HILL-CLIMBING

JEFFREY L. POPYACK

IN SEARCH OF INTELLIGENCE I: HILL CLIMBING

Covered so far:

- AI Overview
- Production Systems
- Agents
- AI Programming (LISP + Python)

PRODUCTION SYSTEMS AND SEARCH

- Starting w/ an **Initial State**
- **Rules** that transform states
  - **Precondition:** determines if rule applicable
  - **Action:** apply rule to state
- Find path to a **state** that satisfies **goal (state)**
  - not necessarily a single state, but a predicate that tells us whether or not a given state satisfies the ***goal condition***

Knowledge Base	Rules	Search Strategy
Defines the problem domain State Representation Initial State Goal condition Global knowledge about problem	a.k.a. "Actions", "Moves" Transform state of system. **If(precondition)** ***can* apply action:** **state ← action(state)**	An algorithm that describes how we will find a path from Initial State to state satisfying goal condition (if such a path exists)

PRODUCTION SYSTEMS AND SEARCH

Instead of writing an algorithm that will solve the problem directly,

- use an algorithm that:
  - decides which of (potentially many) applicable rules should be applied next
  - knows how to recover if no rules are applicable

"FLAIL WILDLY" SEARCH STRATEGY

- Not intelligent
- But may work sometimes
- ..and may not work other times

"Fox, Goose, Corn" example:

- Many states revisited

```
======
state=[[ farmer fox corn ][ goose ]]
Choosing rule[2]=Move farmer and corn
from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose
from Right to Left
======
state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and goose
from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose
from Right to Left
======
state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and goose
from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[1]=Move farmer and corn
from Right to Left
======
```

"FLAIL WILDLY" SEARCH STRATEGY

- Not intelligent
- But may work sometimes
- ..and may not work other times

"Fox, Goose, Corn" example:

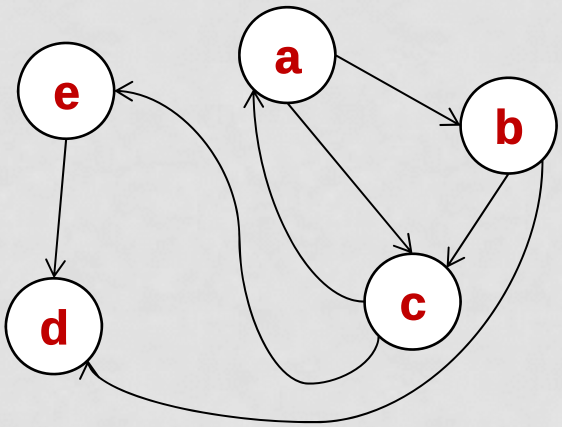
- Many states revisited
- **Step I: Quest for Intelligence**
  - **Reduce stupidity**
  - ***Don't revisit previous states !***

```
======
state=[[ farmer fox corn ][ goose ]]
Choosing rule[2]=Move farmer and corn
from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose
from Right to Left
======
state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and goose
from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose
from Right to Left
======
state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and goose
from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[1]=Move farmer and corn
from Right to Left
======
```

EXPLORING THE PROBLEM GRAPH

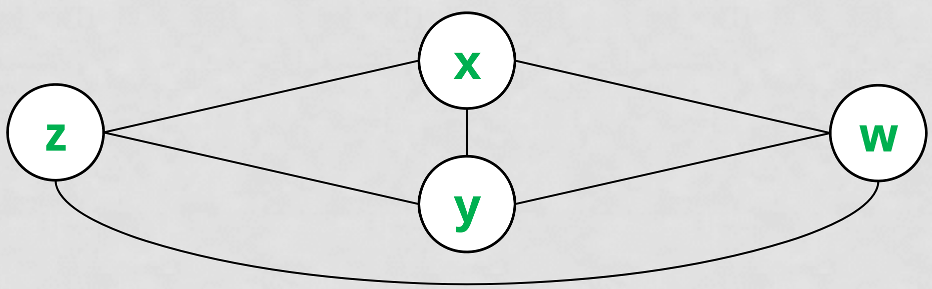
- Building a solution tree
  - Implicit Graph
  - Explicit Graph

EXPLICIT GRAPH



Explicit Graph:

- a set of nodes
- a set of edges connecting nodes
- ***Directed graph***:
  - edges have direction
  - **a** → **b** means you can go from **a** to **b**, but says nothing about whether you can go from **b** to **a**
- ***Undirected graph***:
  - edges do not have direction
  - **x** connected to **y** means you can go from **x** to **y**, or from **y** to **x**



EXPLICIT GRAPH

Find simplest route from **a→e**:

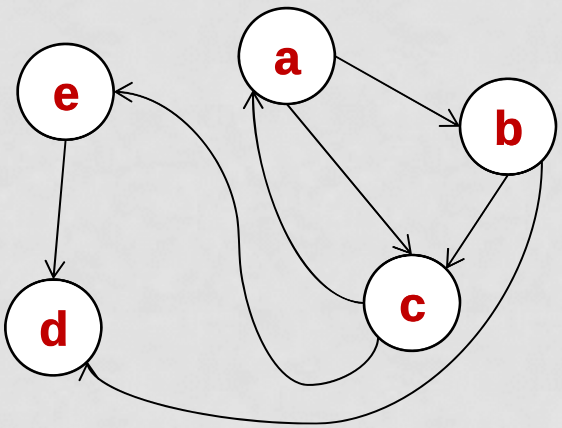
a → b → d **X**

a → b → c → e ☑

a → c → e ☑

a → c → a → c → a → c → e ☑

All nodes, edges known explicitly



IMPLICIT GRAPH

Implicit Graph:

- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:

```
[ farmer fox goose corn ] []
```

IMPLICIT GRAPH

Implicit Graph:

- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:

```
[ farmer fox goose corn ] []
```

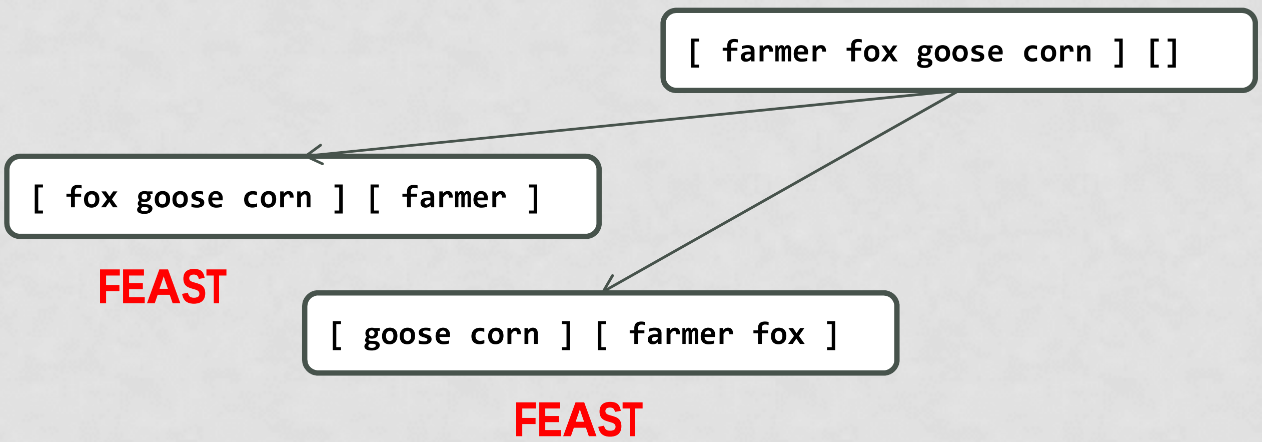
```
[ fox goose corn ] [ farmer ]
```

FEAST

IMPLICIT GRAPH

Implicit Graph:

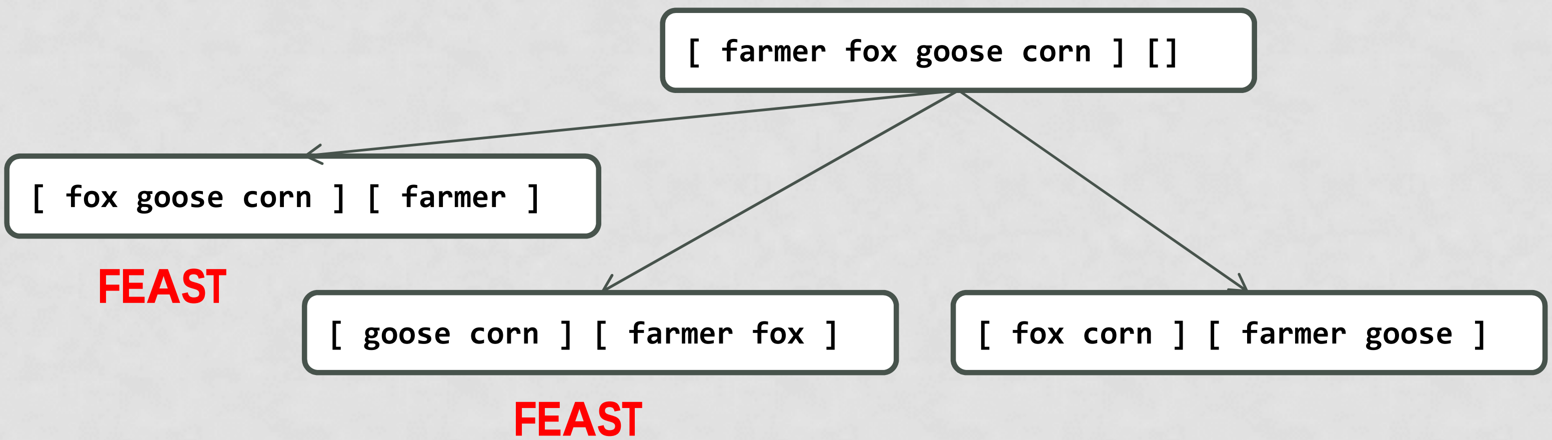
- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:



IMPLICIT GRAPH

Implicit Graph:

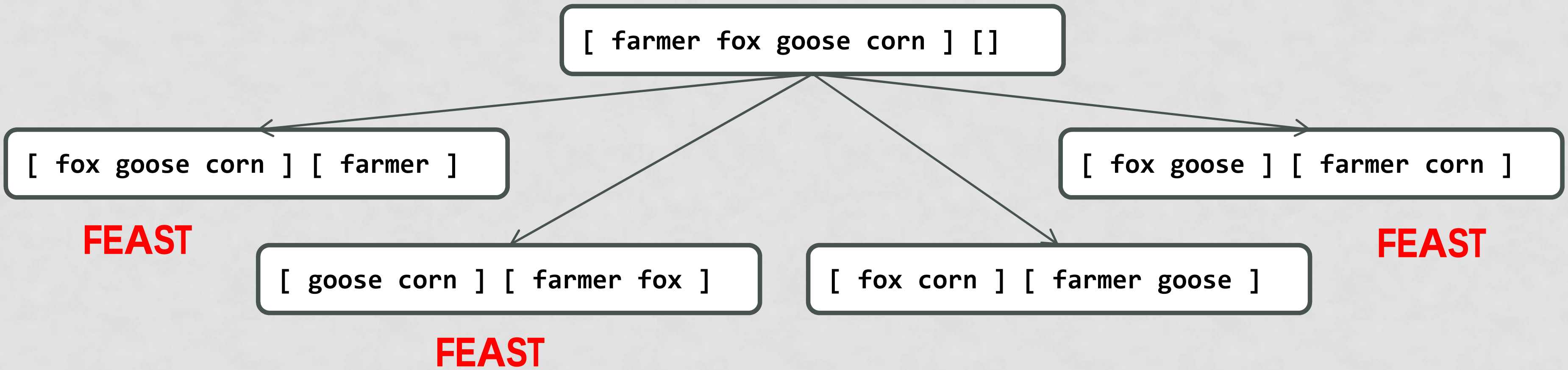
- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:



IMPLICIT GRAPH

Implicit Graph:

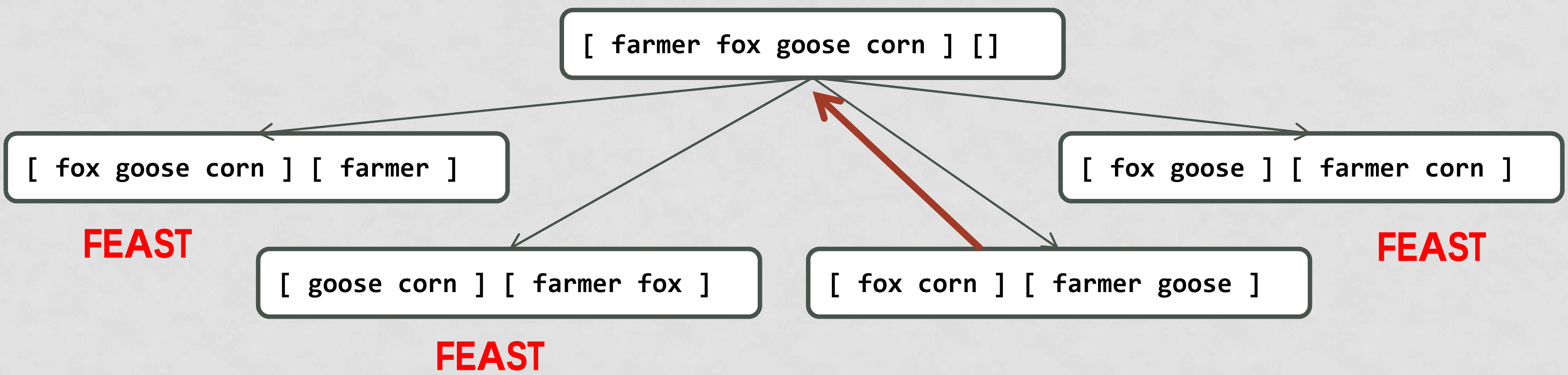
- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:



IMPLICIT GRAPH

Implicit Graph:

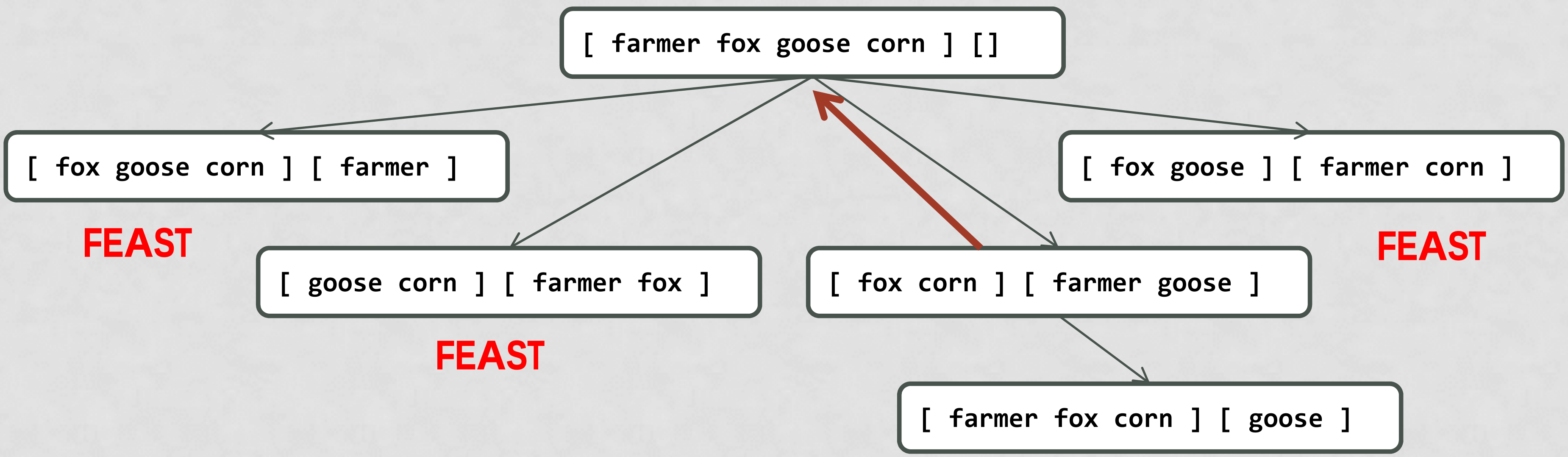
- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:



IMPLICIT GRAPH

Implicit Graph:

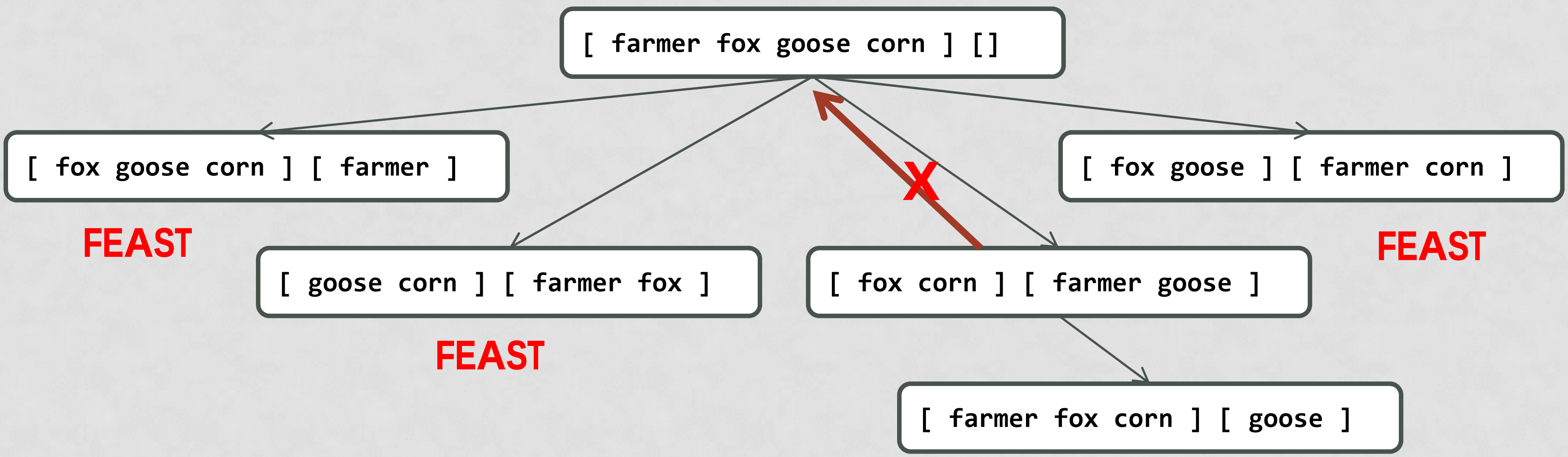
- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:



IMPLICIT GRAPH

Implicit Graph:

- Generated from initial node(s) and rules for generating successors.
- e.g., ***Fox-Goose-Corn*** problem:



- *Note*: solution space is a ***subset*** of a ***graph***
- That subset will be a ***tree*** (no node is its own ancestor)

SEARCH STRATEGIES

Tentative strategy

- Keep track of all moves from start.
- Don't "actually" apply a move - just “tentatively” apply it - and if it doesn't work, try something else.

When applying a rule in a tentative strategy, the “state” includes your partially constructed solution tree. (Knowledge Base includes this)

Irrevocable strategy

- Only keep track of current state - past states are irrelevant
- e.g. solving crossword puzzle in ink: can't "undo" a move:
  - it remains part of solution path, once tried

IDEA: "TENTATIVE" STRATEGY

Tentative Strategy:

- Continue building sub-graph until finding a node satisfying goal condition
- ***Note***: when expanding a node, can tell whether a new node ***n*** has
  - already been visited; or
  - already generated but not yet visited
- ***Note***: can be costly in terms of space & time
  - can "go back" to earlier states and try new directions
  - need storage space for nodes and links
  - may take too long to examine all previous nodes

IDEA: "IRREVOCABLE STRATEGY"

Irrevocable Strategy:

- Choose a move & make the move.
- Keep current state only.
- Not costly in space and time like tentative strategies
- ***Note***: Irrevocable strategies are generally less space-intensive and time-intensive at each step.
  - if a problem can be solved with an irrevocable strategy, it's probably good to do so

PRODUCTION SYSTEMS, REVISITED

Production Systems :

- Initial State
- Goal condition
- Rules:
  - precondition
  - action
- Knowledge Base
- ***Find a sequence of rules which can be applied to the Initial State, to find a state that satisfies the goal condition.***

PRODUCTION SYSTEMS, REVISITED

Production Systems :

- Initial State
- Goal condition
- Rules:
  - precondition
  - action
- Knowledge Base
- ***Find a sequence of rules which can be applied to the Initial State, to find a state that satisfies the goal condition.***

"Flailing Wildly"

```
state ← Initial State
while not goal (state):
    M ← applicable Rules (state)
    choose some m ε M
    state ← applyRule(m, state)
```

PRODUCTION SYSTEMS, REVISITED

Production Systems :

- Initial State
- Goal condition
- Rules:
  - precondition
  - action
- Knowledge Base
- ***Find a sequence of rules which can be applied to the Initial State, to find a state that satisfies the goal condition.***

"Flailing Wildly"

state ← **Initial State**
while not **goal** (state):

  M ← **applicable Rules** (state)

  choose some m ε **M**

  state ← **applyRule**(m, state)

How do we choose?

PRODUCTION SYSTEMS, REVISITED

- ***"Flailing Wildly"*** can find a solution if you're lucky/persistent (and if "undo" rules exists).

- **Commutative Production System**:
- Every state is reachable from every other state
  - Sufficient condition: every rule has an "undo", and the system cannot be decomposed into separate subsystems
  - "Undo" is not a *necessary* condition, though – if it is possible to cycle back to another state, for instance, it is not necessary to have an "undo" for each rule.

HILL-CLIMBING

Is there a way of preventing re-visiting a state ?

Hill-Climbing:

- Create a function **f()** that “measures” a state and a returns a single value in **R**.
- High value of **f()**: good state
- Low value of **f()**: bad state
- Only move in direction that improves value of **f()**
- can't revisit earlier state!
- may not always work ☹

HILL-CLIMBING

Hill-Climbing Strategy:

- Use a function **f(x)** that increases as solution is approached.
- Choose between moves by increasing **f(*state*)**

HILL-CLIMBING

Hill-Climb:

```
state ← Initial State
value ← f(state)
while not goal (state):
    M ← applicable Rules (state)
    for each  m ε M
        nextState ← applyRule (m, state)
        if f(nextState) > value
            value ← f(nextState )
            r ← m
    state ← applyRule(r, state)
```

HILL-CLIMBING

Hill-Climb:

```
state ← Initial State
value ← f(state)
while not goal (state):
    M ← applicable Rules (state)
    for each  m ε M
        nextState ← applyRule (m, state)
        if f(nextState) > value
            value ← f(nextState )
            r ← m
    state ← applyRule(r, state)
```

$$r = \arg\max_{m \in \mathbf{M}} \{ f(\text{applyRule}(m,\text{state})) \}$$

HILL-CLIMBING

Hill-Climb:

```
state ← Initial State
value ← f(state)
while not goal (state):
    M ← applicable Rules (state)
    for each m ε M
        nextState ← applyRule (m, state)
        if f(nextState) > value
            value ← f(nextState )
            r ← m
    state ← applyRule(r, state)
```

$$r = \arg\max_{m \,\varepsilon\, \mathbf{M}} \{ \mathbf{f}\,(\,\mathbf{applyRule(m,state)}\,) \}$$

Can't go backwards, revisit earlier state

Pick a rule that improves state (according to **f**(state)) and gives best improvement

***Note*: if no rule satisfies this, we're stuck.**

HILL-CLIMBING

Hill-Climb:

```
state ← Initial State
value ← f(state)
while not ~~goal (state)~~ stuck :
    M ← applicable Rules (state)
    for each  m ε M
        nextState ← applyRule (m, state)
        if f(nextState) > value
            value ← f(nextState )
            r ← m
    state ← applyRule(r, state)
```

$$r = \mathbf{arg} \max_{m \,\varepsilon\, \mathbf{M}} \{ \mathbf{f}\ \mathbf{(\ applyRule(m,state)}\ \}$$

Can't go backwards, revisit earlier state

Pick a rule that improves state (according to **f**(state)) and gives best improvement

***Note*: if no rule satisfies this, we're stuck.**

HILL-CLIMBING

Hill-Climb:

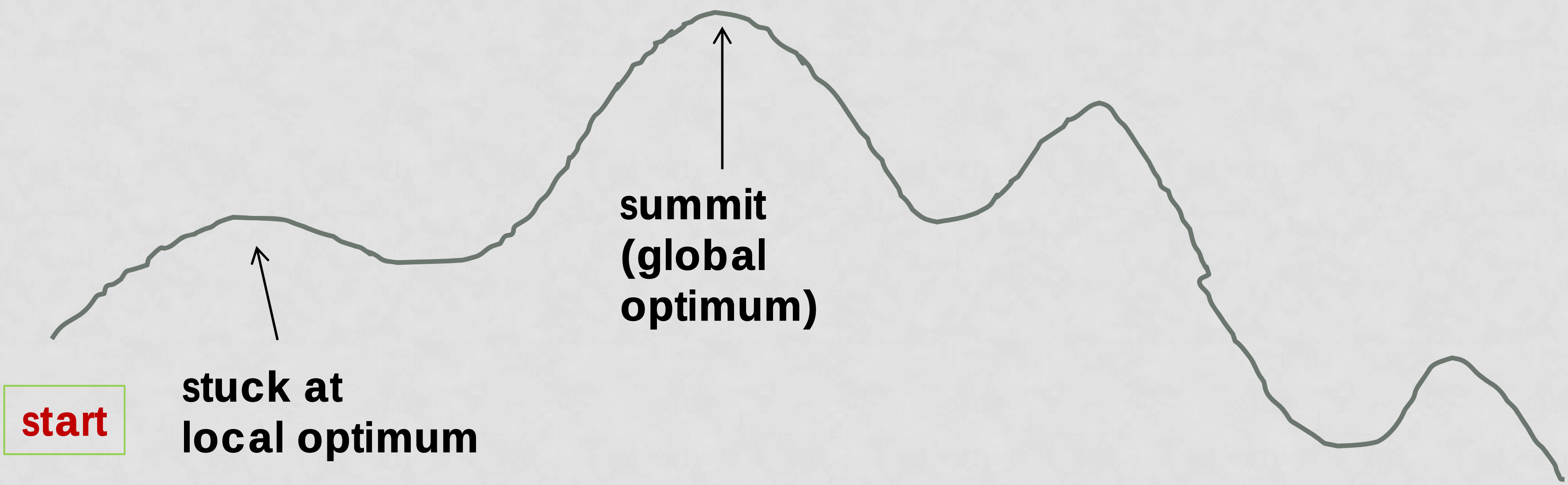
```
state ← Initial State
value ← f(state)
while not goal (state):
    M ← applicable Rules (state)
    for each  m ε M
        nextState ← applyRule (m, state)
        if f(nextState) > value
            value ← f(nextState )
            r ← m
    state ← applyRule(r, state)
```

Note: could modify so that we always choose the alternative providing highest value of **f** - even if it's lower than current point.

We won't get stuck at a local optimum, but we also won't stop at a global optimum, either.

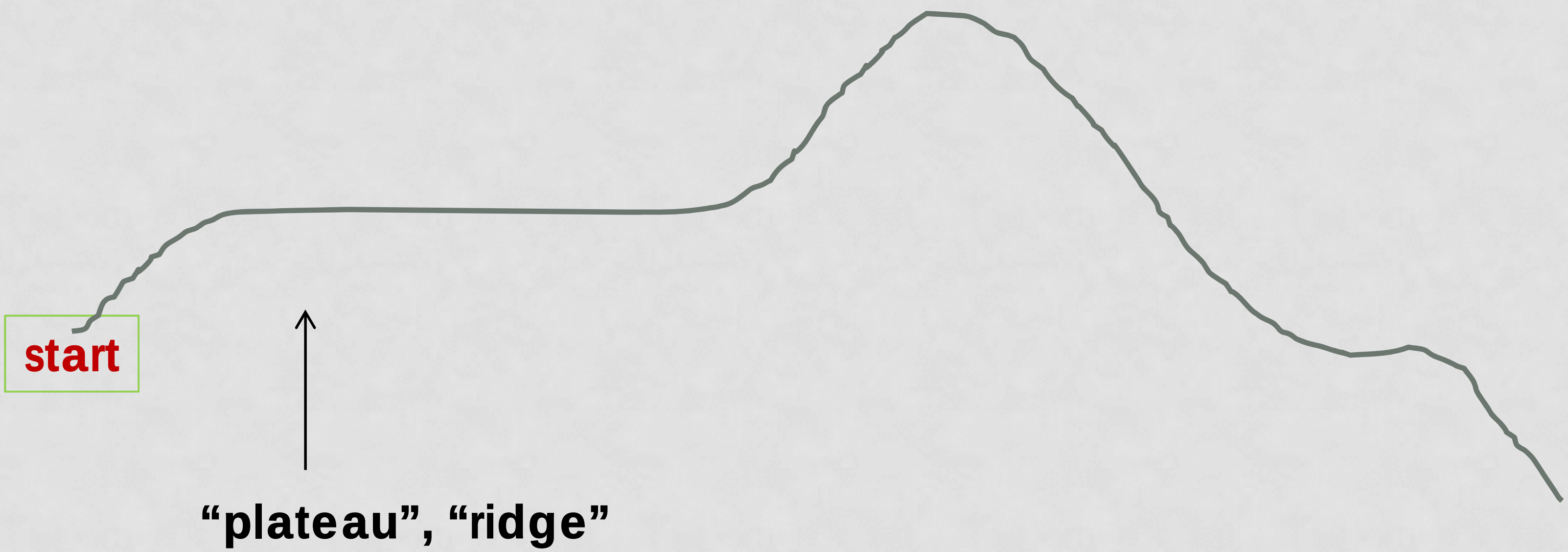
EXAMPLES

Mountain Climbing in Fog



EXAMPLES

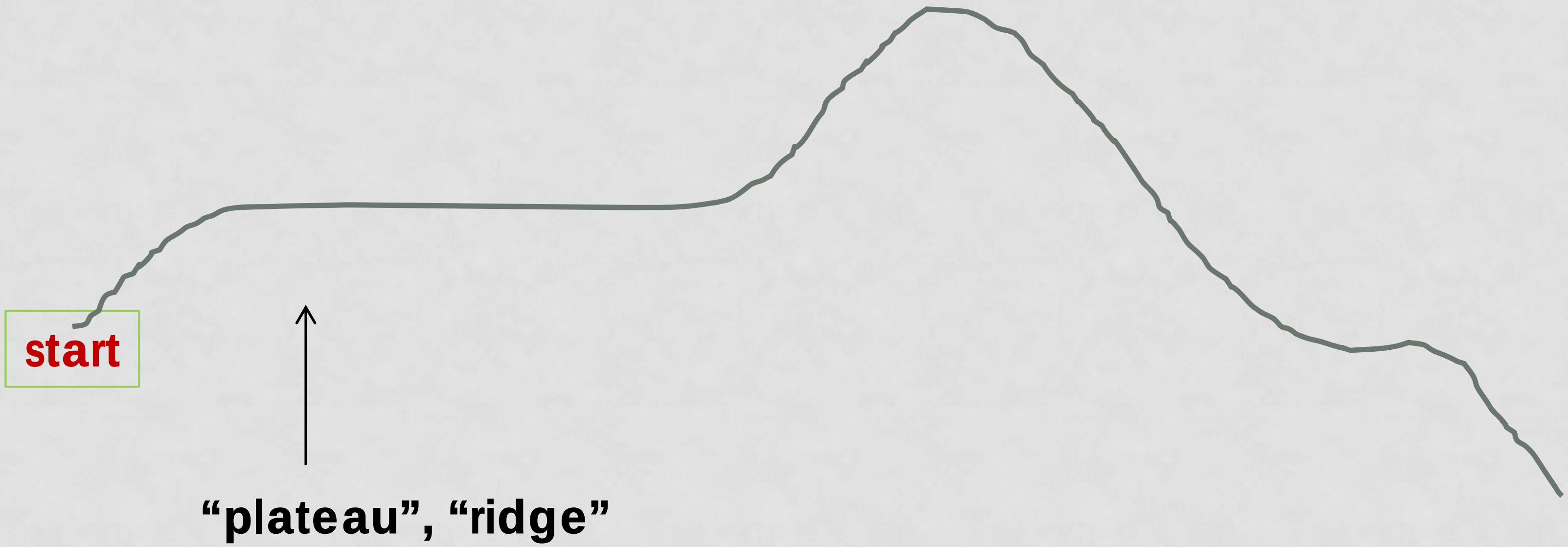
Mountain Climbing in Fog, II



EXAMPLES

Mountain Climbing in Fog, II

IDEA ... run algorithm several times with different starting points



EXAMPLES

- Adjusting multiple audio controls for sound quality
- Finding optimal set of weights for links in a neural net
- Evolving a population of candidate solutions in an evolutionary computing setting
- Swarm intelligence

EXAMPLES

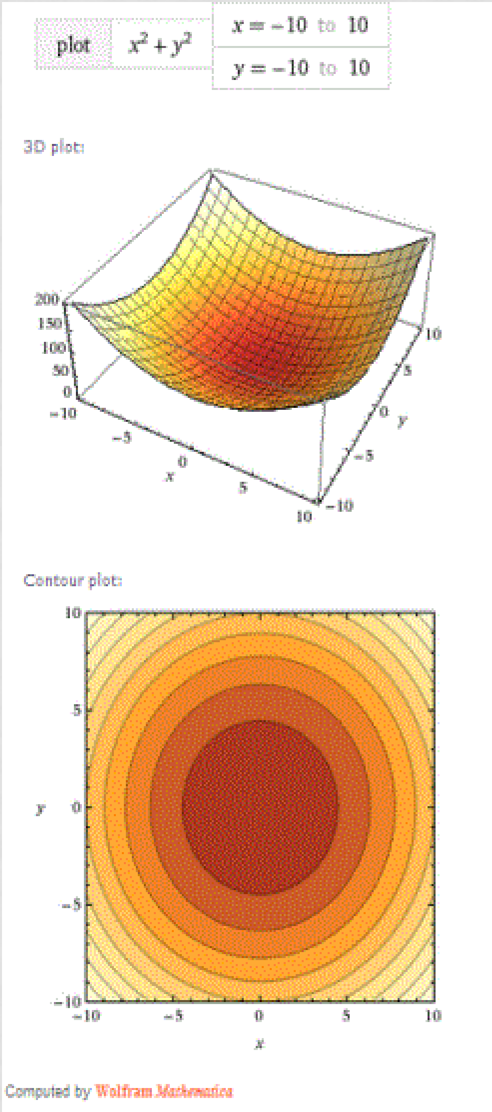
- Adjusting multiple audio controls for sound quality
- Finding optimal set of weights for links in a neural net
- Evolving a population of candidate solutions in an evolutionary computing setting
- Swarm intelligence

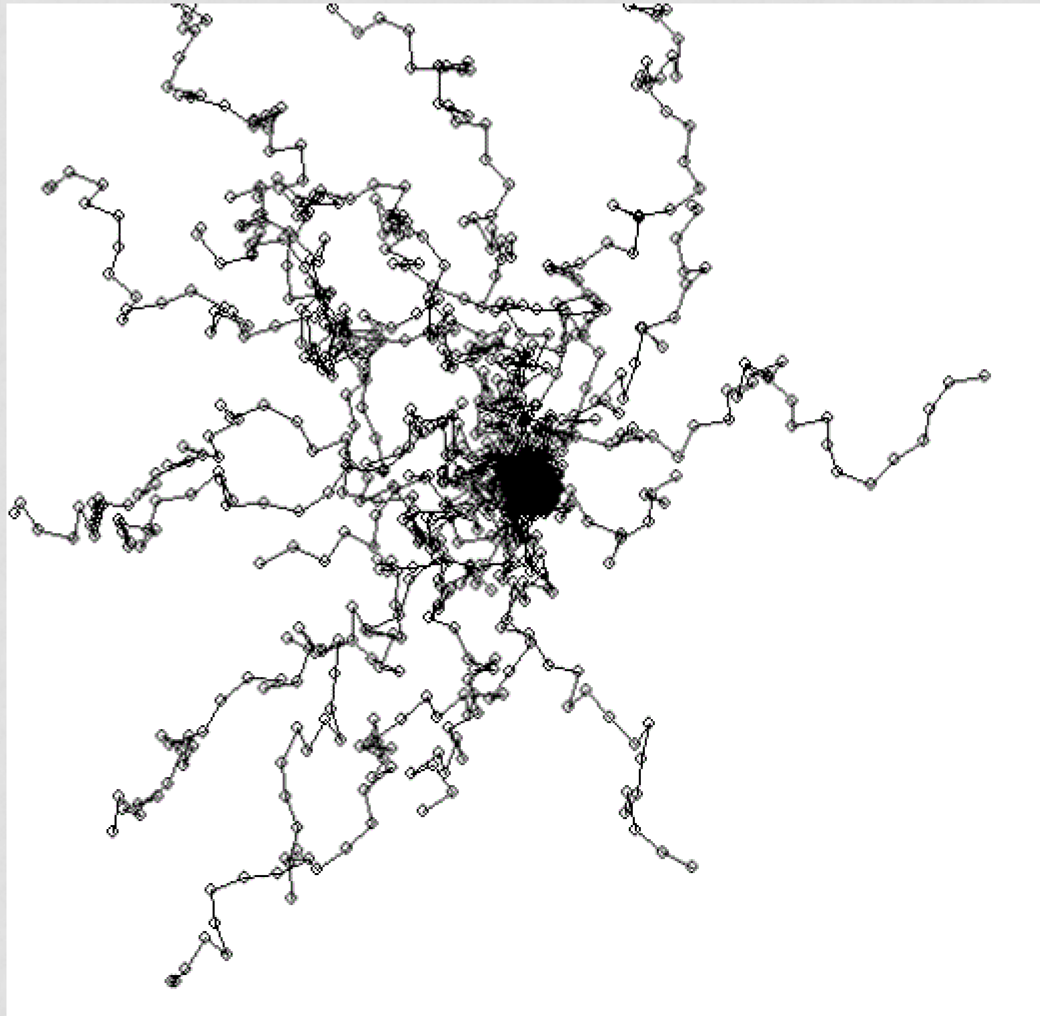
- Some other alternatives :
  - Simulated Annealing
  - Tabu search

Idea - It's ok to step backwards occasionally, as long as overall value of **f** improves over the long run.

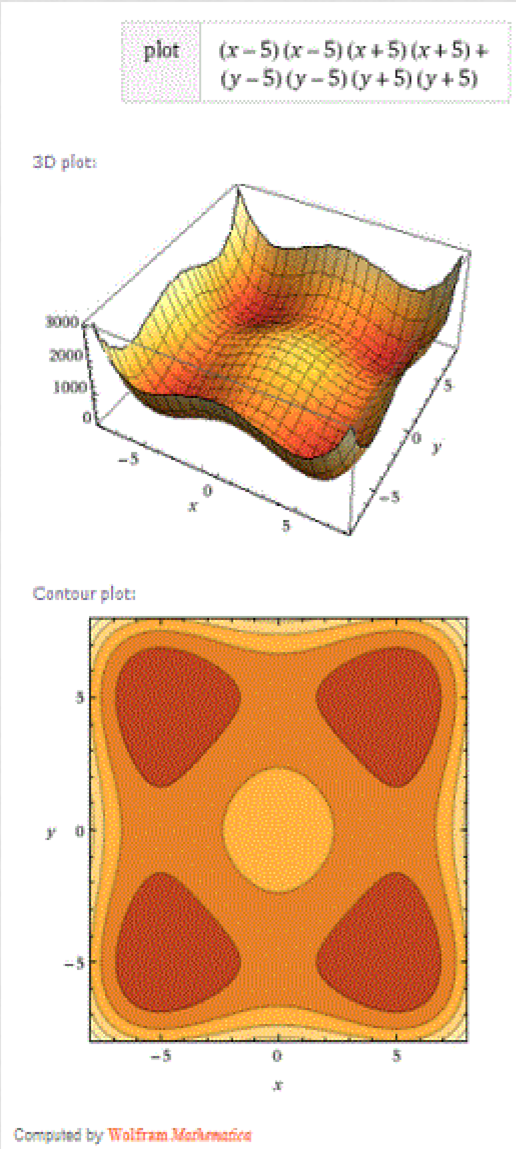
PARTICLE SWARM OPTIMIZATION

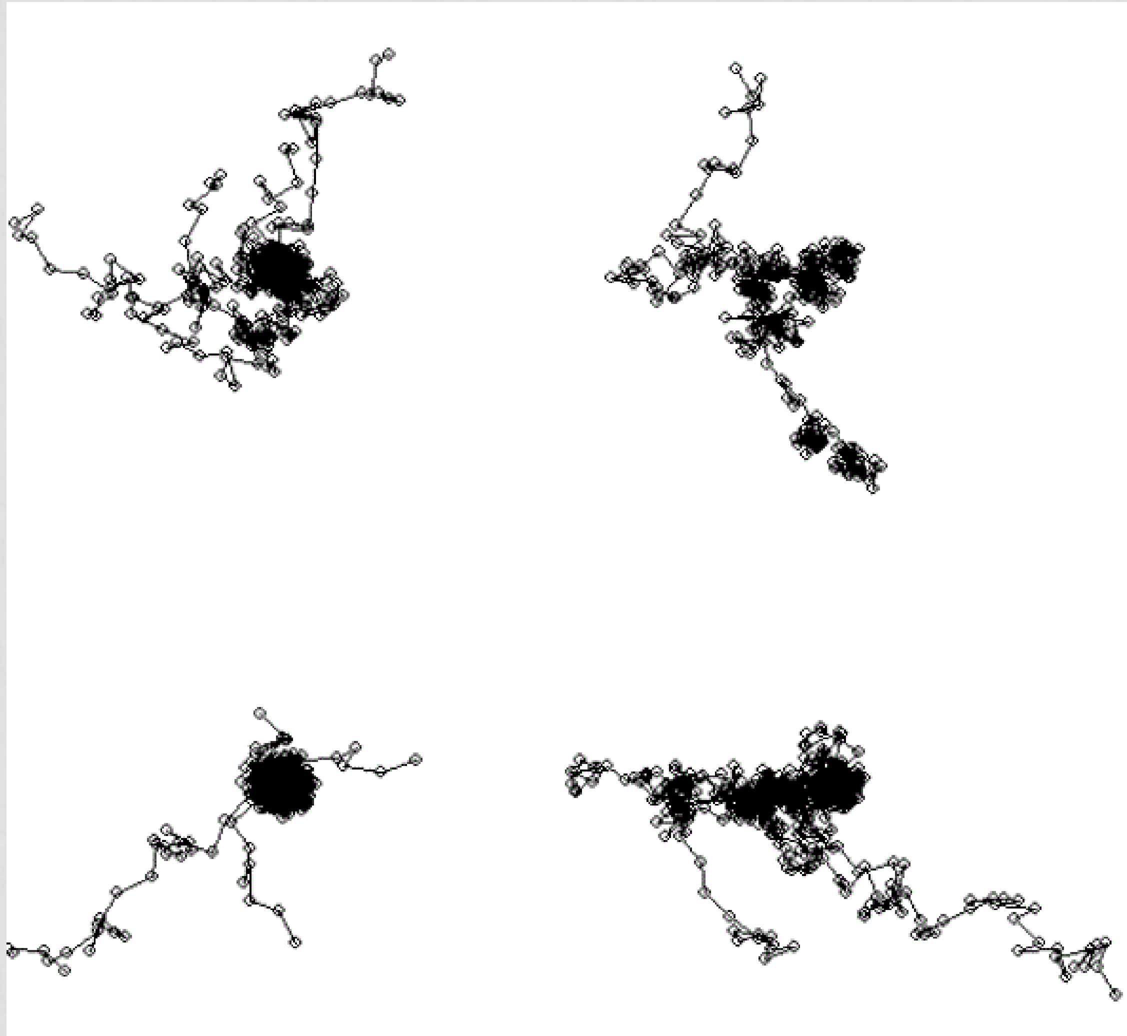
- generate **n** initial guesses $\mathbf{x_1, x_2, \ldots, x_n}$
- compute $\mathbf{f(x_1), f(x_2), \ldots, f(x_n)}$
- move in a random direction
  $\mathbf{x'_{\mathit{1}} = x_{\mathit{1}} + \theta_{\mathit{1}}, \ldots, x'_{\mathit{n}} = x_{\mathit{n}} + \theta_{\mathit{n}}}$
- compute $\mathbf{\{ f(x'_{\mathit{i}}) \}}$
- for each **i**, keep track of **pbest**$_i$ (that particle's best value)
- keep track of **gbest** (global best)
- move each $\mathbf{x'_{\mathit{i}}}$ towards **pbest**$_i$ and **gbest**
- ***particles tend to "swarm" to a best solution***



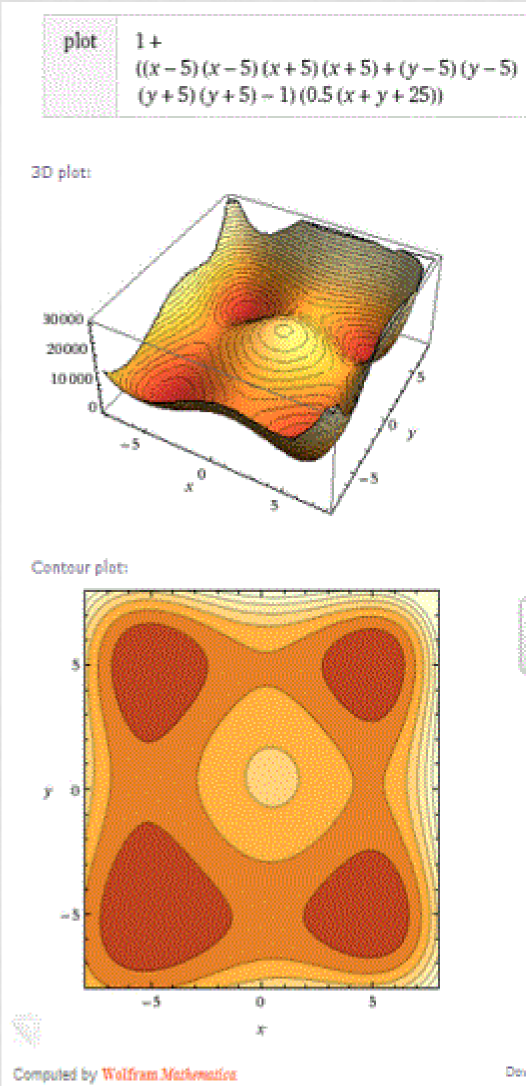


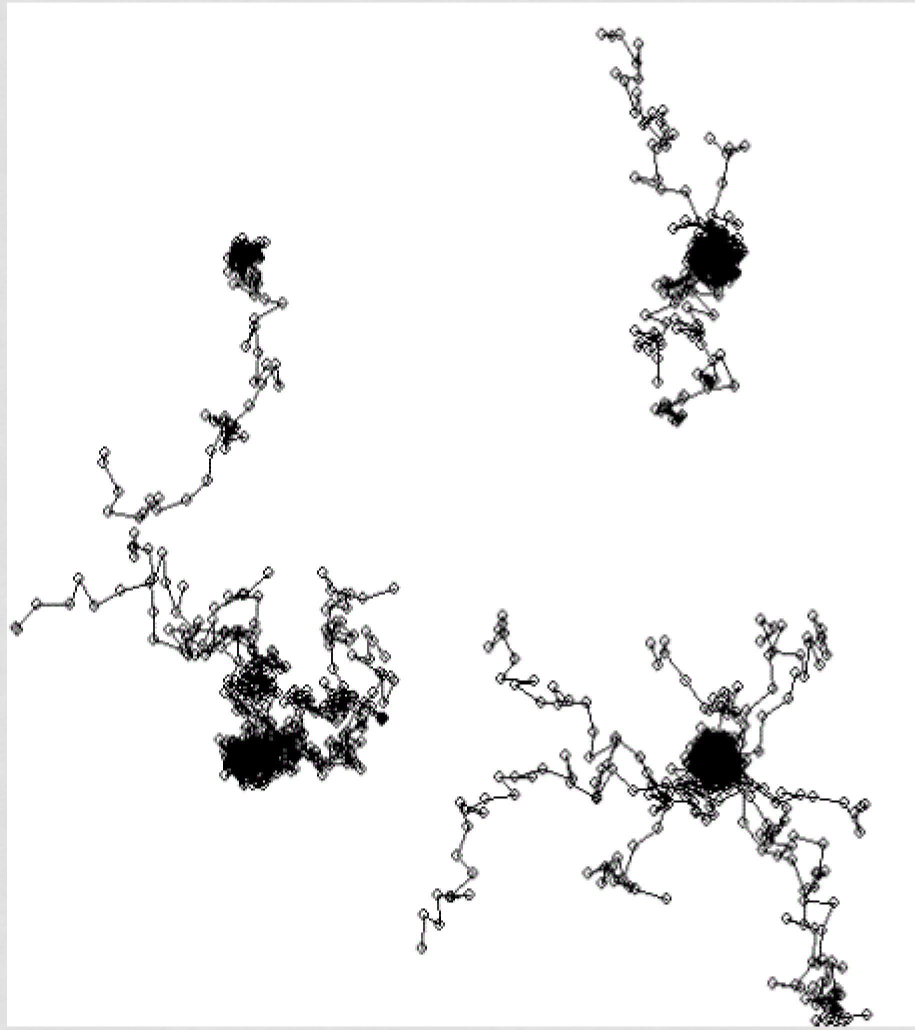
Sample demos with
α=1/3
β=.15



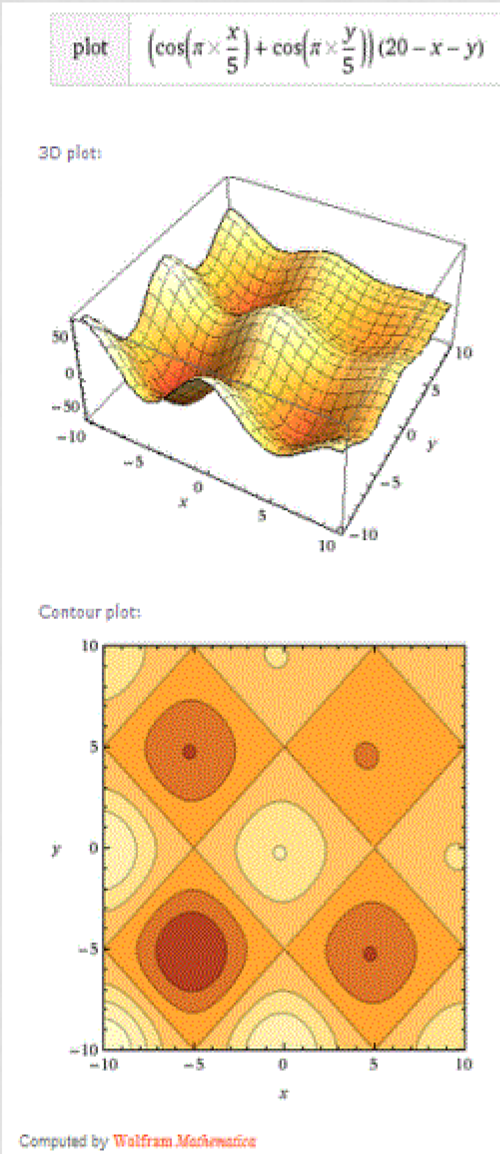


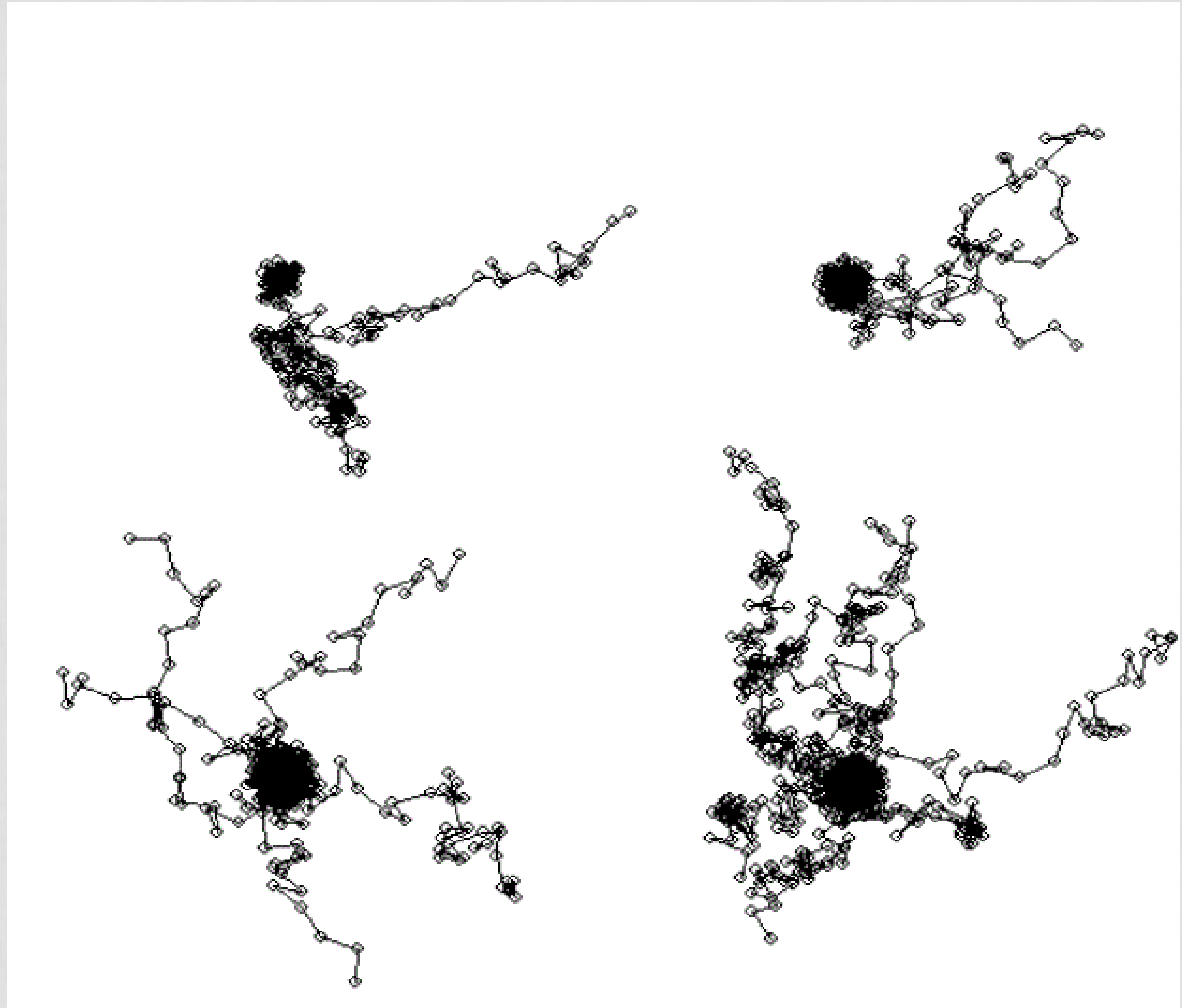
Sample demos with
α=1/3
β=.15



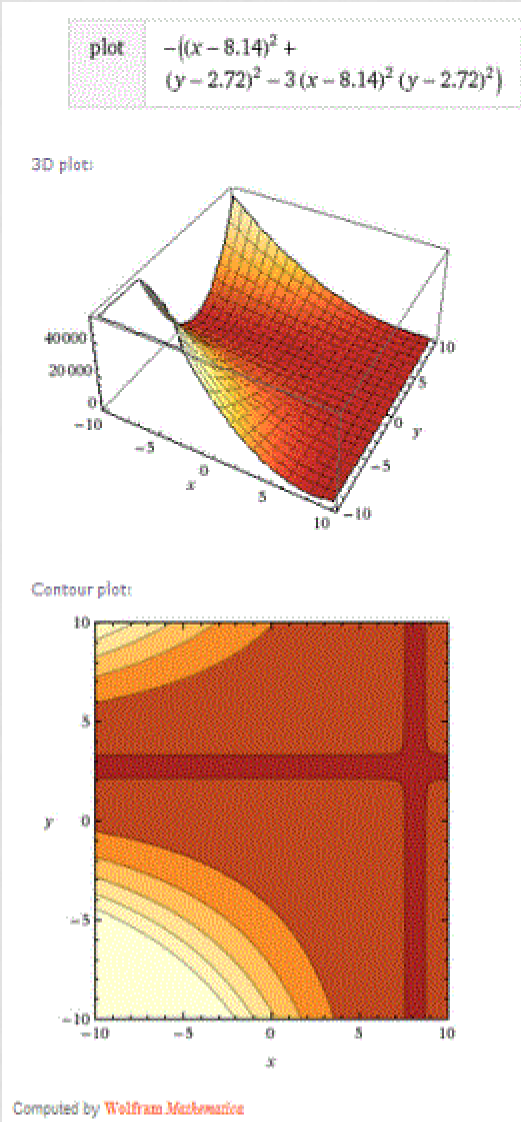


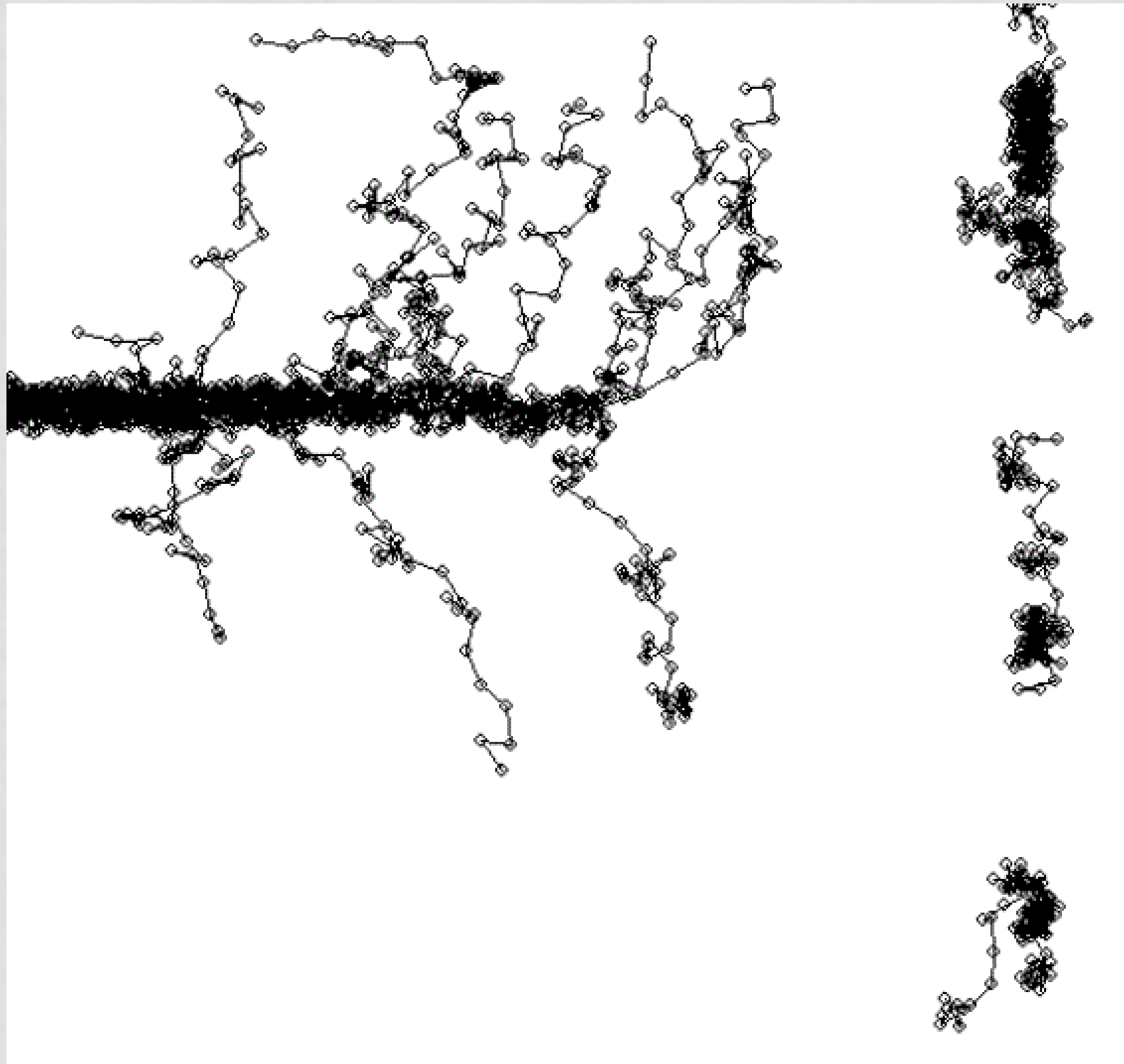
Sample demos with
α=1/3
β=.15





Sample demos with
α=1/3
β=.15





Sample demos with
α=1/3
β=.15