Simplifying Heuristic Search and Domain Modeling With F#

Talk Outline

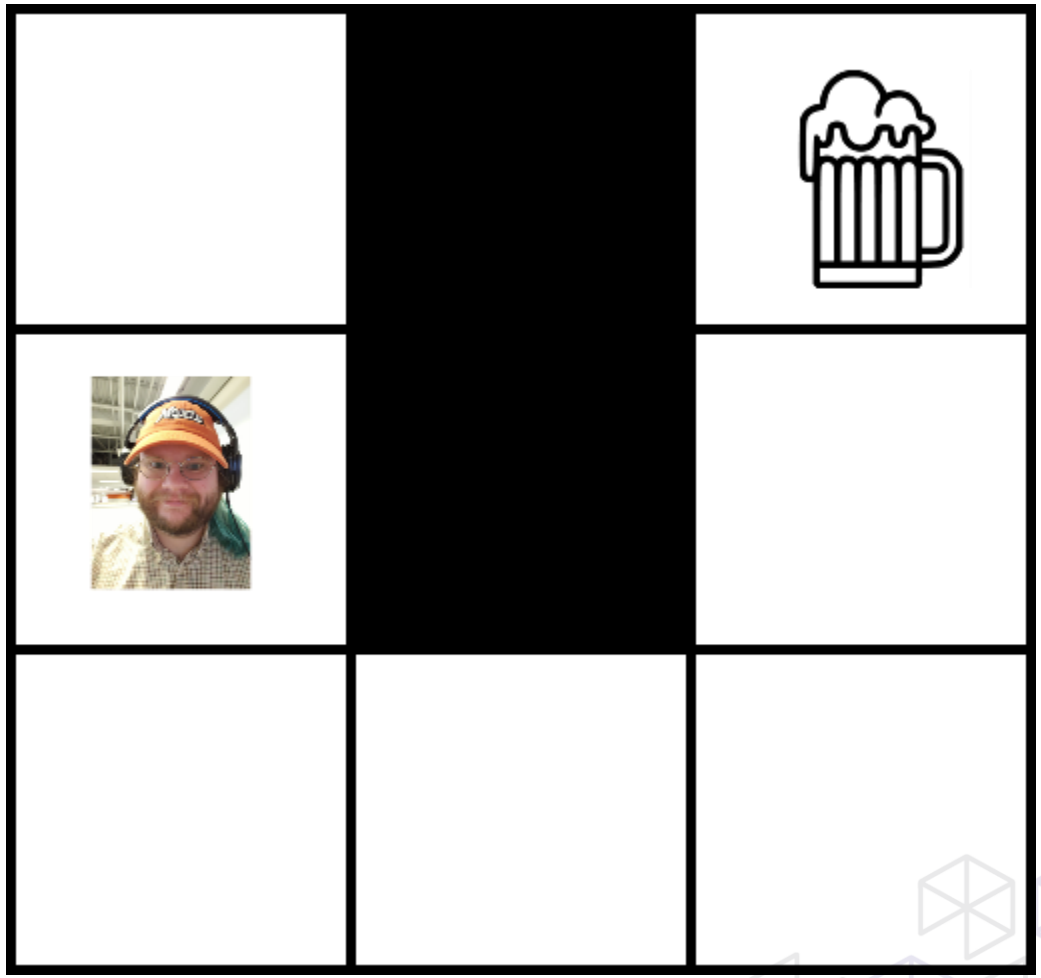
- Introduction
  - Grid Navigation Problem Setup
  - Example Domains - Toys
  - Example Domains - Industrial
- An Algorithm: A*
- Fitting Domains To Interface
- Conclusion

What is Search, Why do I care?

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

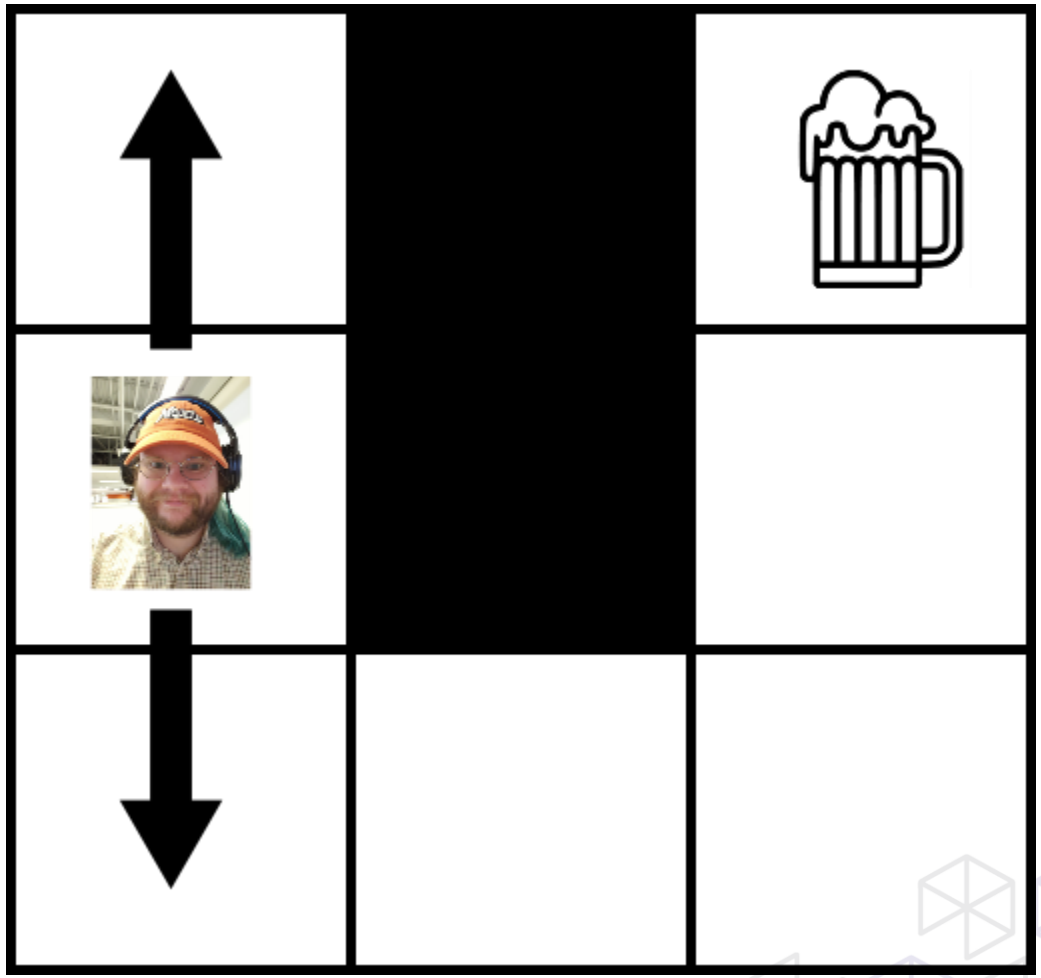
What is Search, Why do I care?

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



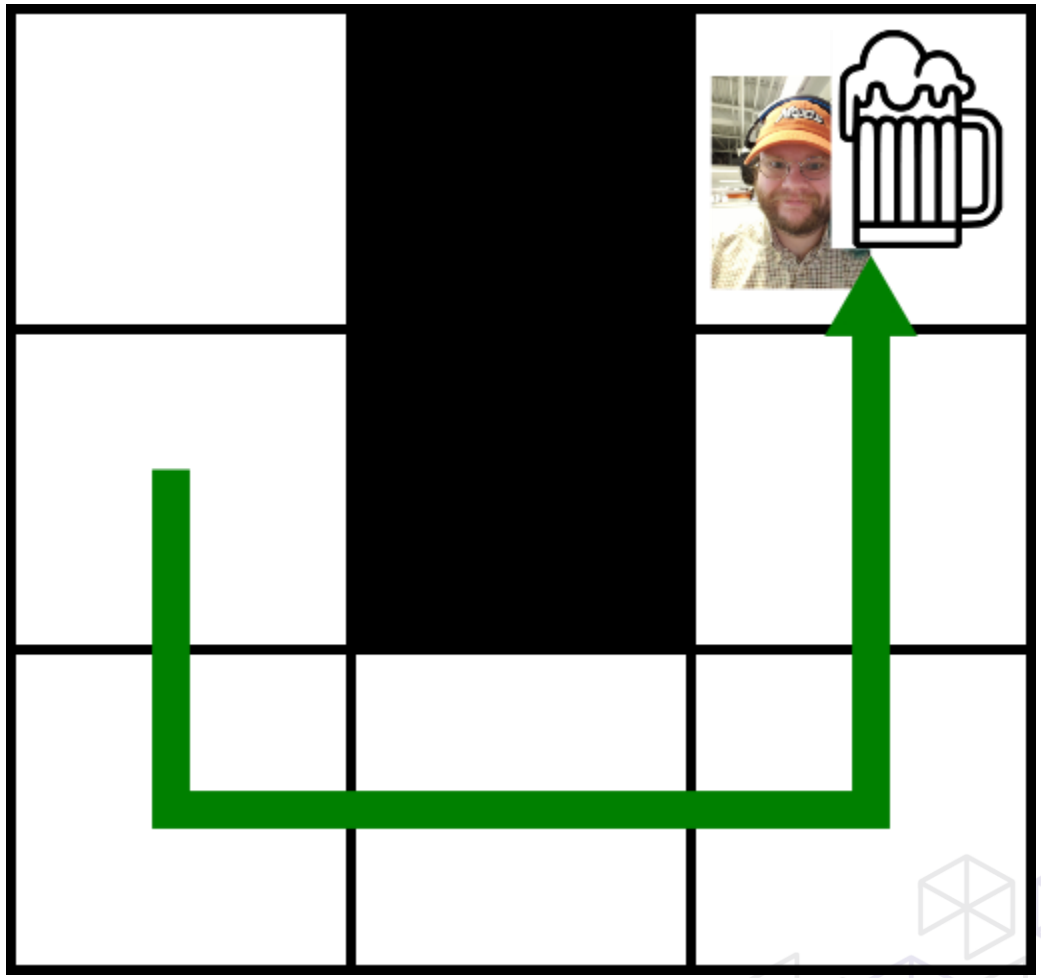
What is Search, Why do I care?

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



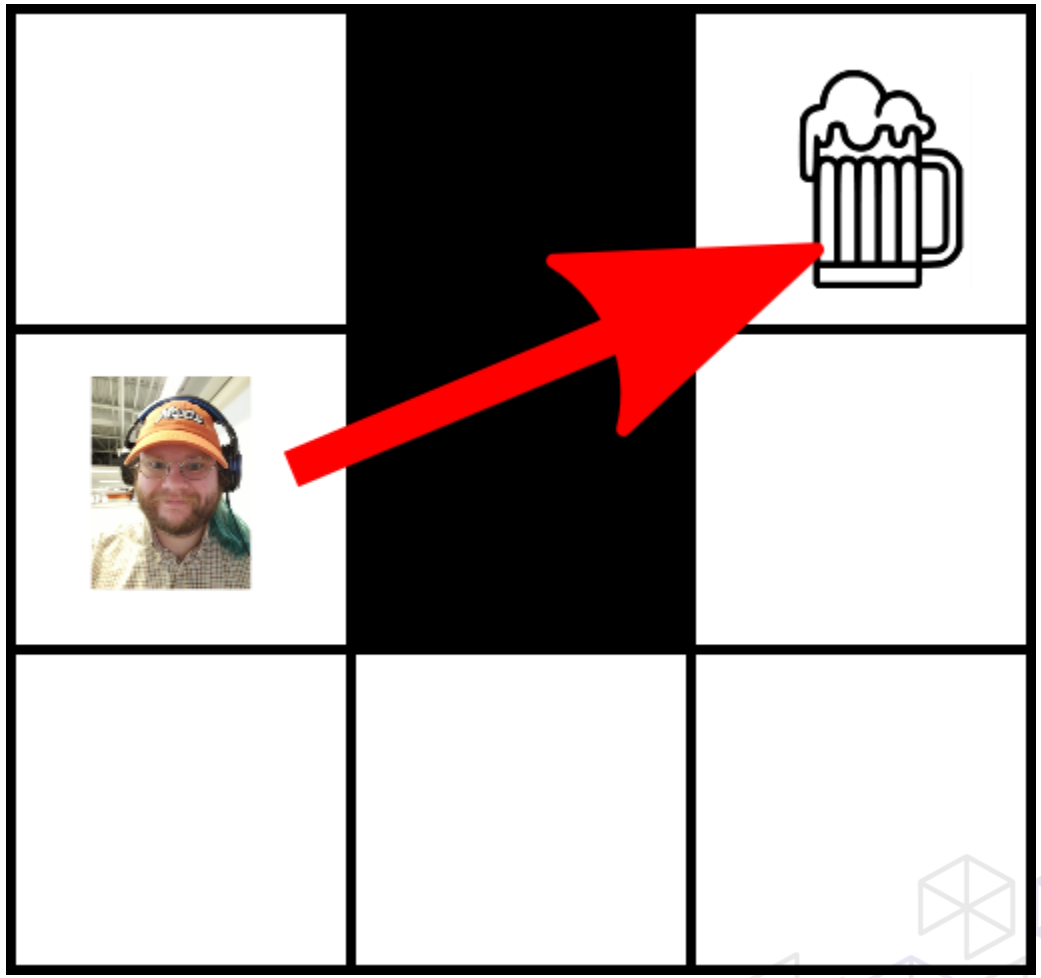
What is Search, Why do I care?

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



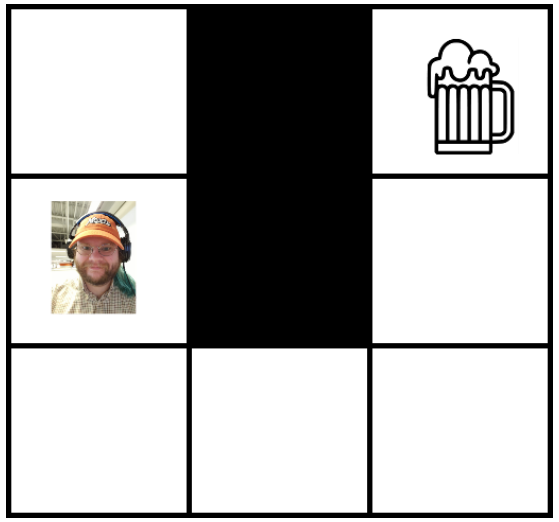
What is Search, Why do I care?

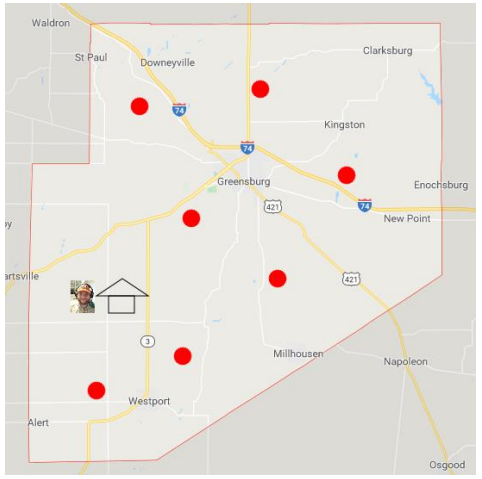
- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics

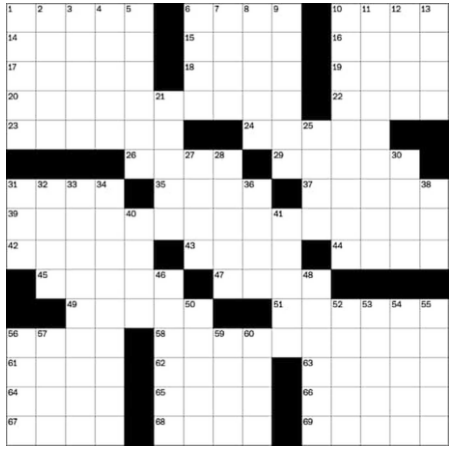


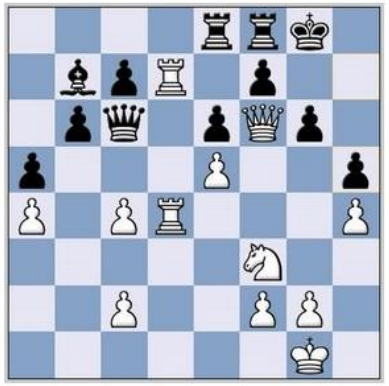
What is Search, Why do I care?

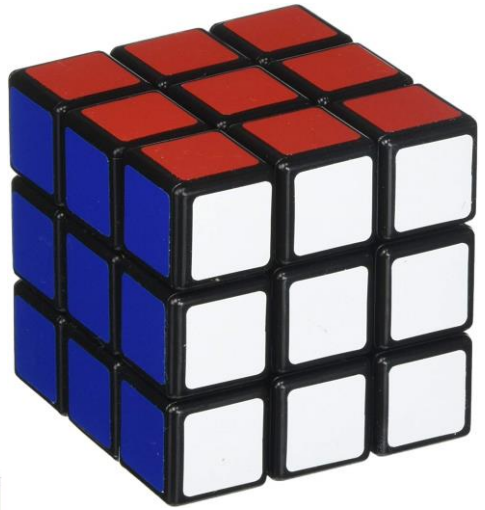
- Search is a *general* technique for solving problems
- The search cares about your problem using this abstraction:
  - States
  - Actions
  - Goals
  - Heuristics





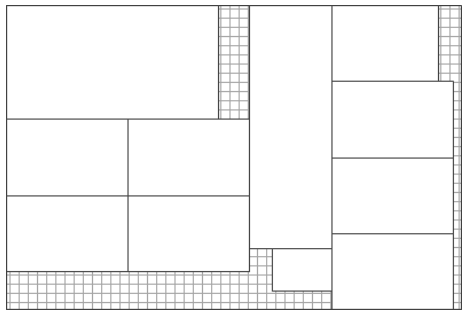


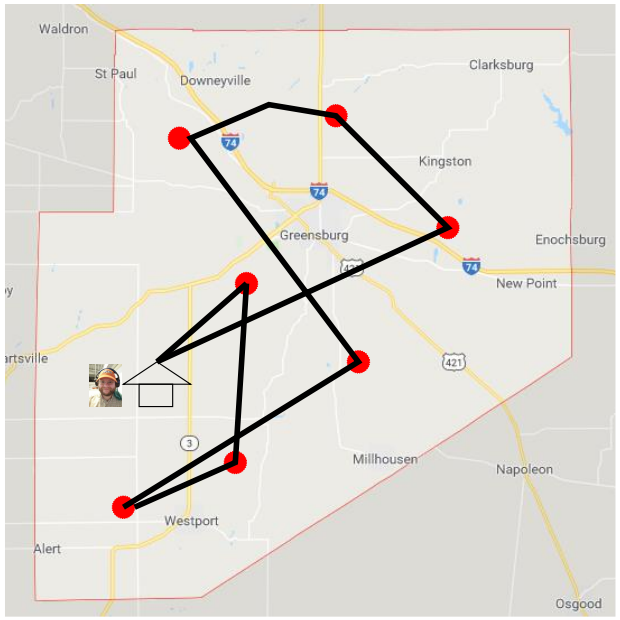


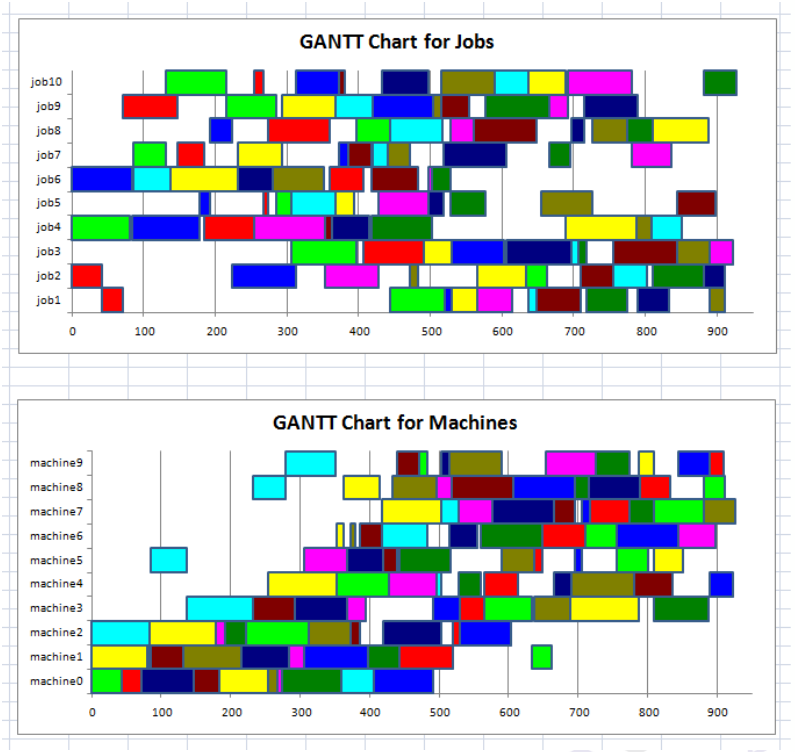


What is Search, Why do I care?

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





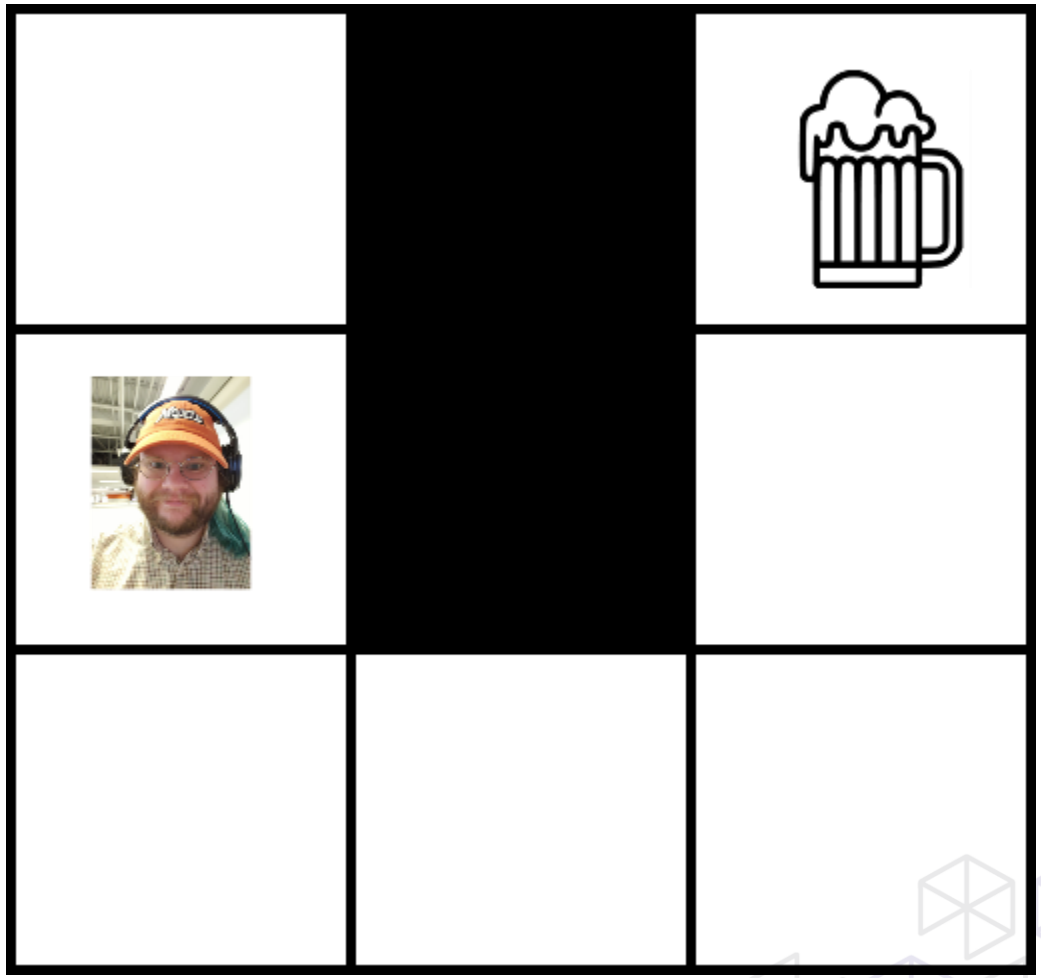


Talk Outline

- Introduction
- Heuristic Search and Types
- An Algorithm: A*
- Fitting Domains To Interface
- Conclusion

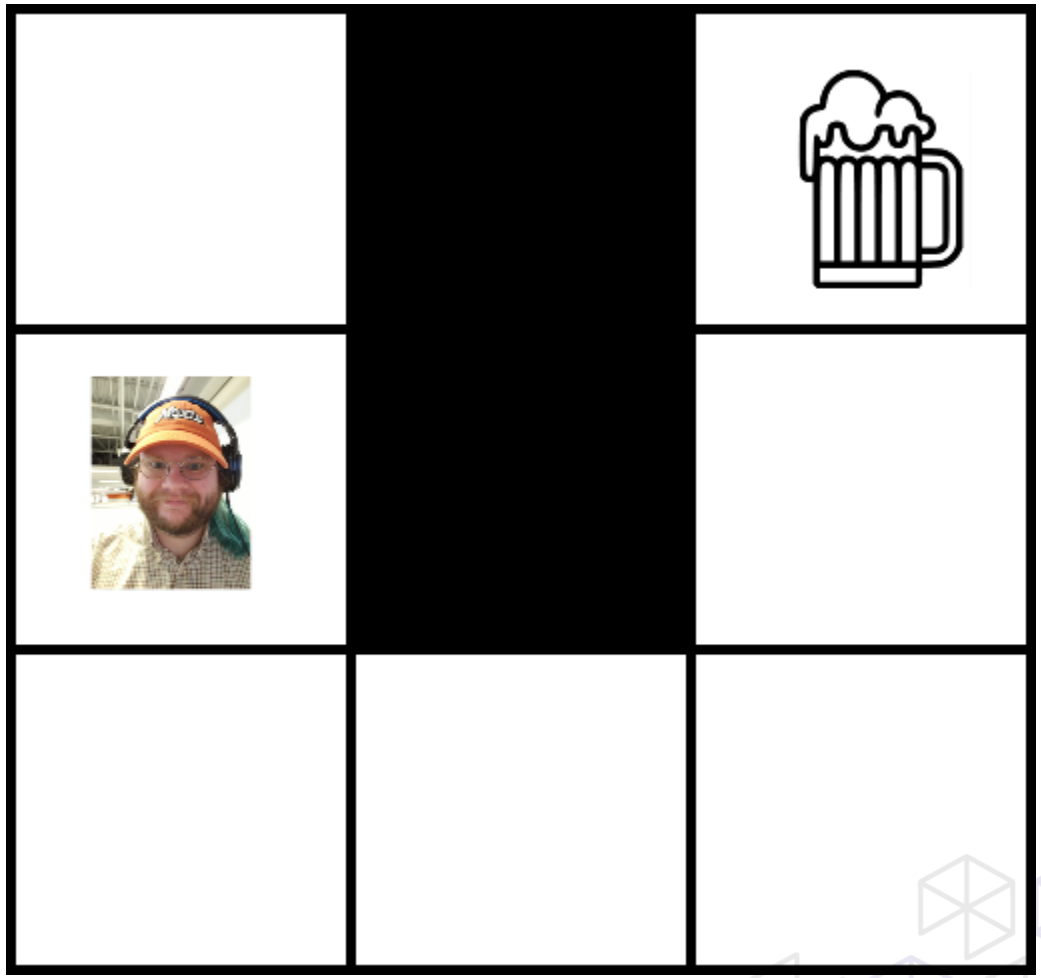
Heuristic Search: Type Perspective

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



Heuristic Search: Type Perspective

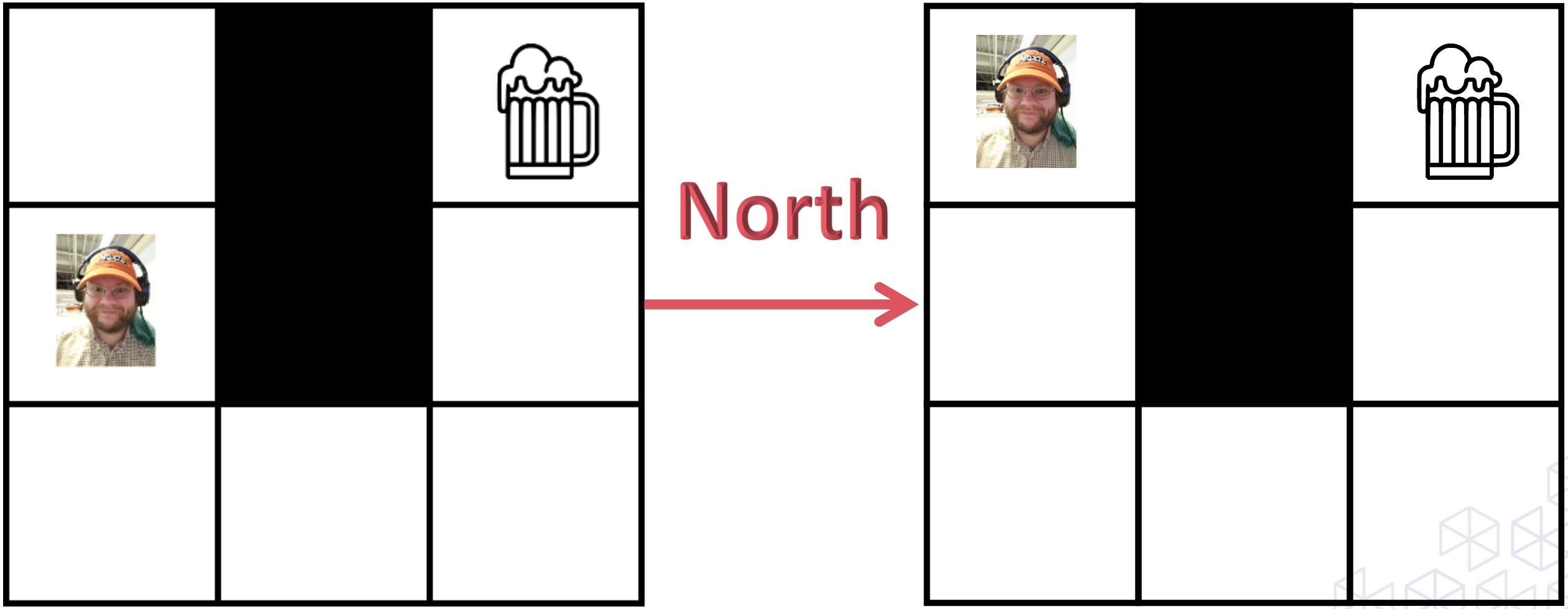
- Search is a technique for solving problems
- These problems look like this:
  - Graphs
    - Nodes
    - Edges
  - Functions on Nodes
    - Goal predicate
    - Heuristic Estimate



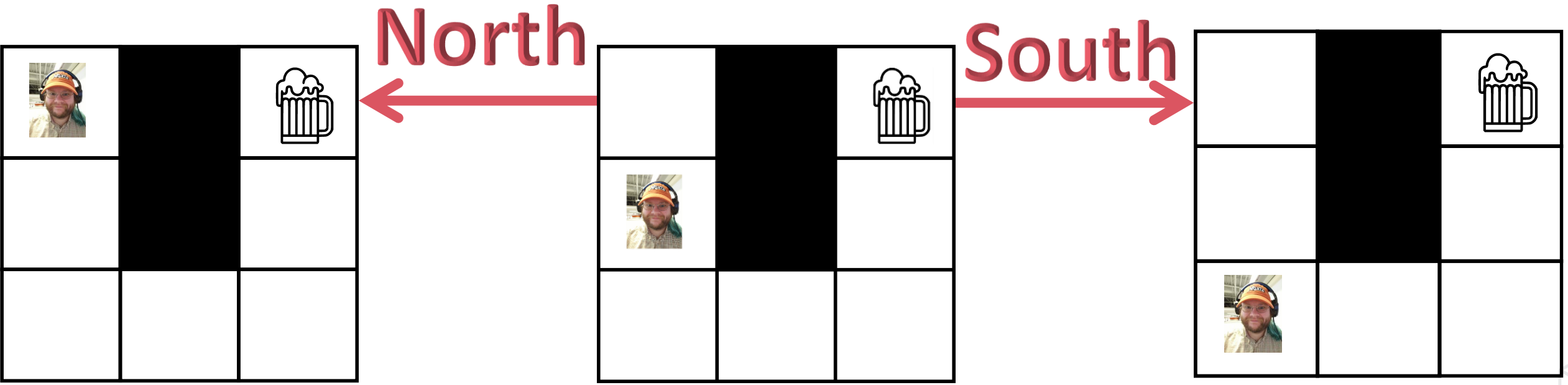
Graph Search: Nodes and Edges

```
4   type Edge<'state, 'action when 'state: comparison and 'action: comparison > =
5       {
6           origin : 'state
7           dest : 'state
8           by_way_of : 'action
9       }
27  type Graph<'state, 'action when 'state: comparison and 'action: comparison>  = {
28      nodes : Set<'state>;
29      edges : Set<Edge<'state, 'action>>;
30  }
```

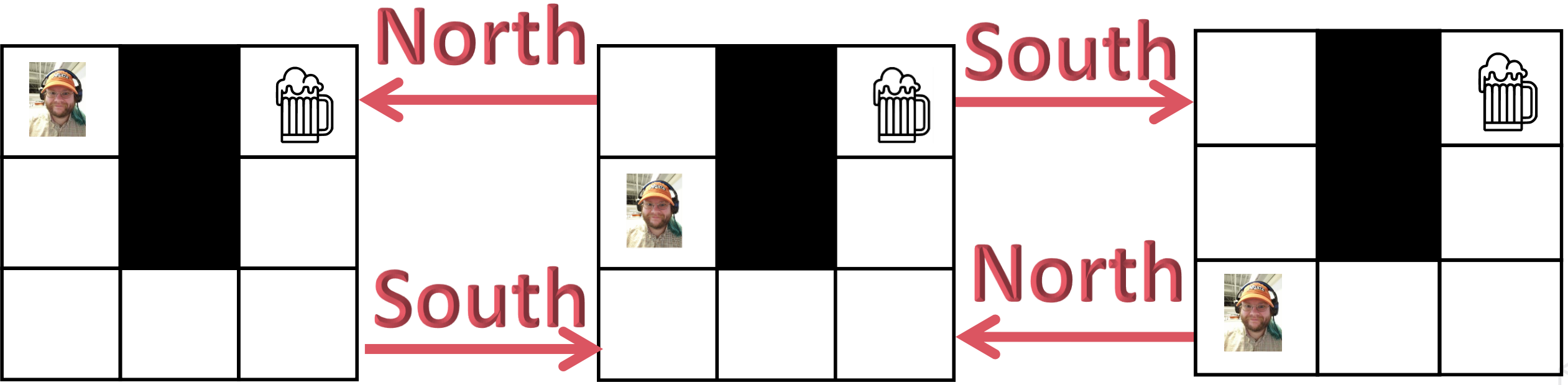
Graph Search: Nodes and Edges



Graph Search: Nodes and Edges



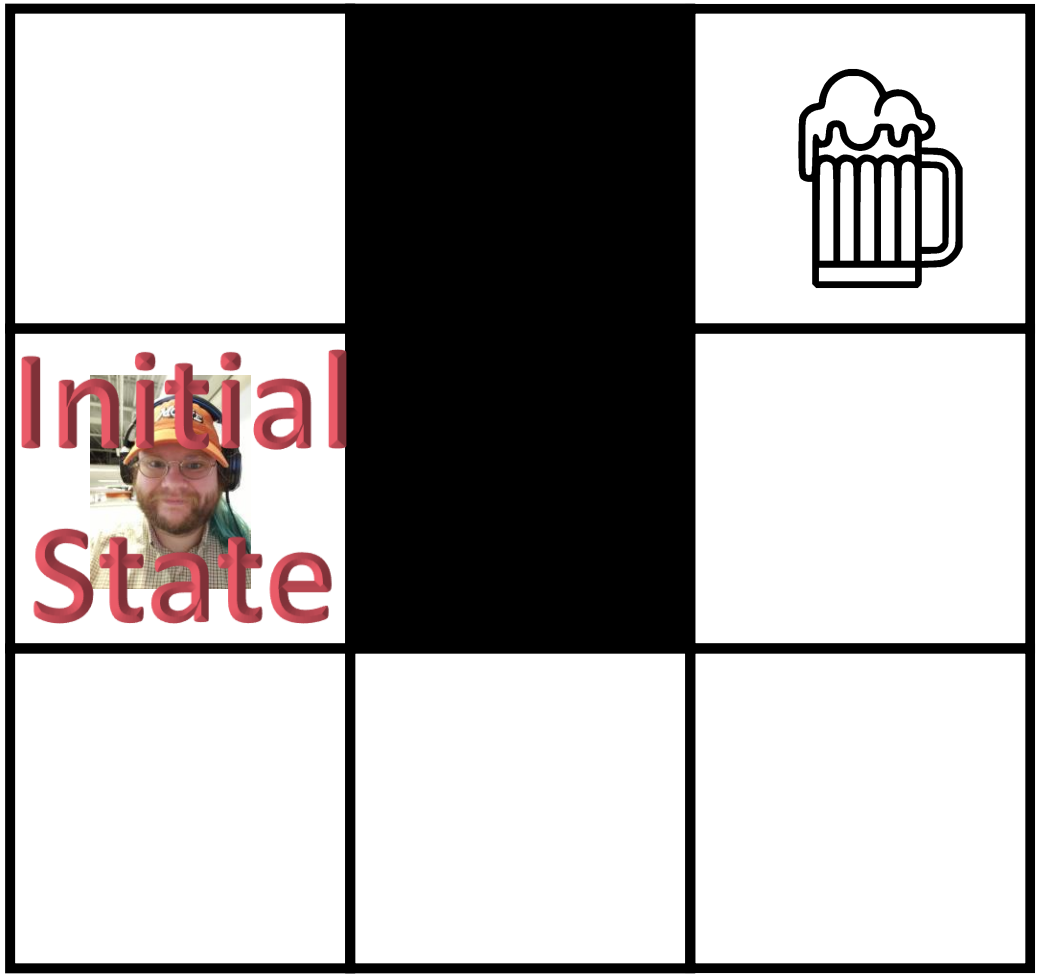
Graph Search: Nodes and Edges

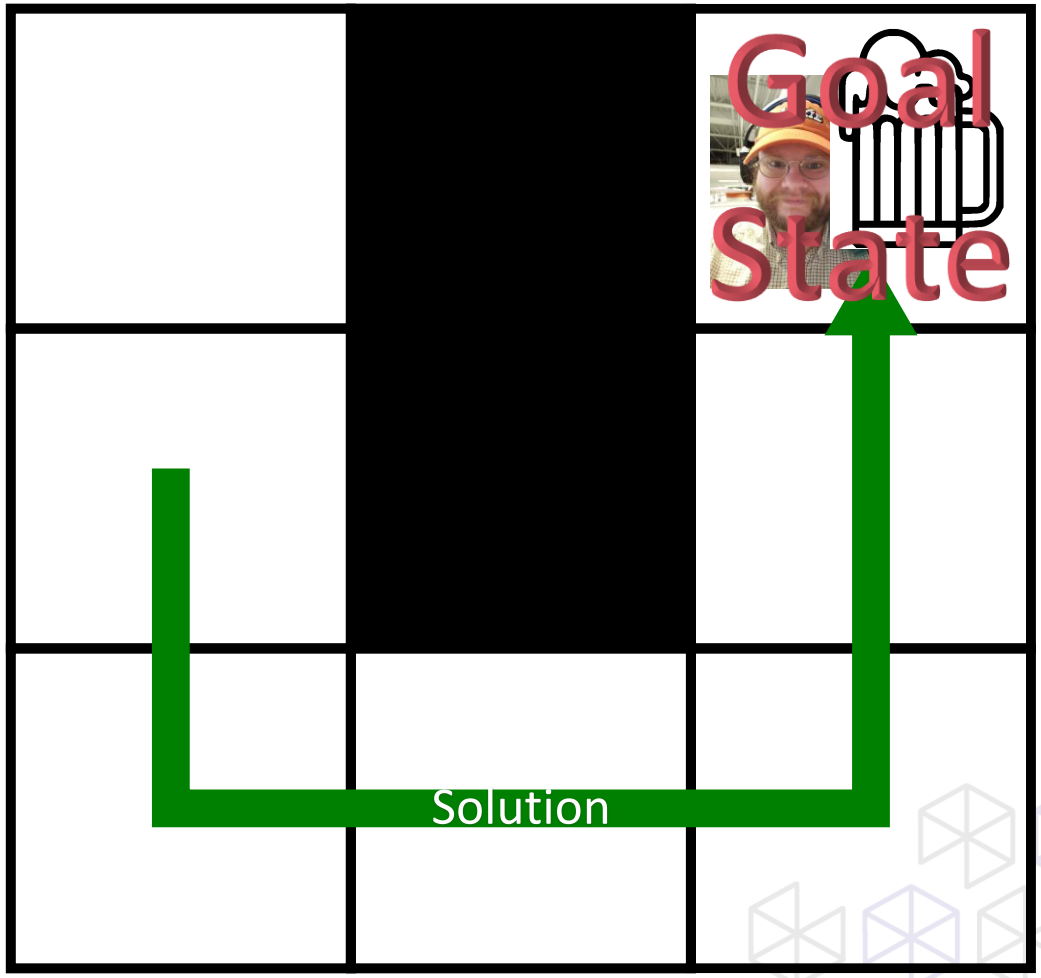


Graph Search: Nodes and Edges

```
4   type Edge<'state, 'action when 'state: comparison and 'action: comparison > =
5       {
6           origin : 'state
7           dest : 'state
8           by_way_of : 'action
9       }
27  type Graph<'state, 'action when 'state: comparison and 'action: comparison>  = {
28      nodes : Set<'state>;
29      edges : Set<Edge<'state, 'action>>;
30  }
```

Heuristic Search: Type Perspective





Heuristic Search: Type Perspective

```
32  type GraphSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
33      graph : Graph<'state, 'action>;
34      initial_state : 'state;
35      goals : Set<'state>;
36  }
37
38  type Solution<'state, 'action when 'state: comparison and 'action: comparison> = {
39      initial_state : 'state;
40      path : Edge<'state, 'action> list;
41  }
```

Heuristic Search: Type Perspective

Explicitly representing the graph is often infeasible due to size!

```
32  type GraphSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
33      graph : Graph<'state, 'action>;
34      initial_state : 'state;
35      goals : Set<'state>;
36  }
37
38  type Solution<'state, 'action when 'state: comparison and 'action: comparison> = {
39      initial_state : 'state;
40      path : Edge<'state, 'action> list;
41  }
```

Heuristic Search: Type Perspective

```
32  type GraphSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
33      graph : Graph<'state, 'action>;
34      initial_state : 'state;
35      goals : Set<'state>;
36  }
37
38  type Solution<'state, 'action when 'state: comparison and 'action: comparison> = {
39      initial_state : 'state;
40      path : Edge<'state, 'action> list;
41  }
```

```
43  type ImplicitGraphSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
44      initial_state : 'state
45      goal_p : 'state -> bool
46      expand : 'state -> Set<Edge<'state, 'action>>
47  }
48
49  // An ImplicitGraphSearchProblem can be converted into a GraphSearchProblem by
50  // taking the transitive closure of initial_state and expand
51
```

Heuristic Search: Type Perspective

```
32  type GraphSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
33      graph : Graph<'state, 'action>;
34      initial_state : 'state;
35      goals : Set<'state>;
36  }
37
38  type Solution<'state, 'action when 'state: comparison and 'action: comparison> = {
39      initial_state : 'state;
40      path : Edge<'state, 'action> list;
41  }
```

So don't represent the graph explicitly. Generate it lazily.

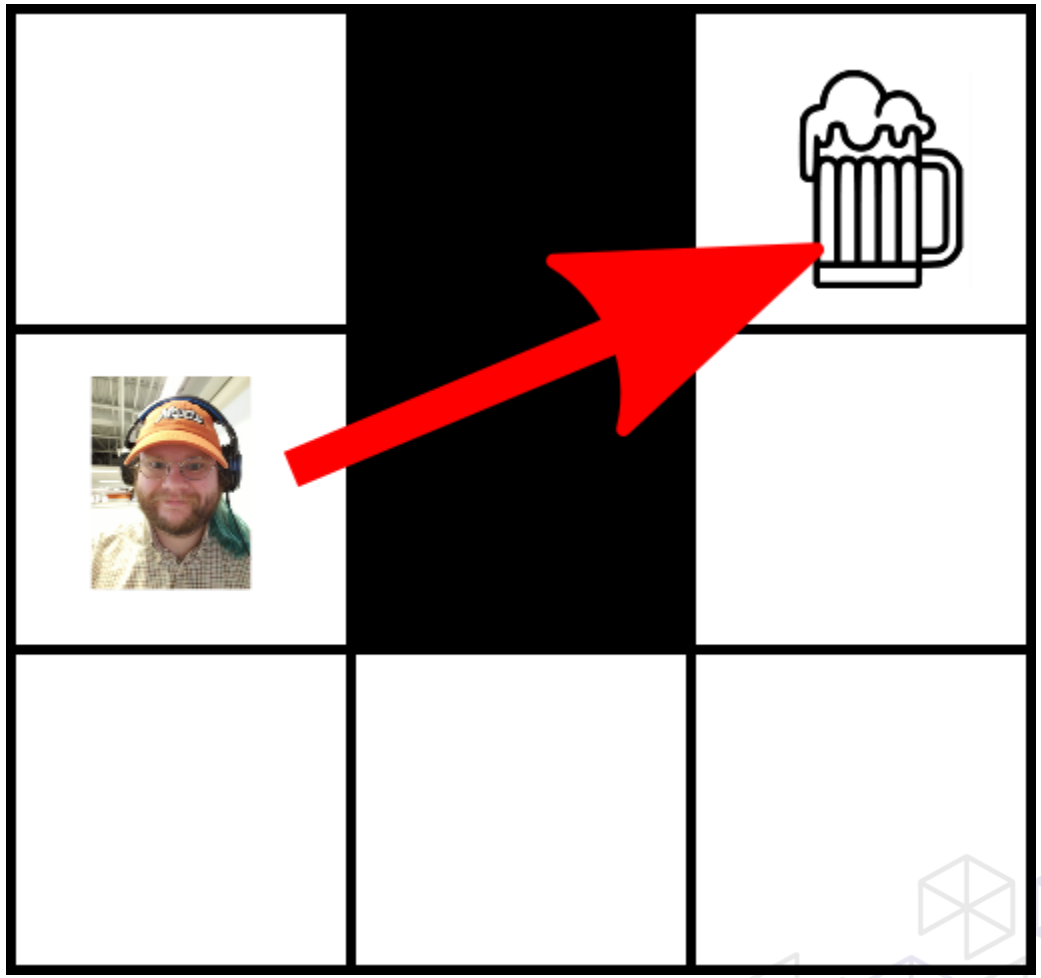
```
43  type ImplicitGraphSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
44      initial_state : 'state
45      goal_p : 'state -> bool
46      expand : 'state -> Set<Edge<'state, 'action>>
47  }
48
49  // An ImplicitGraphSearchProblem can be converted into a GraphSearchProblem by
50  // taking the transitive closure of initial_state and expand
51
```

Heuristic Search: Type Perspective

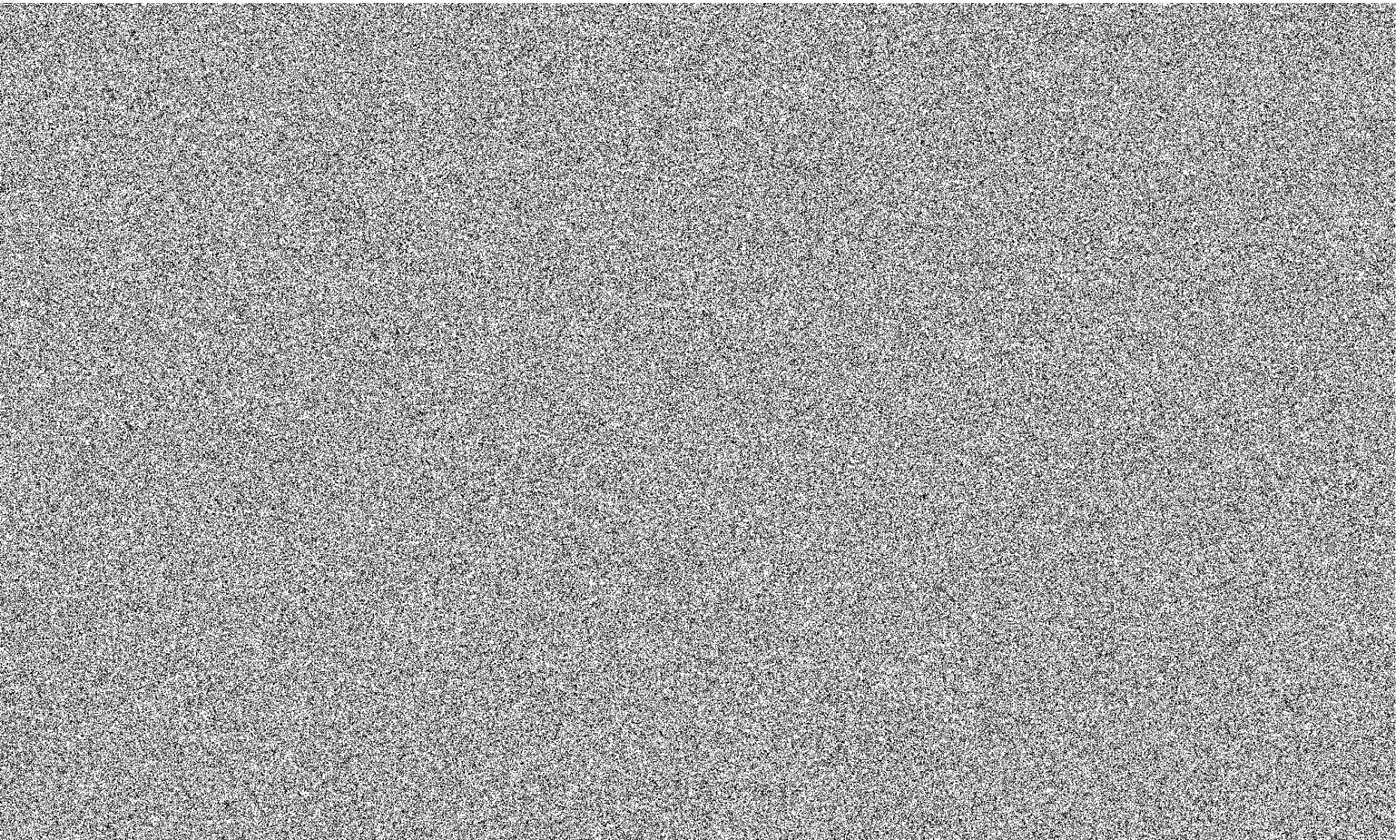
```
52  type HeuristicSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
53      initial_state : 'state
54      goal_p : 'state -> bool
55      expand : 'state -> Set<Edge<'state, 'action>>
56      heuristic : 'state -> float
57  }
58
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

What is Search, Why do I care?

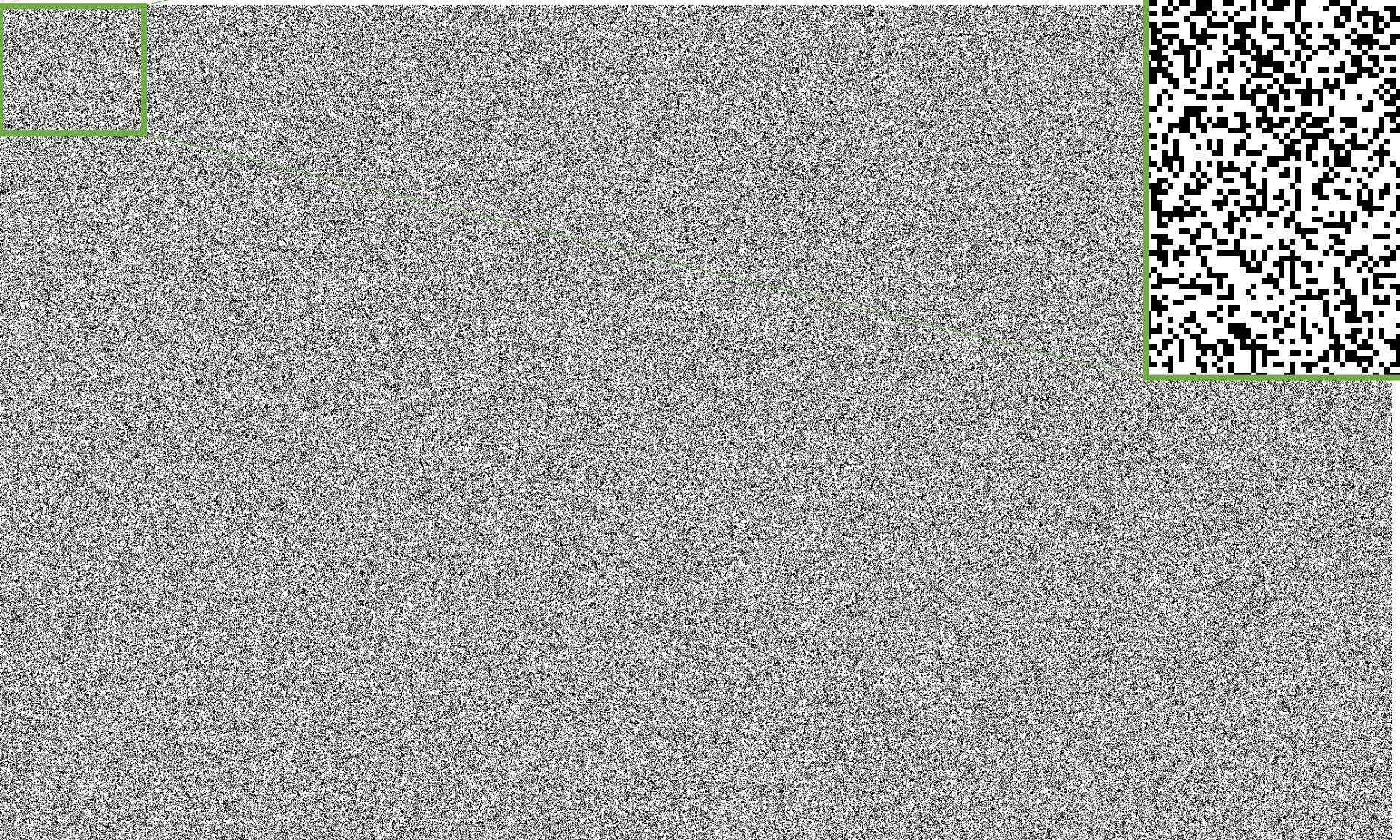
- Search is a technique for solving problems
- These problems look like this:
  - States
  - Actions
  - Goals
  - Heuristics

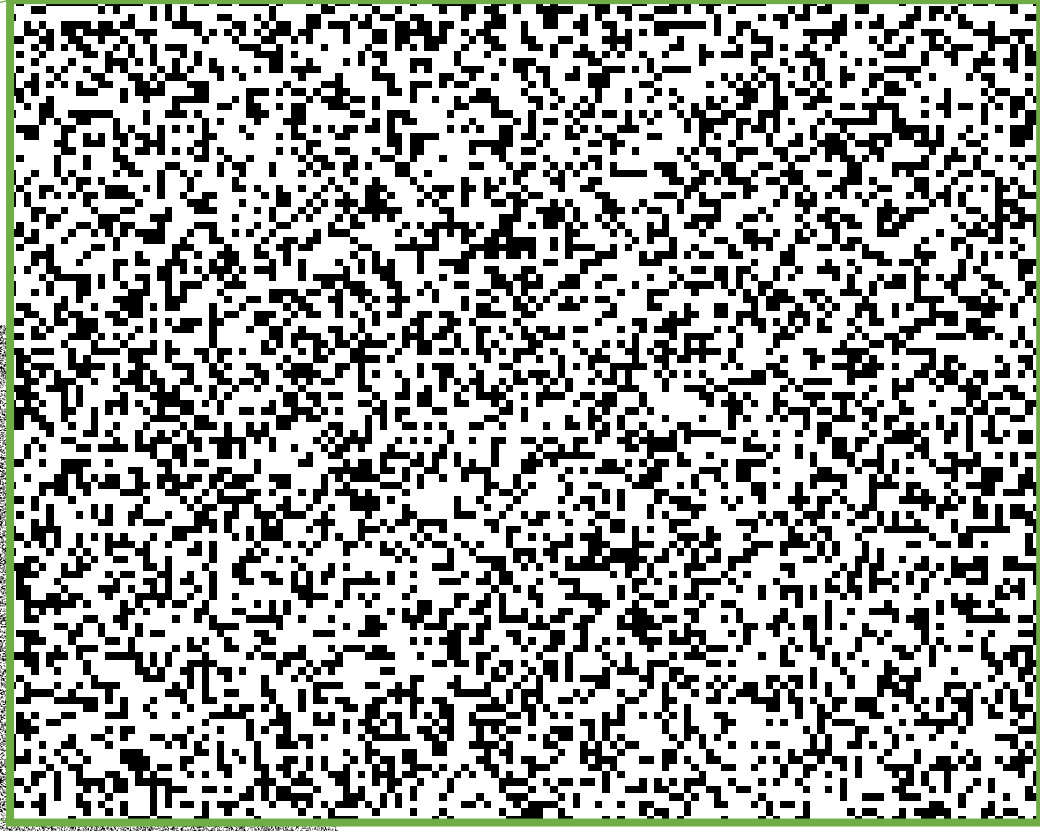


Heuristic Search

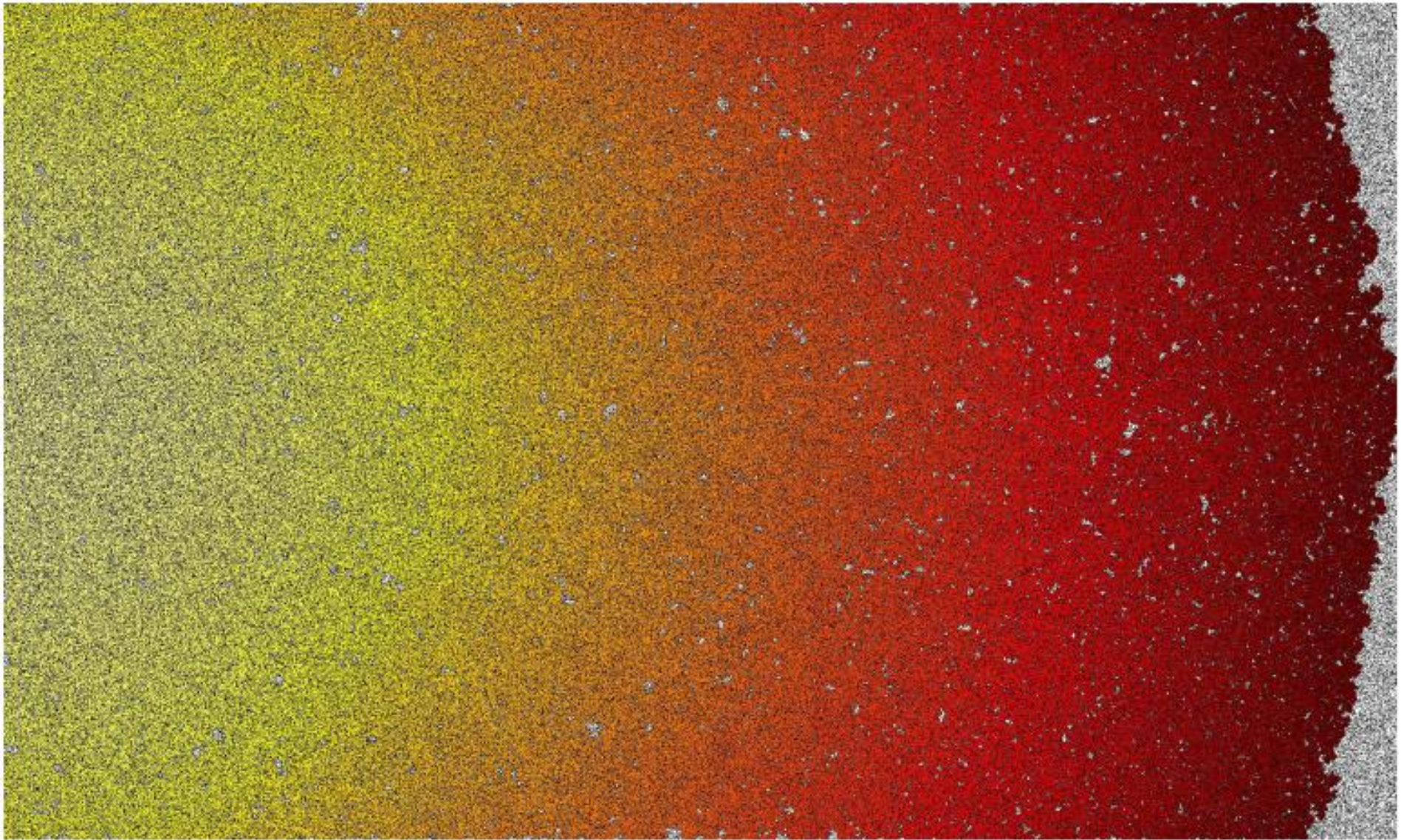


Heuristic Search

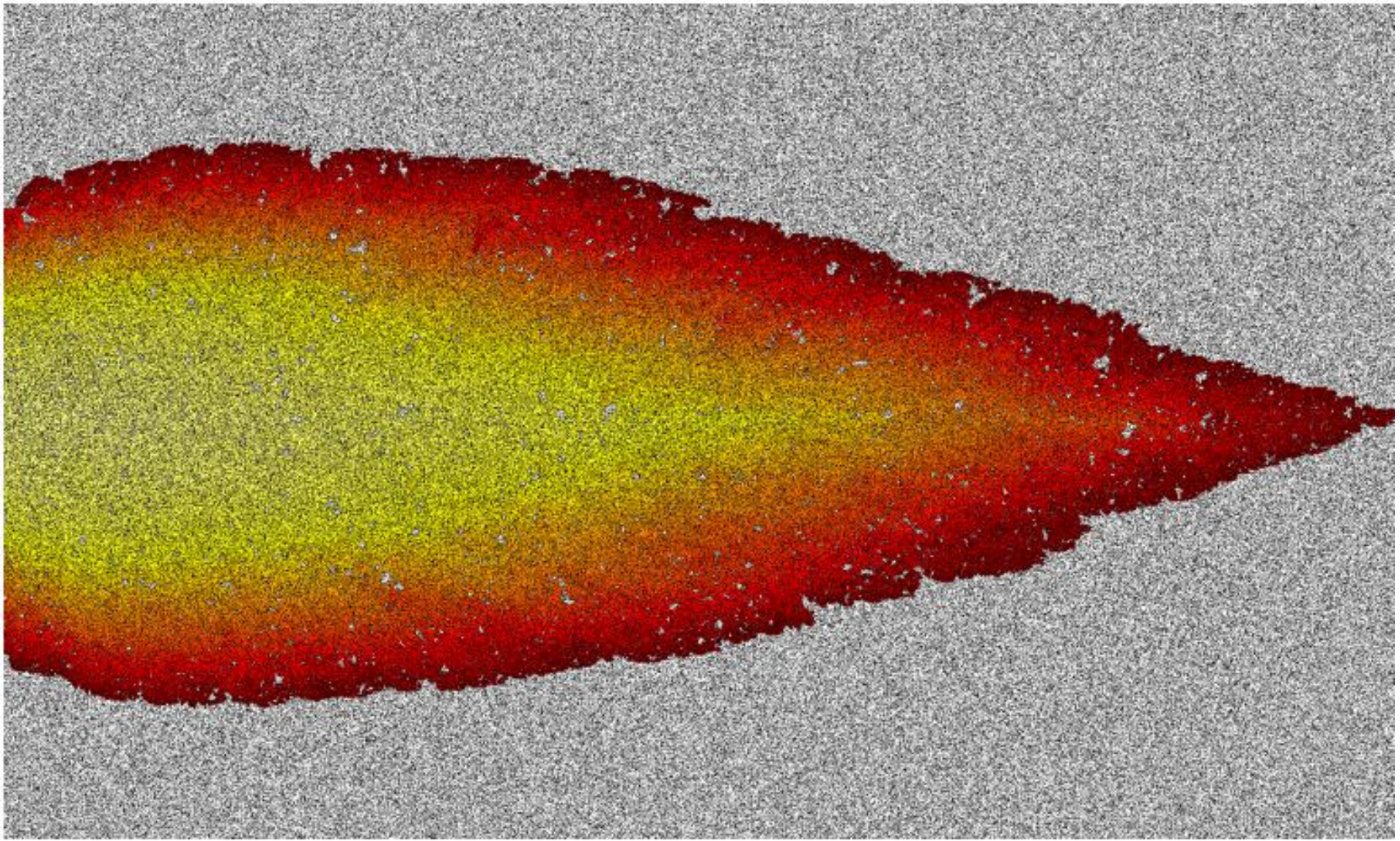




Heuristic Search



Heuristic Search – A*



Heuristic Search: Type Perspective

```
52  type HeuristicSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
53      initial_state : 'state
54      goal_p : 'state -> bool
55      expand : 'state -> Set<Edge<'state, 'action>>
56      heuristic : 'state -> float
57  }
58
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand Goal Heuristic Initial Solution

Heuristic Search: Type Perspective

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand Goal Heuristic Initial Solution

Graph search

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if *fringe* is empty **then return** failure
  node ← REMOVE-FRONT(*fringe*)
  if GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
end

Heuristic Search: Type Perspective

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand | Goal | Heuristic | Initial | Solution

Graph search

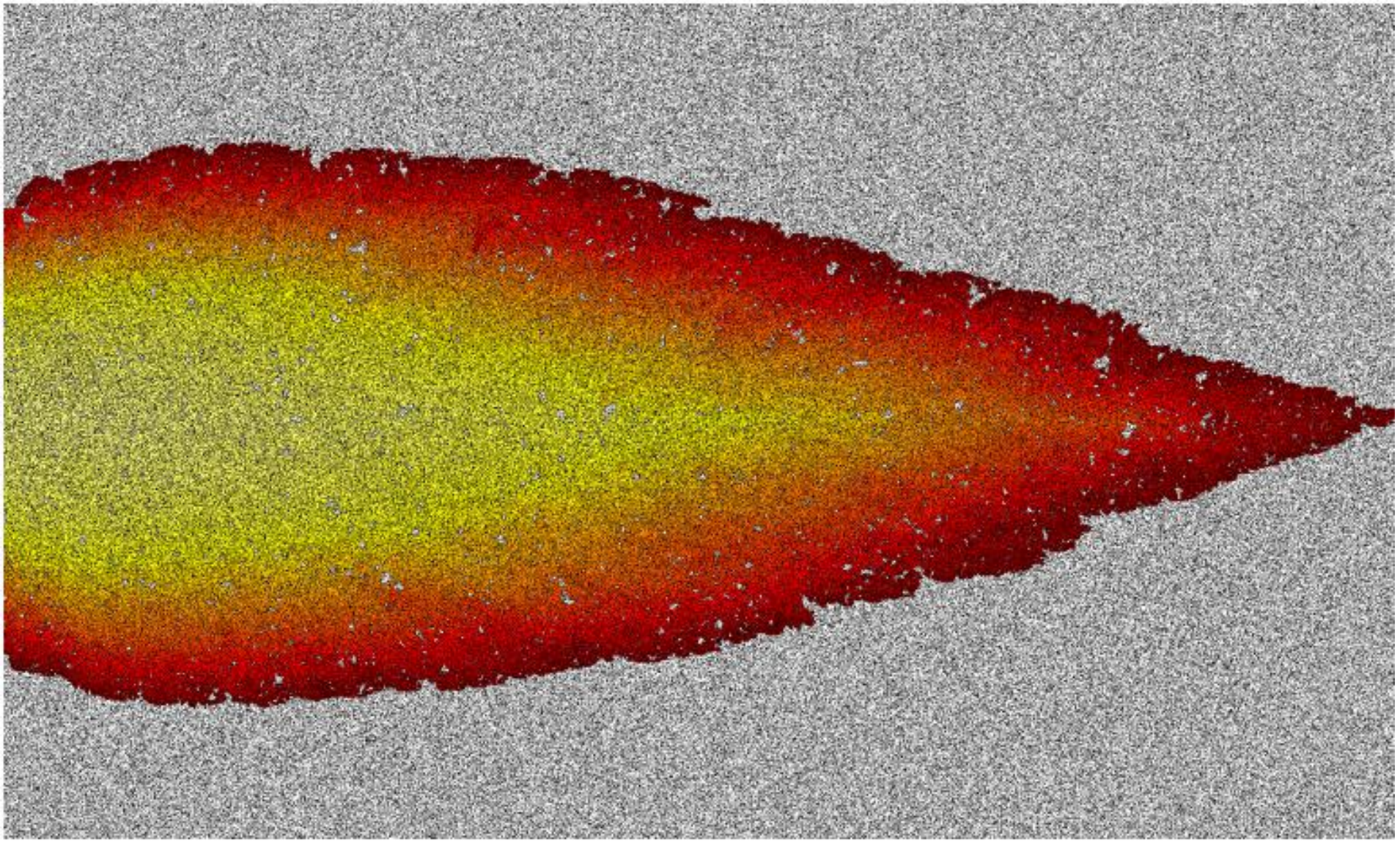
function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if *fringe* is empty **then return** failure
  node ← REMOVE-FRONT(*fringe*)
  if GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
end

Talk Outline

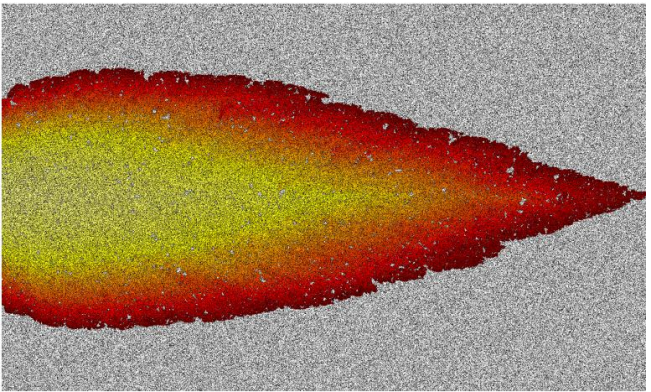
- Introduction
- Heuristic Search and Types
- An Algorithm: A*
- Fitting Domains To Interface
- Conclusion

Heuristic Search – A*



Heuristic Search – A*

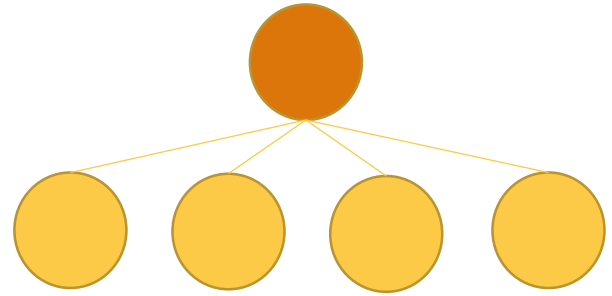
$f(n) = g(n) + h(n)$

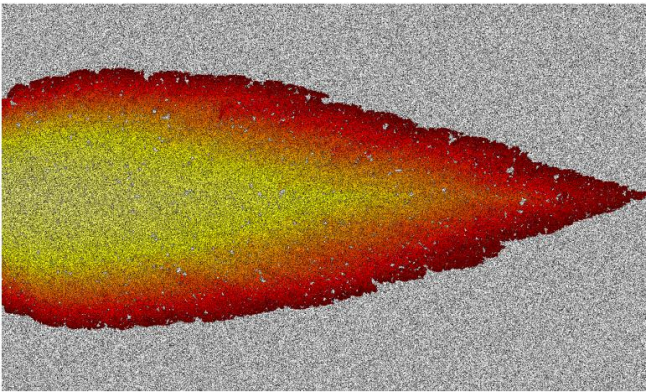


And so on and so forth…

Heuristic Search – A*

$f(n) = g(n) + h(n)$

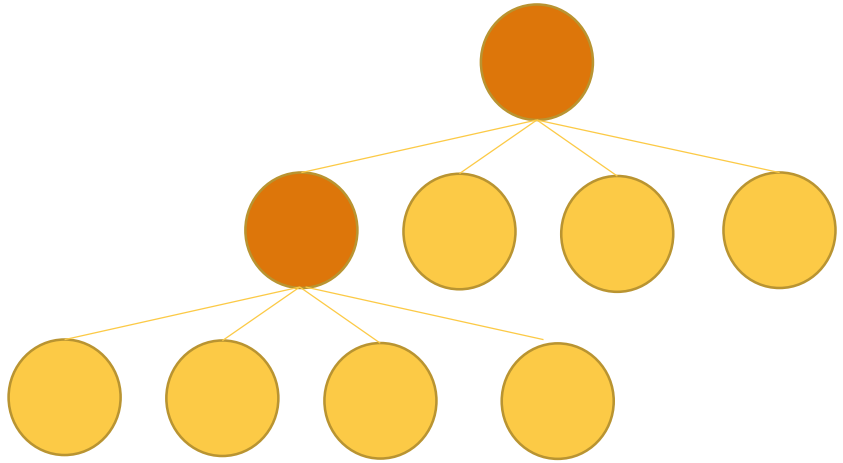


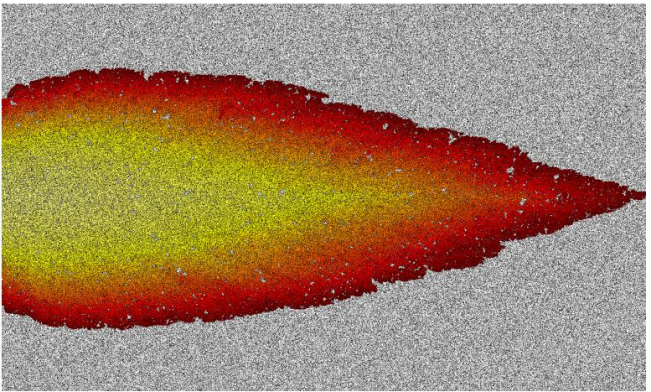


And so on and so forth…

Heuristic Search – A*

$f(n) = g(n) + h(n)$

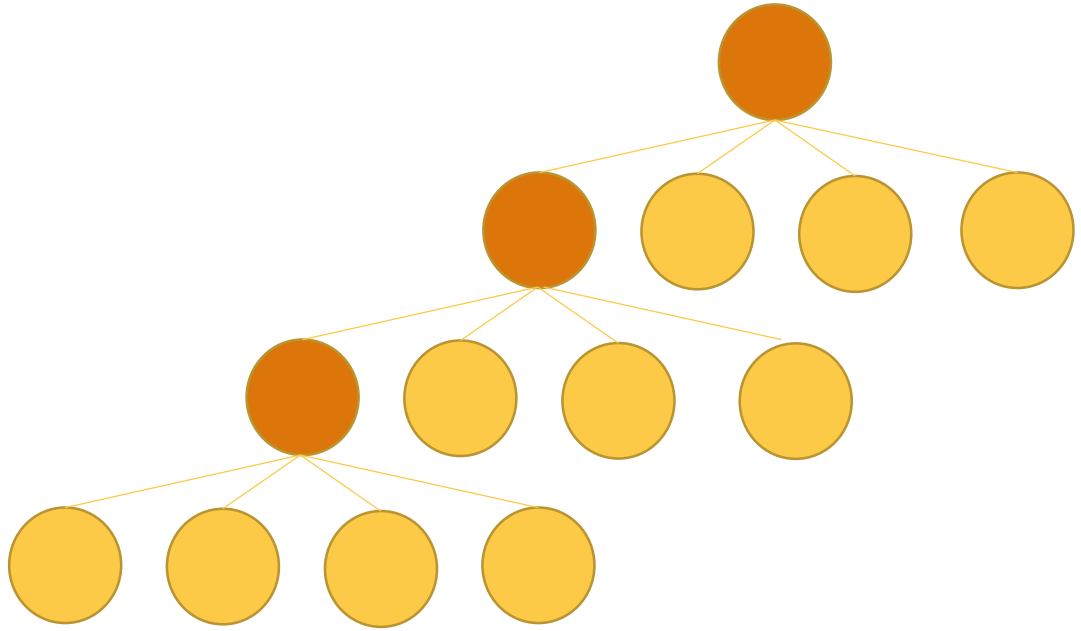


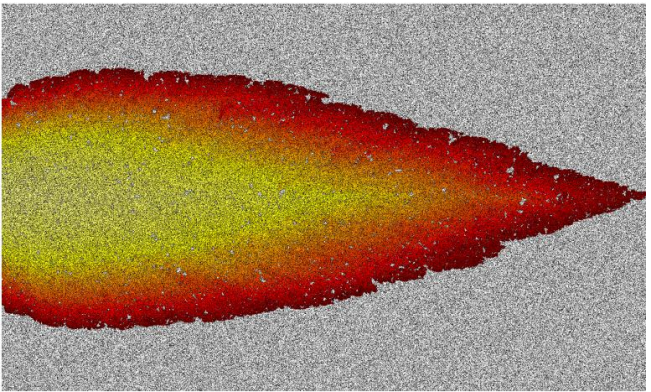


And so on and so forth…

Heuristic Search – A*

$f(n) = g(n) + h(n)$

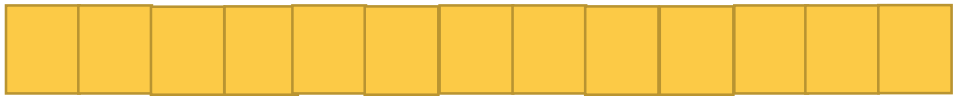


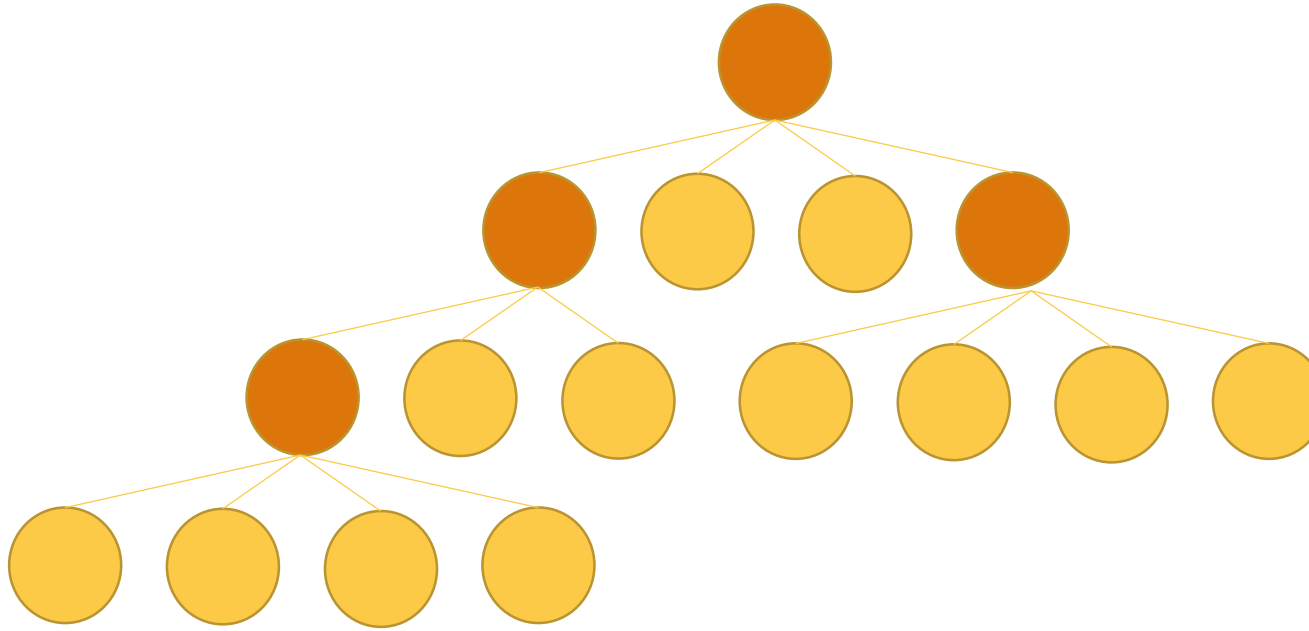


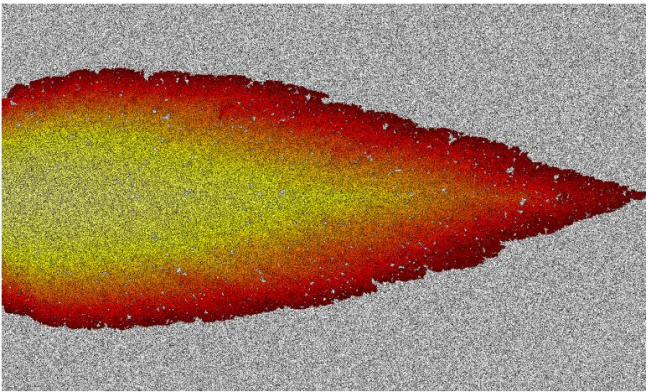
And so on and so forth…

Heuristic Search – A*

f(n) = g(n) + h(n)







And so on and so forth…

Heuristic Search – A*

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand | Goal | Heuristic | Initial | Solution

Graph search

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if *fringe* is empty **then return** failure
  node ← REMOVE-FRONT(*fringe*)
  if GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
end

Heuristic Search – A*

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand | Goal | Heuristic | Initial | Solution

```
82      while not finished do
83          match pop() with
84          | None -> finished <- true
85          | Some current_node ->
86              if node_goal_test current_node then begin
87                  metrics.solution_nodes <- { solution = current_node; found_at_time = DateTime.Now } :: metrics.solution_nodes;
88                  finished <- true end else
89                  begin
90                      let key_val = node_key current_node
91                      let should_expand = consider_node key_val current_node
92                      if should_expand then begin
93                          closedlist <- closedlist.Add(key_val, current_node)
94                          let child_tuples = node_expand current_node
95                          List.iter (consider_child current_node) child_tuples
96                      end
97                  end
98      { metrics with stop_time = Some DateTime.Now }
```

Heuristic Search – A*

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand | Goal | Heuristic | Initial | Solution

```
82      while not finished do
83          match pop() with
84          | None -> finished <- true
85          | Some current_node ->
86              if node_goal_test current_node then begin
87                  metrics.solution_nodes <- { solution = current_node; found_at_time = DateTime.Now } :: metrics.solution_nodes;
88                  finished <- true end else
89                  begin
90                      let key_val = node_key current_node
91                      let should_expand = consider_node key_val current_node
92                      if should_expand then begin
93                          closedlist <- closedlist.Add(key_val, current_node)
94                          let child_tuples = node_expand current_node
95                          List.iter (consider_child current_node) child_tuples
96                      end
97                  end
98      { metrics with stop_time = Some DateTime.Now }
```

Heuristic Search – A*

Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

```
82      while not finished do
83          match pop() with
84          | None -> finished <- true
85          | Some current_node ->
86              if node_goal_test current_node then begin
87                  metrics.solution_nodes <- { solution = current_node; found_at_time = DateTime.Now } :: metrics.solution_nodes;
88                  finished <- true end else
89                  begin
90                      let key_val = node_key current_node
91                      let should_expand = consider_node key_val current_node
92                      if should_expand then begin
93                          closedlist <- closedlist.Add(key_val, current_node)
94                          let child_tuples = node_expand current_node
95                          List.iter (consider_child current_node) child_tuples
96                      end
97                  end
98      { metrics with stop_time = Some DateTime.Now }
```

Heuristic Search – A*

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Expand | Goal | Heuristic | Initial | Solution

```
82      while not finished do
83          match pop() with
84          | None -> finished <- true
85          | Some current_node ->
86              if node_goal_test current_node then begin
87                  metrics.solution_nodes <- { solution = current_node; found_at_time = DateTime.Now } :: metrics.solution_nodes;
88                  finished <- true end else
89                  begin
90                      let key_val = node_key current_node
91                      let should_expand = consider_node key_val current_node
92                      if should_expand then begin
93                          closedlist <- closedlist.Add(key_val, current_node)
94                          let child_tuples = node_expand current_node
95                          List.iter (consider_child current_node) child_tuples
96                      end
97                  end
98      { metrics with stop_time = Some DateTime.Now }
```

Heuristic Search – A*

```
53  let astar_search (iface : DomainInterface.CostHeuristicDuplicateDomainInterface<float, 'state, 'hashvalue>) =
54      let mutable openlist : FSharpx.Collections.IPriorityQueue<'state SearchNode> = FSharpx.Collections.PriorityQueue.empty false
55      let mutable closedlist : FSharpx.Collections.PersistentHashMap<'hash_value, 'state SearchNode> = FSharpx.Collections.PersistentHashMap.empty
56      let root = make_root iface.H iface.InitialState
57      let metrics = initial_metrics ()
58      let node_key = wrap_state_fn iface.Key
59      let node_expand = wrap_state_fn iface.Expand
60      let node_goal_test = wrap_state_fn iface.GoalP
61      let enqueue (node : 'state SearchNode) = openlist <- FSharpx.Collections.PriorityQueue.insert node openlist
62      let consider_child current_node (state, step_cost) =...
67      let pop () =...
71      let consider_node key_val current_node =...
79      enqueue root
80      let mutable finished = false
81      while not finished do
82          match pop() with
83          | None -> finished <- true
84          | Some current_node ->
85              if node_goal_test current_node then begin
86                  metrics.solution_nodes <- { solution = current_node; found_at_time = DateTime.Now } :: metrics.solution_nodes;
87                  finished <- true end else
88                  begin
89                      let key_val = node_key current_node
90                      let should_expand = consider_node key_val current_node
91                      if should_expand then begin
92                          closedlist <- closedlist.Add(key_val, current_node)
93                          let child_tuples = node_expand current_node
94                          List.iter (consider_child current_node) child_tuples
95                      end
96                  end
97      { metrics with stop_time = Some DateTime.Now }
```

Talk Outline

- Introduction
- An Algorithm: A*
- Fitting Domains To Interface
  - Grid Navigation
  - String Edit Distance
- Conclusion

Heuristic Search – What is a Domain?

```
1   module DomainInterface
2
3   type DomainInterface<'cost,  'state> =
4       abstract member Expand : 'state -> ('state * 'cost) list
5       abstract member GoalP : 'state -> bool
6       abstract member InitialState : 'state
```

Heuristic Search – What is a Domain?

```
1   module DomainInterface
2
3   type DomainInterface<'cost,  'state> =
4       abstract member Expand : 'state -> ('state * 'cost) list
5       abstract member GoalP : 'state -> bool
6       abstract member InitialState : 'state
7
8   type DuplicateDomainInterface<'cost, 'state, 'hashvalue> =
9       inherit DomainInterface<'cost, 'state>
10      abstract member Key : 'state -> 'hashvalue
11      abstract member DisplayKey : 'state -> string
```

Heuristic Search – What is a Domain?

```
1   module DomainInterface
2
3   type DomainInterface<'cost,  'state> =
4       abstract member Expand : 'state -> ('state * 'cost) list
5       abstract member GoalP : 'state -> bool
6       abstract member InitialState : 'state
7
8   type DuplicateDomainInterface<'cost, 'state, 'hashvalue> =
9       inherit DomainInterface<'cost, 'state>
10      abstract member Key : 'state -> 'hashvalue
11      abstract member DisplayKey : 'state -> string
12
13  type CostHeuristicDuplicateDomainInterface<'cost, 'state, 'hashvalue> =
14      inherit DuplicateDomainInterface<'cost, 'state, 'hashvalue>
15      abstract member H : 'state -> 'cost
```

Grid Navigation as a Domain

```
202  type GridDomainInterface(problem : Problem) =
203      interface DomainInterface.CostHeuristicDuplicateDomainInterface<float, State, int> with
204          member this.Expand s = expand problem.board s
205          member this.GoalP s = goal_test problem s
206          member this.Key s = key problem.board s
207          member this.H s = manhattan_distance problem s |> float
208          member this.InitialState = make_initial_state problem
```

Grid Navigation as a Domain

```
202    type GridDomainInterface(problem : Problem) =
203        interface DomainInterface.CostHeuristicDuplicateDomainInterface<float, State, int> with
204            member this.Expand s = expand problem.board s
205            member this.GoalP s = goal_test problem s
206            member this.Key s = key problem.board s
207            member this.H s = manhattan_distance problem s |> float
208            member this.InitialState = make_initial_state problem
```

```
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

Grid Navigation as a Domain

```
18  type Position = {
19      x : int
20      y: int
21  }
22
23  type Board = bool [,]
24
25  type Problem = {
26      start : Position
27      finish : Position
28      board : Board
29  }
```

Grid Navigation as a Domain

```
18  type Position = {
19      x : int
20      y: int
21  }
22
23  type Board = bool [,]
24
25  type Problem = {
26      start : Position
27      finish : Position
28      board : Board
29  }
```

```
38  type State =
39      {
40      position : Position;
41      generated_by : Action;
42      }
```

Grid Navigation as a Domain

```
3   type Action =
4   | North
5   | South
6   | East
7   | West
8   | Noop
9
10  let string_of_action = function
11  | North -> "North"
12  | South -> "South"
13  | East -> "East"
14  | West -> "West"
15  | Noop -> ""
```

Grid Navigation as a Domain

```
145 let opposite_action = function
146     | North -> South
147     | South -> North
148     | East -> West
149     | West -> East
150     | Noop -> Noop
151
152 let are_opposite (act1 : Action) (act2 : Action) =
153     act1 = opposite_action act2
154
155 let move (position : Position) = function
156     | North -> { position with y = position.y - 1 }
157     | South -> { position with y = position.y + 1 }
158     | East ->  { position with x = position.x + 1 }
159     | West ->  { position with x = position.x - 1 }
160     | Noop -> position
161
162 let move_state (state : State) action =
163     { position = move state.position action; generated_by = action }
164
165 let expand (board : Board) (state : State) =
166     let possible_actions = [North; South; East; West]
167     let validate_position = legal_position board
168     let consider_action (accum : (State * float) list) (action : Action) =
169         if are_opposite action state.generated_by then accum else
170         let state' = move_state state action
171         if validate_position state'.position then
172                 (state', 1.) :: accum
173             else
174                 accum
175     List.fold consider_action [] possible_actions
```

Grid Navigation as a Domain

```
177  let key (board : Board) (state : State) =
178      let width = board.GetLength 1
179      state.position.x + state.position.y * width
180
181  let make_initial_state (problem : Problem) = {
182      position = problem.start;
183      generated_by = Noop
184  }
185
186  let goal_test (problem : Problem) (state : State) =
187      problem.finish = state.position
188
189  let manhattan_distance (problem : Problem) (state : State) =
190      abs(problem.finish.x - state.position.x) + abs(problem.finish.y - state.position.y)
191
```

Grid Navigation as a Domain

```
202  type GridDomainInterface(problem : Problem) =
203      interface DomainInterface.CostHeuristicDuplicateDomainInterface<float, State, int> with
204          member this.Expand s = expand problem.board s
205          member this.GoalP s = goal_test problem s
206          member this.Key s = key problem.board s
207          member this.H s = manhattan_distance problem s |> float
208          member this.InitialState = make_initial_state problem
```

Talk Outline

- Introduction
- An Algorithm: A*
- Fitting Domains To Interface
  - Grid Navigation
  - String Edit Distance
- Conclusion

String Edit Distance – You Can't Get There From Here

HERE

THERE

String Edit Distance - You Can't Get There From Here

HERE_

THERE

String Edit Distance - You Can't Get There From Here

HERE_

_HERE

THERE

String Edit Distance - You Can't Get There From Here

HERE_

_HERE

THERE

THERE

String Editing as a Domain

```
362    type StringDomainInterface(problem : Problem) =
363        interface DomainInterface.CostHeuristicDuplicateDomainInterface<float, State, string> with
364            member this.Expand s = expand problem s
365            member this.GoalP s = goal_test problem s
366            member this.Key s = key s
367            member this.DisplayKey s = string_of_element_array s.state
368            member this.H s = character_delta problem s
369            member this.InitialState = make_initial_state problem
```

```
52  type HeuristicSearchProblem<'state, 'action when 'state: comparison and 'action: comparison > = {
53      initial_state : 'state
54      goal_p : 'state -> bool
55      expand : 'state -> Set<Edge<'state, 'action>>
56      heuristic : 'state -> float
57  }
58
59  type HeuristicSearchAlgorithm<'state, 'action when 'state: comparison and 'action: comparison> =
60      ('state -> Set<Edge<'state, 'action>>) -> ('state -> bool) -> ('state -> float) -> 'state -> Edge<'state, 'action> list
```

String Editing as a Domain

```
33  type Element =
34  | Alphabetical of Character
35  | Null
36
37  type Update = {
38      index: int
39      character: Character
40  }
41
42  type Action =
43  | Remove of int
44  | Add of Update
45  | Replace of Update
46  | ShiftLeft
47  | ShiftRight
48  | Noop
49
50  type State = {
51      state: Element array
52      generated_by : Action
53  }
54
55  type Problem = {
56      start: Element array
57      finish: Element array
58  }
```

String Editing as a Domain

```
318  let goal_test (instance : Problem) (state : State) =
319      state.state = instance.finish
320
321  let key (state : State) =
322      string_of_element_array state.state
323
324  let make_initial_state (problem : Problem) =
325      {
326          state = problem.start
327          generated_by = Noop
328      }
329
330  let character_delta (problem : Problem) (state : State) =
331      let char_count = function
332      | Null -> 0
333      | _ -> 1
334      let problem_count = Array.fold (fun accum el -> accum + char_count el) 0 problem.finish
335      let state_count = Array.fold (fun accum el -> accum + char_count el) 0 state.state
336      let delta = problem_count - state_count
337      if delta < 0 then float delta * ADD_COST
338      else float delta * DEL_COST
```

String Editing as a Domain

```
298  let expand (problem : Problem) (state : State) =
299      let possible_indices = [0..state.state.Length - 1]
300      let possible_characters = appearing_in problem.finish
301      let possible_updates =
302          List.fold (fun accum index ->
303              List.fold (fun accum2 character -> { index = index; character = character } :: accum2) accum possible_characters) [] possible_indices
304
305      let possible_shifts = [ShiftLeft; ShiftRight]
306      let possible_removes = List.map Remove possible_indices
307      let possible_adds = List.map Add possible_updates
308      let possible_replaces = List.map Replace possible_updates
309
310      let generate_successor (accum : (State*float) list) (action : Action) =
311          if valid_move state action then
312              (apply_nondestructive state action, cost state action) :: accum
313          else
314              accum
315      let possible_actions = possible_shifts @ possible_removes @ possible_adds @ possible_replaces
316      List.fold generate_successor [] possible_actions
```

Talk Outline

- Introduction
- An Algorithm: A*
- Fitting Domains To Interface
  - Grid Navigation
  - String Edit Distance
- Conclusion

Talk Outline

- Introduction
- An Algorithm: A*
- Fitting Domains To Interface
- Conclusion
  - Functional programming is a natural fit for search code
    - Passing in Expand, Goal, Heuristic Functions
    - Currying & Partial Function Application
  - Types make separation of domain, solver easy

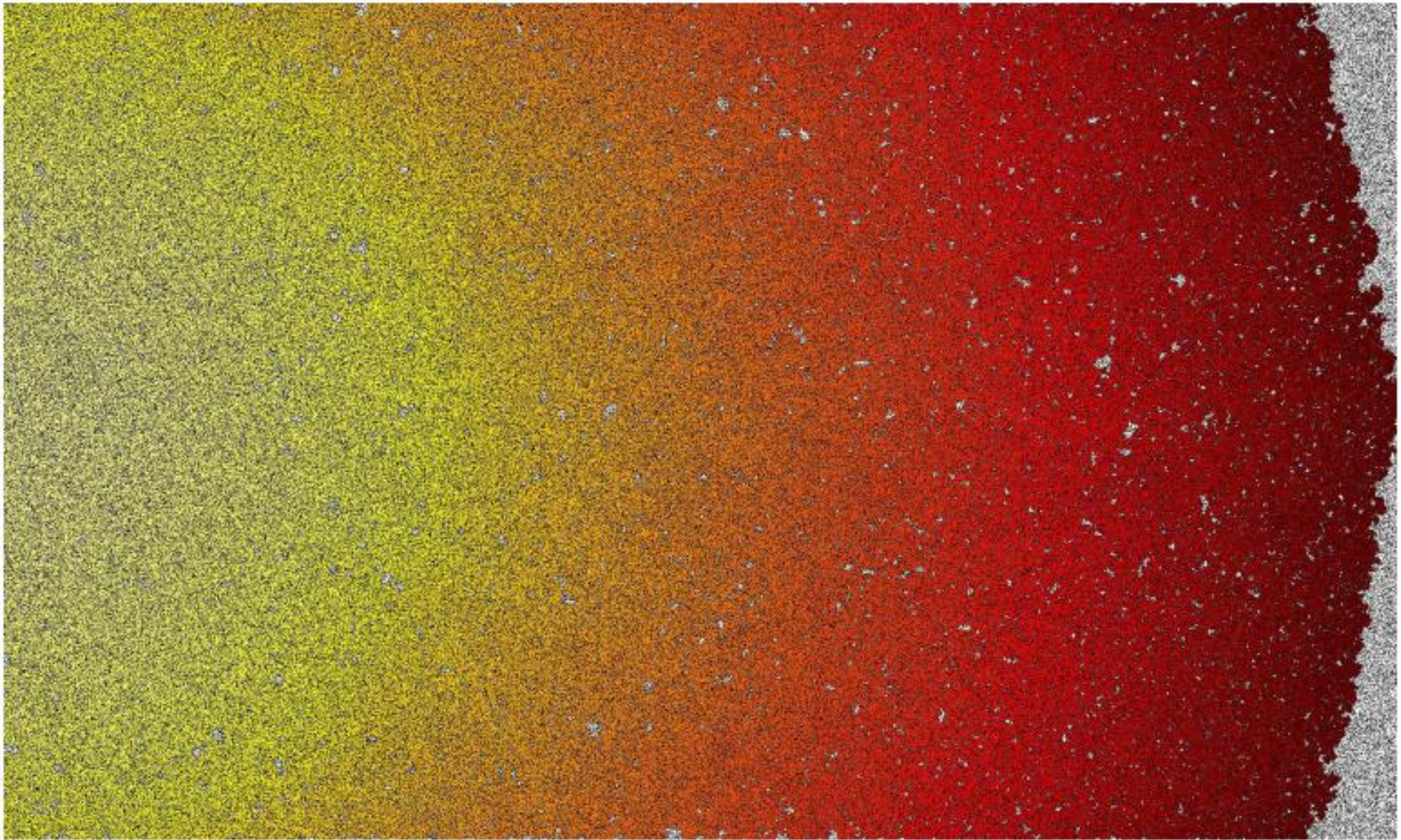
Heuristic Search: Type Perspective

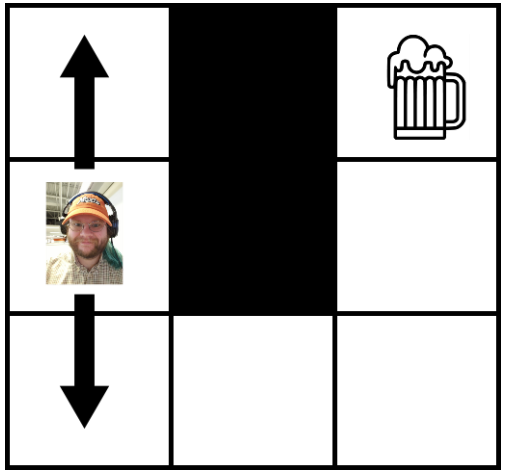
```
62  [<CustomEquality; CustomComparison>]
63  type Action<'T when 'T: comparison> =
64      {
65          cost : float
66          id : 'T
67      }
68      override x.GetHashCode() = // unit -> int
70      override x.Equals(yobj) = // obj -> bool
74      interface System.IComparable with
75          member x.CompareTo yobj = // obj -> int
```

Talk Outline

- Introduction
  - Grid Navigation Problem Setup
  - Example Domains - Toys
  - Example Domains - Industrial
- An Algorithm: A*
- Fitting Domains To Interface
  - Grid Navigation
  - String Edit Distance
- Conclusion

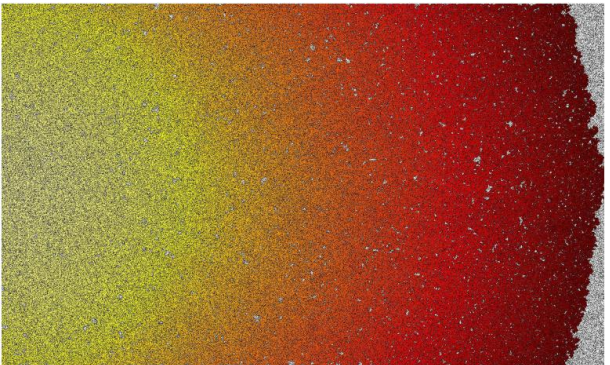
Heuristic Search





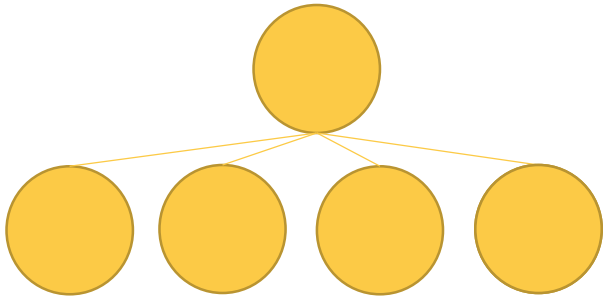
Heuristic Search – Uniform Cost Search

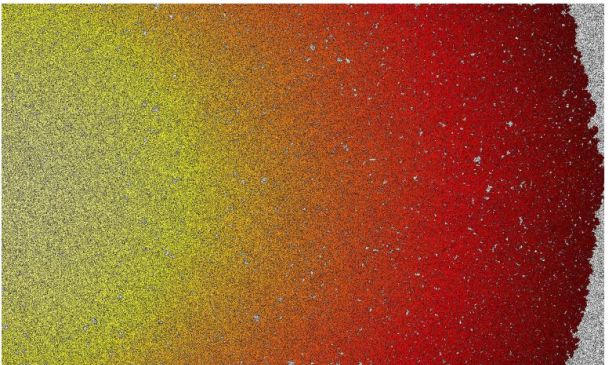
g(n)



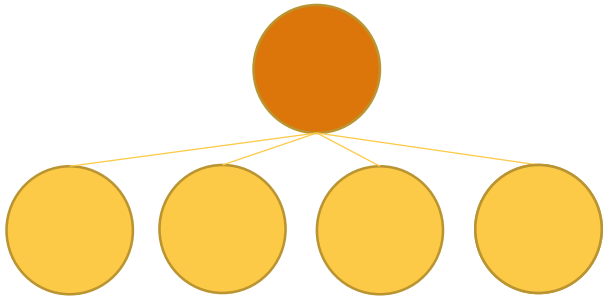
Heuristic Search – Uniform Cost Search

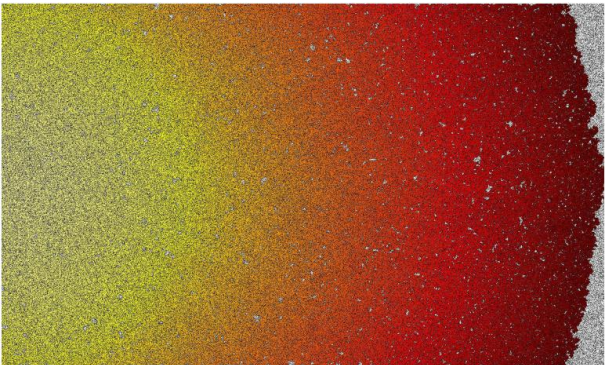
g(n)





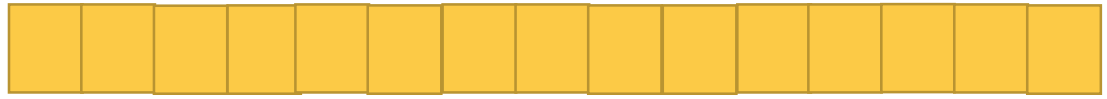
Heuristic Search – Uniform Cost Search

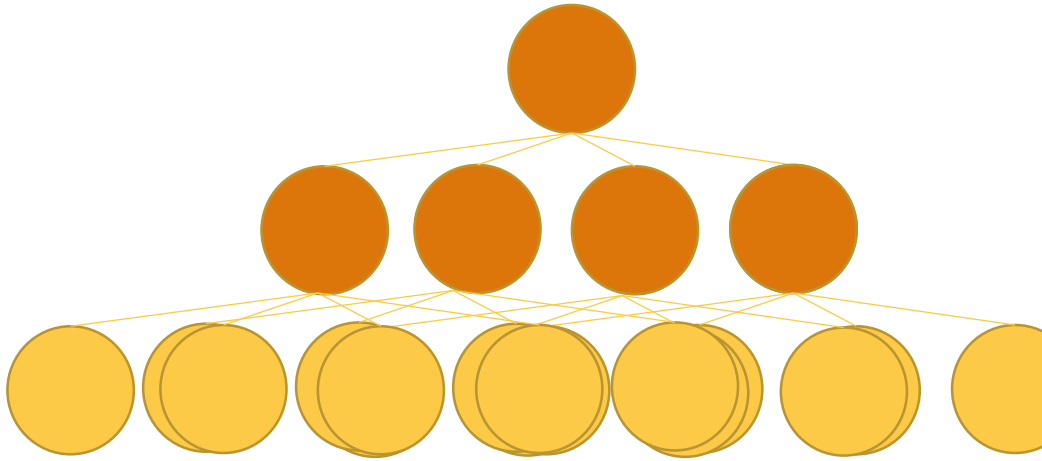




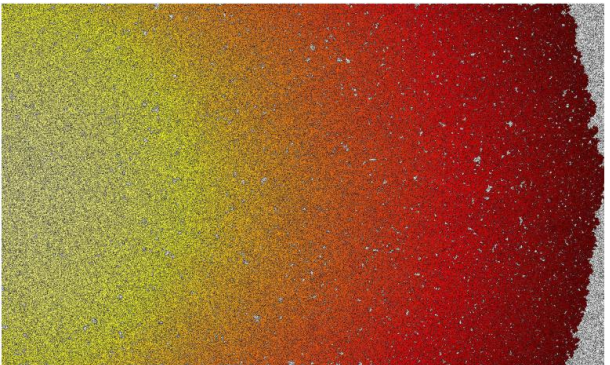
Heuristic Search – Uniform Cost Search

g(n)



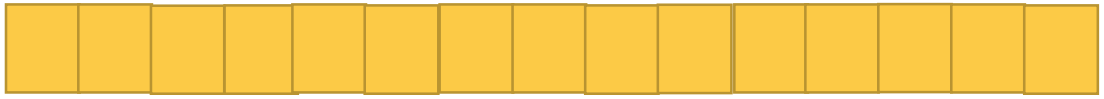


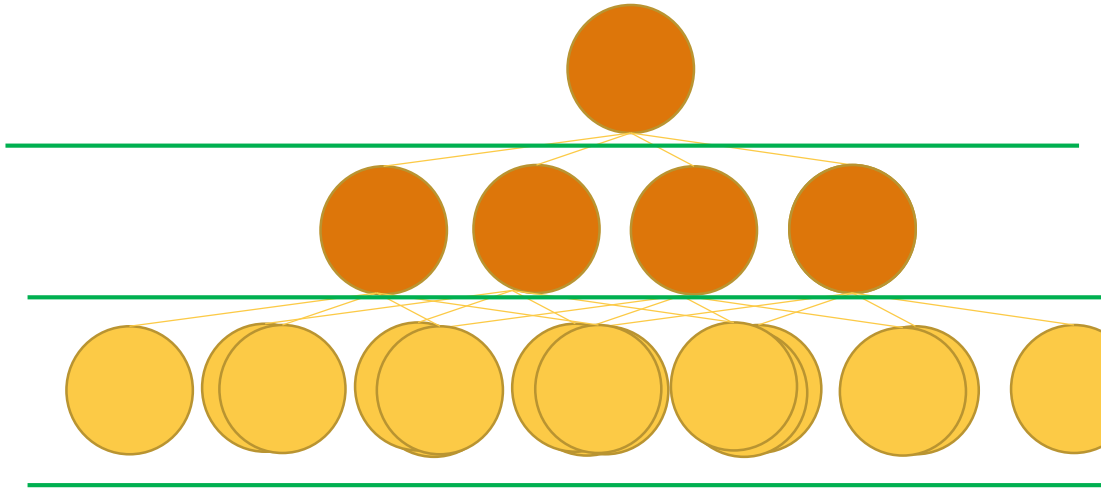
And so on and so forth…



Heuristic Search – Uniform Cost Search

g(n)





And so on and so forth…