityLocations.[p.origin,0] in
    let (maxY : float ref) = ref p.cityLocations.[p.origin,1] in
    for ind in 0 .. p.numberCities - 1 do
        if not s.visited.[ind] then
            let (cx : float) = p.cityLocations.[ind,0] in
            let (cy : float) = p.cityLocations.[ind,1] in
                (if cx < !minX then minX := cx);
                (if cx > !maxX then maxX := cx);
                (if cy < !minY then minY := cy);
                (if cy > !maxY then maxY := cy)
    // have a bounding box on cities and origin
    let deltaX = !maxX - !minX in
    let deltaY = !maxY - !minY in
        deltaX + deltaY
```

One for child ordering

One for pruning

Search is domain agnostic!

```
type TreeSearchInterface(problem) =
    interface TreeSearch.TreeSearch<State, float> with
        member this.InitialState = deterministicInit problem // State
        member this.Goal state = goalTest problem state // State -> bool
        member this.H state = euclidH problem state  // State -> float
        member this.D state = d problem state // State -> int
        member this.Equal s1 s2 = equal s1 s2 // State -> State -> bool
        member this.NumChildren state = numChildren problem state // State -> int
        member this.NthChild state n = nthChild problem state n // State -> int -> float * Stat
        member this.InitialCost = 0. // float
        member this.ChildOrder = None // (float * State -> float * State -> int) option

type InPlaceModificationTreeSearch(problem) =⋯

let solve problem = // Problem -> unit
    let iface = TreeSearchInterface(problem) in
    let solNode, metrics = DFSv2.floatCycles iface in
    //let solNode, metrics = ILDS.ildsCycles iface in
        printfn "%A" metrics;
        match solNode with
        | None -> printfn "No solution"
        | Some s -> ImperativeTreeSearch.getSolution s |> printfn "%A"
```

```
type TreeSearchInterface(problem) =
    interface TreeSearch.TreeSearch<State, int> with
        member this.InitialState = problem.initState // State
        member this.Goal state = goalTest state // State -> bool
        member this.H state = gapHeuristic state  // State -> int
        member this.D state = gapHeuristic state // State -> int
        member this.Equal s1 s2 = s1 = s2 // State -> State -> bool
        member this.NumChildren _ = problem.numCakes - 1 // State -> int
        member this.NthChild state n = flipStack state (n + 2) // State -> int -> int * State
        member this.InitialCost = 0 // int
        member this.ChildOrder = None // (int * State -> int * State -> int) option

let solveNaive problem = // Problem -> unit

let solve problem = // Problem -> unit
    let iface = TreeSearchInterface(problem) in
    let solNode, metrics = DFSv2.intCycles iface in
    //let solNode, metrics = ILDS.ildsCycles iface in
        printfn "%A" metrics;
        match solNode with
        | None -> printfn "No solution"
```