

Search-Based Testing

Generación Automática de Tests - 2018

Repaso: Random Testing



- Es una búsqueda (casi) completamente no guiada
- Si la técnica sistemática no es mejor que random testing, entonces no es valiosa
- Barata & fácil de implementar
- Funciona bastante bien en muchos casos

Repaso: Concolic Testing

- Ejecuta concretamente el test pero guarda la path condition
- Utiliza un constraint solver para crear nuevos inputs



Limitaciones - Ejemplo

```
def testMe(x, y):
    if x == 2 * (y + 1):
        return True
    else:
        return False
```

- **Random Testing:** poca posibilidad de alcanzar la rama “True”
- **Concolic Testing:** la rama “True” no es alcanzable si el constraint solver no soporta aritmética no-lineal

Limitaciones

- Random Testing:



- Dificultad en generar inputs que alcancen código poco probable (distribución uniforme)

- Concolic Testing:

- Tamaño de la path condition
- Capacidad del Constraint Solver



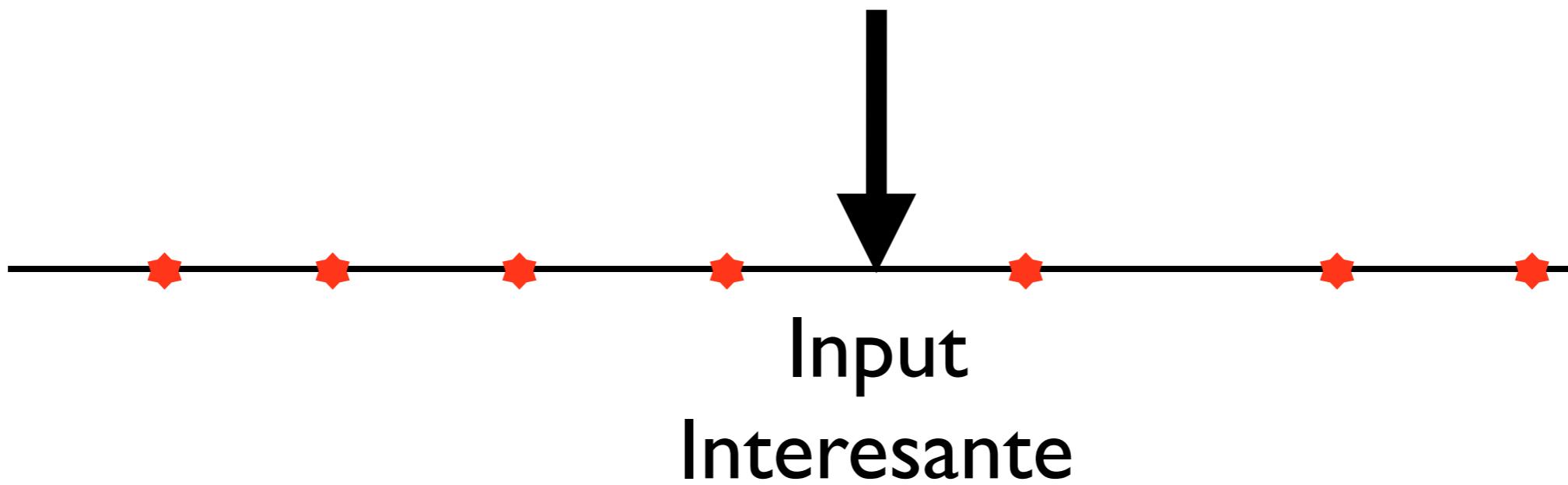
Search-Based Software Engineering

- Transformar los problemas de la Ingeniería de Software en problemas de optimización
- Los espacios de búsqueda en Ingeniería del Software son **GRANDES**
- Aplicar algoritmos de búsqueda meta-heurísticos para resolver estos problemas

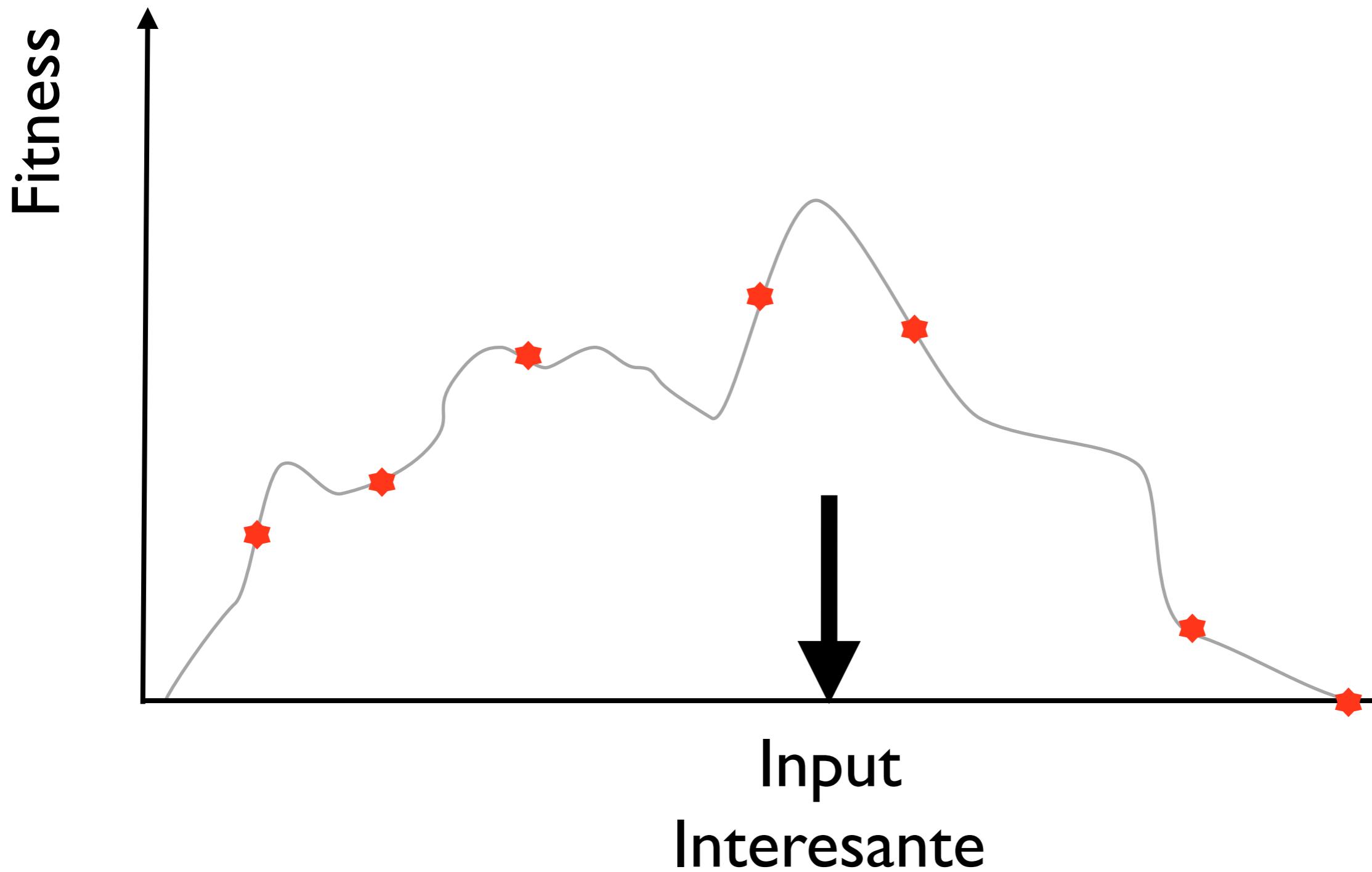
Heurísticas

- Pueden no siempre encontrar la mejor solución
 - Pero encuentran una buena solución en una cantidad razonable de tiempo
 - Sacrifican **completitud** pero ganan **eficiencia**
- Útiles en resolver problemas difíciles:
 - Que no pueden ser resueltos por ningún otro
 - Que tomarían mucho tiempo en ser computados

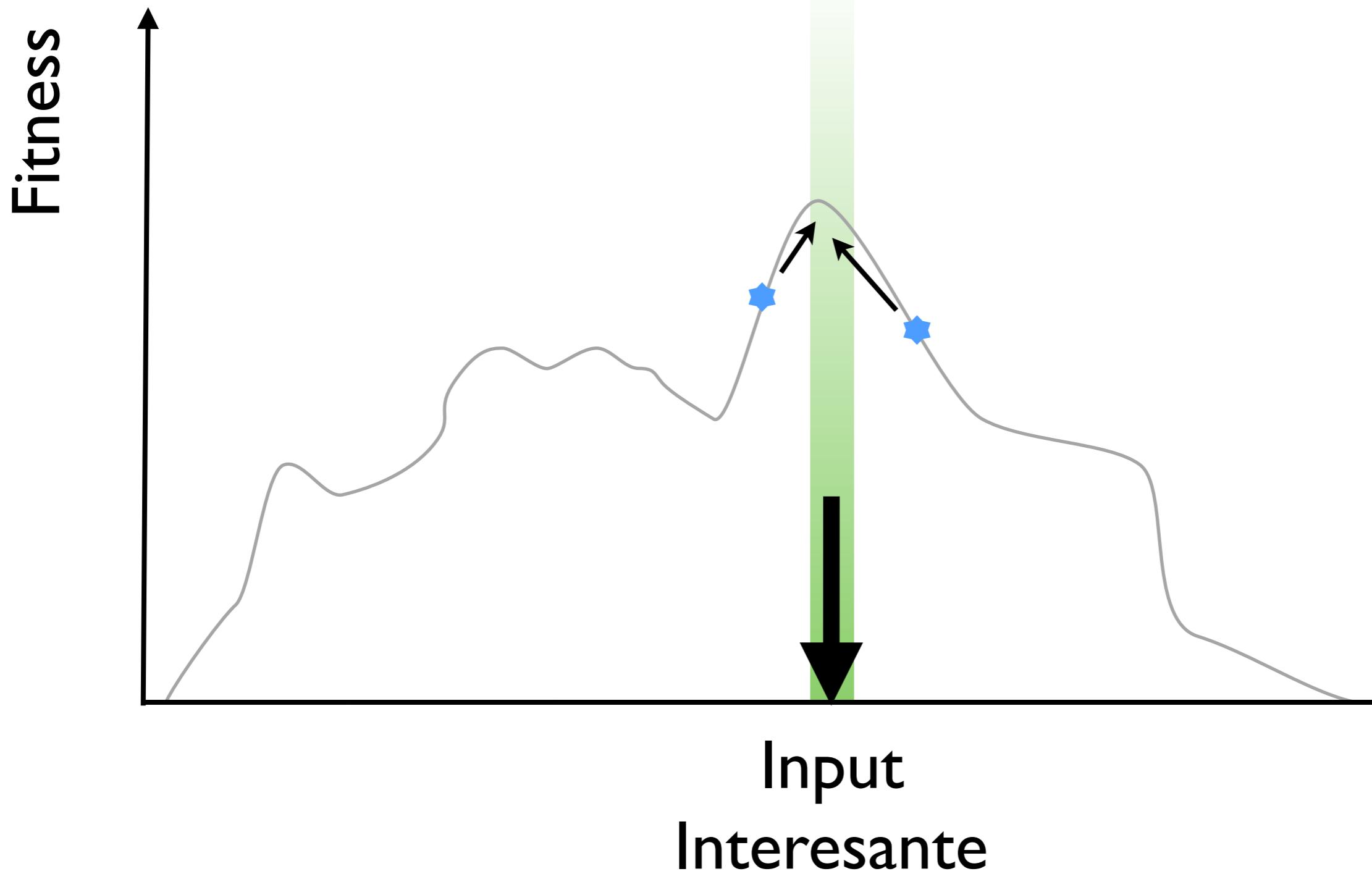
Random Testing



Search-Based Testing

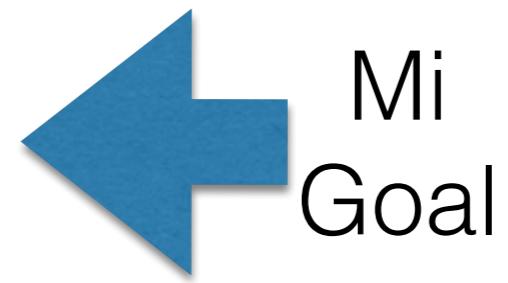


Search-Based Testing



Ejemplo #1

```
def testMe(x, y):  
    if x == y:  
        return True  
    else:  
        return False
```

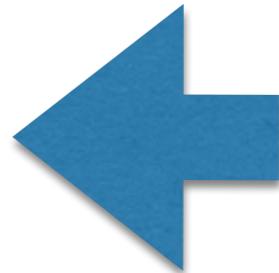


Mi
Goal

Ejemplo #1

SUT

```
def testMe(x, y):  
    if x == y:  
        return True  
    else:  
        return False
```



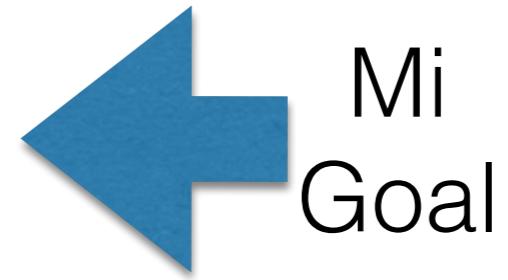
Goal

Test Driver

```
MAX = -10000000  
MIN = +10000000  
  
ret_val = False  
while ret_val!=True:  
    x = random.randint(MIN, MAX)  
    y = random.randint(MIN, MAX)  
    print("Testing x:" + str(x) + ", y:" + str(y))  
    ret_val = testMe(x,y)  
    print("Result=" + str(ret_val))
```

Ejemplo #1

```
def testMe(x, y):  
    if x == y:  
        return True  
    else:  
        return False
```

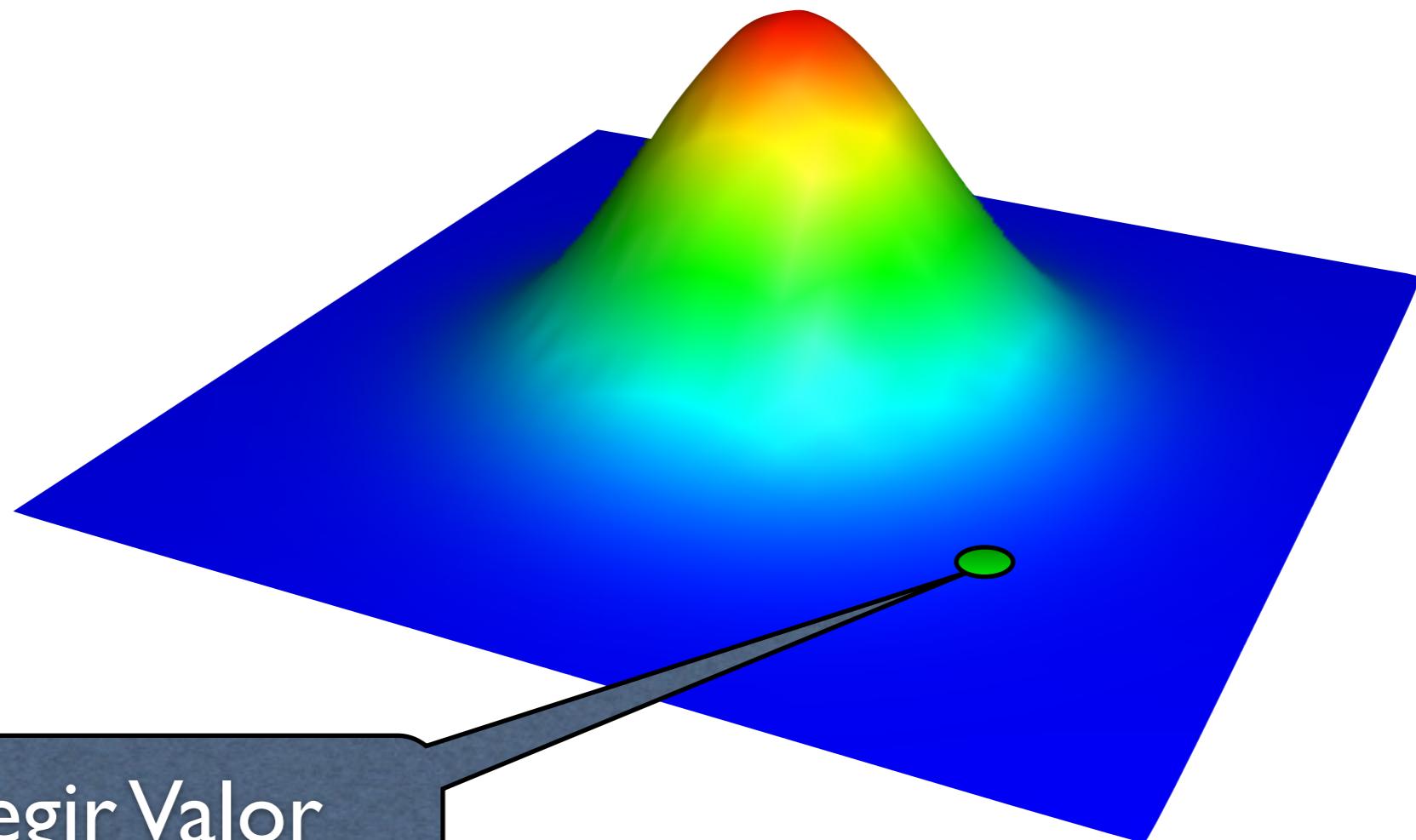


- Si random no funciona ¿Cómo podemos usar una táctica “search-based” para cubrir el goal?

Idea

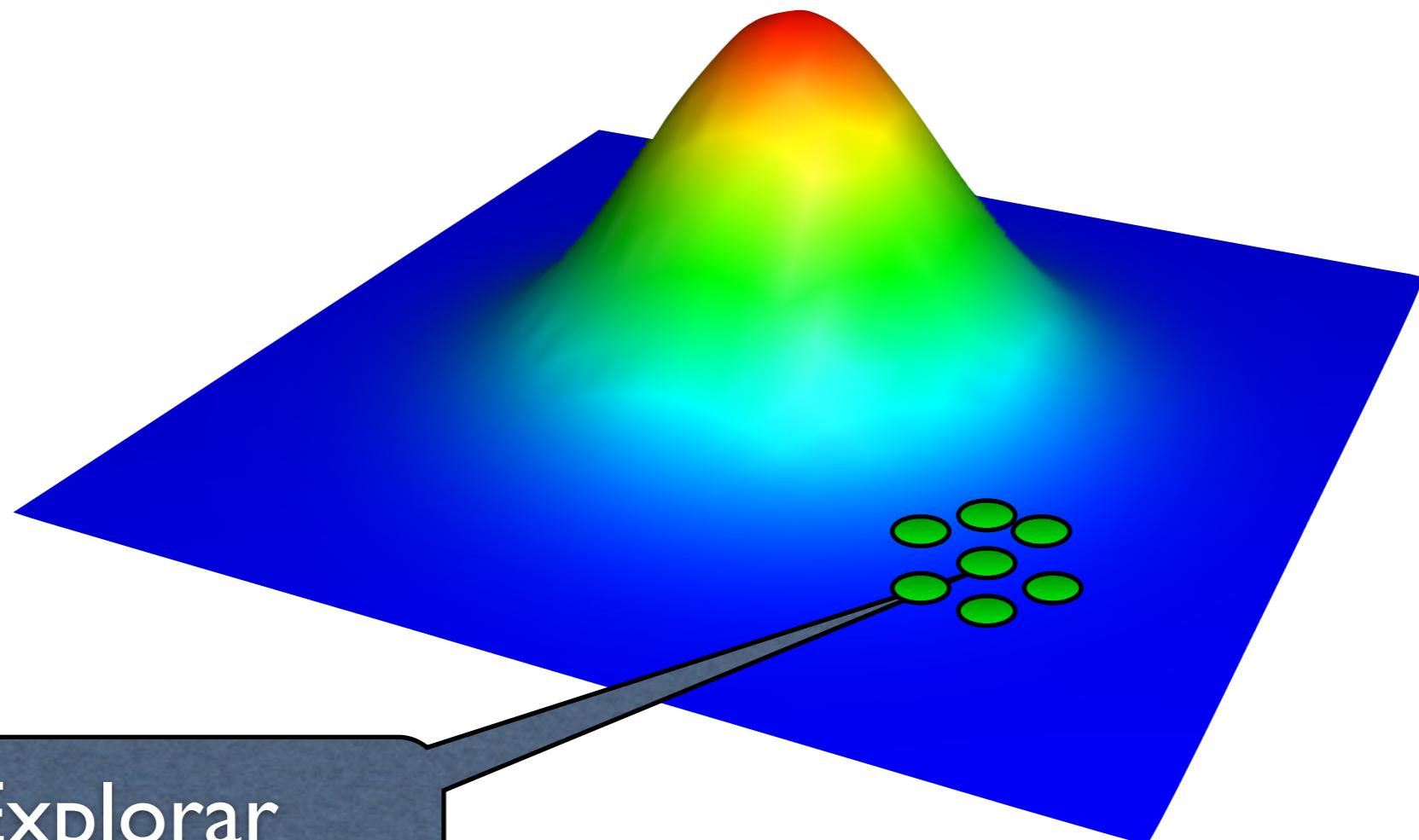
1. Definir una noción de “**cercanía**” del input con respecto al **goal** que queremos alcanzar
2. Por cada nueva solución, explorar el “vecindario” de todas las soluciones
3. Elegir la solución del vecindario que sea mejor que la actual
4. Repetir hasta no poder encontrar mejoras

Hill Climbing



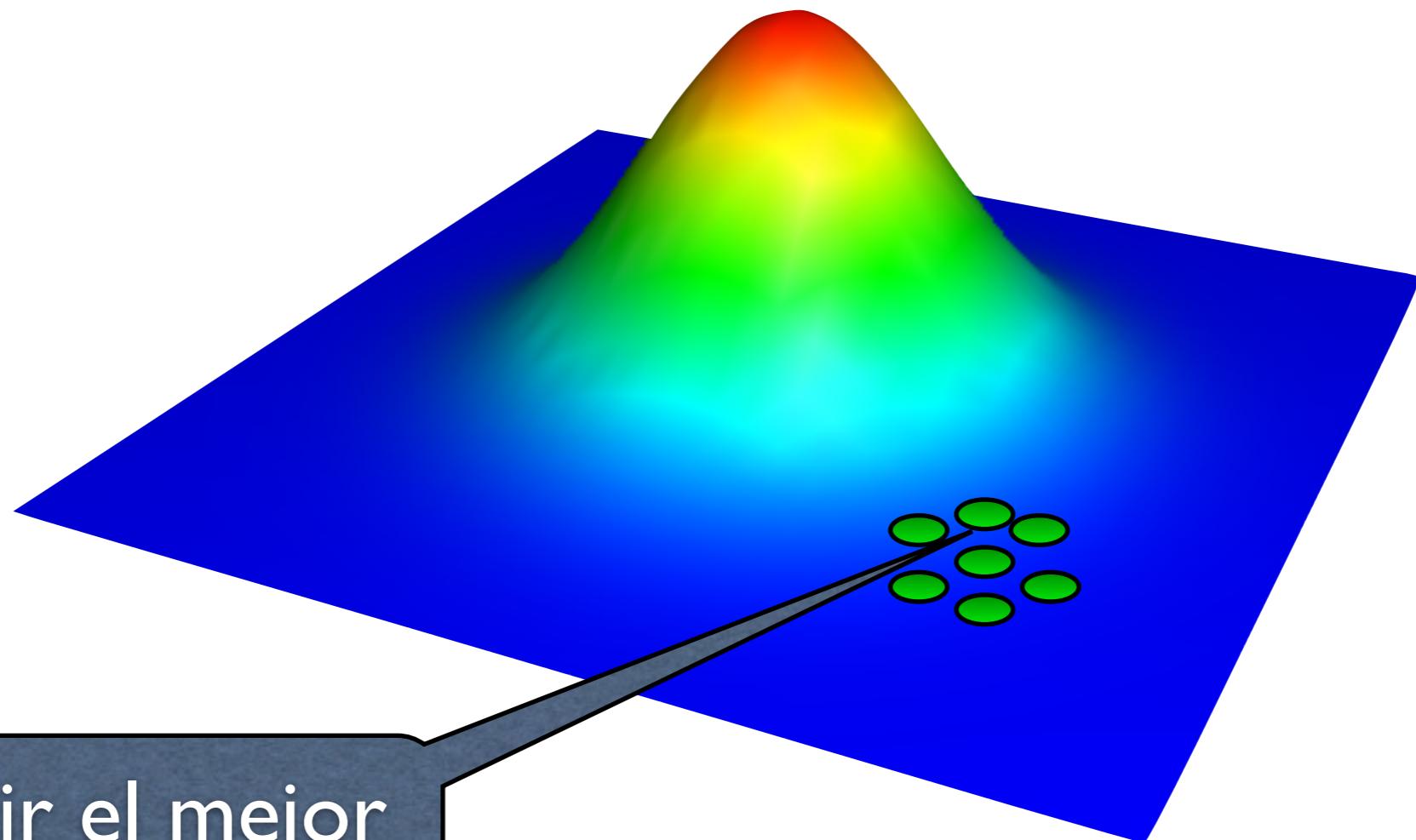
I. Elegir Valor
Aleatorio

Hill Climbing



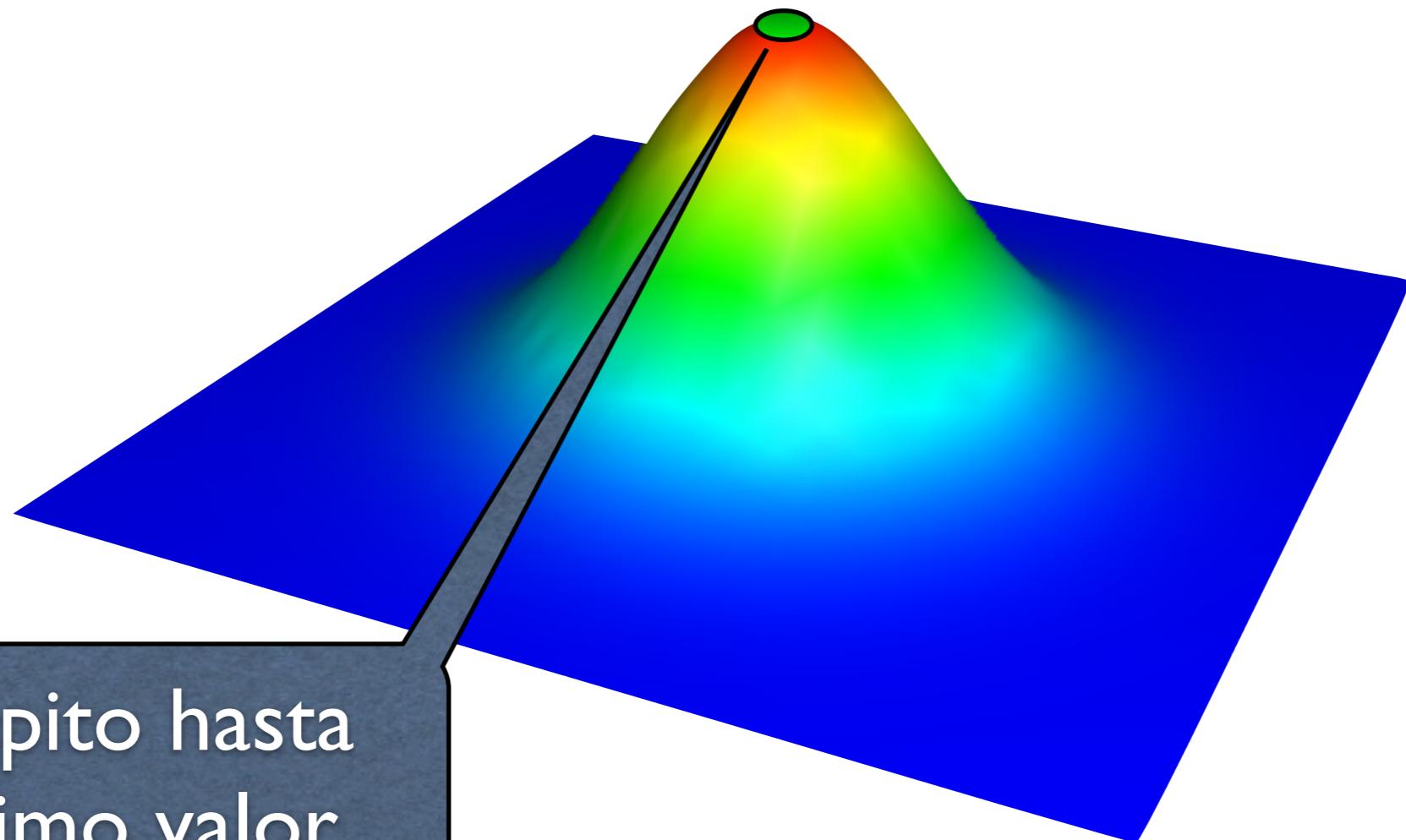
2. Explorar
Vecindario

Hill Climbing



3. Elegir el mejor vecino

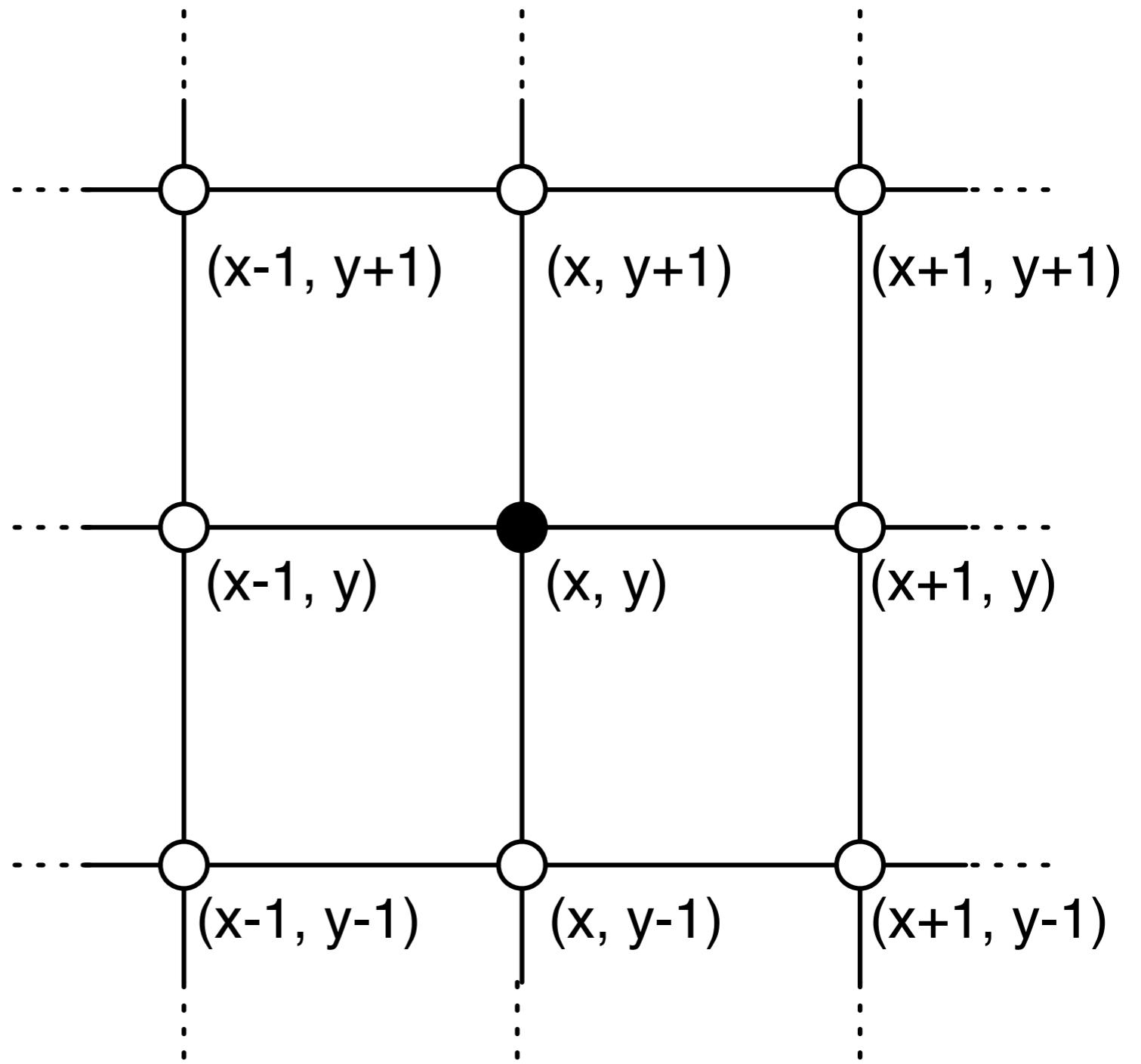
Hill Climbing

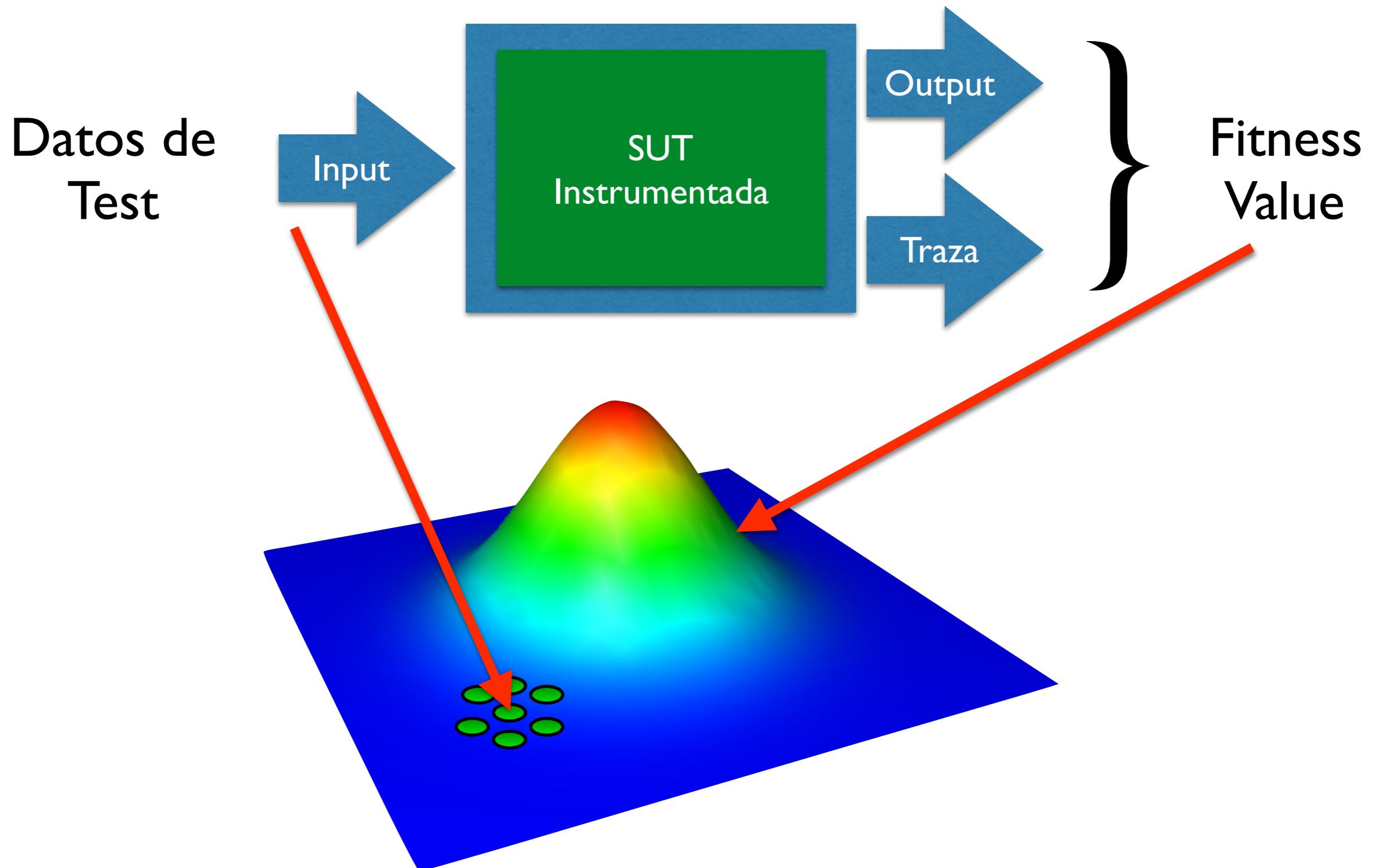


4. Repito hasta
máximo valor

Neighbourhood

```
def testMe(x, y):  
    if x == y:  
        return True  
    else:  
        return False
```





```
def testMe(x, y):
    test_distance = abs(x - y)
    if x == y:
        return True, test_distance
    else:
        return False, test_distance

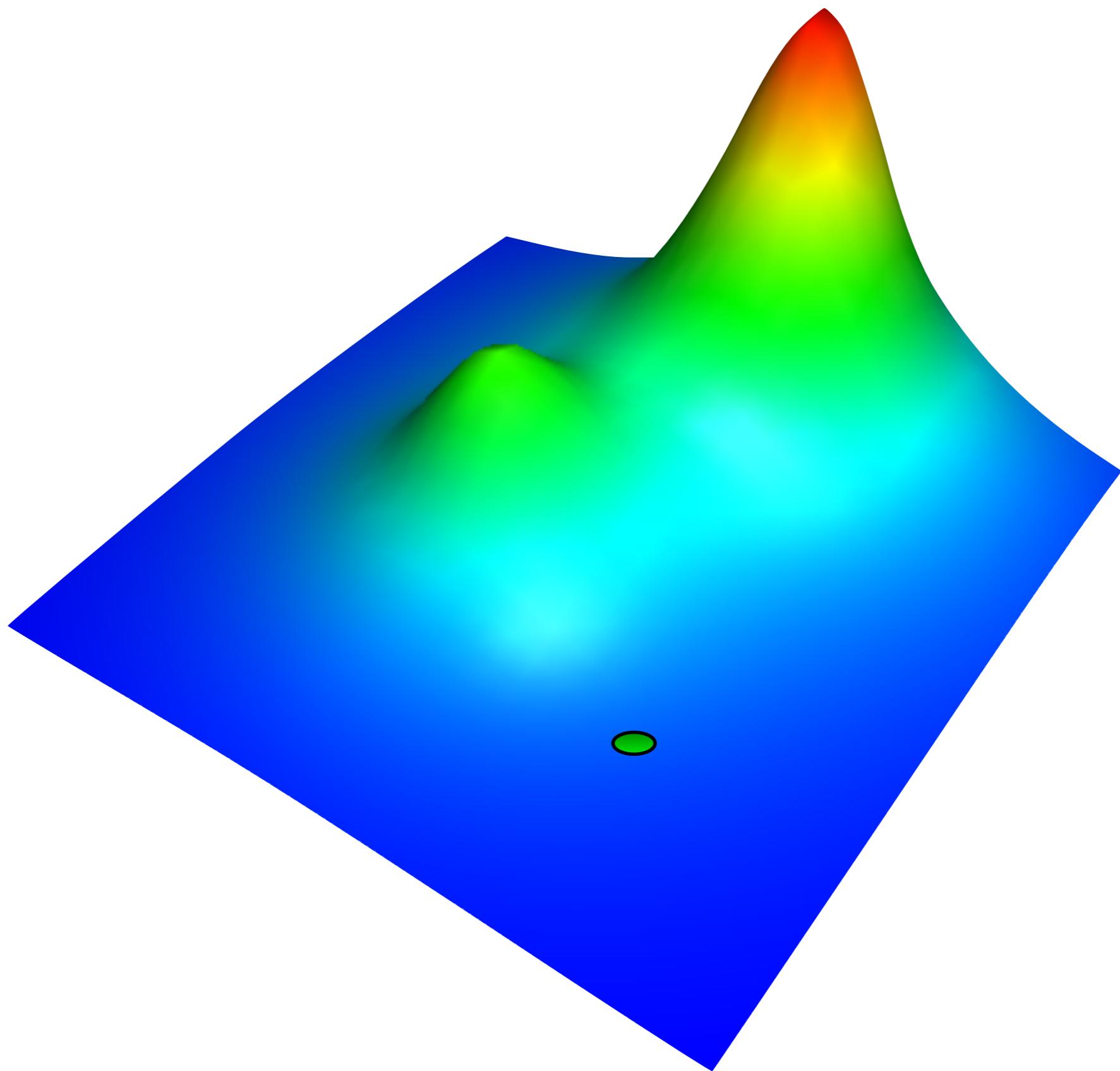
def get_distance(x, y):
    ret_val, test_distance = testMe(x, y)
    return test_distance

def neighbours(x, y):
    return [(x+dx, y+dy) for dx in [-1,0,1] \
             for dy in [-1,0,1] \
             if (dx != 0 or dy != 0) \
             and ((MIN <= x+dx <= MAX) \
             and (MIN <= y+dy <= MAX))]
```

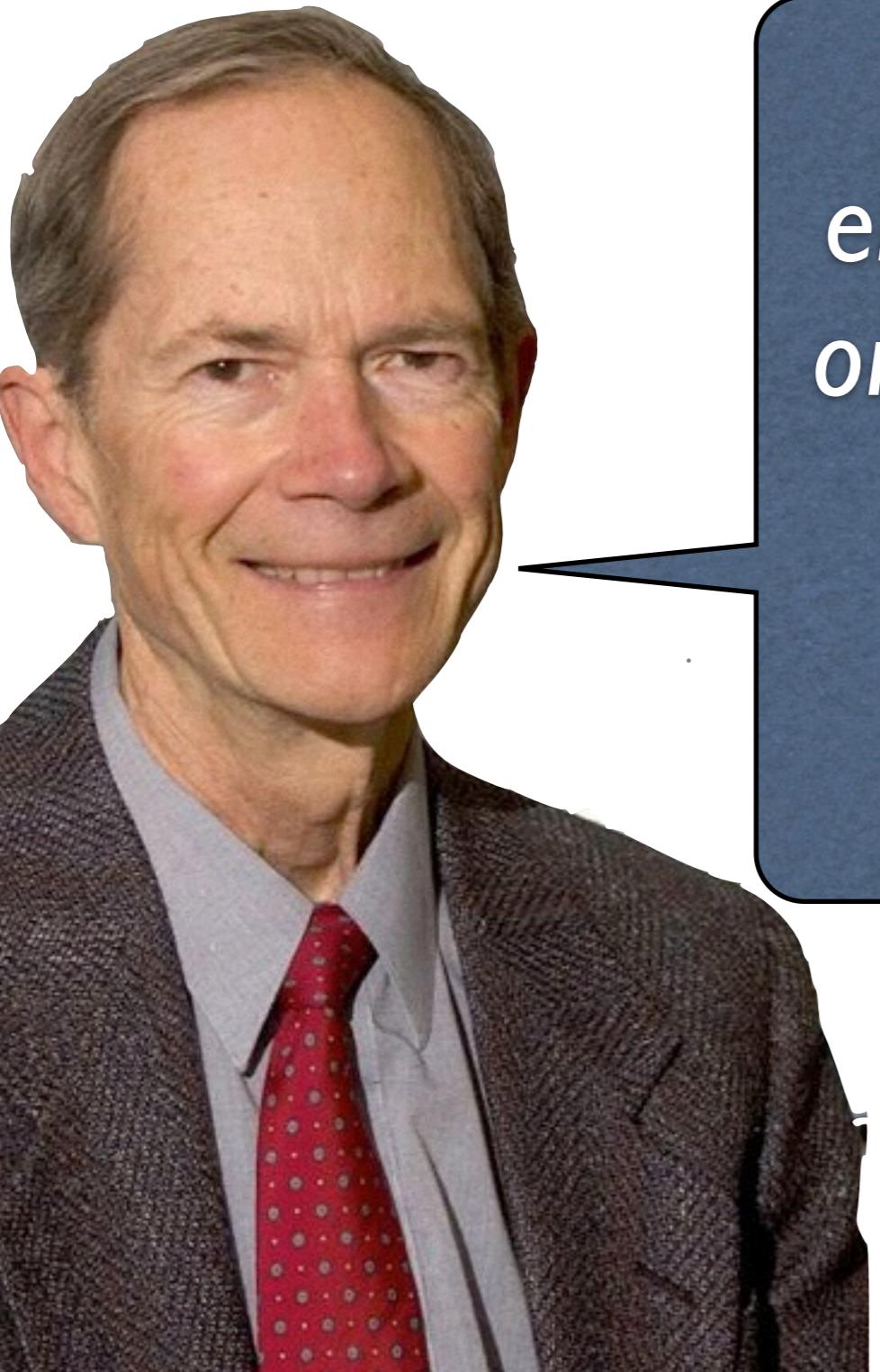
```
MAX = -10000000
MIN = +10000000
```

```
# Elijo al azar una primera solución
x = random.randint(MIN,MAX)
y = random.randint(MIN,MAX)
distance = get_distance(x,y)

# Mientras no haya alcanzado el goal
while distance>0:
    changed = False
    # Busco entre los vecinos
    for (nextx,nexty) in neighbours(x,y):
        new_distance = get_distance(nextx,nexty)
        if new_distance<distance:
            # Existe un vecino que es mejor!
            x = nextx
            y = nexty
            distance = new_distance
            changed = True
    break
    # Ningun vecino es mejor, termino la búsqueda
if not changed:
    break
```



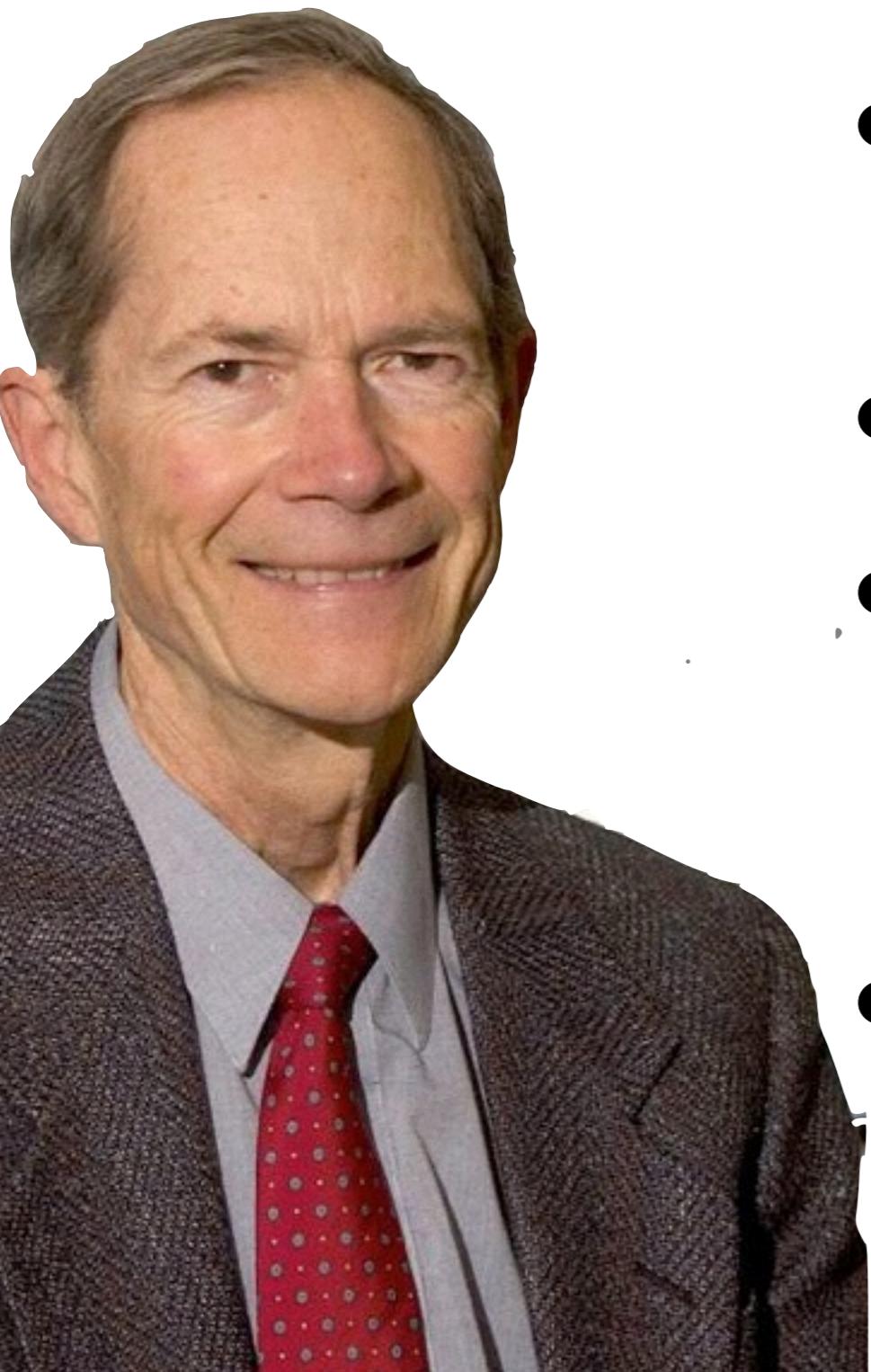
Tabu Search



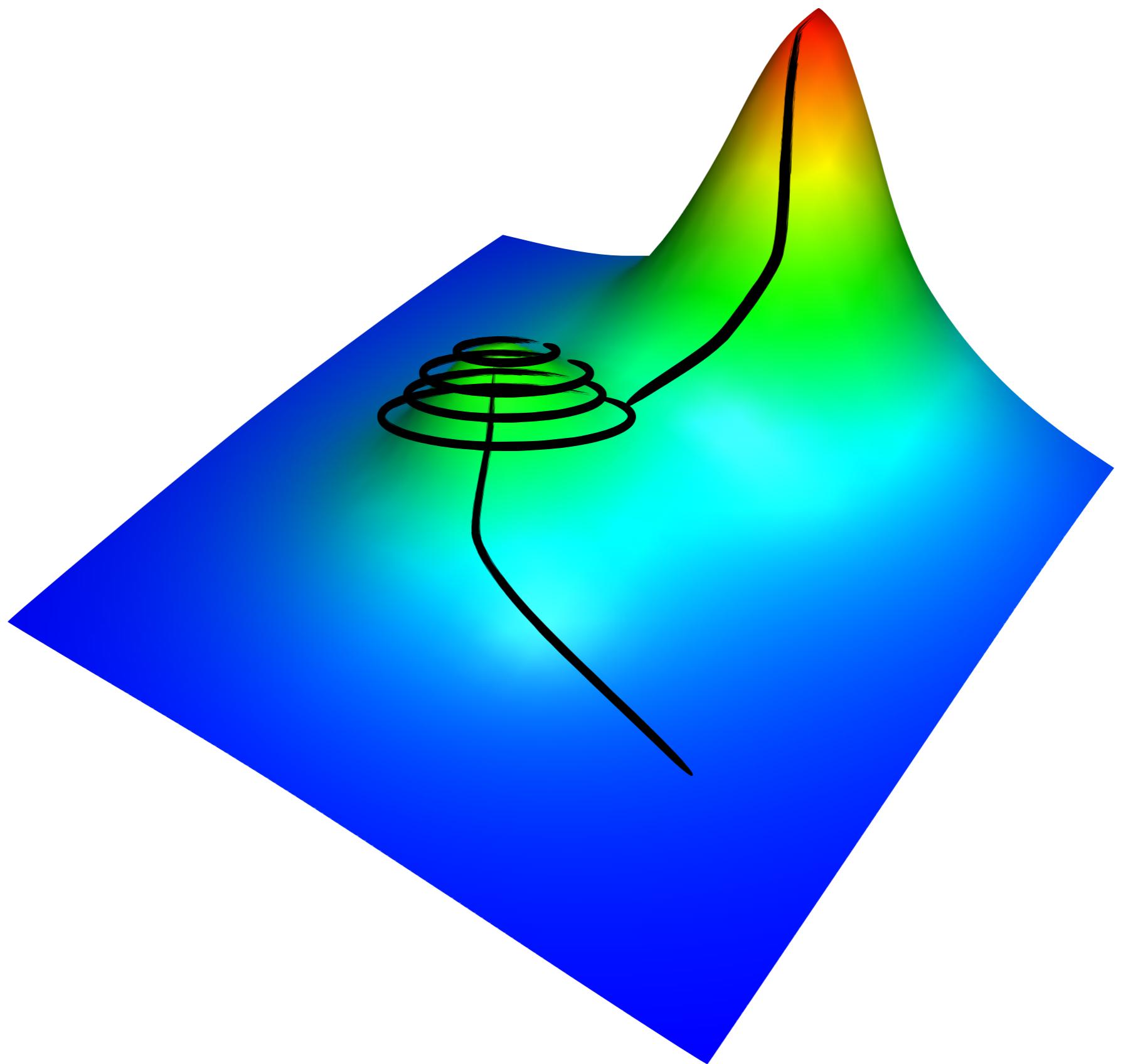
The overall approach is to avoid entrainment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited.

(Fred Glover 1987)

Tabu (taboo) Search



- Almacena los últimos k-movimientos
- “Tabu list”
- Cuando elijo un nuevo input, no puedo tomar ninguna solución que esté en la “Tabu list”
- Evita ciclos de longitud k



Ejemplo #2

```
def testMe(x, y, z):  
    if x * z == 2 * (y + 1):  
        return True  
    else:  
        return False
```

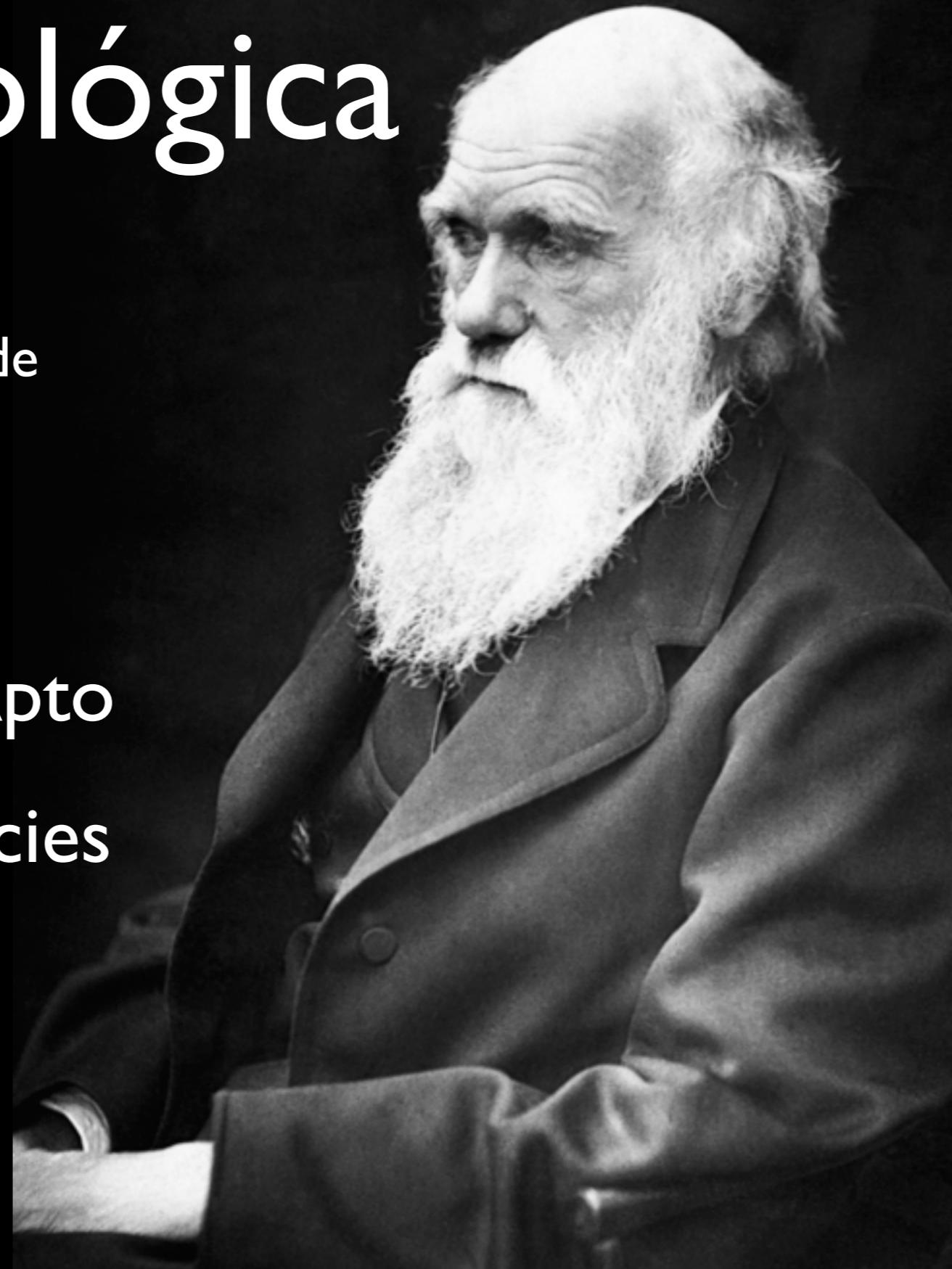
- ¿Qué pasa con este ejemplo?
 - Cuando el neighbourhood es demasiado grande, Hill-Climbing se torna demasiado lento
 - Necesitamos mecanismos más sofisticados de búsqueda

Algoritmos Evolutivos

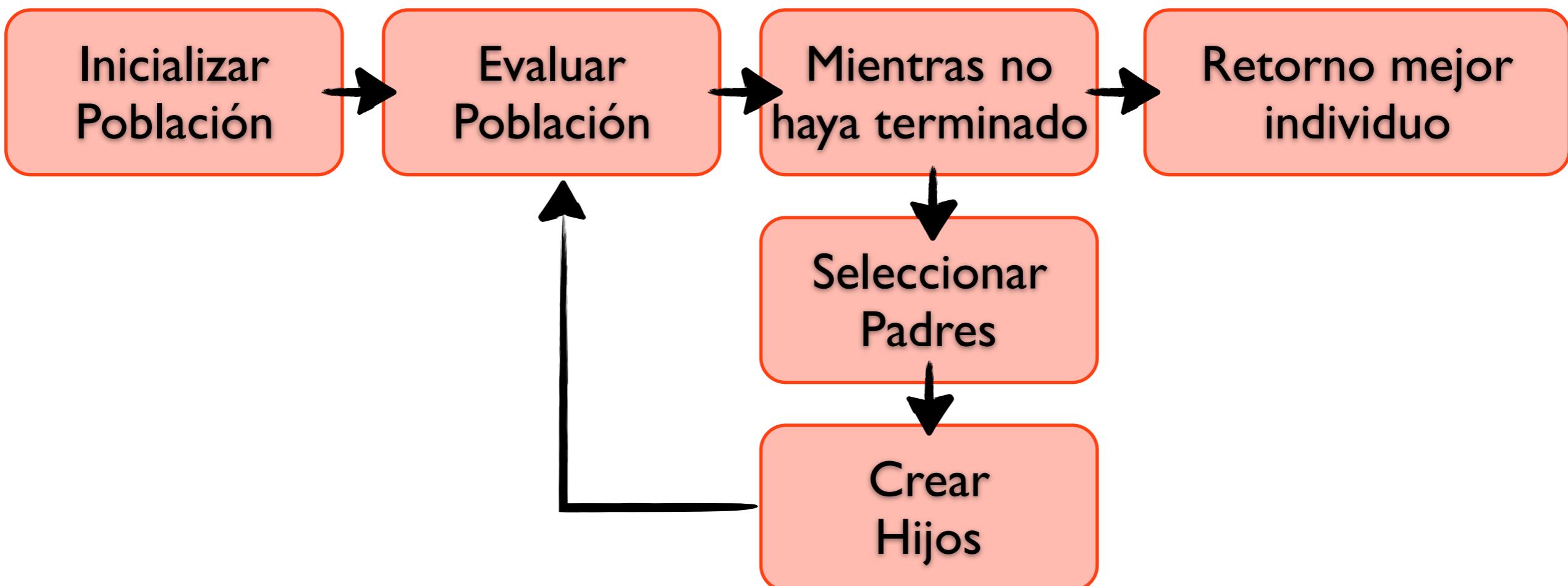


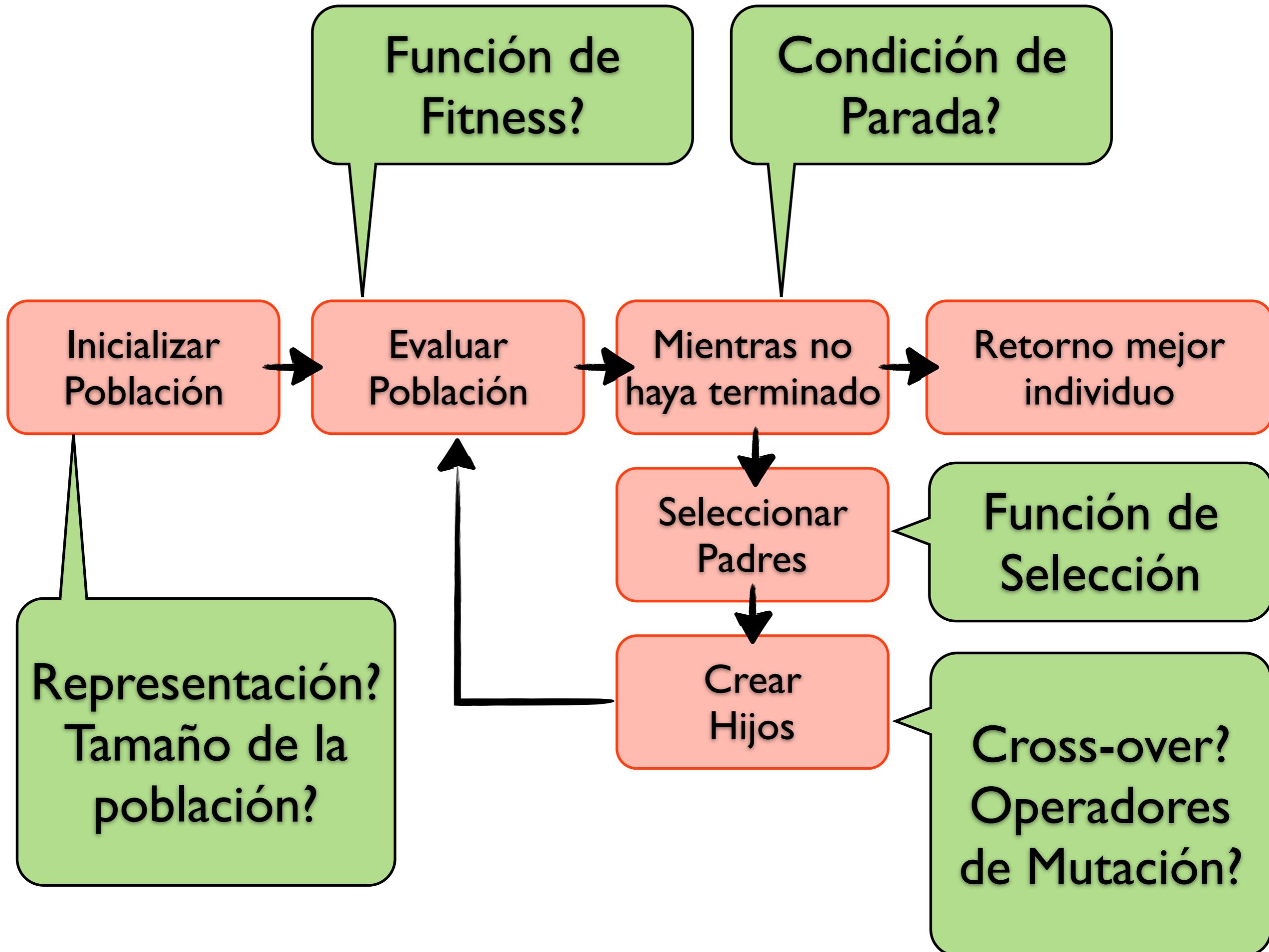
Evolución Biológica

- Gen
Unidad de información • pasado de una generación a la otra
- Selección Natural
- Supervivencia del Más Apto
- Origen de Nuevas Especies



Algoritmos Genéticos





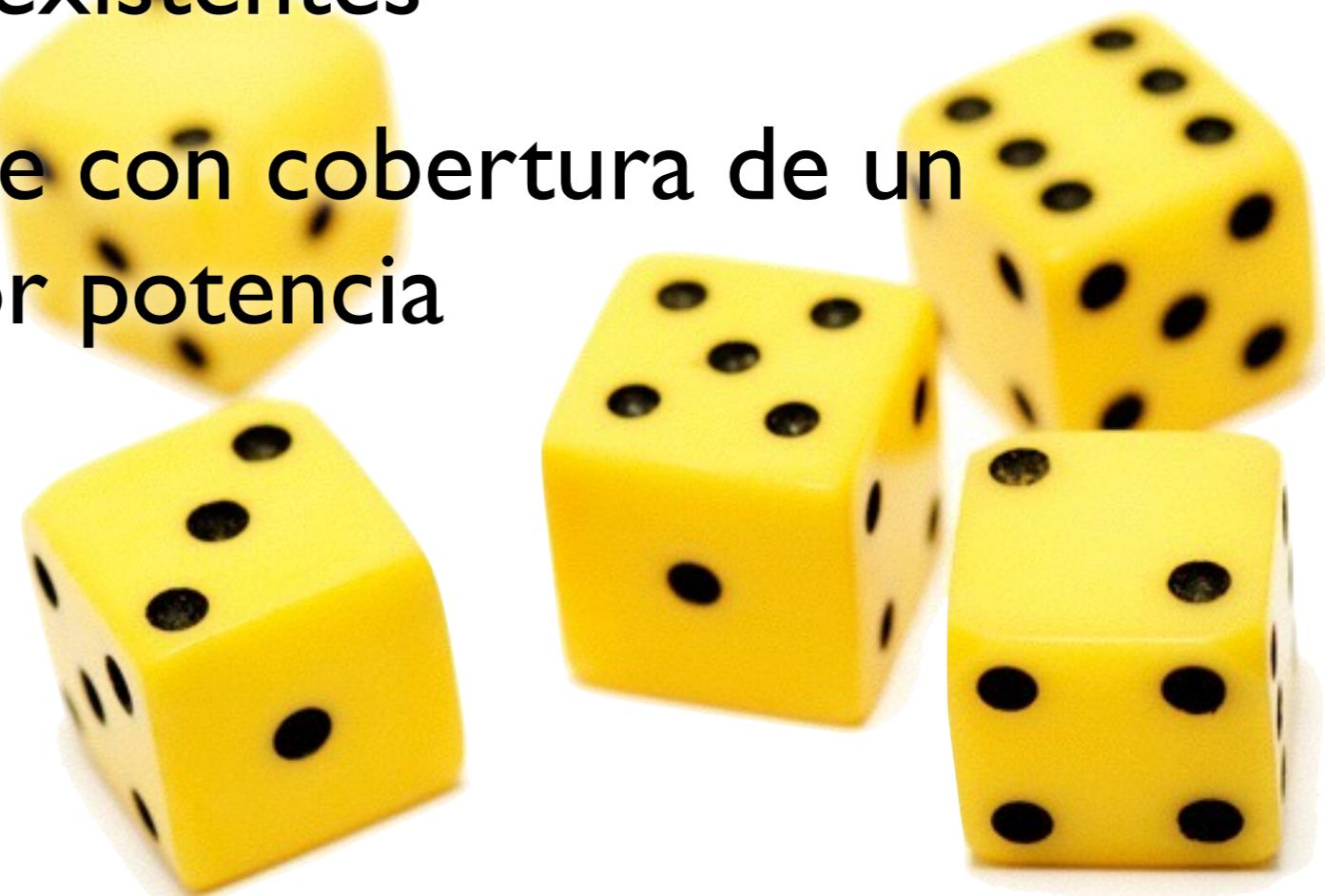
Representación

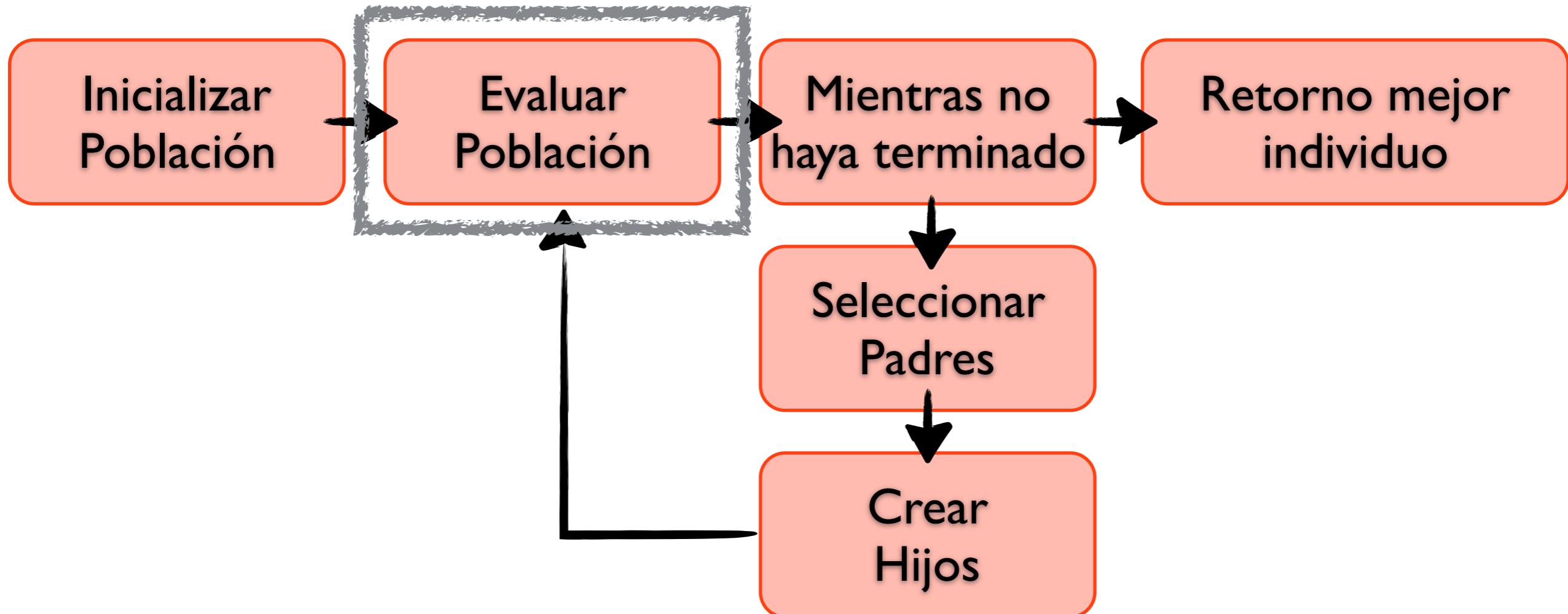
```
def testMe(x, y, z):  
    if x * z == 2 * (y + 1):  
        return True //branch a cubrir  
    else:  
        return False
```

x	y	z
10	10	20

¿Cómo generar la Población Inicial ?

- Aleatoriamente
- Usar soluciones existentes
- Usar un Test Suite con cobertura de un criterio de menor potencia
- Manualmente

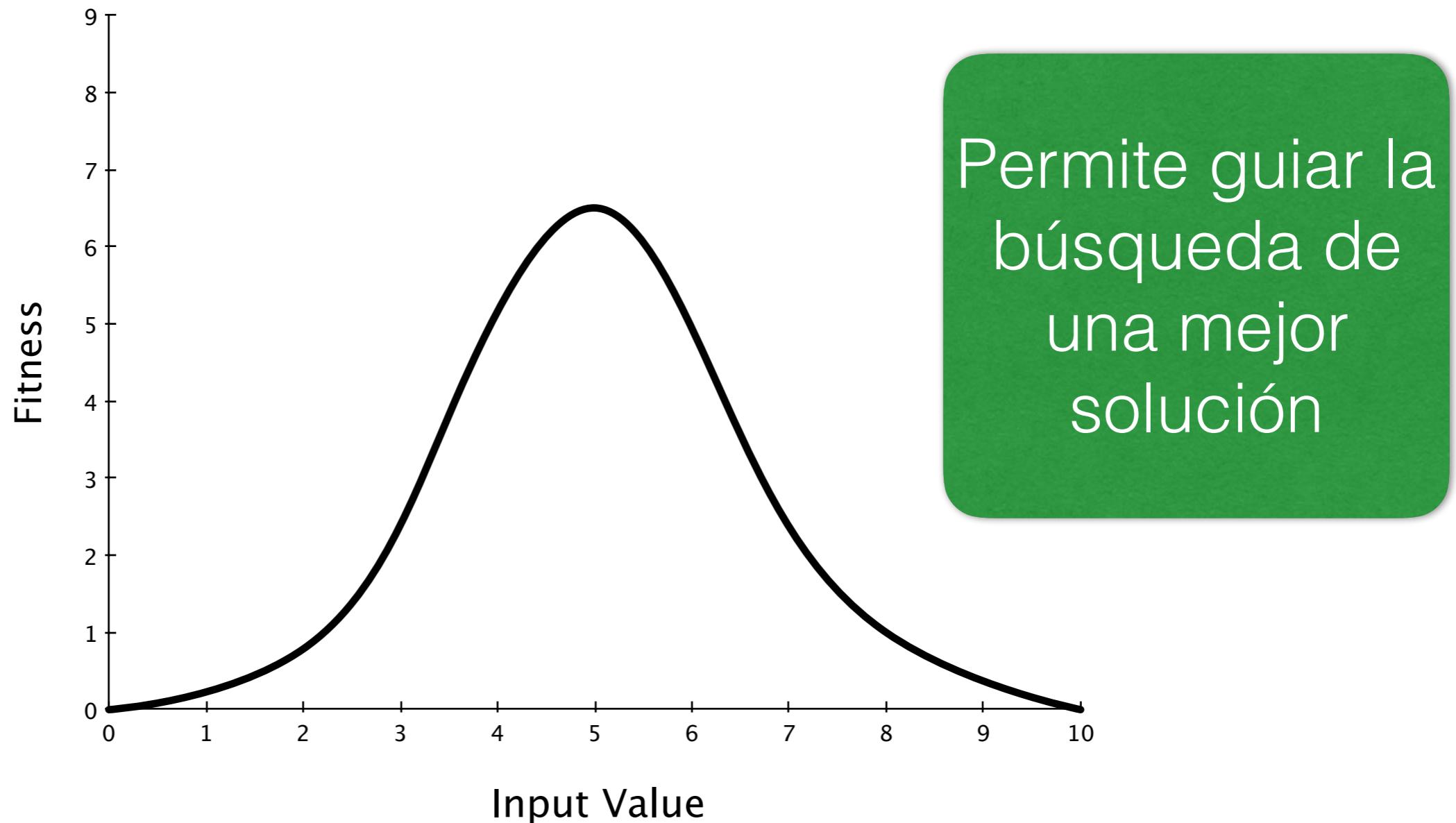




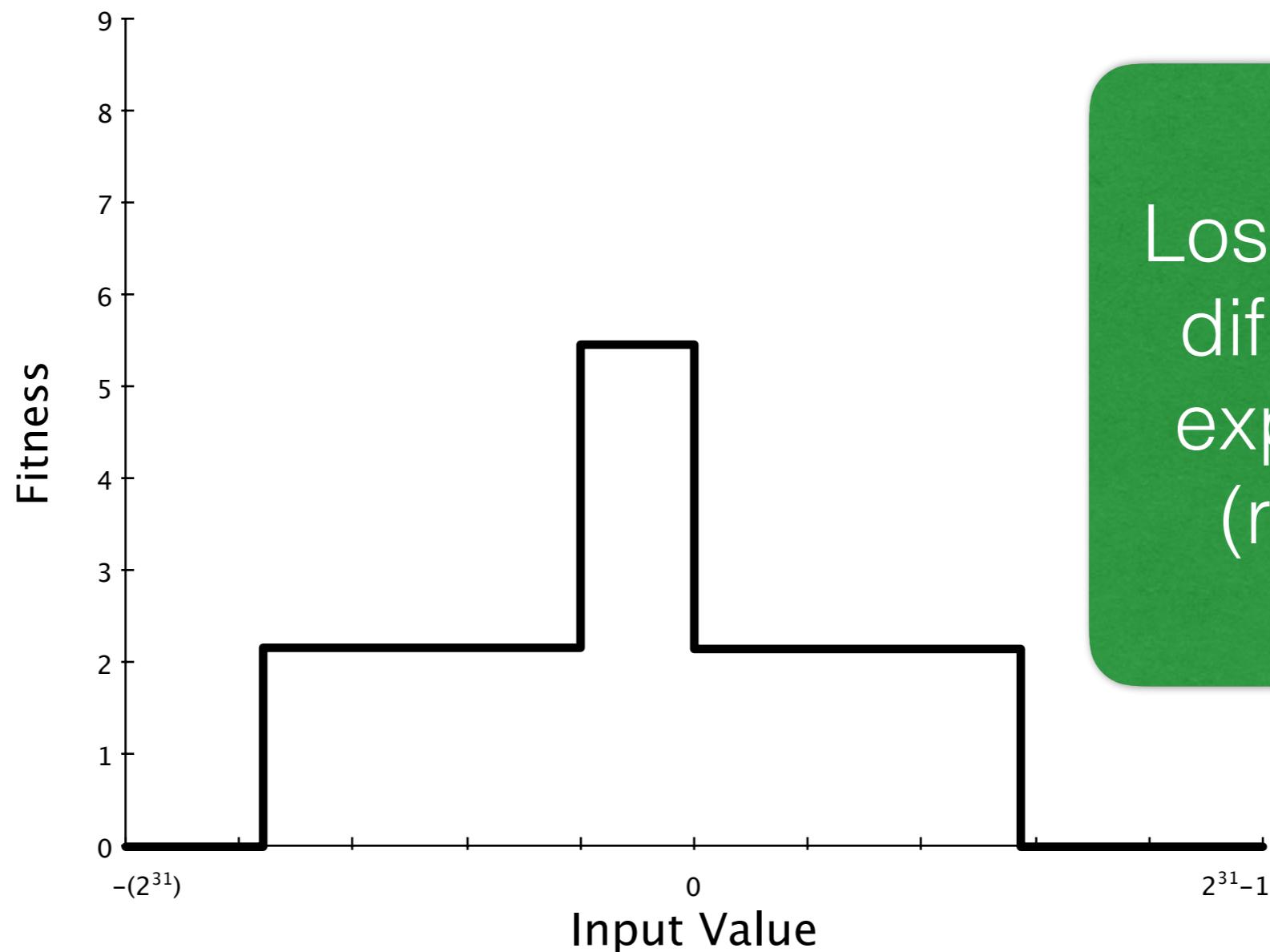
Fitness Function

- Mide cuán bueno es un individuo como solución
 - ¿Es el individuo *A* mejor que el individuo *B*?
- Cuantifica la “**bondad**” de un individuo
- La Fitness Function es específica al problema
 - Determina el “paisaje de búsqueda” (fitness landscape)

Smooth Fitness Landscape



Rugged Fitness Landscape



Los **quiebres**
dificultan la
exploración
(random)

Branch Distance

- Branch Distance = distancia del predicado a ejercitar el branch
- Puede ser el TRUE o el FALSE

```
def testMe(x, y, z):  
    if x * z == 2 * (y + 1):  
        return True //branch a cubrir  
    else:  
        return False
```

Branch Distance

$a == b$

$\text{abs}(a-b) = 0 ? 0 : \text{abs}(a-b)$

$a < b$

$a < b ? 0 : (a - b) + k$

$a \leq b$

$a \leq b ? 0 : (a - b)$

$a > b$

$a > b ? 0 : (b - a)$

$a \geq b$

$a \geq b ? 0 : (b - a) + k$

Branch Distance

$o.m(v_1, \dots, v_n)$

true ? 0 : K

$a \neq b$

$a \neq b ? 0 \text{ else } K$

$A \&& B$

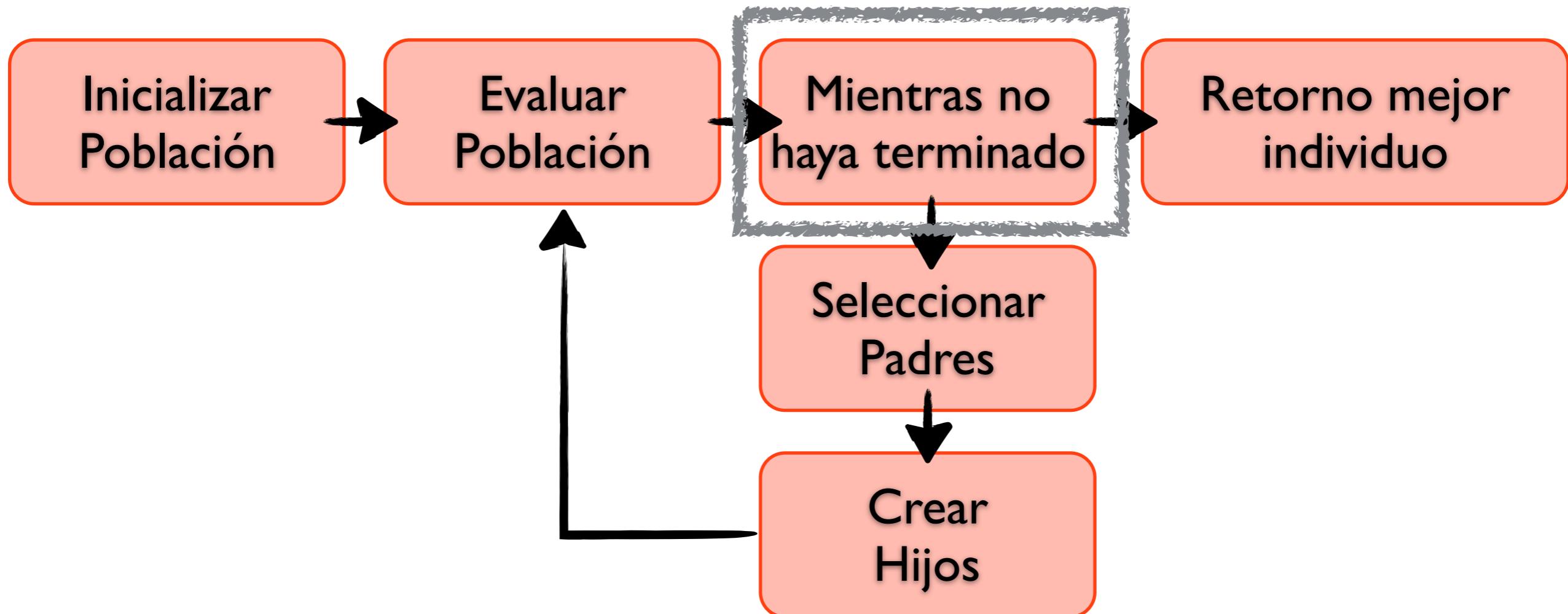
$\text{distance}(A) + \text{distance}(B)$

$A \parallel B$

$\min(\text{distance}(A), \text{distance}(B))$

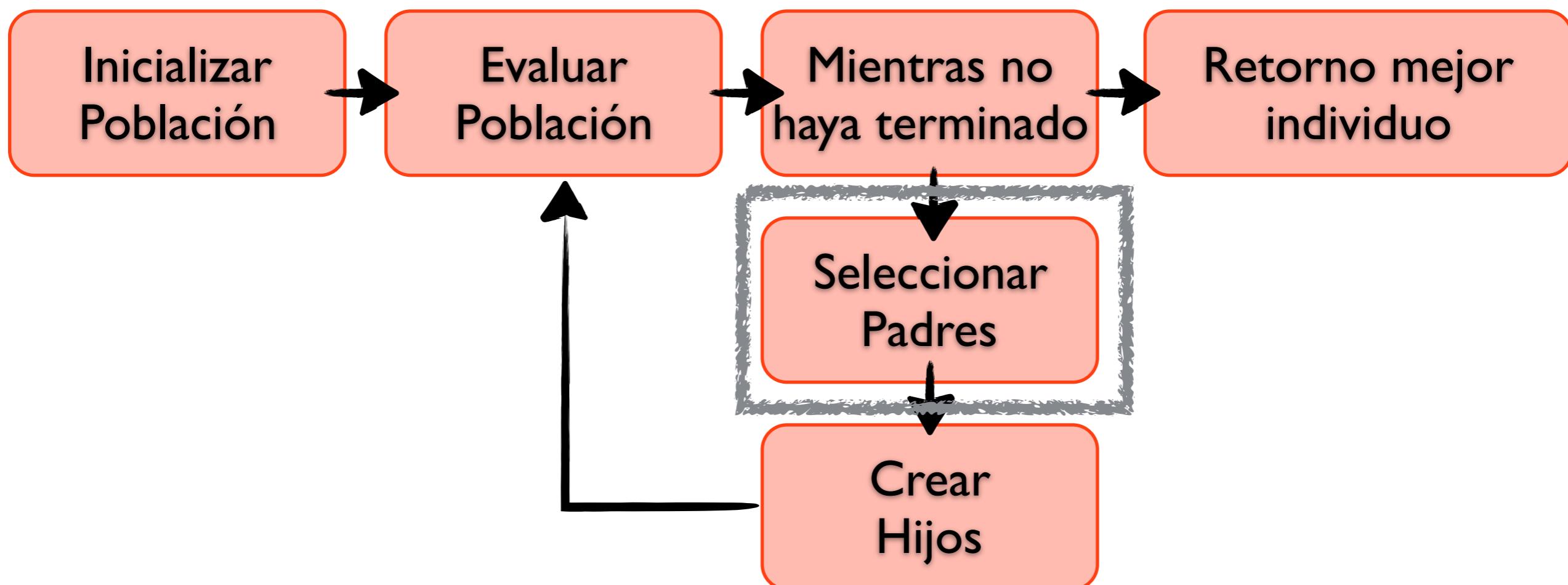
$!a$

Aplicar De Morgan



Condición de Parada

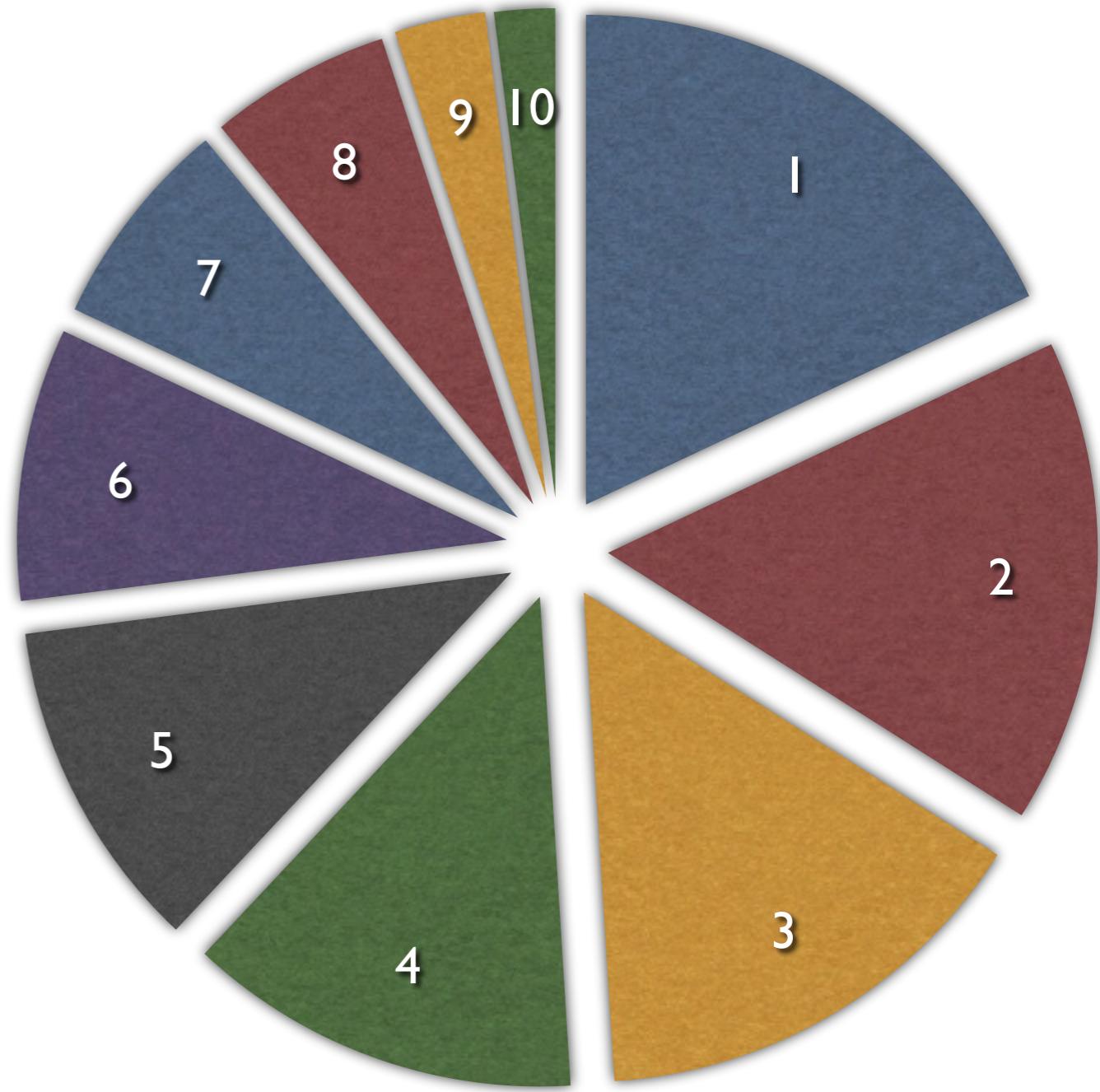
- Algunas posibles opciones:
 - Tiempo Máximo
 - Cantidad máxima de iteraciones
 - Cantidad máxima de evaluaciones del fitness de individuos



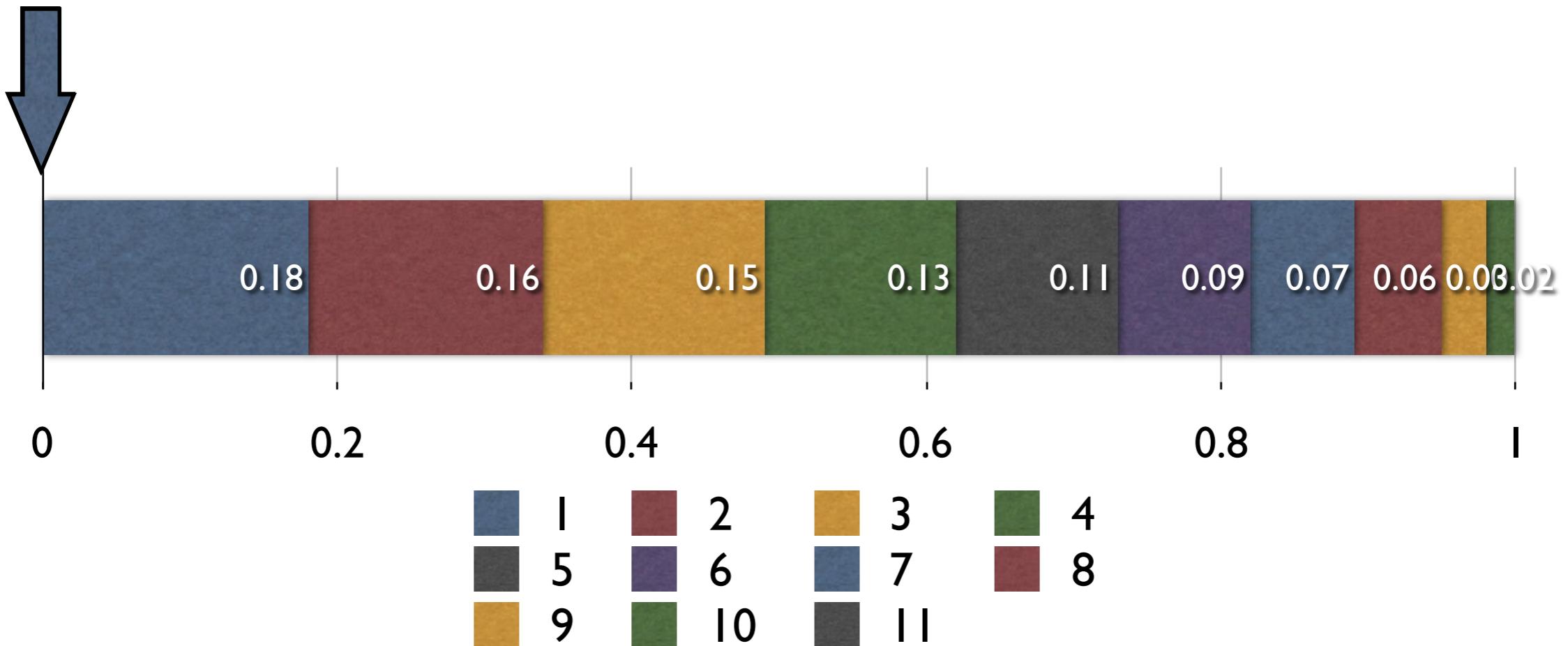
Fitness Proportionate Selection

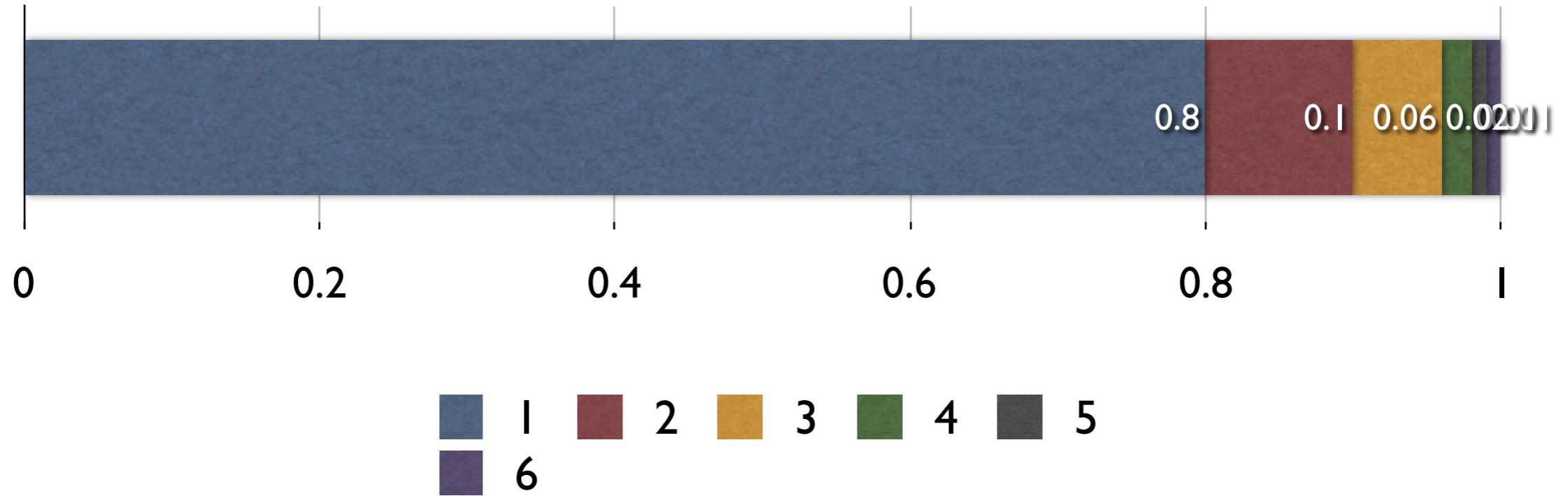
- A.k.a. Roulette wheel selection
- La Probabilidad de elegir un individuo es proporcional a su valor de fitness

Individual	Fitness
1	2
2	1.8
3	1.6
4	1.4
5	1.2
6	1
7	0.8
8	0.6
9	0.4
10	0.2
11	0



- Valor aleatorio: 0,81
- Valor aleatorio: 0,32
- Valor aleatorio: 0,01





- El Individuo I dominará la selección
- ¡No use la rueda de ruleta!

Rank Selection

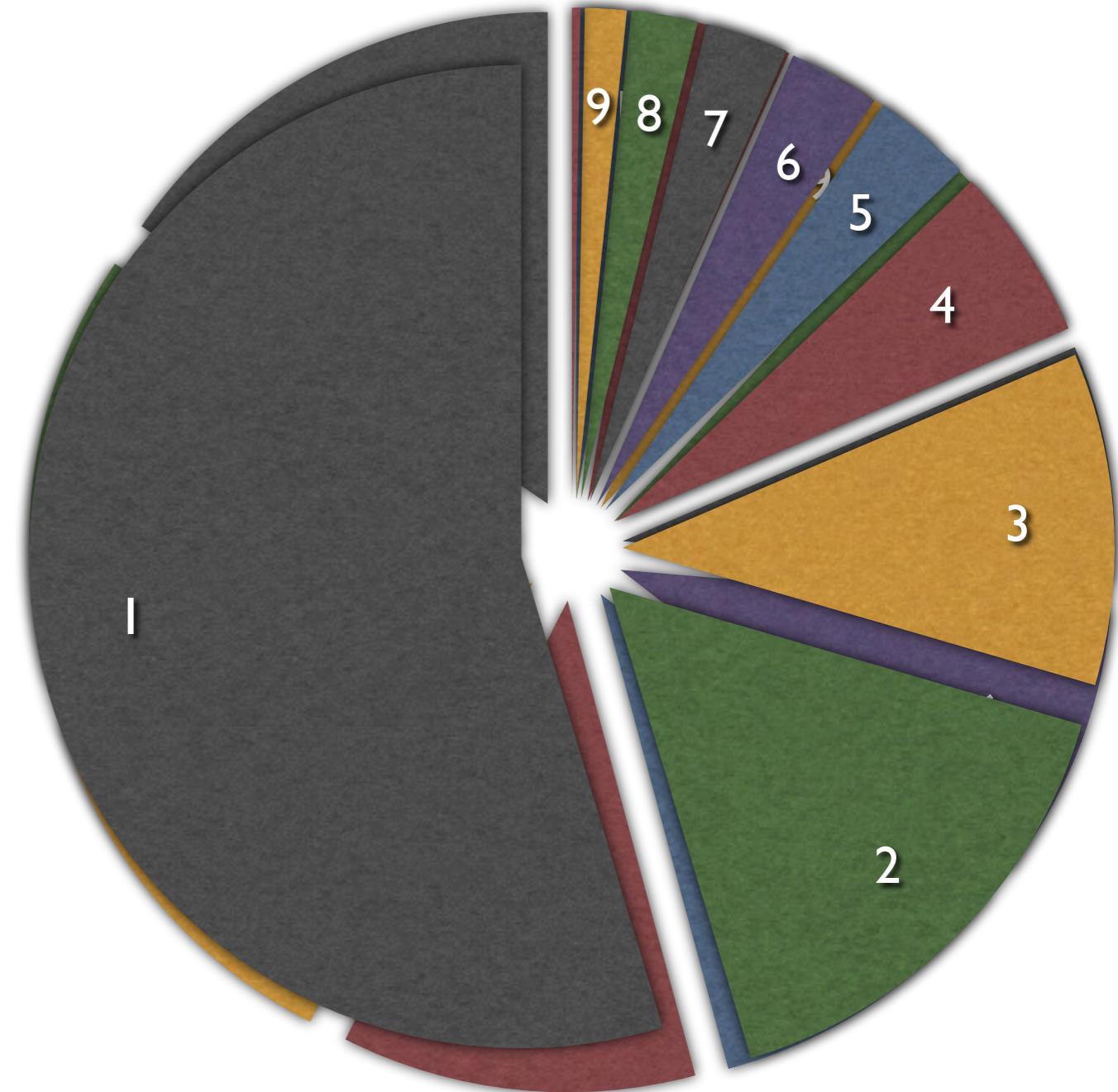
- Ranquear individuos de acuerdo a su fitness
- No hay diferencia si el inidividuo con mayor fitness es diez veces mejor que el segundo o únicamente el 0.001% mejor
- Rank Selection es preferible en la práctica

1

2

3

Individual	Fitness
1	2
2	0.58
3	0.4
4	0.21
5	0.12
6	0.11
7	0.1
8	0.08
9	0.05
10	0.01
11	0



Tournament Selection

- N = Tamaño del Torneo
- Seleccionar N individuos aleatoriamente
- El mejor de los N individuos es seleccionado
- El Tamaño del Torneo define la presión selectiva
- Un individuo con peor fitness aún puede vencer con una cierta probabilidad



Individual	Fitness
1	2
2	1.8
3	1.6
4	1.4
5	1.2
6	1
7	0.8
8	0.6
9	0.4
10	0.2
11	0

Tournament size: 4

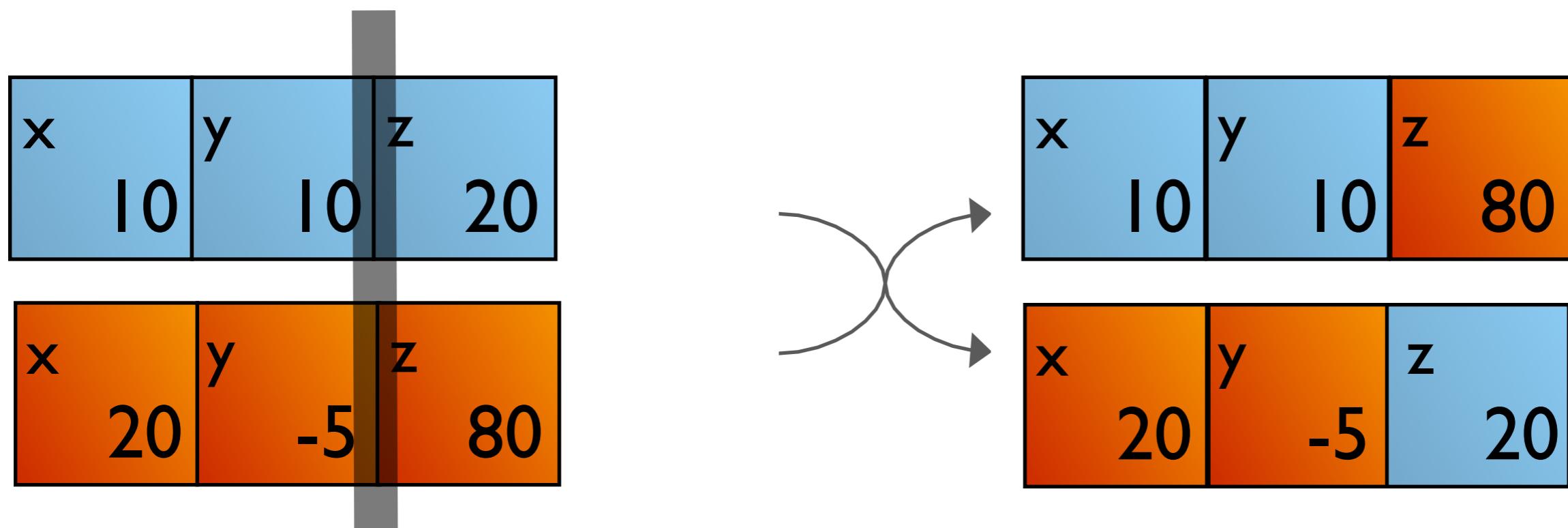
4
6
2
8

2 (1,8)



Crossover

```
def testMe(x, y, z):  
    if x * z == 2 * (y + 1):  
        return True //branch a cubrir  
    else:  
        return False
```

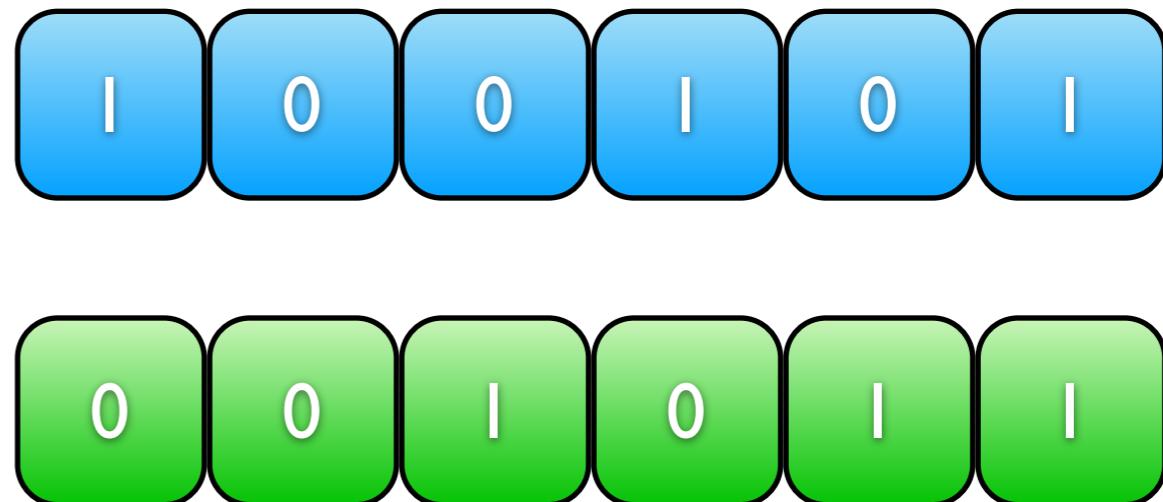


Crossover

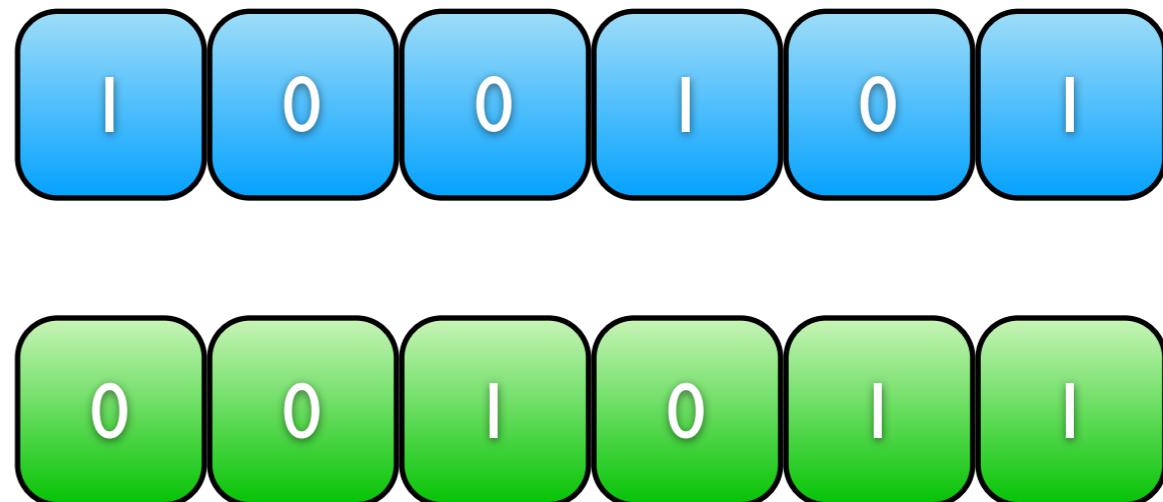
- Single-point crossover
Elegir un único punto en los padres y dividir/unir en ese punto
- Two-point crossover
Elegir dos puntos en los padres e intercambiar la parte interior
- Fixed vs. variable length
Igual punto de crossover en ambos padres - descendencia con tamaño constante
- Uniform crossover
Los Genes son aleatoriamente seleccionados de cada parente

Crossover

Two-point
crossover



Variable length
crossover

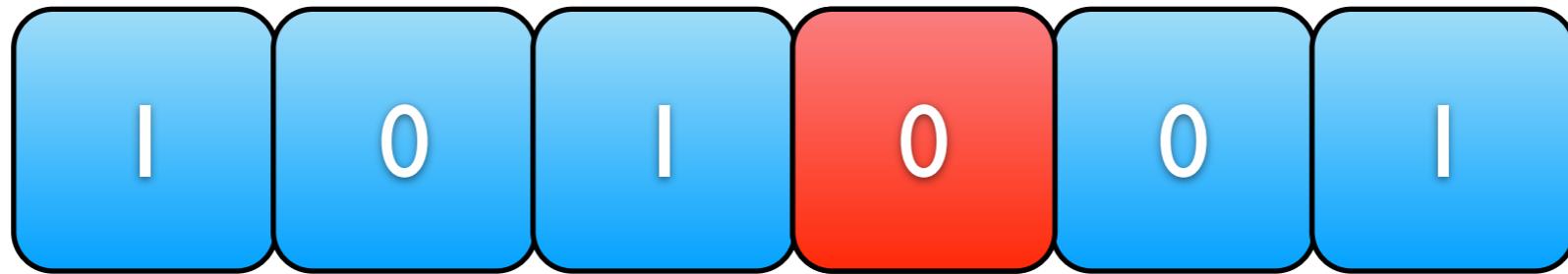


Mutación

```
def testMe(x, y, z):  
    if x * z == 2 * (y + 1):  
        return True //branch a cubrir  
    else:  
        return False
```

x	y	z
20	10	20

Mutación



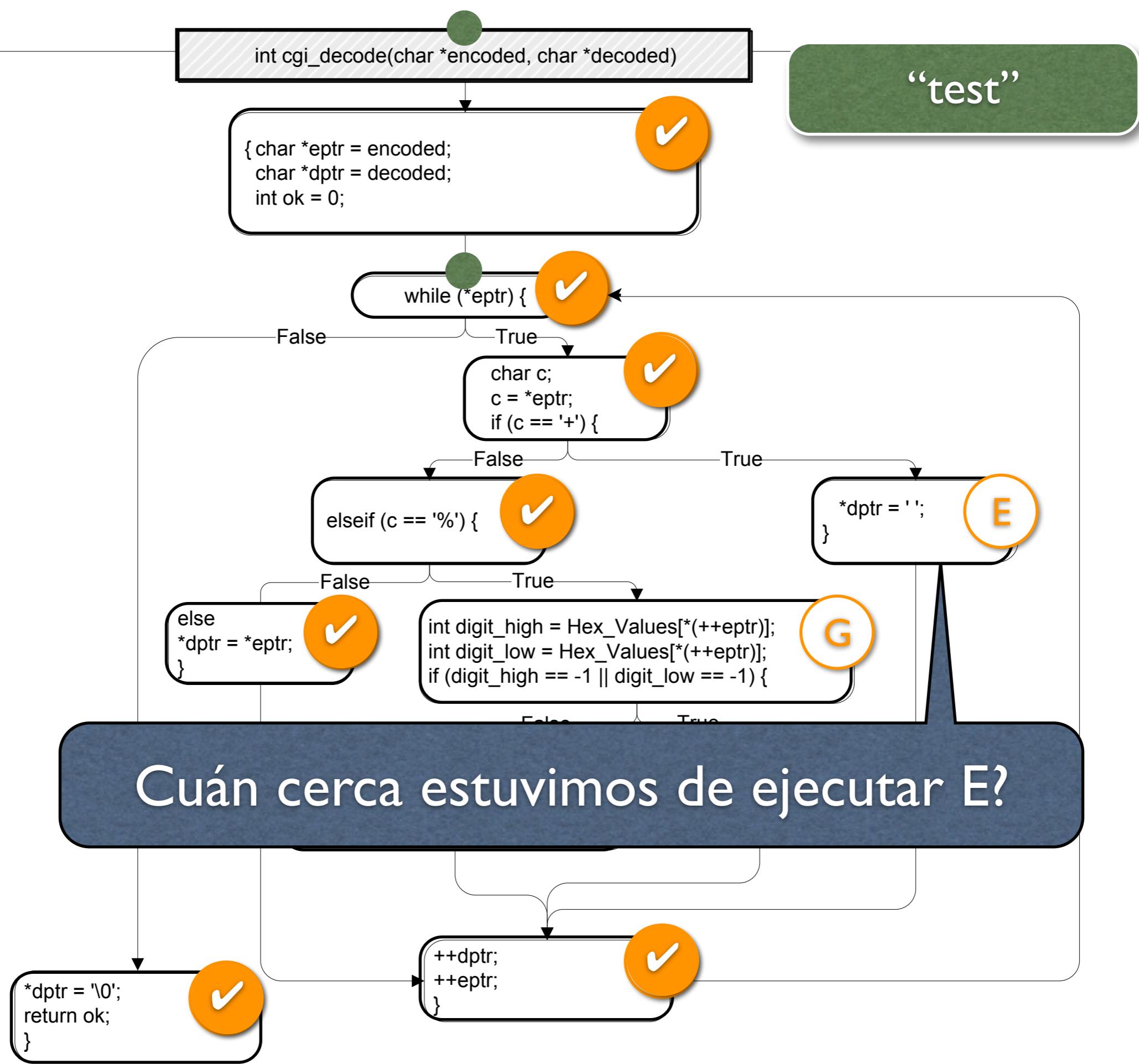
- Cambio en los genes
- La Mutación depende de los operadores genéticos
- Si la representación es binaria - bit flip

Ejemplo:

```
def testMe(x, y, z):
    if x==10:
        if y==20:
            if x * z == 2 * (y + 1):
                return True //branch a cubrir
    return False
```

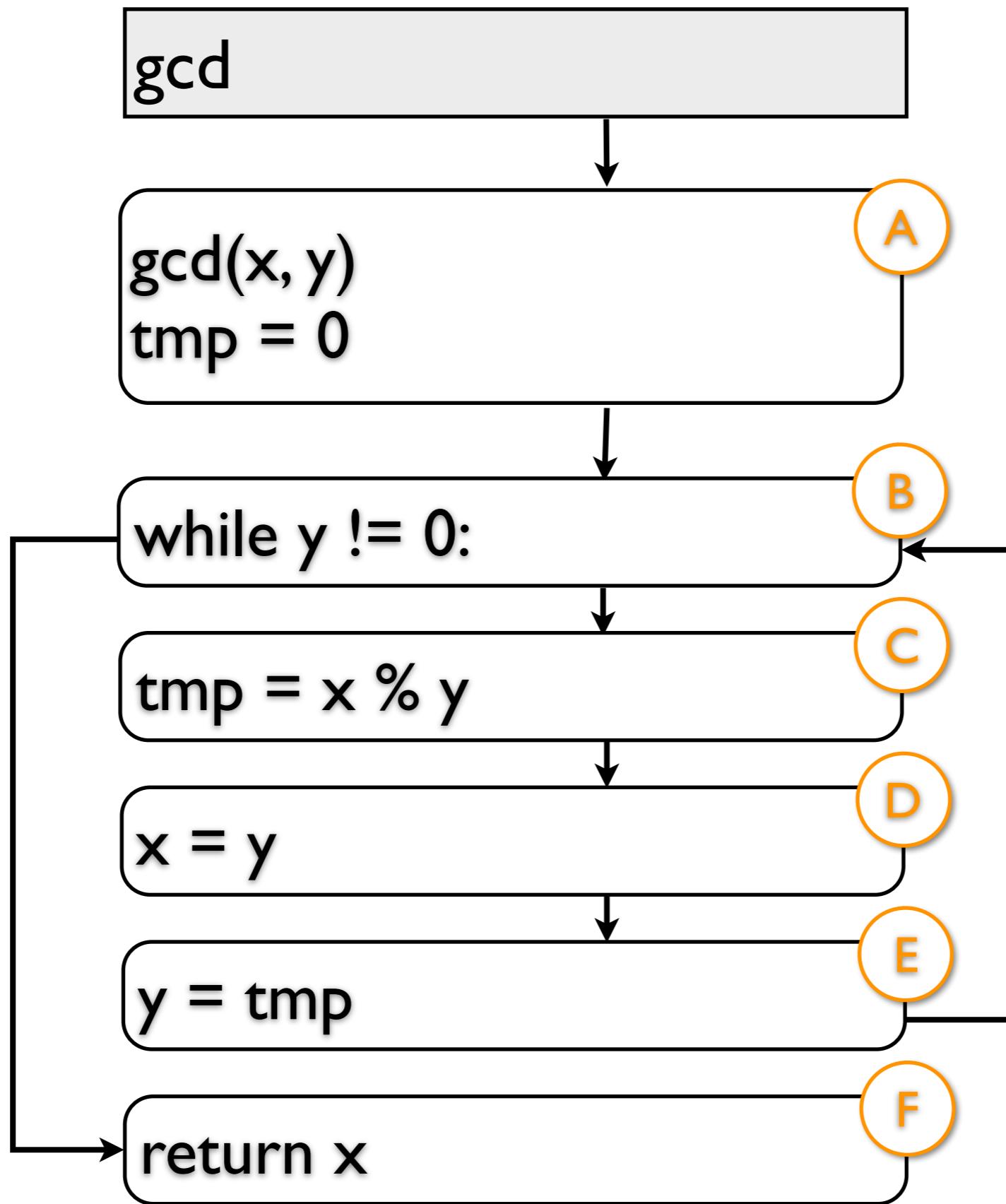
- ¿Qué pasa si el predicado no fue alcanzado por el individuo?

CFG



```
def gcd(x, y):  
    tmp = 0  
    while y != 0:  
        tmp = x % y  
        x = y  
        y = tmp  
  
    return x
```

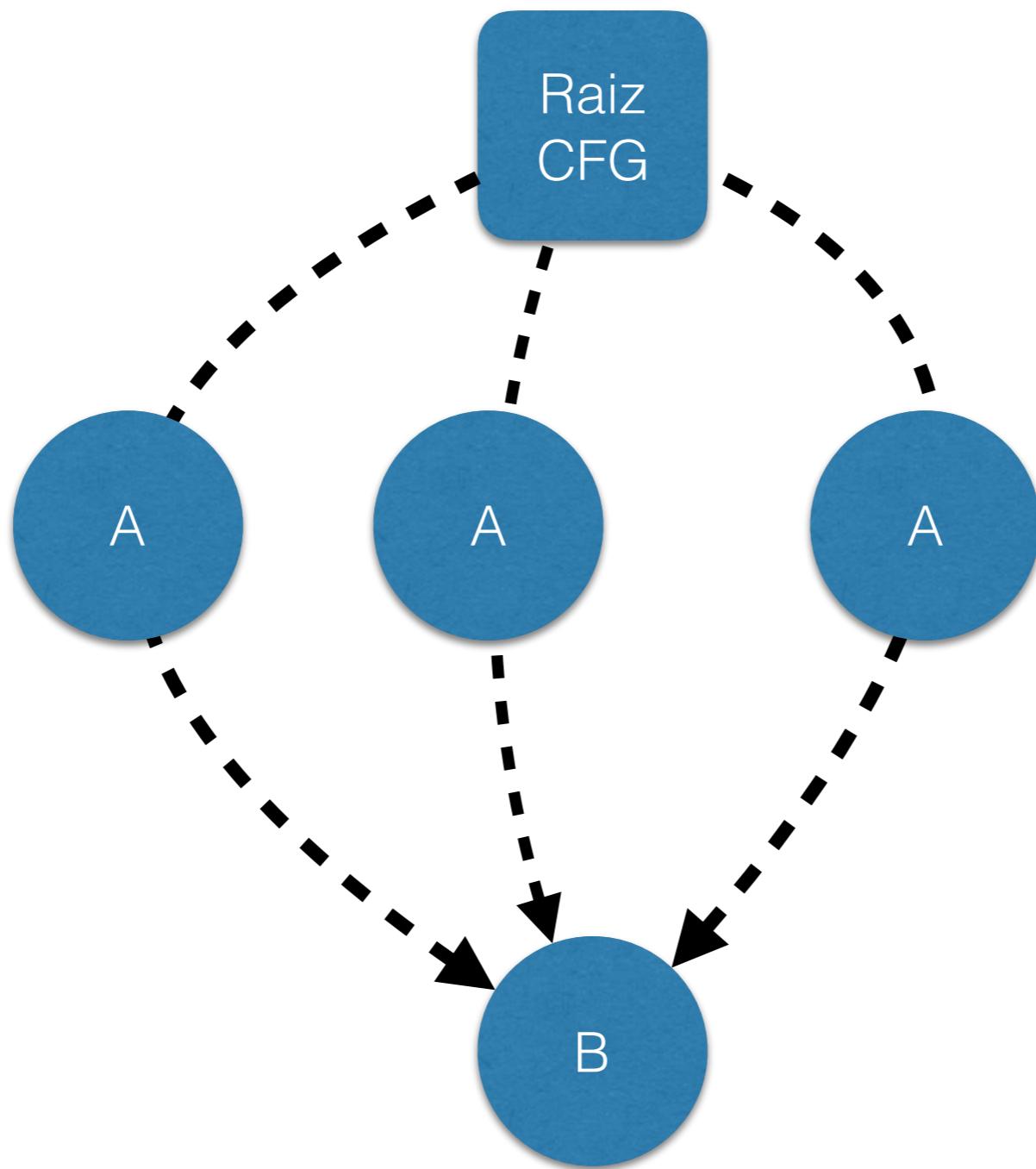
CFG



“Dominators”

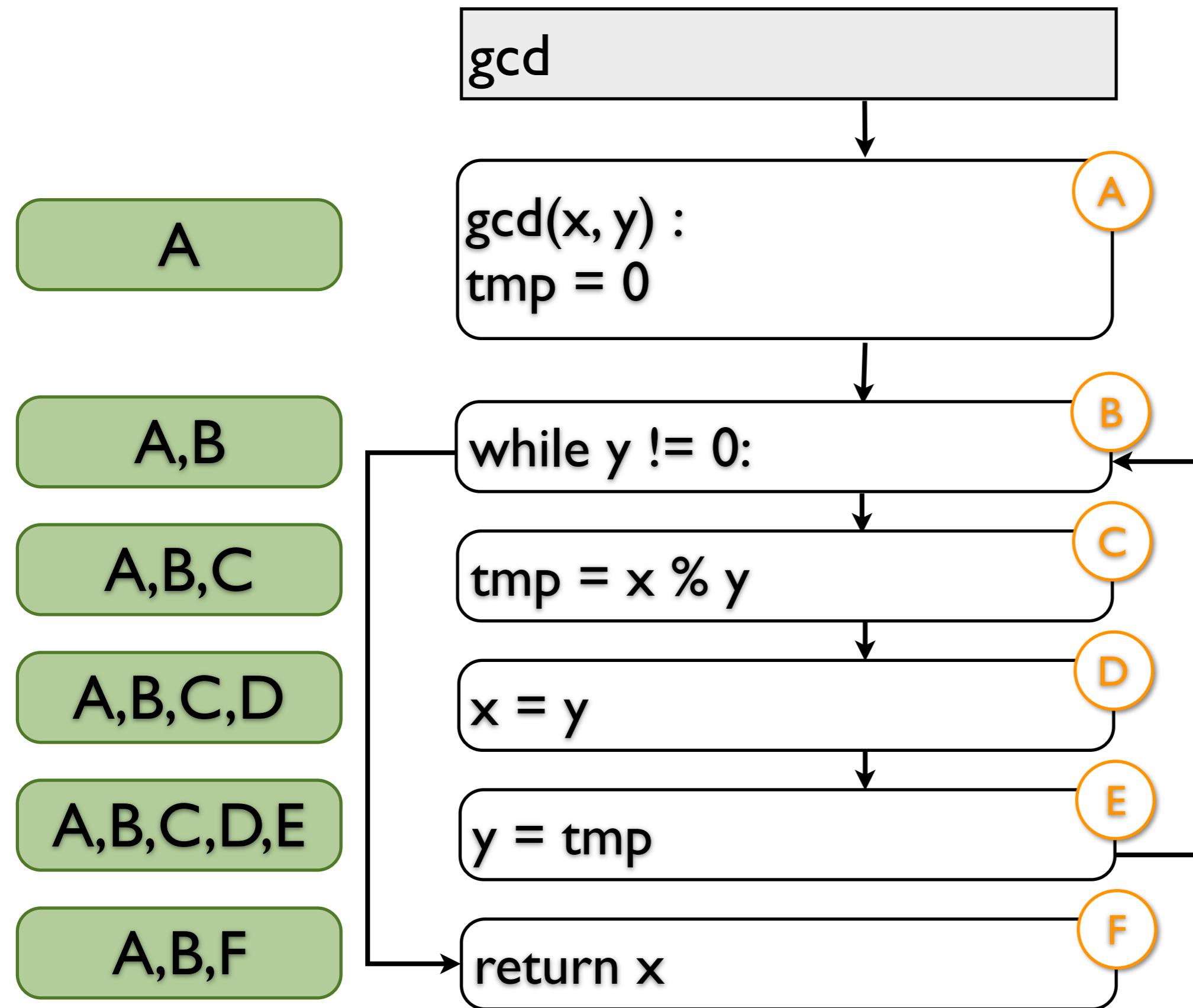
- El Nodo A domina al Nodo B si todo camino hacia B debe pasar por A
- Dominador Inmediato: El dominador mas cercano en todo camino desde la raíz
- La raíz no tiene dominador inmediato

A domina a B si:



Caminos en el CFG

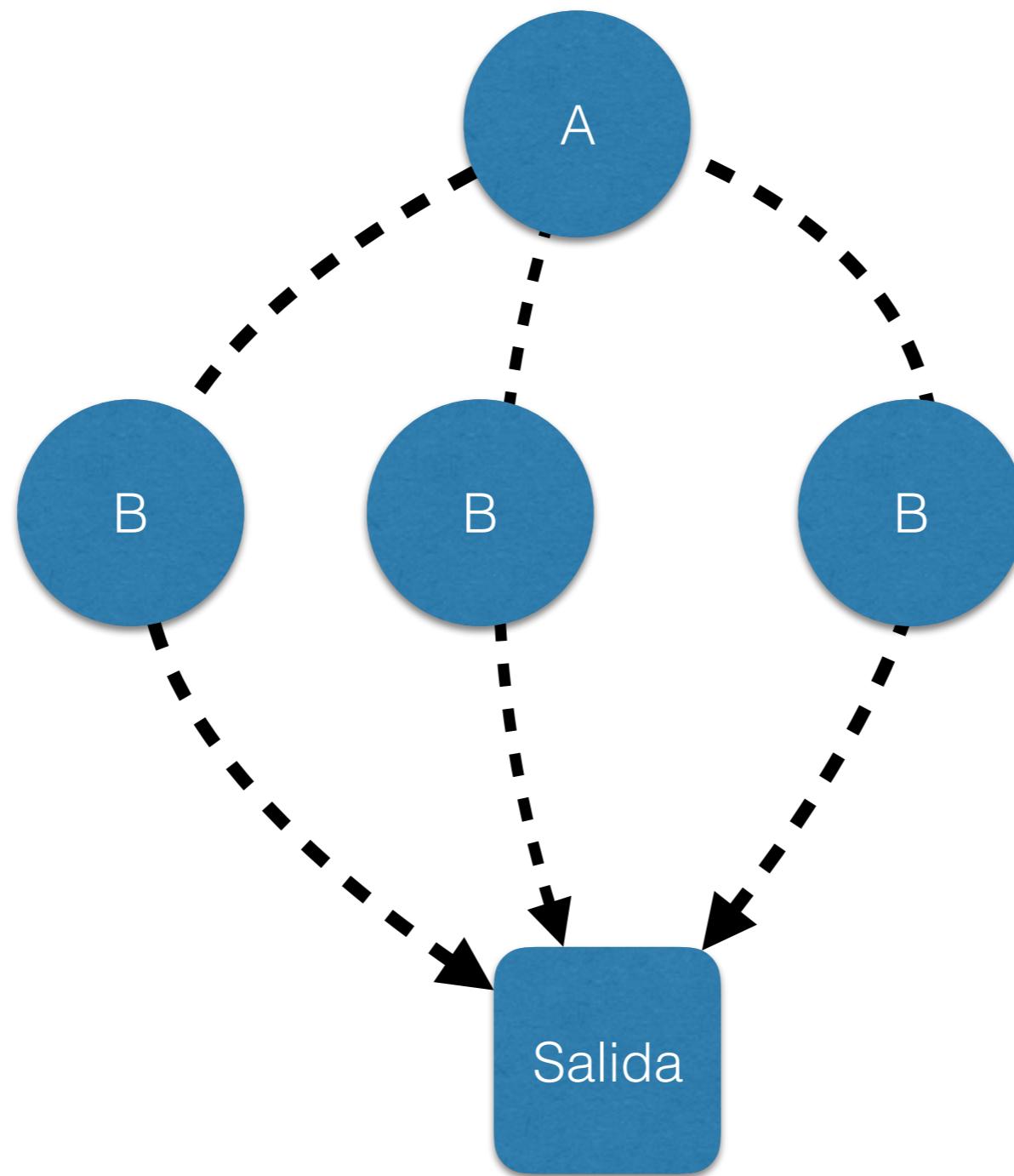
CFG



Post Dominators

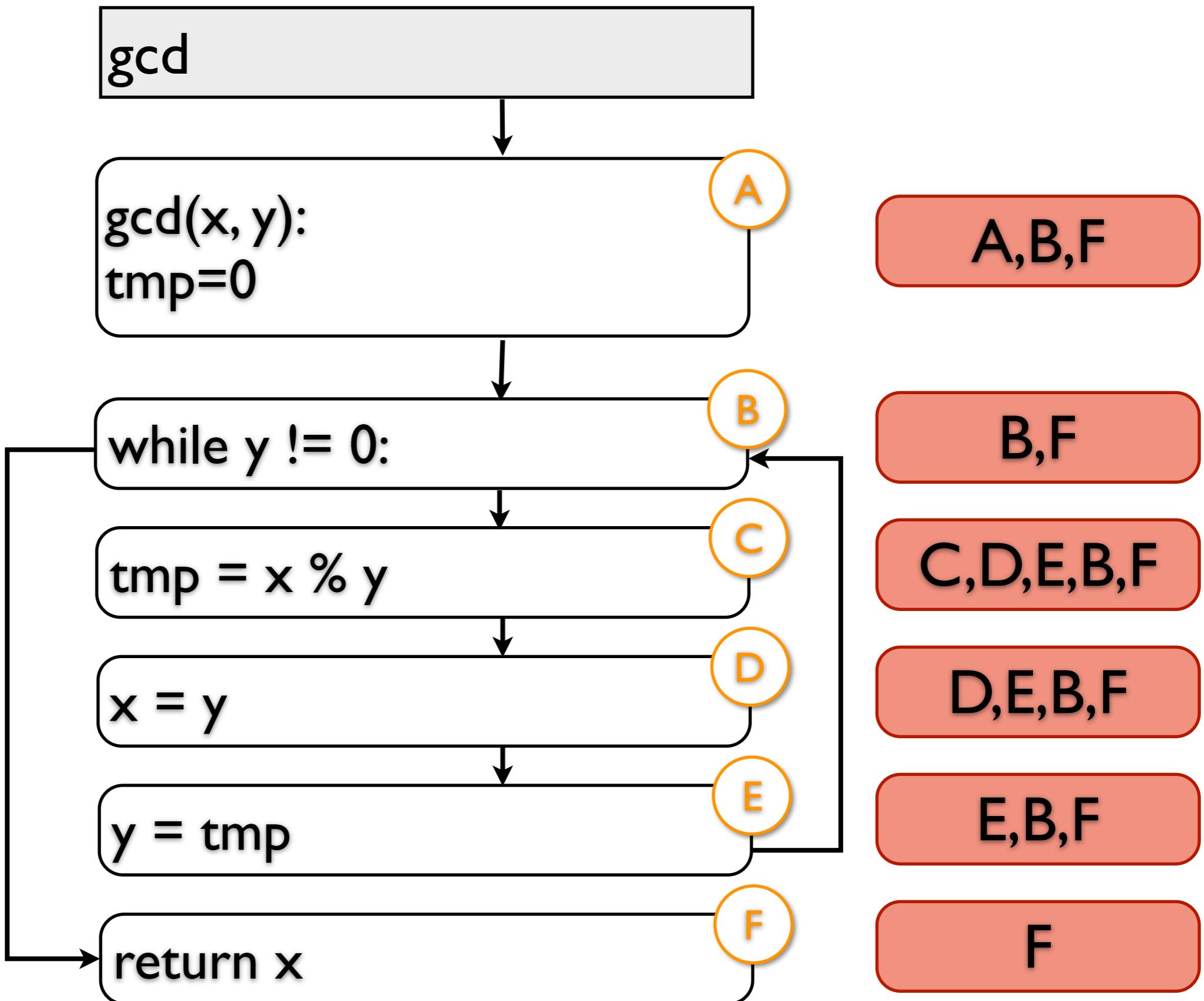
- El Nodo B “post-domina” al Nodo A si todos los caminos desde A a la salida deben pasar por B
- Post dominador Inmediato: El dominador más cercano en todo camino hacia el nodo de salida
- Son Dominadores vistos en reversa (caminos desde el nodo de salida)

B post-domina a A si:



Caminos en el CFG

CFG



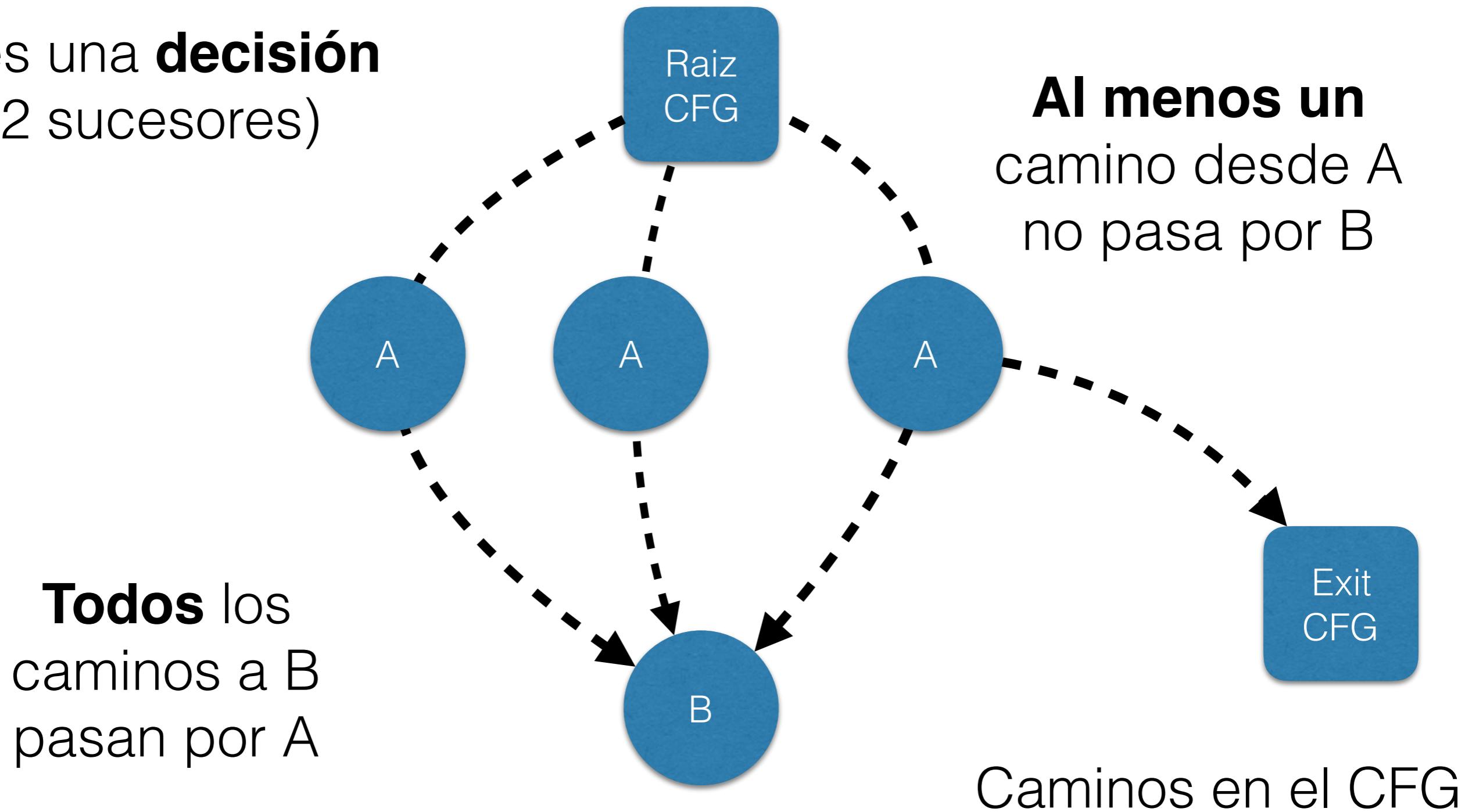
Control Dependence

- B es control-dependiente de A si:
 - A domina a B
 - B no “post-domina” a A
 - A tiene al menos dos sucesores
 - B post-domina a un sucesor de A

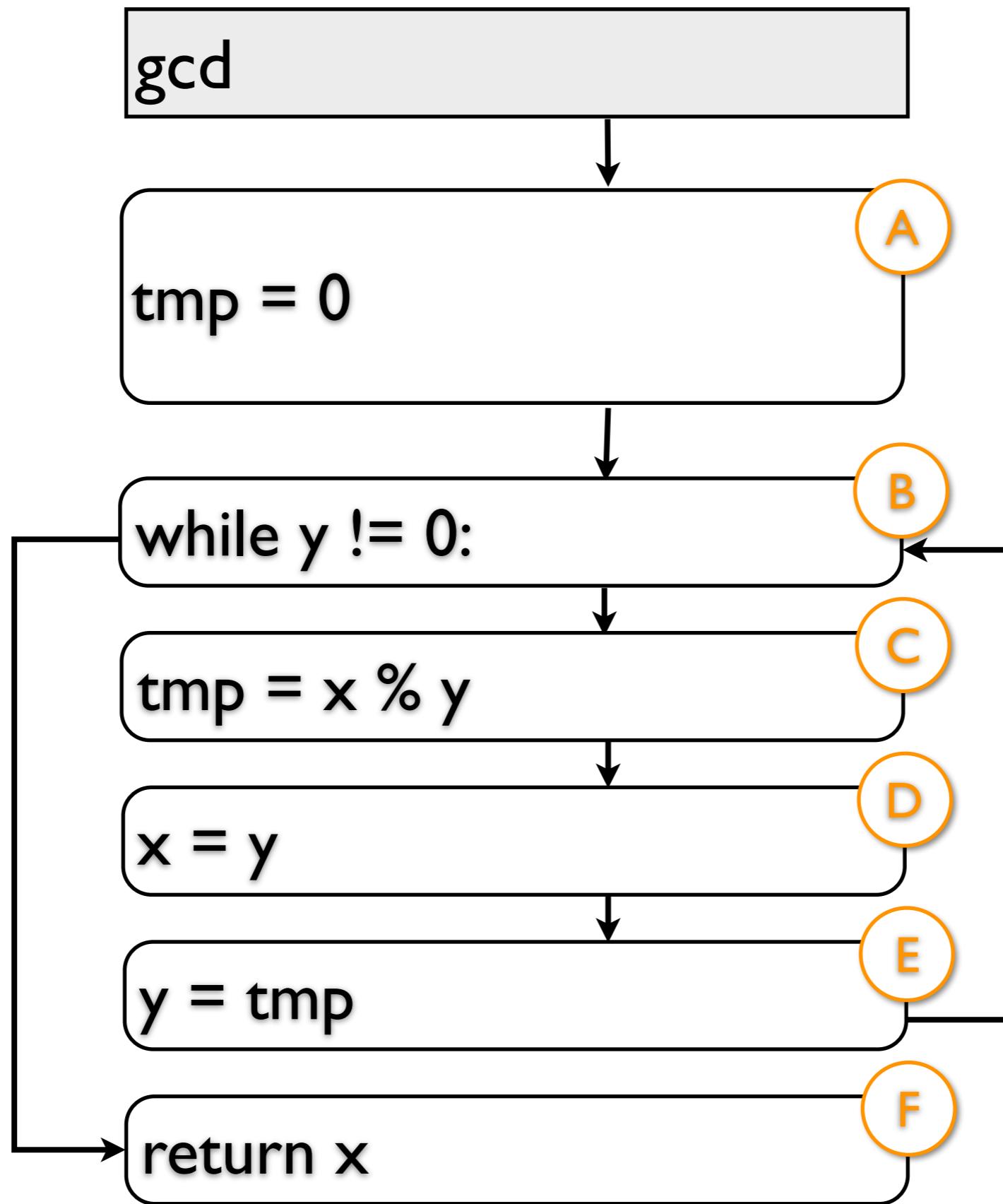
B es control-dependent de A si:

A es una **decisión**
(2 sucesores)

Al menos un
camino desde A
no pasa por B

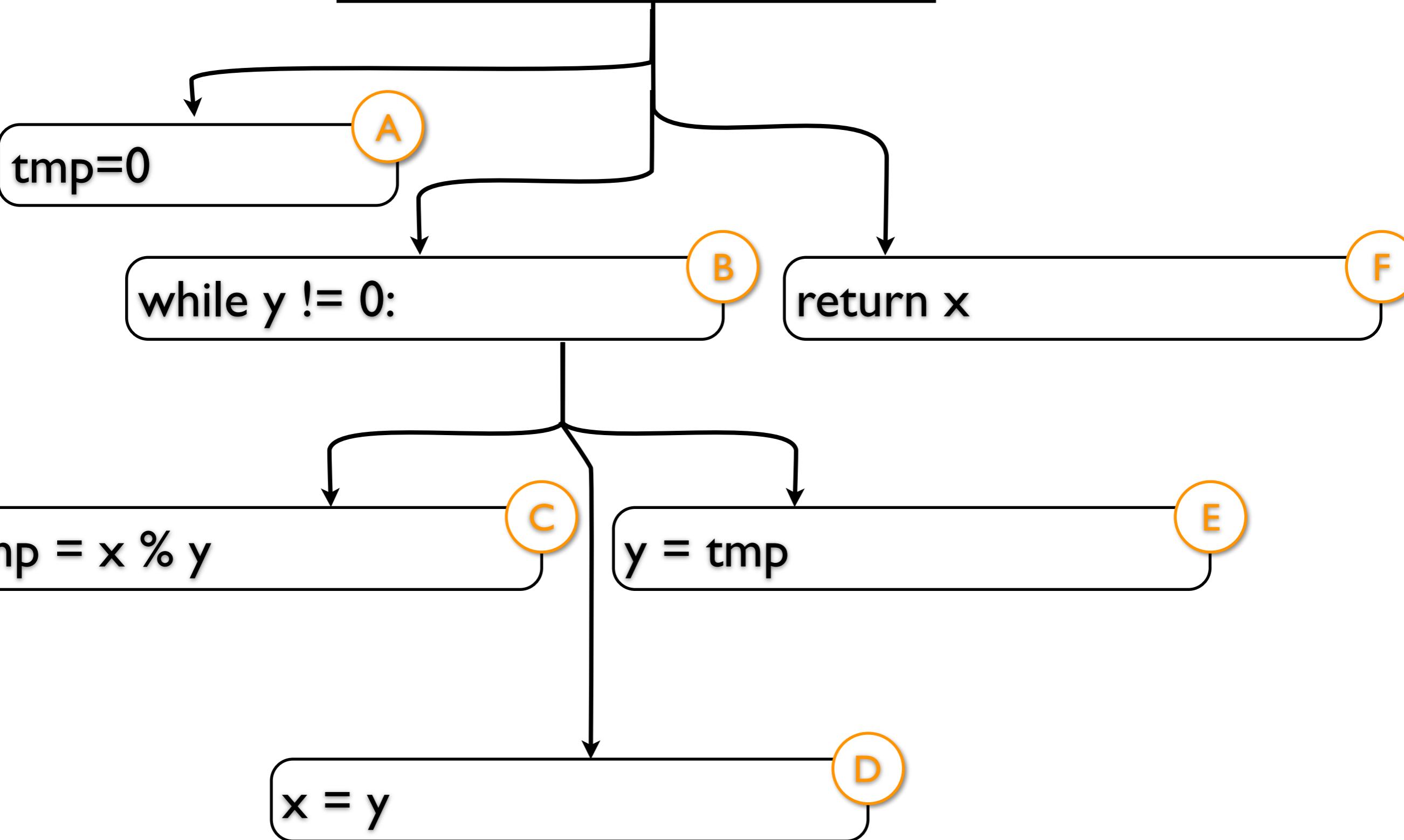


CFG

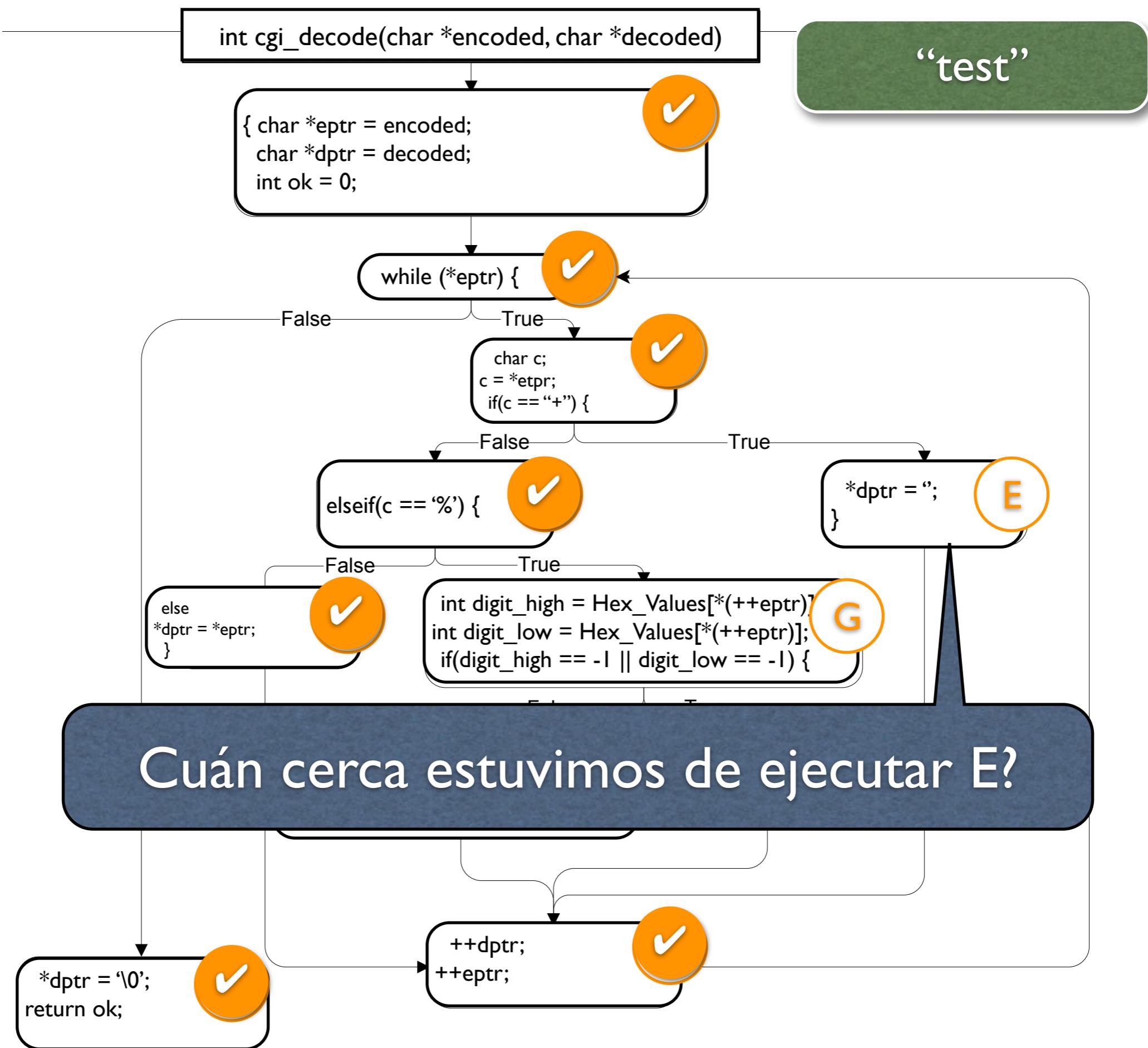


CDG

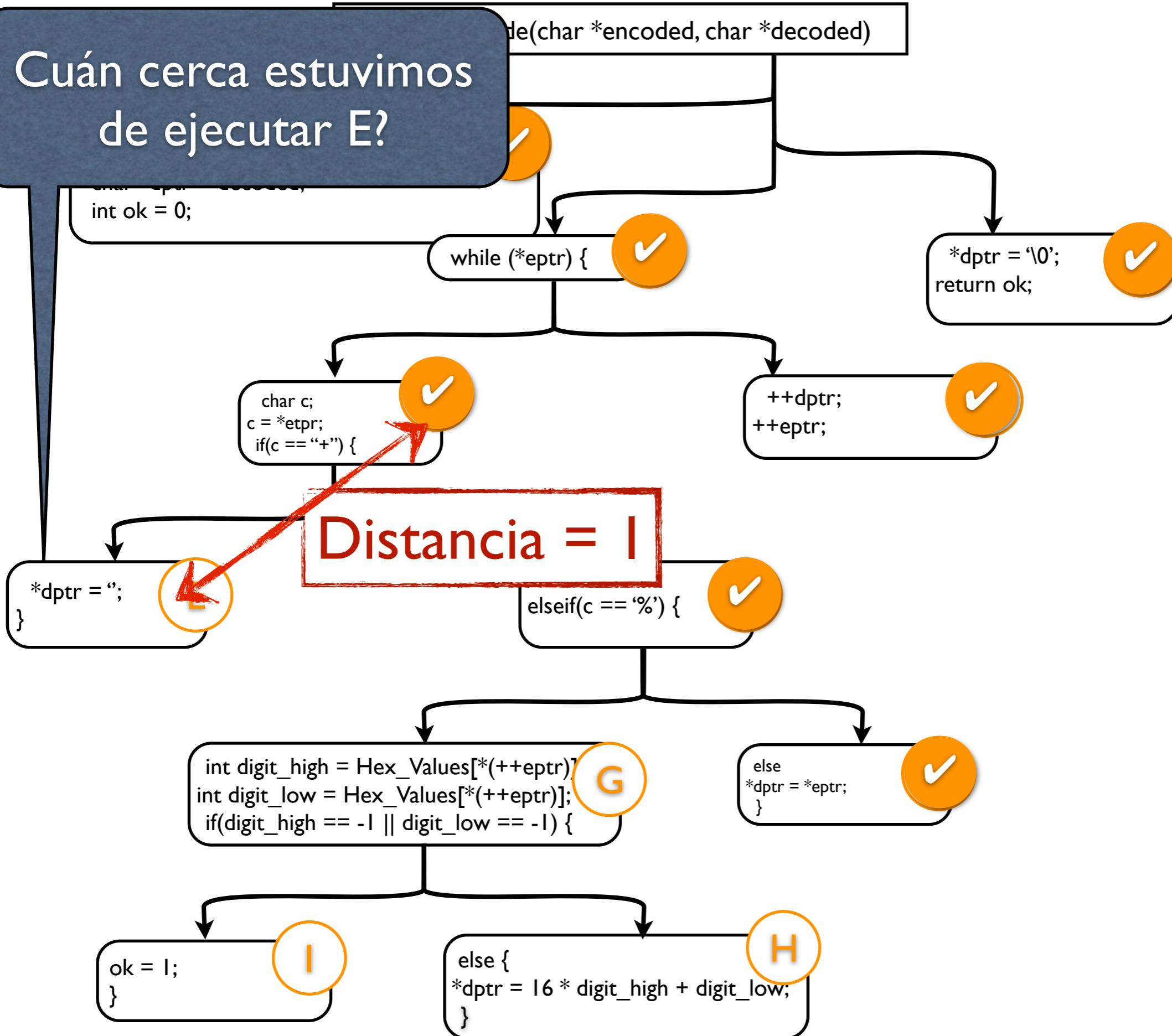
gcd (root)



CFG



Cuán cerca estuvimos de ejecutar E?



Fitness Function

- Dependent=number of control dependent nodes for the current target (i.e. los ancestros del target)
- Executed=number of control dependent (ancestros) successfully executed
- FF=(Dependent-Executed)

Approach Level



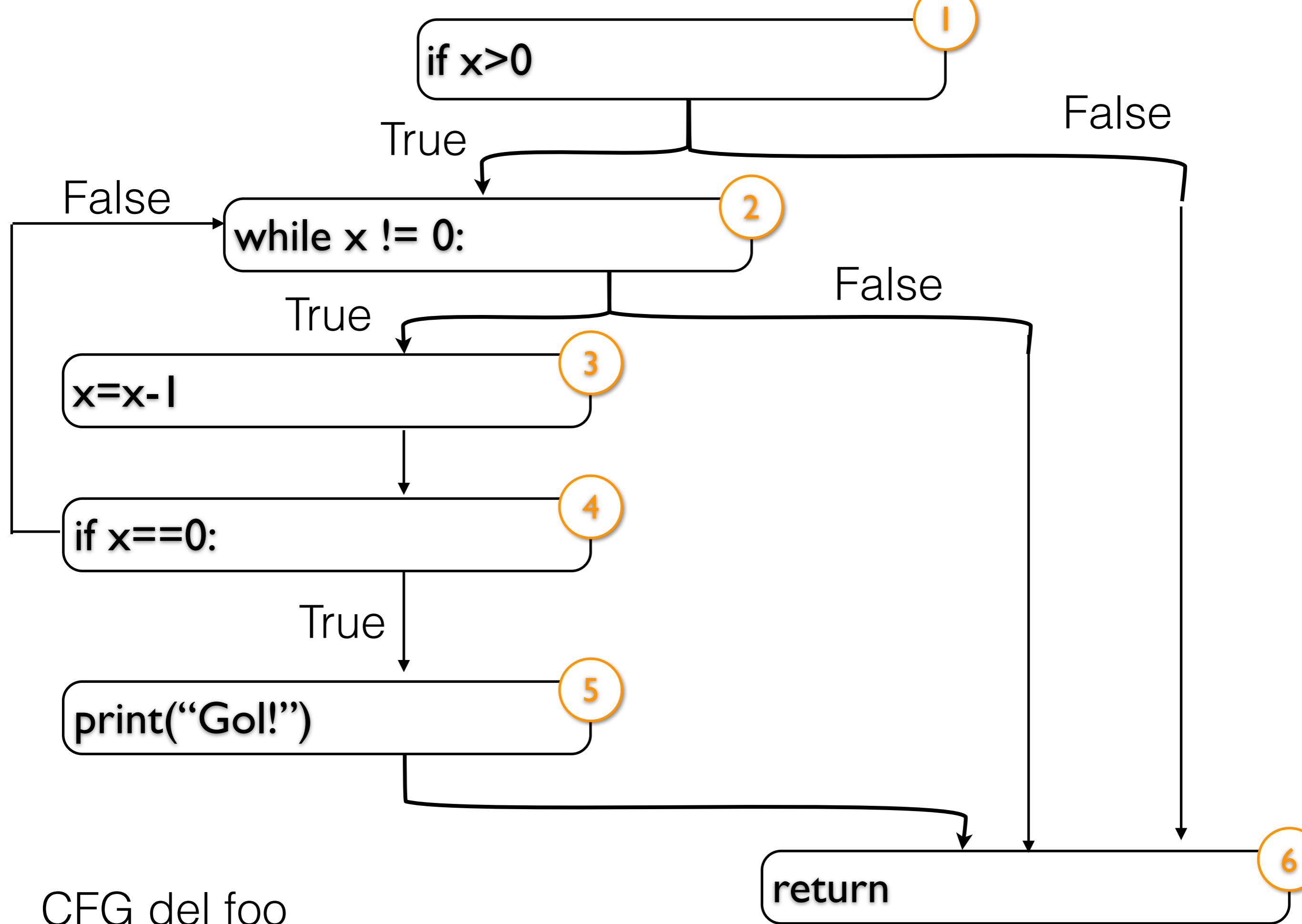
- Control Distance=Number of control dependent edges between goal and chosen path

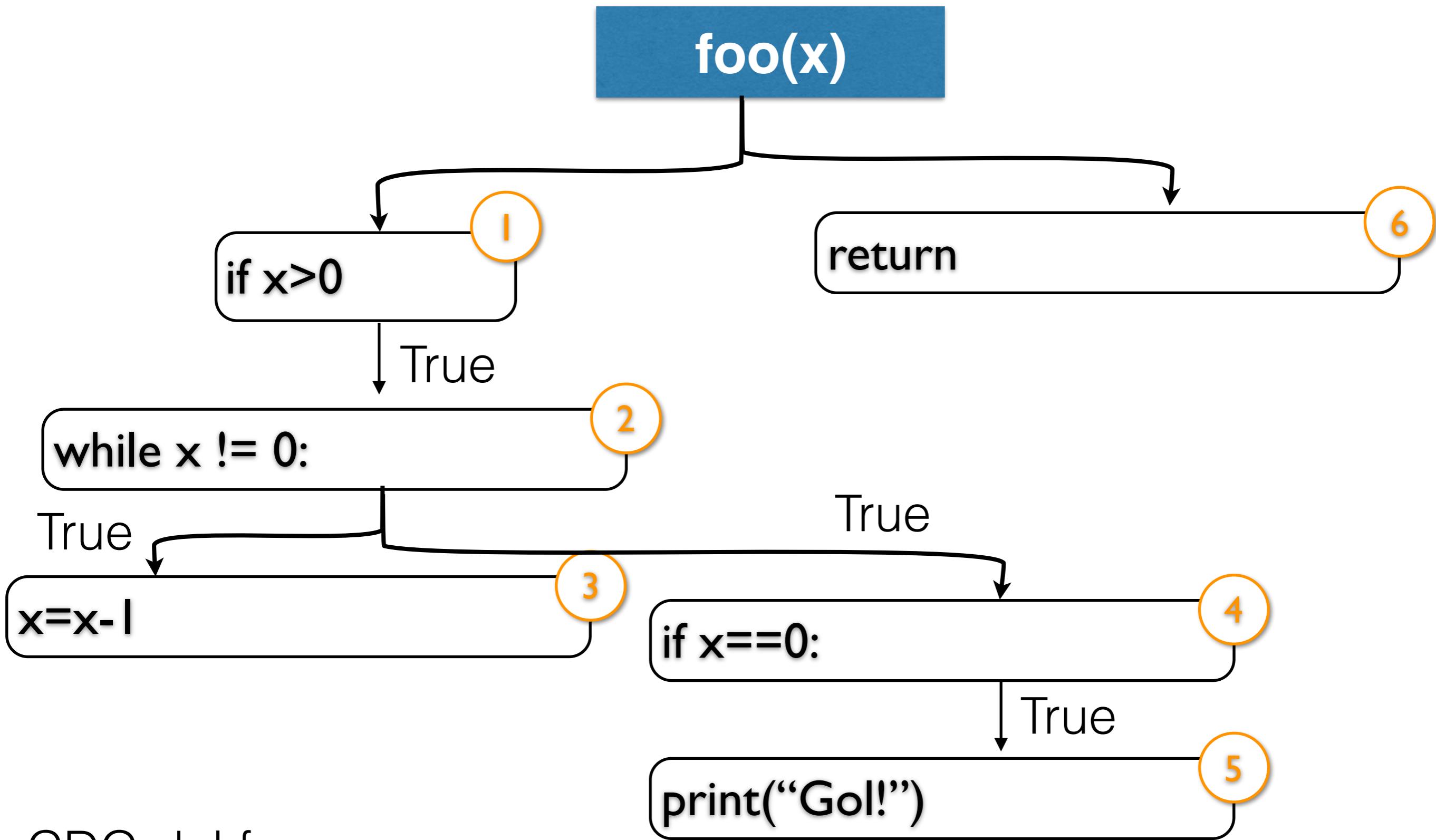
Fitness Function

- Para **guiar** al algoritmo de búsqueda necesitamos que la fitness function combine:
 - El “approach level” (i.e. cuán cerca estoy del predicado)
 - La “branch distance” (i.e. cuán cerca ejercitar el branch)

Fitness Function

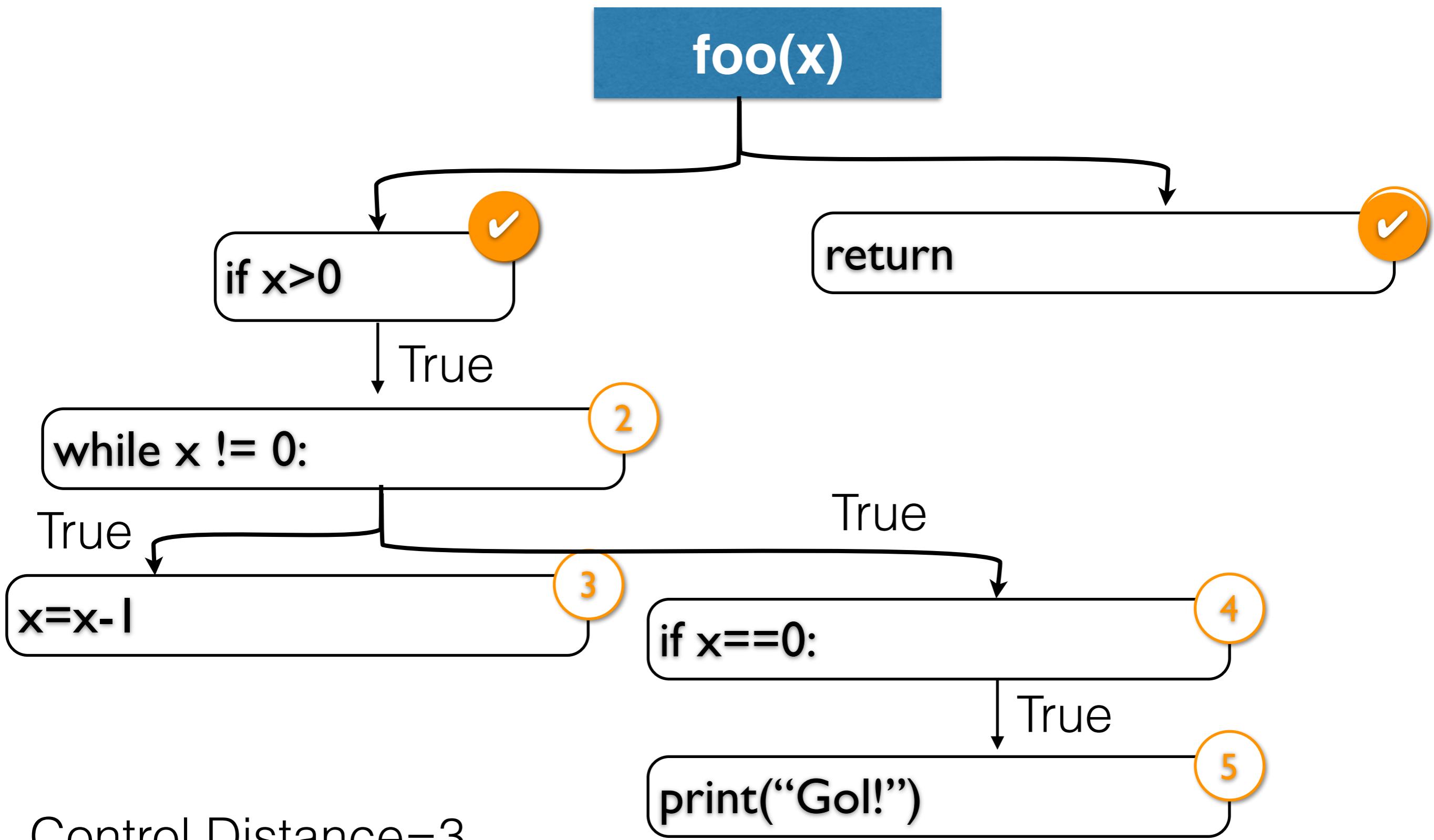
```
def foo(x):
L1: if x>0:
L2:   while x != 0:
L3:     x=x-1
L4:     if x==0:
L5:       print("Gol!")
L6: return
```



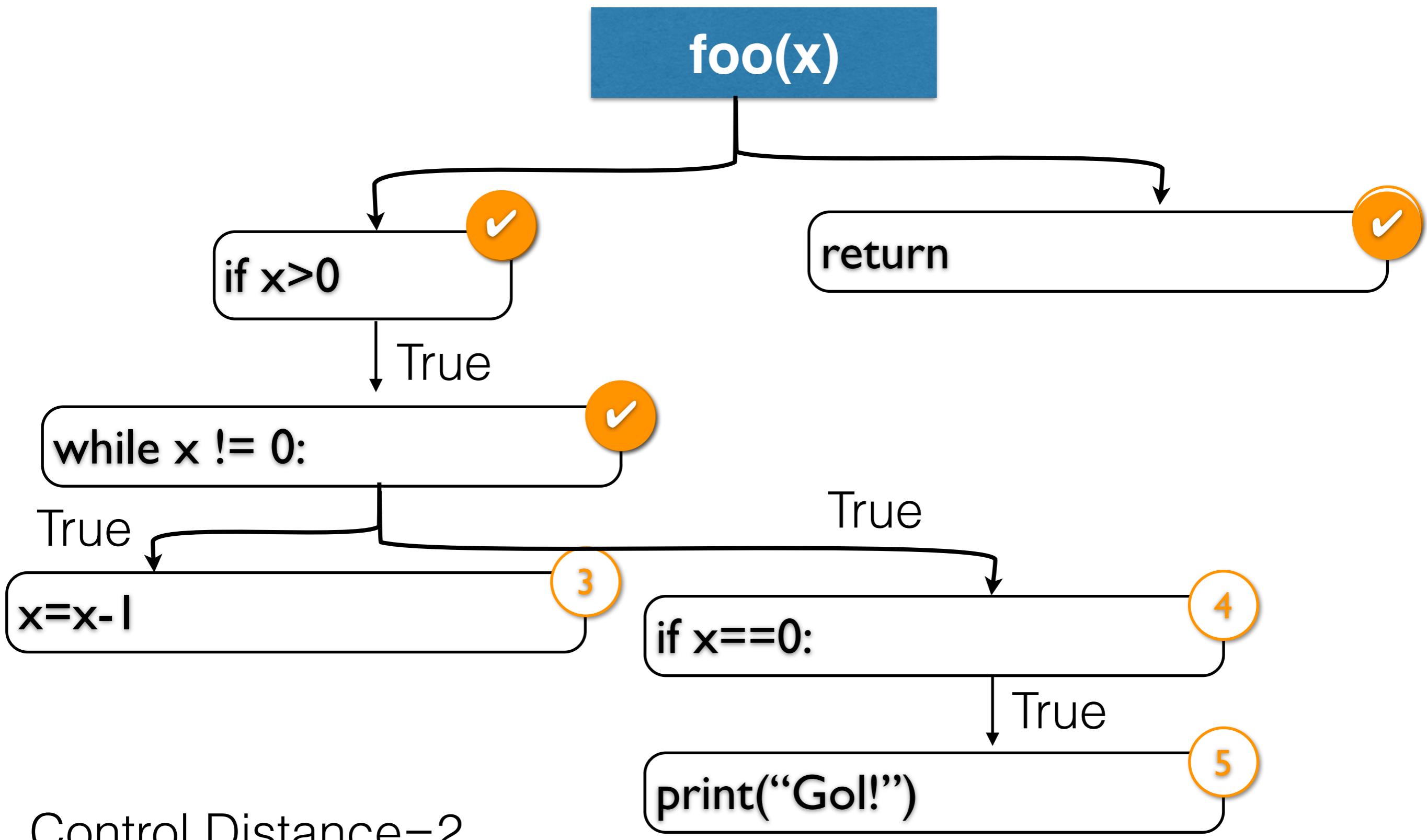


CDG del foo

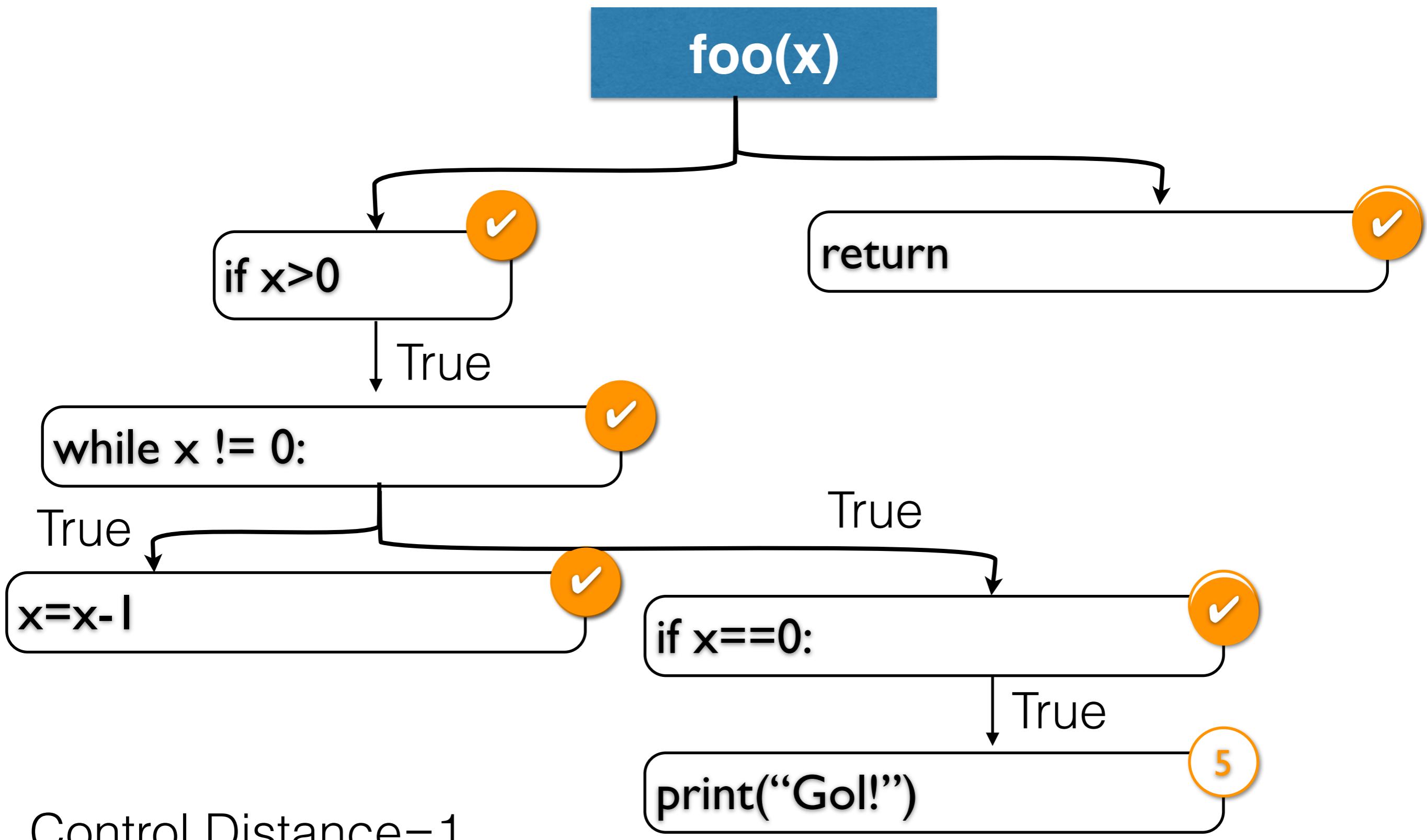
Test1()



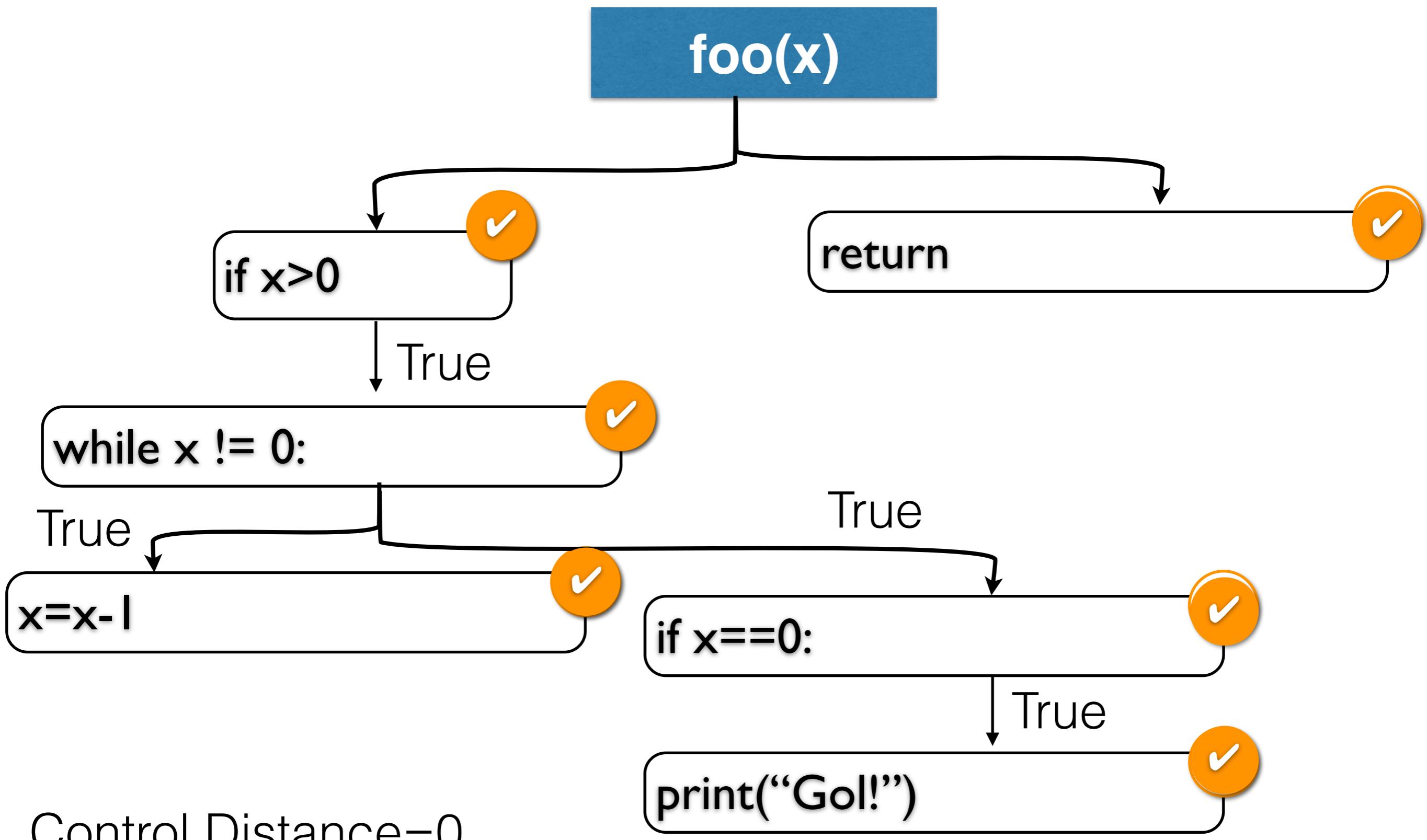
Test2()



Test3()



Test3()

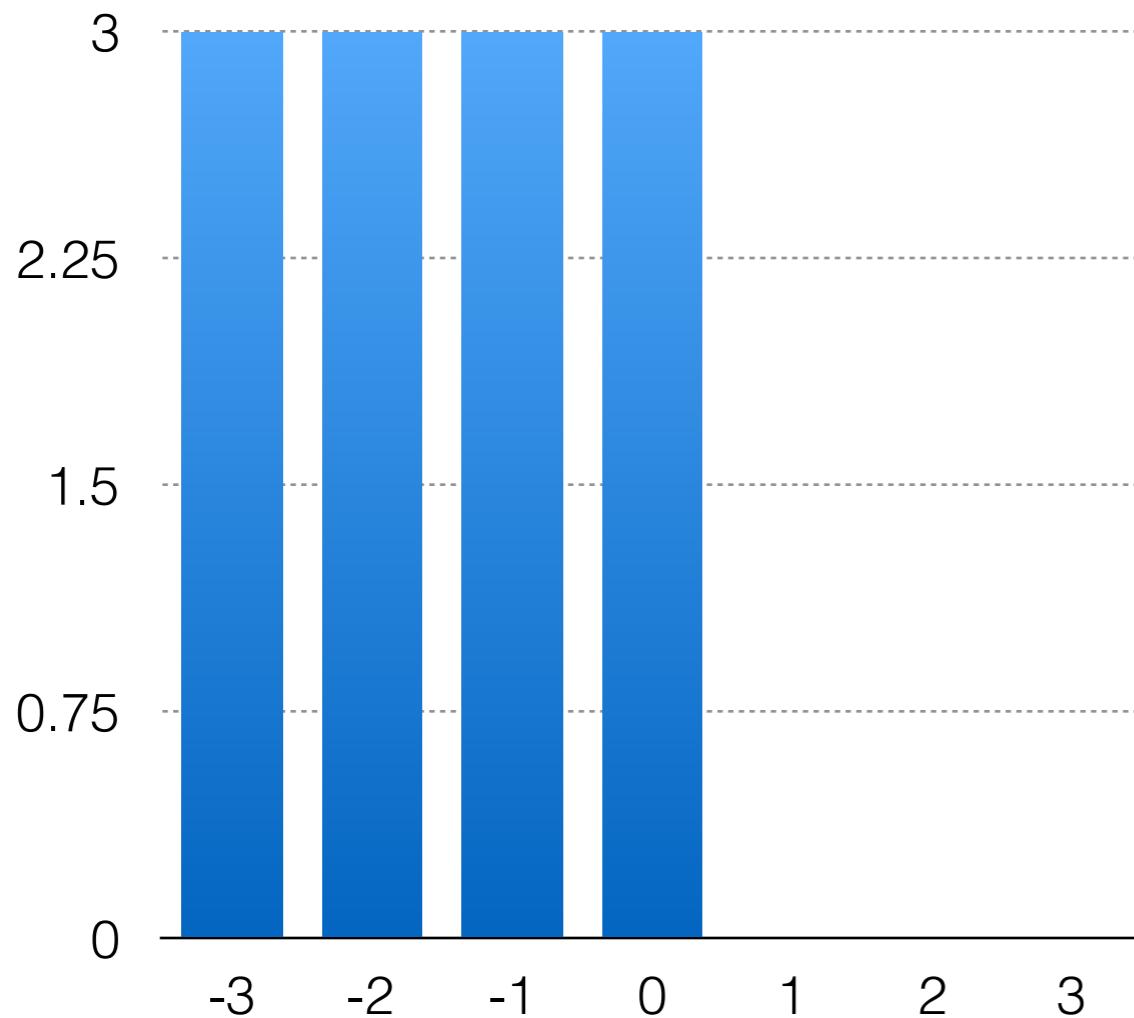


Control Distance

- Sea t un camino acíclico desde la raíz del CDG hasta el branch **G**:
 - Computar el “**approach level** hasta G” (i.e. nodos dependientes - nodos ejecutados)
- Fitness Function= el mínimo **approach level** a G usando alguno de los caminos desde la raíz a G
- ¿Cuál será el landscape usando control distance como fitness function?

Control Distance

- Fitness Function(x) = Control Distance de ejecutar $\text{foo}(x)$



- Rugged Fitness Function
- ¿Cómo podemos mejorar la función de fitness para que ayude guiar la búsqueda?

Control+Branch Distance

- Sea t un camino acíclico desde la raíz del CDG hasta el branch **G**:
 - Computar el “**approach level** hasta G” (i.e. nodos dependientes - nodos ejecutados)
 - Sumar el **branch distance** del predicado mas cercano alcanzo

Control+Branch Distance

- Fitness Function= De todos los caminos de la raíz del CDG al branch G, la mínima suma de
 - Approach Level
 - Branch Distance
- ¿Qué pasará si la branch distance es un número muy grande?

Normalization

- Queremos evitar que la branch distance “**domine**” la función de fitness:
 - Si el último branch es “if $x==100000$ ” (y además $x=1$)
 - Entonces la Branch Distance es 99999
 - Este valor desalienta elegir esta solución

Normalization Functions

- Nos permiten normalizar todos los valores aunque sean de distinta magnitud
 - Ejemplo: $v / v+1$
- El rango de normalización es $[0,1]$
- La Fitness Function aplica la normalización sobre la Branch Distance

Control+ Normalized Branch Distance

- Sea t un camino acíclico desde la raíz del CDG hasta el branch **G**:
 - Computar el “**approach level** hasta G” (i.e. nodos dependientes - nodos ejecutados)
 - Sumar la normalización de la **branch distance** del predicado mas cercano alcanzo
- De todos los caminos de la raíz del CDG al branch G, la mínima suma de approach level y branch distance normalizada.

Control+Normalized Branch Distance

Fitness
Funcition(x)=5,75



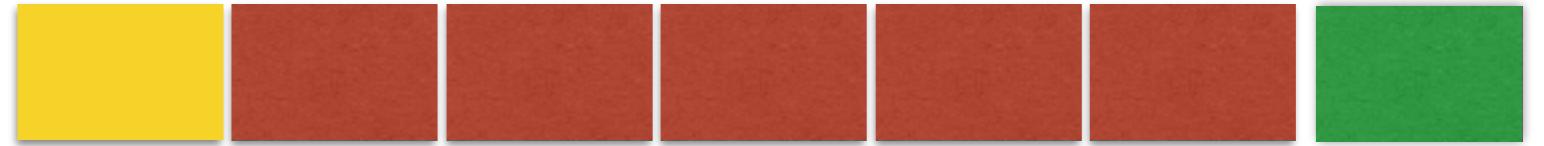
Goal G

Quedan 5 Nodos
Sin Alcanzar

Estoy a una distancia
de 0,75 de cubrir
el branch divergente

Control+Normalized Branch Distance

Fitness
 $\text{Funcition}(x)=5,05$



Quedan 5 Nodos
Sin Alcanzar

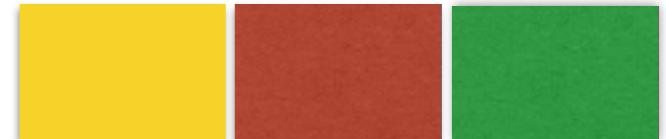
Goal G

Estoy a una distancia
de 0,05 de cubrir
el branch divergente

Control+Normalized Branch Distance

Fitness
Funcition(x)=1,05

Estoy a una distancia
de 0,05 de cubrir
el branch divergente



Goal G

Quedan solo 1 Nodo
Sin Alcanzar

Control+Normalized Branch Distance

Estoy a una distancia
de 0,05 de cubrir
el goal

Fitness
 $\text{Funcition}(x)=0,05$



Goal G

Todos los nodos
han sido alcanzados

```
# INPUT: Program, TargetBranch, Test
# OUTPUT: Fitness to target branch
def fitness(program, targetBranch, test):
    # ejecutar test y computar branch distances
    branchDistances = execute(test)
```

```
# INPUT: Program, TargetBranch, Test
# OUTPUT: Fitness to target branch
def fitness(program, targetBranch, test):
    # ejecutar test y computar branch distances
    branchDistances = execute(test)

    # computar control dependence graph del programa
    cdg = create_control_dependence_graph(program)

    # enumerar todos los caminos que van de la raiz al target branch
    paths=compute_all_paths_from_root(cdg,targetBranch)
```

```
# INPUT: Program, TargetBranch, Test
# OUTPUT: Fitness to target branch
def fitness(program, targetBranch, test):
    # ejecutar test y computar branch distances
    branchDistances = execute(test)

    # computar control dependence graph del programa
    cdg = create_control_dependence_graph(program)

    # enumerar todos los caminos que van de la raiz al target branch
    paths=compute_all_paths_from_root(cdg,targetBranch)

    # fitness minima hasta el momento
    min_path_fitness = sys.maxint
```

```
# INPUT: Program, TargetBranch, Test
# OUTPUT: Fitness to target branch
def fitness(program, targetBranch, test):
    # ejecutar test y computar branch distances
    branchDistances = execute(test)

    # computar control dependence graph del programa
    cdg = create_control_dependence_graph(program)

    # enumerar todos los caminos que van de la raiz al target branch
    paths=compute_all_paths_from_root(cdg,targetBranch)

    # fitness minima hasta el momento
    min_path_fitness = sys.maxint

    # computar el fitness de cada camino al target branch
    for path in paths:
        path_fitness = 0
        for branch in paths:
            if not branch in branchDistances.keys():
                # si el predicado no fue cubierto, agregar 1
                path_fitness += 1
```

```
# OUTPUT: Fitness to target branch
def fitness(program, targetBranch, test):
    # ejecutar test y computar branch distances
    branchDistances = execute(test)

    # computar control dependence graph del programa
    cdg = create_control_dependence_graph(program)

    # enumerar todos los caminos que van de la raiz al target branch
    paths=compute_all_paths_from_root(cdg,targetBranch)

    # fitness minima hasta el momento
    min_path_fitness = sys.maxint

    # computar el fitness de cada camino al target branch
    for path in paths:
        path_fitness = 0
        for branch in paths:
            if not branch in branchDistances.keys():
                # si el predicado no fue cubierto, agregar 1
                path_fitness += 1
            else:
                # si el predicado fue cubierto, normalizar branch distance
                path_fitness += normalize(branchDistance[b])
```

```
# computar control dependence graph del programa
cdg = create_control_dependence_graph(program)

# enumerar todos los caminos que van de la raiz al target branch
paths=compute_all_paths_from_root(cdg,targetBranch)

# fitness minima hasta el momento
min_path_fitness = sys.maxint

# computar el fitness de cada camino al target branch
for path in paths:
    path_fitness = 0
    for branch in paths:
        if not branch in branchDistances.keys():
            # si el predicado no fue cubierto, agregar 1
            path_fitness += 1
        else:
            # si el predicado fue cubierto, normalizar branch distance
            path_fitness += normalize(branchDistance[b])

    # actualizo si el path_fitness es menor
    if min_path_fitness>path_fitness:
        min_path_fitness = path_fitness
```

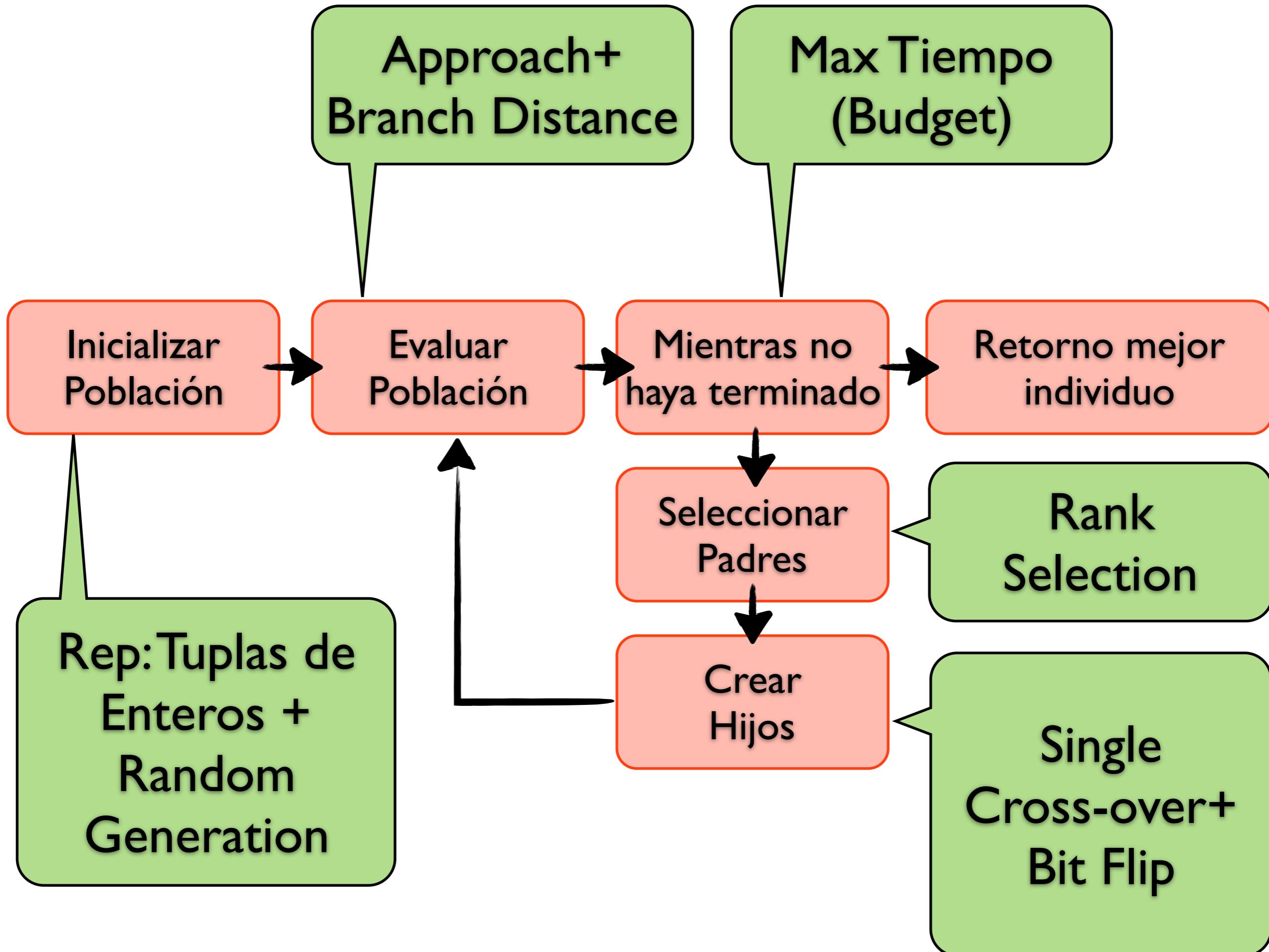
```
# enumerar todos los caminos que van de la raiz al target branch
paths=compute_all_paths_from_root(cdg,targetBranch)

# fitness minima hasta el momento
min_path_fitness = sys.maxint

# computar el fitness de cada camino al target branch
for path in paths:
    path_fitness = 0
    for branch in paths:
        if not branch in branchDistances.keys():
            # si el predicado no fue cubierto, agregar 1
            path_fitness += 1
        else:
            # si el predicado fue cubierto, normalizar branch distance
            path_fitness += normalize(branchDistance[b])

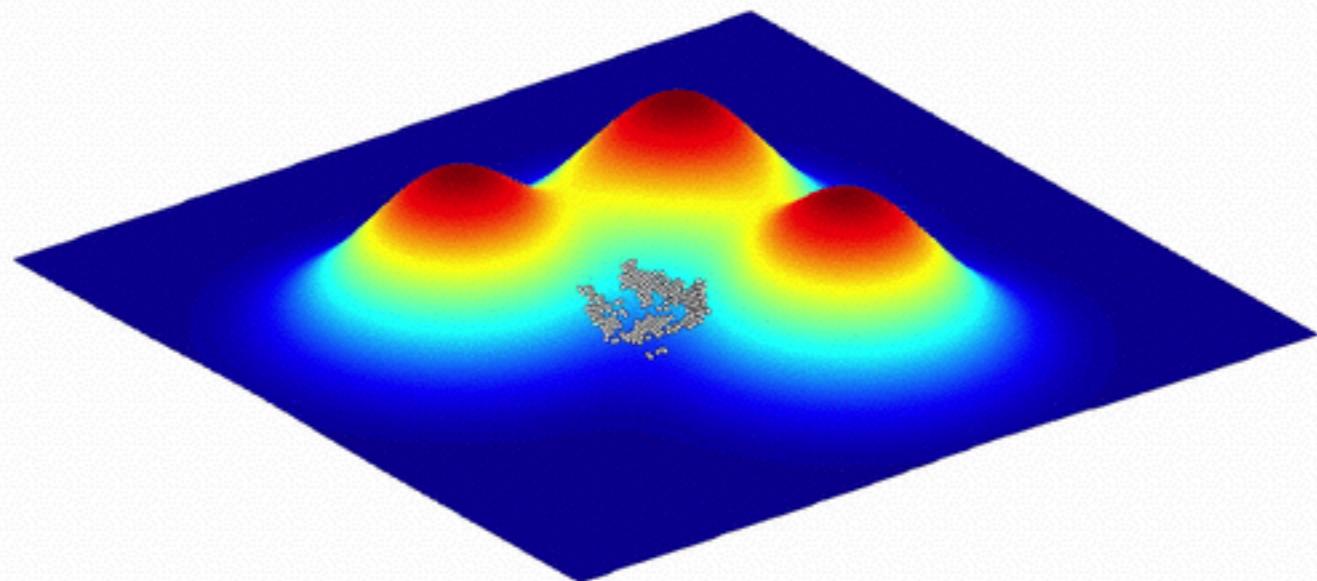
    # actualizo si el path_fitness es menor
    if min_path_fitness>path_fitness:
        min_path_fitness = path_fitness

return min_path_fitness
```



Ejemplo

Static fitness landscape



Population size, $N = 2,304$
Mutation rate, $\mu = 0.05$ per trait

© Randy Olson and Bjørn Østman